

DESIGN ARQUITETURAL DE MICROSSERVIÇOS



Tabela de conteúdos

Introduction	1.1
Microsserviços	1.2
Spring	1.2.1
Construindo uma aplicação com Spring Boot	1.2.2
Entendendo REST	1.2.3
Criando um end-point REST	1.2.4
Recuperando um Recurso	1.2.5
Incluindo um Recurso	1.2.6
Alterando Recursos	1.2.7
Persistindo em um Banco de Dados	1.2.8
Comunicação entre Microsserviços	1.2.9
RabbitMQ	1.3
Principais Conceitos	1.3.1
Cadastro de Livro Assíncrono	1.3.2
Uso em alta disponibilidade (cluster)	1.3.3
Características da Arquitetura Orientada a Microsserviço	1.4
Config Server	1.4.1
Service Discovery com Consul	1.4.2
Service Boot Actuator	1.4.3
Cache com Redis	1.4.4
Gateway	1.4.5
Admin	1.4.6
Design Arquitetural de Microsserviços	1.5
Análise de domínio	1.5.1
Identificando limites de microsserviço	1.5.2
Considerações de dados	1.5.3
Comunicação entre serviços	1.5.4
Design de API	1.5.5
Ingestão de dados e fluxo de trabalho	1.5.6
Gateways de API	1.5.7
Log e monitoramento	1.5.8
Integração contínua	1.5.9
Implantação e Monitoramento	1.6
Monitoramento	1.6.1
Tolerância a Falhas	1.6.2
Testes	1.6.3

Introdução

Seja bem-vindo ao Curso de Design Arquitetural de Microsserviços.

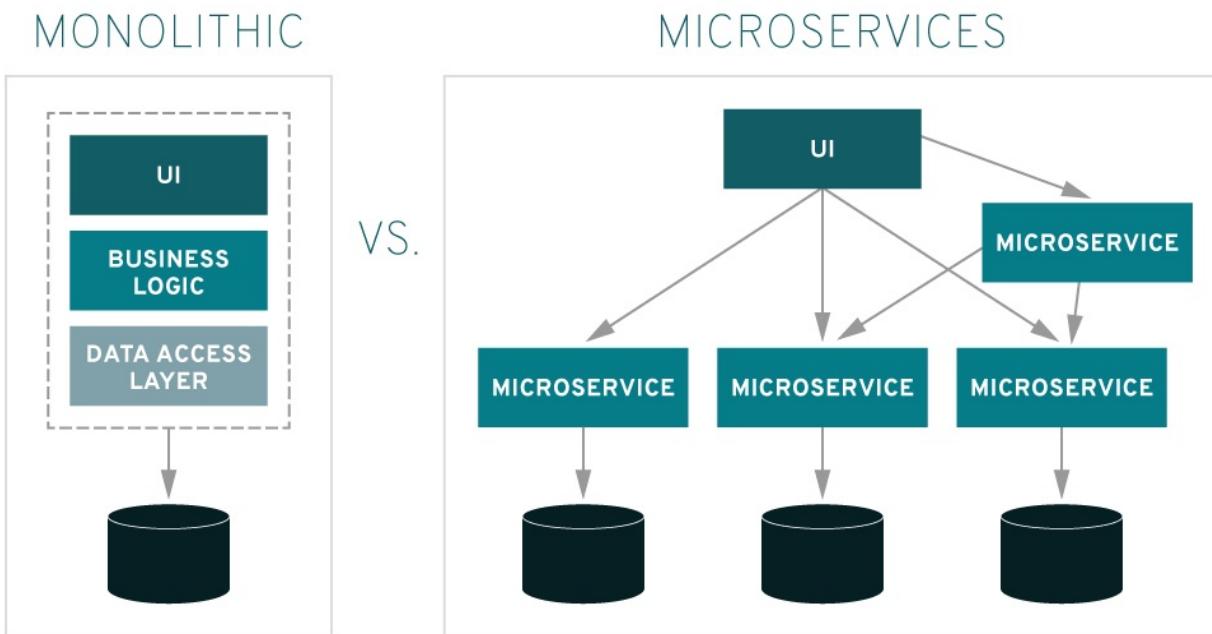
Instrutor: Tiago Lage Payne de Pádua Contatos: tiagolpadua@gmail.com / twitter.com/tiagolpadua / 61 99275-7212

Objetivo Geral: Capacitar os colaboradores do DETEC para o desenvolvimento, implantação e monitoramento dos microsserviços para o projeto das agências modulares e comunitárias utilizando todo o conjunto de ferramentas que envolvem esse novo modelo arquitetural que virão a ratificar a nova arquitetura corporativa.

Microsserviços

Definição de Microsserviços

Microsserviços são uma abordagem arquitetural para a criação de aplicações. O que diferencia a arquitetura de microsserviços das abordagens monolíticas tradicionais é como ela decompõe a aplicação por funções básicas. Cada função é denominada um serviço e pode ser criada e implantada de maneira independente. Isso significa que cada serviço individual pode funcionar ou falhar sem comprometer os demais.

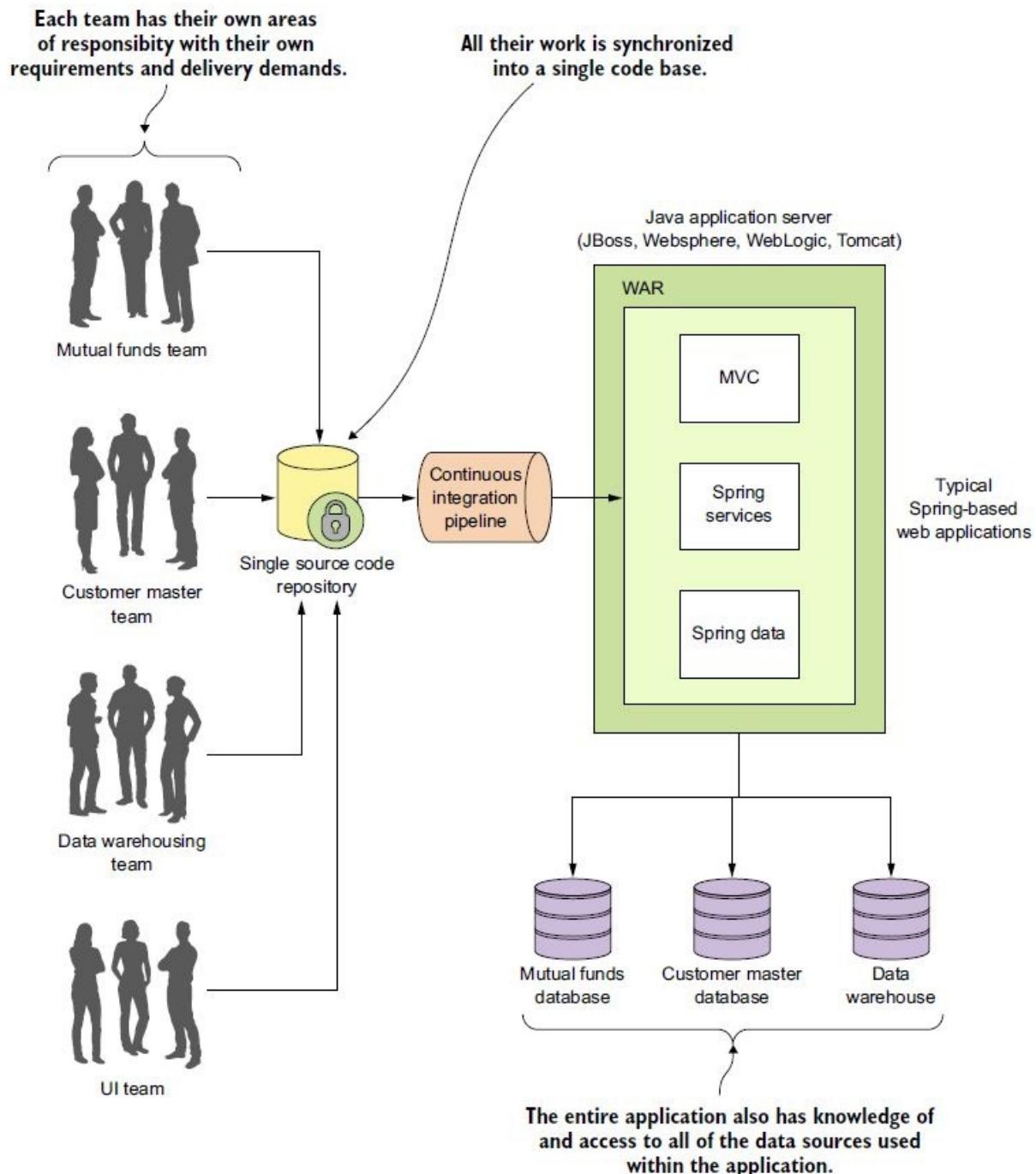


Pense na última vez em que você acessou um site de vendas no varejo. Provavelmente, você usou a barra de pesquisa do site para procurar produtos. Essa pesquisa representa um serviço. Talvez você também tenha visto recomendações de produtos relacionados, extraídas de um banco de dados das preferências dos compradores. Isso também é um serviço. Você adicionou algum item ao carrinho de compras? Isso mesmo, esse é mais um serviço.

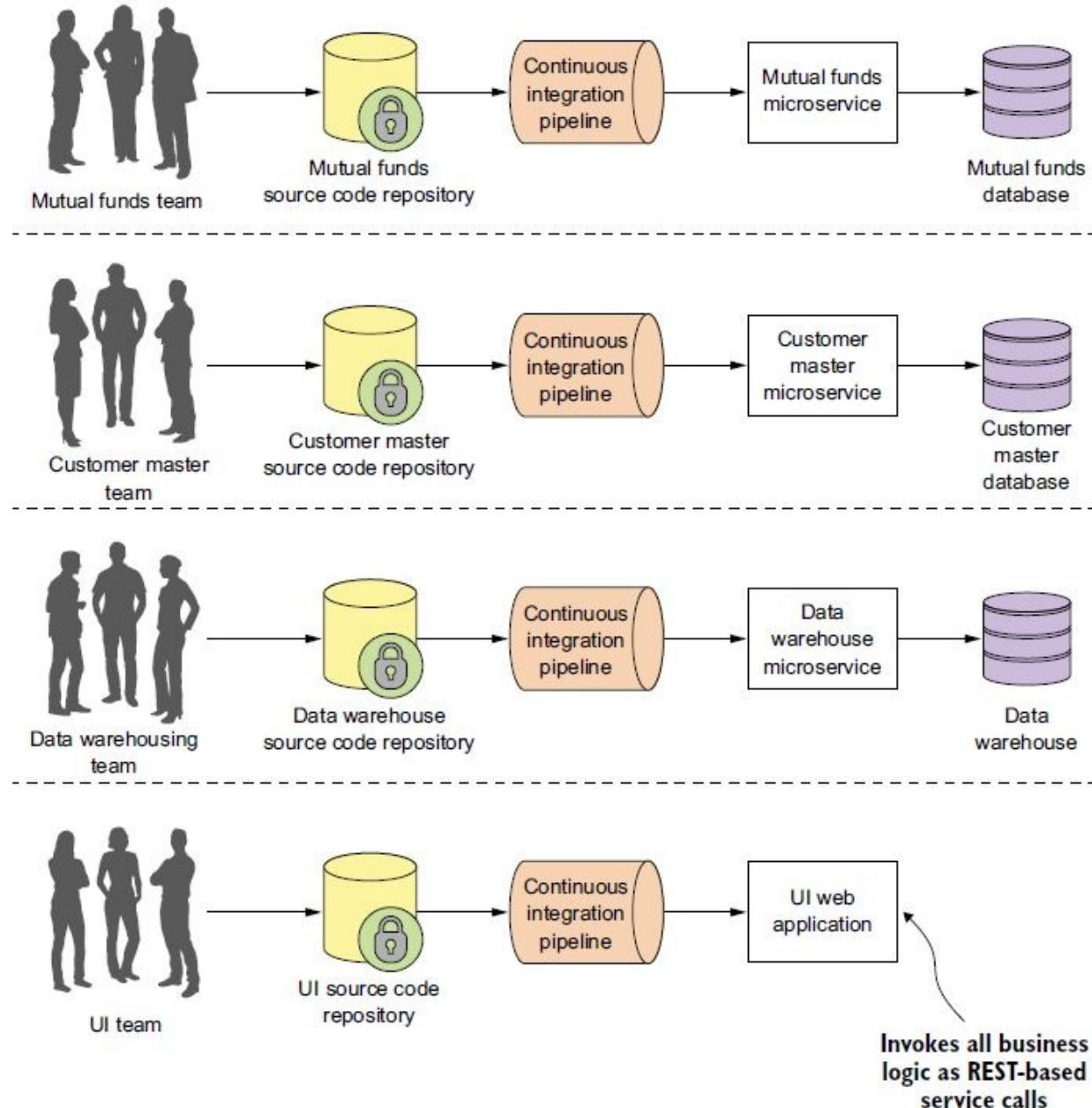
Portanto, um microsserviço é uma função essencial de uma aplicação e é executado independentemente dos outros serviços. No entanto, a arquitetura de microsserviços é mais complexa do que o mero acoplamento flexível das funções essenciais de uma aplicação. Trata-se da restruturação das equipes de desenvolvimento e da comunicação entre serviços de modo a preparar a aplicação para falhas inevitáveis, escalabilidade futura e integração de recursos novos.

Como isso é possível? Com a adaptação dos fundamentos da arquitetura orientada a serviço (SOA) para a implantação de microsserviços.

Uma abordagem monolítica:



Uma abordagem de microsserviços:



Quais são os benefícios da arquitetura de microsserviços?

Com os microsserviços, suas equipes e tarefas rotineiras podem ser tornar mais eficientes por meio do desenvolvimento distribuído. Além disso, é possível desenvolver vários microsserviços ao mesmo tempo. Isso significa que você pode ter mais desenvolvedores trabalhando simultaneamente na mesma aplicação, o que resulta na redução do tempo gasto com desenvolvimento.

- **Lançamento no mercado com mais rapidez:** Como os ciclos de desenvolvimento são reduzidos, a arquitetura de microsserviços é compatível com implantações e atualizações mais ágeis.
- **Altamente escalável:** À medida que a demanda por determinados serviços aumenta, você pode fazer implantações em vários servidores e infraestruturas para atender às suas necessidades.
- **Resiliente:** Os serviços independentes, se construídos corretamente, não afetam uns aos outros. Isso significa que, se um elemento falhar, o restante da aplicação permanece em funcionamento, diferentemente do modelo monolítico.

- **Fácil de implantar:** Como as aplicações baseadas em microsserviços são mais modulares e menores do que as aplicações monolíticas tradicionais, as preocupações resultantes dessas implantações são invalidadas. Isso requer uma coordenação maior, mas as recompensas podem ser extraordinárias.
- **Acessível:** Como a aplicação maior é decomposta em partes menores, os desenvolvedores têm mais facilidade para entender, atualizar e aprimorar essas partes. Isso resulta em ciclos de desenvolvimento mais rápidos, principalmente quando também são empregadas as tecnologias de desenvolvimento ágil.
- **Mais aberta:** Devido ao uso de APIs poliglotas, os desenvolvedores têm liberdade para escolher a melhor linguagem e tecnologia para a função necessária.

O What, Why, e How de uma arquitetura de microsserviços

Há muitos anos que estamos construindo sistemas e melhorando-os. Diversas tecnologias, padrões de arquitetura e melhores práticas surgiram ao longo desses anos. Os microsserviços são um desses padrões de arquitetura que surgiram do mundo do design orientado a domínio, entrega contínua, automação de plataforma e infraestrutura, sistemas escalonáveis, programação poliglota e persistência.

O que é uma arquitetura de microsserviços em poucas palavras? (What)

Robert C. Martin cunhou o termo princípio da responsabilidade única que afirma "reunir as coisas que mudam pela mesma razão e separar as coisas que mudam por diferentes razões".

Uma arquitetura de microsserviços usa essa mesma abordagem e a estende aos serviços fricamente acoplados que podem ser desenvolvidos, implantados e mantidos de forma independente. Cada um desses serviços é responsável por tarefas discretas e pode se comunicar com outros serviços por meio de APIs simples para resolver um problema de negócios complexo e maior.

Principais benefícios de uma arquitetura de microsserviços (Why)

Como os serviços constituintes são pequenos, eles podem ser construídos por uma ou mais equipes pequenas desde o início, separadas por limites de serviço, o que facilita a expansão do esforço de desenvolvimento, se necessário.

Uma vez desenvolvidos, esses serviços também podem ser implantados independentemente uns dos outros e, portanto, é fácil identificar serviços ativos e escaloná-los independentemente do aplicativo inteiro. Os microsserviços também oferecem isolamento aprimorado de falhas, em que, no caso de um erro em um serviço, todo o aplicativo não deixa necessariamente de funcionar. Quando o erro é corrigido, a correção pode ser efetuada apenas para o respectivo serviço, em vez de reimplementar um aplicativo inteiro.

Outra vantagem que uma arquitetura de microsserviços traz é facilitar a escolha da pilha de tecnologia (linguagens de programação, bancos de dados, etc.) que é mais adequada para a funcionalidade necessária (serviço) em vez de ser mais padronizada, solução "bala de prata".

Como eu começo com uma arquitetura de microsserviços? (How)

Espero que agora você esteja convencido de que uma arquitetura de microsserviços pode oferecer algumas vantagens exclusivas em relação às arquiteturas tradicionais e você começou a pensar nesse tipo de abordagem para o seu próximo projeto.

Apróxima pergunta que vem à mente é "Como eu começo?" - e - "Existe um conjunto padrão de princípios que eu possa seguir para me ajudar a construir uma arquitetura de microsserviços de uma maneira melhor?"

Bem, receio que a resposta seja "não".

Embora isso possa não parecer tão promissor, há, no entanto, alguns temas comuns que muitas organizações que adotaram arquiteturas de microsserviços seguiram e com as quais acabaram encontrando sucesso. Vou discutir alguns desses temas comuns abaixo.

1 - Decompor

Uma das maneiras de tornar nosso trabalho mais fácil pode ser realizar a definição de serviços correspondentes aos recursos de negócios. Uma capacidade de negócios é algo que uma empresa faz para fornecer valor a seus usuários finais.

Identificar os recursos de negócios e os serviços correspondentes requer um alto nível de compreensão dos negócios. Por exemplo, os recursos de negócios para um aplicativo de compras on-line podem incluir o seguinte:

- Gerenciamento do catálogo de produtos
- Gerenciamento de estoque
- Gerenciamento de pedidos
- Gerenciamento de entrega
- Gerenciamento de usuários
- Recomendações de produtos
- Gerenciamento de avaliações de Produto

Depois que os recursos de negócios forem identificados, os serviços necessários podem ser construídos correspondendo a cada um desses recursos de negócios identificados.

Cada serviço pode pertencer a uma equipe diferente que se torna especialista nesse domínio específico e a um especialista nas tecnologias mais adequadas para esses serviços específicos. Isso geralmente leva a limites de API mais estáveis e equipes mais estáveis.

2 - Construindo e Implantando

Depois de decidir sobre os limites de serviço desses pequenos serviços, eles podem ser desenvolvidos por uma ou mais equipes pequenas usando as tecnologias mais adequadas para cada finalidade. Por exemplo, você pode optar por criar um Serviço do Usuário em Java com um banco de dados MySQL e um Serviço de Recomendação do Produto com o Scala / Spark.

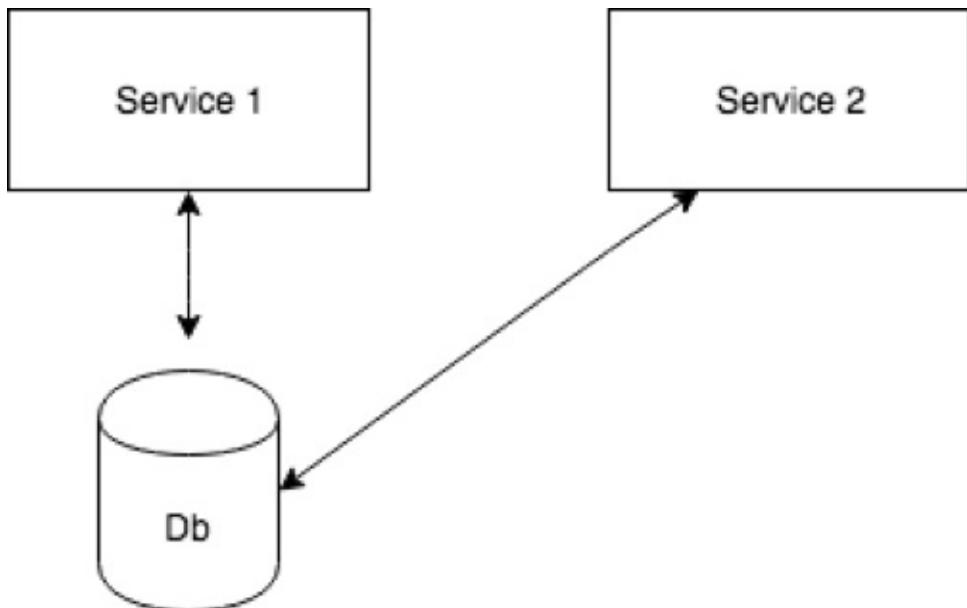
Uma vez desenvolvidos, os pipelines CI/CD podem ser configurados com qualquer um dos servidores de CI disponíveis (Jenkins, TeamCity, Go, etc.) para executar os casos de teste automatizados e implantar esses serviços independentemente em diferentes ambientes (integração, controle de qualidade, preparação e produção).

3 - Projetar os serviços individuais com cuidado

Ao projetar os serviços, defina-os cuidadosamente e pense no que será exposto, quais protocolos serão usados para interagir com o serviço, etc.

É muito importante ocultar qualquer complexidade e detalhes de implementação do serviço e apenas expor o que é necessário para os clientes do serviço. Se detalhes desnecessários forem expostos, torna-se muito difícil mudar o serviço mais tarde, pois haverá muito trabalho para determinar quem está confiando nas várias partes do serviço. Além disso, muita flexibilidade é perdida em poder implantar o serviço de forma independente.

O diagrama abaixo mostra um dos erros comuns no projeto de microsserviços:

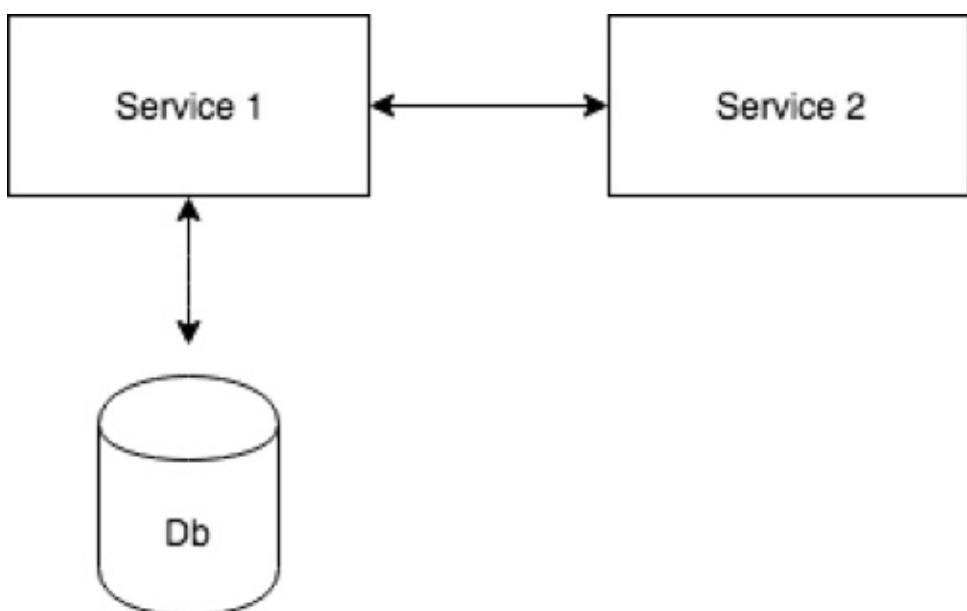


Como você pode ver no diagrama, aqui estamos pegando um serviço (Serviço 1) e armazenando todas as informações necessárias para o serviço em um banco de dados. Quando outro serviço (Serviço 2) é criado, o qual precisa dos mesmos dados, acessamos esses dados diretamente do banco de dados.

Essa abordagem pode parecer razoável e lógica em certos casos - talvez seja fácil acessar dados em um banco de dados SQL ou gravar dados em um banco de dados SQL ou talvez as APIs necessárias ao Serviço 2 não estejam prontamente disponíveis.

Assim que essa abordagem é adotada, o controle é imediatamente perdido para determinar o que está oculto e o que não está. Posteriormente, se o esquema precisar ser alterado, a flexibilidade para fazer essa alteração será perdida, já que você não saberá quem está usando o banco de dados e se a alteração interromperá o Serviço 2 ou não.

Uma abordagem alternativa, e gostaria de apresentar o caminho certo para resolver isso, está abaixo:



O Serviço 2 deve acessar o Serviço 1 e evitar ir diretamente ao banco de dados, preservando assim a máxima flexibilidade para várias alterações de esquema que possam ser necessárias. A preocupação com outras partes do sistema é eliminada desde que você tenha certeza de que os testes para as APIs expostas serão aprovados.

Como mencionado, escolha cuidadosamente os protocolos para comunicação entre serviços. Por exemplo, se o Java RMI for escolhido, o usuário da API não apenas ficará restrito ao uso de uma linguagem baseada em JVM, mas, além disso, o protocolo é bastante frágil porque é difícil manter a compatibilidade com a API.

4 - Descentralizar

Existem organizações que obtiveram sucesso com microsserviços e seguiram um modelo em que as equipes que constroem os serviços cuidam de tudo relacionado a esse serviço. São eles que desenvolvem, implantam, mantêm e suportam. Não há equipes separadas de suporte ou manutenção.

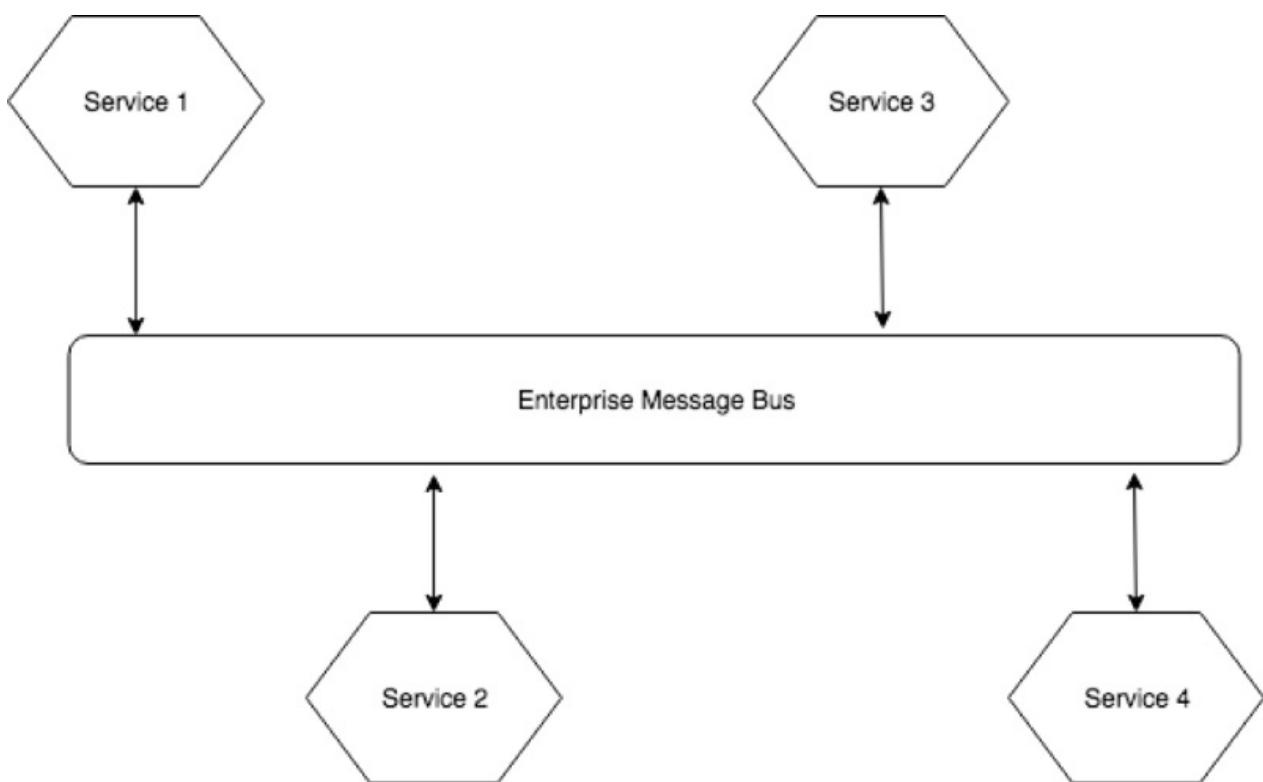
Outra maneira de conseguir o mesmo é ter um modelo interno de código aberto. Ao adotar essa abordagem, o desenvolvedor que precisa de mudanças em um serviço pode verificar o código, trabalhar em um recurso e enviar um *pool-request* em vez de esperar que o proprietário do serviço receba e trabalhe nas alterações necessárias.

Para que este modelo funcione adequadamente, é necessária a documentação técnica adequada, juntamente com instruções de configuração e orientação para cada serviço, para facilitar a coleta e o trabalho do usuário.

Outra vantagem oculta dessa abordagem é que ela mantém os desenvolvedores focados em escrever códigos de alta qualidade, pois eles sabem que os outros estarão olhando para ela.

Existem também alguns padrões arquiteturais que podem ajudar a descentralizar as coisas. Por exemplo, você pode ter uma arquitetura em que a coleção de serviços estejam se comunicando por meio de um barramento de mensagens central.

Esse barramento manipula o roteamento de mensagens de diferentes serviços. Agentes de mensagens como o RabbitMQ são um bom exemplo.



O que tende a acontecer com o tempo é que as pessoas começam a colocar mais e mais lógica nesse barramento central e começa a saber mais e mais sobre o seu domínio. À medida que se torna mais inteligente, isso pode realmente se tornar um problema, pois torna-se difícil fazer mudanças que exigem coordenação entre equipes dedicadas separadas.

Meu conselho geral para esses tipos de arquiteturas seria mantê-los relativamente "estúpidos" e deixá-los apenas lidar com o roteamento. Arquiteturas baseadas em eventos parecem funcionar bem nesses cenários.

5 - Implantação (deploy)

É importante escrever Contratos Direcionados ao Consumidor¹ para qualquer API que esteja sendo dependente. Isso garante que novas alterações nessa API não interrompam sua API.

¹. <https://reflectoring.io/7-reasons-for-consumer-driven-contracts> , <https://spring.io/guides/gs/contract-rest> ↵

Nos contratos orientados ao consumidor, cada API do consumidor captura suas expectativas do provedor em um contrato separado. Todos esses contratos são compartilhados com o fornecedor para que eles obtenham informações sobre as obrigações que devem cumprir para cada cliente individual.

Os contratos orientados pelo consumidor devem passar por testes antes de serem implantados e antes que qualquer alteração seja feita na API. Também ajuda o provedor a saber quais serviços estão dependendo e como outros serviços dependem dele.

Quando se trata de implantar microsserviços independentes, existem dois modelos comuns de implantação.

Vários microsserviços por sistema operacional

Primeiro, **vários microsserviços por sistema operacional** podem ser implantados. Com esse modelo, o tempo é salvo na automação de determinados itens, por exemplo, o host de cada serviço não precisa ser provisionado.

Adesvantagem dessa abordagem é que ela limita a capacidade de alterar e dimensionar os serviços de forma independente. Também cria dificuldades no gerenciamento de dependências. Por exemplo, todos os serviços no mesmo host terão que usar a mesma versão do Java se forem escritos em Java. Além disso, esses serviços independentes podem produzir efeitos colaterais indesejados para outros serviços em execução, o que pode ser um problema muito difícil de reproduzir e resolver.

Um microsserviço por sistema operacional

Devido ao desafio acima, o segundo modelo, no qual um microsserviço por sistema operacional é implantado, é a escolha preferida.

Com esse modelo, o serviço é mais isolado e, portanto, é mais fácil gerenciar dependências e dimensionar serviços de forma independente. Mas você pode se perguntar "não é caro"? Bem, na verdade não.

Asolução tradicional para resolver esse problema é usar o Hypervisors, pelo qual várias máquinas virtuais são provisionadas no mesmo host. Essa abordagem de solução pode ser custosa, já que o próprio processo do hipervisor está consumindo alguns recursos e, é claro, quanto mais VMs forem provisionadas, mais recursos serão consumidos. E é aí que o modelo de contêiner recebe boa tração e é o preferido. O Docker é uma implementação desse modelo.

Fazendo alterações nas APIs de microsserviço existentes enquanto em produção

Outro problema comum tipicamente enfrentado com um modelo de microsserviços é determinar como fazer alterações nas APIs de microsserviço existentes quando outras pessoas a estão usando em produção. Fazer alterações na API de microsserviço pode quebrar o microsserviço que é dependente dele.

Existem diferentes maneiras de resolver esse problema.

Primeiro, versione a sua API e quando as alterações são necessárias para a API, implante a nova versão da API enquanto mantém a primeira versão atualizada. Os serviços dependentes podem ser atualizados em seu próprio ritmo para usar a versão mais recente. Depois que todos os serviços dependentes forem migrados para usar a nova versão do microsserviço alterado, ele poderá ser desativado.

Um problema com esta abordagem é que se torna difícil manter as várias versões. Quaisquer novas alterações ou correções de erros devem ser feitas em ambas as versões.

Por essa razão, uma abordagem alternativa pode ser considerada na qual outro *end-point* é implementado no mesmo serviço quando as mudanças são necessárias. Depois que o novo *end-point* estiver sendo totalmente utilizado por todos os serviços, o *end-point* antigo poderá ser excluído.

Avantagem dessa abordagem é que é mais fácil manter o serviço, pois sempre haverá apenas uma versão da API em execução.

6 - Padrões

Quando várias equipes cuidam de diferentes serviços de forma independente, é melhor introduzir alguns padrões e melhores práticas - tratamento de erros, por exemplo. Como seria de se esperar, os padrões e as melhores práticas não são fornecidos, cada serviço provavelmente lidaria com erros de maneira diferente e, sem dúvida, uma quantidade significativa de código desnecessário seria gravada.

Criar padrões como o Guia de Estilo da API do PayPal é sempre útil em longo prazo. Também é importante informar aos outros o que uma API faz e a documentação da API sempre deve ser feita ao criá-la. Existem ferramentas como o Swagger, que são muito úteis para auxiliar no desenvolvimento em todo o ciclo de vida da API, desde o design e a documentação até o teste e a implantação. A capacidade de criar metadados para sua API e permitir que os usuários aproveitem, permite que eles saibam mais e usem-na com mais eficiência.

Dependências de serviços

Em uma arquitetura de microsserviços, ao longo do tempo, cada serviço começa dependendo de mais e mais serviços. Isso pode introduzir mais problemas à medida que os serviços aumentam, por exemplo, o número de instâncias de serviço e seus locais (host + porta) podem mudar dinamicamente. Além disso, os protocolos e o formato em que os dados são compartilhados podem variar de serviço para serviço.

É aqui que os Gateways de API e a Descoberta de serviços se tornam muito úteis. A implementação de um gateway de API se torna um ponto de entrada único para todos os clientes, e os gateways de API podem expor uma API diferente para cada cliente.

O gateway de API também pode implementar segurança, como verificar se o cliente está autorizado a executar a solicitação. Existem algumas ferramentas como o Zookeeper, que podem ser usadas para a Descoberta de Serviço (embora não tenha sido construída para esse propósito). Existem muitas ferramentas modernas, como o etcd e o Cônslul de Hashicorp, que tratam a Descoberta de Serviços como um cidadão de primeira classe e definitivamente vale a pena olhar para esse problema.

7 - Falhas

Um ponto importante a entender é que os microsserviços não são resilientes por padrão. Haverá falhas nos serviços. Falhas podem acontecer devido a falhas nos serviços dependentes. Além disso, as falhas podem surgir por vários motivos, como erros no código, tempos limite de rede, etc.

O que é crítico em uma arquitetura de microsserviços é garantir que todo o sistema não seja afetado ou diminua quando houver erros em uma parte individual do sistema.

Existem padrões como Bulkhead e Circuits Breaker, que podem ajudar você a obter uma melhor resiliência.

Bulkhead (Anteparo)

O padrão Bulkhead isola os elementos de um aplicativo em pools, de modo que, se um deles falhar, os outros continuarão a funcionar. O padrão é chamado Bulkhead porque se assemelha às partições seccionadas do casco de um navio. Se o casco de um navio estiver comprometido, apenas a seção danificada se enche de água, o que impede que o navio afunde.

Circuit Breaker (Disjuntor)

O padrão de disjuntor envolve uma chamada de função protegida em um objeto de disjuntor, que monitora falhas. Uma vez que uma falha atravessa o limite, o disjuntor desarma e todas as outras chamadas para o disjuntor retornam com um erro, sem que a chamada protegida seja feita para um determinado tempo limite configurado.

Depois que o tempo limite expira, algumas chamadas são permitidas pelo disjuntor para passar e, se elas tiverem êxito, o disjuntor retornará um estado normal. Durante o período em que o disjuntor falhou, os usuários podem ser notificados de que uma determinada parte do sistema está quebrada e o restante do sistema ainda pode ser usado.

Esteja ciente de que fornecer o nível necessário de resiliência para um aplicativo pode ser um desafio multidimensional.

8 - Monitoramento e Logging

Os microsserviços são distribuídos por natureza e o monitoramento e o registro de serviços individuais podem ser um desafio. É difícil passar e correlacionar logs de cada instância de serviço e descobrir erros individuais. Assim como nas aplicações monolíticas, não há um único local para monitorar microsserviços.

Agregação de Log

Para resolver esses problemas, uma abordagem preferencial é aproveitar um serviço de registro centralizado que agregue logs de cada instância de serviço. Os usuários podem pesquisar por esses registros a partir de um ponto centralizado e configurar alertas quando certas mensagens aparecerem.

Ferramentas padrão estão disponíveis e amplamente utilizadas por várias empresas. O ELK Stack é a solução usada com mais freqüência, onde o daemon de registro, o Logstash , coleta e agrupa logs que podem ser pesquisados por meio de um painel do Kibana indexado pelo Elasticsearch.

Agregação de Estatísticas

Semelhante à agregação de logs, a agregação de estatísticas, como CPU e uso de memória, também pode ser aproveitada e armazenada centralmente. Ferramentas como o Graphite fazem um bom trabalho ao empurrar para um repositório central e armazená-lo de maneira eficiente.

Quando um dos serviços downstream é incapaz de lidar com solicitações, deve haver uma maneira de acionar um alerta, e é aí que a implementação de APIs de verificação de integridade em cada serviço se torna importante - elas retornam informações sobre a integridade do sistema.

Um cliente de verificação de integridade, que poderia ser um serviço de monitoramento ou um balanceador de carga, chama o nó de extremidade para verificar periodicamente a integridade da instância do serviço em um determinado intervalo de tempo. Mesmo que todos os serviços downstream sejam saudáveis, ainda pode haver um problema de comunicação downstream entre os serviços. Ferramentas como o projeto Hystrix da Netflix permitem a capacidade de identificar esses tipos de problemas.

Uma última coisa

Agora que abordamos o que é uma arquitetura de microsserviços, e você deseja implantar uma arquitetura de microsserviços e está pensando sobre como começar, gostaria de oferecer um conselho final:

Comece pequeno

Quando você estiver apenas começando a desenvolver microsserviços, comece modestamente com apenas um ou dois serviços, aprenda com eles e, com o tempo e a experiência, adicione mais.

Desenvolvimento de Softwares

Análise Estruturada

Análise estruturada é uma atividade de construção de modelos. Utiliza uma notação que é particular ao método de análise estruturada para com a finalidade de retratar o fluxo e o conteúdo das informações utilizadas pelo sistema, dividir o sistema em partições funcionais e comportamentais e descrever a essência daquilo que será construído.

Análise estruturada é muito difícil de ser modelada, rastrear e gerenciar mudanças manualmente. Por essas e outras razões, as ferramentas DFD tornaram-se uma abordagem preferida. Para uma pré-elaboração de um projeto de desenvolvimento de software. Análise estruturada também contém gráficos que possibilitam ao analista criar

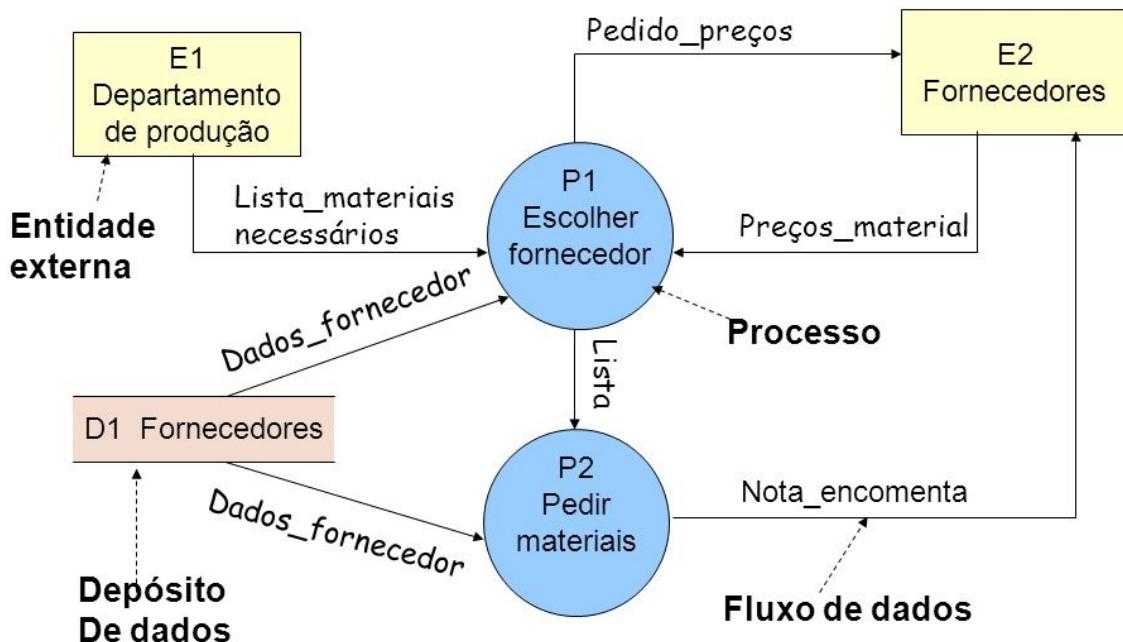
modelos de fluxo de informação, com uma heurística para o uso dos símbolos, juntamente com um dicionário de dados, e narrativas de processamentos como o complemento aos modelos defluxo de informação. Um modelo de fluxo pode ser criado para qualquer sistema baseado em computador, independentemente do tamanho e complexidade.

O dicionário de dados é uma listagem organizada de todos os elementos de dados que são pertinentes ao sistema, com definições precisas e rigorosas, de forma que tanto o usuário como os analistas de sistemas tenham uma compreensão comum das tarefas, das saídas, dos componentes dos depósitos e (até mesmo) dos cálculos intermediários. Atualmente, isto é inserido quase sempre como parte de uma ferramenta de projeto e análise estruturada.

Principais Ferramentas

- Diagrama de Fluxo de Dados (DFD)
- Dicionário de dados (DD)
- Linguagem estruturada
- Tabelas de Decisão
- Diagrama Entidade-Relacionamento (DER)
- Diagrama de Transição de Estados (DTE)
- Diagrama de Fluxo de Dados - DFD

Análise Estruturada - DFD



- Dicionário de Dados - DD

TABELA: Tb_Aluno		Cadastro de alunos				
Informações pertinentes ao aluno						
CAMPO LÓGICO	CAMPO FÍSICO	TIPO	PK	FK (Tabela/Campo)	RESTRIÇÕES	OBSERVAÇÕES
Código	Alu_codigo	SMALLINT	PK		NÃO NULO E MAIOR QUE ZERO	Campo auto-incremento
Nome	Alu_nome	VARCHAR(100)			NÃO NULO	Informar se usuário incluir somente uma palavra.
Data de Nascimento	ento	DATE			Data mínima < HOJE	
Código da turma	Tur_codigo	SMALLINT		Tb_Turma/Tur_codigo	ZERO	
Sexo	Alu_sexo	CHAR(1)			Somente "M" ou "F"	M = Masculino / F = Feminino
Nome do pai	Alu_pai	VARCHAR(100)				
Nome da mãe	Alu_mae	VARCHAR(100)				
Nota Media	Not_codigo	SMALLINT		Tb_Acompanhamento Aluno/aco_codigo		
Registro do Aluno	Alu_RA	SMALLINT			NÃO NULO E MAIOR QUE ZERO	Campo auto-incremento
Informações complementares	Alu_Complemento	BLOB				
Telefone	Alu_telefone	INTEGER				Telefone para contato
Celular	Alu_celular	INTEGER				Celular para contato
Endereço	Alu_endereco	VARCHAR(150)				Endereço do aluno

Orientada a Objetos

A programação orientada a objetos é diferente da programação estruturada. Na programação orientada a objeto, funções e os dados estão juntos, formando o objeto. Essa abordagem cria uma nova forma de analisar, projetar e desenvolver programas, é uma forma mais abstrata, e genérica, que permite um maior reaproveitamento dos códigos, e facilita a sua manutenção.

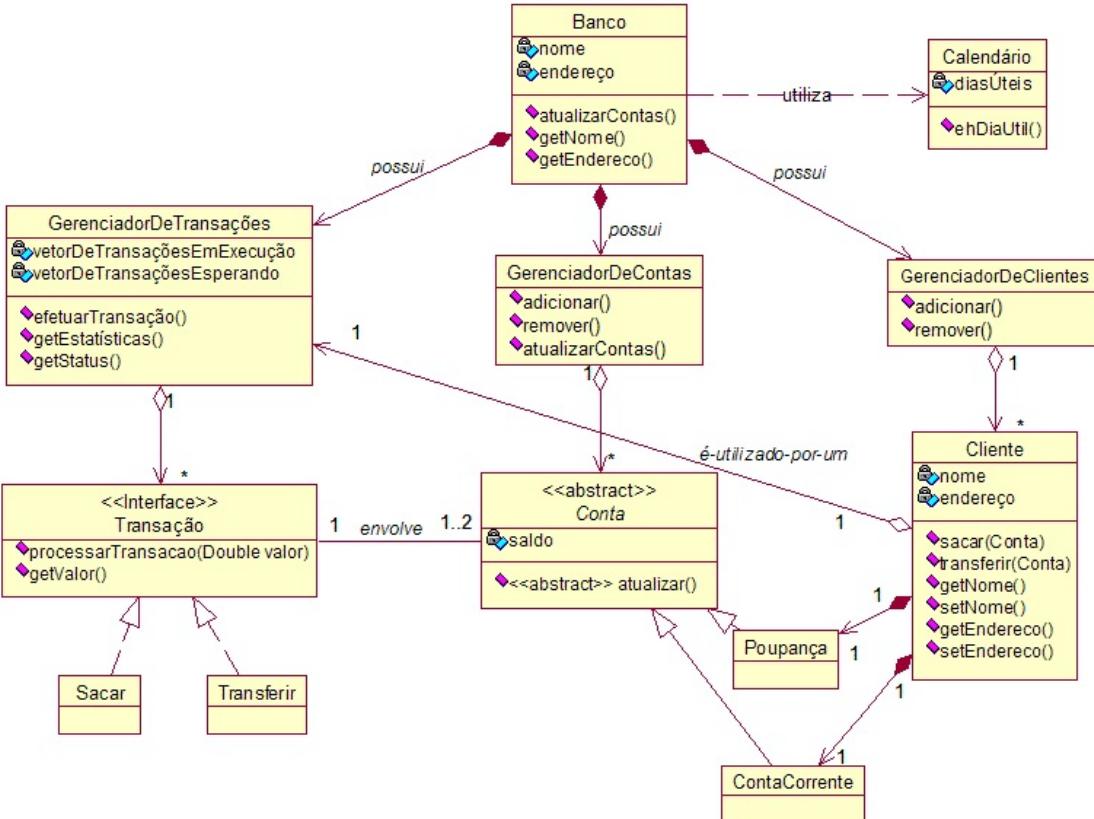
Observe que a modelagem orientada a objeto, não é somente uma nova forma de programar, mas uma nova forma de pensar um problema, de forma abstrata, utilizando conceitos do mundo real e não conceitos computacionais. Na programação orientada a objeto o conceito de objeto deve acompanhar todo o ciclo de desenvolvimento do software.

APOO também inclui uma nova notação e exige do analista/programador o conhecimento dessa notação (diagramas de classe, diagramas de interação, diagramas de sequência, etc.). Atualmente existem centenas de bibliotecas, cuidadosamente desenhadas, para dar suporte aos programadores menos sofisticados. Desta forma os programadores podem montar seus programas unindo as bibliotecas externas com alguns objetos que criaram, ou seja, poderão montar suas aplicações rapidamente, contando com módulos pré-fabricados.

O usuário final verá todos os ícones e janelas da tela como objetos e associará a manipulação desses objetos visuais à manipulação dos objetos reais que eles representam. Enxerga o mundo como objetos com estrutura de dados e comportamentos.

O objetivo é desenvolver uma série de modelos de análise, satisfazendo um conjunto de requisitos definidos pelo cliente. O problema não está em aprender como programar em uma linguagem OO, mas sim em aprender a explorar as vantagens que as linguagens OO oferecem. Portanto, para o sucesso de um projeto OO é necessário seguir boas práticas de engenharia discutidas na literatura e pesquisando padrões já consolidados e aprovados.

- Diagrama de Classes



SOA - Service-oriented architecture

A arquitetura orientada a serviços (SOA) é uma maneira de organizar software.

SOA envolve a implantação de serviços, que são unidades lógicas executadas em rede.

Um serviço tem as seguintes características:

- Lida com um processo de negócios, como calcular uma cotação de seguro ou distribuir e-mail; lida com uma tarefa técnica, como acessar um banco de dados; ou fornece dados de negócios e detalhes técnicos para construir uma interface gráfica;
- Pode acessar outros serviços. Com a tecnologia de *runtime* apropriada, ele pode acessar um programa tradicional e responder a diferentes tipos de solicitantes, como aplicações da Web;
- É relativamente independente de outro software. As alterações feitas a um solicitante exigem poucas ou nenhuma alteração no serviço. Alterações na lógica interna de um serviço exigem poucas ou nenhuma alteração no solicitante. A independência relativa do serviço e outro software é chamada de **baixo acoplamento**;

Um serviço pode lidar com interações dentro de sua empresa e entre sua empresa e seus fornecedores, parceiros e clientes.

ASOA implica um estilo de desenvolvimento focado no negócio como um todo, na modularidade e reutilização. SOA não é apenas para novas aplicações. Você pode migrar aplicações existentes nos seguintes casos:

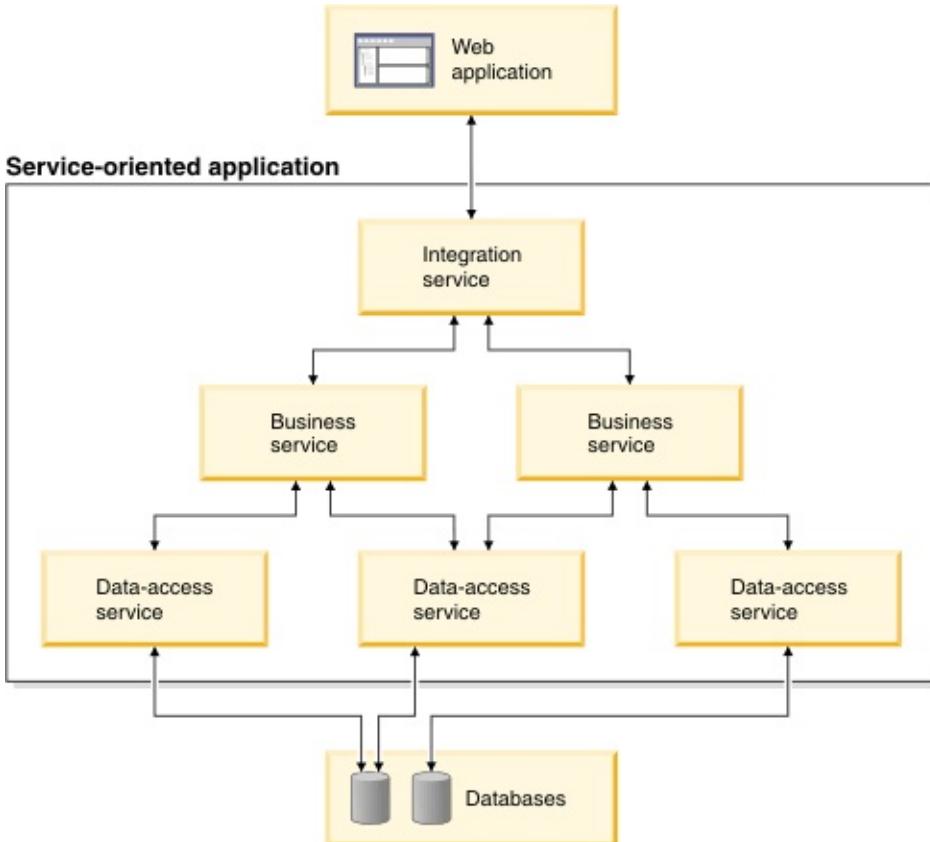
- As aplicações são monolíticas, combinando a lógica da interface com o usuário, o processamento de negócios e o acesso a dados, de forma que a atualização de um tipo de lógica requer que sua empresa teste vários tipos de comportamentos;
- As aplicações são difíceis de entender porque sua lógica é monolítica e foi repetidamente corrigida em vez de reescrita à medida que os requisitos foram alterados. As atualizações levam tempo extra enquanto os desenvolvedores tentam decifrar a lógica; e à medida que a complexidade aumenta, erros adicionais acompanham as atualizações;

- O inventário de aplicações possui lógica duplicada. Pedidos de mudança são perturbadores, exigindo mudanças em vários lugares;

Do ponto de vista de um desenvolvedor, uma mudança para orientação a serviços é uma mudança na ênfase, e muitos aspectos da tarefa de desenvolvimento não são afetados.

Aplicações orientadas a serviços

Uma aplicação orientado a serviços é uma aplicação composta principalmente de serviços, que geralmente estão em uma hierarquia.



O nível mais alto contém um ou mais serviços de integração, cada um dos quais controla um fluxo de atividades, como o processamento de uma solicitação do cliente para cobertura de seguro. Cada serviço de integração invoca um ou mais serviços de negócios.

O segundo nível é composto de serviços que cada um cumpre uma tarefa de negócios de nível relativamente baixo. Por exemplo, um serviço de integração pode invocar uma série de serviços de negócios para verificar os detalhes fornecidos por um agente de apólices de seguros. Se os serviços de negócios retornarem valores considerados como "emita uma apólice", o serviço de integração chamará outro serviço de negócios. O segundo serviço de negócios calcula uma cotação e retorna a cotação para o software, como uma aplicação da Web, que chamou a aplicação orientada a serviços.

O terceiro nível consiste em serviços de acesso a dados, cada um dos quais lida com a tarefa relativamente técnica de ler e gravar em áreas de armazenamento de dados, como bancos de dados e filas de mensagens. Um serviço de acesso a dados é mais frequentemente invocado a partir da camada de negócios, mas o fácil acesso dos serviços permite diferentes usos. Por exemplo, um solicitante desse tipo de aplicação da Web pode acessar um serviço de acesso a dados para atribuir valores iniciais em um formulário.

O ponto central é a flexibilidade. Alguns serviços de integração fornecem diferentes operações para diferentes solicitantes e alguns invocam outros serviços de integração. Além disso, um solicitante pode acessar diferentes tipos de serviços de dentro de uma aplicação orientada a serviços. O solicitante pode acessar um serviço de integração em um ponto e um serviço de negócios em outro.

Web e troca de dados binários

Qual é a característica que define de um serviço da web? Para algumas pessoas, a resposta é que o serviço troca dados em um formato baseado em texto chamado SOAP. Outras pessoas insistem que a característica que define de um serviço web é que o serviço troca dados na World Wide Web por meio do software de comunicações Hypertext Transfer Protocol (HTTP).

Para entender os diferentes estilos de serviços da Web, considere a estrutura de uma mensagem de solicitação HTTP, onde os dados são transmitidos de um navegador para um servidor da Web, e a estrutura da mensagem de resposta HTTP, se houver, que retorna.

Amensagem de solicitação HTTP possui três componentes:

1. Amensagem começa com um método HTTP para identificar o que o destinatário deve fazer com a mensagem;
2. Várias dados que estão no cabeçalho e fornecem informações que não são específicas dos dados da sua empresa. Por exemplo, essas informações podem ser os detalhes sobre o agente do usuário (ou seja, o navegador solicitante). Cada cabeçalho é um par de nome e valor: User-Agent: Mozilla/4.0 ...
3. O corpo da entidade é o dado da solicitação, se houver. Se o método HTTP for GET (uma solicitação de dados), o entity-body estará vazio na maioria dos casos.

Amensagem de resposta HTTP possui três componentes:

1. Amensagem começa com um código de resposta HTTP para indicar se os dados da solicitação foram processados;
2. Várias dados que estão no cabeçalhos, incluindo um, Content-Type , que identifica o formato dos dados no corpo da entidade da resposta. O formato é específico; por exemplo, um é para uma imagem do tipo JPEG. Se o formato for Hypertext Markup Language (HTML), a resposta será uma página da web. Pelo menos três outros tipos de conteúdo fornecem dados de negócios para uso em uma aplicação da Web ou outro solicitante: Extensible Markup Language (XML); SOAP, que é um dialeto XML; e JavaScript Object Notation (JSON), que são dados que são facilmente processados pelo JavaScript;
3. O corpo da entidade são os dados de resposta, se houver;

Afrase *serviço da Web* implica a transmissão de dados em um formato baseado em texto. Por outro lado, um serviço de troca binária troca dados em um formato associado a uma linguagem de computador específica ou a um fornecedor específico.

O uso de serviços de troca binária fornece vários benefícios:

- Permite uma resposta em tempo de execução mais rápida do que é possível com serviços da Web;
- Evita a necessidade de manter arquivos de configuração;
- Evita a necessidade de aprender as tecnologias relacionadas aos serviços da Web tradicionais;

Adesvantagem do uso de serviços de trocas binárias é a acessibilidade reduzida. Um serviço de troca binária está diretamente acessível apenas ao software que transmite dados no formato binário esperado pelo serviço.

Implicações do SOA nos negócios

SOAtem várias implicações importantes para os negócios. Primeiro, quando cada componente é uma unidade relativamente autônoma, sua empresa pode responder a mudanças comerciais ou tecnológicas mais rapidamente e com menos gastos e confusão.

Acapacidade de uma empresa responder rapidamente e bem à mudança é conhecida como agilidade. Um SOAbem elaborado aumenta a agilidade ao longo do tempo.

SOAtambém afeta como as pessoas colaboram. Além dos serviços mais técnicos, um serviço bem escrito tem baixa granularidade. Em um serviço de baixa granularidade, a área de preocupação é ampla o suficiente para que os demandantes do negócio possam entender o propósito do serviço, mesmo que saibam pouco sobre software. Quando uma coleção de serviços de baixa granularidade lida com os procedimentos comerciais de uma empresa, os analistas de negócios e profissionais de software podem compartilhar informações com conhecimento, incluir

usuários em deliberações precoces sobre o objetivo e escopo de cada serviço e entender as implicações de mudar um procedimento de negócios. A facilidade da comunicação humana é um benefício importante da SOA; Este fato sugere que a arquitetura pode se tornar o principal princípio organizador do processamento de negócios.

Serviços bem projetados têm maior probabilidade de serem reutilizáveis. As empresas podem se beneficiar da reutilização de pelo menos duas maneiras: evitando as despesas de desenvolvimento de novos softwares e aumentando a confiabilidade do inventário de software ao longo do tempo. Você pode fazer testes menos extensivos se um serviço existente for colocado em um novo aplicativo, em comparação com os testes necessários para implantar o software que foi escrito do zero.

Você pode usar o SOA para tornar os processos de negócios e os dados mais disponíveis. Por exemplo, imagine agentes em uma empresa de seguros sentados em estações de trabalho e invocando um processo de mainframe para cotar preços de seguro para clientes específicos. Em resposta à pressão da concorrência, a empresa quer permitir que os clientes solicitem cotações na web, que tradicionalmente não tem link direto com um mainframe. O que é necessário ser feito antes que a empresa possa aceitar dados pessoais de clientes em um navegador, processar os dados por meio de software analítico no mainframe e responder ao cliente? A solução inclui o desenvolvimento de novos serviços para lidar com a interação entre o navegador e o software analítico.

Microsserviços

O estilo arquitetural de microsserviços é semelhante ao da arquitetura orientada para serviços (SOA), que já é consagrado no desenvolvimento de aplicações.

Nos primórdios do desenvolvimento de aplicações, até mesmo as alterações mais insignificantes em uma aplicação pronta exigiam uma atualização de toda a aplicação, com um ciclo próprio de garantia da qualidade (QA). Isso, provavelmente, atrasava o trabalho de muitas subequipes. Muitas vezes, essa abordagem é chamada de "monolítica" porque o código-fonte da aplicação toda era incorporado em uma única unidade de implantação, como .war ou .ear. Se a atualização de alguma das partes causasse erros, era necessário desativar a aplicação inteira e corrigir o problema. Embora essa abordagem ainda seja viável para aplicações menores, as empresas em ampla expansão não podem se dar ao luxo de sofrer com tempo de inatividade.

A arquitetura orientada a serviço serve pra resolver essa questão, pois estrutura as aplicações em serviços distintos e reutilizáveis que se comunicam por meio de um Enterprise Service Bus (ESB). Nessa arquitetura, os serviços individuais, cada um deles organizado em torno de um processo de negócios específico, aderem a um protocolo de comunicação, como SOAP, ActiveMQ ou Apache Thrift, para que sejam compartilhados por meio do ESB. Quando reunidos, esse pacote de serviços, integrados por meio de um ESB, formam uma aplicação.

Por um lado, isso permite criar, testar e ajustar os serviços de maneira simultânea, eliminando os ciclos de desenvolvimento monolíticos. No entanto, por outro lado, o ESB representa um ponto único de falha no sistema inteiro. Portanto, todo o esforço empregado para eliminar uma estrutura monolítica, de certo modo, serviu apenas para criar outra: o ESB, que potencialmente pode congestionar toda a organização.

Então, qual é a diferença entre a SOA e a arquitetura de microsserviços?

Os microsserviços podem se comunicar entre si, normalmente sem monitoração de estado. Portanto, as aplicações criadas dessa maneira podem ser mais tolerantes a falhas e depender menos de um único ESB. Além disso, as equipes de desenvolvimento podem escolher as ferramentas que desejarem, pois os microsserviços podem se comunicar por meio de interfaces de programação de aplicações (APIs) independentes de linguagem.

Levando em consideração a história da SOA, os microsserviços não são uma ideia completamente nova. Porém, eles se tornaram mais viáveis graças aos avanços nas tecnologias de containerização. Com os containers Linux, agora é possível executar várias partes de uma aplicação de maneira independente no mesmo hardware e com um controle muito maior sobre os componentes individuais e ciclos de vida.

O maior desafio para a adoção de qualquer arquitetura nova é dar o primeiro passo. Você quer criar aplicações novas ou transformar as antigas? Em ambos os casos, é bom refletir sobre os benefícios e os desafios de criar microsserviços.

Fatores Chaves de Sucesso para Implementação de Microsserviço

Se a sua organização pretende migrar para a arquitetura de microsserviços, tenha em mente que será necessário implementar mudanças não somente nas aplicações, mas também no modo como as pessoas trabalham. As mudanças organizacionais e culturais podem ser, em parte, consideradas como desafios, porque cada equipe terá um ritmo próprio de implantação e será responsável por um serviço exclusivo, com um conjunto próprio de clientes. Talvez essas não sejam preocupações típicas para os desenvolvedores, mas elas serão essenciais para o sucesso da arquitetura de microsserviços.

Além das mudanças na cultura e nos processos, a complexidade e a eficiência são outros dois grandes desafios da arquitetura baseada em microsserviços. John Frizelle, arquiteto de plataformas do Red Hat Mobile, definiu as oito categorias de desafios a seguir em sua palestra no Red Hat Summit de 2017:

1. **Compilação:** é necessário dedicar um tempo à identificação das dependências entre os serviços. É preciso estar ciente de que concluir uma compilação pode gerar muitas outras devido a essas dependências. Também é necessário levar em consideração como os microsserviços afetam os dados.
2. **Testes:** os testes de integração, assim como os testes end-to-end, podem ser mais difíceis e importantes como jamais foram. Saiba que uma falha em uma parte da arquitetura pode provocar outra mais adiante, dependendo de como os serviços foram projetados para embasar uns aos outros.
3. **Controle de versão:** ao atualizar para versões novas, lembre-se de que a compatibilidade com as versões anteriores pode ser rompida. É possível resolver esse problema usando a lógica condicional, mas isso pode se tornar outra complicação rapidamente. Como alternativa, você pode colocar no ar várias versões ativas para clientes diferentes, mas essa solução é mais complexa em termos de manutenção e gerenciamento.
4. **Implantação:** sim, isso também é um desafio, pelo menos na configuração inicial. Para facilitar a implantação, primeiro é necessário investir bastante na automação, pois a complexidade dos microsserviços é demais para a implantação manual. Pense sobre como e em que ordem os serviços serão implementados.
5. **Geração de logs:** com os sistemas distribuídos, é necessário ter logs centralizados para unificar tudo. Caso contrário, é impossível gerenciar o escalonamento.
6. **Monitoramento:** é crítico ter uma visualização centralizada do sistema para identificar as fontes de problemas.
7. **Depuração:** a depuração remota não é uma opção e não funciona com centenas de serviços. Infelizmente, no momento não há uma única resposta para como realizar depurações.
8. **Conectividade:** considere a detecção de serviços, seja de maneira centralizada ou integrada.



Fontes

- <https://www.redhat.com/pt-br/topics/microservices/what-are-microservices>
- <https://medium.com/@maheshwar.ligade/software-architecture-what-why-how-34061f105dc2>
- <http://grupotads2014.blogspot.com/2014/04/analise-estruturada-vs-orientada-objetos.html>
- https://www.ibm.com/support/knowledgecenter/en/SSMQ79_9.5.1/com.ibm.ecl.pg.doc/topics/peg1_serv_overview.html

Spring

Spring Framework

O Spring Framework é uma solução leve e um potencial imediato para a criação de aplicativos corporativos. No entanto, o Spring é modular, permitindo que você use apenas as partes que você precisa, sem ter que trazer o resto. Você pode usar o contêiner IoC, com o Struts no topo, mas também pode usar apenas o código de integração do Hibernate ou a camada de abstração do JDBC. O Spring Framework suporta gerenciamento de transações declarativas, acesso remoto à sua lógica por meio de RMI ou serviços da Web e várias opções para persistir seus dados. Ele oferece uma estrutura MVC completa e permite que você integre o AOP (Aspect Oriented Programming) de forma transparente ao seu software.

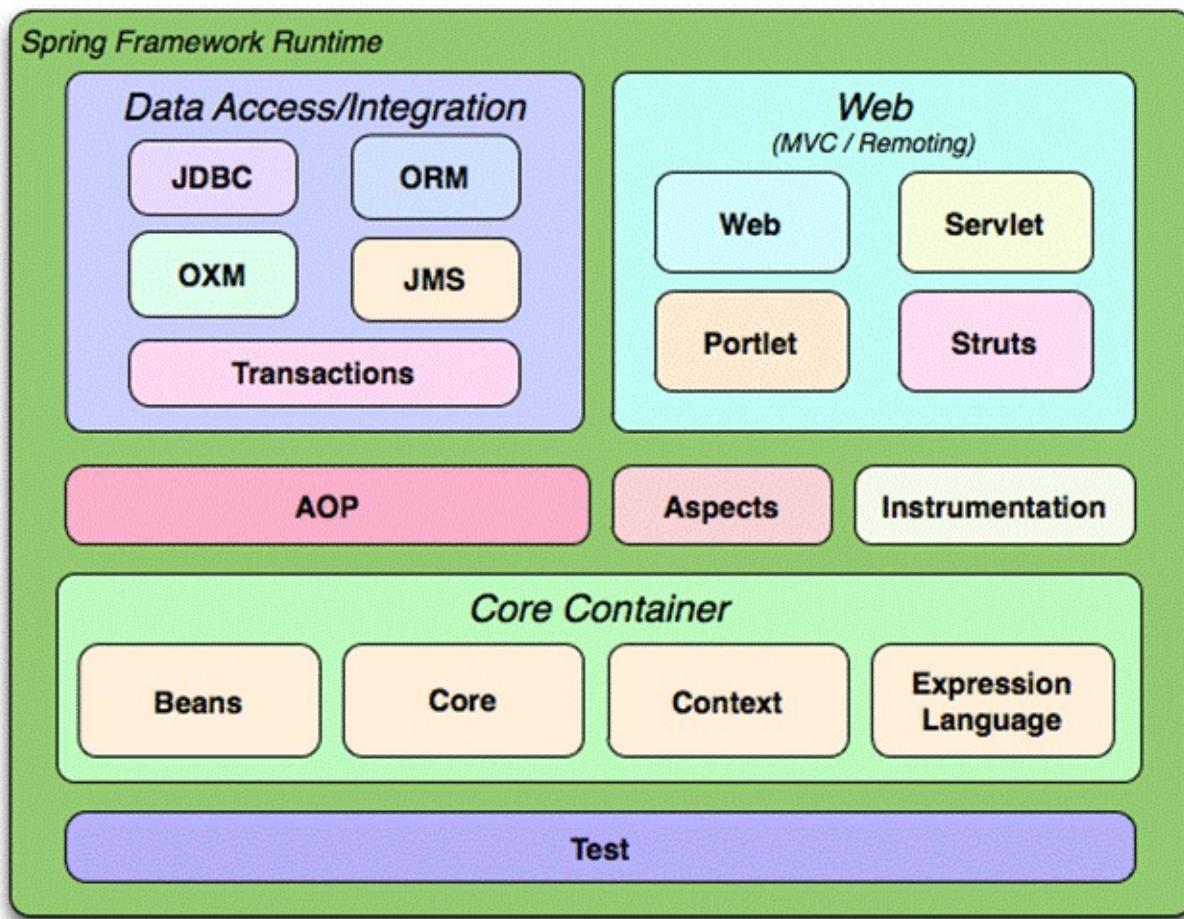
O Spring é projetado para ser não intrusivo, o que significa que o código de lógica do seu domínio geralmente não possui dependências no próprio framework. Na sua camada de integração (como a camada de acesso a dados), existirão algumas dependências na tecnologia de acesso a dados e nas bibliotecas do Spring. No entanto, deve ser fácil isolar essas dependências do restante da sua base de código.

O Spring Framework é uma plataforma Java que fornece suporte abrangente à infraestrutura para o desenvolvimento de aplicativos Java. Spring lida com a infraestrutura para que você possa se concentrar no seu aplicativo.

O Spring permite que você construa aplicativos a partir de "objetos Java simples" (POJOs) e aplique serviços corporativos de maneira não invasiva a POJOs. Esse recurso se aplica ao modelo de programação Java SE e ao Java EE completo e parcial.

Exemplos de como você, como desenvolvedor de aplicativos, pode usar a vantagem da plataforma Spring:

- Faça um método Java executar em uma transação de banco de dados sem ter que lidar com APIs de transação;
- Torne um método Java local um procedimento remoto sem precisar lidar com APIs remotas;
- Torne um método Java local uma operação de gerenciamento sem ter que lidar com APIs JMX;
- Torne um método Java local um manipulador de mensagens sem ter que lidar com as APIs do JMS;



Spring Boot

O Spring Boot facilita a criação de aplicativos baseados em Spring autônomos e de produção que você pode executar. Possui uma visão "opinativa" da plataforma Spring e de bibliotecas de terceiros, para que você possa começar com o mínimo de ruído. A maioria dos aplicativos Spring Boot precisa de uma configuração de Spring muito pequena.

Você pode usar o Spring Boot para criar aplicativos Java que podem ser iniciados usando o `java -jar`. Também fornece uma ferramenta de linha de comando que executa "spring scripts".

Os principais objetivos são:

- Prover uma experiência de introdução radicalmente mais rápida e amplamente acessível para todo o desenvolvimento do Spring;
- Ser primordialmente "opinativo", mas sair caminho rapidamente, quando os requisitos começam a divergir dos padrões;
- Fornecer vários recursos não funcionais que são comuns a grandes classes de projetos (como servidores incorporados, segurança, métricas, verificações de integridade e configuração externalizada);
- Absolutamente nenhuma geração de código e nenhuma necessidade de configuração XML;

Spring Cloud

Desenvolver, implantar e operar aplicativos em nuvem deve ser tão fácil quanto (se não mais fácil que) aplicativos locais. Esse é e deve ser um princípio governante por trás de qualquer plataforma, biblioteca ou ferramenta de nuvem. O Spring Cloud - uma biblioteca de código aberto - facilita o desenvolvimento de aplicativos JVM para a nuvem. Com ele, os aplicativos podem se conectar a serviços e descobrir informações sobre o ambiente de nuvem com facilidade.

Uma das muitas vantagens de executar um aplicativo na nuvem é a fácil disponibilidade de uma variedade de serviços. Em vez de gerenciar hardware, instalação, operação, backups, etc., basta criar e vincular serviços com um clique de um botão ou um comando shell.

Como os aplicativos acessam esses serviços? Por exemplo, se você tiver um banco de dados relacional vinculado ao seu aplicativo, precisará criar um objeto DataSource com base nesse serviço. É aqui que o Spring Cloud ajuda. Ele remove todo o trabalho necessário para acessar e configurar conectores de serviço e permite que você se concentre no uso desses serviços. Também expõe informações sobre a instância do aplicativo (endereço do host, porta, nome, etc.).

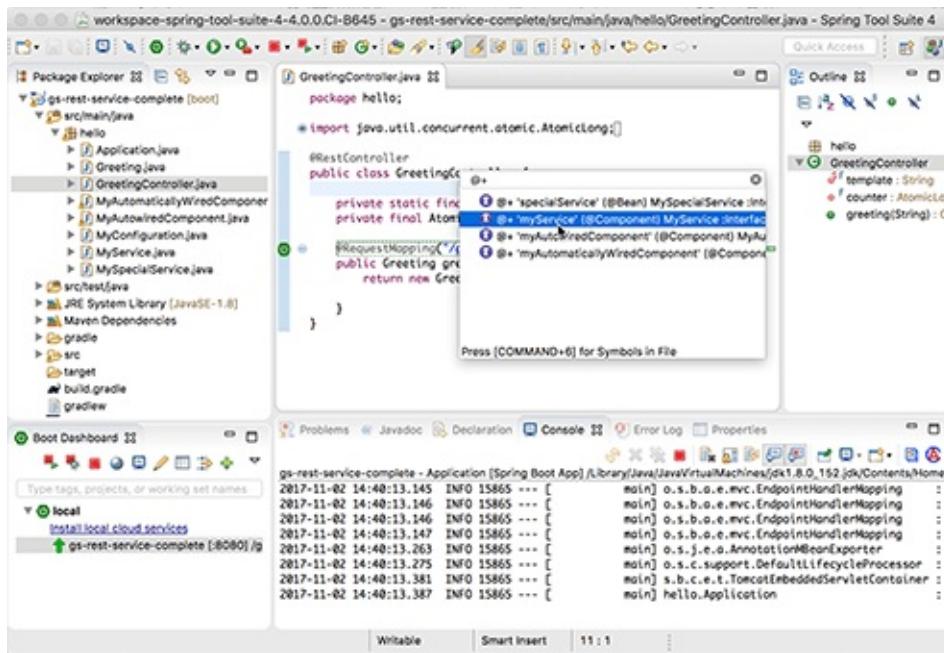
O Spring Cloud faz tudo isso de maneira independente da nuvem por meio do conceito de um Cloud Connector. Você também pode estendê-lo a outras nuvens implementando uma interface e aproveitando o restante da biblioteca. Em seguida, basta adicionar a biblioteca que contém a extensão ao classpath do seu aplicativo, não há necessidade de bifurcar e construir o Spring Cloud.

O Spring Cloud também reconhece que não pode atender a todos os serviços em todas as nuvens. Portanto, embora ofereça suporte a muitos serviços comuns prontos para uso, permite que você (ou o provedor de serviços) estenda sua funcionalidade a outros serviços. Da mesma forma que se estende para outras nuvens, você adiciona o jar que contém suas extensões de serviço ao caminho de classe do seu aplicativo.

Finalmente, ele apresenta um suporte especial para aplicativos Spring (em um módulo separado), incluindo aplicativos Spring Boot na forma de suporte à configuração Java e XML e a exposição de propriedades de aplicativos e serviços em uma forma fácil de consumir. Este é o único módulo no Spring Cloud que depende do Spring. Outros provedores de estrutura podem contribuir com suporte específico para suas estruturas de maneira similar.

Spring Tools

O Spring Tools oferece suporte para o desenvolvimento de aplicativos corporativos baseados em Spring e Spring Boot.



Fontes

- <https://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/>
- <https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-introducing-spring-boot.html>
- <https://spring.io/blog/2014/06/03/introducing-spring-cloud>

Construindo uma aplicação com Spring Boot

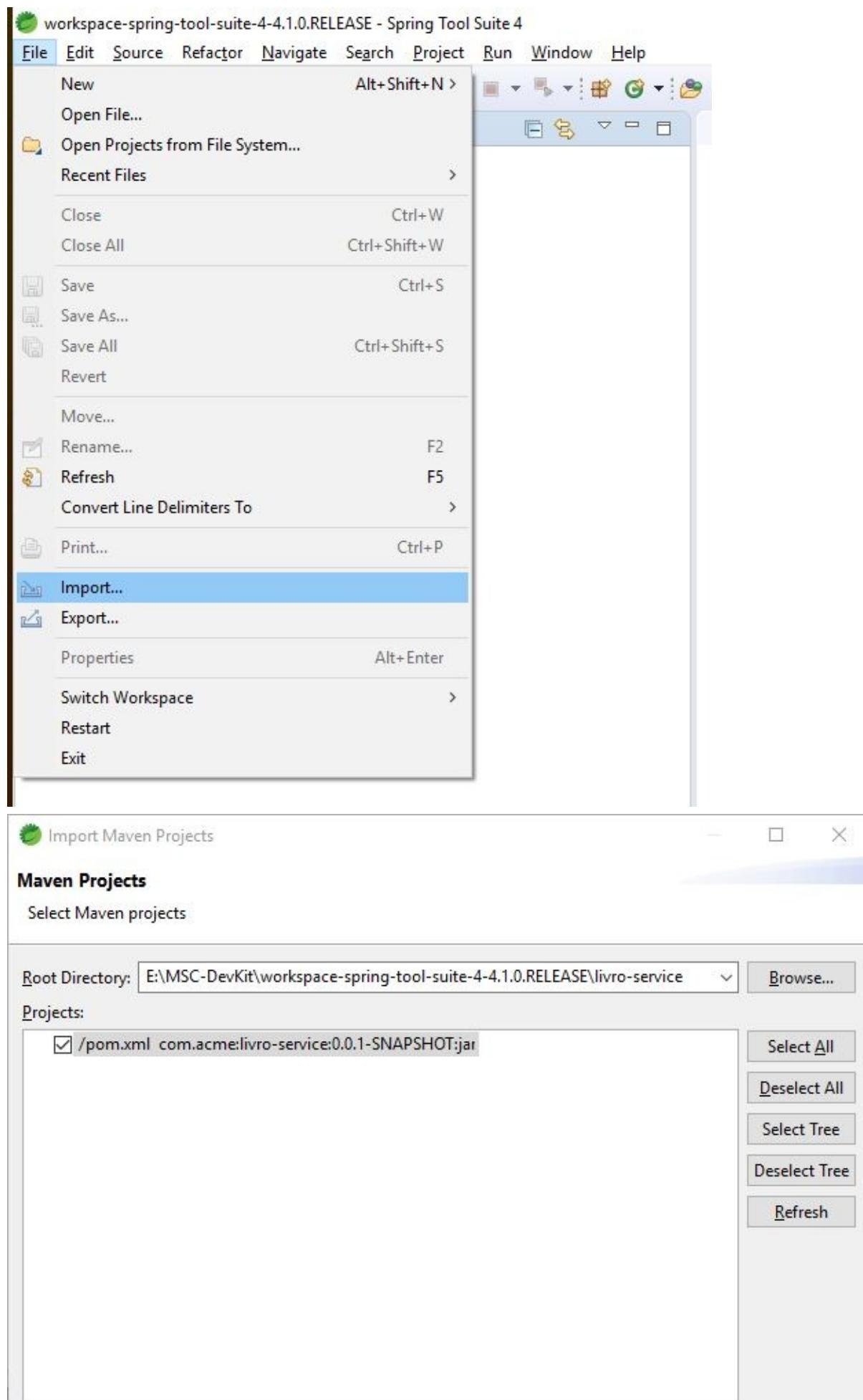
Começaremos criando o esqueleto e nosso primeiro microsserviço, o microsserviço de livros:

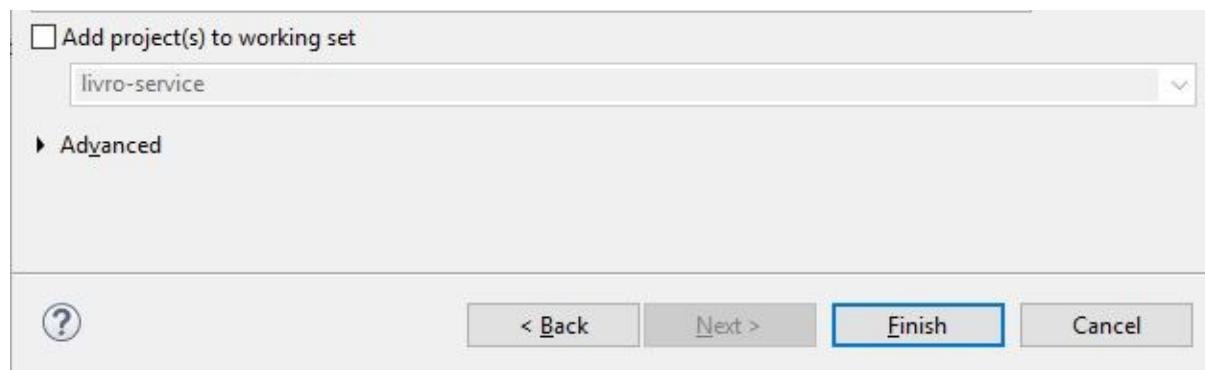
The screenshot shows the Spring Initializr interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below that, it says "Generate a Maven Project with Java 2.1.2 and Spring Boot".
Project Metadata
Artifact coordinates: com.acme
Group: com.acme
Artifact: livro-service
Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies: Web, Security, JPA, Actuator, Devtools...
Selected Dependencies: Web
Generate Project alt + ⌘
Don't know what to look for? Want more options? [Switch to the full version.](#)
start.spring.io is powered by Spring Initializr and Pivotal Web Services

Fazer o download do arquivo gerado

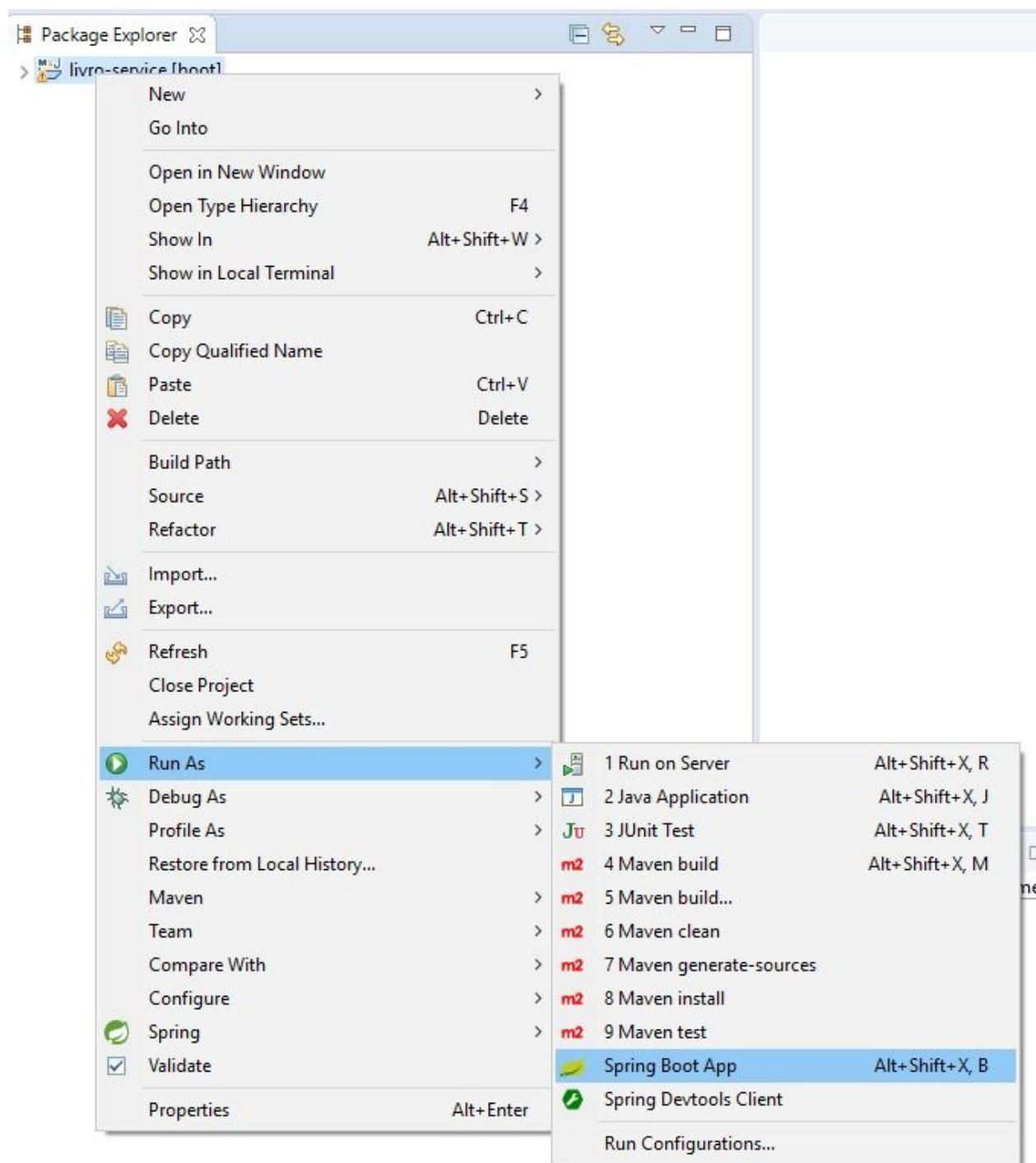
Extrair em: C:\MSC-DevKit\workspace-spring-tool-suite-4-4.1.0.RELEASE

Importar o projeto:





Executar o projeto:



Acompanhar o log:

Ver se a aplicação foi inicializada em <http://localhost:8080>:

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sat Jan 19 18:10:18 BRST 2019
There was an unexpected error (type=Not Found, status=404).
No message available.

Adicionando um conteúdo estático

Clicar com o botão direiro em src/main/resources/static -> New -> Other... -> HTML File -> index.html

Adicionar o seguinte conteúdo:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Microsserviço Livros</title>
</head>
<body>
    <h1>Microsserviço de Livros</h1>
</body>
</html>
```

Agora poderemos acessar novamente <http://localhost:8080> e ver a página que criamos:

Microsserviço de Livros

Executando via linha de comandos

Run as -> Maven Install

Via cmd iniciar o jar com o comando java -jar target\livro-service-0.0.1-SNAPSHOT.jar :

```
PS E:\MSC-DevKit\workspace-spring-tool-suite-4-4.1.0.RELEASE\livro-service>E:\MSC-DevKit\jdk-11.0.2\bin\java -jar target\livro-service-0.0.1-SNAPSHOT.jar

:: Spring Boot ::          (v2.1.2.RELEASE)

2019-01-19 18:56:57.378 INFO 18388 --- [           main] c.a.l.LivroServiceApplication        : Starting LivroServiceApplication v0.0.1-SNAPSHOT
on DESKTOP-60KS300 with PID 18388 (E:\MSC-DevKit\workspace-spring-tool-suite-4-4.1.0.RELEASE\livro-service\target\livro-service-0.0.1-SNAPSHOT.jar started by tiago in E:\MSC-DevKit\workspace-spring-tool-suite-4-4.1.0.RELEASE\livro-service)
2019-01-19 18:56:57.384 INFO 18388 --- [           main] c.a.l.LivroServiceApplication        : No active profile set, falling back to default profiles: default
2019-01-19 18:56:59.892 INFO 18388 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2019-01-19 18:56:59.942 INFO 18388 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-01-19 18:56:59.944 INFO 18388 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.14]
2019-01-19 18:56:59.962 INFO 18388 --- [           main] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: [e:\MSC-DevKit\jdk-11.0.2\bin;c:\WINDOWS\Sun\Java\bin;c:\WINDOWS\system32;c:\WINDOWS;c:\ProgramData\dockerDesktop\version-bin;c:\Program Files\docker\resources\bin;c:\Program Files (x86)\Common Files\Oracle\Java\javapath;c:\ProgramData\oracle\Java\javapath;c:\WINDOWS\system32;c:\WINDOWS;c:\WINDOWS\System32\Wbem;c:\WINDOWS\system32\WindowsPowerShell\v1.0\;c:\WINDOWS\system32\config\systemprofile\.dnx\bin;c:\Program Files\Microsoft DNx\dnvm;c:\Program Files\Microsoft Network Monitor 3;c:\Program Files (x86)\Skype\Phone\;e:\Program Files\Git\cmd;c:\WINDOWS\System32\OpenSSH\;c:\Program Files\Microsoft SQL Server\Client SDK\ODBC\13.0\Tools\Binn\;c:\Program Files (x86)\Microsoft SQL Server\140\Tools\Binn\;c:\Program Files\Microsoft SQL Server\140\Tools\Binn\;c:\Program Files\Microsoft VS Code\bin;c:\Program Files (x86)\Calibre2\;c:\Program Files\nodejs\;e:\Program Files\Oracle\VirtualBox\;c:\mobkit\Android\SDK\emulator\;c:\mobkit\Android\SDK\platform-tools\;c:\Users\tiago\AppData\local\Programs\Python\Python37\Scripts\;c:\Users\tiago\AppData\Local\Programs\Python\Python37\;c:\ProgramData\Oracle\Java\javapath;c:\WINDOWS\system32;c:\WINDOWS;c:\WINDOWS\System32\WindowsPowerShell\v1.0\;c:\Program Files (x86)\Skype\Phone\;c:\Program Files (x86)\Git\cmd;c:\Program Files (x86)\Ibm Computer Solutions\UltraEdit\;c:\kdi\node\nodejs\;c:\kdi\node\npm\;c:\Python27\;c:\phantomjs-2.1.1\bin\;c:\Users\tiago\AppData\local\Microsoft\WindowsApps\;e:\Program Files\Microsoft VS Code\bin\;c:\Users\tiago\AppData\Local\Microsoft\WindowsApps\;c:\Users\tiago\AppData\Roaming\npm\;e:\DevkitMS\apache-maven-3.6.0\bin\;]
2019-01-19 18:57:00.079 INFO 18388 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[]   : Initializing Spring embedded WebApplicationContext
2019-01-19 18:57:00.079 INFO 18388 --- [           main] o.s.web.context.ContextLoader        : Root WebApplicationContext: initialization completed in 2603 ms
2019-01-19 18:57:00.417 INFO 18388 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-01-19 18:57:00.573 INFO 18388 --- [           main] o.s.b.a.w.s.WelcomePageHandlerMapping : Adding welcome page: class path resource [static/index.html]
2019-01-19 18:57:00.757 INFO 18388 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2019-01-19 18:57:00.766 INFO 18388 --- [           main] c.a.l.LivroServiceApplication        : Started LivroServiceApplication in 4.203 seconds
(JVM running for 4.892)
```

Construindo um JAR executável

Você pode executar o aplicativo a partir da linha de comando com Gradle ou Maven. Ou você pode criar um único arquivo JAR executável que contém todas as dependências, classes e recursos necessários e executá-lo. Isso facilita o envio, a versão e a implantação do serviço como um aplicativo durante todo o ciclo de vida de desenvolvimento, em diferentes ambientes e assim por diante.

Se você estiver usando o Maven, poderá executar o aplicativo usando `./mvnw spring-boot:run`. Ou você pode construir o arquivo JAR com o pacote `./mvnw clean`.

O procedimento irá criar um JAR executável. Você também pode optar por criar um arquivo WAR clássico.

Asaída de log é exibida. O serviço deve ficar funcional dentro de alguns segundos.

Entendendo o projeto base

- pom.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <!-- Diz ao Maven para incluir as dependências do Spring Boot Starter Kit -->
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.2.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <groupId>com.acme</groupId>
  <artifactId>livro-service</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>livro-service</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>

    <!-- Diz ao Maven para incluir as dependências da Web do Spring Boot -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Diz ao Maven para incluir as dependências de testes do Spring Boot -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <!-- Diz ao Maven para incluir plugins de maven específicos do Spring para construir e implementar aplicações Spring Boot
-->
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>

```

O plugin Spring Boot Maven fornece muitos recursos convenientes:

- Ele coleta todos os jars classpath e cria um jar único e executável, o que torna mais conveniente executar e transportar seu serviço;
- Ele procura o método `public static void main()` para sinalizar como uma classe executável;
- Ele fornece um resolvedor de dependência integrado que define o número da versão para corresponder às dependências do Spring Boot. Você pode substituir qualquer versão que desejar, mas será o padrão para o conjunto de versões escolhido do Boot;
- `src/main/java/com/acme/livroservice/LivroServiceApplication.java`

```

package com.acme.livroservice;

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

/* @SpringBootApplication diz ao framework Spring Boot que esta é a classe de bootstrap para o projeto */
@SpringBootApplication
public class LivroServiceApplication {

    public static void main(String[] args) {

        /* Chamada para iniciar todo o serviço de inicialização do Spring */
        SpringApplication.run(LivroServiceApplication.class, args);
    }

}

```

`@SpringBootApplication` é uma anotação de conveniência que inclui todos os itens a seguir:

- `@Configuration` marca a classe como uma fonte de definições de bean para o contexto do aplicativo;
- `@EnableAutoConfiguration` informa ao Spring Boot para começar a adicionar beans com base nas configurações do caminho de classe, outros beans e várias configurações de propriedade;
- Normalmente você adicionaria o `@EnableWebMvc` para um aplicativo Spring MVC, mas o Spring Boot o adiciona automaticamente quando vê o **spring-webmvc** no classpath. Isso sinaliza o aplicativo como um aplicativo da Web e ativa comportamentos-chave, como a configuração de um `DispatcherServlet`.
- O `@ComponentScan` diz ao Spring para procurar outros componentes, configurações e serviços no pacote `livroservice`, permitindo que ele encontre os controladores;

O método `main()` usa o método `SpringApplication.run()` do Spring Boot para iniciar um aplicativo. Você percebeu que não havia uma única linha de XML? Nenhum arquivo **web.xml** também. Este aplicativo da web é 100% puro e você não precisa lidar com a configuração de qualquer canal ou infraestrutura.

Saiba o que você pode fazer com o Spring Boot

O Spring Boot oferece uma maneira rápida de construir aplicativos. Ele analisa seu caminho de classe e os beans que você configurou, faz suposições razoáveis sobre o que está faltando e adiciona-o. Com o Spring Boot, você pode se concentrar mais nos recursos de negócios e menos na infraestrutura.

Por exemplo, no caso do Spring MVC existem vários beans específicos que quase sempre se precisa, e o Spring Boot os adiciona automaticamente. Um aplicativo Spring MVC também precisa de um contêiner de servlet, portanto, o Spring Boot configura automaticamente o Tomcat incorporado.

Este é apenas um exemplo da configuração automática que o Spring Boot fornece. Ao mesmo tempo, o Spring Boot não atrapalha. Isso deixa você no controle com pouco esforço da sua parte.

O Spring Boot não gera código nem faz edições nos seus arquivos. Em vez disso, quando você inicia o aplicativo, o Spring Boot liga dinamicamente os beans e as configurações e os aplica ao contexto do seu aplicativo.

Inicializando o git na pasta do projeto

```

> git init
Initialized empty Git repository in E:/MSC-DevKit/workspace-spring-tool-suite-4-4.1.0.RELEASE/livro-service/.git/

```

Para comitar determinado estado do projeto, faça:

```

> git add .
> git commit -am "Mensagem de commit"

```

Fontes

- <https://spring.io/guides/gs/spring-boot/>

Entendendo REST

REST (Representational State Transfer) foi introduzido em 2000 por Roy Fielding em sua tese de doutorado. REST é um estilo arquitetural para projetar sistemas distribuídos. Não é um padrão, mas um conjunto de restrições, como ser sem estado, ter uma relação cliente/servidor e uma interface uniforme. O REST não está estritamente relacionado ao HTTP, mas é mais comumente associado a ele.

Princípios do REST

- **Recursos** expõem URLs de estrutura de diretórios facilmente compreendidos;
- As **representações** transferem JSON ou XML para representar objetos e atributos de dados;
- **Mensagens** usam métodos HTTP explicitamente (por exemplo, GET, POST, PUT e DELETE);
- Interações **sem estado** não armazenam nenhum contexto de cliente no servidor entre as solicitações. As dependências de estado limitam e restringem a escalabilidade. O cliente mantém o estado da sessão;

Métodos HTTP

Use métodos HTTP para mapear operações CRUD (criar, recuperar, atualizar, excluir) para solicitações HTTP.

GET

Recuperar informação. As solicitações GET devem ser seguras e idempotentes, ou seja, independentemente de quantas vezes ela se repete com os mesmos parâmetros, os resultados são os mesmos. Eles podem ter efeitos colaterais, mas o usuário não os espera, portanto não podem ser críticos para a operação do sistema. As solicitações também podem ser parciais ou condicionais.

Recuperar um endereço com um ID de 1:

```
GET /enderecos/1
```

POST

Solicita que o recurso na URI faça alguma coisa com a entidade fornecida. Geralmente, o POST é usado para criar uma nova entidade, mas também pode ser usado para atualizar uma entidade.

Criar um novo endereço:

```
POST /enderecos
```

PUT

Armazena uma entidade em um URI. PUT pode criar uma nova entidade ou atualizar uma existente. Uma solicitação PUT é idempotente. Idempotência é a principal diferença entre as expectativas de PUT e uma solicitação POST.

Modifique o endereço com um ID de 1:

```
PUT /enderecos/1
```

Nota: PUT substitui uma entidade existente. Se apenas um subconjunto de elementos de dados for fornecido, o restante será substituído por vazio ou nulo.

PATCH

Atualiza apenas os campos especificados de uma entidade em um URI. Uma solicitação PATCH não é segura nem idempotente. Isso porque uma operação PATCH não pode garantir que todo o recurso tenha sido atualizado.

```
PATCH /enderecos/1
```

DELETE

Solicita que um recurso seja removido; no entanto, o recurso não precisa ser removido imediatamente. Pode ser um pedido assíncrono ou de longa duração.

Exclua um endereço com um ID de 1:

```
DELETE /enderecos/1
```

Códigos de status HTTP

Códigos de status indicam o resultado da solicitação HTTP.

- **1XX** - informativo
- **2XX** - sucesso
- **3XX** - redirecionamento
- **4XX** - erro do cliente
- **5XX** - erro do servidor

Media types (Tipos de mídia)

Os cabeçalhos HTTP `Accept` e `Content-Type` podem ser usados para descrever o conteúdo enviado ou solicitado em uma solicitação HTTP. O cliente pode definir `Accept` para `application/json` se estiver solicitando uma resposta em JSON. Por outro lado, ao enviar dados, a configuração do `Content-Type` para `application/xml` informa ao cliente que os dados que estão sendo enviados na solicitação são XML.

Fontes

- <https://spring.io/understanding/REST>

Criando um end-point REST

Agora, iremos criar nosso primeiro web service RESTful com Spring.

O que será criado

Criaremos um serviço que irá aceitar um request HTTP GET para:

```
http://localhost:8080/livros
```

E responder com a representação JSON a seguir:

```
[  
  {  
    "id": 1,  
    "autor": "Miguel de Cervantes",  
    "titulo": "Don Quixote",  
    "preco": 44  
,  
  {  
    "id": 2,  
    "autor": "J. R. R. Tolkien",  
    "titulo": "O Senhor dos Anéis",  
    "preco": 23  
  }  
]
```

Crie uma classe de representação de recurso

Com o projeto que já criamos anteriormente, agora é possível criar um web service.

Começamos o processo pensando em interações de serviço.

O serviço responderá solicitações GET para /livros. A solicitação GET deve retornar uma resposta 200 OK com JSON no corpo que representa uma lista de livros.

Para modelar a representação do livro, cria-se uma classe de representação de recurso. Forneça um objeto java simples com campos, construtores e acessadores para os dados de id, autor, título e preço:

- `/src/main/java/com/acme/livroservice/Livro.java`

```

package com.acme.livroservice;

public class Livro {
    private Long id;
    private String autor;
    private String titulo;
    private Double preco;

    public Livro() {
        super();
    }

    public Livro(Long id, String autor, String titulo, Double preco) {
        super();
        this.id = id;
        this.autor = autor;
        this.titulo = titulo;
        this.preco = preco;
    }

    /* Getters e Setters */

    // Dica: O método abaixo pode ser gerado automaticamente clicando com o botão direito na classe
    // e em Source -> Generate toString()
    @Override
    public String toString() {
        return "Livro [id=" + id + ", autor=" + autor + ", titulo=" + titulo + ", preco=" + preco + "]";
    }
}

```

Criar um controlador de recursos

Na abordagem do Spring para a criação de serviços Web RESTful, as solicitações HTTP são tratadas por um controlador. Esses componentes são facilmente identificados pela anotação `@RestController`, e a classe `LivrosController` abaixo lida com solicitações `GET` para `/livros` retornando uma lista de livros:

- src/main/java/com/acme/livroservice/LivrosController.java

```

package com.acme.livroservice;

import java.util.ArrayList;
import java.util.List;

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

@RestController
public class LivrosController {

    @RequestMapping("/livros")
    public List<Livro> getLivros() {
        ArrayList<Livro> livros = new ArrayList<Livro>();

        Livro l1 = new Livro(11, "Miguel de Cervantes", "Don Quixote", 144.0);
        Livro l2 = new Livro(21, "J. R. R. Tolkien", "O Senhor dos Anéis", 123.0);

        livros.add(l1);
        livros.add(l2);

        return livros;
    }
}

```

Este controlador é conciso e simples, mas há muita coisa acontecendo embaixo do capô. Vamos dividi-lo passo a passo.

A anotação `@RequestMapping` garante que as solicitações HTTP `/livros` sejam mapeadas para o método `getLivros()`.

O código não especifica `GET`, `PUT` ou `POST` pois `@RequestMapping` mapeia todas as operações HTTP por padrão.

Utiliza-se `@RequestMapping(method=GET)` para restringir esse mapeamento.

A implementação do corpo do método cria e retorna um novo objeto `ArrayList<Livro>` com uma lista de livros que é populada.

Uma diferença fundamental entre um controlador de páginas tradicional e o controlador de serviço da web RESTful é a maneira como o corpo da resposta HTTP é criado. Em vez de depender de uma tecnologia de visualização para executar a renderização do lado do servidor dos dados dos livros para HTML, esse controlador de serviço da Web RESTful simplesmente preenche e retorna um objeto `ArrayList<Livro>`. Os dados do objeto serão gravados diretamente na resposta HTTP como JSON.

Esse código usa a anotação `@RestController` que marca a classe como um controlador em que cada método retorna um objeto de domínio em vez de um modo de exibição. É uma abreviação de `@Controller` e `@ResponseBody` reunidos.

O objeto `ArrayList<Livro>` deve ser convertido em JSON. Graças ao suporte ao conversor de mensagens HTTP do Spring, não é necessário fazer essa conversão manualmente. Como `Jackson 2` está no classpath, o conversor `MappingJackson2HttpMessageConverter` do Spring é automaticamente escolhido para converter a lista de livros em JSON.

Agora já podemos compilar e executar nossa aplicação, acessando o endereço <http://localhost:8080/livros>, devemos ter uma resposta como a seguinte:

```
[  
 {  
   "id": 1,  
   "autor": "Miguel de Cervantes",  
   "titulo": "Don Quixote",  
   "preco": 44  
,  
 {  
   "id": 2,  
   "autor": "J. R. R. Tolkien",  
   "titulo": "O Senhor dos Anéis",  
   "preco": 23  
 }  
 ]
```

Mas e agora, se quisermos adicionar mais um livro? Devemos então alterar o código da aplicação:

- src/main/java/com/acme/livroservice/LivrosController.java

```

package com.acme.livroservice;

import java.util.ArrayList;
import java.util.List;

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

@RestController
public class LivrosController {

    @RequestMapping("/livros")
    public List<Livro> getLivros() {
        ArrayList<Livro> livros = new ArrayList<Livro>();

        Livro l1 = new Livro(1, "Miguel de Cervantes", "Don Quixote", 144.0);
        Livro l2 = new Livro(2, "J. R. R. Tolkien", "O Senhor dos Anéis", 123.0);
        Livro l3 = new Livro(3, "Antoine de Saint-Exupéry", "O Pequeno Príncipe", 152.0);

        livros.add(l1);
        livros.add(l2);
        livros.add(l3);

        return livros;
    }
}

```

Após compilar e executar nossa aplicação, acessando o endereço <http://localhost:8080/livros>, devemos ter uma resposta como a seguinte:

```
[
{
    "id": 1,
    "autor": "Miguel de Cervantes",
    "titulo": "Don Quixote",
    "preco": 44
},
{
    "id": 2,
    "autor": "J. R. R. Tolkien",
    "titulo": "O Senhor dos Anéis",
    "preco": 23
},
{
    "id": 3,
    "autor": "Antoine de Saint-Exupéry",
    "titulo": "O Pequeno Príncipe",
    "preco": 52
}
]
```

Mas para isso, tivemos que parar o servidor, alterar o código, e executar o servidor novamente...

A boa notícia é que podemos deixar este processo mais ágil!

Developer Tools

O Spring Boot inclui um conjunto adicional de ferramentas que podem tornar a experiência de desenvolvimento de aplicativos um pouco mais agradável. O módulo `spring-boot-devtools` pode ser incluído em qualquer projeto para fornecer recursos adicionais de tempo de desenvolvimento. Para incluir suporte a devtools, adicione a dependência do módulo à sua compilação, conforme mostrado na listagem a seguir para Maven:

- `pom.xml`

```
<!-- Código anterior omitido -->
<dependencies>

    <!-- Dependências atuais omitidas -->

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>

</dependencies>
<!-- Código posterior omitido -->
```

Restart automático

Aplicativos que usam `spring-boot-devtools` são reiniciados automaticamente sempre que os arquivos no caminho de classe são alterados. Isso pode ser um recurso útil ao trabalhar em um IDE, pois fornece um loop de feedback muito rápido para alterações de código. Por padrão, qualquer entrada no caminho de classe que aponta para uma pasta é monitorada quanto a alterações. Observe que determinados recursos, como ativos estáticos e templates de visualização, não precisam reiniciar o aplicativo para que as alterações sejam percebidas.

Testando o restart automático

Inclua a dependência do `spring-boot-devtools` no `pom.xml` do projeto, execute novamente o projeto, altere o valor dos livros e verifique que, mesmo sem reiniciar manualmente a aplicação, as alterações são percebidas ao acessar <http://localhost:8080/livros>

Recuperando um Recurso

Agora que já temos uma base sólida, podemos começar a criar outras funcionalidades ao nosso microsserviço.

Recuperando um livro específico

Criaremos um serviço que irá aceitar um request HTTP GET para:

```
http://localhost:8080/livros/:id
```

E responder com a representação JSON como a seguir:

```
{
  "id": 1,
  "autor": "Miguel de Cervantes",
  "titulo": "Don Quixote",
  "preco": 44
}
```

Criar mais um método no controlador

Inicialmente vamos apenas incluir um novo método na classe:

- src/main/java/com/acme/livroservice/LivrosController.java

```
package com.acme.livroservice;

import java.util.ArrayList;
import java.util.List;

// Novidade aqui
import org.springframework.web.bind.annotation.PathVariable;

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

@RestController
public class LivrosController {

    @RequestMapping("/livros")
    public List<Livro> getLivros() {
        // Código atual omitido
    }

    // Novidade aqui
    @RequestMapping("/livros/{id}")
    public Livro getLivroPorId(@PathVariable Long id) {
        System.out.println("id: " + id);
        Livro l = new Livro(11, "Miguel de Cervantes", "Don Quixote", 144.0);
        return l;
    }
}
```

Vamos fazer o `System.out.println("id: " + id)` somente para conferir se realmente o parâmetro está sendo capturado.

Com isso, acessando no navegador o endereço <http://localhost:8080/livros/1> obtemos o seguinte JSON:

```
{
  "id": 1,
  "autor": "Miguel de Cervantes",
  "titulo": "Don Quixote",
  "preco": 144
}
```

Muito bem, mas seria mais interessante que tivessémos uma lista de livros em memória e ela pudesse ser utilizada tanto para listar os livros quanto para pesquisar.

Precisamos deixar uma mesma lista de livros disponível para todos os métodos. Poderíamos pensar em manter uma variável estática, porém, o Spring nos fornece uma maneira mais elegante de fazer isso.

Beans e Escopos

O escopo de um bean define o ciclo de vida e a visibilidade desse bean nos contextos nos quais ele é usado.

O Spring define seis tipos de escopos:

- **singleton**: Definir um bean com escopo **singleton** significa que o contêiner cria uma única instância desse bean, e todas as solicitações para esse nome de bean retornarão o mesmo objeto, que é armazenado em cache. Quaisquer modificações no objeto serão refletidas em todas as referências ao bean. Este escopo é o valor padrão se nenhum outro escopo for especificado;
- **prototype**: Um bean com escopo de **prototype** retornará uma instância diferente toda vez que for solicitado do contêiner;
- **request**: O escopo de **request** cria uma instância de bean para uma única solicitação HTTP;
- **session**: O escopo de **session** é criado para uma sessão HTTP;
- **application**: O escopo de **application** cria a instância do bean para o ciclo de vida de um `ServletContext`. Isso é semelhante ao escopo singleton, mas há uma diferença muito importante em relação ao escopo do bean. Quando beans tem escopo **application**, a mesma instância do bean é compartilhada entre vários aplicativos baseados em servlet em execução no mesmo `ServletContext`, enquanto os beans com escopo singleton para um único contexto de aplicativo;
- **websocket**: Os beans do escopo do `WebSocket` quando acessados pela primeira vez são armazenados nos atributos da sessão do `WebSocket`. A mesma instância do bean é então retornada sempre que esse bean é acessado durante toda a sessão do `WebSocket`. Também podemos dizer que exibe comportamento singleton, mas limitado a uma sessão `WebSocket` apenas.

Os últimos quatro escopos mencionados request, session, application e websocket estão disponíveis apenas para aplicativos web.

Tornando a lista de livros um `singleton`

Agora que já entendemos como funcionam os escopos no Spring, podemos tornar nossa lista de livros um **singleton** para que possa ser compartilhada em nosso microsserviço.

Primeiramente devemos criar um método **fábrica** que será responsável pela criação do bean quando ele for requisitado. Podemos colocar este método diretamente na classe `LivroServiceApplication`:

- `src/main/java/com/acme/livroservice/LivroServiceApplication.java`

```

package com.acme.livroservice;

import java.util.ArrayList;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class LivroServiceApplication {

    // Novidade aqui
    @Bean
    public ArrayList<Livro> listaLivros() {

        ArrayList<Livro> livros = new ArrayList<Livro>();

        Livro l1 = new Livro(11, "Miguel de Cervantes", "Don Quixote", 144.0);
        Livro l2 = new Livro(21, "J. R. R. Tolkien", "O Senhor dos Anéis", 123.0);
        Livro l3 = new Livro(31, "Antoine de Saint-Exupéry", "O Pequeno Príncipe", 152.0);
        Livro l4 = new Livro(41, "Charles Dickens", "Um Conto de Duas Cidades", 35.0);

        livros.add(l1);
        livros.add(l2);
        livros.add(l3);
        livros.add(l4);

        return livros;
    }

    public static void main(String[] args) {
        SpringApplication.run(LivroServiceApplication.class, args);
    }
}

```

Agora vamos ajustar nosso `controller` para que utilize este bean:

- src/main/java/com/acme/livroservice/LivrosController.java

```

package com.acme.livroservice;

// Código atual omitido

import javax.annotation.Resource;

@RestController
public class LivrosController {

    // Novidade aqui
    @Resource
    private ArrayList<Livro> listaLivros;

    @RequestMapping("/livros")
    public List<Livro> getLivros() {

        // Novidade aqui
        return listaLivros;
    }

    @RequestMapping("/livros/{id}")
    public Livro getLivroPorId(@PathVariable Long id) {
        // Novidade aqui
        return listaLivros.stream().filter(l -> l.getId().equals(id)).findFirst().orElse(null);
    }
}

```

Muito bom! Teste agora no navegador e veja que já é possível pesquisar um livro a partir de seu ID, mas o que acontece caso um ID de livro inexistente seja informado?

Retornando códigos de status customizados

É importante expressar claramente o resultado de uma solicitação para um cliente e usar a semântica completa e rica do protocolo HTTP. Por exemplo, se algo der errado com uma solicitação, enviar um código de erro específico para cada tipo de problema possível permitirá que o cliente exiba uma mensagem de erro apropriada para o usuário.

O Spring fornece algumas maneiras principais de retornar códigos de status personalizados de suas classes do Controller:

- Usando um `ResponseEntity` ;
- Usando a anotação `@ResponseStatus` em classes de exceção;

Essas opções não são mutuamente exclusivas; longe disso, eles podem realmente complementar uma a outra.

Retornando códigos de status por meio de um `ResponseEntity`

```
@RequestMapping(value = "/controller", method = RequestMethod.GET)
@ResponseBody
public ResponseEntity sendViaResponseEntity() {
    return new ResponseEntity(HttpStatus.NOT_ACCEPTABLE);
}
```

Retornando códigos de status por meio de uma exceção

Vamos adicionar um segundo método ao controlador para demonstrar como usar uma exceção para retornar um código de status:

```
@RequestMapping(value = "/exception", method = RequestMethod.GET)
@ResponseBody
public ResponseEntity sendViaException() {
    throw new ForbiddenException();
}
```

Ao receber uma requisição GET para `/exception`, o Spring lançará uma `ForbiddenException`. Esta é uma exceção personalizada que definiremos em uma classe separada:

```
@ResponseStatus(HttpStatus.FORBIDDEN)
public class ForbiddenException extends RuntimeException {}
```

Nenhum código é necessário nesta exceção. Todo o trabalho é feito pela anotação `@ResponseStatus`.

Neste caso, quando a exceção é lançada, o controlador que a lançou retorna uma resposta com o código de resposta 403 (Forbidden). Se necessário, você também pode adicionar uma mensagem na anotação que será retornada junto com a resposta.

Nesse caso, a classe ficaria assim:

```
@ResponseStatus(value = HttpStatus.FORBIDDEN, reason="Para mostrar uma mensagem customizada")
public class ForbiddenException extends RuntimeException {}
```

É importante observar que, embora seja tecnicamente possível fazer uma exceção retornar qualquer código de status, na maioria dos casos, faz sentido usar exceções para códigos de erro (4XX e 5XX).

ResponseStatusException

`ResponseStatusException` é uma alternativa programática ao `@ResponseStatus` e é a classe base para exceções usadas para aplicar um código de status a uma resposta HTTP. É uma `RuntimeException` e, portanto, não precisa ser explicitamente adicionada a uma assinatura de método.

Spring fornece 3 construtores para gerar `ResponseStatusException`:

```
ResponseStatusException(HttpStatus status)
ResponseStatusException(HttpStatus status, java.lang.String reason)
ResponseStatusException(
    HttpStatus status,
    java.lang.String reason,
    java.lang.Throwable cause
)
```

Argumentos do construtor de `ResponseStatusException`:

- `status` - um status HTTP definido para resposta HTTP
- `reason` - uma mensagem explicando a exceção definida como resposta HTTP
- `cause` - uma causa `Throwable` do `ResponseStatusException`

Nota: no Spring, o `HandlerExceptionResolver` intercepta e processa qualquer exceção gerada e não tratada por um Controller.

Um desses manipuladores, `ResponseStatusExceptionResolver`, procura por qualquer `ResponseStatusException` ou exceções não identificadas anotadas por `@ResponseStatus` e, em seguida, extrai o código e a razão HTTP Status e as inclui na resposta HTTP.

O uso de `ResponseStatusException` tem alguns benefícios:

- Em primeiro lugar, exceções do mesmo tipo podem ser processadas separadamente e diferentes códigos de status podem ser definidos na resposta, reduzindo o acoplamento;
- Em segundo lugar, evita a criação de classes de exceção adicionais desnecessárias;
- Por fim, ele fornece mais controle sobre o tratamento de exceções, já que as exceções podem ser criadas programaticamente;

Alterando nosso end-point para retornar o código de erro 404 caso o livro não seja encontrado

Agora que já sabemos como retornar um código de erro, vamos ajustar nosso `controller` para que lance a exceção caso o livro não seja encontrado:

- `src/main/java/com/acme/livroservice/LivrosController.java`

```

package com.acme.livroservice;

// Código atual omitido

// Novidade aqui
import org.springframework.web.server.ResponseStatusException;

@RestController
public class LivrosController {

    // Código atual omitido

    @RequestMapping("/livros/{id}")
    public Livro getLivroPorId(@PathVariable Long id) {
        return listaLivros.stream().filter(l -> l.getId().equals(id)).findFirst()
            // Novidade aqui
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
    }
}

```

É hora de testarmos este novo recurso de nossa aplicação, acesse <http://localhost:8080/livros/9999>:

Vemos que de fato, agora o código de retorno HTTP do serviço passou a ser 404, no entanto, também é retornada uma página de erro desnecessária.

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Jan 20 00:57:43 BRST 2019
 There was an unexpected error (type=Not Found, status=404).
 Livro não encontrado
 org.springframework.web.server.ResponseStatusException: 404 NOT_FOUND "Livro não encontrado"
 at com.acme.livroservice.LivrosController.lambda\$1(LivrosController.java:28)
 at java.base/java.util.Optional.orElseThrow(Optional.java:408)
 at com.acme.livroservice.LivrosController.getLivroPorId(LivrosController.java:28)
 at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at java.base/jdk/internal/reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
 at java.base/jdk/internal/reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

Status	Method	File	Domain	Cause	Type	Transferred	Size	0 ms	160 ms	320 ms
404	GET	999	localhost:8080	document	html	5,87 KB	5,73 KB	22 ms		
200	GET	favicon.ico	localhost:8080	img	x-icon	cached	946 B			

2 requests | 6,65 KB / 5,87 KB transferred | Finish: 331 ms | DOMContentLoaded: 325 ms | load: 349 ms

Incluindo um Recurso

Nosso microsserviço já é capaz de listar os livros e buscar um livro a partir de seu código, além disso, ele retorna códigos de erro HTTP adequados na ocorrência de erros, nosso próximo passo é permitir a inclusão de novos livros.

Definindo uma URL base padrão para o recurso

Mas antes permitir a inclusão de novos livros, podemos fazer uma melhoria em nosso código, você já deve ter percebido que a URL base do controller irá se repetir em todos os métodos e, para evitar a repetição de código e facilitar a manutenção, podemos centralizar esta configuração:

- src/main/java/com/acme/livroservice/LivrosController.java

```
package com.acme.livroservice;

// Código atual omitido

@RestController
// Novidade aqui
@RequestMapping("/livros")
public class LivrosController {

    @Resource
    private ArrayList<Livro> listaLivros;

    // Novidade aqui
    @RequestMapping("/{id}")
    public Livro getLivroPorId(@PathVariable Long id) {
        return listaLivros.stream().filter(l -> l.getId().equals(id)).findFirst()
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
    }
}
```

Ótimo, agora se precisarmos alterar a URL base de nosso recurso precisaremos alterar apenas em um local.

Tratando requisições do tipo POST

Nossa próxima função será algo como:

```
public void adicionarLivro(Livro livro) {
    listaLivros.add(livro);
}
```

Mas o problema é definir a URL para a qual o recurso será mapeado, pois requisições do tipo GET <http://localhost:8080/livros> e POST <http://localhost:8080/livros> devem ser mapeadas para funções diferentes.

Para resolver isso, devemos alterar as anotações `RequestMapping` para anotações mais específicas como `GetMapping` e `PostMapping`.

- src/main/java/com/acme/livroservice/LivrosController.java

```

package com.acme.livroservice;

// Código atual omitido

// Novidade aqui
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;

@RestController
// Novidade aqui
@RequestMapping("/livros")
public class LivrosController {

    @Resource
    private ArrayList<Livro> listaLivros;

    // Novidade aqui
    @GetMapping
    public List<Livro> getLivros() {
        return listaLivros;
    }

    // Novidade aqui
    @GetMapping("/{id}")
    public Livro getLivroPorId(@PathVariable Long id) {
        return listaLivros.stream().filter(l -> l.getId().equals(id)).findFirst()
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
    }

    // Novidade aqui
    @PostMapping
    public void adicionarLivro(Livro livro) {
        System.out.println("Função adicionarLivro acionada");
    }
}

```

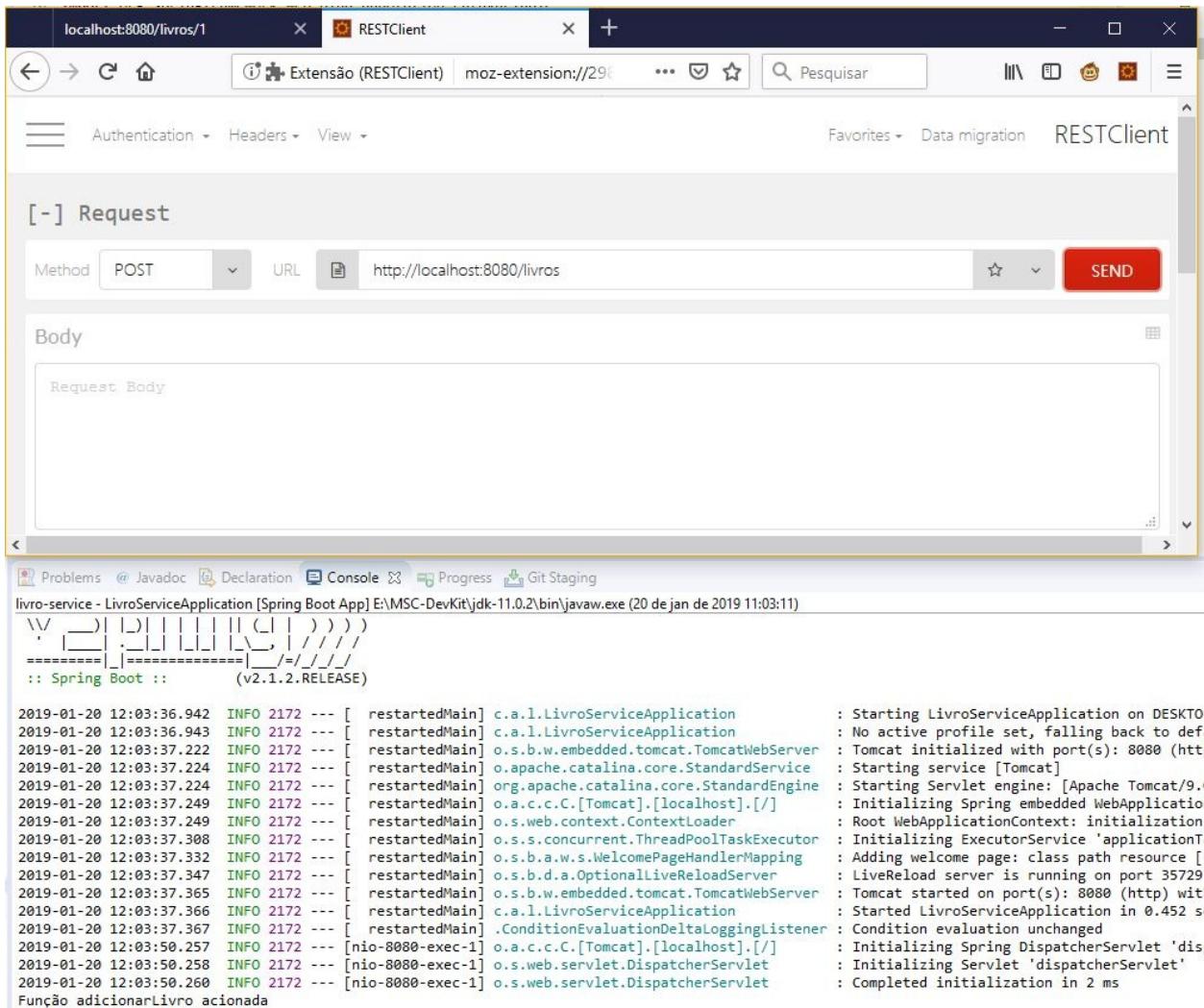
Mas como poderemos testar as chamadas para um end-point que espera o método POST se sempre que colocamos um endereço na navegador ele faz um GET?

Depurando end-points REST

Para podermos depurar acionamentos a end-points REST podemos utilizar uma série de ferramentas, tanto em linha de comando como ferramentas gráficas, entre elas:

- curl: Ferramenta de linha de comando Linux - <https://curl.haxx.se/>;
- RESTClient: Extensão do Firefox - <https://addons.mozilla.org/pt-BR/firefox/addon/restclient>;
- Postman: - Extensão do Chrome - <https://www.getpostman.com/>;

Qualquer uma destas ferramentas permite a execução de requisições HTTP arbitrárias, informando a URL, método HTTP, cabeçalhos e um corpo da requisição.



Ótimo, agora podemos testar outros tipos de requisição com a ajuda destas ferramentas, nosso próximo passo é enviar um livro através da requisição REST e recebê-lo em nosso método.

Enviando dados para o End-Point

Utilizando a anotação `@RequestBody` podemos anotar um parâmetro da função do controller que será preenchido com os dados da requisição.

- src/main/java/com/acme/livroservice/LivrosController.java

```

package com.acme.livroservice;

// Código atual omitido

// Novidade aqui
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;

@RestController
// Novidade aqui
@RequestMapping("/livros")
public class LivrosController {

    @Resource
    private ArrayList<Livro> listaLivros;

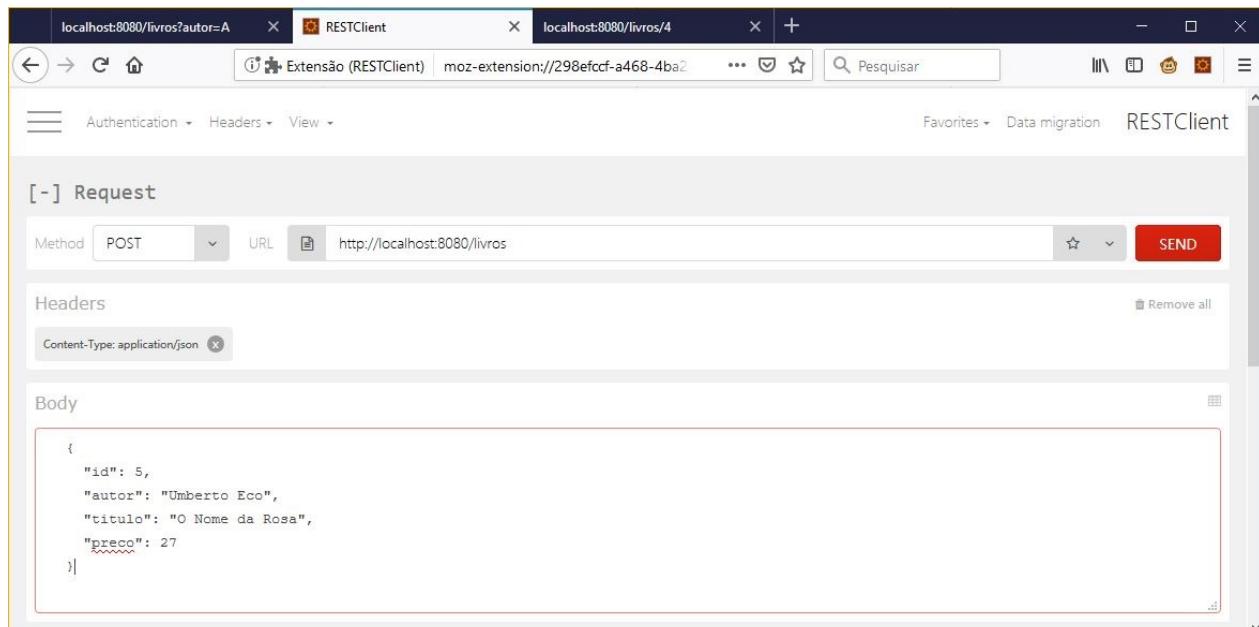
    // Novidade aqui
    @GetMapping
    public List<Livro> getLivros() {
        return listaLivros;
    }

    // Novidade aqui
    @GetMapping("/{id}")
    public Livro getLivroPorId(@PathVariable Long id) {
        return listaLivros.stream().filter(l -> l.getId().equals(id)).findFirst()
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
    }

    @PostMapping
    public void adicionarLivro(@RequestBody Livro livro) {
        System.out.println("Função adicionarLivro acionada");
    }
}

```

Mas como poderemos enviar os dados do livro? É aí que entra novamente o auxílio do RESTClient. Vamos realizar uma nova requisição contra o end-point <http://localhost:8080/livros> utilizando o método http POST, mas agora, iremos enviar os dados do livro a ser cadastrado no corpo da requisição como um JSON, não podemos nos esquecer também de incluir um cabeçalho na requisição com `Content-Type: application/json`:



Vamos também ajustar nosso controller para que imprima no console os dados recebidos do livro:

- src/main/java/com/acme/livroservice/LivrosController.java

```
// Código atual omitido

@RestController
@RequestMapping("/livros")
public class LivrosController {

    // Código atual omitido

    @PostMapping
    public void adicionarLivro(@RequestBody Livro livro) {
        System.out.println(livro);
    }
}
```

Ao efetuarmos a chamada POST utilizando o RESTClient devemos ver algo como a linha abaixo no console:

```
Livro [id=5, autor=Umberto Eco, titulo=O Nome da Rosa, preco=27.0]
```

Para que o livro possa de fato ser incluído na listagem, devemos encontrar um ID válido para ele e inserir o novo livro em `listaLivros`, faremos isso agora:

- src/main/java/com/acme/livroservice/LivrosController.java

```
// Código atual omitido

@RestController
@RequestMapping("/livros")
public class LivrosController {

    // Código atual omitido

    @PostMapping
    public void adicionarLivro(@RequestBody Livro livro) {
        Long max = listaLivros.stream().mapToLong(l -> l.getId()).max().orElse(0);
        livro.setId(max + 1);
        System.out.println(livro);
        listaLivros.add(livro);
    }
}
```

Faça alguns testes incluindo mais livros, não é necessário informar o campo ID no JSON, pois ele será calculado.

Mas ainda temos espaço para melhorar, pois na realidade, a resposta adequada a um método POST de inclusão bem sucedido é 201, e, ele deve retornar a entidade que acabou de ser incluída:

- src/main/java/com/acme/livroservice/LivrosController.java

```
// Código atual omitido

// Novidade aqui
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@RestController
@RequestMapping("/livros")
public class LivrosController {

    // Código atual omitido

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Livro adicionarLivro(@RequestBody Livro livro) {
        Long max = listaLivros.stream().mapToLong(l -> l.getId()).max().orElse(0);
        livro.setId(max + 1);
        System.out.println(livro);
        listaLivros.add(livro);
        return livro;
    }
}
```

Neste momento não validaremos as entidades, isso será feito nos próximos capítulos.

Logs no Spring Boot

Deixar códigos do tipo `System.out.println` espalhados em nosso projeto não é uma boa prática de programação, no entanto, as informações obtidas são muito importantes, tanto durante a fase de desenvolvimento da aplicação quanto no momento de operação para identificarmos eventuais erros que podem estar ocorrendo.

Neste ponto, o Spring Boot é um framework muito útil pois nos permite esquecer a maioria das definições de configuração, muitas das quais ele autonomiza com propriedade.

No caso de log, a única dependência obrigatória é o Apache Commons Logging.

Não precisaremos nem incluí-lo como dependência pois no Spring 5 (Spring Boot 2.x) ele é fornecido pelo módulo `spring-jcl` do Spring Framework.

O nível de log padrão do Logger é predefinido para INFO, o que significa que as mensagens TRACE e DEBUG não estão visíveis.

Para ativá-los sem alterar a configuração, podemos passar os argumentos `--debug` ou `--trace` na linha de comando:

```
java -jar target/spring-boot-logging-0.0.1-SNAPSHOT.jar --trace
```

O Spring Boot também nos dá acesso a uma configuração de nível de log mais refinada por meio de variáveis de ambiente que podem ser configuradas no arquivo `application.properties`:

```
logging.level.root=WARN
logging.level.com.baeldung=TRACE
```

Abaixo, temos um exemplo de como realizar o log de informações em um controller:

```
@RestController
public class LoggingController {

    Logger logger = LoggerFactory.getLogger(LoggingController.class);

    @RequestMapping("/")
    public String index() {
        logger.trace("A TRACE Message");
        logger.debug("A DEBUG Message");
        logger.info("An INFO Message");
        logger.warn("A WARN Message");
        logger.error("An ERROR Message");

        return "Howdy! Check out the Logs to see the output...";
    }
}
```

Melhorando nossos Logs

Agora que já sabemos como exibir corretamente as mensagens de log usando o Spring Boot, vamos ajustar nosso controller com mensagens de log adequadas:

- `src/main/java/com/acme/livroservice/LivrosController.java`

```
// Código atual omitido

// Novidade aqui
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@RestController
@RequestMapping("/livros")
public class LivrosController {

    // Novidade aqui
    Logger logger = LoggerFactory.getLogger(LivrosController.class);

    @Resource
    private ArrayList<Livro> listaLivros;

    @GetMapping
    public List<Livro> getLivros() {

        // Novidade aqui
        logger.info("getLivros");

        return listaLivros;
    }

    @GetMapping("/{id}")
    public Livro getLivroPorId(@PathVariable Long id) {

        // Novidade aqui
        logger.info("getLivroPorId: " + id);

        return listaLivros.stream().filter(l -> l.getId().equals(id)).findFirst()
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Livro adicionarLivro(@RequestBody Livro livro) {

        // Novidade aqui
        logger.info("adicionarLivro: " + livro);

        Long max = listaLivros.stream().mapToLong(l -> l.getId()).max().orElse(0);
        livro.setId(max + 1);
        listaLivros.add(livro);

        // Novidade aqui
        logger.info("adicionarLivro: " + livro + " adicionado com sucesso");

        return livro;
    }
}
```

Alterando Recursos

Ótimo, agora os usuários de nosso microsserviço já podem listar, recuperar e incluir recursos. Chegou a hora de permitirmos a alteração e a exclusão de recursos.

Alterando um Recurso

A alteração de recursos é muito semelhante à criação, no entanto, utilizamos o método PUT. Outra diferença é que a URL do recurso deve conter o id do livro a ser alterado, por exemplo <http://localhost:8080/livros/4> a seguir temos o código proposto:

- src/main/java/com/acme/livroservice/LivrosController.java

```
// Código atual omitido

// Novidade aqui
import org.springframework.web.bind.annotation.PutMapping;

@RestController
@RequestMapping("/livros")
public class LivrosController {

    // Código atual omitido

    // Novidade aqui
    @PutMapping("/{id}")
    public Livro atualizarLivro(@RequestBody Livro livro, @PathVariable Long id) {
        logger.info("atualizarLivro: " + livro + " id: " + id);
        Livro livroSalvo = listaLivros.stream().filter(l -> l.getId().equals(id)).findFirst()
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));

        livroSalvo.setAutor(livro.getAutor());
        livroSalvo.setPreco(livro.getPreco());
        livroSalvo.setTitulo(livro.getTitulo());

        return livroSalvo;
    }
}
```

Teste a funcionalidade utilizando o RESTClient, pra isso, altere um recurso existente e em seguida o recupere para verificar se as alterações foram realmente persistidas, em seguida, tente alterar um recurso inexistente e veja se o microsserviço está devolvendo um erro correspondente adequado.

Excluindo um Recurso

Falta pouco para completarmos as funcionalidades do CRUD (Create Retrieve Update Delete) de nosso microsserviço, mas ainda precisamos ser capazes de excluir livros, faremos isso agora, é uma funcionalidade simples dado o que já fizemos anteriormente.

O método HTTP utilizado para se apagar um recurso é o DELETE, vejamos como ficará nosso código:

```
// Código atual omitido

// Novidade aqui
import org.springframework.web.bind.annotation.DeleteMapping;

@RestController
@RequestMapping("/livros")
public class LivrosController {

    // Código atual omitido

    // Novidade aqui
    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void excluirLivro(@PathVariable Long id) {
        logger.info("excluirLivro: " + id);

        Livro livro = listaLivros.stream().filter(l -> l.getId().equals(id)).findFirst()
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));

        listaLivros.remove(livro);
    }
}
```

Pesquisando por Recursos

Concluímos todas as funcionalidades básicas de nosso microserviço, mas vamos colocar ainda uma funcionalidade adicional, que é a possibilidade de pesquisar livros com base em seus atributos de nome e autor, faremos isso utilizando anotação `RequestParam`:

```
// Código atual omitido

// Novidade aqui
import java.util.Optional;
import java.util.stream.Collectors;
import org.springframework.web.bind.annotation.RequestParam;

@RestController
@RequestMapping("/livros")
public class LivrosController {

    // Código atual omitido

    // Novidade aqui
    @GetMapping
    public List<Livro> getLivros(@RequestParam("autor") Optional<String> autor,
                                   @RequestParam("titulo") Optional<String> titulo) {
        logger.info("getLivros - autor: " + autor.orElse("Não informado") + " titulo: " + titulo.orElse("Não informado"));

        List<Livro> listaRetorno = listaLivros;

        if (autor.isPresent()) {
            listaRetorno = listaRetorno.stream()
                .filter(l -> l.getAutor().toUpperCase().contains(autor.get().toUpperCase()))
                .collect(Collectors.toList());
        }

        if (titulo.isPresent()) {
            listaRetorno = listaRetorno.stream()
                .filter(l -> l.getTitulo().toUpperCase().contains(titulo.get().toUpperCase()))
                .collect(Collectors.toList());
        }

        return listaRetorno;
    }
}
```


Persistindo em um Banco de Dados

Nossas funcionalidades do CRUD estão completas, iremos agora ajustar nosso projeto para que faça a persistência em uma base de dados.

H2

O H2 é um banco de dados Java leve e de código aberto. Pode ser incorporado em aplicativos Java ou executado no modo cliente-servidor. Principalmente, o banco de dados H2 pode ser configurado para ser executado como banco de dados em memória, o que significa que os dados não persistirão no disco. Por causa do banco de dados embutido, ele não é usado em produção, mas é usado principalmente para desenvolvimento e testes.

Este banco de dados pode ser usado no modo incorporado ou no modo de servidor. A seguir estão as principais características do banco de dados H2:

- Extremamente rápido, código aberto, API JDBC
- Disponível nos modos incorporado e servidor; bancos de dados na memória
- Aplicativo de console baseado em navegador
- Pequena pegada - cerca de 1,5 MB de tamanho de arquivo jar

JPA

Qualquer aplicativo corporativo executa operações em banco de dados, armazenando e recuperando grandes quantidades de dados. Apesar de todas as tecnologias disponíveis para gerenciamento de armazenamento, os desenvolvedores de aplicativos normalmente têm muito trabalho para executar operações de banco de dados com eficiência.

Geralmente, os desenvolvedores Java usam muito código ou usam um framework proprietário para interagir com o banco de dados, enquanto que, usando o JPA, a carga de interação com o banco de dados é reduzida significativamente. Ele forma uma ponte entre modelos de objetos (programa Java) e modelos relacionais (programa de banco de dados).

Java Persistence API é uma coleção de classes e métodos para armazenar persistentemente grandes quantidades de dados em um banco de dados com um mínimo de esforço do desenvolvedor.

Adicionando Dependências ao `pom.xml` de Nossa Aplicação

Vamos incluir as dependências `spring-boot-starter-data-jpa` e `h2` em nosso `pom.xml`:

- `pom.xml`

```

<!-- Código anterior omitido -->
<dependencies>

    <!-- Dependências atuais omitidas -->

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!--
        Este escopo indica que a dependência não é necessária para compilação, mas para execução.
        Ele está no classpath do runtime de execução e teste, mas não no classpath.
    -->

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
<!-- Código posterior omitido -->

```

Transformando a classe Livro em uma Entidade

Agora, vamos transformar a classe `Livro` em uma entidade que pode ser persistida em um banco de dados utilizando JPA:

- `/src/main/java/com/acme/livroservice/Livro.java`

```

package com.acme.livroservice;

// Novidades aqui
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

// Novidades aqui
@Entity
public class Livro {

    // Novidades aqui
    @Id @GeneratedValue Long id;

    private String autor;
    private String titulo;
    private Double preco;

    public Livro(String autor, String titulo, Double preco) {
        super();
        this.autor = autor;
        this.titulo = titulo;
        this.preco = preco;
    }

    // Código atual omitido
}

```

- `@Entity` é uma anotação JPA para tornar esse objeto pronto para armazenamento em um banco de dados baseado em JPA;
- `id` foi marcado com anotações de JPA para indicar que é a chave primária e preenchida automaticamente pelo provedor de JPA;
- Um construtor personalizado é criado quando precisamos criar uma nova instância, mas ainda não temos um id;

Com essa definição de objeto de domínio, agora podemos recorrer ao Spring Data JPA para lidar com as interações tediosas do banco de dados. Os repositórios do Spring Data são interfaces com métodos que suportam leitura, atualização, exclusão e criação de registros em um armazenamento de dados de backend. Alguns repositórios também suportam paginação de dados e ordenação, quando apropriado. O Spring Data sintetiza implementações baseadas em convenções encontradas na nomenclatura dos métodos na interface.

Um repositório de dados JPA

Vamos então criar o repositório JPA que lidará com as operações do banco de dados:

- /src/main/java/com/acme/livroservice/LivroRepository.java

```
package com.acme.livroservice;

import org.springframework.data.jpa.repository.JpaRepository;

public interface LivroRepository extends JpaRepository<Livro, Long> {
```

Essa interface estende o `JpaRepository` do Spring Data JPA, especificando o tipo de domínio como `Livro` e o tipo de id como `Long`. Esta interface, embora vazia na superfície, embute vários recursos:

- Criação de novas instâncias;
- Atualização das existentes;
- Exclusão;
- Pesquisa (um, todos, por propriedades simples ou complexas);

A solução de repositório do Spring Data possibilita contornar os detalhes do armazenamento de dados e, em vez disso, soluciona a maioria dos problemas usando a terminologia específica do domínio.

Pré-carregando com dados fictícios

Atualmente estamos criando um bean e o carregando com dados fictícios, vamos fazer a mesma coisa com o banco de dados H2, no entanto, o Spring fornece um mecanismo adequado para esta finalidade.

Para isso vamos criar a classe `LoadDatabase`:

- /src/main/java/com/acme/livroservice/LoadDatabase.java

```

package com.acme.livroservice;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
class LoadDatabase {

    Logger logger = LoggerFactory.getLogger(LivrosController.class);

    @Bean
    public CommandLineRunner initDatabase(LivroRepository repository) {
        return args -> {
            logger.info("Preloading " + repository.save(new Livro("Miguel de Cervantes", "Don Quixote", 144.0)));
            logger.info("Preloading " + repository.save(new Livro("J. R. R. Tolkien", "O Senhor dos Anéis", 123.0)));
            logger.info("Preloading "
                    + repository.save(new Livro("Antoine de Saint-Exupéry", "O Pequeno Príncipe", 152.0)));
            logger.info(
                    "Preloading " + repository.save(new Livro("Charles Dickens", "Um Conto de Duas Cidades", 35.0)));
        };
    }
}

```

O que acontece quando o projeto é executado?

O Spring Boot executará todos os beans `CommandLineRunner` quando o contexto do aplicativo for carregado.

Este runner solicitará uma cópia do `EmployeeRepository` que você acabou de criar.

Ao usá-lo, ele criará duas entidades e as armazenará.

Ao consultar o log, você verá algo como:

```

...
2019-01-20 19:58:12.681 INFO 24752 --- [ restartedMain] com.acme.livroservice.LivrosController : Preloading Livro [id=1, a
utor=Miguel de Cervantes, titulo=Don Quixote, preco=144.0]
2019-01-20 19:58:12.684 INFO 24752 --- [ restartedMain] com.acme.livroservice.LivrosController : Preloading Livro [id=2, a
utor=J. R. R. Tolkien, titulo=O Senhor dos Anéis, preco=123.0]
2019-01-20 19:58:12.688 INFO 24752 --- [ restartedMain] com.acme.livroservice.LivrosController : Preloading Livro [id=3, a
utor=Antoine de Saint-Exupéry, titulo=O Pequeno Príncipe, preco=152.0]
2019-01-20 19:58:12.693 INFO 24752 --- [ restartedMain] com.acme.livroservice.LivrosController : Preloading Livro [id=4, a
utor=Charles Dickens, titulo=Um Conto de Duas Cidades, preco=35.0]
...

```

Ajustando o Controller para que use o Repositório

Agora que temos um repositório funcional, devemos apagar o bean `listaLivros` que criamos na classe `LivroServiceApplication`:

- src/main/java/com/acme/livroservice/LivroServiceApplication.java

```

package com.acme.livroservice;

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

@SpringBootApplication
public class LivroServiceApplication {

    // O código que fabricava o bean foi removido

    public static void main(String[] args) {
        SpringApplication.run(LivroServiceApplication.class, args);
    }
}

```

Agora vamos ajustar o controller de fato:

- src/main/java/com/acme/livroservice/LivrosController.java

```

package com.acme.livroservice;

import java.util.List;
import java.util.Optional;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.server.ResponseStatusException;

@RestController
@RequestMapping("/livros")
public class LivrosController {

    Logger logger = LoggerFactory.getLogger(LivrosController.class);

    private final LivroRepository repository;

    LivrosController(LivroRepository repository) {
        this.repository = repository;
    }

    @GetMapping
    public List<Livro> getLivros(@RequestParam("autor") Optional<String> autor,
                                  @RequestParam("titulo") Optional<String> titulo) {
        logger.info("getLivros - autor: " + autor.orElse("Não informado") + " titulo: " + titulo.orElse("Não informado"));

        return repository.findAll();
    }

    @GetMapping("/{id}")
    public Livro getLivroPorId(@PathVariable Long id) {
        logger.info("getLivroPorId: " + id);
        return repository.findById(id).orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Livro adicionarLivro(@RequestBody Livro livro) {
        logger.info("adicionarLivro: " + livro);
    }
}

```

```

        return repository.save(livro);
    }

    @PutMapping("/{id}")
    public Livro atualizarLivro(@RequestBody Livro livro, @PathVariable Long id) {
        logger.info("atualizarLivro: " + livro + " id: " + id);
        return repository.findById(id)
            .map(livroSalvo -> {
                livroSalvo.setAutor(livro.getAutor());
                livroSalvo.setTitulo(livro.getTitulo());
                livroSalvo.setPreco(livro.getPreco());
                return repository.save(livroSalvo);
            })
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void excluirLivro(@PathVariable Long id) {
        logger.info("excluirLivro: " + id);
        repository.deleteById(id);
    }
}

```

Note que `LivroRepository` é injetado pelo construtor no controlador.

Tudo deve funcionar normalmente, porém, irá perceber que não conseguimos mais buscar livros pelo autor e título, iremos providenciar este último ajuste agora.

Criteria Queries

O Spring Data JPA fornece várias maneiras de lidar com entidades, incluindo métodos de consulta e consultas personalizadas de JPQL. No entanto, às vezes precisamos de uma abordagem mais programática: por exemplo, Criteria API ou QueryDSL.

A Criteria API oferece uma maneira programática de criar consultas digitadas, o que nos ajuda a evitar erros de sintaxe. Ainda mais, quando o usamos com a API Metamodel, ele faz verificações em tempo de compilação se usamos os nomes e tipos de campos corretos.

No entanto, tem suas desvantagens: temos que escrever uma lógica detalhada com código verboso.

Usando JPA Specifications

O Spring Data introduziu a interface `org.springframework.data.jpa.domain.Specification` para encapsular um único predicado, podemos fornecer métodos para criar instâncias de especificação e então usá-los. Para isso, precisamos que o nosso repositório estenda `org.springframework.data.jpa.repository.JpaRepositorySpecificationExecutor`. Essa interface declara métodos úteis para trabalhar com especificações.

Infelizmente, não há nenhum método, ao qual podemos passar várias especificações como argumentos. Em vez disso, obtemos métodos de utilitário na interface `org.springframework.data.jpa.domain.Specification`. Por exemplo, combinando duas instâncias de especificação com lógica `and` e `where`.

Vamos então preparar nosso repositório para trabalhar com `Specifications`:

- `/src/main/java/com/acme/livroservice/LivroRepository.java`

```

package com.acme.livroservice;

import org.springframework.data.jpa.domain.Specification;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;

public interface LivroRepository extends JpaRepository<Livro, Long>, JpaSpecificationExecutor<Livro> {

    static Specification<Livro> autorContem(String autor) {
        return (livro, cq, cb) -> cb.like(cb.lower(livro.get("autor")), "%" + autor.toLowerCase() + "%");
    }

    static Specification<Livro> tituloContem(String titulo) {
        return (livro, cq, cb) -> cb.like(cb.lower(livro.get("titulo")), "%" + titulo.toLowerCase() + "%");
    }
}

```

Em seguida, vamos alterar nosso controller para que utilize as Specifications :

- src/main/java/com/acme/livroservice/LivrosController.java

```

package com.acme.livroservice;

// Código atual omitido

// Novidade aqui
import org.springframework.data.jpa.domain.Specification;

@RestController
@RequestMapping("/livros")
public class LivrosController {

    // Código atual omitido

    @GetMapping
    public List<Livro> getLivros(@RequestParam("autor") Optional<String> autor,
                                   @RequestParam("titulo") Optional<String> titulo) {
        logger.info(
            "getLivros - autor: " + autor.orElse("Não informado") + " titulo: " + titulo.orElse("Não informado"));

        // Novidades aqui
        if (autor.isPresent()) {
            return repository.findAll(LivroRepository.autorContem(autor.get()));
        } else if (titulo.isPresent()) {
            return repository.findAll(LivroRepository.tituloContem(titulo.get()));
        } else if (autor.isPresent() && titulo.isPresent()) {
            return repository.findAll(
                Specification.where(LivroRepository.autorContem(autor.get())).and(LivroRepository.tituloContem(titulo.get())));
        } else {
            return repository.findAll();
        }
    }

    // Código atual omitido
}

```

Console Web do H2

O banco de dados H2 fornece um console baseado em navegador que o Spring Boot pode configurar automaticamente para você.

Para acessá-lo, utilize o end-point `/h2-console`, os parâmetro de conexão são os seguintes:

- JDBC URL: `jdbc:h2:mem:testdb`
- User Name: `sa`

- **Password:** ``

Comunicação entre Microsserviços

RestTemplate

Nosso serviço de livros já está funcionando e podemos consumir seus dados utilizando um navegador por exemplo, porém uma outra maneira útil de consumir um serviço da web REST é programaticamente. Para ajudá-lo com essa tarefa, o Spring fornece uma classe conveniente chamada `RestTemplate`. O `RestTemplate` torna a interação com a maioria dos serviços RESTful um comando de uma linha. E pode até vincular esses dados a tipos de domínio personalizados.

Importando o Projeto de Avaliações

Para exemplificar a comunicação entre microsserviços, devemos primeiro importar o projeto `avaliacao-service`, este projeto já está configurado com um microsserviço com funcionalidades de CRUD de avaliações de livros:

<https://github.com/tiagolpadua/msc-avaliacao-service/archive/inicial.zip>

Importe o projeto no Spring Tools Suite, compile e execute. Teste seu funcionamento acessando:

<http://localhost:8081/avaliacoes>

Obtendo um livro a partir do microsserviço de avaliações

Podemos agora testar a inclusão de uma avaliação utilizando o RESTClient, para isso, realize operações do tipo POST contra o end-point <http://localhost:8081/avaliacoes> passando no corpo da requisição o seguinte JSON:

```
{
  "livroId": 2,
  "nota": 1
}
```

Em seguida, liste as avaliações pela URL <http://localhost:8081/avaliacoes> e verifique que a avaliação foi corretamente inserida.

Agora, tente inserir uma nova avaliação, no entanto, informando um ID de um livro inexistente:

```
{
  "livroId": 999,
  "nota": 1
}
```

A inclusão da avaliação deverá ser bem sucedida, mas isto é um problema, pois não existe livro com o ID especificado. Nossa demanda agora é ajustar o serviço de inclusão de avaliações para que verifique se o livro realmente existe antes de proceder a inclusão da avaliação.

Obtendo um JSON simples

Vamos começar de forma simples e realizar uma solicitação GET com um exemplo rápido usando a API

`getForEntity()` :

AvaliacoesController

```

@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Avaliacao adicionarAvaliacao(@RequestBody Avaliacao avaliacao) {
    logger.info("adicionarAvaliacao: " + avaliacao);

    RestTemplate restTemplate = new RestTemplate();
    String livroResourceUrl = "http://localhost:8080/livros/";
    ResponseEntity<String> response = restTemplate.getForEntity(livroResourceUrl + avaliacao.getLivroId(), String.class);

    logger.info("response.getBody(): " + response.getBody());

    return repository.save(avaliacao);
}

```

Se observarmos o Log, devemos ver algo como:

```

2019-01-29 16:28:07.187 INFO 2252 --- [nio-8081-exec-1] o.a.c.c.C.[Tomcat-3].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2019-01-29 16:28:07.188 INFO 2252 --- [nio-8081-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2019-01-29 16:28:07.192 INFO 2252 --- [nio-8081-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 4 ms
2019-01-29 16:28:07.195 INFO 2252 --- [nio-8081-exec-1] c.a.a.AvaliacoesController : adicionarAvaliacao: Avaliacao [id=null, livroId=1, nota=1]
2019-01-29 16:28:07.216 INFO 2252 --- [nio-8081-exec-1] c.a.a.AvaliacoesController : response.getBody(): {"id": 1,"autor": "Miguel de Cervantes", "titulo": "Don Quixote", "preco": 144.0}
Hibernate:
    call next value for hibernate_sequence
Hibernate:
    insert
    into
        avaliacao
        (livro_id, nota, id)
    values
        (?, ?, ?)

```

Note que os dados do autor do livro foram retornados na forma de uma string JSON.

Observe que temos acesso total à resposta HTTP - para que possamos fazer coisas como verificar o código de status para garantir que a operação seja realmente bem-sucedida ou trabalhar com o corpo real da resposta:

```

@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Avaliacao adicionarAvaliacao(@RequestBody Avaliacao avaliacao) {
    logger.info("adicionarAvaliacao: " + avaliacao);

    RestTemplate restTemplate = new RestTemplate();
    String livroResourceUrl = "http://localhost:8080/livros/";
    ResponseEntity<String> response = restTemplate.getForEntity(livroResourceUrl + avaliacao.getLivroId(), String.class);

    logger.info("response.getBody(): " + response.getBody());

    // Novidade aqui
    ObjectMapper mapper = new ObjectMapper();
    JsonNode root = mapper.readTree(response.getBody());
    JsonNode autor = root.path("autor");

    logger.info("autor: " + autor);

    return repository.save(avaliacao);
}

```

O log deverá conter uma saída como:

```

2019-01-29 16:33:35.485 INFO 2252 --- [io-8081-exec-10] c.a.a.AvaliacoesController : autor: "Miguel de Cervantes"

```

Estamos trabalhando com o corpo da resposta como uma String padrão aqui - e usando Jackson (e a estrutura de nó JSON que Jackson fornece) para verificar alguns detalhes.

Recuperando POJO em vez de JSON

Também podemos mapear a resposta diretamente para um Resource DTO, para fazer isso, primeiro (por simplicidade) vamos copiar a classe `Livro` para o projeto de avaliação, e em seguida:

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Avaliacao adicionarAvaliacao(@RequestBody Avaliacao avaliacao) throws IOException {
    logger.info("adicionarAvaliacao: " + avaliacao);

    RestTemplate restTemplate = new RestTemplate();
    String livroResourceUrl = "http://localhost:8080/livros/";

    ResponseEntity<Livro> responseLivro = restTemplate.getEntity(livroResourceUrl + avaliacao.getId(), Livro.class);

    logger.info("responseLivro.getBody(): " + responseLivro.getBody());

    return repository.save(avaliacao);
}
```

Validando se o livro existe

Vamos testar o que ocorrem em dois cenários:

1. O ID livro não existe;
2. O serviço de livros está indisponível;

Para validarmos se o livro de fato existe podemos simplesmente acionar o método e ver se ocorre uma exception:

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Avaliacao adicionarAvaliacao(@RequestBody Avaliacao avaliacao) throws IOException {
    logger.info("adicionarAvaliacao: " + avaliacao);

    RestTemplate restTemplate = new RestTemplate();
    String livroResourceUrl = "http://localhost:8080/livros/";

    // Novidades aqui
    try {
        ResponseEntity<Livro> responseLivro = restTemplate.getEntity(livroResourceUrl + avaliacao.getId(),
            Livro.class);
        logger.info("Livro " + responseLivro.getBody().getTitulo() + " localizado");
    } catch (HttpClientErrorException ex) {
        logger.error("Ocorreu um erro na comunicação com o serviço de livros", ex);
        if (ex.getRawStatusCode() == HttpStatus.NOT_FOUND.value()) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST,
                "Livro vinculado a avaliação não foi encontrado.");
        } else {
            throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE,
                "Ocorreu um erro não esperado na comunicação com o serviço de livros: " + ex.getMessage());
        }
    } catch (ResourceAccessException ex) {
        logger.error("Ocorreu um erro na comunicação com o serviço de livros", ex);
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE,
            "Ocorreu um erro não esperado na comunicação com o serviço de livros: " + ex.getMessage());
    }

    return repository.save(avaliacao);
}
```

Adicionando um timeout

Podemos configurar o RestTemplate para o tempo limite usando ClientHttpBuilderFactory, observe que existem vários tipos de timeout a serem configurados:

```
private ClientHttpBuilderFactory getClientHttpBuilderFactory() {
    int timeout = 10000;
    HttpComponentsClientHttpBuilderFactory clientHttpBuilderFactory
        = new HttpComponentsClientHttpBuilderFactory();
    clientHttpBuilderFactory.setConnectTimeout(timeout);
    clientHttpBuilderFactory.setConnectionRequestTimeout(timeout);
    clientHttpBuilderFactory.setReadTimeout(timeout);
    return clientHttpBuilderFactory;
}

@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Avaliacao adicionarAvaliacao(@RequestBody Avaliacao avaliacao) throws IOException {
    logger.info("adicionarAvaliacao: " + avaliacao);

    // Novidade aqui
    RestTemplate restTemplate = new RestTemplate(getClientHttpBuilderFactory());

    // Código atual omitido
}
```

É necessário também incluir uma nova dependência:

pom.xml

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
</dependency>
```

Para testarmos esta funcionalidade, vamos ajustar `LivrosController` para que atrasse a resposta:

LivrosController

```
@GetMapping("/{id}")
public Livro getLivroPorId(@PathVariable Long id) {
    logger.info("getLivroPorId: " + id);

    // Novidade aqui
    try {
        Thread.sleep(5000);
    } catch (InterruptedException ex) {
        Thread.currentThread().interrupt();
    }

    return repository.findById(id)
        .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
}
```

Já temos o serviço de avaliações corretamente configurado para validar a existência dos livros antes de incluir uma avaliação. Agora, temos que realizar a operação inversa, caso um livro seja excluído, é necessário que as avaliações a ele relacionadas também sejam excluídas.

O primeiro passo é ajustar o repositório de avaliações para incluir a função que fará a exclusão:

AvaliacaoRepository

```

package com.acme.avaliacaoservice;

import javax.transaction.Transactional;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;

public interface AvaliacaoRepository extends JpaRepository<Avaliacao, Long>, JpaSpecificationExecutor<Avaliacao> {
    @Transactional
    @Modifying
    @Query("delete from Avaliacao a where livroId = ?1")
    void deleteAvaliacaoPorLivroId(Long livroId);
}

```

Agora, vamos criar um end-point REST que irá proceder de fato a exclusão:

AvaliacoesController

```

@DeleteMapping("/livro/{livroId}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteAvaliacaoPorLivroId(@PathVariable Long livroId) {
    logger.info("deleteAvaliacaoPorLivroId: " + livroId);
    repository.deleteAvaliacaoPorLivroId(livroId);
}

```

Ótimo, agora é a hora de ajustarmos o end-point do serviço de exclusão de livros para acionar o serviço de exclusão de avaliações:

LivrosController

```

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void excluirLivro(@PathVariable Long id) {
    logger.info("excluirLivro: " + id);

    RestTemplate restTemplate = new RestTemplate();
    String avaliacaoResourceUrl = "http://localhost:8081/avaliacoes/livro/";

    try {
        restTemplate.delete(avaliacaoResourceUrl + id);
        logger.info("Avaliações vinculadas excluídas com sucesso");
    } catch (ResourceAccessException | HttpClientErrorException ex) {
        logger.error("Ocorreu um erro na comunicação com o serviço de avaliações", ex);
        throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE,
            "Ocorreu um erro não esperado na comunicação com o serviço de livros: " + ex.getMessage());
    }

    repository.deleteById(id);
}

```

Tudo está funcionando agora, ao excluir um livro, as avaliações relacionadas àquele livro também são excluídas, mas se paramos para pensar um pouco veremos que ainda existem algumas "pontas soltas":

- O serviço de exclusão de livros aguarda que as avaliações sejam excluídas para então proceder a exclusão do livro, ou seja, é uma operação síncrona, será que isso é realmente necessário?
- Caso a operação de exclusão de avaliações esteja indisponível, a exclusão de livros também ficará, será que realmente é necessário?

RabbitMQ

RabbitMQ é um servidor de mensageria de código aberto (open source) desenvolvido em Erlang, implementado para suportar mensagens em um protocolo denominado *Advanced Message Queuing Protocol (AMQP)*. Ele possibilita lidar com o tráfego de mensagens de forma rápida e confiável, além de ser compatível com diversas linguagens de programação, possuir interface de administração nativa e ser multiplataforma.

Dentre as aplicabilidades do RabbitMQ estão possibilitar a garantia de assincronicidade entre aplicações, diminuir o acoplamento entre aplicações, distribuir alertas, controlar fila de trabalhos em background.

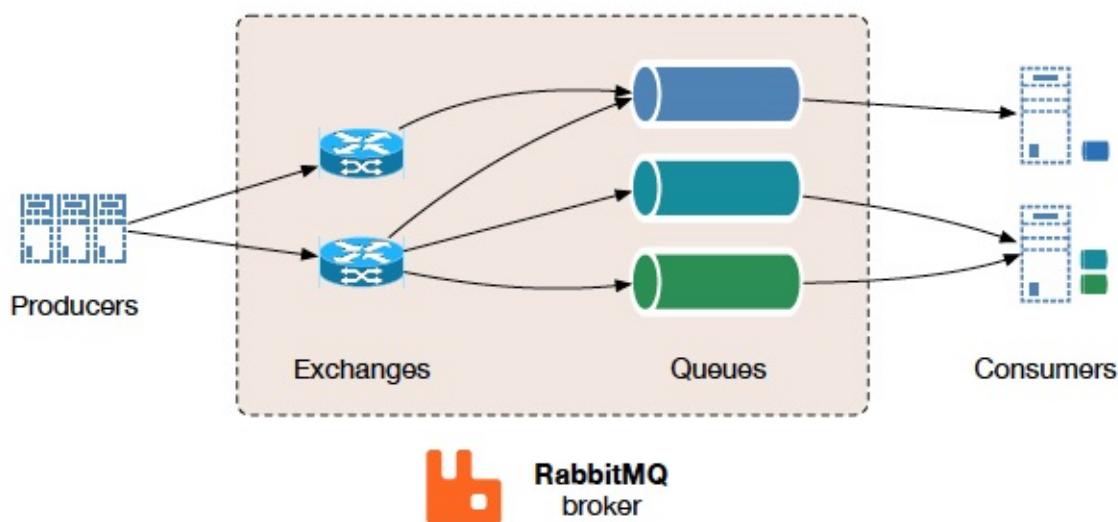
Outras características do RabbitMQ:

- É desenvolvido em Erlang;
- É considerado rápido e confiável;
- Compatível com os principais sistemas operacionais;
- Suporta diversas plataformas de desenvolvimento. Bibliotecas de conexão com o RabbitMQ estão disponíveis em diversas linguagens de programação;

Visão Geral

O RabbitMQ é um *message broker*: recebe e encaminha mensagens. Você pode pensar como uma agência dos correios: quando você coloca a carta que deseja postar em uma caixa postal, pode ter certeza de que o carteiro entregará a correspondência ao seu destinatário. Nessa analogia, o RabbitMQ é uma caixa postal, um correio e um carteiro.

A principal diferença entre o RabbitMQ e os correios é que ele não lida com papel, ao invés disso ele aceita, armazena e envia dados binários - mensagens.



Fontes

- <https://www.rabbitmq.com/tutorials/amqp-concepts.html>
- <https://www.rabbitmq.com/tutorials/tutorial-one-java.html>
- <https://spring.io/guides/gs/messaging-rabbitmq/>

Principais Conceitos

O que é o AMQP 0-9-1?

O AMQP 0-9-1 (*Advanced Message Queuing Protocol*) é um protocolo de mensagens que permite que aplicativos clientes se comuniquem com brokers de mensagens.

O papel dos brokers

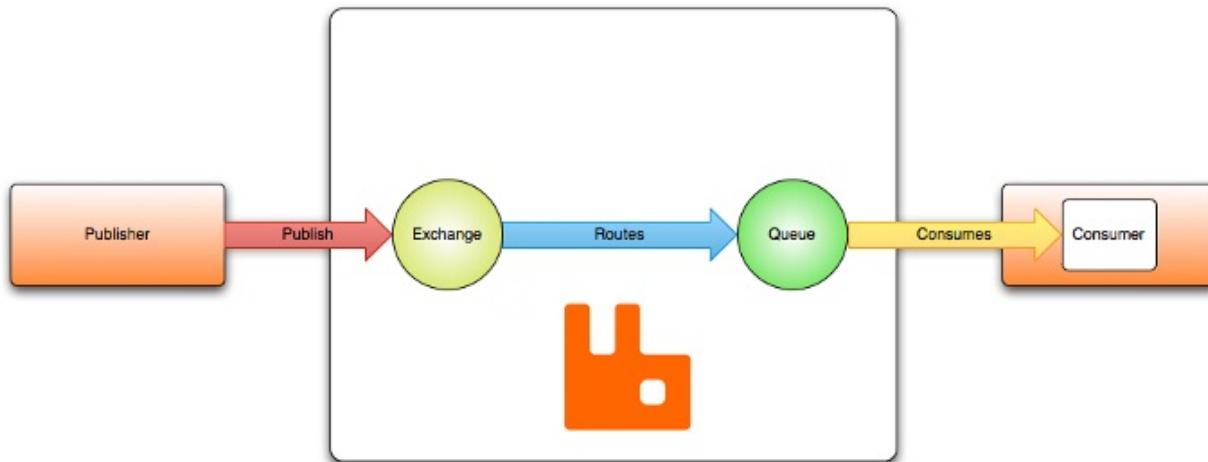
Os brokers de mensagens recebem mensagens de *publishers* (aplicativos que as publicam, também conhecidos como *producers*) e os encaminham aos *consumers* (aplicativos que as processam).

Como é um protocolo de rede, os *publishers*, os *consumers* e o *broker* podem residir em máquinas diferentes.

O modelo AMQP 0-9-1 em resumo

O Modelo AMQP 0-9-1 tem a seguinte visão de mundo: as mensagens são publicadas em *exchanges*, que são frequentemente comparadas a agências postais ou caixas de correio. *Exchanges* então distribuem cópias de mensagens para filas (*queues*) usando regras chamadas *bindings*. Em seguida, os brokers do AMQP entregam mensagens aos consumidores inscritos em filas ou os consumidores buscam/extraem mensagens das filas sob demanda.

"Hello, world" example routing



Ao publicar uma mensagem, os publicadores podem especificar vários atributos de mensagem (meta-dados da mensagem). Alguns desses metadados podem ser usados pelo broker, no entanto, o restante é completamente opaco para o broker e é usado apenas pelos aplicativos que recebem a mensagem.

As redes não são confiáveis e os aplicativos podem falhar ao processar mensagens, portanto, o modelo AMQP tem uma noção de confirmações de mensagem (*acknowledgements*): quando uma mensagem é entregue a um consumidor, o consumidor notifica o broker automaticamente ou assim que o desenvolvedor do aplicativo optar por fazê-lo. Quando confirmações de mensagem são utilizadas, um broker somente removerá completamente uma mensagem de uma fila quando receber uma notificação para essa mensagem (ou grupo de mensagens).

Em determinadas situações, por exemplo, quando uma mensagem não pode ser roteada, as mensagens podem ser devolvidas aos editores, eliminadas ou, se o intermediário *broker* tiver uma extensão, colocadas em uma chamada "fila de devoluções". Os *publishers* escolhem como lidar com situações como essa publicando mensagens usando determinados parâmetros.

Filas, *exchanges* e *bindings* são coletivamente referidas como entidades AMQP.

O AMQP é um protocolo programável

O AMQP 0-9-1 é um protocolo programável no sentido de que as entidades e os esquemas de roteamento do AMQP 0-9-1 são definidos principalmente pelos próprios aplicativos, não pelo administrador do *broker*. Consequentemente, a provisão é feita para operações de protocolo que declaram filas e *exchanges*, definem *bindings* entre elas, assinam filas e assim por diante.

Isso dá aos desenvolvedores de aplicativos muita liberdade, mas também exige que eles estejam cientes dos possíveis conflitos de definição. Na prática, os conflitos de definição são raros e geralmente indicam um erro de configuração.

As aplicações declaram as entidades do AMQP 0-9-1 de que precisam, definem os esquemas de roteamento necessários e podem optar por excluir as entidades do AMQP 0-9-1 quando elas não são mais usadas.

Exchanges e tipos de Exchange

As *exchanges* são entidades AMQP nas quais as mensagens são enviadas. As *exchanges* levam uma mensagem e encaminham para zero ou mais filas. O algoritmo de roteamento usado depende do tipo de *exchange* e das regras chamadas de *bindings*. Brokers AMQP 0-9-1 fornecem quatro tipos de *exchange*:

Nome	Nomes pré-declarados padrão
Direct exchange	(String vazia) e amq.direct
Fanout exchange	amq.fanout
Topic exchange	amq.topic
Headers exchange	amq.match (e amq.headers no RabbitMQ)

Além do tipo de *exchange*, as *exchanges* são declaradas com um número de atributos, sendo os mais importantes:

- Nome
- Durabilidade (se as *exchange* sobrevivem ao reinício do *broker*)
- Exclusão automática (se a *exchange* é excluída quando a última fila é desvinculada dela)
- Argumentos (opcional, usados por plug-ins e recursos específicos do *broker*)

As *exchanges* podem ser duradouras ou transitórias. As *exchanges* duráveis sobrevivem ao reinício do *broker* enquanto as *exchange* transientes não (elas precisam ser redeclaradas quando o *broker* volta a ficar on-line). Nem todos os cenários e casos de uso exigem que as *exchanges* sejam duráveis.

Exchange Padrão

A *exchange* padrão é uma *exchange* direta sem nome (string vazia) pré-declarada pelo *broker*. Ela tem uma propriedade especial que a torna muito útil para aplicativos simples: cada fila criada é automaticamente vinculada a ela com uma chave de roteamento (*routing key*) que é igual ao nome da fila.

Por exemplo, quando você declara uma fila com o nome "*search-indexing-online*", o *broker* AMQP 0-9-1 irá vincular-la à *exchange* padrão usando "*search-indexing-online*" como a chave de roteamento. Portanto, uma mensagem publicada na *exchange* padrão com a chave de roteamento "*search-indexing-online*" será roteada para a fila "*search-indexing-*

online". Em outras palavras, a *exchange* padrão faz parecer que é possível entregar mensagens diretamente para as filas, mesmo que isso não seja tecnicamente o que está acontecendo.

Direct Exchange

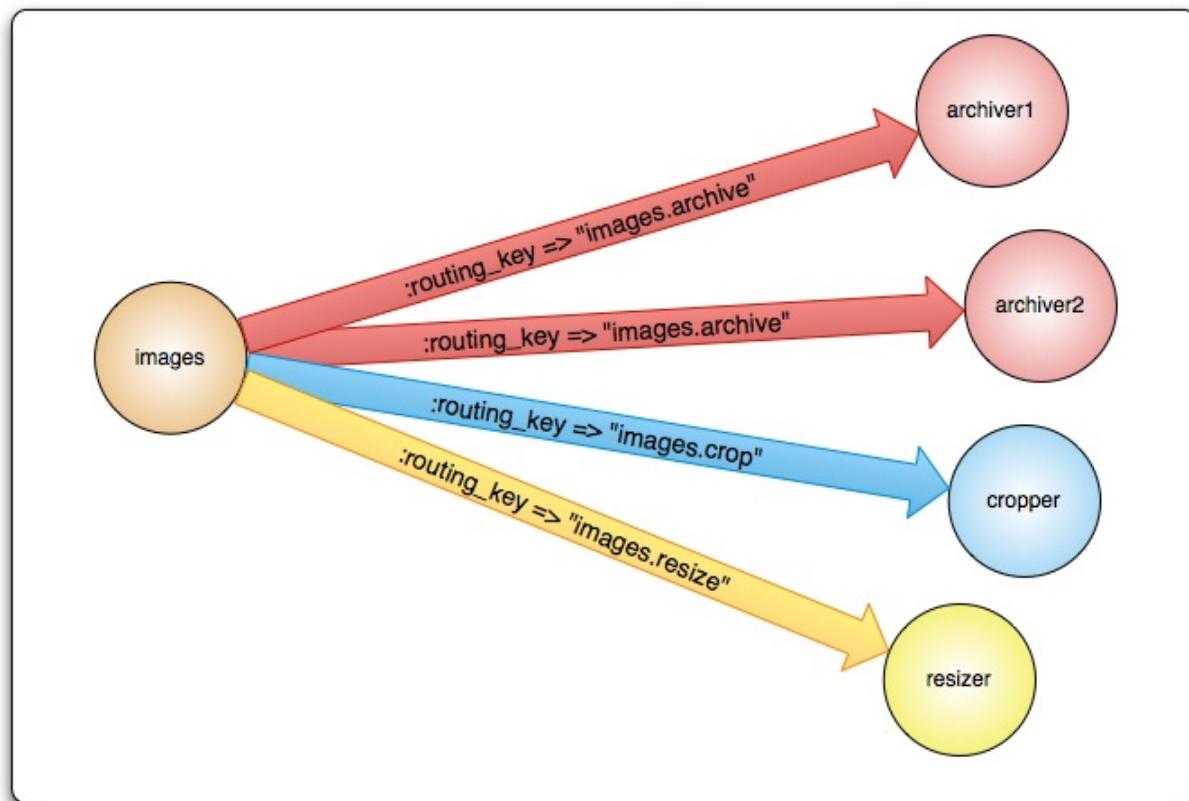
Uma *Direct Exchange* envia mensagens para filas com base na chave de roteamento de mensagens. Uma *Direct Exchange* é ideal para o roteamento *unicast* de mensagens (embora elas também possam ser usadas para roteamento *multicast*). Como isso funciona:

- Uma fila liga-se à *exchange* com uma chave de roteamento K
- Quando uma nova mensagem com a chave de roteamento R chega à troca direta, a troca a encaminha para a fila se K = R

As *Direct Exchanges* costumam ser usadas para distribuir tarefas entre vários trabalhadores (instâncias do mesmo aplicativo) de maneira rotativa. Ao fazer isso, é importante entender que, no AMQP 0-9-1, as mensagens são平衡adas por carga entre os consumidores e não entre as filas.

Uma *Direct Exchange* pode ser representada graficamente da seguinte maneira:

Direct exchange routing



Fanout Exchange

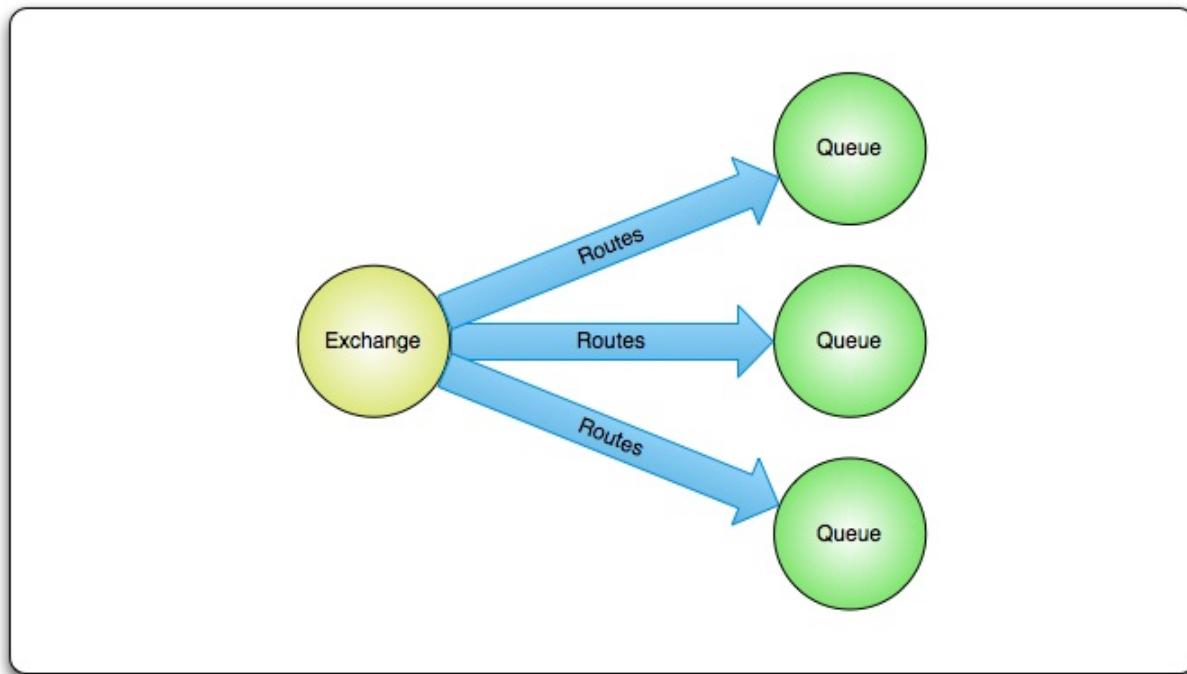
Uma *Fanout Exchange* encaminha mensagens para todas as filas que estão vinculadas a ela e a chave de roteamento é ignorada. Se N filas estiverem vinculadas a uma *Fanout Exchange*, quando uma nova mensagem for publicada nessa *exchange*, uma cópia da mensagem será entregue a todas as filas de N. As *Fanout Exchange* são ideais para o roteamento de transmissão de mensagens.

Como uma *Fanout Exchange* fornece uma cópia de uma mensagem a cada fila vinculada a ela, seus casos de uso são bastante semelhantes:

- Os jogos on-line massivos multi-jogador (MMO) podem usá-la para atualizações de leaderboard ou outros eventos globais
- Os sites de notícias esportivas podem usar *Fanout Exchange* para distribuir atualizações de pontuação para clientes móveis quase em tempo real
- Sistemas distribuídos podem transmitir várias atualizações de estado e configuração. Os bate-papos de grupo podem distribuir mensagens entre os participantes usando uma *Fanout Exchange*

Uma troca de fanout pode ser representada graficamente da seguinte forma:

Fanout exchange routing



Topic Exchange

As *Topic Exchanges* encaminham mensagens para uma ou várias filas com base na correspondência entre uma chave de roteamento de mensagem e o padrão que foi usado para ligar uma fila a uma *exchange*. *Topic Exchange* é geralmente usada para implementar várias variações de padrões de *publish/subscribe*. *Topic Exchanges* são comumente usadas para o roteamento *multicast* de mensagens.

As *Topic Exchanges* têm um conjunto muito amplo de casos de uso. Sempre que um problema envolve vários consumidores/aplicativos que selecionam seletivamente o tipo de mensagens que desejam receber, o uso de *Topic Exchange* deve ser considerado.

Exemplos de uso:

- Distribuir dados relevantes para uma localização geográfica específica, por exemplo, pontos de venda
- Processamento de tarefas em segundo plano feito por vários trabalhadores, cada um capaz de lidar com um conjunto específico de tarefas
- Atualizações de preço de ações (e atualizações sobre outros tipos de dados financeiros)
- Atualizações de notícias que envolvem categorização ou marcação (por exemplo, apenas para um determinado esporte ou equipe)

- Orquestração de serviços de diferentes tipos na nuvem
- Arquitetura distribuída / software específico do sistema operacional ou empacotamento em que cada construtor pode manipular apenas uma arquitetura ou sistema operacional

Headers Exchange

Uma *Headers Exchange* é projetada para roteamento em vários atributos que são mais facilmente expressos como cabeçalhos de mensagens do que uma chave de roteamento. *Headers Exchange* ignoram o atributo de chave de roteamento. Em vez disso, os atributos usados para roteamento são obtidos do atributo *headers*. Uma mensagem é considerada correspondente se o valor do cabeçalho *for* igual ao valor especificado no *binding*.

É possível vincular uma fila a uma troca de cabeçalhos usando mais de um cabeçalho para correspondência. Nesse caso, o *broker* precisa de mais uma informação do desenvolvedor do aplicativo, ou seja, deve considerar as mensagens com algum dos cabeçalhos correspondentes ou todas elas? É para isso que serve o argumento de ligação "x-match". Quando o argumento "x-match" é definido como "any", apenas um valor de cabeçalho correspondente é suficiente. Alternativamente, definindo "x-match" para "todos" determina que todos os valores devem corresponder.

Headers Exchange podem ser encaradas como "trocas diretas turbinadas". Como elas são roteadas com base em valores de cabeçalho, elas podem ser usadas como *direct exchanges*, em que a chave de roteamento não precisa ser uma string; poderia ser um inteiro ou um hash (dicionário) por exemplo.

Filas (Queues)

As filas no modelo AMQP 0-9-1 são muito semelhantes às filas em outros sistemas de enfileiramento de mensagens e tarefas: elas armazenam mensagens que são consumidas pelos aplicativos. As filas compartilham algumas propriedades com *exchanges*, mas também possuem algumas propriedades adicionais:

- Nome
- Durável (se a fila sobreviverá a uma reinicialização do *broker*)
- Exclusivo (usado por apenas uma conexão e a fila será excluída quando essa conexão for fechada)
- Exclusão automática (a fila que teve pelo menos um consumidor é excluída quando o último consumidor cancela a inscrição)
- Argumentos (opcional; usado por plugins e recursos específicos do *broker*, como mensagem TTL, limite de tamanho da fila, etc)

Antes que uma fila possa ser usada, ela deve ser declarada. Declarar uma fila fará com que ela seja criada se ainda não existir. A declaração não terá efeito se a fila já existir e seus atributos forem os mesmos da declaração. Quando os atributos de fila existentes não são os mesmos da declaração, uma exceção de nível de canal com o código 406 (`PRECONDITION_FAILED`) será lançada.

Nomes de Filas

Os aplicativos podem escolher nomes de filas ou solicitar que o *broker* crie um nome para eles. Os nomes das filas podem ter até 255 bytes de caracteres UTF-8. Um *broker* AMQP 0-9-1 pode gerar um nome de fila exclusivo em nome de um aplicativo. Para usar esse recurso, passe uma string vazia como o argumento do nome da fila. O nome gerado será retornado ao cliente com a resposta da declaração de fila.

Nomes de filas começando com "amq." são reservados para uso interno pelo *broker*. As tentativas de declarar uma fila com um nome que viole essa regra resultarão em uma exceção no nível do canal com o código de resposta 403 (`ACCESS_REFUSED`).

Durabilidade da fila

As filas duráveis são mantidas em disco e, assim, sobrevivem às reinicializações do *broker*. Filas que não são duráveis são chamadas de transitórias. Nem todos os cenários e casos de uso obrigam as filas a serem duráveis.

Adurabilidade de uma fila não torna as mensagens encaminhadas para essa fila duráveis. Se o *broker* sair do ar e, em seguida, retornado, a fila durável será declarada novamente durante a inicialização do *broker*, no entanto, somente as mensagens persistentes serão recuperadas.

Bindings (Ligações)

Os *bindings* são regras que as *exchanges* usam (entre outras coisas) para rotear mensagens para filas. Para instruir uma *exchange* E para rotear mensagens para uma fila Q, Q tem que ser ligado a E. As *bindings* podem ter um atributo de chave de roteamento opcional usado por alguns tipos de *exchange*. O objetivo da chave de roteamento é selecionar determinadas mensagens publicadas em uma *exchange* para serem roteadas para a fila de ligação. Em outras palavras, a chave de roteamento age como um filtro.

Uma analogia para demonstrar:

- Fila é como o seu destino na cidade de Nova York
- *Exchange* é como o aeroporto JFK
- *Bindings* são rotas de JFK para o seu destino. Pode haver zero ou muitas maneiras de alcançá-lo

Ter essa camada de indireção permite cenários de roteamento que são impossíveis ou muito difíceis de implementar usando a publicação diretamente em filas e também elimina certa quantidade de trabalho duplicado que os desenvolvedores de aplicativos precisam fazer.

Se a mensagem AMQP não puder ser roteada para qualquer fila (por exemplo, porque não há ligações para a *exchange* para a qual ela foi publicada), ela será eliminada ou retornada ao publicador, dependendo dos atributos da mensagem que o publicador configurou.

Consumidores

Armazenar mensagens em filas é inútil, a menos que os aplicativos possam consumi-las. No Modelo AMQP 0-9-1, existem duas maneiras de os aplicativos fazerem isso:

- Receber mensagens entregues a eles ("push API")
- Buscar mensagens conforme necessário ("pull API")

Com a "API push", os aplicativos precisam indicar interesse em consumir mensagens de uma fila específica. Quando o fazem, dizemos que registram um consumidor ou, simplesmente, assinam uma fila. É possível ter mais de um consumidor por fila ou registrar um consumidor exclusivo (exclui todos os outros consumidores da fila enquanto está consumindo).

Cada consumidor (assinatura) possui um identificador chamado tag do consumidor. Pode ser usado para cancelar a assinatura de mensagens. Tags de consumidor são apenas strings.

Mensagem Acknowledgements

Aplicativos do consumidor - aplicativos que recebem e processam mensagens - ocasionalmente podem falhar ao processar mensagens individuais ou, às vezes, travar. Há também a possibilidade de problemas de rede causando problemas. Isso levanta uma questão: quando o broker AMQP deve remover mensagens de filas? A especificação AMQP 0-9-1 propõe duas opções:

- Depois que o *broker* envia uma mensagem para um aplicativo (usando os métodos AMQP `basic.deliver` ou `basic.get-ok`);

- Depois que o aplicativo envia de volta uma confirmação (usando o método AMQP `basic.ack`).

A primeira opção é chamada de modelo de reconhecimento automático, enquanto a segunda é chamada de modelo de reconhecimento explícito. Com o modelo explícito, o aplicativo escolhe quando é hora de enviar uma confirmação. Pode ser logo após o recebimento de uma mensagem ou após sua persistência em um armazenamento de dados antes do processamento ou após o processamento completo da mensagem (por exemplo, buscar com êxito uma página da Web, processá-la e armazená-la em algum armazenamento de dados persistente).

Se um consumidor morrer sem enviar uma confirmação, o intermediário AMQP o entregará a outro consumidor ou, se nenhum estiver disponível no momento, o agente aguardará até que pelo menos um consumidor seja registrado para a mesma fila antes de tentar a nova entrega.

Rejeitando Mensagens

Quando um aplicativo consumidor recebe uma mensagem, o processamento dessa mensagem pode ou não ter êxito. Um aplicativo pode indicar ao intermediário que o processamento da mensagem falhou (ou não pode ser realizado no momento) ao rejeitar uma mensagem. Ao rejeitar uma mensagem, um aplicativo pode solicitar ao *broker* que a descarte ou enfileire novamente. Quando houver apenas um consumidor em uma fila, certifique-se de não criar loops infinitos de entrega de mensagens rejeitando e enfileirando novamente uma mensagem do mesmo consumidor repetidas vezes.

Acknowledgements Negativos

As mensagens são rejeitadas com o método AMQP `basic.reject`. Há uma limitação que o `basic.reject` tem: não há como rejeitar mensagens múltiplas como você pode fazer com os *acknowledgements*. No entanto, se você estiver usando o RabbitMQ, haverá uma solução. O RabbitMQ fornece uma extensão AMQP 0-9-1, conhecida como *negative acknowledgements* ou *nacks*.

Prefetching de Mensagens

Para casos em que vários consumidores compartilham uma fila, é útil poder especificar quantas mensagens cada consumidor pode enviar de uma vez antes de enviar a próxima confirmação. Isso pode ser usado como uma técnica simples de平衡amento de carga ou para melhorar o rendimento se as mensagens tendem a ser publicadas em lotes. Por exemplo, se um aplicativo de produção enviar mensagens a cada minuto devido à natureza do trabalho que está fazendo.

Observe que o RabbitMQ suporta apenas a *prefetching* no nível do canal, não a conexão ou a *prefetching* baseada no tamanho.

Atributos de mensagem e payload

Mensagens no modelo AMQP possuem atributos. Alguns atributos são tão comuns que a especificação AMQP 0-9-1 os define e os desenvolvedores de aplicativos não precisam pensar no nome exato do atributo. Alguns exemplos são:

- Tipo de conteúdo
- Codificação de conteúdo
- Chave de roteamento
- Modo de entrega (persistente ou não)
- Prioridade de mensagem
- Timestamp de publicação de mensagem
- Período de Expiração
- ID do aplicativo do editor

Alguns atributos são usados pelos *brokers* do AMQP, mas a maioria está aberta a interpretações por aplicativos que os recebem. Alguns atributos são opcionais e conhecidos como cabeçalhos. Eles são semelhantes aos X-Headers no HTTP. Atributos de mensagem são definidos quando uma mensagem é publicada.

As mensagens AMQP também têm um *payload* (os dados que elas carregam), que os *brokers* do AMQP tratam como uma matriz de bytes opaca. O *broker* não inspecionará ou modificará o *payload*. É possível que as mensagens contenham apenas atributos e nenhum *payload*. É comum usar formatos de serialização como JSON, Thrift, Protocol Buffers e MessagePack para serializar dados estruturados para publicá-los como *payload* da mensagem. Os pares AMQP geralmente usam os campos "content-type" e "content-encoding" para comunicar essas informações, mas isso é apenas por convenção.

As mensagens podem ser publicadas como persistentes, o que faz com que o *broker* do AMQP as mantenha em disco. Se o servidor for reiniciado, o sistema garante que as mensagens persistentes recebidas não sejam perdidas. A simples publicação de uma mensagem em uma *exchange* durável ou o fato de a(s) fila(s) para a qual ela é roteada serem duráveis não torna a mensagem persistente: tudo depende do modo de persistência da própria mensagem. A publicação de mensagens como persistentes afeta o desempenho (assim como ocorre com os armazenamentos de dados, a durabilidade tem um certo custo no desempenho).

Acknowledgements (confirmação) de Mensagem

Como as redes não são confiáveis e os aplicativos falham, geralmente é necessário ter algum tipo de confirmação de processamento. Às vezes, é necessário apenas reconhecer o fato de que uma mensagem foi recebida. Às vezes, confirmações significam que uma mensagem foi validada e processada por um consumidor, por exemplo, verificada como tendo dados obrigatórios e persistiu em um armazenamento de dados ou indexada.

Essa situação é muito comum, por isso o AMQP 0-9-1 tem um recurso interno chamado de confirmação de mensagem (às vezes chamado de acks) que os consumidores usam para confirmar a entrega e/ou o processamento da mensagem. Se um aplicativo trava (o *broker* AMQP percebe isso quando a conexão é fechada), se uma confirmação de uma mensagem era esperada mas não recebida pelo intermediário AMQP, a mensagem é reenviada (e possivelmente entregue imediatamente a outro consumidor, se houver existente).

Ter reconhecimentos embutidos no protocolo ajuda os desenvolvedores a construir um software mais robusto.

Métodos AMQP 0-9-1

O AMQP 0-9-1 é estruturado como um número de métodos. Os métodos são operações (como os métodos HTTP) e não têm nada em comum com métodos em linguagens de programação orientadas a objeto. Os métodos AMQP são agrupados em classes. Classes são apenas agrupamentos lógicos de métodos AMQP. A referência AMQP 0-9-1 contém detalhes completos de todos os métodos AMQP.

Vamos dar uma olhada na classe de *exchanges*, um grupo de métodos relacionados a operações em *exchanges*. Inclui as seguintes operações:

- `exchange.declare`
- `exchange.declare-ok`
- `exchange.delete`
- `exchange.delete-ok`

As operações acima formam pares lógicos: `exchange.declare` e `exchange.declare-ok`, `exchange.delete` e `exchange.delete-ok`. Essas operações são "solicitações" (enviadas por clientes) e "respostas" (enviadas por *brokers* em resposta às "solicitações" mencionadas anteriormente).

Na maioria das vezes estes métodos são transparentes ao desenvolvedor uma vez que são utilizados pelas bibliotecas específicas das linguagens para efetuarem a comunicação com os *brokers*.

Conexões

As conexões AMQP são geralmente de longa duração. O AMQP é um protocolo em nível de aplicativo que usa o TCP para entrega confiável. As conexões AMQP usam autenticação e podem ser protegidas usando TLS (SSL). Quando um aplicativo não precisa mais estar conectado a um *broker* AMQP, ele deve fechar a conexão AMQP de forma controlada em vez de abruptamente.

Canais

Alguns aplicativos precisam de várias conexões para um *broker* AMQP. No entanto, é indesejável manter muitas conexões TCP abertas ao mesmo tempo, pois isso consome recursos do sistema e dificulta a configuração de *firewalls*. As conexões do AMQP 0-9-1 são multiplexadas com canais que podem ser considerados como "conexões leves que compartilham uma única conexão TCP".

Para aplicativos que usam vários processos/threads para processamento, é muito comum abrir um novo canal por thread/processo e não compartilhar canais entre eles.

A comunicação em um canal específico é completamente separada da comunicação em outro canal, portanto, todo método AMQP também transporta um número de canal que os clientes usam para descobrir para qual canal o método é (e, portanto, qual manipulador de eventos precisa ser chamado, por exemplo).

Hosts Virtuais

Para possibilitar que um único broker hospede vários "ambientes" isolados (grupos de usuários, trocas, filas e assim por diante), o AMQP inclui o conceito de hosts virtuais (*vhosts*). Eles são semelhantes aos hosts virtuais usados por muitos servidores Web populares e fornecem ambientes completamente isolados nos quais as entidades do AMQP residem. Os clientes AMQP especificam quais *vhosts* eles querem usar durante a negociação de conexão AMQP.

AMQP é extensível

O AMQP 0-9-1 possui vários pontos de extensão:

- Os tipos de *exchange* personalizados permitem que os desenvolvedores implementem esquemas de roteamento que tipos de troca *exchange* prontos para o uso não cobrem bem, por exemplo, roteamento baseado em geodados;
- A declaração de *exchanges* e filas pode incluir atributos adicionais que o *broker* pode usar. Por exemplo, a mensagem por fila TTL no RabbitMQ é implementada dessa maneira.
- Extensões específicas do *broker* ao protocolo. Por exemplo, extensões que o RabbitMQ implementa;
- Novas classes de método AMQP 0-9-1 podem ser introduzidas;
- Os *brokers* podem ser estendidos com *plug-ins* adicionais, por exemplo, o *frontend* de gerenciamento do RabbitMQ e a API HTTP são implementados como um plug-in;

Esses recursos tornam o Modelo AMQP 0-9-1 ainda mais flexível e aplicável a uma ampla variedade de problemas.

AMQP 0-9-1 ecossistema de clientes

Existem muitos clientes AMQP 0-9-1 para muitas linguagens e plataformas de programação populares. Alguns deles seguem a terminologia AMQP de perto e apenas fornecem a implementação dos métodos AMQP. Alguns outros possuem recursos adicionais, métodos de conveniência e abstrações. Alguns dos clientes são assíncronos (non-blocking), alguns são síncronos (blocking), alguns suportam ambos os modelos. Alguns clientes suportam extensões específicas do fornecedor (por exemplo, extensões específicas do RabbitMQ).

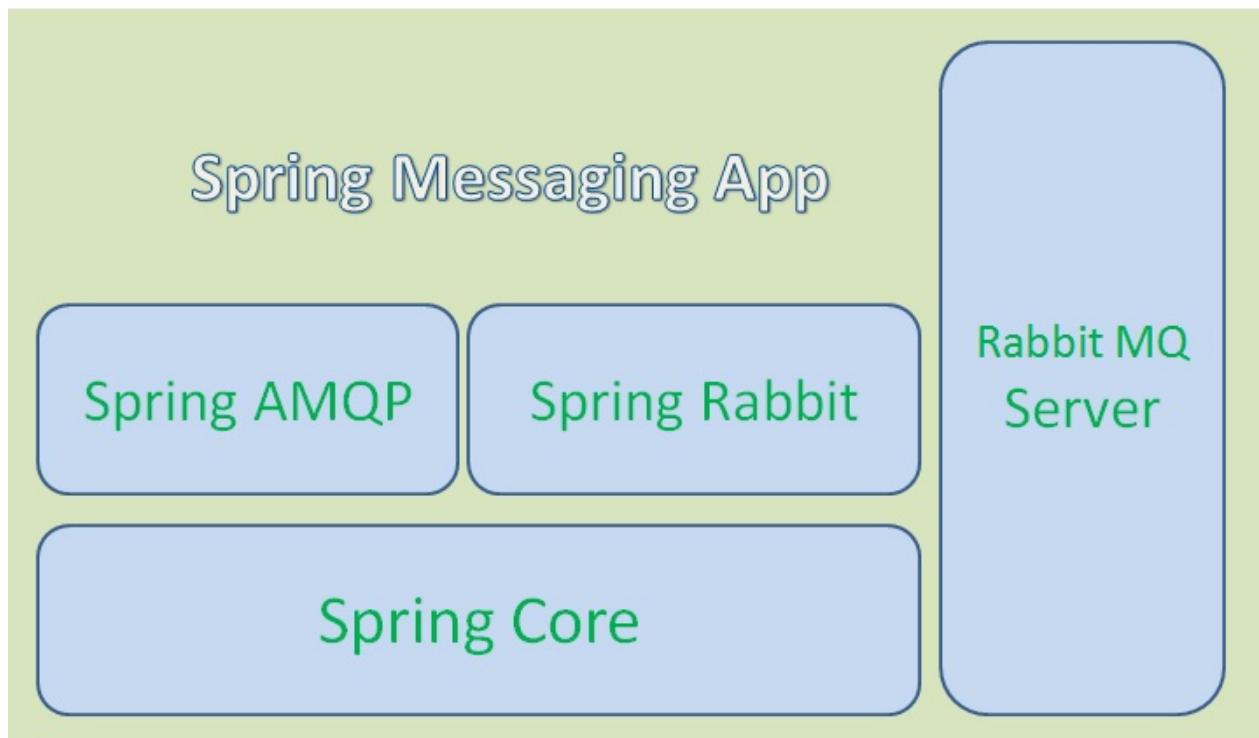
Como um dos principais objetivos do AMQP é a interoperabilidade, é uma boa idéia que os desenvolvedores entendam as operações do protocolo e não se limitem à terminologia de uma determinada biblioteca cliente. Dessa forma, comunicar-se com os desenvolvedores usando bibliotecas diferentes será significativamente mais fácil.

Cadastrando um livro de forma assíncrona

Spring AMQP

O projeto Spring AMQP aplica os principais conceitos do Spring ao desenvolvimento de soluções de mensagens baseadas no AMQP. Ele fornece um "modelo" como uma abstração de alto nível para enviar e receber mensagens. Ele também fornece suporte para POJOs orientados por mensagens com um "listener container". Essas bibliotecas facilitam o gerenciamento de recursos do AMQP enquanto promovem o uso de injeção de dependência e configuração declarativa.

O projeto consiste em duas partes; O `spring-amqp` é a abstração base e o `spring-rabbit` é a implementação do RabbitMQ.



Criando um cadastro de livro que "demora"

Em nosso controller vamos adicionar um método que cadastra um livro mas que demora a responder:

Agora vamos ajustar o controller de fato:

- src/main/java/com/acme/livroservice/LivrosController.java

```
// Código atual omitido
import java.util.concurrent.TimeUnit;

@RestController
@RequestMapping("/livros")
public class LivrosController {

    // Código atual omitido
    @PostMapping("/demorado")
    @ResponseStatus(HttpStatus.CREATED)
    public Livro adicionarLivroDemorado(@RequestBody Livro livro) throws InterruptedException {
        return salvarLivroDemorado(livro);
    }

    public Livro salvarLivroDemorado(Livro livro) throws InterruptedException {
        logger.info("adicionarLivroDemorado iniciou: " + livro);
        TimeUnit.SECONDS.sleep(3);
        Livro livroSalvo = repository.save(livro);
        logger.info("adicionarLivroDemorado terminou: " + livroSalvo);
        return livroSalvo;
    }
}
```

Utilizando o RESTClient do Firefox, veja se realmente o livro está sendo salvo mas a requisição deve estar demorando 3 segundos para responder.

Um cenário para a utilização de filas

Vamos imaginar que temos um problema que é o tempo de processamento do salvamento dos livros está demorando, para resolver isso, podemos fazer com que nossa aplicação, ao receber a solicitação de cadastramento de um livro, envie esta solicitação para uma fila e responda imediatamente ao solicitante. A fila será processada na medida da disponibilidade dos recursos. Utilizaremos o RabbitMQ para esta funcionalidade.

Adicionando Dependências ao pom.xml de Nossa Aplicação

Vamos incluir as dependências `spring-boot-starter-amqp` em nosso `pom.xml`:

- `pom.xml`

```
<!-- Código anterior omitido -->
<dependencies>

    <!-- Dependências atuais omitidas -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-amqp</artifactId>
    </dependency>

</dependencies>
<!-- Código posterior omitido -->
```

Agora vamos criar algumas constantes que nos auxiliarão no restante do processo:

```

package com.acme.livroservice;

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

@SpringBootApplication
public class LivroServiceApplication {

    // Novidades aqui
    static final String MATRICULA = "NNNNNNNN";
    static final String LIVRO_DIRECT_EXCHANGE_NAME = "livro-direct-exchange-" + MATRICULA;
    static final String CADASTRAR_LIVRO_QUEUE_NAME = "cadastrar_livro_queue_" + MATRICULA;
    static final String CADASTRAR_LIVRO_ROUTING_KEY = "livro.cadastrar." + MATRICULA;

    public static void main(String[] args) {
        SpringApplication.run(LivroServiceApplication.class, args);
    }
}

```

Crie um receptor de mensagem RabbitMQ

Com qualquer aplicativo baseado em mensagens, você precisa criar um receptor que responda às mensagens publicadas.

- src/main/java/com/acme/livroservice/Receiver.java

```

package com.acme.livroservice;

import java.util.concurrent.TimeUnit;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class Receiver {
    Logger logger = LoggerFactory.getLogger(Receiver.class);

    @RabbitListener(queues = LivroServiceApplication.CADASTRAR_LIVRO_QUEUE_NAME)
    public void receiveMessageCadastrarLivro(String message) throws InterruptedException {
        logger.info("Recebeu <" + message + ">");
    }
}

```

O **Receiver** é um POJO simples que define um método para receber mensagens. Poderia ser utilizado qualquer outro nome desejado. **@RabbitListener** é uma anotação utilizada para mapear mensagens destinadas a determinada fila.

Enviando as mensagens

Agora, ajustaremos nosso controller para que envie as mensagens ao *broker*.

- src/main/java/com/acme/livroservice/LivrosController.java

```

package com.acme.livroservice;

// Novidade aqui
import org.springframework.rabbit.core.RabbitTemplate;

// Código atual omitido

@RestController
@RequestMapping("/livros")
public class LivrosController {

    Logger logger = LoggerFactory.getLogger(LivrosController.class);

    private final LivroRepository repository;

    // Novidade aqui
    private final RabbitTemplate rabbitTemplate;

    // Novidade aqui
    LivrosController(LivroRepository repository, RabbitTemplate rabbitTemplate) {
        this.repository = repository;
        this.rabbitTemplate = rabbitTemplate;
    }

    // Código atual omitido

    // Novidade aqui
    @PostMapping("/async/direct")
    @ResponseStatus(HttpStatus.CREATED)
    public void adicionarLivroAsyncDirect(@RequestBody Livro livro) throws InterruptedException {
        logger.info("adicionarLivroAsyncDirect iniciou: " + livro);
        rabbitTemplate.convertAndSend(LivroServiceApplication.LIVRO_DIRECT_EXCHANGE_NAME, LivroServiceApplication.CADASTRAR_LIVRO_ROUTING_KEY, livro.toString());
        logger.info("adicionarLivroAsyncDirect terminou");
    }
}

```

Configurando o endereço do broker

A configuração do endereço do broker é feita no arquivo `application.properties`, deixaremos o log com nível `DEBUG` para que possamos ver as mensagens que estão sendo enviadas.

`/src/main/resources/application.properties`

```

logging.level.org.springframework.amqp=DEBUG
spring.rabbitmq.host=[IP-DO-HOST]

```

Criando as Exchanges, Queues e Bindings

O próximo passo é realizar a criação das Exchanges, Queues e Bindings via interface de administração do RabbitMQ: [http://\[IP-DO-HOST\]:15672](http://[IP-DO-HOST]:15672)

- Crie uma Direct Exchange com o nome: `livro-direct-exchange-NNNNNNNN`
- Crie uma Queue com o nome: `livro_queue_NNNNNNNN`
- Crie um Binding entre a exchange e a queue com o valor: `livro.cadastrar.NNNNNNNN`

(substitua NNNNNNNN pela sua matrícula)

Testando o envio de mensagens

Ótimo, agora já é possível ver as mensagens sendo enviadas e processadas por nossa aplicação.

```

2019-02-05 22:34:49.390 INFO 19352 --- [nio-8080-exec-2] com.acme.livroservice.LivrosController : adicionarLivroAssincrono
iniciou: Livro [id=null, autor=string, titulo=string, preco=0.0]
2019-02-05 22:34:49.391 DEBUG 19352 --- [nio-8080-exec-2] o.s.amqp.rabbit.core.RabbitTemplate : Executing callback Rabbit
Template$$Lambda$907/0x000000080086cc40 on RabbitMQ Channel: Cached Rabbit Channel: AMQChannel(amqp://guest@127.0.0.1:5672/,2)
, conn: Proxy@587878da Shared Rabbit Connection: SimpleConnection@199a59d7 [delegate=amqp://guest@127.0.0.1:5672/, localPort=
56092]
2019-02-05 22:34:49.391 DEBUG 19352 --- [nio-8080-exec-2] o.s.amqp.rabbit.core.RabbitTemplate : Publishing message (Body:
'Livro [id=null, autor=string, titulo=string, preco=0.0]' MessageProperties [headers={}, contentType=text/plain, contentEncodi
ng=UTF-8, contentLength=55, deliveryMode=PERSISTENT, priority=0, deliveryTag=0])on exchange [cadastrar_livro_queue_NNNNNNNN], routingKey = [livro.cadastrar.NNNNNNNN]
2019-02-05 22:34:49.391 INFO 19352 --- [nio-8080-exec-2] com.acme.livroservice.LivrosController : adicionarLivroAssincrono
terminou
2019-02-05 22:34:49.405 DEBUG 19352 --- [pool-3-thread-5] o.s.a.r.listener.BlockingQueueConsumer : Storing delivery for cons
umerTag: 'amq.ctag-0Sunur9xyt4x5ovgRsD5Qg' with deliveryTag: '2' in Consumer@50dd38ea: tags=[[amq.ctag-0Sunur9xyt4x5ovgRsD5Qg]
], channel=Cached Rabbit Channel: AMQChannel(amqp://guest@127.0.0.1:5672/,1), conn: Proxy@587878da Shared Rabbit Connection: S
impleConnection@199a59d7 [delegate=amqp://guest@127.0.0.1:5672/, localPort= 56092], acknowledgeMode=AUTO local queue size=0
2019-02-05 22:34:49.406 DEBUG 19352 --- [cTaskExecutor-1] o.s.a.r.listener.BlockingQueueConsumer : Received message: (Body:
'Livro [id=null, autor=string, titulo=string, preco=0.0]' MessageProperties [headers={}, contentType=text/plain, contentEncodin
g=UTF-8, contentLength=0, receivedDeliveryMode=PERSISTENT, priority=0, redelivered=false, receivedExchange=cadastrar_livro_QUE
UE_NNNNNNNN, receivedRoutingKey=livro.cadastrar.NNNNNNNN, deliveryTag=2, consumerTag=amq.ctag-0Sunur9xyt4x5ovgRsD5Qg, consumer
Queue=cadastrar_livro_queue_NNNNNNNN])
2019-02-05 22:34:49.406 DEBUG 19352 --- [cTaskExecutor-1] .a.r.l.a.MessagingMessageListenerAdapter : Processing [GenericMessag
e [payload=Livro [id=null, autor=string, titulo=string, preco=0.0], headers={amqp_receivedDeliveryMode=PERSISTENT, amqp_receiv
edRoutingKey=livro.cadastrar.NNNNNNNN, amqp_contentEncoding=UTF-8, amqp_receivedExchange=cadastrar_livro_queue_NNNNNNNN, amqp_
deliveryTag=2, amqp_consumerQueue=cadastrar_livro_queue_NNNNNNNN, amqp_redelivered=false, id=1f7b0044-9de5-3537-4ce3-69a3ca5bb
a9d, amqp_consumerTag=amq.ctag-0Sunur9xyt4x5ovgRsD5Qg, contentType=text/plain, timestamp=1549413289406}]]
2019-02-05 22:34:49.406 INFO 19352 --- [cTaskExecutor-1] com.acme.livroservice.LivrosController : Recebeu <Livro [id=null,
autor=string, titulo=string, preco=0.0]>
2019-02-05 22:34:52.406 INFO 19352 --- [cTaskExecutor-1] com.acme.livroservice.LivrosController : Processou <Livro [id=null
, autor=string, titulo=string, preco=0.0]>

```

Envio de objetos na mensagem

Um payload de uma mensagem é um array de bytes, deste modo, podemos enviar representações serializadas do objeto e recebê-las para facilitar o processamento, vamos fazer uma alteração e permitir que o objeto "Livro" seja enviado serializado na mensagem.

O primeiro passo é tornar o Livro um objeto serializável:

- /src/main/java/com/acme/livroservice/Livro.java

```

package com.acme.livroservice;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
// Novidade aqui
public class Livro implements Serializable {
    private static final long serialVersionUID = 1L;

    // Código atual omitido
}

```

Vamos ajustar em seguida a classe `LivrosController` para que faça o envio de uma instância do próprio Livro e não sua representação textual:

- src/main/java/com/acme/livroservice/LivrosController.java

```
// Código atual omitido
public class LivrosController {

    // Código atual omitido
    @PostMapping("/async/direct")
    @ResponseStatus(HttpStatus.CREATED)
    public void adicionarLivroAsyncDirect(@RequestBody Livro livro) throws InterruptedException {
        logger.info("adicionarLivroAsyncDirect iniciou: " + livro);
        rabbitTemplate.convertAndSend(LivroServiceApplication.LIVRO_DIRECT_EXCHANGE_NAME, LivroServiceApplication.CADASTRAR_LIVRO_ROUTING_KEY, livro);
    }
}
```

O `Receiver` também deve ser ajustado, e agora já poderá fazer a persistência do objeto recebido:

- src/main/java/com/acme/livroservice/Receiver.java

```
package com.acme.livroservice;

import java.util.concurrent.TimeUnit;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

@Component
public class Receiver {

    Logger logger = LoggerFactory.getLogger(Receiver.class);

    private final LivroRepository repository;

    Receiver(LivroRepository repository) {
        this.repository = repository;
    }

    @RabbitListener(queues = LivroServiceApplication.CADASTRAR_LIVRO_QUEUE_NAME)
    public void receiveMessage(Livro livro) throws InterruptedException {
        logger.info("Recebeu <" + livro.toString() + ">");
        TimeUnit.SECONDS.sleep(3);
        repository.save(livro);
        logger.info("Processou <" + livro.toString() + ">");
    }
}
```

Se consultar o log, verá que agora o conteúdo enviado é bem diferente, em especial o **contentType=application/x-java-serialized-object**:

```
2019-02-05 22:48:10.227 DEBUG 18832 --- [nio-8080-exec-9] o.s.amqp.rabbit.core.RabbitTemplate : Publishing message (Body: '[B@74824f3b[byte[243]])' MessageProperties [headers={}, contentType=application/x-java-serialized-object, contentLength=243, deliveryMode=PERSISTENT, priority=0, deliveryTag=0])on exchange [cadastrar_livro_queue_NNNNNNNN], routingKey = [livro.cadastrar.NNNNNNNN]
```

Conseguimos enviar o objeto ao *broker*, recebê-lo em seguida e fazer sua persistência no banco. Porém estamos trafegando dados binários, o que dificulta a integração de nossa aplicação com outras tecnologias (talvez um serviço NodeJS poderia enviar o livro para persistência).

Para evitar este problema, iremos alterar novamente o projeto para que o Livro seja enviado no formato JSON.

Em `LivroServiceApplication` devemos incluir os métodos **producerJackson2MessageConverter** e **rabbitTemplate**:

src/main/java/com/acme/livroservice/LivroServiceApplication.java

```

package com.acme.livroservice;

import org.springframework.amqp.rabbit.connectionConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class LivroServiceApplication {

    static final String MATRICULA = "NNNNNNNN";
    static final String LIVRO_DIRECT_EXCHANGE_NAME = "livro-direct-exchange-" + MATRICULA;
    static final String CADASTRAR_LIVRO_QUEUE_NAME = "cadastrar_livro_queue_" + MATRICULA;
    static final String CADASTRAR_LIVRO_ROUTING_KEY = "livro.cadastrar." + MATRICULA;

    @Bean
    public RabbitTemplate rabbitTemplate(final ConnectionFactory connectionFactory,
                                         MessageConverter producerJackson2MessageConverter) {
        final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
        rabbitTemplate.setMessageConverter(producerJackson2MessageConverter);
        return rabbitTemplate;
    }

    @Bean
    public Jackson2JsonMessageConverter producerJackson2MessageConverter() {
        return new Jackson2JsonMessageConverter();
    }

    public static void main(String[] args) {
        SpringApplication.run(LivroServiceApplication.class, args);
    }
}

```

Tudo deve continuar funcionando como antes, porém, ao incluirmos um livro, a saída do log deve conter algo como **contentType=application/json**:

```

2019-02-05 22:51:17.948 DEBUG 23344 --- [nio-8080-exec-1] o.s.amqp.rabbit.core.RabbitTemplate      : Publishing message (Body:
'{"id":null,"autor":"string","titulo":"string","preco":0.0}' MessageProperties [headers={__TypeId__=com.acme.livroservice.Livr
o}, contentType=application/json, contentEncoding=UTF-8, contentLength=58, deliveryMode=PERSISTENT, priority=0, deliveryTag=0]
on exchange [cadastrar_livro_queue_NNNNNNNN], routingKey = [livro.cadastrar.NNNNNNNN]

```

Perceba que o conteúdo do `Body` é um JSON e `contentType` agora é `application/json`.

Distribuindo a carga de trabalho

Levante mais duas instâncias do microsserviço em portas distintas e faça novamente a solicitação de cadastro de livro assíncrono, perceba que o trabalho é distribuído entre as instâncias:

```

$ java -jar livro-service-0.0.1-SNAPSHOT.jar --server.port=7081
$ java -jar livro-service-0.0.1-SNAPSHOT.jar --server.port=7082

```

Excluindo livros de forma assíncrona

Faça o mesmo agora para a funcionalidade de exclusão de livros, para isso, crie um end-point de exclusão de livros assíncrono, uma queue, um binding e um novo método no receiver.

Excluindo avaliações de um livro de forma assíncrona

Já que estamos excluindo os livros de forma assíncrona, seria interessante também realizar a exclusão das avaliações relacionadas a este livro de forma assíncrona.

Multicast de Mensagens

Para exercitarmos o envio de mensagens em "multicast", podemos utilizar o padrão fanout de exchange, para isso, podemos utilizar o recurso de criação programática de Exchanges/Queues/Bindings do RabbitMQ.

Primeiro, vamos ajustar **LivroServiceApplication**:

```
package com.acme.livroservice;

// Código atual omitido

@SpringBootApplication
public class LivroServiceApplication {

    static final String MATRICULA = "NNNNNNNN";
    static final String LIVRO_DIRECT_EXCHANGE_NAME = "livro-direct-exchange-" + MATRICULA;
    static final String LIVRO_FANOUT_EXCHANGE_NAME = "livro-fanout-exchange";
    static final String CADASTRAR_LIVRO_QUEUE_NAME = "cadastrar_livro_queue_" + MATRICULA;
    static final String CADASTRAR_LIVRO_ROUTING_KEY = "livro.cadastrar." + MATRICULA;

    // Novidades aqui
    static final String EXCLUIR_LIVRO_QUEUE_NAME = "excluir_livro_queue_" + MATRICULA;
    static final String EXCLUIR_LIVRO_ROUTING_KEY = "livro.excluir." + MATRICULA;

    @Bean
    public Queue cadastrarLivroQueue() {
        return new Queue(CADASTRAR_LIVRO_QUEUE_NAME);
    }

    @Bean
    public FanoutExchange fanoutExchange() {
        return new FanoutExchange(LIVRO_FANOUT_EXCHANGE_NAME);
    }

    @Bean
    public Binding binding(Queue cadastrarLivroQueue, FanoutExchange fanoutExchange) {
        return BindingBuilder.bind(cadastrarLivroQueue).to(fanoutExchange);
    }

    // Código atual omitido
}
```

Agora vamos ajustar o controller para que tenha um método de publicação de mensagens multicast:

- src/main/java/com/acme/livroservice/LivrosController.java

```
// Código atual omitido
import java.util.concurrent.TimeUnit;

@RestController
@RequestMapping("/livros")
public class LivrosController {

    // Código atual omitido

    // Novidade aqui
    @PostMapping("/async/fanout")
    @ResponseStatus(HttpStatus.CREATED)
    public void adicionarLivroAsyncFanout(@RequestBody Livro livro) throws InterruptedException {
        logger.info("adicionarLivroAsyncFanout iniciou: " + livro);
        rabbitTemplate.convertAndSend(LivroServiceApplication.LIVRO_FANOUT_EXCHANGE_NAME, "/*", livro);
        logger.info("adicionarLivroAsyncFanout terminou");
    }
}
```

Mensagens por tópico

Já implementamos um cadastro de livros via *direct exchanges*, *fanout exchanges* e agora, iremos fazer o mesmo utilizando uma *topic exchange*.

Para desabilitar temporariamente o RabbitMQ e evitar erros de conexão durante o restante do treinamento, anote a classe Receiver com `@Profile("disabled")`

Fontes

- <https://spring.io/projects/spring-amqp>
- <https://spring.io/guides/gs/messaging-rabbitmq/>
- <https://thepracticaldeveloper.com/2016/10/23/produce-and-consume-json-messages-with-spring-boot-amqp/>
- <https://springbootdev.com/2017/09/15/spring-boot-and-rabbitmq-direct-exchange-example-messaging-custom-java-objects-and-consumes-with-a-listener/>

Uso em alta disponibilidade (cluster)

A alta disponibilidade é obtida através do espelhamento de filas.

Por padrão, o conteúdo de uma fila em um cluster RabbitMQ está localizado em um único nó (o nó no qual a fila foi declarada). Isso está em contraste com `exchanges` e `ligações`, que sempre podem ser consideradas em todos os nós. As filas podem, opcionalmente, ser espelhadas em vários nós.

Cada fila espelhada consiste em um mestre e um ou mais espelhos. O mestre está hospedado em um nó comumente referido como o nó mestre. Cada fila tem seu próprio nó mestre. Todas as operações para uma determinada fila são aplicadas primeiro no nó principal da fila e, em seguida, propagadas para espelhos. Isso envolve a publicação em fila, a entrega de mensagens aos consumidores, o rastreamento de confirmações dos consumidores e assim por diante.

Espelhamento de fila implica um cluster de nós. Portanto, ele não é recomendado para uso em uma WAN (embora, é claro, os clientes ainda possam se conectar de forma tão próxima e tão necessária).

Mensagens publicadas na fila são replicadas para todos os espelhos. Os consumidores estão conectados ao mestre, independentemente do nó ao qual se conectam, com os espelhos entregando mensagens que foram reconhecidas no mestre. O espelhamento de filas, portanto, aumenta a disponibilidade, mas não distribui a carga entre os nós (todos os nós participantes realizam todo o trabalho).

Se o nó que hospeda o mestre da fila falhar, o espelho mais antigo será promovido para o novo mestre, desde que seja sincronizado. Os espelhos não sincronizados também podem ser promovidos, dependendo dos parâmetros de espelhamento de fila.

Existem vários termos comumente usados para identificar réplicas primárias e secundárias em um sistema distribuído. Normalmente usa-se "mestre" para se referir à réplica primária de uma fila e "espelho" para réplicas secundárias. No entanto, você encontrará o termo "escravo" usado em alguns pontos. Isso ocorre porque as ferramentas de CLI do RabbitMQ historicamente usam o termo "escravo" para se referir a secundários. Portanto, ambos os termos são atualmente usados de forma intercambiável, porém é uma terminologia legada.

Características da Arquitetura Orientada a Microsserviço

"Microsserviços" – um novo nome nas populosas ruas da arquitetura de software. Embora nossa tendência natural é ignorar estas novidades com um olhar de desprezo, essa nova terminologia descreve um estilo de sistema que achamos cada vez mais atraente. Temos visto muitos projetos usando este formato nos últimos anos e os resultados tem sido positivos, tanto que para muitos dos nossos colegas esta se tornou a forma padrão para desenvolver aplicações empresariais. Infelizmente, entretanto, não existe muita informação que descreva o que são microsserviços e como implementá-los.

Em resumo, microsserviço é uma abordagem para desenvolver uma única aplicação como uma suíte de serviços, cada um rodando em seu próprio processo e se comunicando através de mecanismos leves, geralmente através de uma API HTTP. Estes serviços são construídos através de pequenas responsabilidades e publicados em produção de maneira independente através de processos de deploys automatizados. Existe um gerenciamento centralizado mínimo destes serviços, que podem serem escritos em diferentes linguagens e usarem diferentes tecnologias para armazenamento de dados.

Para começar explicando o que é o padrão de microsserviços, podemos compará-lo ao padrão monolítico: uma aplicação monolítica é feita como uma única unidade. Aplicações empresariais são geralmente construídas em três partes principais: uma interface para o cliente (tais como páginas HTML e Javascript rodando em um navegador no computador do usuário), um banco de dados (várias tabelas em um mesmo lugar, geralmente um sistema de banco de dados relacional) e uma aplicação server-side. A aplicação server-side irá manipular as requisições HTTP, executar toda lógica de domínio, receber e atualizar os dados da base de dados e por fim, selecionar e popular os blocos HTML para enviar ao navegador. Esta aplicação server-side é monolítica – uma única unidade lógica executável. Qualquer mudança no sistema consiste em publicar uma nova versão da aplicação server-side.

Este tipo de servidor monolítico é uma forma natural de construir um sistema. Toda sua lógica para manipular uma requisição é executada em um único processo, permitindo que você use as características básicas da sua linguagem para separar sua aplicação em classes, funções e namespaces. Com algum cuidado, você pode rodar e testar a aplicação no computador do desenvolvedor e usar uma seqüência de integração para garantir que todas as mudanças sejam propriamente testadas e implantadas em produção. Você pode escalar uma aplicação monolítica horizontalmente rodando diversas instâncias através de um *load-balancer*.

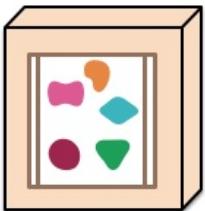
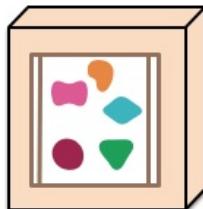
Aplicações monolíticas podem ser bem-sucedidas, porém as pessoas começarão a ficar frustradas – especialmente quando muitas aplicações começarem a serem implantadas na nuvem. Ciclos de mudanças começam a ficar amarrados – uma pequena alteração feita em uma parte pequena do software faz com que toda a aplicação monolítica necessite ser republicada. Com o passar do tempo ficará cada vez mais difícil manter uma estrutura modular, sendo difícil separar as mudanças que deveriam afetar somente um módulo. Para escalar é necessário escalar toda a aplicação, ao invés de escalar somente as partes que necessitem de maiores recursos.

- Figura 1: Aplicações monolíticas e microsserviços.

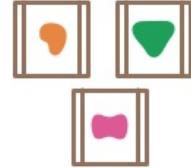
Uma aplicação monolítica coloca toda sua funcionalidade em um único processo...



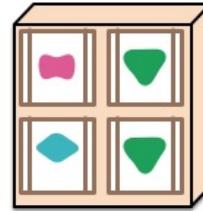
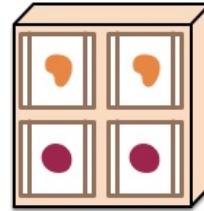
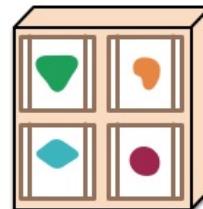
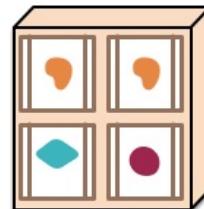
... e escala replicando a aplicação monolítica em vários servidores



Uma arquitetura em microsserviços põe cada elemento de uma funcionalidade em um serviço separado ...



... e escala distribuindo estes serviços entre os servidores, replicando quando necessário.



Estas frustrações levaram ao padrão de microsserviços: aplicações como uma suíte de serviços. Além do fato de que os serviços são implantados e escalam de maneiras independentes, cada serviço também provê uma fronteira bem definida entre os módulos, permitindo até mesmo que diferentes serviços sejam escritos em diferentes linguagens de programação. Eles podem inclusive ser administrados por times diferentes.

Nós não tomamos o padrão de microsserviços como algo novo, suas raízes remetem no mínimo aos princípios de design do próprio Unix. Porém nós cremos que poucas pessoas consideram uma arquitetura de microsserviços e que o desenvolvimento de muitos softwares seriam melhores se adotassem esse padrão.

Não podemos dizer que existe uma definição formal para uma arquitetura em micro serviços, mas podemos tentar descrever o que temos visto como características comuns em arquiteturas que caibam no padrão. Como em qualquer definição que descreva características em comum, nem todas as arquiteturas em microsserviços tem todas as características, mas nós acreditamos que a maioria das arquiteturas em microsserviços exibam a maior parte das características a serem citadas. Como membros ativos na comunidade sobre este tema, temos tentado descrever o que temos visto em nosso próprio trabalho e através de esforços semelhantes em times que nós temos conhecimento.

Componentização via serviços

Vemos na indústria de softwares um desejo de construir sistemas plugando componentes entre si, tal como vemos as coisas serem feitas no mundo físico. Durante as últimas décadas, percebemos progressos consideráveis através de várias bibliotecas que fazem parte da maior parte das plataformas de desenvolvimento.

Quando falamos sobre componentes, encontramos a dificuldade em definir o que é um componente. Nossa definição é que um componente é uma unidade de software que é substituída ou atualizada de maneira independente.

Arquiteturas em microsserviços usarão bibliotecas, mas buscam primeiramente organizar seu próprio software dividindo em serviços. Nós definimos bibliotecas como componentes que são usados em um programa através de chamadas de função diretamente em memória, enquanto serviços são componentes em processos diferentes que se comunicam através de mecanismos tais como requisições via web services ou chamadas de código remotas (este é um conceito diferente de objeto de serviço, encontrado em muitas linguagens orientadas a objeto).

Uma das principais razões para usar serviços como componentes (ao invés de bibliotecas) é que serviços são publicados de maneira independente. Se você tem uma aplicação que consiste em diversas bibliotecas em um único processo, uma mudança em qualquer componente resulta em ter que republicar toda sua aplicação. Mas se esta aplicação é dividida em múltiplos serviços, você pode esperar que diversas mudanças em um único serviço exijam uma republicação somente no serviço alterado. Isso não é algo absoluto, pois algumas mudanças criam alterações nas interfaces entre os serviços, mas o dever de uma boa arquitetura em microsserviços é minimizar este impacto, criando interfaces coesas e mecanismos para evolução entre os serviços.

Outra consequência em usar serviços como componentes é ter uma interface mais explícita. A maioria das linguagens não tem uma boa forma de definir explicitamente uma interface do tipo *Published Interface*. Freqüentemente só conseguimos impedir uma violação no encapsulamento de um componente através da documentação e muita disciplina, o que leva a um alto acoplamento entre os componentes. Serviços ajudam a evitar esse problema, usando mecanismos de chamadas remotas.

Usar serviços desta forma tem alguns efeitos colaterais. Chamadas remotas são mais custosas que chamadas dentro do mesmo processo e APIs remotas precisam ser granulares, o que torna ainda mais complicado para usar. Se você precisa mudar as responsabilidades entre os componentes, tais mudanças de comportamento são mais difíceis de fazer do que quando você consegue ultrapassar as fronteiras entre os processos.

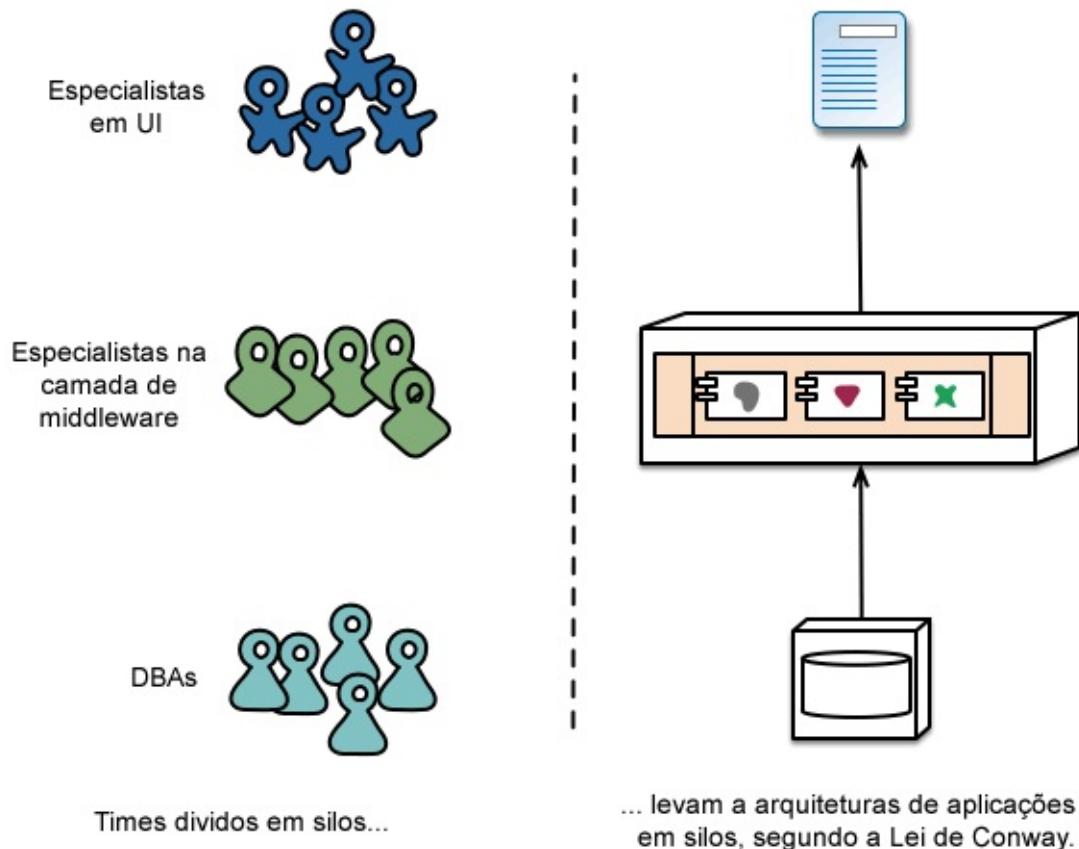
Em um primeiro momento, nós podemos ver quais serviços apontam para processos em execução, mas isso é só uma primeira impressão. Um serviço pode consistir em diversos processos que serão sempre desenvolvidos e publicados juntos, como uma única aplicação e um banco de dados que é usado por cada serviço.

Organizado através das áreas do negócio

Quando procuramos dividir uma grande aplicação em partes, o foco geralmente é na camada de tecnologia, levando os times a serem divididos entre aqueles que cuidam da interface, da lógica server-side e do banco de dados. Quando times são divididos desta forma, até mesmo mudanças simples podem exigir bastante tempo e aprovação financeira em projetos que envolvam diversos times. Ao fugir destes problemas, o time pode acabar trazendo a lógica para as aplicações que eles têm acesso. Em outras palavras, lógica em todos os lugares. Este é um exemplo da Lei de Conway em ação.

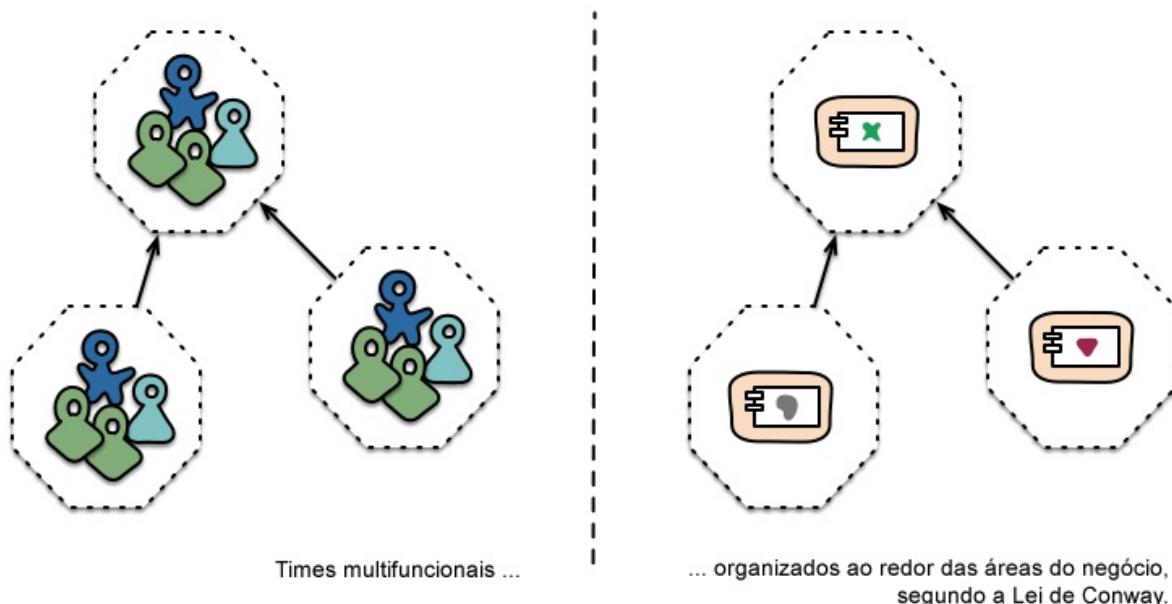
Qualquer organização que desenha um sistema (definido de forma ampla) irá produzir um design cuja estrutura é uma cópia da estrutura de comunicação da própria organização. — Melvyn Conway, 1967

- Figura 2: Lei de Conway em ação



A abordagem proposta pelos micros-serviços para esta divisão é diferente, ao organizar os times ao redor das **áreas do negócio**. Assim os serviços possuirão uma implementação satisfatória para uma determinada área do negócio, incluindo uma interface com usuário, armazenamento de dados persistente e qualquer outra necessidade externa. Conseqüentemente, os times se tornam multifuncionais, levando toda bagagem necessária para o desenvolvimento: interface com o usuário, banco de dados e o gerenciamento do projeto.

- Figura 3: Limites dos serviços reforçados pelos limites dos times



Uma empresa organizada desta forma é a www.comparethemarket.com. Times multifuncionais ficam responsáveis por construir e operar cada produto e cada produto é quebrado em serviços individuais se comunicando via um barramento de mensagens.

Grandes aplicações monolíticas podem sempre ser modularizadas ao redor das áreas do negócio também, embora nem sempre este seja o caso mais comum. Certamente devemos incentivar um time grande construindo uma aplicação monolítica a se dividir.

Qual é o tamanho de um microsserviço?

Embora o termo "microsserviço" tenha se tornado um nome popular para este estilo de arquitetura, seu nome remete a um significado infeliz focando no tamanho do serviços e isso se dá por conta da palavra "micro". Nas nossas conversas com aqueles que já usam os microsserviços, temos visto diversos tamanhos de serviços. Os maiores que notamos seguem a noção da Amazon de "Two Pizza Team" (ou seja, o time inteiro consegue ser alimentados por duas pizzas), significando que não pode se ter mais do que uma dúzia de pessoas. Em uma escala menor, vimos cenários onde um time de meia dúzia de pessoas conseguia cuidar de meia dúzia de serviços.

Isto leva a questionar se esta diferença entre "serviços a cada dúzia de pessoas" e "serviços por pessoa" deveria estar debaixo da categoria de microsserviços. Até então pensamos que é melhor agrupá-los, mas é possível que mudemos nossa mente conforme explorarmos este estilo no futuro.

Produtos e não projetos

Vemos que a maior parte dos esforços para o desenvolvimento de uma aplicação usa um modelo de projeto: onde o dever é entregar algum software que é por si só considerado o fim. Após isso o software é entregue para quem cuidará da manutenção e o time que construiu o software é desfeito.

Microsserviços tentam evitar este modelo, dando preferência a idéia de que um time deve possuir o produto durante todo seu tempo de vida. Uma inspiração para isso é a noção da Amazon de "*you build, you run it*", onde um time de desenvolvimento tem a responsabilidade completa de um software em produção. Isto traz os desenvolvedores para o contato diário com seus softwares, sabendo como se comportam em produção e tendo contato com seus usuários, uma vez que eles terão que lidar com uma parte do fardo do suporte.

A mentalidade de produto unida com as áreas do negócio. Ao invés de enxergar o software como um conjunto de funcionalidades a serem criadas, existe um comprometimento durante todo o processo de desenvolvimento em como um software pode ajudar seus usuários a desenvolverem as capacidades do negócio.

Não existe um motivo para que esta mesma abordagem não seja adotada em aplicações monolíticas, mas uma granularidade menor dos serviços pode facilitar a relação entre os desenvolvedores de serviços e seus usuários.

Endpoints inteligentes e fluxo de comunicação simples

Ao construir estruturas para comunicação entre diferentes processos, vimos muitos produtos e abordagens que dificultam a tarefa criando grandes inteligências no mecanismo de comunicação por si só. Um bom exemplo disso é o Enterprise Service Bus (ESB), onde produtos ESB freqüentemente incluem formas sofisticadas de roteamento de mensagens, transformação e aplicação das regras de negócio.

A comunidade que usa os microsserviços propõe uma abordagem alternativa: endpoints inteligentes e fluxos de comunicação simples. Aplicações construídas a partir de microsserviços devem ser tão desacopladas e coesas quanto possível – elas devem possuir seu próprio domínio lógico e agir mais como um filtro de uma forma parecida com o Unix clássico – recebendo uma requisição, aplicando a lógica apropriada e produzindo uma resposta. Isso pode ser gerenciado usando protocolos REST simples ao invés de protocolos complexos como WS-Choreography ou BPEL, ou o gerenciamento através de uma ferramenta central.

Os dois protocolos mais usados são o HTTP usando APIs e um sistema de mensagens simples. Melhor definição para isso é:

"Seja feito **de** Web, não **atrás** da Web." — Ian Robinson

Microsserviços usam os princípios e protocolos que a Web usa (e em um contexto maior, o Unix também). Freqüentemente os recursos usados podem ser cacheados com pouco esforço pelos desenvolvedores e o time que cuida da operação.

A segunda abordagem geralmente usada é a comunicação através de um barramento de mensagens simples. A infraestrutura usada é simples (porque age somente como um roteador de mensagens), tais como o RabbitMQ ou ZeroMQ que não fazem muito mais que prover uma base de assincronismo confiável – a inteligência continua existindo nos endpoints que estão produzindo e consumindo mensagens nos serviços.

Na estrutura monolítica os componentes são executados no mesmo processo e a comunicação entre eles é feita via invocação de métodos ou chamadas de função. O maior problema em mudar de uma estrutura monolítica para de microsserviços está em mudar o padrão de comunicação. Amudança de uma comunicação nativa que usa chamadas de métodos para chamadas remotas pode precisar de muitas iterações – o que não performa bem. Na prática, você precisa substituir uma comunicação que gera muitas iterações para uma abordagem menos granular.

Governança descentralizada

Uma das consequências de governanças centralizadas é a tendência de padronizar tudo em uma única plataforma tecnológica. A experiência mostra que esta abordagem é limitada – nem todo problema é um prego, nem toda solução é um martelo. Nós preferimos usar a ferramenta certa para o trabalho e, embora aplicações monolíticas também possam usar diferentes linguagens, isso não é comum.

Ao quebrar componentes monolíticos em serviços temos a escolha de como construir cada um deles de maneira diferente. Você quer usar Node.js para levantar uma página simples de relatórios? Faça. C++ é uma escolha boa para um componente real-time? Ótimo. Você quer usar diferentes bancos de dados para ajustar melhor o comportamento de um componente? Nós temos a tecnologia para fazer isso.

Claro que, porque você pode fazer algo, não significa que você deva fazê-lo – mas particionar o seu sistema desta forma significa que você tem a opção.

Times construindo microsserviços também preferem uma abordagem diferente aos padrões. Ao invés de usar um conjunto definido de padrões escritos em algum papel, eles preferem a idéia de produzir ferramentas úteis que outros desenvolvedores possam usar para resolver problemas similares aos que eles têm enfrentado. Estas ferramentas são extraídas geralmente das próprias implementações e compartilhadas com um grupo maior, algumas vezes usando modelos open sources públicos. Agora que git e github se tornaram o sistema de controle de versão padrão, práticas open sources se tornaram mais e mais comuns nas estruturas internas das empresas.

Netflix é um bom exemplo de uma organização que segue esta filosofia. Compartilhando códigos úteis e acima de tudo, testados em batalha, eles encorajam outros desenvolvedores resolverem problemas similares de forma similar e ainda deixa a porta aberta para adotar abordagens diferentes caso se queira. Bibliotecas compartilhadas tendem a focar em problemas comuns de armazenagem de dados, comunicação entre processos e, como discutiremos logo mais abaixo, automação da infraestrutura.

Para a comunidade que usa microsserviços, serviços que possuem muitas responsabilidades são particularmente indesejados. Isso não significa que essa comunidade não valorize os padrões de contratos entre serviços. Pelo contrário, uma vez usando microsserviços, existem muito mais contratos. O que acontece é que a comunidade está simplesmente procurando uma maneira diferente de gerenciar as responsabilidades em seus serviços. Padrões como Tolerant Reader e o Consumer-Driven Contracts são freqüentemente aplicados aos microsserviços. Estes padrões de contrato entre serviços permitem que tudo se desenvolva de maneira independente. Executar o pattern Consumer Driven como parte da sua aplicação aumenta a confiança e permite um feedback rápido sobre o funcionamento do seus serviços. Conhecemos um time na Austrália que procura construir novos serviços usando o padrão Consumer Driven. Eles usam ferramentas simples que os permitem criar a definição de um serviço. E isto se torna parte da aplicação de maneira automática, antes mesmo que o código do serviço tenha sido sequer criado. O serviço é então construído até o ponto que o contrato venha ser satisfeito, uma forma elegante para evitar o problema

de 'YAGNI' (é um princípio do Extreme Programming e uma exortação para não adicionar features até você saber que precisa delas) ao construir um novo software. Estas técnicas e as ferramentas criadas ao redor delas limitam a necessidade de um controle central entre os contratos ao diminuir o acoplamento entre os serviços.

Talvez o apogeu desta governança descentralizada é a cultura construa/ponha em execução popularizada pela Amazon. Times são responsáveis por todos os aspectos do software que eles constroem, incluindo a operação na escala 24/7. Este tipo de responsabilidade definitivamente não é uma norma, mas temos visto mais e mais companhias forçando esta tarefa para as equipes de desenvolvimento. Netflix é outra organização que tem adotado esta cultura. Estar acordado toda noite até as 3 horas da madrugada dando suporte a produção é certamente uma forma poderosa de incentivar o foco na qualidade ao escrever o seu código. Estas idéias estão o mais longe possível da forma tradicional de governança centralizada.

Microsserviços e SOA

Quando falamos de microsserviços uma questão comum é se isso é simplesmente um tipo de Arquitetura Orientada a Serviços (SOA) que nós vimos uma década atrás. Existe mérito nessa dúvida, porque microsserviços é um estilo muito similar ao que os defensores do SOA pregam. O problema é que SOA significa muitas coisas diferentes, e na maior parte do tempo que vamos atrás de algo chamado "SOA", encontramos diferenças significantes ao estilo que temos descrito aqui, devido ao foco nas ESBs usados para integrar aplicações monolíticas.

Temos visto muitas implementações remendadas de orientação a serviços – que vai desde a tendência em ocultar a complexidade das ESBs com longas e fracassadas iniciativas que custam milhões (e entregam nenhum valor) ou mesmo apresentam estruturas centralizadas que inibem ativamente mudanças, sendo algumas das vezes difícil de enxergar até que esses problemas ocorram.

Certamente muitas das técnicas em uso pelos praticantes de microsserviços cresceram das experiências de integração entre serviços feitas por desenvolvedores de grandes organizações. O padrão Tolerant Reader é um exemplo disso - seja o mais tolerante possível ao ler dados de um serviço, se você está consumindo um arquivo XML, use apenas os elementos de que precisa, ignore tudo o que não precisa. Iniciativas para usar a Web também contribuíram: o uso de protocolos simples é uma outra abordagem vinda destas experiências, que é uma reação contrária aos padrões centralizadores que alcançaram uma complexidade que é, francamente, de tirar o fôlego (Sempre que você precisa de uma analogia para definir outras abstrações você sabe que está em um grande problema).

Estas práticas comuns do SOA têm levado a alguns defensores de microsserviços a rejeitarem a denominação de SOA completamente, embora outros considerem microsserviços como uma forma de SOA – talvez a orientação a serviços feita da forma certa. De qualquer forma, o fato de que SOA significa diversas coisas diferentes nos faz pensar que é bom ter um termo que defina melhor esta forma arquitetural que estamos descrevendo.

Muitas linguagens, muitas opções

O crescimento da JVM como uma plataforma é justamente o exemplo mais recentemente da mistura entre linguagens dentro de uma plataforma comum. Por décadas, tem sido uma prática comum migrar para uma linguagem de mais alto nível para obter vantagens de abstrações de mais alto nível. Por sua vez, delega-se as linguagens de baixo nível a escrita de códigos mais sensíveis a performance. Entretanto, muitos softwares monolíticos não precisam deste nível de otimização de performance ou sequer DSLs e altos níveis de abstração combinadas. Pelo contrário, softwares monolíticos geralmente usam uma única linguagem e tendem a limitar o número de tecnologias em uso.

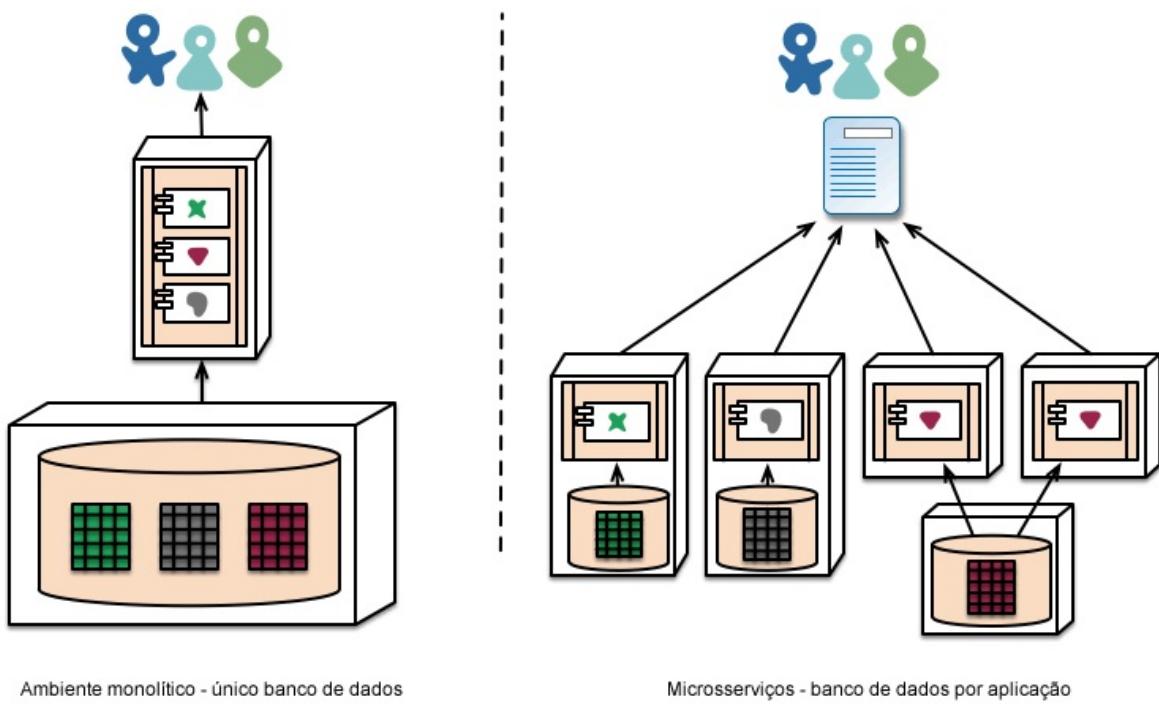
Administração descentralizada de dados

A descentralização da administração de dados pode ser feita de diferentes formas. De forma mais abstrata, isto significa que o modelo conceitual de domínio pode ser diferente entre os sistemas. Este cenário é um problema comum ao fazer integração em uma empresa grande, onde a visão que o setor de vendas tem de um cliente pode ser

diferente do ponto de vista do suporte. Algumas coisas que são chamadas de cliente na visão de vendas, podem nem sequer parecer na visualização do suporte. Objetos podem ter diferentes atributos e (pior) atributos em comum, com diferentes significados.

Este é um problema comum entre aplicações, mas podem ocorrer também dentro das aplicações, particularmente quando estas aplicações são divididas em componentes separados. Uma forma útil de pensar a respeito disto é a noção Domain Driven Design de Bounded Context. O DDD divide um domínio complexo em múltiplos contextos limitados e mapeia o relacionamento entre eles. Este processo é útil para ambas arquiteturas monolíticas e de microsserviços, mas existe uma relação natural entre microsserviços e os limites de um contexto que conforme descrevemos anteriormente, reforçam as separações.

Assim como microsserviços descentralizam decisões sobre os modelos conceituais, eles também descentralizam decisões sobre o armazenamento de dados. Enquanto aplicações monolíticas preferem uma única base de dados lógica para a persistência de dados, empresas frequentemente preferem uma única base de dados para uma variedade de aplicações – e muitas destas decisões são dirigidas por modelos comerciais vendidos através de licenças. Microsserviços preferem permitir que cada serviço gerencie sua própria base de dados, quer através de diferentes instâncias usando a mesma tecnologia de banco de dados, ou até mesmo usando diferentes sistemas de banco de dados – uma abordagem chamada Polyglot Persistence. Você pode usar uma persistência poliglota em uma aplicação monolítica, mas isso aparece com mais freqüência com microsserviços.



A responsabilidade descentralizada para os dados através dos microsserviços tem implicações para a administração de atualizações. A abordagem comum para lidar com atualizações tem sido usar transações para garantir a atualização de múltiplos recursos. Esta abordagem é usada freqüentemente em estruturas monolíticas.

O uso de transações como estas ajuda com a consistência, mas impõem um acoplamento temporário significante, que é problemático quando existem múltiplos serviços. Transações distribuídas são nitidamente difíceis de implementar e como consequência, arquiteturas em microsserviços enfatizam a coordenação sem transação entre serviços, com o reconhecimento explícito que consistência pode ser somente eventual e os problemas gerados por isto são lidados com operações que compensem esta questão.

A forma de gerenciar estas inconsistências é um novo desafio para muitos times de desenvolvimento. Geralmente as empresas lidam com um certo grau de inconsistência com o objetivo de responder rapidamente a sua demanda, possuindo algum processo para lidar com erros. Esta troca é vantajosa enquanto o custo de corrigir os erros for menor que o custo da perda gerada por ter uma consistência maior.

Padrões amplamente testados e padrões impostos

Os times que usam microsserviços tendem a desestimular o uso dos padrões simplesmente impostos pelos grupos de arquiteturas enterprise, e tendem a usar alegremente (e até evangelizar) o uso de padrões abertos tais como HTTP, ATOM e outros micro formatos.

Adiferença chave é como os padrões são criados e como eles são propostos. Padrões gerenciados por grupos como o IETF somente se tornam padrões quando existirem diversas implementações no ar e projetos open-source bem-sucedidos.

Estes padrões formam um mundo a parte de muitos no ambiente corporativo, que são freqüentemente desenvolvidos por grupos que tem pouca experiência em programação, ou são fortemente influenciados por vendedores.

Automação da Infraestrutura

Técnicas para automação da infra-estrutura tiveram grande evolução nos últimos anos – como a computação na nuvem e a própria AWS em particular, que permite reduzir a complexidade de construir, publicar e operar microsserviços.

Muitos dos produtos ou sistemas construídos com microsserviços têm sido feitos por times com uma extensa experiência em Continuos Delivery e seu precursor, a Integração Contínua. Times que constroem software desta forma fazem uso intenso das técnicas de automação de infra-estrutura. A imagem abaixo ilustra um processo de build:

- Figura 5: um fluxo básico de build

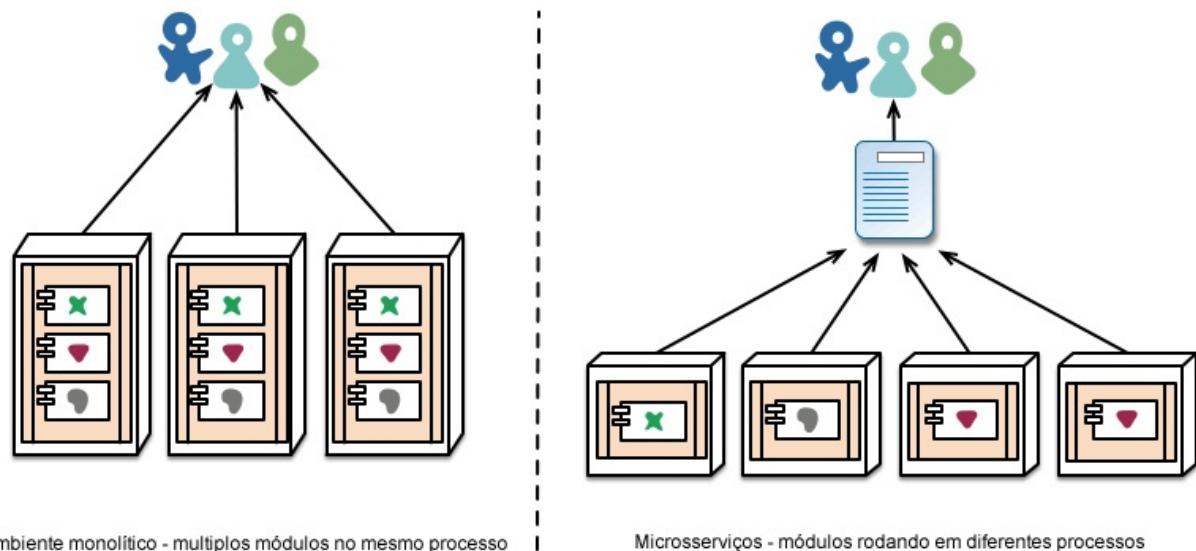


Uma vez que este não é um artigo de Continuos Delivery, vamos chamar a atenção para somente algumas características chave aqui. Nós queremos o máximo de confiança possível que nosso software esteja funcionando, então rodamos diversos testes automatizados. Publicar um software funcionando significa que automatizamos a implantação para cada novo ambiente.

Uma aplicação monolítica pode ser construída, testada e publicada através destes ambientes facilmente. Você percebe que uma vez automatizadas as etapas que vão até publicação em produção de um aplicação monolítica, publicar mais aplicações não assusta mais. E lembrando que um dos objetivos do Continuous Delivery é fazer que o processo de publicação se torne algo chato – indiferente de ser uma ou três aplicações, uma vez que isto continue sendo chato, não há problemas.

Outra área onde nós vemos times usando uma extensa automação da infra-estrutura é a administração de microsserviços em produção. Em contraste da nossa afirmação acima, que quando o processo de publicação é chato não tem muita diferença entre aplicações monolíticas e microsserviços, o panorama operacional para cada tipo é bem diferente.

- Figura 6: A forma de publicação freqüentemente é diferente.



Facilite fazer a coisa correta

Um efeito colateral que encontramos no aumento de automação (como uma consequência da entrega e publicação contínua) é a criação de ferramentas úteis que ajudem os desenvolvedores e o pessoal da operação. Ferramentas para a criação de artefatos, administração de bases de código, levantamento de serviços ou de monitoramento de logs são muito comuns agora. O melhor exemplo na internet é provavelmente o conjunto de ferramentas open source da Netflix, mas existem outras incluindo o Dropwizard, que temos usado extensamente.

Projetado para a falha

Uma consequência do uso de serviços como componentes é que as aplicações precisam ser desenhadas de maneira que possam tolerar a falha dos serviços. Qualquer chamada de serviço poderá falhar devido a uma indisponibilidade do mesmo, e o cliente precisa responder a isso da maneira mais tranquila possível. Isto é uma desvantagem em comparação ao design monolítico porque introduz uma complexidade adicional ao ter que administrar esta situação. Como consequência, os times que usam microsserviços constantemente tem que refletir sobre como as falhas dos serviços podem afetar a experiência do usuário. O Simian Army da Netflix leva os serviços e até mesmos os datacenters a falha durante o dia para testar tanto a resiliência da aplicação quanto o monitoramento.

Esta forma de testar de maneira automática o ambiente em produção pode ser o suficiente para dar ao pessoal de suporte o tipo de arrepios que precedem a uma semana intensa de trabalho. Não queremos dizer que os padrões de arquiteturas monolíticas não são capazes de ter configurações sofisticadas de monitoramento – somente afirmamos que isto é menos comum em nossa experiência.

Uma vez que os serviços podem falhar a qualquer momento, é importante ser capaz de detectar falhas rapidamente e, se possível, restaurar o serviço automaticamente. Microsserviços põe bastante ênfase no monitoramento em tempo real, checando elementos de arquitetura (por exemplo, quantas requisições por segundo o banco de dados está tendo) e métricas relevantes ao negócio (como quantos pedidos por minutos são recebidos). O monitoramento semântico pode prover um alerta antecipado de algo indo mal e levar os times de desenvolvimento a investigar o caso.

Isto é particularmente importante para uma arquitetura em microsserviços porque elas dão preferência a adaptação e a colaboração entre eventos, que são resultados de um comportamento de emergência. Enquanto muitos especialistas se alegram ao descobrir situações de emergência por acaso, a verdade é que esse comportamento pode levar a coisas ruins. Monitoramento é vital para identificar rapidamente um comportamento errado e assim conseguir corrigí-lo.

Aplicações monolíticas podem ser construídas para serem transparentes como um microsserviço – na verdade, elas deveriam ser. A diferença é que você precisa obrigatoriamente saber quando os serviços que rodam em diferentes processos são desconectados. Com bibliotecas dentro do mesmo processo este tipo de transparência é menos útil.

Times que praticam microsserviços esperam ver monitoramentos sofisticados e configurações individuais de log para cada serviço, em dashboards exibindo status de execução (up/down) e uma variedade de métricas relevantes para a operação do negócio. Detalhes sobre o status dos disjuntores (circuit breakers), taxa de transferência atual e latência são outros exemplos que freqüentemente encontramos no mundo real.

O Circuit Breaker e o código pronto para a produção

O padrão Circuit Breaker (disjuntor) aparece no Release It! junto com outros padrões como o Bulkhead e o Timeout. Implementados juntos, estes padrões são cruciais quando construímos aplicações que se comunicam. Este post no blog da Netflix realiza um grande trabalho ao explicar como eles aplicam estes padrões.

Considerando chamadas síncronas prejudiciais

Sempre que você tem um certo número de chamadas síncronas entre serviços você vai encontrar um efeito multiplicador na lentidão. Esta lentidão no seu sistema é simplesmente o produto da lentidão de componentes individuais. Você deve tomar uma decisão: transformar suas chamadas em assíncronas ou gerenciar esta lentidão. No www.guardian.co.uk eles implementaram uma regra simples em sua nova plataforma – uma chamada síncrona por requisição do usuário, enquanto na Netflix, seu redesign na API transformou toda a criação de APIs em recursos assíncronos.

Design que evolui

Times que trabalham com microsserviços geralmente possuem uma experiência em desenhar aplicações para evoluírem e enxergam na decomposição dos serviços uma ferramenta avançada que permite controlar mudanças em suas aplicações sem necessidade de evitá-las. Controlar mudanças não significa necessariamente reduzi-las – com atitudes e ferramentas corretas você pode fazer frequentes, rápidas e precisas mudanças em seu software.

Sempre que você quebra um sistema em componentes, você se vê diante da decisão de como dividi-lo em partes – quais são os princípios em que vamos decidir esta divisão da aplicação? Apropriedade chave de um componente é a noção da sua atualização acontecer de maneira independente – o que implica em olharmos para pontos onde imaginamos reescrever o componente sem afetar nada a sua volta. Desenvolvedores que utilizam microsserviços avançam ainda mais neste ponto ao esperarem explicitamente que muitos serviços sejam destruídos no futuro, ao invés de ser permanentes a longo prazo.

O site do The Guardian é um bom exemplo de como uma aplicação foi desenhada e construída como uma aplicação monolítica, mas evoluiu em direção aos microsserviços. O sistema monolítico ainda existe como o coração do website, mas eles preferem adicionar novas features usando microsserviços que consomem a API no sistema monolítico. Esta abordagem é particularmente útil para features que são herdadas temporariamente, como páginas específicas de um evento esportivo. Tal parte do site pode ser facilmente criada usando linguagens de desenvolvimento rápido e removidas uma vez que o evento termine. Temos visto abordagens similares em uma instituição financeira onde novos serviços são adicionados para uma oportunidade de mercado e descartados após alguns meses ou mesmo semanas.

Esta ênfase de que os serviços possam ser substituíveis é, em um caso especial, um princípio mais amplo de design que obtém sua modularidade através do padrão de mudança. Você deseja manter coisas que mudam ao mesmo tempo dentro do mesmo módulo. Partes de um sistema que muda raramente devem estar em diferentes serviços. Se você se encontrar repetidamente mudando dois serviços ao mesmo tempo, isto é um sinal que eles devem ser unidos.

Colocar componentes em serviços permite um planejamento de lançamento mais granular. Em um sistema monolítico, qualquer mudança requer uma publicação da aplicação por inteiro. Com microsserviços, entretanto, você precisa somente republicar o serviço que foi modificado. Isto simplifica e agiliza o processo de publicação. Em contrapartida, você tem que se preocupar com as mudanças que possam quebrar os serviços que as consomem. A abordagem tradicional de integração é tentar lidar com esse problema usando versionamento, mas a preferência no mundo dos microsserviços é usar o versionamento como último recurso. Podemos evitar diversos versionamentos ao desenhar serviços para serem tolerantes a mudanças em seus fornecedores o máximo possível.

Microsserviços são o futuro?

Nosso principal objetivo neste artigo é explicar as maiores idéias e princípios por trás dos microsserviços. Ao gastar tempo fazendo isto, nós pensamos claramente que o estilo arquitetural dos microsserviços é algo importante – digno de sérias considerações voltadas para aplicações empresariais. Recentemente construímos diversos sistemas usando este estilo e sabemos de outros que usaram e aprovaram esta abordagem.

Entre aqueles que, até onde temos conhecimento, são de certa forma pioneiros neste estilo arquitetural estão a Amazon, Netflix, The Guardian, o Serviço Digital Governamental do Reino Unido, realstate.com.au e o comparethemarket.com. O circuito de conferências em 2013 foi cheio de exemplos de companhias que caminharam para algo que pode ser classificado como microsserviços – incluindo o Travis CI. Além disso existem diversas organizações que há bastante tempo tem usado algo que chamamos de microsserviços, mas que sequer utilizam este nome (frequentemente isso é chamado de SOA, embora, como falamos, SOA aparece em diversas formas contraditórias.)

Temos a certeza que microsserviços são a futura direção para as arquiteturas de software. Embora nossas experiências tenham sido até então positivas em comparação as aplicações monolíticas, nós estamos conscientes do fato que não se passou tempo o suficiente para um julgamento final.

Em geral, as verdadeiras consequências das suas decisões de arquitetura se tornam evidentes somente anos depois que você as tomou. Temos visto projetos onde um bom time, com um forte desejo por modularidade, que constrói uma arquitetura monolítica que se deteriora com os anos. Muitas pessoas acreditam que este problema é menos provável com microsserviços, desde que os limites dos serviços sejam explícitos. Enquanto não tivermos vistos vários sistemas com idade suficiente, nós não podemos avaliar verdadeiramente quão maduras são as arquiteturas em microsserviços.

Existem certos motivos pelos quais algumas pessoas podem esperar que os microsserviços amadureçam de maneira tímida. Qualquer esforço para componentizar um sistema depende de como o software é adaptado para os componentes. É difícil imaginar exatamente onde o limite de cada componente deva ser definido. Designs que evoluem devem reconhecer a dificuldade de definir corretamente seus limites e por isso a importância de ser fácil refatorá-los.

Outro problema é que caso os componentes não sejam divididos de maneira clara, tudo que você está fazendo é tirando a complexidade de dentro de um componente para as conexões entre eles. E isso não move simplesmente a complexidade de lugar, mas põe em um lugar é que menos explícito e difícil de controlar. É fácil pensar que as coisas são melhores quando você as encontra dentro de um componente pequeno e simples, abrindo mão de ter conexões complexas entre os serviços.

Finalmente, há o fator da maturidade do time. Novas técnicas tendem a ser adotadas por times mais capacitados. Mas uma técnica que é mais efetiva para um time mais capacitado não vai necessariamente funcionar para times menos capazes. Nós vimos muitos casos de times menos capacitados construírem arquiteturas monolíticas bagunçadas, porém, leva tempo para ver o que acontece quando este tipo de bagunça acontece com microsserviços. Um time pobre irá sempre criar um sistema pobre – e é muito difícil dizer se os microsserviços neste caso reduzem a bagunça ou a deixa pior.

Um argumento razoável que ouvimos é que você não deve começar com uma arquitetura em microsserviços. Ao invés de começar com uma arquitetura monolítica, mantenha-a modular, e divida em microsserviços uma vez que a arquitetura monolítica se torne um problema (embora este conselho não seja o ideal, uma vez que uma interface em

processo de mudança não é geralmente uma boa interface para os serviços).

Fonte

- <https://martinfowler.com/articles/microservices.html>

Config Server

Ao desenvolver um aplicativo em nuvem, um problema é manter e distribuir a configuração para nossos serviços. Nós realmente não queremos perder tempo configurando cada ambiente antes de dimensionar nosso serviço horizontalmente ou arriscar violações de segurança embutindo a configuração em nosso aplicativo.

Para resolver isso, vamos consolidar toda a nossa configuração em um único repositório Git e conectá-lo a um aplicativo que gerencia uma configuração para todos os nossos aplicativos.

O Spring Cloud Config Server fornece uma API baseada em recursos HTTP para configuração externa (pares nome-valor ou conteúdo YAML equivalente). O servidor pode ser incorporado em um aplicativo Spring Boot, usando a anotação `@EnableConfigServer`. Consequentemente, a seguinte aplicação é um servidor de configuração:

ConfigServer.java

```
@SpringBootApplication
@EnableConfigServer
public class ConfigServer {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServer.class, args);
    }
}
```

Como todos os aplicativos Spring Boot, ele é executado na porta 8080 por padrão, mas você pode alterá-lo para a porta 8888 mais convencional de várias maneiras. O mais fácil, que também define um repositório de configuração padrão, é executá-lo com `spring.config.name=configserver` (há um `configserver.yml` no jar do Config Server). Outra forma é usar seu próprio `application.properties`, conforme mostrado no exemplo a seguir:

application.properties

```
server.port: 8888
spring.cloud.config.server.git.uri: file://${user.home}/config-repo
```

Onde `${user.home}/config-repo` é um repositório git contendo arquivos YAML e de propriedades.

A lista a seguir mostra uma receita para criar o repositório git no exemplo anterior:

```
$ cd $HOME
$ mkdir config-repo
$ cd config-repo
$ git init .
$ echo info.foo: bar > application.properties
$ git add -A .
$ git commit -m "Add application.properties"
```

Usar o sistema de arquivos local para o seu repositório git é destinado apenas para teste. Você deve usar um servidor para hospedar seus repositórios de configuração em produção.

O clone inicial do seu repositório de configuração pode ser rápido e eficiente se você mantiver apenas arquivos de texto nele. Se você armazenar arquivos binários, especialmente os grandes, poderá ocorrer atrasos na primeira solicitação de configuração ou encontrar erros de falta de memória no servidor.

Repositório de Ambiente

Onde você deve armazenar os dados de configuração para o Config Server? A estratégia que rege esse comportamento é o `EnvironmentRepository`, servindo objetos `Environment`. Este ambiente é uma cópia superficial do domínio do Spring Environment (incluindo `propertySources` como o recurso principal). Os recursos do ambiente são parametrizados por três variáveis:

- `{application}`, que mapeia para `spring.application.name` no lado do cliente;
- `{profile}`, que mapeia para `spring.profiles.active` no cliente (lista separada por vírgulas);
- `{label}`, que é um recurso do lado do servidor que rotula um conjunto "versionado" de arquivos de configuração;

As implementações de repositório geralmente se comportam como um aplicativo Spring Boot, carregando arquivos de configuração de um `spring.config.name` igual ao parâmetro `{application}` e `spring.profiles.active` igual ao parâmetro `{profiles}`. As regras de precedência para perfis também são as mesmas de um aplicativo Spring Boot regular: Perfis ativos têm precedência sobre padrões e, se houver vários perfis, o último ganha (semelhante à inclusão de entradas em um Mapa).

O seguinte aplicativo cliente de amostra possui esta configuração de autoinicialização:

bootstrap.yml

```
spring:
  application:
    name: foo
  profiles:
    active: dev,mysql
```

Como de costume com um aplicativo Spring Boot, essas propriedades também podem ser definidas por variáveis de ambiente ou argumentos de linha de comando.

Se o repositório for baseado em arquivo, o servidor criará um `Environment` a partir de `application.yml` (compartilhado entre todos os clientes) e `foo.yml` (com `foo.yml` tomando precedência). Se os arquivos YAML tiverem documentos dentro deles que apontam para perfis Spring, eles serão aplicados com precedência mais alta (na ordem dos perfis listados). Se houver arquivos YAML (ou propriedades) específicos do perfil, eles também serão aplicados com maior precedência do que os padrões. Uma precedência mais alta se traduz em um `PropertySource` listado anteriormente no ambiente. (Essas mesmas regras se aplicam em um aplicativo de inicialização Spring Boot standalone).

Você pode definir `spring.cloud.config.server.accept-empty` como `false` para que o servidor retorne um status HTTP 404, se o aplicativo não for encontrado. Por padrão, esse sinalizador é definido como `true`.

Git Backend

A implementação padrão do `EnvironmentRepository` usa um backend do Git, que é muito conveniente para gerenciar upgrades e ambientes físicos e para alterações de auditoria. Para alterar a localização do repositório, você pode definir a propriedade de configuração `spring.cloud.config.server.git.uri` no Config Server (por exemplo, `application.yml`). Se você configurá-lo com um prefixo `file`, ele deverá funcionar em um repositório local para que você possa começar de forma rápida e fácil sem um servidor. No entanto, nesse caso, o servidor opera diretamente no repositório local sem cloná-lo. Para escalar o Config Server e torná-lo altamente disponível, você precisa ter todas as instâncias do servidor apontando para o mesmo repositório, portanto, apenas um sistema de arquivos compartilhado funcionaria. Mesmo nesse caso, é melhor usar o protocolo ssh: para um repositório de sistema de arquivos compartilhado, para que o servidor possa cloná-lo e usar uma cópia de trabalho local como um cache.

Essa implementação do repositório mapeia o parâmetro `{label}` do recurso HTTP para um rótulo git (ID de commit, nome de branch ou tag).

Para mais opções de backend, consulte https://cloud.spring.io/spring-cloud-config/multi/multi__spring_cloud_config_server.html.

Montando o Servidor

Primeiramente vamos criar na unidade c: o repositório GIT que conterá as configurações.

```
c:\Users\foo>c:  
c:>mkdir config-repo  
c:>cd config-repo  
c:\config-repo>git init .  
Initialized empty Git repository in c:/config-repo/.git/
```

Agora, importe a pasta criada no Spring Tools para facilitar nosso trabalho: File -> Import -> General -> Projects from Folder or Archive e selecione a pasta que acabou de criar.

Para montar o servidor de fato, acesse <https://start.spring.io/>, em Group altere para com.acme, em Artifact digite config-server e nas dependências busque por "Config Server", gire o projeto, faça o download e importe no Spring Tools.

Ajuste o arquivo **application.properties** com os seguintes valores:

application.properties

```
server.port=8888  
spring.cloud.config.server.git.uri=file:///c:/config-repo
```

Ajuste o arquivo **ConfigServerApplication.java** incluindo a anotação `@EnableConfigServer`:

ConfigServerApplication.java

```
package com.acme.configserver;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.config.server.EnableConfigServer;  
  
@SpringBootApplication  
@EnableConfigServer  
public class ConfigServerApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(ConfigServerApplication.class, args);  
    }  
}
```

Como este servidor não tem nenhuma página é interessante adicionar um arquivo de `index.html` em `/src/main/resources/static/index.html` para que possamos identificar se está ele está "no ar":

index.html

```
<!DOCTYPE html>  
<html>  
<head>  
<meta charset="UTF-8">  
<title>Config Server</title>  
</head>  
<body>  
    <h1>Config Server</h1>  
</body>  
</html>
```

Inicie o servidor e acesse o endereço <http://localhost:8888>, a página criada deve ser exibida.

Copiando as configurações de livro-service para o repositório de configurações

Mova o arquivo `application.properties` do projeto **livro-service** para a pasta do repositório de configurações - `config-repo` e renomeie para `livro-service.properties`.

Não se esqueça de comitar as alterações no repositório:

```
c:\config-repo>git add .
c:\config-repo>git commit -m "configurações"
```

Agora, acesse a URL <http://localhost:8888/livro-service/default> e verifique que as configurações da aplicação estão disponíveis para consulta.

Spring Cloud Config Client

Um aplicativo Spring Boot pode aproveitar imediatamente o Spring Config Server (ou outras fontes de propriedades externas fornecidas pelo desenvolvedor do aplicativo). Ele também ganha alguns recursos úteis adicionais relacionados a eventos de mudança do ambiente.

O comportamento padrão de qualquer aplicativo que tenha o Spring Cloud Config Client no classpath é o seguinte: Quando um Config Client é iniciado, ele se conecta ao Config Server (por meio da propriedade de configuração `bootstrap` `spring.cloud.config.uri`) e inicializa o Spring Environment com fontes de propriedades remotas.

O resultado é que todos os aplicativos clientes que desejam consumir o Config Server precisam de um `bootstrap.properties` (ou uma variável de ambiente) com o endereço do servidor definido em `spring.cloud.config.uri` (o padrão é "<http://localhost:8888>").

Adicionando Dependências ao `pom.xml` de Nossa Aplicação

Vamos incluir as dependências `spring-cloud-starter-config` em nosso `pom.xml`, também será necessário incluir o gerenciador de dependências `spring-cloud-dependencies` e a propriedade `spring-cloud.version`:

- `pom.xml`

```

<!-- Código anterior omitido -->

<properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>Greenwich.RELEASE</spring-cloud.version>
</properties>

<dependencies>

    <!-- Dependências atuais omitidas -->

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<!-- Código posterior omitido -->

```

Precisamos também criar o arquivo `/src/main/resources/bootstrap.properties` que conterá o endereço do servidor de configuração utilizado, o profile atual e o nome da aplicação:

bootstrap.properties

```

spring.cloud.config.name=livro-service
spring.cloud.config.uri=http://localhost:8888
spring.profiles.active=default

```

Para temos certeza de que nossa configuração está sendo corretamente lida do Config Server, crie uma cópia arquivo **livro-service.properties** na pasta **config-repo** com o nome **livro-service-qa.properties** e, neste arquivo, inclua uma configuração de porta diferente para nossa aplicação: `server.port=9090`

Por fim, altere o arquivo de bootstrap de livro-service para o profile `qa` e veja se a aplicação subirá agora com a porta 9090:

bootstrap.properties

```

spring.cloud.config.name=livro-service
spring.cloud.config.uri=http://localhost:8888
spring.profiles.active=qa

```

Fontes

- <https://www.baeldung.com/spring-cloud-bootstrapping>
- https://cloud.spring.io/spring-cloud-config/multi/multi__spring_cloud_config_server.html

Service Discovery com Consul

A descoberta de serviço é um dos princípios fundamentais de uma arquitetura baseada em microsserviço. Tentar configurar manualmente cada cliente ou alguma forma de convenção pode ser muito difícil de fazer e pode ser muito frágil. O Consul fornece serviços de descoberta de serviços por meio de uma API HTTP e DNS. O Spring Cloud Consul aproveita a API HTTP para registro e descoberta de serviços. Isso não impede que aplicativos que não sejam da Spring Cloud aproveitem a interface do DNS. Os servidores da Consul Agents são executados em um cluster que se comunica por meio de um protocolo gossip (peer-to-peer communication) e usa o protocolo de consenso do Raft.

Como ativar

Para ativar o Consul Service Discovery é necessário incluir a dependência `spring-cloud-starter-consul-discovery`.

Registrando com o Consul

Quando um cliente se registra no Consul, ele fornece metadados sobre si mesmo, como host e porta, id, nome e tags. Uma verificação HTTP é criada por padrão que o Consul atinge o end-point /health a cada 10 segundos. Se a verificação de integridade falhar, a instância do serviço será marcada como crítica.

Exemplo do cliente Consul:

```
@SpringBootApplication
@RestController
public class Application {

    @RequestMapping("/")
    public String home () {
        return "Olá mundo";
    }

    public static void main (String [] args) {
        new SpringApplicationBuilder (Application.class) .web (true) .run (args);
    }

}
```

Se o cliente Consul estiver localizado em algum lugar diferente de localhost:8500, a configuração é necessária para localizar o cliente. Exemplo:

`application.yml`.

```
spring:
  cloud:
    consul:
      host: localhost
      port: 8500
```

Cuidado: Se você usar o Spring Cloud Consul Config, os valores acima precisarão ser colocados em `bootstrap.yml` em vez de `application.yml`.

O nome do serviço padrão, o ID da instância e a porta, obtidos do Ambiente, são `${spring.application.name}`, o Spring Context ID e `${server.port}` respectivamente.

Para desativar o Consul Discovery Client, você pode definir `spring.cloud.consul.discovery.enabled` como `false`.

Para desativar o registro de serviço, você pode definir `spring.cloud.consul.discovery.register` como `false`.

Verificação de Saúde HTTP

A verificação de integridade de uma instância Consul é padronizada como "/health", que é a localização padrão do endpoint em um aplicativo Spring Boot Actuator.

Iniciando o Consul

Para iniciar uma instância local do Consul, execute o seguinte comando:

```
> consul agent -dev -bind=127.0.0.1
```

Você pode verificar se o serviço está operacional consultando o endereço <http://localhost:8500>

Adicionando Dependências ao pom.xml de Nossa Aplicação

Vamos incluir as dependências `spring-cloud-starter-consul-discovery` em nosso `pom.xml`:

- `pom.xml`

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

Para que nossa aplicação consiga se registrar junto ao Consul ela precisa ter um ID, vamos adicionar mais uma configuração:

bootstrap.properties

```
spring.cloud.consul.discovery.instanceId=${spring.cloud.config.name}:${random.value}
```

Habilitando o Discovery Client

Precisamos incluir a anotação `@EnableDiscoveryClient` à classe `LivroServiceApplication` para habilitar a aplicação como cliente de um serviço de discovery.

LivroServiceApplication.java

```
// Código anterior omitido

@SpringBootApplication
// Novidade aqui
@EnableDiscoveryClient
public class LivroServiceApplication {
    // Código atual omitido
}
```

Tudo certo, execute agora a aplicação e consulte o status da mesma na tela de gerenciamento do Consul <http://localhost:8500>, hum, um **Service 'application' check** está com erro, o problema é que não colocamos o Spring Boot Actuator em nossa aplicação, faremos isso em seguida.

Service Boot Actuator

Em suma, o Actuator traz recursos prontos para produção para o nosso aplicativo.

Monitorar nosso aplicativo, coletar métricas, entender o tráfego ou o estado do nosso banco de dados torna-se trivial com essa dependência.

O principal benefício desta biblioteca é que podemos obter ferramentas de nível de produção sem ter que implementar esses recursos por conta própria.

O Actuator é usado principalmente para expor informações operacionais sobre o aplicativo em execução - saúde, métricas, informações, dump, env etc. Ele usa nós de extremidade HTTP ou JMX para nos permitir interagir com ele.

Uma vez que esta dependência esteja no caminho de classe, vários end-points estarão disponíveis automaticamente. Como na maioria dos módulos Spring, podemos facilmente configurá-lo ou estendê-lo de várias maneiras.

Iniciando

Para ativar o Spring Boot Actuator, precisamos apenas adicionar a dependência do `spring-boot-actuator` ao nosso `pom.xml`:

- `pom.xml`

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Habilitando os End-points

O Actuator vem com a maioria dos end-points desativados. Assim, os únicos dois disponíveis por padrão são `/health` e `/info`.

Se quisermos habilitar todos eles, poderíamos definir na configuração `management.endpoints.web.exposure.include=*`. Alternativamente, poderíamos listar terminais que deveriam ser habilitados programaticamente.

End-points predefinidos

Vamos dar uma olhada em alguns end-points disponíveis:

- `/auditevents` - lista eventos relacionados à auditoria de segurança, como login/logout do usuário;
- `/beans` - retorna todos os beans disponíveis em nosso BeanFactory;
- `/conditions` - anteriormente conhecido como `/autoconfig`, cria um relatório das condições de configuração automática;
- `/configprops` - nos permite buscar todos os beans `@ConfigurationProperties`;
- `/env` - retorna as propriedades atuais do ambiente;
- `/flyway` - fornece detalhes sobre nossas migrações de banco de dados Flyway;
- `/health` - resume o status de integridade de nosso aplicativo;
- `/heapdump` - cria e retorna um dump de heap da JVM usada pelo nosso aplicativo;
- `/info` - retorna informações gerais. Pode ser dados personalizados, informações de compilação ou detalhes sobre o último commit;
- `/liquibase` - comporta-se como `/flyway` mas para Liquibase;
- `/logfile` - retorna logs comuns de aplicativos;

- /loggers - nos permite consultar e modificar o nível de log do nosso aplicativo;
- /metrics - detalha as métricas do nosso aplicativo;
- /prometheus - retorna métricas como a anterior, mas formatada para funcionar com um servidor Prometheus;
- /scheduledtasks - fornece detalhes sobre cada tarefa agendada dentro do nosso aplicativo;
- /sessions - lista sessões HTTP, uma vez que estamos usando o Spring Session;
- /shutdown - executa um desligamento normal do aplicativo;
- /threaddump - realiza um dump da thread da JVM;

Habilitando o Spring Boot Actuator em nossa aplicação

Conforme descrito, devemos incluir a dependência `spring-boot-starter-actuator` no `pom.xml` da aplicação e a configuração `management.endpoints.web.exposure.include=*` no arquivo de configuração da aplicação.

Faça estes ajustes, reinicie a aplicação e veja que agora ela está passando nos Health Checks do Consul.

Ajustando o config-server para se registrar ao Consul

Podemos aproveitar e registrar nosso config server junto ao Consul.

Primeiramente, incluiremos as dependências necessárias:

pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Agora, devemos ajustar o arquivo de configuração para que se comunique com o Consul:

application.properties

```
server.port=8888

spring.cloud.config.server.git.uri=file:///c:/config-repo
spring.cloud.config.name=config-server

spring.cloud.consul.host=localhost
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.instanceId=${spring.cloud.config.name}:${random.value}
spring.cloud.consul.discovery.serviceName=${spring.cloud.config.name}

management.endpoints.web.exposure.include=*
```

O último passo é adicionar a anotação na classe principal da aplicação:

ConfigServerApplication.java

```

package com.acme.configserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
@EnableDiscoveryClient
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}

```

Ótimo, agora podemos compilar e iniciar o config-server e consultar no Consul seu status.

Os arquivos de configuração de livro-service devem estar desta maneira:

bootstrap.properties

```

spring.cloud.config.name=livro-service
spring.profiles.active=default

spring.cloud.config.uri=http://localhost:8888

spring.cloud.consul.host=localhost
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.instanceId=${spring.cloud.config.name}:${random.value}
spring.cloud.consul.discovery.serviceName=${spring.cloud.config.name}

management.endpoints.web.exposure.include=*

```

config-repo/livro-service.properties

```

server.port=8080

spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

spring.cache.type=redis
spring.cache.redis.time-to-live=5000
spring.redis.host=localhost
spring.redis.port=6379

```

Agora, ao abrir a interface de gerenciamento do Consul (<http://localhost:8500/>) devem ser listados 3 serviços ativos, `config-server`, `consul` e `livro-service`.

Porém, se o Consul é um serviço de descoberta, ele não poderia então ser utilizado para que a aplicação `livro-service` "descubra" o endereço do `config-server`?

Se você usar uma implementação do DiscoveryClient, como Spring Cloud Consul, poderá ter o registro do Config Server com o Discovery Service. No entanto, no modo padrão "Config First", os clientes não podem utilizar o registro.

Se você preferir usar o DiscoveryClient para localizar o Config Server. O resultado é que todos os aplicativos clientes precisam de um `bootstrap.yml` (ou uma variável de ambiente) com a configuração de descoberta apropriada. O preço para usar esta opção é uma ida e volta extra da rede na inicialização, para localizar o registro do serviço. O benefício é que, desde que o Discovery Service seja um ponto fixo, o Config Server pode alterar de endereço. O ID de serviço padrão é `configserver`, mas você pode alterá-lo no cliente configurando `spring.cloud.config.discovery.serviceId` (e no servidor, da maneira usual para um serviço, como definindo `spring.application.name`).

Todas as implementações do cliente de descoberta suportam algum tipo de mapa de metadados. Algumas propriedades adicionais do Config Server podem precisar ser configuradas em seus metadados de registro de serviço para que os clientes possam se conectar corretamente. Se o Config Server estiver protegido com o HTTP Basic, você poderá configurar as credenciais como usuário e senha. Além disso, se o Config Server tiver um caminho de contexto, você poderá definir o configPath.

Vamos alterar novamente o arquivo de configuração de `livro-service`:

bootstrap.properties

```
spring.cloud.config.name=livro-service
spring.profiles.active=default

spring.cloud.config.discovery.serviceId=config-server
spring.cloud.config.fail-fast=true

spring.cloud.consul.host=localhost
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.instanceId=${spring.cloud.config.name}:${random.value}
spring.cloud.consul.discovery.serviceName=${spring.cloud.config.name}

management.endpoints.web.exposure.include=*
```

Em alguns casos, você pode querer falhar a inicialização de um serviço se não puder se conectar ao Config Server. Se esse for o comportamento desejado, defina a propriedade de configuração de inicialização como `spring.cloud.config.fail-fast=true` para fazer com que o cliente pare com uma Exceção.

Com tudo configurado, nossa aplicação deve estar funcionando normalmente.

build info

Um toque final que pode ser adicionado é a exposição das informações de build das aplicações através do Actuator, para isso, altere o `pom.xml`

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>

      <!-- Novidade aqui -->
      <executions>
        <execution>
          <goals>
            <goal>build-info</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Agora construa e execute novamente o projeto, acessando a url `http://localhost:8080/actuator/info` devem estar disponíveis as informações de build.

Cache com Redis

Introdução ao Redis

Redis é uma **estrutura de dados em memória** open source (licenciado pelo BSD), usado como banco de dados, cache e message broker. Ele suporta estruturas de dados como strings, hashes, lists, sets, sorted sets com range queries, bitmaps, hyperloglogs, geospatial indexes com radius queries e streams. O Redis possui replicação integrada, suporte a script Lua, transações e diferentes níveis de persistência em disco, e fornece alta disponibilidade via Redis Sentinel e particionamento automático com o Redis Cluster.

Você pode executar **operações atômicas** nesses dados, como anexar a uma string; incrementar o valor em um hash; incluir um elemento em uma lista; calcular um conjunto de intersecção, união e diferença; ou obter o membro com a classificação mais alta em um conjunto classificado.

Para alcançar seu excelente desempenho, o Redis trabalha com um conjunto de **dados na memória**. Dependendo do seu caso de uso, você pode persistir despejando o conjunto de dados no disco de vez em quando ou anexando cada comando a um log. A persistência pode ser opcionalmente desativada, se você precisar apenas de um cache em memória rico em recursos e em rede.

O Redis também suporta replicação assíncrona master-slave para a configuração, com primeira sincronização não bloqueante muito rápida, reconexão automática com ressincronização parcial em divisão de rede.

Outras características incluem:

- Transações
- Pub/Sub
- Scripting Lua
- Chaves com tempo de vida limitado
- Failover automático

Você pode usar o Redis com a maioria das linguagens de programação.

O Redis é escrito em ANSI C e funciona na maioria dos sistemas POSIX, como Linux, *BSD, OS X, sem dependências externas. Linux e OSX são os dois sistemas operacionais nos quais o Redis é desenvolvido e mais testado, e recomendamos o uso do Linux para implantação. O Redis pode funcionar em sistemas derivados do Solaris, como o SmartOS, mas o suporte é o melhor esforço. Não há suporte oficial para o Windows, mas a Microsoft desenvolve e mantém um port Win-64 do Redis.

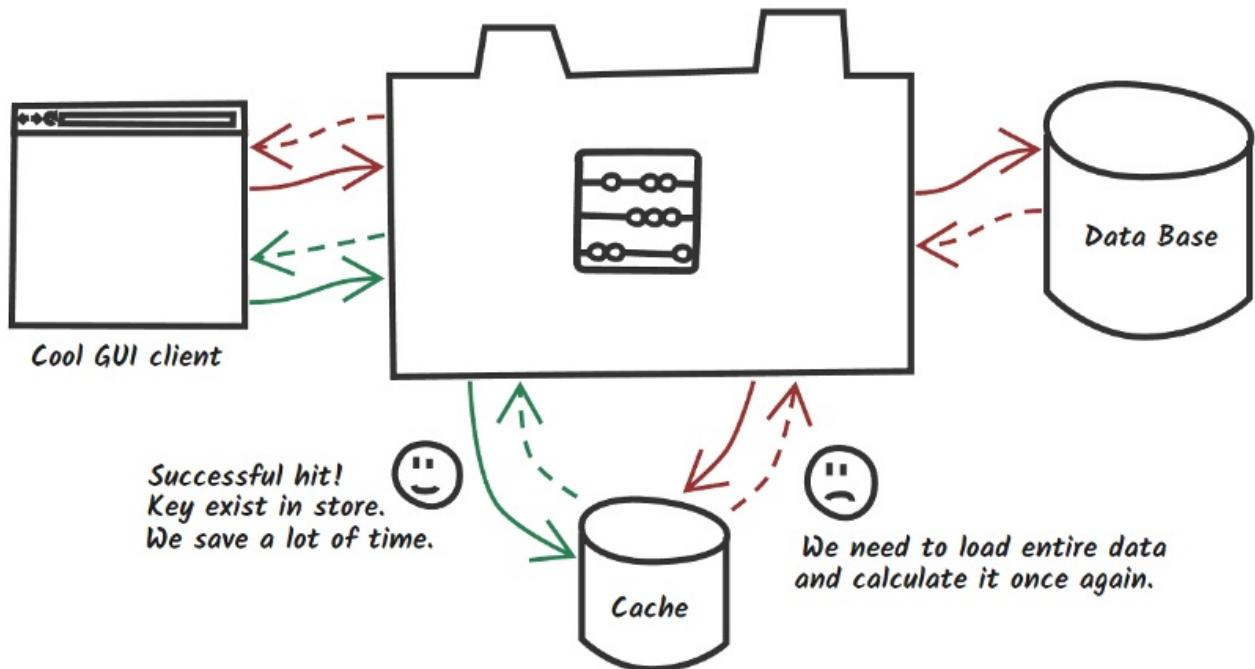
Vantagens

Seguem-se algumas vantagens do Redis.

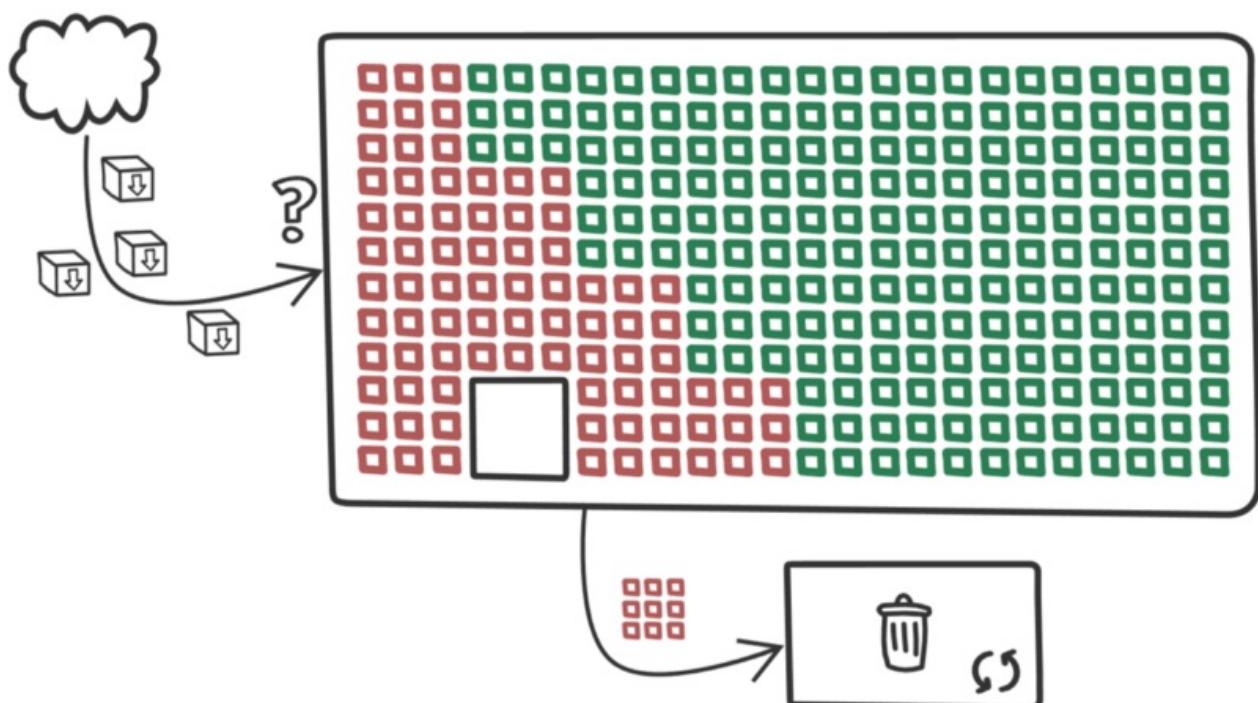
- **Excepcionalmente rápido** - o Redis é muito rápido e pode realizar cerca de 110000 SETs por segundo, cerca de 81000 GETs por segundo;
- **Suporta tipos de dados avançados** - O Redis suporta nativamente a maioria dos tipos de dados que os desenvolvedores já conhecem, como list, set, sorted set, e hashes. Isso facilita a solução de diversos problemas, pois sabemos qual problema pode ser melhor tratado por qual tipo de dados;
- **As operações são atômicas** - Todas as operações do Redis são atômicas, o que garante que, se dois clientes acessarem simultaneamente, o servidor Redis receberá o valor atualizado;
- **Ferramenta multiserviços** - Redis é uma ferramenta com vários utilitários e pode ser usada em vários casos de uso, como armazenamento em cache, filas de mensagens (Redis suporta nativamente Pub/Sub), quaisquer dados de curta duração em seu aplicativo, como web sessões de aplicativos, contagens de acessos a páginas da Web, etc.

Cache Spring Boot com Redis

Cada um de nós encontrou a situação quando uma aplicação funciona lentamente. Mesmo o melhor código sofrerá com carga alta. O cache pode ser uma maneira rápida e relativamente barata de aumentar o desempenho e reduzir o tempo de resposta.



De maneira simplificada, o armazenamento em cache é uma das estratégias de desempenho utilizada quando nos deparamos com serviços de longa duração. O resultado da invocação pode ser colocado dentro de um armazenamento rápido na memória e usado na próxima vez sem a dispendiosa execução do método. Seguindo o fluxo "verde", você pode perceber que se encontrarmos dados solicitados no cache (chamados "cache hit"), economizamos tempo e recursos. O fluxo vermelho representa o pior cenário (chamado de "cache miss") quando o cache não armazena os dados esperados e você precisa carregá-lo e recalculá-lo desde o início com uma viagem extra para o cache, o que aumenta o tempo de resposta. Então, como este sistema funciona?



Com uma grande simplificação quando novos dados chegam, eles são colocados no primeiro *bucket* vazio, mas quando o cache está cheio, o processo de limpeza é executado de acordo com o algoritmo de evicção selecionado. Alguns dados são salvos pois são usados com muita frequência ou satisfazem outras condições para o algoritmo escolhido. O resto dos dados são candidatos a serem removidos. No mundo ideal, o cache irá despejar apenas dados antigos até encontrar lugar para novos dados. Com o Spring e o Redis, podemos criar um aplicativo simples e considerar como diferentes fatores podem afetar nossa camada de armazenamento em cache.

Instalação

Para podermos utilizar o Redis em nosso projeto, primeiro precisamos instalar seu executável e em seguida ativar o serviço em: Windows -> Serviços -> Redis -> Iniciar

Em seguida, é possível obter informações sobre a instância do Redis em execução pelo utilitário "Redis Client", para isso, inicie a aplicação "Redis Client" e no prompt de comando que se abrirá digite "info":

```
127.0.0.1:6379> info
# Server
redis_version:3.2.100
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:dd26f1f93c5130ee
redis_mode:standalone
os:Windows
arch_bits:64
...
...
```

Um comando útil do Redis Client é a limpeza de cache, que pode ser feita com o comando "flushall":

```
127.0.0.1:6379> flushall
OK
```

O comando "monitor" nos permite monitorar a atividade do cache e é interessante ativá-lo em nossa estação em tempo de desenvolvimento:

```
127.0.0.1:6379> monitor
OK
1548598467.605571 [0 127.0.0.1:50299] "INFO"
1548598477.617364 [0 127.0.0.1:50299] "INFO"
1548598487.631003 [0 127.0.0.1:50299] "INFO"
1548598497.644296 [0 127.0.0.1:50299] "INFO"
1548598498.403433 [0 127.0.0.1:50384] "GET" "livros::3"
1548598498.407438 [0 127.0.0.1:50384] "SET" "livros::3" "\xac\xed\x00\x05sr\x00\x1bcom.acme.livroservice.Livro\x00\x00\x00\x00
\x00\x00\x00\x01\x02\x00\x04L\x00\x05autort\x00\x12Ljava/lang/String;L\x00\x02idt\x00\x10Ljava/lang/Long;L\x00\x05precot\x00\x
12Ljava/lang/Double;L\x00\x06tituloq\x00~\x00\x01xpt\x00\x19Antoine de Saint-Exup\xc3\x94rysr\x00\x0ejava.lang.Long;\x8b\xe4\x
90\xcc\x8f#\xfdf\x02\x00\x01J\x00\x05valuexr\x00\x10java.lang.Number\x86\xac\x95\x1d\x0b\x94\xe0\x8b\x02\x00\x00xp\x00\x00\x00\x00
\x00\x00\x00\x00\x03sr\x00\x10java.lang.Double\x80\xb3\xc2J)k\xfb\x04\x02\x00\x01D\x00\x05valueqx\x00~\x00\x0e@\x00\x00\x00\x00\x00
\x00\x00t\x00\x130 Pequeno Pr\xc3\xadncipe" "PX" "5000"
1548598507.660697 [0 127.0.0.1:50299] "INFO"
1548598517.677073 [0 127.0.0.1:50299] "INFO"
1548598522.057056 [0 127.0.0.1:50384] "KEYS" "livros::*"
1548598527.693947 [0 127.0.0.1:50299] "INFO"
1548598534.540792 [0 127.0.0.1:50384] "GET" "livros::3"
1548598534.543111 [0 127.0.0.1:50384] "SET" "livros::3" "\xac\xed\x00\x05sr\x00\x1bcom.acme.livroservice.Livro\x00\x00\x00\x00
\x00\x00\x00\x01\x02\x00\x04L\x00\x05autort\x00\x12Ljava/lang/String;L\x00\x02idt\x00\x10Ljava/lang/Long;L\x00\x05precot\x00\x
12Ljava/lang/Double;L\x00\x06tituloq\x00~\x00\x01xpt\x00\x19Antoine de Saint-Exup\xc3\x94rysr\x00\x0ejava.lang.Long;\x8b\xe4\x
90\xcc\x8f#\xfdf\x02\x00\x01J\x00\x05valuexr\x00\x10java.lang.Number\x86\xac\x95\x1d\x0b\x94\xe0\x8b\x02\x00\x00xp\x00\x00\x00\x00
\x00\x00\x00\x00\x03sr\x00\x10java.lang.Double\x80\xb3\xc2J)k\xfb\x04\x02\x00\x01D\x00\x05valueqx\x00~\x00\x0e@\x00\x00\x00\x00\x00
\x00\x00t\x00\x130 Pequeno Pr\xc3\xadncipe" "PX" "5000"
```

Habilitando Cache em Nossa Projeto

O primeiro passo é nos certificarmos que a classe que será armazenada em cache (Livro) implementa a interface

`Serializable` :

Livro

```
@Entity
public class Livro implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id @GeneratedValue Long id;
    private String autor;
    private String titulo;
    private Double preco;

    // Código atual omitido
}
```

Configuração e Dependências

Precisamos colocar `spring-boot-started-data-redis` como dependência do projeto. A configuração básica pode ser definida no arquivo de propriedades:

```
spring.cache.type=redis
spring.redis.host=192.168.99.100
spring.redis.port=6379
```

Abstração de cache

O Spring Framework fornece uma camada de abstração com um conjunto de anotações para suporte a cache que pode operar em conjunto com várias implementações de cache, como Redis, EhCache, Hazelcast, Infinispan e muito mais. O baixo acoplamento é sempre muito bem vindo :)

`@Cacheable` - Preencher o cache após a execução do método, a próxima chamada com os mesmos argumentos será omitida e o resultado será carregado do cache. Anotação fornece recurso útil chamado cache condicional. Em alguns casos, nem todos os dados devem ser armazenados em cache, por ex. Você quer armazenar na memória apenas posts mais populares.

`@CachePut` - Anotação permite atualizar a entrada no cache e suportar as mesmas opções, como a anotação em cache.

`@CacheEvict` - Remove entrada do cache, pode ser tanto condicional quanto global para todas as entradas no cache específico.

`@EnableCaching` - Anotação assegura que o post processor irá verificar todos os beans que tentam encontrar métodos demarcados e criará proxy para interceptar todas as chamadas;

`@Caching` - Agrega várias anotações do mesmo tipo quando, por exemplo, você precisa usar diferentes condições e caches.

`@CacheConfig` - Anotação em nível de classe permite especificar valores globais para anotações como nome do cache ou gerador de chaves.

Pontos de Atenção

Dados obsoletos: Dados muito dinâmicos tornam-se desatualizados rapidamente. Sem atualização ou mecanismo de expiração, o cache servirá conteúdo obsoleto.

Grande memória nem sempre é igual a maior taxa de acertos: Quando você atingir uma quantidade específica de memória, a taxa de acertos não aumentará. Neste ponto, muitas coisas dependerão das políticas de despejo e da natureza dos dados.

Otimização prematura: Vamos testar seus serviços sem cache, pode ser que seus medos se provem infundados, "A otimização prematura é a raiz de todo o mal".

Escondendo mau desempenho: Cache nem sempre é uma resposta para serviços lentos, tente otimizá-los antes, porque quando você os coloca após a camada de cache, você irá esconder possíveis erros de arquitetura.

Não compartilhe seu cache de redis: Redis funciona em uma única thread. Outras equipes podem usar seu armazenamento para outras finalidades extensivas. Todas as colisões de dados, comandos pesados (KEYS, SORT, etc.) podem bloquear seu cache e aumentar o tempo de execução. Em caso de problemas de desempenho, verifique o comando SLOWLOG.

Parâmetro de configuração [maxmemory]: Se você considera executar seu cache com persistência de captura instantânea, deverá usar menos da metade da memória disponível como *maxmemory* para evitar erros de memória.

Monitorização: Você deve monitorar seu cache, de tempos em tempos o INFO não gerará problemas de desempenho, mas o MONITOR pode reduzir drasticamente o throughput.

Cacheando os Livros

O primeiro passo é incluir a dependência `spring-boot-starter-data-redis` em nosso **pom.xml**:

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

Agora, vamos colocar em nosso arquivo de propriedades as configurações básicas de cache:

application.properties

```
spring.cache.type=redis
spring.redis.host=localhost
spring.redis.port=6379
```

Em seguida, adicionamos a anotação `@EnableCaching` na classe `LivroServiceApplication`:

LivroServiceApplication.java

```
package com.acme.livroservice;

// Código atual omitido
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
// Novidade aqui
@EnableCaching
public class LivroServiceApplication {
    // Código atual omitido
}
```

Agora, em nosso controller, devemos decorar os métodos que interagem com o cache:

```

package com.acme.livroservice;

// Código atual omitido
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;

@RestController
@RequestMapping("/livros")
public class LivrosController {

    // Código atual omitido

    @GetMapping("/{id}")
    // Novidade aqui
    @Cacheable(value = "livros", key = "#id")
    public Livro getLivroPorId(@PathVariable Long id) {
        logger.info("getLivroPorId: " + id);
        return repository.findById(id)
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
    }

    @PutMapping("/{id}")
    // Novidade aqui
    @CachePut(value = "livros", key = "#livro.id")
    public Livro atualizarLivro(@RequestBody Livro livro, @PathVariable Long id) {
        logger.info("atualizarLivro: " + livro + " id: " + id);
        return repository.findById(id).map(livroSalvo -> {
            livroSalvo.setAutor(livro.getAutor());
            livroSalvo.setTitulo(livro.getTitulo());
            livroSalvo.setPreco(livro.getPreco());
            return repository.save(livroSalvo);
        }).orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
    }

    @DeleteMapping("/{id}")
    // Novidade aqui
    @CacheEvict(value = "livros", allEntries=true)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void excluirLivro(@PathVariable Long id) {
        logger.info("excluirLivro: " + id);
        repository.deleteById(id);
    }
}

```

Execute o projeto e faça algumas solicitações para os livros, você verá que a primeira solicitação para detalhar um livro de fato aciona o método, mas da segunda solicitação em diante o método não é acionado, sendo retornado o conteúdo que está cacheado.

Para incluir um TTL (time to live) do cache, adicione a propriedade `spring.cache.redis.time-to-live` ao arquivo de propriedades:

application.properties

```
spring.cache.redis.time-to-live=5000
```

Um toque final interessante é configurar o cache para ser desabilitado durante a execução dos testes, uma vez que o teste pode ser realizado em um ambiente de CI/CD onde o servidor Redis pode estar indisponível, para isso, adicione a propriedade `spring.data.redis.repositories.enabled=false` no arquivo **application.properties** da pasta

`/src/test/resources :`

application.properties

```
spring.data.redis.repositories.enabled=false
```

Fontes

- <https://redis.io/topics/introduction>
- <https://www.tutorialspoint.com/redis/>
- <https://www.baeldung.com/spring-data-redis-tutorial>
- <https://medium.com/@MatthewFTech/spring-boot-cache-with-redis-56026f7da83a>

Gateway

Em tempos em que centenas de dispositivos interagem com Microserviços e servidores baseado em APIs, um Gateway de APIs pode se ser uma porta de entrada única para a sua arquitetura interna.

Vale lembrar que ter um Gateway de APIs é uma escolha óbvia quando falamos de aumentar segurança, experiência do usuário e facilidade para construção de um ecossistema digital.

Necessidades

- Agranularidade das APIs fornecidas pelos microserviços é frequentemente diferente do que um cliente precisa. Os microserviços geralmente fornecem APIs refinadas, o que significa que os clientes precisam interagir com vários serviços. Por exemplo, um cliente que precisa dos detalhes de um produto precisa buscar dados de vários serviços.
- Clientes diferentes precisam de dados diferentes. Por exemplo, a versão do navegador de desktop de uma área de trabalho de página de detalhes do produto é tipicamente mais elaborada do que a versão para celular.
- O desempenho da rede é diferente para diferentes tipos de clientes. Por exemplo, uma rede móvel é normalmente muito mais lenta e tem latência muito maior do que uma rede não móvel. E, claro, qualquer WAN é muito mais lenta que uma LAN. Isso significa que um cliente móvel nativo usa uma rede que possui características de desempenho muito diferentes de uma LAN usada por um aplicativo da Web do lado do servidor. O aplicativo da Web do lado do servidor pode fazer várias solicitações para serviços de back-end sem afetar a experiência do usuário, pois um cliente móvel pode fazer apenas alguns.
- O número de instâncias de serviço e suas localizações (host + porta) muda dinamicamente
- Particionar em serviços pode mudar ao longo do tempo e deve ser escondido dos clientes
- Os serviços podem usar um conjunto diversificado de protocolos, alguns dos quais podem não ser amigáveis para a web

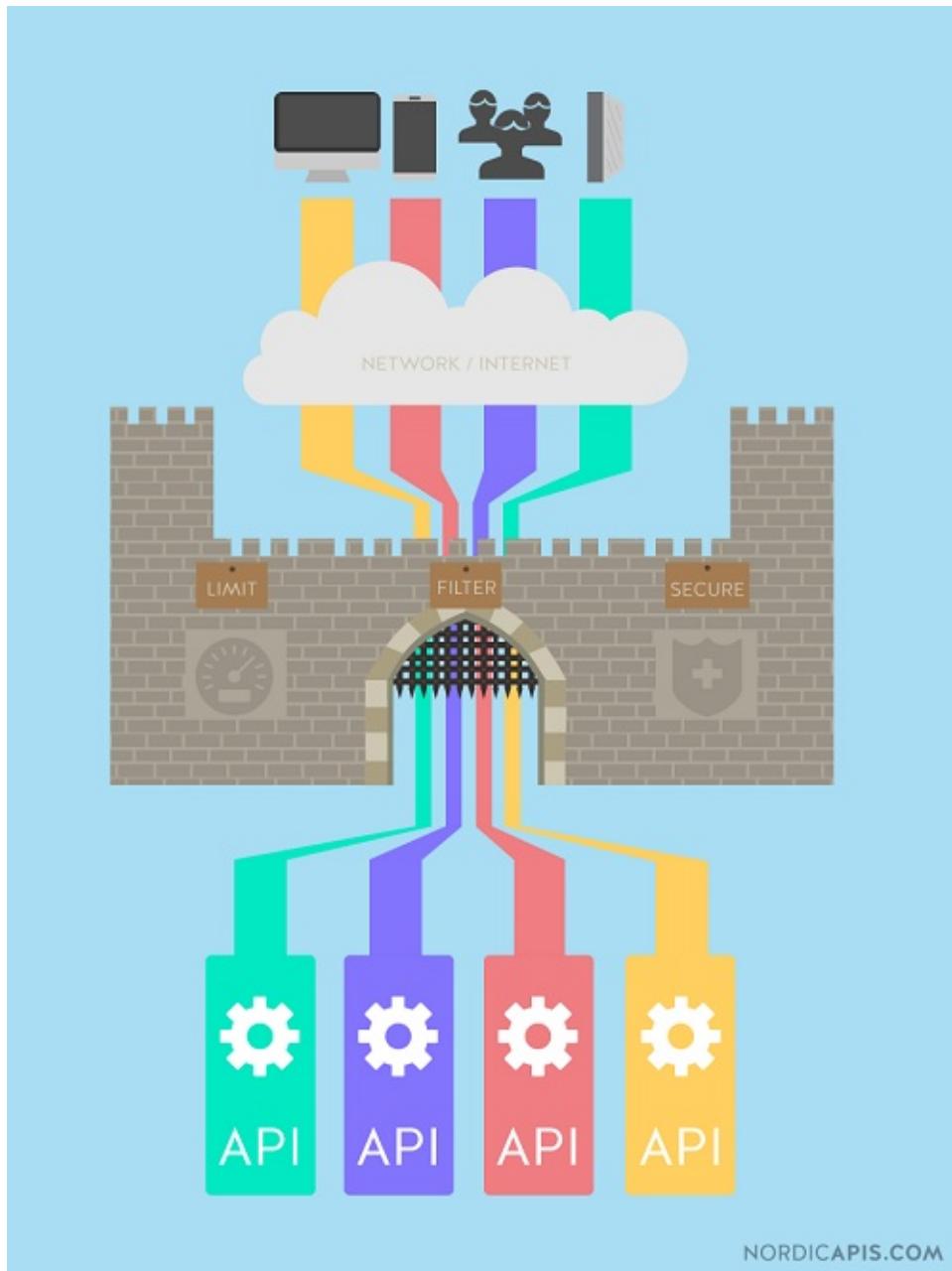
O que é um API Gateway?

Basicamente, o Gateway é uma interface que recebe as chamadas para seus sistemas internos, sendo uma grande porta de entrada.

Ele pode atuar de cinco diferentes maneiras:

- **Filtro** para o tráfego de chamadas dos diferentes meios (web, mobile, cloud, entre outros);
- **Única porta de entrada** para as diversas APIs que você deseja expor;
- **Componente essencial** do gerenciamento de APIs, como no API Suite;
- **Roteador** do tráfego nas APIs e de Rate Limit;
- **Mecanismo de segurança**, com autenticação, log e muito mais;

O acesso para o Gateway pode ser feito de muitos dispositivos diferentes. Por isso, ele deve possuir o poder de unificar as chamadas feitas e conseguir entregar ao usuário um conteúdo que pode ser acessado de qualquer tipo de navegador e sistema.



Diferenças entre API Gateway e API Management

Quando a discussão sobre APIs chega no nível de controle e gerenciamento, sempre existem dúvidas quanto a Gateway e Management.

O Gateway é responsável por criar uma **camada sobre suas APIs**, para uma arquitetura unificada.

O Management por sua vez possui um **escopo mais amplo**, pois enquanto o Gateway é responsável pelo "redirecionamento" e filtro de chamadas, uma solução de Management conta com analytics, controle de versão, business data, entre outras coisas.

Sendo assim, o Gateway acaba sendo uma parte da solução mais completa (como o Management), para um controle total de suas APIs.

Os 6 Benefícios de um API Gateway

Seguem 6 benefícios de se ter um Gateway como a porta de entrada de APIs:

1. Separação de Camada de Aplicação e diferentes requisições: Um dos melhores benefícios dessa camada é que um Gateway consegue separar claramente APIs e microserviços implementados, das pessoas que irão efetivamente utilizar elas.
2. Aumento de simplicidade para o consumidor: Utilizando um Gateway, você consegue mostrar ao seu usuário final um front-end único com sua coleção de APIs, podendo ser muito mais transparente com os usuários da API;
3. Melhoria no desenvolvimento: Separação das funcionalidades e propósitos não apenas faz com que o desenvolvimento dê muito mais foco ao que realmente é necessário, mas também ajuda o servidor a aguentar a demanda de informação pelos serviços utilizados; Por exemplo: um serviço que é chamado poucas vezes durante o dia precisa de menos recursos do que o chamado a toda hora, aproveitando melhor a performance de sua máquina.
4. Buffer Zone contra ataques: Com a utilização de vários serviços independentes e controlados pelo Gateway, qualquer ataque em sua aplicação não irá afetar o seu sistema como um todo, apenas aquele serviço, mantendo tudo funcionando perfeitamente. Isso é a Buffer Zone. Além da segurança, essa estratégia deixa tudo muito mais simples para o usuário, pois todas as outras funcionalidades mantém-se normais, não causando "stress".
5. Dedicação de Serviços em favor de User Experience: Com a estratégia de independência dos serviços das APIs, um desenvolvedor consegue ter toda a documentação necessária para a utilização de maneira muito mais simplificada, podendo otimizar o seu tempo e se dedicando exclusivamente a sua atividade. Deste modo, você consegue ter SDKs de utilização para cada API separadamente de modo a deixar sua documentação o mais específica possível;
6. Log de Atividades antecipando erros: Como todas as chamadas aos seus serviços passarão pelo Gateway, o controle de todas elas é muito simples. Esse tipo de log consegue dar um poder altíssimo para o dono da API. Com ele, é possível achar todos os erros que podem derrubar seu serviço, e até mesmo quem é o responsável por um bom consumo da API. Assim, você consegue prever a quantidade de chamadas possíveis evitando qualquer tipo de problema para o seu usuário.

Gateways como uma feature de Segurança

No universo das APIs, um dos assuntos mais abordados é sempre a segurança, e possuir um Gateway de APIs é uma das melhores soluções no mercado para conseguir ter o controle integral de sua API.

Digo isso, pois essa ferramenta contempla o chamado CID de forma quase impecável (a sigla em inglês é CIA: Confidentiality, Integrity, Availability).

Confidencialidade

Ao isolar os servidores que possuem cada tipo de informação do seu sistema utilizando um API Gateway, a confidencialidade dos dados é garantida evitando muitos tipos de ataque em sua aplicação.

Os seus servidores são desenhados e criados para resistir a invasões e manipulação de dados.

Segregando os dados na exposição da API, você consegue criar um estreitamento no caminho da informação, e neste caminho você possui total controle, sabendo até mesmo os dados que irão ser levados como resposta antes mesmo dele deixar seu servidor.

Integridade

Quando você possui um API Gateway, todos os dados de chamada e de retorno são controlados de forma automática, fazendo com que você possua garantia de que cada request em seu servidor irá ser tratado de forma única.

Deste modo, a integridade dos dados é uma certeza dentro do seu domínio.

Disponibilidade

Um dos maiores desafios de uma API é a disponibilidade 100%. Esse é um desafio de todo fornecedor de serviços.

Mesmo se sua API não estiver tão suscetível a ataques, os servidores podem sofrer perda de energia, queda da conexão e até erros humanos específicos.

É impossível imunizar totalmente a API, obviamente. Porém, se a sua API for atacada ou sofrer quedas constantes, um API Gateway possui uma segurança excelente, e pode ajudar a aumentar drasticamente sua Disponibilidade.

Você ainda consegue distribuir os seus Gateways perante os servidores dos microserviços, e possuir um roteamento de chamadas tão elaborado que fica praticamente impossível as quedas afetarem e derrubarem a sua API.

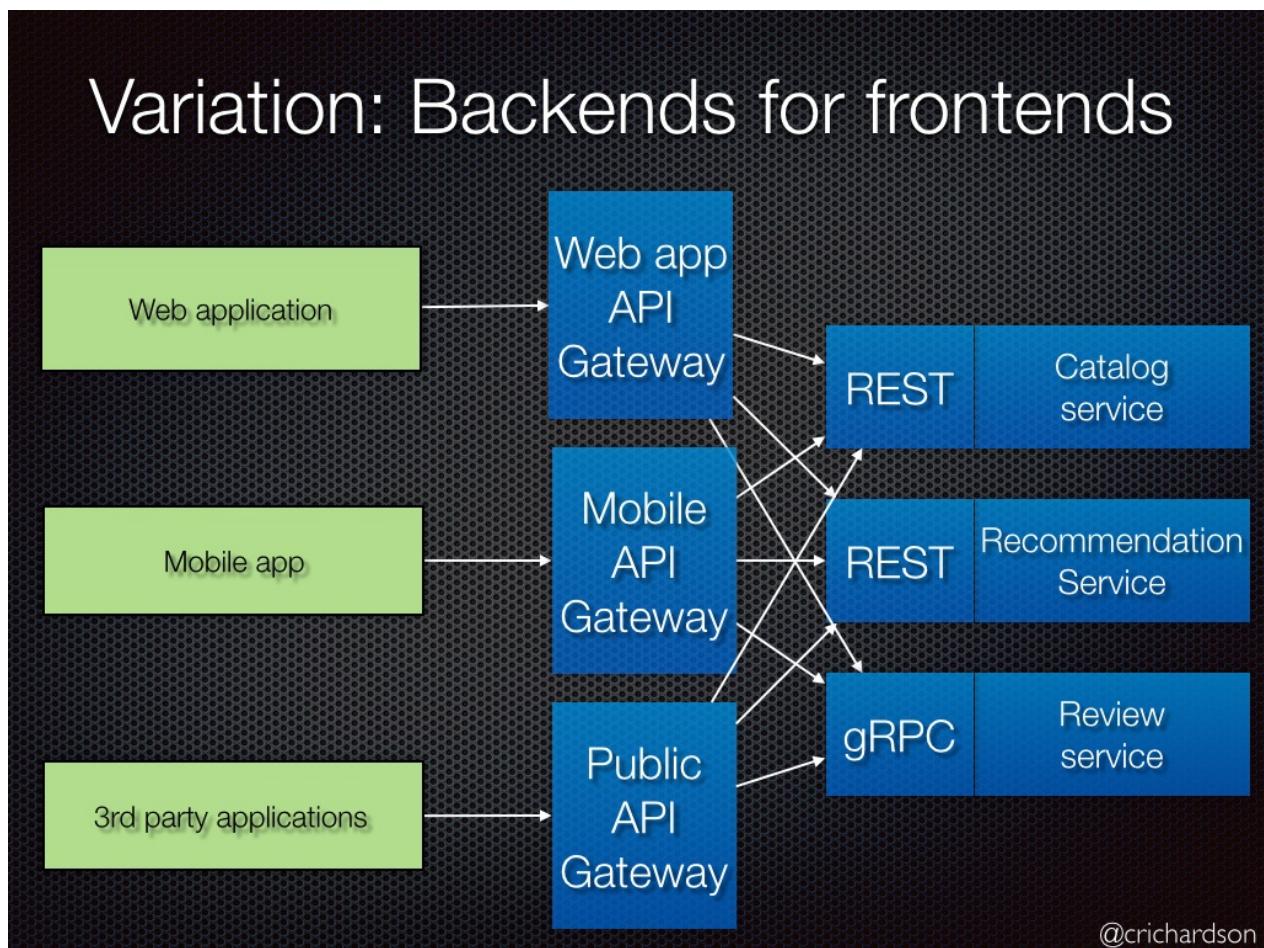
O seu Gateway

Com certeza, um API Gateway é uma das ferramentas no mercado mais efetivas para segurança, controle e desenvolvimento de sua API.

Para um melhor aproveitamento de toda sua estratégia digital, utilizar uma ferramenta como essa lado a lado do conceito de microserviços faz com que sua arquitetura seja totalmente governada e controlada, te levando um passo a frente de seus concorrentes.

Variação: backend para front-end

Uma variação desse padrão é o padrão Backend for Front-End. Ele define um gateway de API separado para cada tipo de cliente.



Neste exemplo, existem três tipos de clientes: aplicativo da web, aplicativo móvel e aplicativo externo de terceiros. Existem três gateways de API diferentes. Cada um deles fornece uma API para seu cliente.

Spring Cloud Gateway

É uma ferramenta que fornece mecanismos de roteamento prontos para uso, geralmente usados em aplicativos de microsserviços, como forma de ocultar vários serviços por trás de uma única fachada.

O Spring Cloud Gateway tem como objetivo fornecer uma maneira simples, mas eficaz, de rotear para APIs e fornecer soluções para questões como segurança, monitoramento/métricas e resiliência.

Recursos do Spring Cloud Gateway

- Construído no Spring Framework 5, Project Reactor e Spring Boot 2.0;
- Capaz de combinar rotas em qualquer atributo de solicitação;
- Predicados e filtros são específicos para rotas;
- Integração do Circuit Breaker Hystrix;
- Integração Spring Cloud DiscoveryClient
- Fácil de escrever Predicados e Filtros
- Limitação de Request Rate;
- Reescrita de caminho;

Routing Handler

Com foco em roteamento de solicitações, o Spring Cloud Gateway encaminha solicitações para um *Gateway Handler Mapping* - que determina o que deve ser feito com solicitações correspondentes a uma rota específica.

Vamos começar com um exemplo rápido de como o *Gateway Handler* resolve as configurações de rota usando o RouteLocator:

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route("r1", r -> r.host("*.baeldung.com")
            .and()
            .path("/baeldung")
            .uri("http://baeldung.com"))
        .route(r -> r.host("*.baeldung.com")
            .and()
            .path("/myOtherRouting")
            .filters(f -> f.prefixPath("/myPrefix"))
            .uri("http://othersite.com")
            .id("myOtherID"))
        .build();
}
```

Observe como usamos os principais blocos de construção desta API:

- **Route** - a API principal do gateway. É definido por uma determinada identificação (ID), um destino (URI) e um conjunto de predicados e filtros;
- **Predicate** - um predicado do Java 8 - que é usado para correspondência de solicitações HTTP usando cabeçalhos, métodos ou parâmetros;
- **Filter** - um WebFilter Spring padrão;

Dynamic Routing

Assim como o Zuul, o Spring Cloud Gateway fornece meios para rotear solicitações para diferentes serviços.

A configuração de roteamento pode ser criada usando Java puro (RouteLocator) ou usando a configuração de propriedades:

```
spring:
  application:
    name: gateway-service
  cloud:
    gateway:
      routes:
        - id: baeldung
          uri: baeldung.com
        - id: myOtherRouting
          uri: localhost:9999
```

Supporte ao Spring Cloud DiscoveryClient

O Spring Cloud Gateway pode ser facilmente integrado às bibliotecas de Service Discovery and Registry, como o Eureka Server e o Consul:

```
@Configuration
@EnableDiscoveryClient
public class GatewayDiscoveryConfiguration {

  @Bean
  public DiscoveryClientRouteDefinitionLocator
    discoveryClientRouteLocator(DiscoveryClient discoveryClient) {
    return new DiscoveryClientRouteDefinitionLocator(discoveryClient);
  }
}
```

Monitoramento

O Spring Cloud Gateway faz uso da API do Actuator, uma biblioteca bem conhecida do Spring-Boot que fornece vários serviços prontos para monitorar o aplicativo.

Depois que a API do Actuator é instalada e configurada, os recursos de monitoramento do gateway podem ser visualizados acessando endpoint `/gateway/`.

Utilizando um Gateway API em nosso projeto

O primeiro passo é importarmos o projeto `avaliacoes-service` e executá-lo, deste modo, ficaremos com os seguintes serviços executados localmente:

- `http://localhost:8080` -> `livro-service`
- `http://localhost:8081` -> `avaliacao-service`
- `http://localhost:8888` -> `config-server`
- `http://localhost:8500` -> `Consul`
- `http://localhost:15672` -> `RabbitMQ`
- `http://localhost:6379` -> `Redis` (sem interface Web)

Nosso objetivo é disponibilizar nossos microsserviços através de uma única porta, que será nosso API Gateway, neste modo ficaremos com:

- `http://localhost:9090/livros` -> `http://localhost:8080/livros`
- `http://localhost:9090/avaliacoes` -> `http://localhost:8080/avaliacoes`

Criando um projeto de Gateway

Acesse <https://start.spring.io/>, em Group altere para `com.acme`, em Artifact digite `gateway` e nas dependências busque por `Gateway`, `Consul Discovery`, `Config Client` e `Actuator`, gere o projeto, faça o download e importe no Spring Tools.

Podemos ajustar as configurações para que o Gateway utilize as configurações armazenadas através do Config Server assim como utilizar o serviço de discovery do Consul para encontrá-lo (como já fizemos com as outras aplicações):

bootstrap.properties

```
spring.cloud.config.name=gateway
spring.profiles.active=default

spring.cloud.config.discovery.serviceId=config-server
spring.cloud.config.fail-fast=true

spring.cloud.consul.host=localhost
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.instanceId=${spring.cloud.config.name}:${random.value}
spring.cloud.consul.discovery.serviceName=${spring.cloud.config.name}

management.endpoints.web.exposure.include=*
```

O último passo é criar um arquivo de configurações para nosso gateway no config-repo, neste caso, utilizaremos o formato yml pois existem configurações multivvaloradas:

gateway.yml

```
server:
  port: 9090

spring:
  cloud:
    gateway:
      routes:
        - id: livro_service_route
          uri: http://localhost:8080
          predicates:
            - Path=/livros
        - id: avalicacao_service_route
          uri: http://localhost:8081
          predicates:
            - Path=/avaliacoes
      config:
        name: gateway
      consul:
        host: localhost
        port: 8500
        discovery:
          instanceId: ${spring.cloud.config.name}:${random.value}
          serviceName: ${spring.cloud.config.name}

management:
  endpoints:
    web:
      exposure:
        include: ***
```

Acesse <http://localhost:9090/livros> e <http://localhost:9090/avaliacoes> para testar o funcionamento do Gateway.

Utilizando o Discovery e o Load Balancer

Uma vez que nossos microsserviços já se encontram disponíveis para descoberta no Consul não seria adequado que nosso gateway buscassem esta informação de lá? Isto é possível de ser feito, primeiro, devemos ativar o serviço de cliente de Discovery na classe `GatewayApplication`:

GatewayApplication.java

```

package com.acme.gateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
// Novidade aqui
@EnableDiscoveryClient
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}

```

O segundo passo é ajustar as URLs de nossos microsserviços utilizando o esquema "lb" seguido do nome do serviço ao invés de sua URL:

gateway.yml

```

server:
  port: 9090

spring:
  cloud:
    gateway:
      routes:
        - id: livro_service_route
          # Novidade aqui
          uri: lb://livro-service
          predicates:
            - Path=/livros
        - id: avaliacao_service_route
          # Novidade aqui
          uri: lb://avaliacao-service
          predicates:
            - Path=/avaliacoes
      config:
        name: gateway
      consul:
        host: localhost
        port: 8500
        discovery:
          instanceId: ${spring.cloud.config.name}:${random.value}
          serviceName: ${spring.cloud.config.name}

management:
  endpoints:
    web:
      exposure:
        include: "*"

```

Nosso gateway agora já deve estar funcionando corretamente descobrindo os serviços a partir do Consul.

Fontes

- <https://nordicapis.com/api-gateways-direct-microservices-architecture/>
- <https://microservices.io/patterns/apigateway.html>
- <https://www.baeldung.com/spring-cloud-gateway>
- <https://www.baeldung.com/spring-cloud-gateway-pattern>

Admin

O Spring Boot Admin é um aplicativo da Web, usado para gerenciar e monitorar aplicativos Spring Boot. Cada aplicativo é considerado como um cliente e se registra no servidor admin. Nos bastidores, a mágica é feita pelos *end-points* do Spring Boot Actuator.

Configurações

Podemos utilizar o site do Spring Initializr para realizar a criação de um projeto de admin, para isso, faça as seguintes seleções:

- Group: com.acme
- Artifact: admin
- Dependencies: Spring Boot Admin (Server), Consul Discovery, Actuator

Realizar o download, descompactar o projeto e importá-lo no STS.

O **pom.xml** da aplicação deve estar semelhante ao conteúdo abaixo:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.2.RELEASE</version>
    <relativePath /> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.acme</groupId>
  <artifactId>admin</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>admin</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
    <spring-boot-admin.version>2.1.1</spring-boot-admin.version>
    <spring-cloud.version>Greenwich.RELEASE</spring-cloud.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>de.codecentric</groupId>
      <artifactId>spring-boot-admin-starter-server</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-consul-discovery</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>

    <dependency>
```

```

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
</dependencies>

<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-dependencies</artifactId>
<version>${spring-cloud.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
<dependency>
<groupId>de.codecentric</groupId>
<artifactId>spring-boot-admin-dependencies</artifactId>
<version>${spring-boot-admin.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>

<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<executions>
<execution>
<id>build-info</id>
<goals>
<goal>build-info</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>

```

Para o arquivo **AdminApplication.java** devem ser incluídas algumas anotações, conforme abaixo:

```

package com.acme.admin;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

import de.codecentric.boot.admin.server.config.EnableAdminServer;

@EnableAdminServer
@SpringBootApplication
@EnableDiscoveryClient
public class AdminApplication {

    public static void main(String[] args) {
        SpringApplication.run(AdminApplication.class, args);
    }
}

```

O arquivo **bootstrap.properties** deve ter as configurações padrão das aplicações da cloud:

```
spring.cloud.config.name=admin
spring.profiles.active=default

spring.cloud.config.discovery.serviceId=config-server
spring.cloud.config.fail-fast=true

spring.cloud.consul.host=localhost
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.instanceId=${spring.cloud.config.name}:${random.value}
spring.cloud.consul.discovery.serviceName=${spring.cloud.config.name}

management.endpoints.web.exposure.include=*
```

Por fim, o arquivo **admin.properties** no **config-repo** deve ter as seguintes configurações:

```
server.port=9091
spring.boot.admin.discovery.ignored-services=consul
```

O Consul deve ser ignorado da monitoração uma vez que não conta com os **end-points** do Spring Actuator que são necessários pelo Admin.

Acessando o endereço <http://localhost:9091> teremos acesso à administração das instâncias de serviços do Spring Boot em execução.

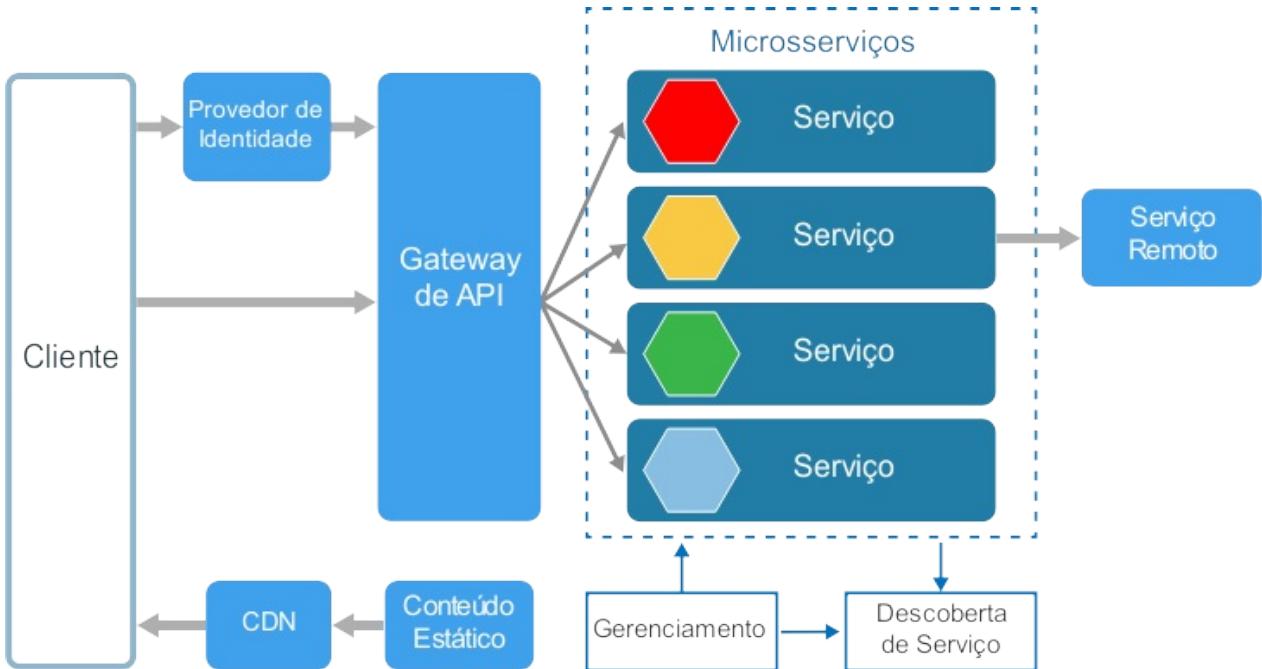
Um toque especial em nossas aplicações

Podemos customizar o visual da inicialização de nossa aplicação utilizando um banner customizado, para isso, baixe o arquivo a seguir e coloque-o na pasta `src/main/resources` de suas aplicações

<https://raw.githubusercontent.com/tiagolpadua/msc-files/master/banner.txt>

Design Arquitetural de Microsserviços

Uma arquitetura de microsserviços consiste em uma coleção de pequenos serviços autônomos. Cada serviço é independente e deve implementar uma única funcionalidade comercial.



Em alguns aspectos, microsserviços são a evolução natural das arquiteturas orientadas a serviços (SOA), mas há diferenças entre microsserviços e SOA. Aqui estão algumas características que definem um microsserviço:

- Em uma arquitetura de microsserviços, os serviços são pequenos, independentes e fracamente acoplados.
- Cada serviço é uma base de código separado, que pode ser gerenciado por uma equipe de desenvolvimento pequena.
- Os serviços podem ser implantados de maneira independente. Uma equipe pode atualizar um serviço existente sem recompilar e reimplantar o aplicativo inteiro.
- Os serviços são responsáveis por manter seus próprios dados ou o estado externo. Isso é diferente do modelo tradicional, em que uma camada de dados separada lida com a persistência de dados.
- Os serviços comunicam-se entre si por meio de APIs bem definidas. Detalhes da implementação interna de cada serviço ficam ocultos de outros serviços.
- Os serviços não precisam compartilhar a mesma pilha de tecnologia, bibliotecas ou estruturas.

Além para os próprios serviços, alguns outros componentes aparecem em uma arquitetura de microsserviços típica:

Gerenciamento. O componente de gerenciamento é responsável por colocar serviços em nós, identificar falhas, rebalancear serviços entre nós e assim por diante.

Descoberta de Serviço. Mantém uma lista de serviços e em quais nós eles estão localizados. Habilita a pesquisa de serviço para localizar o ponto de extremidade para um serviço.

Gateway de API. O gateway de API é o ponto de entrada para os clientes. Os clientes não chamam serviços diretamente. Em vez disso, eles chamam o gateway de API, que encaminha a chamada para os serviços apropriados no back-end. O gateway de API pode agragar as respostas de vários serviços e retornar a resposta agregada.

As vantagens de usar um gateway de API incluem:

- Desacoplar os clientes dos serviços. Os serviços podem ter controle de versão ou ser refatorado sem necessidade de atualizar todos os clientes.
- Os serviços podem usar protocolos de mensagens que não sejam amigáveis à Web, como AMQP.

- O Gateway de API pode executar outras funções abrangentes, como autenticação, registro em log, terminação SSL e balanceamento de carga.

Quando usar essa arquitetura

Considere esse estilo de arquitetura para:

- Aplicativos grandes que precisem de uma alta velocidade de liberação.
- Aplicativos complexos que precisem ser altamente dimensionável.
- Aplicativos com domínios avançados ou muitos subdomínios.
- Uma organização que consista em pequenas equipes de desenvolvimento.

Benefícios

Implantações independentes. Você pode atualizar um serviço sem reimplantar o aplicativo inteiro e, em seguida, reverter ou efetuar roll forward de uma atualização se algo der errado. Correções de bugs e liberações de recurso são mais fáceis de gerenciar e menos arriscadas.

Desenvolvimento independente. Uma única equipe de desenvolvimento pode criar, testar e implantar um serviço. O resultado é inovação contínua e um ritmo mais rápido de liberação.

Equipes pequenas e focadas. As equipes podem se concentrar em um serviço. O escopo menor de cada serviço torna a base de código mais fácil de entender e é mais fácil para novos membros da equipe fazerem a expansão.

Isolamento de falha. Se um serviço falhar, ele não derrubará o aplicativo inteiro. No entanto, isso não significa que você obtém resiliência gratuitamente. Você ainda precisa seguir as melhores práticas de resiliência e padrões de design.

Pilhas de tecnologia mistas. As equipes podem escolher a tecnologia mais adequada para seu serviço.

Dimensionamento granular. Os serviços podem ser dimensionados de maneira independente. Ao mesmo tempo, a densidade mais alta de serviços por VM significa que os recursos de VM são totalmente usados. Usando restrições de posicionamento, um serviço pode corresponder a um perfil de VM (alta utilização da CPU e de memória e assim por diante).

Desafios

- **Complexidade.** Um aplicativo de microsserviços tem mais partes móveis que o aplicativo monolítico equivalente. Cada serviço é mais simples, mas o sistema como um todo é mais complexo.
- **Desenvolvimento e teste.** Desenvolver com relação a dependências de serviço exige uma abordagem diferente. As ferramentas existentes não são necessariamente projetadas para funcionar com dependências de serviço. Refatorar entre limites de serviços pode ser difícil. Também pode ser um desafio testar as dependências de serviço, especialmente quando o aplicativo está evoluindo rapidamente.
- **Falta de governança.** A abordagem descentralizada para compilar microsserviços tem vantagens, mas também pode causar problemas. Você pode acabar com muitos idiomas e estruturas diferentes que tornam difícil a manutenção do aplicativo. Pode ser útil estabelecer alguns padrões para todo o projeto, sem restringir excessivamente a flexibilidade das equipes. Isso se aplica especialmente a funcionalidades abrangentes como registro em log.
- **Latência e congestionamento de rede.** O uso de muitos serviços granulares pequenos pode resultar em mais comunicação entre serviços. Além disso, se a cadeia de dependências de serviço ficar muito longa (o serviço A chama o B, que chama o C...) a latência adicional poderá se tornar um problema. Você precisará projetar APIs com cuidado. Evite APIs excessivamente prolixas, pense em formatos de serialização e procure locais para usar padrões de comunicação assíncrona.
- **Integridade de dados.** Cada microsserviço deve ser responsável pela própria persistência de dados. Assim, a consistência dos dados pode ser um desafio. Adote consistência eventual quando possível.

- **Gerenciamento.** Ter êxito com microsserviços requer uma cultura DevOps madura. Registro em log correlacionado entre serviços pode ser desafiador. Normalmente, o registro em log deve correlacionar várias chamadas de serviço para uma operação de um único usuário.
- **Controle de versão.** As atualizações de um serviço não devem interromper os serviços que dependerem delas. Vários serviços podem ser atualizados a qualquer momento, portanto, sem design cuidadoso, você pode ter problemas com compatibilidade com versões anteriores ou futuras.
- **Conjunto de qualificações.** Os microsserviços são sistemas altamente distribuídos. Avalie cuidadosamente se a equipe tem as habilidades e a experiência para ser bem-sucedida.

Práticas recomendadas

- Modele os serviços em torno de domínio da empresa.
- Descentralize tudo. Equipes individuais são responsáveis por projetar e criar serviços. Evite compartilhar esquemas de dados ou códigos.
- O armazenamento de dados deve ser privado para o serviço que é o proprietário dos dados. Use o melhor armazenamento para cada serviço e tipo de dados.
- Os serviços comunicam-se por meio de APIs bem projetadas. Evite o vazamento de detalhes da implementação. As APIs devem modelar o domínio, não a implementação interna do serviço.
- Evite acoplamento entre serviços. Causas de acoplamento incluem protocolos de comunicação rígidos e esquemas de banco de dados compartilhados.
- Transfira problemas abrangentes, como autenticação e terminação SSL, para o gateway.
- Mantenha o conhecimento de domínio fora do gateway. O gateway deve tratar e rotear solicitações de cliente sem qualquer conhecimento das regras de negócios ou da lógica do domínio. Caso contrário, o gateway se tornará uma dependência e poderá causar um acoplamento entre serviços.
- Os serviços devem ter um acoplamento flexível e alta coesão funcional. Funções que provavelmente mudarão juntas devem ser empacotadas e implantadas juntas. Se residirem em serviços separados, esses serviços acabarão sendo fortemente acoplados, porque uma alteração em um serviço exigirá atualizar outro. Uma comunicação excessivamente prolixas entre dois serviços pode ser um sintoma de acoplamento forte e coesão baixa.
- Isole falhas. Use estratégias de resiliência para impedir que falhas em um serviço distribuam-se em cascata. Consulte Padrões de resiliência e Design de aplicativos resilientes.

Fonte

- <https://docs.microsoft.com/pt-br/azure/architecture/guide/architecture-styles/microservices>

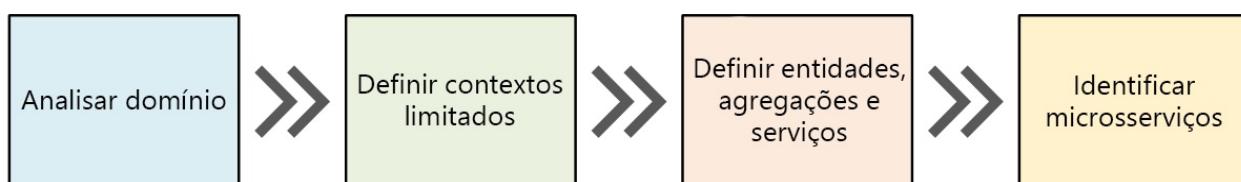
Análise de domínio

Um dos maiores desafios de microsserviços é definir os limites de serviços individuais. A regra geral é que um serviço deve fazer "algo" — mas colocar essa regra em prática requer uma consideração cuidadosa. Não há nenhum processo mecânico que produz o design "certo". Você precisa pensar profundamente em suas metas, requisitos e domínio de negócios. Caso contrário, você pode terminar com um design aleatório que exibe algumas características indesejáveis, como dependências ocultas entre serviços, acoplamento rígido ou então interfaces mal projetadas.

Microsserviços devem ser criados em torno de capacidades comerciais, não de camadas horizontais como acesso a dados ou mensagens. Além disso, eles devem ter um acoplamento flexível e alta coesão funcional. Os microsserviços serão acoplados flexivelmente se você puder alterar um serviço sem a necessidade de outros serviços serem atualizados simultaneamente. Um microsserviço é coeso se ele tem uma finalidade única e bem definida, como gerenciar contas de usuário ou acompanhar o histórico de entrega. Um serviço deve encapsular o conhecimento do domínio e abstrair esse conhecimento de clientes. Por exemplo, um cliente deve ser capaz de agendar um serviço de drone sem conhecer os detalhes do algoritmo de agendamento ou como a frota de drones é gerenciada.

O DDD (design orientado a domínio) fornece uma estrutura que pode ajudar você pela maior parte do processo de obtenção de um conjunto de microsserviços bem projetado. O DDD tem duas fases diferentes, a estratégica e a tática. No DDD estratégico, você está definindo a estrutura em grande escala do sistema. O DDD estratégico ajuda a garantir que sua arquitetura permaneça concentrada em capacidades comerciais. O DDD tático fornece um conjunto de padrões de design que você pode usar para criar o modelo de domínio. Esses padrões incluem entidades, agregações e serviços de domínio. Esses padrões táticos lhe ajudarão a criar microsserviços que são acoplados flexivelmente e também coesos.

- Diagrama de um processo de design orientado por domínio (DDD)



1. Comece analisando o domínio corporativo para entender os requisitos funcionais do aplicativo. A saída desta etapa é uma descrição informal do domínio, que pode ser redefinido em um conjunto mais formal de modelos de domínio.
2. Em seguida, defina os *contextos limitados* do domínio. Cada contexto limitado contém um modelo de domínio que representa um subdomínio específico do aplicativo maior.
3. Dentro de um contexto limitado, aplique padrões DDD táticos para definir entidades, agregações e serviços de domínio.
4. Use os resultados da etapa anterior para identificar os microsserviços em seu aplicativo.

É importante lembrar que DDD é um processo iterativo e em andamento. Limites de serviço não são fixos de forma imutável. Conforme um aplicativo evolui, você pode decidir dividir um serviço em vários serviços menores.

Analizar o domínio

Usar uma abordagem DDD lhe ajudará a criar microsserviços de modo que cada serviço se adeque naturalmente a um requisito de negócios funcional. Ele pode lhe ajudar a evitar a armadilha de permitir que os limites organizacionais ou opções de tecnologia ditem o seu design.

Antes de escrever qualquer código, será necessária uma vista panorâmica do sistema que você está criando. O DDD começa modelando o domínio corporativo e criando uma modelo de domínio. O modelo de domínio é um modelo abstrato do domínio corporativo. Destila e organiza os dados de conhecimento do domínio e fornece uma linguagem comum para desenvolvedores e especialistas de domínio.

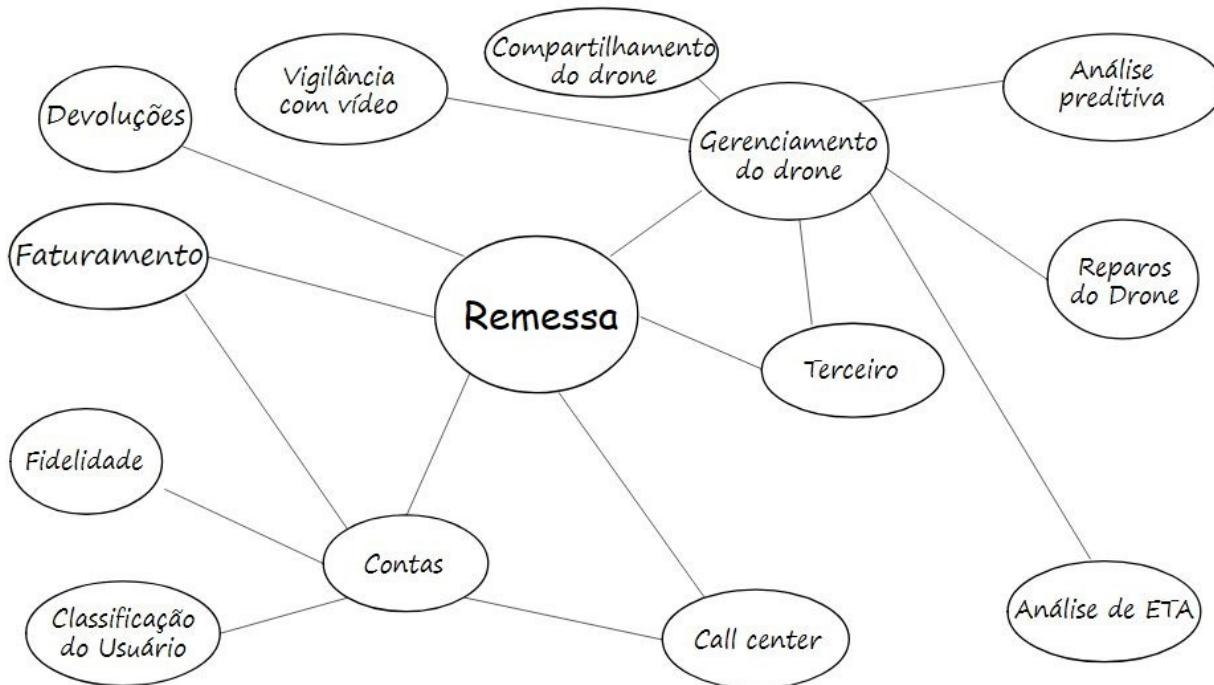
Iniciar o mapeamento de todas as funções de negócios e as respectivas conexões. Isso provavelmente será um esforço colaborativo envolvendo especialistas de domínio, arquitetos de software e outros stakeholders. Você não precisa usar nenhum formalismo específico. Esboce um diagrama ou desenhe em quadro de comunicações.

Conforme você preencher o diagrama, poderá começar a identificar subdomínios discretos. Quais funções estão intimamente relacionadas? Quais funções são essenciais para os negócios e quais fornecem serviços complementares? O que é o grafo de dependência? Durante a fase inicial, você não se importa com tecnologias ou detalhes de implementação. Dito isso, você deverá tomar nota do local em que o aplicativo precisará se integrar com sistemas externos como CRM, processamento do pagamento ou sistemas de cobrança.

Entrega por Drones: analisando o domínio corporativo

Após algumas análises de domínio iniciais, a equipe da Fabrikam criou um esboço aproximado que representa o domínio de Entrega por Drones.

- Diagrama de um domínio de Entrega por Drones



- **Remessa** é colocado no centro do diagrama, já que é essencial para os negócios. Tudo no diagrama existe para habilitar essa funcionalidade.
- **Gerenciamento de drones** também é importante para os negócios. Funcionalidades que estão intimamente relacionada ao gerenciamento de drones incluem o reparo de drones e o uso de análise preditiva para prever quando drones precisam de manutenção e reparo.
- A **Análise de ETA** fornece estimativas de tempo para coleta e entrega.
- O **transporte de terceiros** permitirá que o aplicativo agende métodos de transporte alternativos se um pacote não puder ser enviado inteiramente por drone.
- **Compartilhamento de drones** é uma possível extensão do negócio principal. A empresa pode ter capacidade de drones excedente durante determinadas horas e pode alugar drones que, caso contrário, ficariam ociosos. Este recurso não estará na versão inicial.
- **Vigilância por vídeo** é outra área para a qual empresa pode expandir mais tarde.
- **Contas de usuário, Faturamento e Call center** são subdomínios que dão suporte ao negócio principal.

Observe que neste momento no processo, não tomamos nenhuma decisão sobre implementação ou tecnologias. Alguns dos subsistemas podem envolver sistemas externos de software ou serviços de terceiros. Mesmo assim, o aplicativo deve interagir com esses sistemas e serviços, portanto, é importante incluí-los no modelo de domínio.

Observação: Quando um aplicativo depende de um sistema externo, há um risco de que a API ou o esquema de dados do sistema externo vaze dados para seu aplicativo, comprometendo consequentemente o design de arquitetura. Isso é especialmente verdadeiro com sistemas legados que podem não seguir práticas recomendadas modernas e que podem usar esquemas de dados complicados ou APIs obsoletas. Nesse caso, é importante ter um limite bem definido entre esses sistemas externos e o aplicativo. Considere o uso do Padrão do Estrangulador (Migrar incrementalmente um sistema herdado substituindo gradualmente partes específicas de funcionalidade por serviços e aplicativos novos) ou Padrão de Camada Anticorrupção (Implementar uma camada de fachada ou de adaptador entre diferentes subsistemas que não compartilham a mesma semântica) para essa finalidade.

Definir contextos limitados

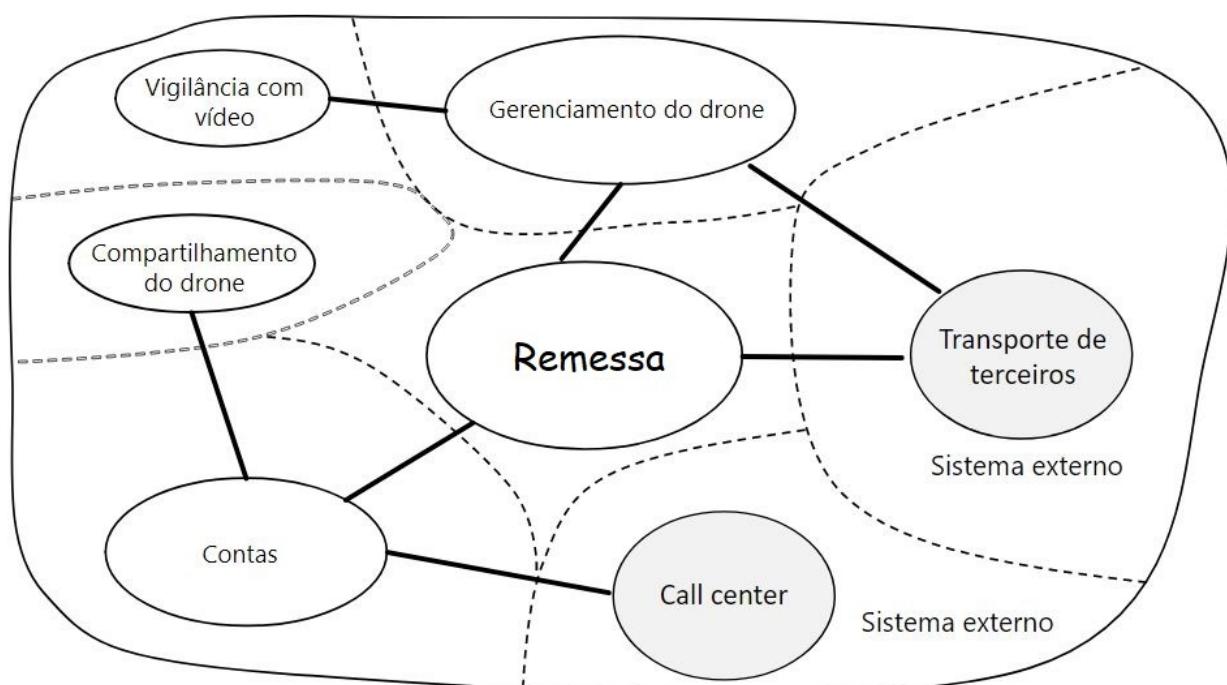
O modelo de domínio inclui representações de objetos reais no mundo — usuários, drones, pacotes e assim por diante. Mas isso não significa que todas as partes do sistema precisam usar as mesmas representações para as mesmas coisas.

Por exemplo, subsistemas que manipulam análise preditiva e reparo de drones precisarão representar muitas características físicas dos drones, como seu histórico de manutenção, quilometragem, idade, número do modelo, características de desempenho e assim por diante. Porém, quando é hora de agendar uma entrega, esses elementos não importam. O subsistema de agendamento só precisa saber se um drone está disponível e o ETA para coleta e entrega.

Se tentássemos criar um único modelo para ambos esses subsistemas, ele seria desnecessariamente complexo. Também seria mais difícil para o modelo evoluir ao longo do tempo, porque as alterações precisariam atender a várias equipes trabalhando em subsistemas separados. Portanto, geralmente é melhor criar modelos separados que representam a mesma entidade do mundo real (nesse caso, um drone) em dois contextos diferentes. Cada modelo contém apenas os recursos e os atributos que são relevantes no contexto específico dele.

É aqui que o conceito DDD de contextos limitados entra em cena. Um contexto limitado é simplesmente o limite em um domínio em que um modelo de domínio específico se aplica. Examinando o diagrama anterior, é possível agrupar as funcionalidades dependendo de várias funções compartilharem ou não um único modelo de domínio.

- Diagrama de contextos limitados



Contextos limitados não são necessariamente isolados uns dos outros. Neste diagrama, as linhas sólidas, conectando os contextos limitados representam lugares onde dois contextos limitados interagem. Por exemplo, a *Remessa* depende de contas de usuário obterem informações sobre os clientes e do gerenciamento de drones agendar drones da frota.

No livro *Domain Driven Design*, Eric Evans descreve vários padrões para manter a integridade de um modelo de domínio quando ele interage com outro contexto limitado. Um dos principais princípios de microserviços é que os serviços se comunicam por meio de APIs bem definidas. Essa abordagem corresponde a dois padrões que Evans chama de serviço de host aberto e linguagem de programação publicada. A ideia de serviço de host aberto é que um subsistema define um protocolo formal (API) para outros subsistemas se comunicarem com ele. A linguagem de programação publicada estende essa ideia publicando a API de uma forma que outras equipes podem usar para escrever código de clientes.

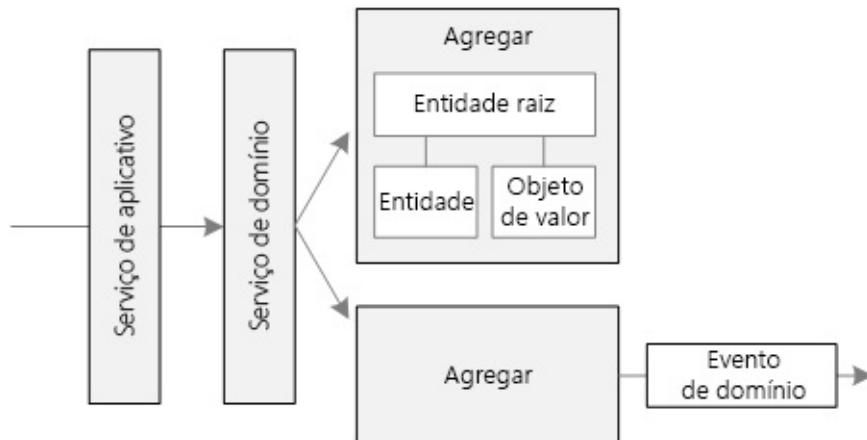
Para o restante dessa jornada, nos concentraremos no contexto limitado de *Remessa*.

DDD tático

Durante a fase estratégica de DDD, você está mapeando fora do domínio corporativo e definindo contextos limitados para seus modelos de domínio. DDD tático é quando você define os modelos de domínio com mais precisão. Os padrões táticos são aplicados dentro de um único contexto limitado. Em uma arquitetura de microserviços, estamos interessados principalmente nos padrões de entidade e de agregação. Aplicar esses padrões nos ajudará a identificar limites naturais para os serviços em nosso aplicativo. Como um princípio geral, um microserviço não deve ser menor do que uma agregação e não pode exceder um contexto limitado. Primeiro, examinaremos os padrões táticos. Em seguida, os aplicaremos ao contexto limitado de remessa no aplicativo de entrega por drones.

Visão geral dos padrões táticos

- Diagrama de padrões táticos no design orientado por domínio



Entidades. Uma entidade é um objeto com uma identidade exclusiva que persiste ao longo do tempo. Por exemplo, em um aplicativo de serviços bancários, clientes e contas seriam entidades.

- Uma entidade tem um identificador exclusivo no sistema, que pode ser usado para pesquisar pela entidade ou recuperá-la. Isso não significa que o identificador é sempre exposto diretamente aos usuários. Ele pode ser um GUID ou uma chave primária em um banco de dados.
- Uma identidade pode abranger vários contextos limitados e pode durar mais que o tempo de vida do aplicativo. Por exemplo, números de conta bancária ou IDs emitidas pelo governo não estão vinculados ao tempo de vida de um aplicativo específico.
- Os atributos de uma entidade podem se alterar ao longo do tempo. Por exemplo, o nome ou endereço de uma pessoa pode ser alterado, mas ela ainda é a mesma pessoa.
- Uma entidade pode conter referências a outras entidades.

Objetos de valor. Um objeto de valor não tem identidade. Ele é definido somente pelos valores de seus atributos. Objetos de valor também são imutáveis. Para atualizar um objeto de valor, você sempre cria uma nova instância para substituir a antiga. Objetos de valor podem ter métodos que encapsulam a lógica do domínio, mas esses métodos não devem ter efeitos colaterais sobre o estado do objeto. Exemplos comuns de objetos de valor incluem valores de datas e horas, moedas e cores.

Agregações. Uma agregação define um limite de consistência em torno de uma ou mais entidades. Exatamente uma entidade em uma agregação é a raiz. Pesquisa é feita usando o identificador da entidade raiz. Quaisquer outras entidades na agregação são filhas da raiz e são referenciadas da raiz pelos ponteiros a seguir.

Afinalidade de uma agregação é modelar invariáveis transacionais. Coisas no mundo real têm redes de relações complexas. Os clientes criam ordens, as ordens contêm produtos, os produtos têm fornecedores e assim por diante. Se o aplicativo modifica vários objetos relacionados, como ele garante a consistência? Como manter o controle de invariáveis e impô-las?

Aplicativos tradicionais têm usado frequentemente transações de banco de dados para impor a consistência. Em um aplicativo distribuído, no entanto, isso muitas vezes não é viável. Uma única transação empresarial pode abranger vários repositórios de dados, ser demorada ou envolver serviços de terceiros. Por fim, cabe ao aplicativo, não à camada de dados, impor as invariáveis necessárias para o domínio. É isso que as agregações destinam-se a modelar.

Observação: Um agregado pode consistir em uma única entidade, sem entidades filho. O que compõe uma agregação é o limite transacional.

Domínio e serviços de aplicativo. Na terminologia DDD, um serviço é um objeto que implementa uma lógica sem conter nenhum estado. Evans faz distinção entre serviços de domínio, que encapsulam a lógica do domínio, e serviços de aplicativo, que fornecem funcionalidades técnicas como autenticação de usuário ou envio de uma mensagem SMS. Serviços de domínio geralmente são usados para modelar comportamento que abrange várias entidades.

Observação: O termo serviço está sobrecarregado no desenvolvimento de software. Aqui a definição não está diretamente relacionada a microsserviços.

Eventos de domínio. Eventos de domínio podem ser usados para notificar outras partes do sistema quando algo acontece. Como o nome sugere, eventos de domínio devem significar algo dentro do domínio. Por exemplo, "um registro foi inserido em uma tabela" não é um evento de domínio. "Uma entrega foi cancelada" é um evento de domínio. Eventos de domínio são especialmente relevantes em uma arquitetura de microsserviços. Já que microsserviços são distribuídos e não compartilham armazenamentos de dados, eventos de domínio fornecem uma maneira para os microsserviços se coordenarem entre si.

Há alguns outros padrões de DDD não listados aqui, incluindo módulos, repositórios e fábricas. Eles poderão ser padrões úteis para quando você estiver implementando um microsserviço, mas eles são menos relevantes ao projetar os limites entre microsserviços.

Entrega por Drones: Explorando os padrões

Começaremos com os cenários com os quais o contexto limitado de Remessa deve lidar.

- Um cliente pode solicitar que um drone colete os produtos de uma empresa que está registrada com o serviço de entrega por drones.
- O remetente gera uma marcação (código de barras ou RFID) para colocar no pacote.
- Um drone coletará um pacote no local de origem e o entregará no local de destino.
- Quando um cliente agenda uma entrega, o sistema fornece um ETA com base em informações de rota, condições climáticas e dados históricos.
- Quando o drone está em trânsito, um usuário pode acompanhar a localização atual e a ETA mais recente.
- Até que um drone tenha coletado o pacote, o cliente pode cancelar uma entrega.
- O cliente é notificado quando a entrega é concluída.
- O remetente pode solicitar confirmação de entrega do cliente, na forma de uma assinatura ou impressão digital.

- Os usuários podem pesquisar o histórico de uma entrega concluída.

Desses cenários, a equipe de desenvolvimento identificou as entidades a seguir.

- Entrega
- Pacote
- Drone
- Conta
- Confirmação
- Notificação
- Marca

Os quatro primeiros, entrega, pacote, drone e conta, são todos agregações que representam limites de consistência transacional. Confirmações e Notificações são entidades filho de Entregas e Marcações são entidades filho de Pacotes.

Os **objetos de valor** neste projeto incluem Localização, ETA, PackageWeight e PackageSize.

Para ilustrar, aqui está um diagrama UML da agregação de Entrega. Observe que ele contém referências a outras agregações, incluindo Conta, Pacote e Drone.

- Diagrama UML da agregação de Entrega

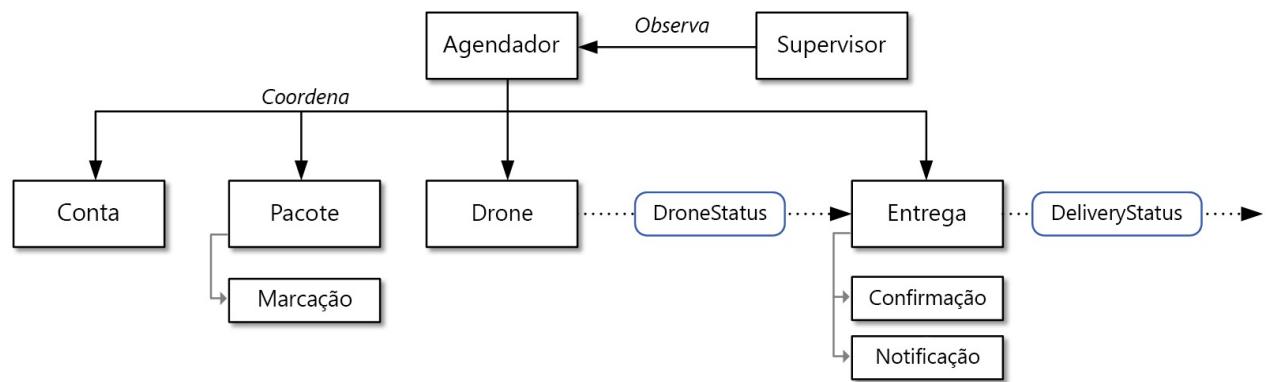


Há dois eventos de domínio:

- Enquanto um drone está em trânsito, a entidade Drone envia eventos DroneStatus que descrevem a localização do drone e seu status (em curso, descarregou).
- A entidade Entrega envia eventos DeliveryTracking sempre que o estágio de uma entrega é alterado. Isso inclui DeliveryCreated, DeliveryRescheduled, DeliveryHeadedToDropoff e DeliveryCompleted.

Observe que esses eventos descrevem itens que são significativos no modelo de domínio. Eles descrevem algo sobre o domínio e não estão ligados a um constructo de linguagem de programação específico.

A equipe de desenvolvimento identificou mais uma área de funcionalidade que não se adéqua a nenhuma das entidades descritas até agora. Alguma parte do sistema precisa coordenar todas as etapas envolvidas no agendamento ou atualização de uma entrega. Portanto, a equipe de desenvolvimento adicionou dois serviços de domínio ao design: um Agendador que coordena as etapas e um Supervisor que monitora o status de cada etapa para detectar se alguma etapa falhou ou atingiu o tempo limite. Essa é uma variação do Padrão de Supervisor de Agente do Agendador (Coordena um conjunto de ações distribuídas como uma única operação. Se qualquer uma das ações falhar, tenta tratar as falhas de forma transparente ou então desfaz o trabalho que foi executado para que toda a operação tenha êxito ou falhe como um todo).



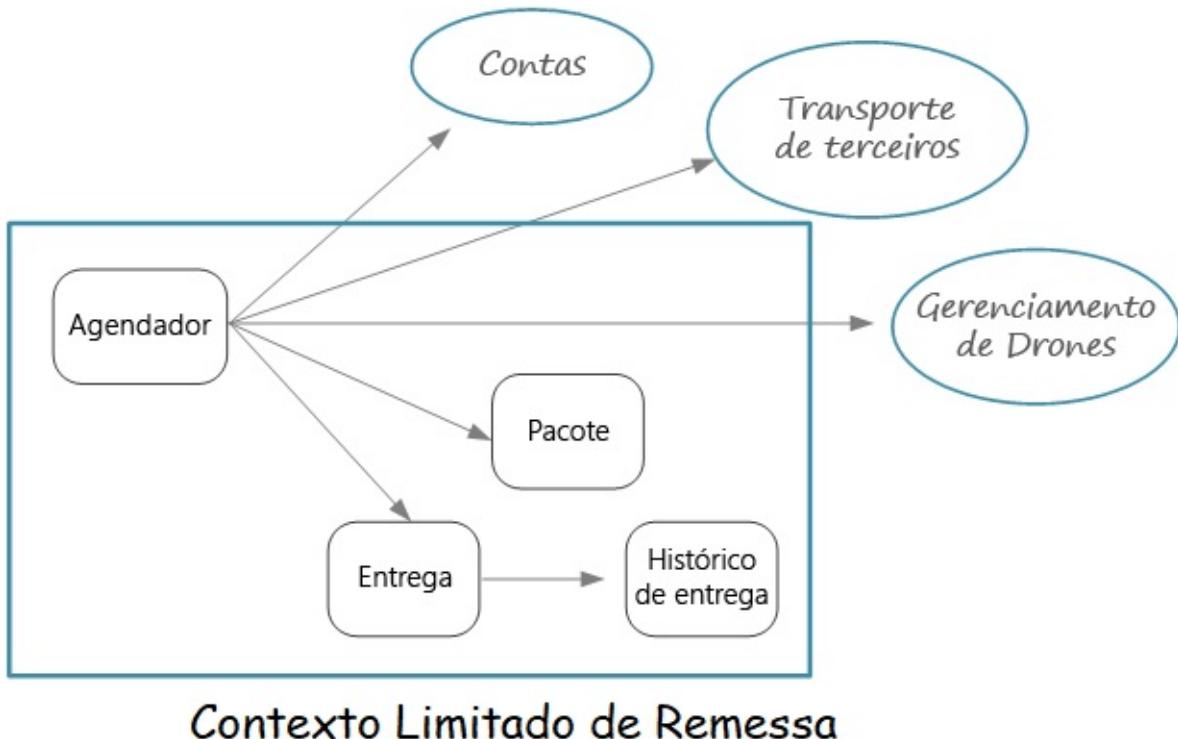
Fonte

- <https://docs.microsoft.com/pt-br/azure/architecture/microservices/domain-analysis>

Identificando limites de microsserviço

Qual é o tamanho correto de um microsserviço? Muitas vezes você ouve algo como, "não muito grande e não muito pequeno" — e, embora isso seja correto, não é muito útil, na prática. Mas, se você iniciar de um modelo de domínio cuidadosamente desenvolvido, será mais fácil pensar sobre microsserviços.

- Diagrama de contextos limitados



Do modelo de domínio a microsserviços

Anteriormente, definimos um conjunto de contextos limitados para o aplicativo de entrega por drone. Então, analisamos mais de perto um desses contextos limitados, o contexto limitado de Remessa, e identificamos um conjunto de entidades, agregados e serviços de domínio para esse contexto limitado.

Agora estamos prontos para passar do modelo de domínio para o design do aplicativo. Aqui está uma abordagem que você pode usar para derivar microsserviços do modelo de domínio.

1. Comece com um contexto limitado. Em geral, a funcionalidade em um microsserviço não deve abranger mais de um contexto limitado. Por definição, um contexto limitado marca o limite de um modelo de domínio específico. Se você achar que um microsserviço combina diferentes modelos de domínio, é sinal de que precisa voltar e refinar sua análise de domínio.
2. Em seguida, examine os agregados no seu modelo de domínio. Agregados são, frequentemente, bons candidatos a microsserviços. Um agregado bem projetado exibe muitas das características de um microsserviço bem projetado, por exemplo:
3. Agregados são derivados de requisitos de negócios, e não de questões técnicas, como acesso a dados ou mensagens.
4. Um agregado deve ter alta coesão funcional.
5. Um agregado é um limite de persistência.
6. Agregados devem ser acoplados de forma flexível.
7. Serviços de domínio também são bons candidatos a microsserviços. Serviços de domínio são operações sem estado entre vários agregados. Um exemplo típico é um fluxo de trabalho que envolve vários microsserviços.

8. Finalmente, considere os requisitos não funcionais. Observe fatores como tamanho da equipe, tipos de dados, tecnologias, requisitos de escalabilidade, requisitos de disponibilidade e requisitos de segurança. Esses fatores podem levá-lo a decompor ainda mais um microsserviço em dois ou mais serviços menores, ou fazer o contrário e combinar vários microsserviços em um.

Depois de identificar os microsserviços em seu aplicativo, valide o design conforme os seguintes critérios:

- Cada serviço tem uma única responsabilidade.
- Não há nenhuma chamada de comunicação entre os serviços. Se a divisão da funcionalidade em dois serviços gerar excesso de comunicação, isso poderá ser um sintoma de que essas funções pertencem ao mesmo serviço.
- Cada serviço é pequeno o suficiente para ser criado por uma pequena equipe trabalhando de forma independente.
- Não há nenhuma interdependência que exige que dois ou mais serviços sejam implantados em sincronia. Deve ser sempre possível implantar um serviço sem redistribuir outros serviços.
- Os serviços não estão acoplados de forma firme e podem evoluir de forma independente.
- Os limites do serviço não criam problemas de consistência ou integridade de dados. Às vezes, é importante manter a consistência dos dados colocando a funcionalidade em um único microsserviço. Dito isto, considere se você realmente precisa de consistência forte. Existem estratégias para abordar a consistência eventual em um sistema distribuído, e os benefícios dos serviços de decomposição geralmente superam os desafios de gerenciar a consistência eventual.

Acima de tudo, é importante ser pragmático e lembrar-se de que o design orientado por domínio é um processo iterativo. Em caso de dúvida, comece com microsserviços de granulação grosseira. É mais fácil dividir um microsserviço em dois serviços menores do que refatorar a funcionalidade em vários microsserviços existentes.

Entrega por Drones: Definindo os microsserviços

Lembre-se de que a equipe de desenvolvimento identificou as quatro agregações — Entrega, Pacote, Drone e Conta — e dois serviços de domínio, Agendador e o Supervisor.

Entrega e Pacote são candidatos óbvios a microsserviços. O Agendador e o Supervisor coordenam as atividades executadas por outros microsserviços, então, faz sentido implementar esses serviços de domínio como microsserviços.

Drone e Conta são interessantes porque pertencem a outros contextos limitados. Uma opção é Agendador chamar contextos limitados de Drone e Conta diretamente. Outra opção é criar microsserviços de Drone e Conta dentro do contexto de envio limitado. Esses microsserviços fariam a mediação entre os contextos delimitados, expondo as APIs ou os esquemas de dados mais adequados ao contexto de Remessa.

Os detalhes dos contextos limitados de Drone e Conta estão além do escopo desta orientação. Mas, aqui estão alguns fatores a considerar nessa situação:

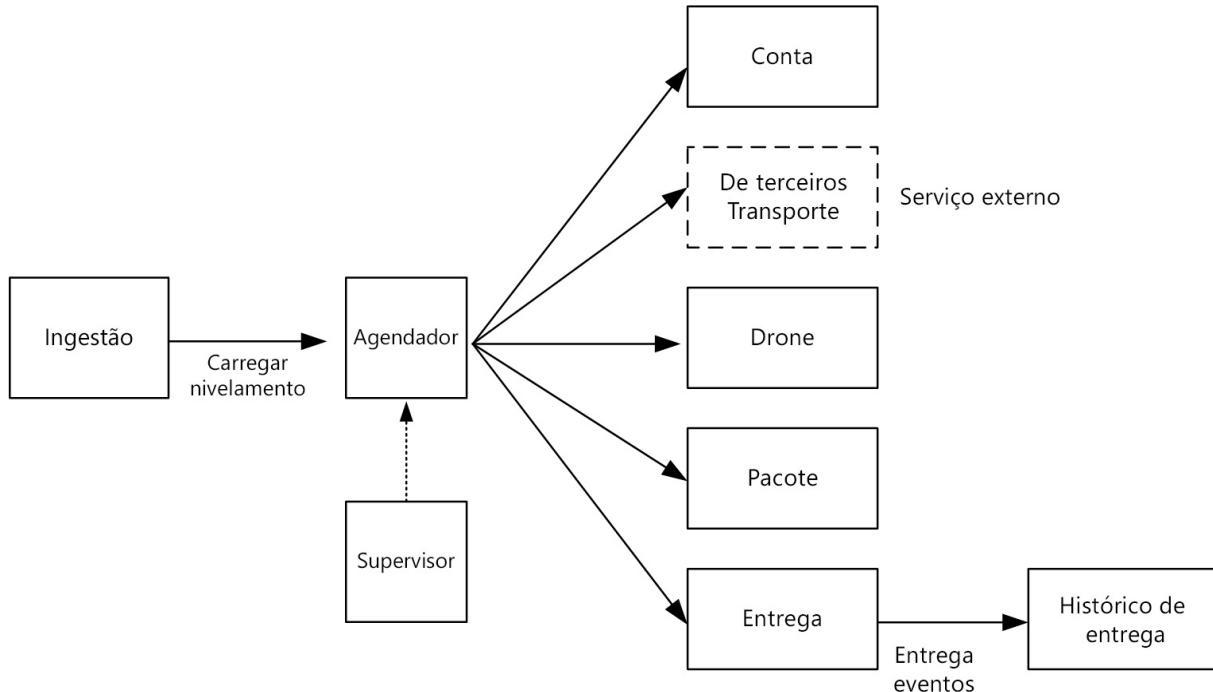
- Qual é a sobrecarga de rede ao chamar diretamente outro contexto limitado?
- O esquema de dados para o outro contexto limitado é adequado para esse contexto ou é melhor ter um esquema adaptado a esse contexto limitado?
- O outro contexto limitado é sistema legado? Em caso afirmativo, crie um serviço que atue como camada anticorrupção para converter entre o sistema legado e o aplicativo moderno.
- Como é a estrutura de equipe? É fácil se comunicar com a equipe responsável pelo outro contexto limitado? Se não for fácil, a criação de um serviço que faça a mediação entre os dois contextos pode ajudar a atenuar o custo da comunicação entre equipes.

Até agora, não consideramos nenhum requisito não funcional. Pensando nos requisitos de taxa de transferência do aplicativo, a equipe de desenvolvimento decidiu criar um microsserviço de Ingestão separado, responsável pela ingestão de solicitações do cliente. Esse microsserviço implementará o nivelamento de carga, colocando as solicitações recebidas em um buffer para processamento. O Agendador fará a leitura das solicitações do buffer e executará o fluxo de trabalho.

Requisitos não funcionais levaram a equipe a criar um serviço adicional. Todos os serviços até agora foram sobre o processo de agendamento e entrega de pacotes em tempo real. Mas o sistema também precisa armazenar o histórico de cada entrega no armazenamento de longo prazo para análise dos dados. A equipe considerou atribuir essa responsabilidade ao serviço de Entrega. No entanto, os requisitos de armazenamento de dados são muito diferentes para a análise histórica versus operações em andamento. Portanto, a equipe decidiu criar um serviço de Histórico de Entrega separado, que escutará eventos DeliveryTracking do serviço de Entrega e gravará os eventos no armazenamento de longo prazo.

O diagrama a seguir mostra o design neste ponto:

- Diagrama de design



Escolhendo um opção de computação

O termo computação refere-se ao modelo de hospedagem para os recursos de computação em que seu aplicativo é executado. Para uma arquitetura de microsserviços, duas abordagens são especialmente populares:

- Um orquestrador de serviços que gerencia serviços em execução em nós dedicado (VMs).
- Uma arquitetura sem servidor que funciona como um serviço (FaaS).

Embora essas não sejam as únicas opções, são abordagens comprovadas para criação de microsserviços. Um aplicativo pode incluir ambas as abordagens.

Orquestradores de serviço

Um orquestrador manipula tarefas relacionadas à implantação e ao gerenciamento de um conjunto de serviços. Essas tarefas incluem a colocação de serviços em nós, monitoramento da integridade dos serviços, reinicialização de serviços não íntegros, balanceamento de carga de tráfego de rede em instâncias de serviço, descoberta de serviço, dimensionamento do número de instâncias de um serviço e aplicação das atualizações de configuração. Entre os orquestradores populares estão Kubernetes, DC/OS, Docker Swarm e Service Fabric.

Contêineres

Às vezes, as pessoas falam sobre contêineres e microsserviços como se fossem a mesma coisa. Embora isso não seja verdade — você não precisa de contêineres para criar microsserviços — os contêineres têm alguns benefícios particularmente relevantes para microsserviços, como:

- **Portabilidade.** Uma imagem de contêiner é um pacote autônomo, que é executado sem a necessidade de instalar bibliotecas ou outras dependências. Isso facilita a implantação. Contêineres podem ser iniciados e interrompidos rapidamente, portanto, você pode criar novas instâncias para lidar com mais carga ou para se recuperar de falhas de nó.
- **Densidade.** Contêineres são leves em comparação com a execução de uma máquina virtual, porque eles compartilham os recursos do sistema operacional. Isso possibilita empacotar vários contêineres em um único nó, o que é especialmente útil quando o aplicativo é composto por muitos pequenos serviços.
- **Isolamento de recurso.** Você pode limitar a quantidade de memória e a CPU disponível para um contêiner, o que ajuda a garantir que um processo sem controle não esgote os recursos do host.

Server Less (Funções como um serviço)

Com uma arquitetura sem servidor, não é possível administrar VMs nem a infraestrutura de rede virtual. Em vez disso, você implanta o código, e o serviço de hospedagem coloca esse código em uma VM e o executa. Essa abordagem tende a favorecer pequenas funções granulares que sejam coordenadas usando gatilhos baseados em eventos. Por exemplo, uma mensagem colocada em uma fila pode disparar uma função que lê da fila e processa a mensagem.

Orquestrador ou Server Less?

Aqui estão alguns fatores a considerar ao escolher entre uma abordagem de orquestrador e uma abordagem sem servidor.

Capacidade de gerenciamento um aplicativo sem servidor é fácil de gerenciar, porque a plataforma gerencia todos os recursos de computação por você. Um orquestrador abstrai alguns aspectos de gerenciamento e configuração de um cluster, mas ele não oculta completamente as VMs subjacentes. Com um orquestrador, você terá de pensar sobre problemas como,平衡amento de carga, uso de CPU e de memória e rede.

Flexibilidade e controle. Um orquestrador fornece muito controle sobre como configurar e gerenciar seus serviços e o cluster. Adesvantagem é a complexidade adicional. Com uma arquitetura sem servidor, você perde um pouco do controle porque esses detalhes são abstraídos.

Portabilidade. Todos os orquestradores listados aqui (Kubernetes, DC/OS, Docker Swarm e Service Fabric) podem ser executados localmente ou em várias nuvens públicas.

Integração de aplicativos. Pode ser um desafio criar um aplicativo complexo usando uma arquitetura sem servidor.

Custo. Com um orquestrador, você paga pelas VMs em execução no cluster. Com um aplicativo sem servidor, você paga apenas pelos recursos de computação real consumidos. Em ambos os casos, você precisa considerar o custo dos serviços adicionais, como armazenamento, bancos de dados e serviços de mensagens.

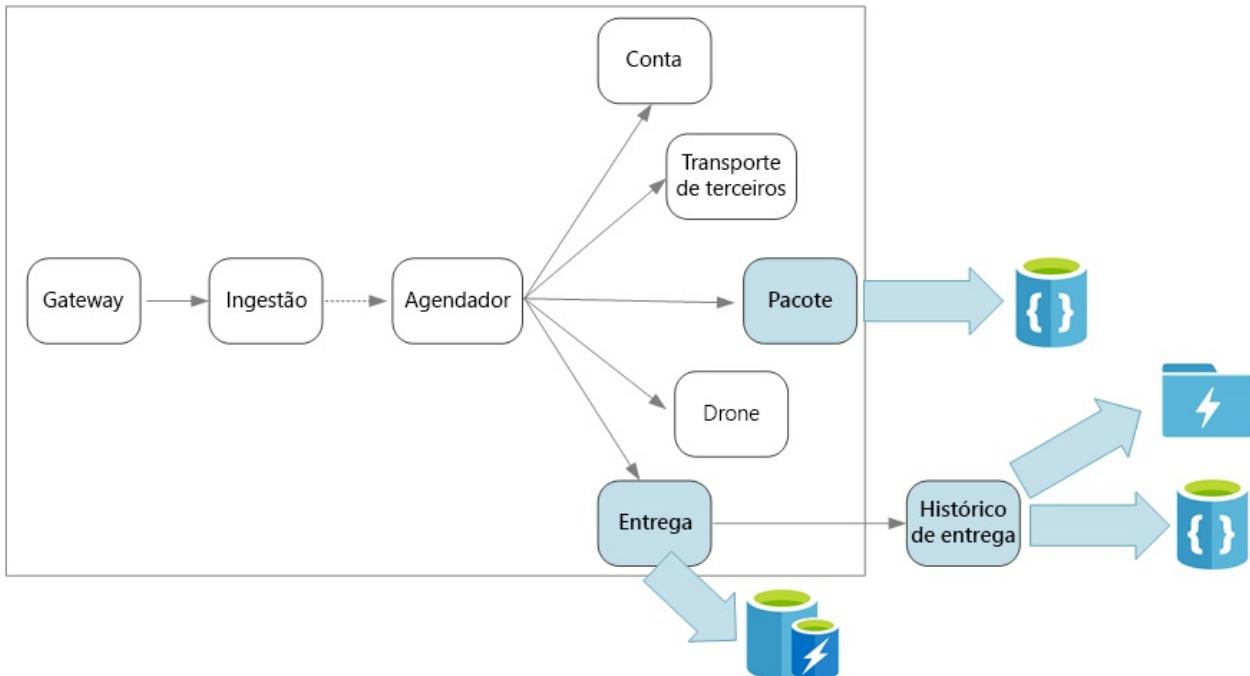
Escalabilidade. Em geral, quando se trabalha sem servidor, há o dimensionamento automático para atender à demanda, com base no número de eventos recebidos. Com um orquestrador, você pode expandir, aumentando o número de instâncias de serviço em execução no cluster. Você também pode dimensionar adicionando VMs ao cluster.

Fonte

- <https://docs.microsoft.com/pt-br/azure/architecture/microservices/microservice-boundaries>

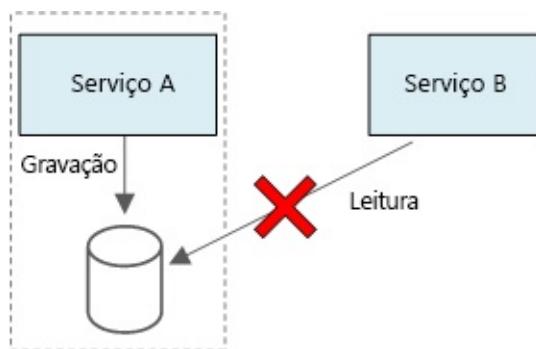
Considerações de dados

Quais são as considerações para gerenciar dados em uma arquitetura de microsserviços? Os principais desafios são como cada microsserviço gerencia os próprios dados, a integridade e a consistência de dados.



Um princípio básico dos microsserviços é que cada serviço gerencia seus próprios dados. Dois serviços não devem compartilhar um armazenamento de dados. Em vez disso, cada serviço é responsável pelo próprio armazenamento de dados particular, que não pode ser acessado diretamente por outros serviços.

O motivo para essa regra é evitar acoplamento não intencional entre serviços, que poderá ocorrer se os serviços compartilharem os mesmos esquemas de dados subjacentes. Se houver alteração de esquema de dados, a alteração deverá ser coordenada em cada serviço que dependa desse banco de dados. Ao isolar o armazenamento de dados de cada serviço, podemos limitar o escopo da alteração e preservar a agilidade de implantações verdadeiramente independentes. Outro motivo é que cada microsserviço pode ter seus próprios modelos de dados, consultas ou padrões de leitura/gravação. O uso de um armazenamento de dados compartilhado limita a capacidade de cada equipe de otimizar o armazenamento para seu serviço específico.



Essa abordagem leva naturalmente a uma persistência poliglota — o uso de várias tecnologias de armazenamento de dados em um único aplicativo. Um serviço pode exigir os recursos de esquema na leitura de um banco de dados de documento. Outro pode precisar da integridade referencial fornecida por um RDBMS. Cada equipe pode fazer a melhor escolha para seu serviço. Para obter mais informações sobre os princípios gerais da persistência poliglota, consulte Usar o melhor armazenamento de dados para o trabalho.

Observação: Não há problema em os serviços compartilharem o mesmo servidor de banco de dados físico. O problema ocorre quando os serviços compartilham o mesmo esquema ou a leitura e a gravação para o mesmo conjunto de tabelas de banco de dados.

Desafios

Alguns desafios são provenientes dessa abordagem distribuída ao gerenciamento de dados. Primeiro, pode haver redundância entre os armazenamentos de dados, com o mesmo item de dados aparecendo em vários lugares. Por exemplo, dados podem ser armazenados como parte de uma transação, armazenados em outro local para análise, geração de relatórios ou arquivamento. Dados duplicados ou particionados podem levar a problemas de integridade e consistência. Quando as relações de dados abrangem vários serviços, não é possível usar técnicas de gerenciamento de dados tradicionais para impor as relações.

A modelagem de dados tradicional usa a regra "um fato em um único local". Cada entidade aparece exatamente uma vez no esquema. Outras entidades podem manter referências a ele, mas não duplicá-lo. Avantagem óbvia à abordagem tradicional é que as atualizações são feitas em um único local, o que evita problemas com consistência de dados. Em uma arquitetura de microsserviços, considere como as atualizações são propagadas entre os serviços e como gerenciar eventual consistência quando os dados aparecem em vários locais sem uma consistência forte.

Abordagens ao gerenciamento de dados

Não há uma abordagem adequada para todos os casos, mas, a seguir, temos algumas diretrizes gerais que podem ajudá-lo no gerenciamento de dados em uma arquitetura de microsserviços.

- Adote consistência eventual quando possível. Entenda os lugares no sistema onde uma consistência forte ou as transações ACID são necessárias e os locais onde a consistência eventual é aceitável.
- Quando você precisa de garantias de consistência forte, um serviço pode representar a fonte de verdade para determinada entidade, exposta por meio de uma API. Outros serviços podem conter sua própria cópia ou um subconjunto de dados, que eventualmente são consistentes com os dados mestres, mas não são considerados como fonte da verdade. Por exemplo, imagine um sistema de comércio eletrônico com um serviço de pedido de cliente e um serviço de recomendação. O serviço de recomendação pode escutar eventos do serviço de pedidos, mas se um cliente solicitar reembolso, será o serviço de pedidos, e não o serviço de recomendação, que terá todo o histórico da transação.
- Para transações, use padrões como Supervisor de Agente do Agendador (coordena um conjunto de ações distribuídas como uma única operação. Se qualquer uma das ações falhar, tenta tratar as falhas de forma transparente ou então desfaz o trabalho que foi executado para que toda a operação tenha êxito ou falhe como um todo) e transações de compensação (desfaz o trabalho executado por uma série de etapas, que juntas definem uma operação eventualmente consistente, se uma ou mais das etapas falhar) para manter os dados consistentes em vários serviços. Talvez seja necessário armazenar dados adicionais que capturem o estado de uma unidade de trabalho que abrange vários serviços, para evitar uma falha parcial entre os serviços. Por exemplo, manter um item de trabalho em uma fila durável, enquanto uma transação de várias etapas está em andamento.
- Armazene apenas os dados que um serviço precisa. Um serviço pode precisar apenas de um subconjunto de informações sobre uma entidade de domínio. Por exemplo, no contexto limitado de Remessa, precisamos saber qual cliente está associado a uma entrega específica. Mas não precisamos do endereço de cobrança do cliente — que é gerenciado pelo contexto limitado de Contas. Pensar cuidadosamente o domínio e usar uma abordagem DDD, pode ajudar.
- Considere se os serviços são coerentes e acoplados de forma flexível. Se dois serviços trocam informações continuamente entre si, resultando em APIs de conversa, será necessário redigir os limites do serviço, mesclando dois serviços ou refatorando suas funcionalidades.

- Use um estilo de arquitetura baseado em eventos. Nesse estilo de arquitetura, um serviço publica um evento quando há alterações em suas entidades ou modelos públicos. Serviços interessados podem assinar esses eventos. Por exemplo, outro serviço pode usar os eventos para construir uma visão materializada dos dados que seja mais adequada a consultas.
- Um serviço que tem eventos deve publicar um esquema que pode ser usado para automatizar a serialização e desserialização de eventos, para evitar acoplamento entre publicadores e assinantes. Considere o esquema JSON ou uma estrutura como Microsoft Bond, Protobuf ou Avro.
- Em grande escala, os eventos podem se tornar um gargalo no sistema, portanto, considere usar agregação ou processamento em lote para reduzir a carga total.

Entrega por Drones: Escolhendo os repositórios de dados

Mesmo com apenas alguns serviços, o contexto limitado de Remessa ilustra vários dos pontos abordados nesta seção.

Quando um usuário agenda uma nova entrega, a solicitação do cliente inclui informações sobre a entrega, como locais de retirada e entrega, e sobre o pacote, como tamanho e peso. Essas informações definem uma unidade de trabalho, que o serviço de ingestão envia para os Hubs de Eventos. É importante que a unidade de trabalho permaneça no armazenamento persistente enquanto o serviço do Agendador executa o fluxo de trabalho, para que nenhuma solicitação de entrega seja perdida.

Os vários serviços de back-end cuidam de diferentes partes da informação na solicitação e também têm diferentes perfis de leitura e gravação.

Serviço de Entrega

O serviço de Entrega armazena informações sobre cada entrega agendada no momento ou em andamento. Ele escuta eventos de drone e acompanha o status das entregas em andamento. Ele também envia eventos de domínio com atualizações de status da entrega.

Espera-se que os usuários verifiquem com frequência o status de uma entrega enquanto aguardam o pacote. Portanto, o serviço de Entrega exige um armazenamento de dados que enfatize a taxa de transferência (leitura e gravação) em um armazenamento de longo prazo. Além disso, o serviço de Entrega não realiza consultas nem análises complexas, simplesmente busca o status mais recente de determinada entrega. A equipe do serviço de Entrega escolheu o Cache Redis por seu alto desempenho de leitura e gravação. As informações armazenadas no Redis têm duração relativamente curta. Quando uma entrega é concluída, o serviço de histórico da entrega é o sistema de registro.

Serviço de histórico da entrega

O serviço de histórico da entrega escuta os eventos de status do serviço de entrega. Ele armazena esses dados em armazenamento de longo prazo. Há dois diferentes casos de uso de dados históricos, com diferentes requisitos de armazenamento de dados.

O primeiro cenário é a agregação de dados com o objetivo de análise, a fim de otimizar os negócios ou melhorar a qualidade do serviço. Observe que o serviço de histórico de entrega não executa análise real dos dados. Só é responsável pela inclusão e pelo armazenamento. Para este cenário, o armazenamento deve ser otimizado para análise de dados em um grande conjunto de dados, usando uma abordagem de esquema na leitura para acomodar uma variedade de fontes de dados.

Serviço de pacote

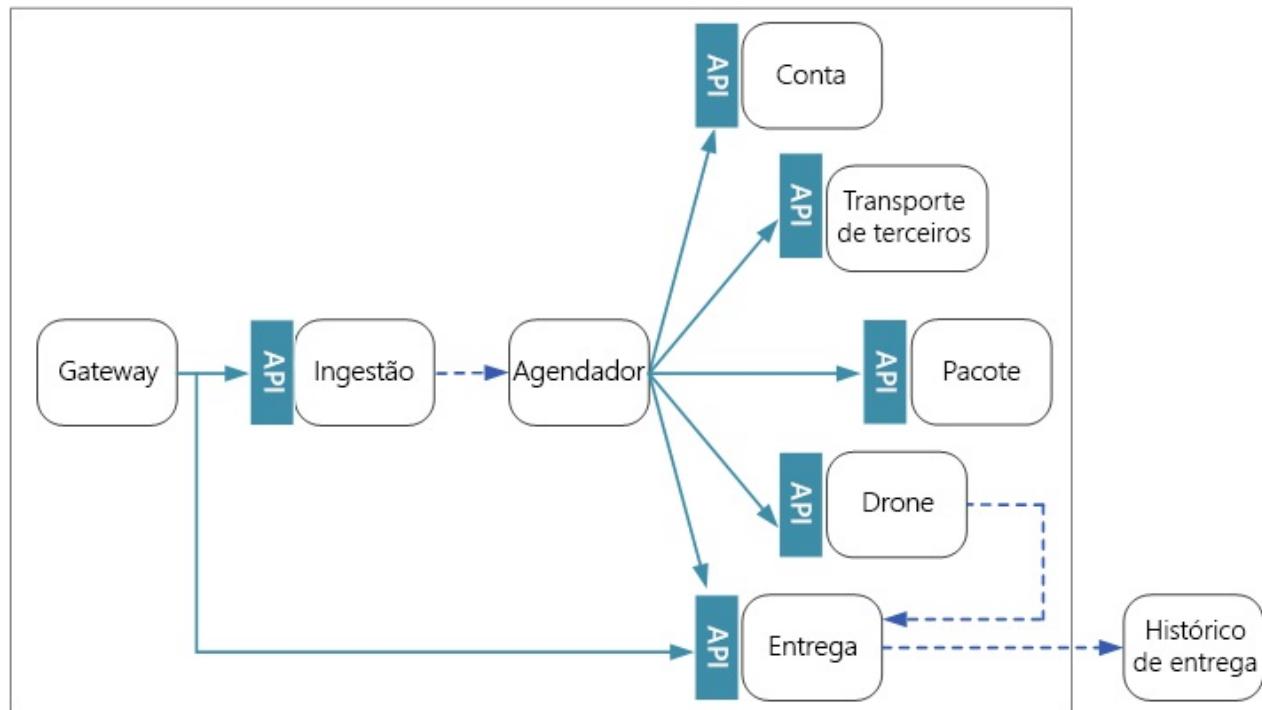
O serviço do pacote armazena informações sobre todos os pacotes. Os requisitos de armazenamento para o pacote são:

- Armazenamento de longo prazo;
- Capaz de lidar com um alto volume de pacotes, que requerem alta taxa de transferência de gravação;
- Suporte a consultas simples por ID do pacote. Nenhum requisito de integridade referencial ou junções complexas;

Como os dados do pacote não são relacionais, um banco de dados orientado a documentos é apropriado.

Comunicação entre serviços

A comunicação entre os microsserviços deve ser eficiente e robusta. Com vários serviços pequenos interagindo para concluir uma única transação, isso pode ser um desafio. Vamos examinar as compensações entre o sistema de mensagens assíncrono em relação às APIs síncronas. Em seguida, vamos observar alguns dos desafios ao criar uma comunicação entre serviços resiliente e a função que uma malha de serviço pode desempenhar.



Desafios

Aqui estão alguns dos principais desafios decorrentes da comunicação de serviço a serviço. As malhas de serviço, descritas mais adiante, são projetadas para lidar com muitos desses desafios.

Resiliência. Pode haver dezenas ou até mesmo centenas de instâncias de qualquer microsserviço. Uma instância pode falhar por vários motivos. Pode haver uma falha no nível de nó, como uma falha de hardware ou uma reinicialização da VM. Uma instância pode falhar ou ficar sobrecarregada com solicitações e, assim, impossibilitada de processar solicitações novas. Qualquer um desses eventos pode fazer com que uma chamada de rede falhe. Há dois padrões de design que podem ajudar a tornar as chamadas de rede de serviço mais resilientes:

- **Repetição**¹. Uma chamada de rede pode falhar por causa de uma falha temporária que desaparece por si só. Em vez de falhar totalmente, o autor da chamada normalmente deverá repetir a operação um determinado número de vezes ou até que um período de tempo limite configurado expire. No entanto, se uma operação não for idempotente, as repetições poderão causar efeitos colaterais não intencionais. Achamada original talvez seja bem-sucedida, mas o autor da chamada nunca receberá uma resposta. Se o autor da chamada fizer novas tentativas, a operação poderá ser invocada duas vezes. Em geral, não é seguro repetir os métodos POST ou PATCH, uma vez que não há garantias de que eles sejam idempotentes.

¹. Permite que um aplicativo trate falhas transitórias quando tentar se conectar a um serviço ou recurso de rede ao repetir de forma transparente uma operação com falha. Isso pode melhorar a estabilidade do aplicativo. ↩

- **Disjuntor (circuit breaker)**². Um número excessivo de solicitações com falha pode causar um gargalo, já que as solicitações pendentes se acumulam na fila. Essas solicitações bloqueadas podem reter recursos críticos do sistema, como memória, threads, conexões de banco de dados e outros, e provocar falhas em cascata. O uso do

padrão de Disjuntor pode impedir que um serviço tente repetir várias vezes uma operação que provavelmente falhará.

2. Trate as falhas que possam consumir uma quantidade variável de tempo para serem recuperadas ao se conectar a um serviço ou recurso remoto. Isso pode melhorar a estabilidade e a resiliência de um aplicativo. ↵

Balanceamento de carga. Quando o serviço "A" chama o serviço "B", a solicitação deve alcançar uma instância em execução do serviço "B". No Kubernetes, o tipo de recurso Service fornece um endereço IP estável para um grupo de pods. O tráfego de rede para o endereço IP do serviço é encaminhado para um pod por meio de regras de iptable. Por padrão, um pod aleatório é escolhido. Uma malha de serviço pode fornecer algoritmos de balanceamento de carga mais inteligentes com base na latência observada ou em outras métricas.

Rastreamento distribuído. Uma única transação pode abranger vários serviços. Isso pode dificultar o monitoramento do desempenho geral e da integridade do sistema. Mesmo que cada serviço gere logs e métricas, sem alguma forma associá-los, eles serão de utilidade limitada.

Controle de versão do serviço. Quando uma equipe implanta uma nova versão de um serviço, ela deve evitar a interrupção de qualquer outro serviço ou cliente externo que dependa dele. Além disso, talvez você queira executar várias versões de uma serviço lado a lado e rotear solicitações para uma versão específica.

Criptografia de TLS e autenticação de TLS mútua. Por motivos de segurança, convém criptografar o tráfego entre os serviços com TLS e usar a autenticação de TLS mútua para autenticar os autores de chamadas.

Sistema de mensagens síncrono versus assíncrono

Há dois padrões básicos de mensagens que os microsserviços podem utilizar para se comunicarem com outros microsserviços:

1. Comunicação síncrona. Nesse padrão, um serviço chama uma API que outro serviço expõe usando um protocolo, como o HTTP ou o gRPC. Esta opção é um padrão de sistema de mensagens síncrono porque o autor da chamada aguarda uma resposta do receptor.
2. Transmissão de mensagens assíncronas. Nesse padrão, um serviço envia a mensagem sem aguardar uma resposta, e um ou mais serviços processam a mensagem de maneira assíncrona.

É importante distinguir entre uma E/S assíncrona e um protocolo assíncrono. Uma E/S assíncrona indica que o thread de chamada não será bloqueado enquanto a E/S não for concluída. Isso é importante para o desempenho, mas é um detalhe de implementação em termos de arquitetura. Um protocolo assíncrono indica que o remetente não aguardará uma resposta. O HTTP será um protocolo síncrono, mesmo que um cliente HTTP utilize a E/S assíncrona ao enviar uma solicitação.

Há vantagens e desvantagens para cada padrão. O paradigma de solicitação/resposta é bem compreendido, de modo que a criação de uma API pode parecer mais natural do que a criação de um sistema de mensagens. No entanto, o sistema de mensagens assíncrono apresenta algumas vantagens que podem ser muito úteis em uma arquitetura de microsserviços:

- **Acoplamento reduzido.** O remetente da mensagem não precisa saber sobre o consumidor.
- **Vários assinantes.** Ao usar um modelo pub/sub, vários consumidores podem assinar para receber eventos.
- **Isolamento de falha.** Se o consumidor falhar, o remetente ainda poderá enviar mensagens. As mensagens serão removidas quando o consumidor recuperá-las. Essa capacidade é especialmente útil em uma arquitetura de microsserviços, uma vez que cada serviço tem seu próprio ciclo de vida. Um serviço pode se tornar indisponível ou ser substituído por uma versão mais recente a qualquer momento. O sistema de mensagens assíncrono pode controlar o tempo de inatividade intermitente. Por outro lado, as APIs síncronas exigem que o serviço de downstream esteja disponível, ou a operação falhará.

- **Capacidade de resposta.** Um serviço de upstream poderá responder mais rapidamente se ele não aguardar os serviços de downstream. Isso é especialmente útil em uma arquitetura de microsserviços. Se houver uma cadeia de dependências de serviço (o serviço A chama o serviço B, que chama o C e assim por diante), a espera pelas chamadas síncronas poderá adicionar quantidades de latência inaceitáveis.
- **Nivelamento de carga.** Uma fila pode atuar como um buffer para nivelar a carga de trabalho, de modo que os destinatários possam processar as mensagens em seu próprio ritmo.
- **Fluxos de trabalho.** As filas podem ser usadas para gerenciar um fluxo de trabalho, marcando a mensagem após cada etapa no fluxo de trabalho.

No entanto, há também alguns desafios para usar o sistema de mensagens assíncrono com eficiência.

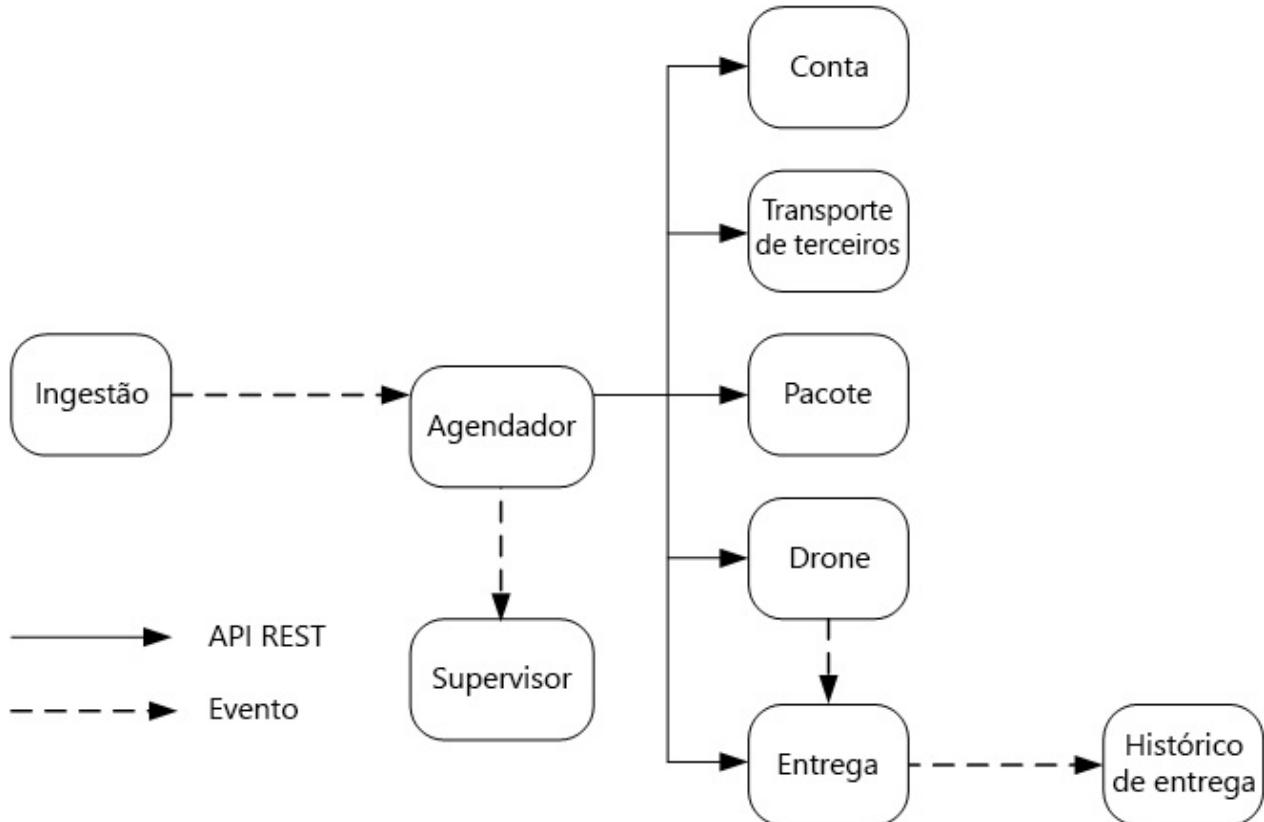
- **Acoplamento com a infraestrutura de mensagens.** O uso de uma infraestrutura de mensagens específica pode causar um acoplamento estreito com essa infraestrutura. Será difícil mudar para outra infraestrutura de mensagens posteriormente.
- **Latência.** A latência de ponta a ponta para uma operação poderá ser alta se as filas de mensagens forem preenchidas.
- **Custo.** Nas taxas de transferência altas, o custo monetário da infraestrutura de mensagens pode ser significativo.
- **Complexidade.** Controlar o sistema de mensagens assíncrono não é uma tarefa fácil. Por exemplo, você deve lidar com mensagens duplicadas, seja ao eliminar a duplicação ou ao tornar as operações idempotentes. Também é difícil implementar a semântica de solicitação-resposta usando o sistema de mensagens assíncrono. Para enviar uma resposta, você precisa de outra fila, além de uma maneira de correlacionar as mensagens de solicitação e de resposta.
- **Taxa de transferência.** Se as mensagens exigirem uma semântica de fila, a fila poderá se tornar um gargalo no sistema. Cada mensagem exige, pelo menos, uma operação de fila e uma operação de remoção da fila. Além disso, a semântica de fila geralmente exige algum tipo de bloqueio na infraestrutura de mensagens. Se a fila for um serviço gerenciado, poderá haver latência adicional, uma vez que a fila é externa à rede virtual do cluster. Você pode mitigar esses problemas por meio de mensagens de lote, mas isso complica o código. Se as mensagens não exigirem a semântica de fila, você poderá usar o fluxo de evento em vez de uma fila. Para obter mais informações, consulte Arquitetura orientada a eventos.

Entrega por Drones: escolhendo os padrões de mensagens

Com essas considerações em mente, a equipe de desenvolvimento fez as seguintes opções de design para o aplicativo de entrega por drone:

- O serviço Ingestão expõe uma API de REST pública que os aplicativos clientes usam para agendar, atualizar ou cancelar as entregas.
- O serviço Ingestão usa os Hubs de Eventos para enviar mensagens assíncronas para o serviço Agendador. As mensagens assíncronas são necessárias para implementar o nívelamento de carregamento que é necessário para a ingestão de dados.
- Os serviços Contabilidade, Entrega, Empacotamento, Drone e Transporte de Terceiros expõem APIs de REST internas. O serviço Agendador chama essas APIs para executar uma solicitação de usuário. Um motivo para usar APIs síncronas é que o Agendador precisa obter uma resposta de cada um dos serviços de downstream. Uma falha em qualquer um dos serviços de downstream indica que toda a operação falhou. No entanto, um problema potencial é a quantidade de latência que é introduzida ao chamar os serviços de back-end.
- Se algum serviço de downstream tiver uma falha não transitória, toda a transação deverá ser marcada como com falha. Para lidar com este caso, o serviço Agendador envia uma mensagem assíncrona para o Supervisor, para que o Supervisor possa agendar transações de compensação.
- O serviço Entrega expõe uma API pública que os clientes podem usar para obter o status de uma entrega.

- Enquanto um drone está em trânsito, o serviço Drone envia eventos que contêm a localização e o status atual do drone. O serviço Entrega segue esses eventos para acompanhar o status de uma entrega.
- Quando o status de uma entrega é alterado, o serviço Entrega envia um evento de status de entrega, como `DeliveryCreated` ou `DeliveryCompleted`. Qualquer serviço pode assinar esses eventos. No projeto atual, o serviço Entrega é o único assinante, mas poderá haver outros assinantes posteriormente. Por exemplo, os eventos poderão ir para um serviço de análise em tempo real. E, uma vez que o Agendador não precisa aguardar uma resposta, a adição de mais assinantes não afeta o caminho principal do fluxo de trabalho.



Observe que os eventos de status de entrega são derivados de eventos de localização de drone. Por exemplo, quando um drone alcança um local de entrega e solta um pacote, o serviço Entrega converte isso em um evento `DeliveryCompleted`. Este é um exemplo de raciocínio em termos de modelos de domínio. Conforme descrito anteriormente, o Gerenciamento de Drone pertence a um contexto limitado separado. Os eventos de drone transmitem a localização física de um drone. Por outro lado, os eventos de entrega representam as alterações no status de uma entrega, que é uma entidade de negócios diferente.

Usando uma malha de serviço

Uma malha de serviço é uma camada de software que gerencia a comunicação de serviço a serviço. As malhas de serviço são projetadas para abordar muitas das preocupações listadas na seção anterior e para levar a responsabilidade por essas preocupações para longe dos próprios microsserviços e para dentro de uma camada compartilhada. Amalha de serviço atua como um proxy que intercepta a comunicação de rede entre os microsserviços no cluster.

Observação: Amalha de serviço é um exemplo do Padrão embaixador —, um serviço auxiliar que envia as solicitações de rede em nome do aplicativo.

No momento, as principais opções para uma malha de serviço no Kubernetes são linkerd e Istio. Ambas as tecnologias estão evoluindo rapidamente. No entanto, alguns recursos que o linkerd e o Istio têm em comum incluem:

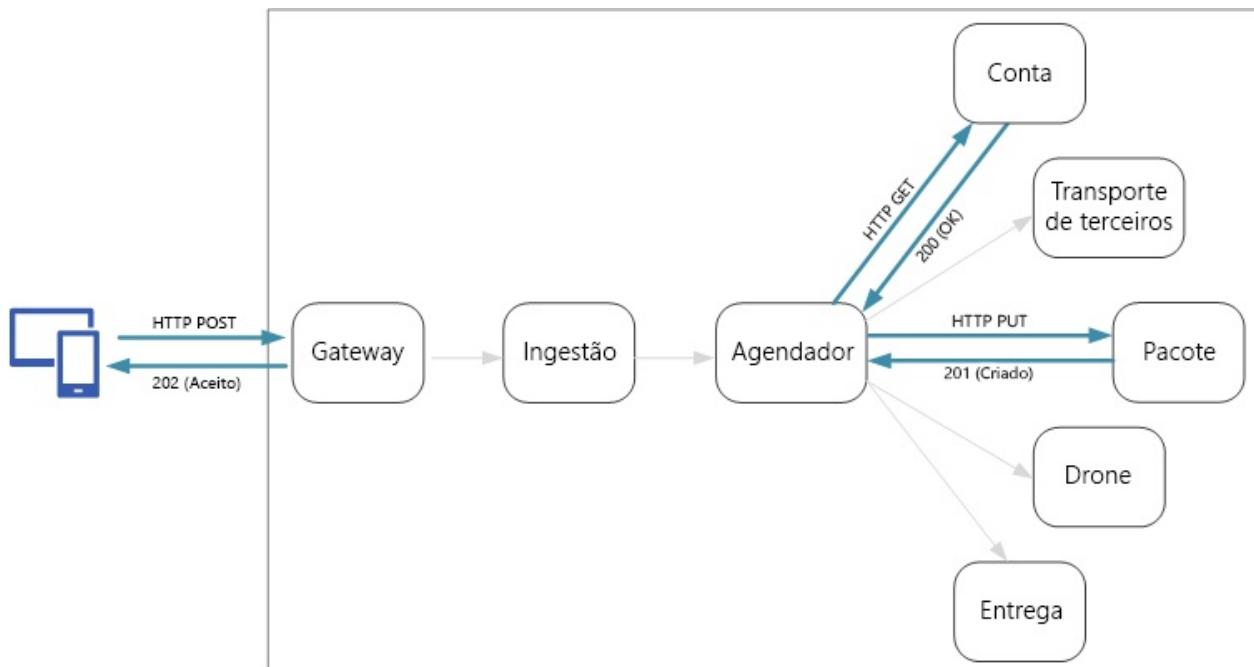
- O balanceamento de carga no nível da sessão, com base nas latências observadas ou no número de solicitações pendentes. Isso pode melhorar o desempenho em relação ao balanceamento de carga da camada 4 fornecido pelo Kubernetes.
- Roteamento da camada 7 com base no caminho da URL, no cabeçalho de Host, na versão da API ou em outras regras no nível de aplicativo.
- Repetição de solicitações com falha. Uma malha de serviço reconhece os códigos de erro de HTTP e pode repetir automaticamente as solicitações com falha. Você pode configurar o número máximo de repetições, juntamente com um período de tempo limite para delimitar a latência máxima.
- Interrupção de circuito. Se uma instância falhar de modo consistente nas solicitações, a malha de serviço a marcará temporariamente como indisponível. Após um período de retirada, ele tentará a instância novamente. Você pode configurar o disjuntor com base em vários critérios, como o número de falhas consecutivas.
- Amalha de serviço captura métricas sobre chamadas entre serviços, como o volume de solicitação, a latência, as taxas de êxito e de erro e os tamanhos das respostas. Amalha de serviço também habilita o rastreamento distribuído ao adicionar informações de correlação para cada salto em uma solicitação.
- Autenticação de TLS mútua para chamadas de serviço a serviço.

Você precisa de uma malha de serviço? O valor agregado a um sistema distribuído é, de fato, interessante. Se não tiver uma malha de serviço, você precisará considerar cada um dos desafios mencionados no início deste capítulo. Você pode resolver problemas de repetição, no disjuntor e de rastreamento distribuído sem uma malha de serviço, mas uma malha do serviço transfere essas problemas dos serviços individuais para uma camada dedicada. Por outro lado, a malha de serviço é uma tecnologia relativamente nova, que ainda está em desenvolvimento. A implantação de uma malha de serviço adiciona complexidade à instalação e à configuração do cluster. Pode haver implicações de desempenho porque as solicitações agora são roteadas por meio do proxy da malha de serviço e também porque os serviços extras agora estão sendo executados em cada nó no cluster. Você deve realizar testes de carga e de desempenho minuciosos antes de implementar uma malha de serviço na produção.

Design de API

Um bom design de API é importante em uma arquitetura de microserviços porque toda a troca de dados entre os serviços ocorre por meio de mensagens ou de chamadas à API. As APIs devem ser eficientes para evitar a criação de E/S com ruídos¹. Uma vez que os serviços são projetados por equipes que trabalham de forma independente, as APIs devem ter esquemas semânticos e de controle de versão bem definidos para que as atualizações não interrompam outros serviços.

¹. O efeito cumulativo de um grande número de solicitações de E/S pode ter um impacto significativo no desempenho e capacidade de resposta. ↩



É importante distinguir entre os dois tipos de API:

- As APIs públicas que os aplicativos do cliente chamam.
- As APIs de back-end que são usadas para comunicação entre serviços.

Esses dois casos de uso têm requisitos um pouco diferentes. Uma API pública deve ser compatível com aplicativos cliente, normalmente os aplicativos de navegador ou os aplicativos móveis nativos. Na maioria das vezes, isso significa que a API pública usará o REST sobre o HTTP. No entanto, para as APIs de back-end, você precisa levar em conta o desempenho da rede. Dependendo da granularidade dos seus serviços, a comunicação entre serviços pode resultar em uma grande quantidade de tráfego de rede. Os serviços podem rapidamente se tornar um limite de E/S. Por esse motivo, considerações como a velocidade de serialização e o tamanho de payload se tornam mais importantes. Algumas alternativas populares para usar o REST sobre o HTTP incluem gRPC, Apache Avro e Apache Thrift. Esses protocolos são compatíveis com a serialização binária e geralmente são mais eficientes do que o HTTP.

Considerações

Aqui estão algumas questões a serem consideradas ao escolher como implementar uma API.

REST versus RPC. Considere as compensações entre o uso de uma interface no estilo REST em vez de uma interface no estilo RPC.

- O REST modela recursos, o que pode ser uma maneira natural de expressar seu modelo de domínio. Ele define uma interface uniforme com base nos verbos HTTP, o que incentiva a evolução. Ele tem uma semântica bem definida em termos de idempotência, efeitos colaterais e códigos de resposta. E impõe uma comunicação

sem monitoração de estado, o que melhora a escalabilidade.

- O RPC está mais voltado para as operações ou para os comandos. Uma vez que as interfaces de RPC se parecem com chamadas de método locais, talvez você acabe criando APIs extremamente ruidosas. No entanto, isso não significa que o RPC deve ser ruidoso. Isso apenas indica que você precisa ter cuidado ao criar a interface.

Para obter uma interface RESTful, a opção mais comum é REST sobre HTTP usando JSON. Para obter uma interface no estilo RPC, há várias estruturas populares, incluindo gRPC, Apache Avro e Apache Thrift.

Eficiência. Considere a eficiência em termos de velocidade, memória e tamanho de payload. Normalmente, uma interface baseada em gRPC é mais rápida que REST sobre HTTP.

IDL (linguagem IDL). Uma IDL é usada para definir os métodos, os parâmetros e os valores retornados de uma API. Um IDL pode ser usado para gerar o código do cliente, o código de serialização e a documentação da API. Os IDLs também podem ser consumidos por ferramentas de teste da API, como o Postman. Estruturas como gRPC, Avro e Thrift definem suas próprias especificações de IDL. O REST sobre HTTP não tem um formato padrão de IDL, mas uma opção comum é o OpenAPI (anteriormente conhecido como Swagger). Você também pode criar uma API de REST HTTP sem usar uma linguagem de definição formal, mas perderá os benefícios da geração de código e de teste.

Serialização. Como os objetos são serializados eletronicamente? As opções incluem formatos baseados em texto (principalmente JSON) e formatos binários, como um buffer de protocolo. Os formatos binários geralmente são mais rápidos do que os formatos baseados em texto. No entanto, o JSON tem vantagens em termos de interoperabilidade porque a maioria das linguagens e das estruturas são compatíveis com a serialização JSON. Alguns formatos de serialização exigem um esquema fixo e alguns exigem a compilação de um arquivo de definição de esquema. Nesse caso, você precisará incorporar essa etapa no processo de build.

Suporte de frameworks e linguagem. O HTTP é compatível com praticamente todas os frameworks e linguagens. gRPC, Avro e Thrift têm bibliotecas para C++, C#, Java e Python. Thrift e gRPC também são compatíveis com Go.

Compatibilidade e interoperabilidade. Se escolher um protocolo como gRPC, talvez você precisará de uma camada de conversão de protocolo entre a API pública e o back-end. Um gateway pode executar essa função. Se você estiver usando uma malha de serviço, considere quais protocolos são compatíveis com ela. Por exemplo, linkerd tem suporte interno para HTTP, Thrift e gRPC.

Uma recomendação de linha de base é escolher REST sobre HTTP, a menos que você precise dos benefícios de desempenho de um protocolo binário. O REST sobre HTTP não requer nenhuma biblioteca especial. Ele cria um acoplamento mínimo, uma vez que os autores da chamada não precisam de um stub de cliente para se comunicarem com o serviço. Há ecossistemas avançados de ferramentas para dar suporte às definições de esquema, teste e monitoramento de pontos de extremidade de HTTP RESTful. Por fim, o HTTP é compatível com clientes de navegador, portanto, você não precisa de uma camada de conversão de protocolo entre o cliente e o back-end.

No entanto, se escolher o REST sobre HTTP, você deverá fazer testes de carga e de desempenho no início do processo de desenvolvimento para validar se ele funciona bem o suficiente para o seu cenário.

Projeto de API RESTful

Aqui estão algumas considerações específicas para ter em mente:

- Fique atento às APIs que vazam detalhes de implementação internos ou refletem um esquema de banco de dados interno. A API deve modelar o domínio. É um contrato entre serviços e, de modo ideal, só deve ser alterado quando novas funcionalidades forem adicionadas, não apenas porque você refatorou algum código ou normalizou uma tabela de banco de dados.
- Diferentes tipos de cliente, como o aplicativo móvel e o navegador da Web da área de trabalho, podem exigir tamanhos diferentes de payload ou padrões de interação. Considere o uso do padrão back-ends para front-ends² para criar back-ends separados para cada cliente, que expõem uma interface ideal para esse cliente.

². Crie serviços de back-end separados a serem consumidos por aplicativos de front-end específico ou interfaces. Esse padrão é útil quando você deseja evitar a personalização de um único back-end para várias interfaces. Esse padrão foi descrito pela primeira vez por Sam Newman. ↪

- Para as operações com efeitos colaterais, considere torná-los idempotentes e implementá-los como métodos PUT. Isso habilitará tentativas seguras e poderá melhorar a resiliência.
- Os métodos HTTP podem ter uma semântica assíncrona, em que o método retorna uma resposta imediatamente, mas o serviço realiza a operação assíncrona. Nesse caso, o método deve retornar um código de resposta HTTP 202, que indica que a solicitação foi aceita para processamento, mas o processamento ainda não foi concluído.

Mapeamento de REST para padrões de DDD

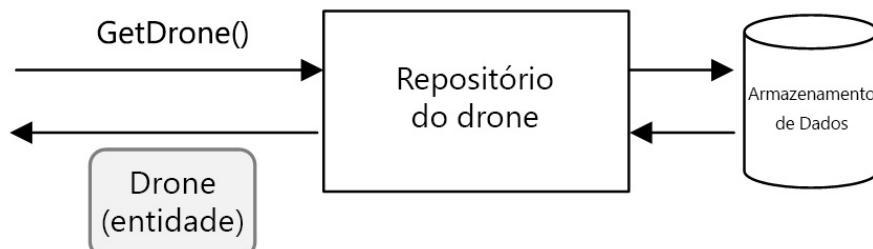
Os padrões como entidade, agregação e objeto de valor são projetados para colocar determinadas restrições nos objetos em seu modelo de domínio. Em muitas discussões de DDD, os padrões são modelados usando conceitos da linguagem orientada a objeto (OO), como construtores ou getters e setters de propriedade. Por exemplo, os objetos de valor devem ser imutáveis. Em uma linguagem de programação OO, você aplicaria isso ao atribuir os valores no construtor e tornando as propriedades somente leitura:

```
export class Location {
    readonly latitude: number;
    readonly longitude: number;

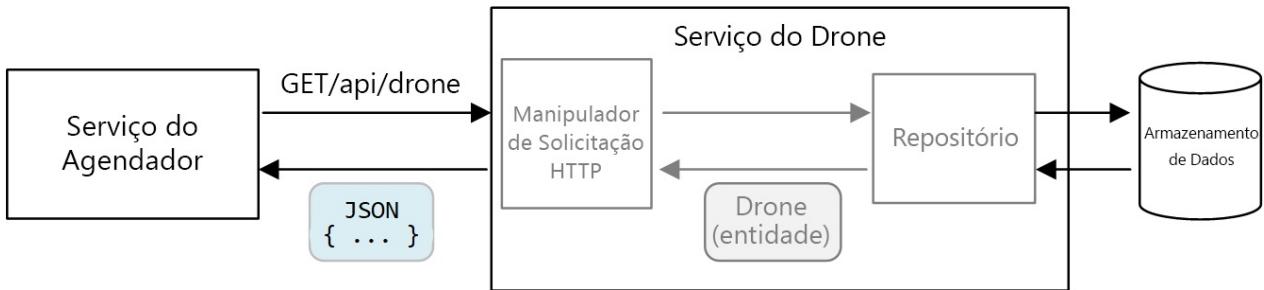
    constructor(latitude: number, longitude: number) {
        if (latitude < -90 || latitude > 90) {
            throw new RangeError('latitude must be between -90 and 90');
        }
        if (longitude < -180 || longitude > 180) {
            throw new RangeError('longitude must be between -180 and 180');
        }
        this.latitude = latitude;
        this.longitude = longitude;
    }
}
```

Esses tipos de práticas recomendadas de codificação são particularmente importantes ao criar um aplicativo monolítico tradicional. Com uma base de código grande, muitos subsistemas podem usar o objeto Location, portanto, é importante que o objeto imponha o comportamento correto.

Outro exemplo é o padrão Repositório, que garante que outras partes do aplicativo não façam leituras ou gravações diretas no armazenamento de dados:



Em uma arquitetura de microserviços, no entanto, os serviços não compartilham a mesma base de código, nem os repositórios de dados. Em vez disso, eles se comunicam por meio de APIs. Considere o caso em que o serviço Agendador solicita informações sobre um drone do serviço Drone. O serviço Drone tem seu modelo interno de drone expresso através de código. Mas o Agendador não o vê. Em vez disso, ele recupera uma representação da entidade — do drone, talvez um objeto JSON em uma resposta de HTTP.



O serviço Agendador não pode modificar os modelos internos de serviço do Drone ou gravar no armazenamento de dados de serviço do Drone. Isso indica que o código que implementa o serviço Drone tem uma área de superfície exposta menor em comparação com o código em um monolito tradicional.

Você pode modelar muitos dos padrões de DDD por meio das APIs REST.

Por exemplo:

- As agregações naturalmente mapeiam para os recursos em REST. Por exemplo, a agregação Entrega deve ser exposta como um recurso pela API Entrega.
- As agregações são os limites de consistência. As operações em agregações nunca devem deixar uma agregação em um estado inconsistente. Portanto, você deve evitar criar APIs que permitem que um cliente manipule o estado interno de uma agregação. Em vez disso, favoreça as APIs de alta granularidade que expõem agregações como recursos.
- As entidades têm identidades exclusivas. No REST, os recursos têm identificadores exclusivos na forma de URLs. Crie URLs de recursos que correspondam à identidade de domínio da entidade. O mapeamento da URL para a identidade de domínio pode parecer vago para o cliente.
- As entidades filho de uma agregação podem ser alcançadas por meio da navegação da entidade raiz. Se você seguir os princípios HATEOAS, as entidades filho poderão ser acessadas por meio de links na representação da entidade pai.
- Uma vez que os objetos de valor são imutáveis, as atualizações são executadas, substituindo o objeto de valor inteiro. No REST, implante as atualizações por meio das solicitações PUT ou PATCH.
- Um repositório permite aos clientes consultar, adicionar ou remover objetos em uma coleção, abstraindo os detalhes do armazenamento de dados subjacente. No REST, uma coleção pode ser um recurso distinto, com métodos para consultar a coleção ou adicionar novas entidades na coleção.

Quando você projeta suas APIs, pense em como elas expressam o modelo de domínio, não apenas nos dados dentro do modelo, mas também nas operações de negócios e nas restrições nos dados.

Conceito de DDD	Equivalente a REST	Exemplo
Agregação	Recurso	<code>{ "1":1234, "status":"pending" ... }</code>
Identidade	URL	<code>https://delivery-service/deliveries/1</code>
Entidades filho	Links	<code>{ "href": "/deliveries/1/confirmation" }</code>
Atualizar objetos de valor	PUT ou PATCH	<code>PUT https://delivery-service/deliveries/1/dropoff</code>
Repositório	Coleção	<code>https://delivery-service/deliveries?status=pending</code>

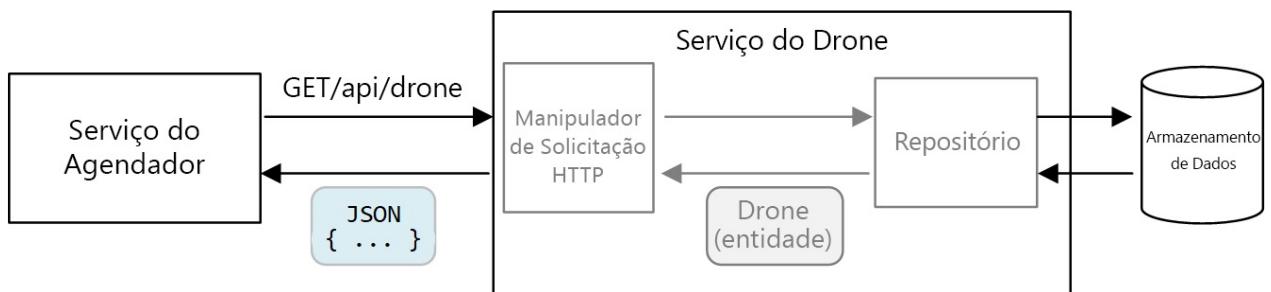
Controle de versão de API

Uma API é um contrato entre um serviço e os clientes ou os consumidores do serviço. Se uma API for alterada, haverá o risco de interromper clientes que dependem da API, sejam eles clientes externos ou de outros microserviços. Portanto, é uma boa ideia minimizar a quantidade de alterações de API que você faz. Geralmente, as alterações na

implementação subjacente não exigem nenhum alteração na API. No entanto, de modo realista, em algum momento você desejará adicionar novos recursos ou novas capacidades que exigem a alteração de uma API existente.

Sempre que possível, torne as alterações na API compatíveis com as versões anteriores. Por exemplo, evite remover um campo de um modelo, uma vez que isso pode interromper os clientes que esperam que o campo exista. Adição de um campo não interrompe a compatibilidade, visto que os clientes devem ignorar todos os campos que não compreendem em uma resposta. No entanto, o serviço deve tratar o caso em que um cliente mais antigo omite o novo campo em uma solicitação.

Supporte para o controle de versão em seu contrato de API. Se você fizer uma alteração de API de interrupção, apresente uma nova versão de API. Continue a oferecer suporte à versão anterior e permita que os clientes selezionem qual versão será chamada. Há algumas maneiras de fazer isso. Uma delas é expor as duas versões no mesmo serviço. Outra opção é executar as duas versões do serviço lado a lado e rotear as solicitações para uma versão ou para a outra com base nas regras de roteamento de HTTP.

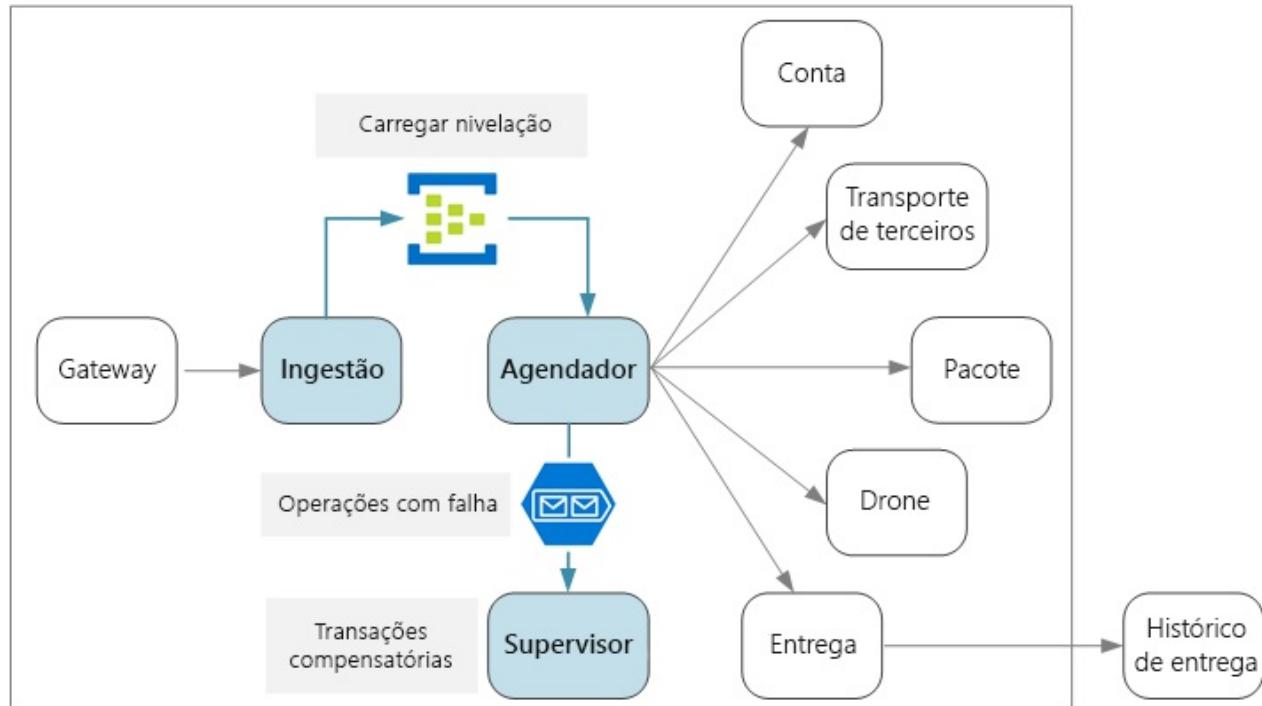


Há um custo para dar suporte a várias versões em termos de tempo de desenvolvedor, de teste e de sobrecarga operacional. Portanto, é conveniente substituir versões antigas o mais rápido possível. Para as APIs internas, a equipe que tem a API pode trabalhar com outras equipes para ajudá-las a migrar para a nova versão. Isso deve ser feito quando for útil ter um processo de controle de várias equipes. Para as APIs externas (públicas), poderá ser mais difícil substituir uma versão de API, especialmente se a API for consumida por terceiros ou por aplicativos cliente nativos.

Quando uma implementação de serviço for alterada, será útil marcar a alteração com uma versão. A versão fornece informações importantes ao solucionar problemas de erros. Ela pode ser muito útil para a análise da causa raiz saber exatamente qual versão do serviço foi chamada. Considere o uso do controle de versão semântico para versões de serviço. O controle de versão semântico usa um formato MAJOR.MINOR.PATCH. No entanto, os clientes deverão selecionar somente uma API com o número de versão principal ou, possivelmente, a versão secundária se houver alterações significativas (mas não interruptivas) entre as versões secundárias. Em outras palavras, é razoável para os clientes selecionar entre a versão 1 e a versão 2 de uma API, mas não é coerente selecionar a versão 2.1.3. Se permitir esse nível de granularidade, você correrá o risco de ter que dar suporte à proliferação de versões.

Ingestão de dados e fluxo de trabalho

Os microsserviços normalmente têm um fluxo de trabalho que abrange vários serviços para uma única transação. O fluxo de trabalho deve ser confiável; ele não pode perder transações nem deixá-las em um estado parcialmente concluído. Além disso, o controle da taxa de ingestão de solicitações de entrada é algo crítico. Com muitos serviços pequenos comunicando-se entre si, um grande volume de solicitações de entrada pode sobrecarregar a comunicação entre os serviços.



O fluxo de trabalho da entrega por drone

No aplicativo de entrega por drone, as seguintes operações devem ser executadas para o agendamento de uma entrega:

1. Verificar o status da conta do cliente (serviço de Conta).
2. Criar uma nova entidade de pacote (serviço de Pacote).
3. Verificar se é necessário algum serviço de transporte terceirizado para a entrega, de acordo com os locais da retirada e da entrega (serviço de Transporte terceirizado).
4. Agendar um drone para a retirada (serviço de Drone).
5. Criar uma nova entidade de entrega (serviço de Entrega).

Essa é a essência do aplicativo inteiro, portanto o processo todo deve ter alto desempenho e ser confiável. Alguns desafios específicos devem ser tratados:

- **Nivelamento de carga.** O excesso de solicitações de clientes pode sobrecarregar o sistema com o tráfego de rede entre os serviços. Isso também pode sobrecarregar as dependências de back-end, como os serviços de armazenamento ou remotos. Isso pode ocasionar a limitação dos serviços realizando as chamadas, o que cria pressão de retorno no sistema. Portanto, é importante nivelar a carga das solicitações que chegam ao sistema, armazenando-as em buffer ou colocando-as em fila para processamento.
- **Entrega garantida.** Para evitar a remoção de solicitações de clientes, o componente de ingestão deverá assegurar a entrega de mensagens pelo menos uma vez.

- **Manipulação de erros.** Se algum dos serviços retornar um código de erro ou apresentar uma falha não transitória, a entrega não poderá ser agendada. Um código de erro pode indicar uma condição de erro esperada (por exemplo, a conta do cliente está suspensa) ou um erro de servidor inesperado (HTTP 5xx). Há também a possibilidade de um serviço não estar disponível, fazendo com que a chamada de rede atinja o tempo limite.

Em primeiro lugar, examinaremos o lado da ingestão da equação: como o sistema pode ingerir solicitações de entrada do usuário com alta produtividade. Em seguida, consideraremos como o aplicativo de entrega por drone pode implementar um fluxo de trabalho confiável. Na verdade, o design do subsistema de ingestão afeta o back-end do fluxo de trabalho.

Ingestão

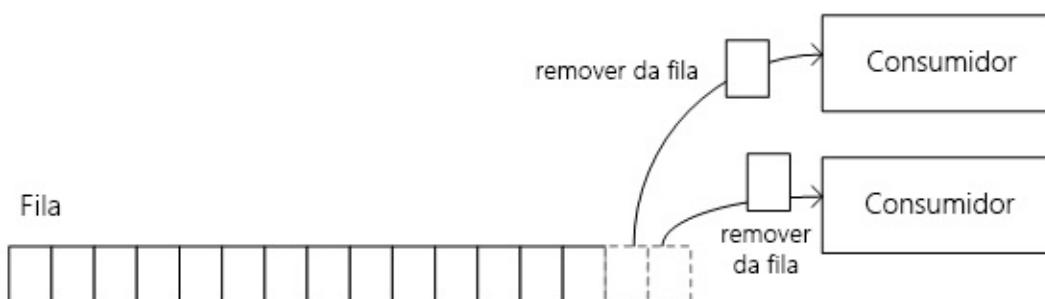
Com base nos requisitos comerciais, a equipe de desenvolvimento identificou os seguintes requisitos não funcionais para a ingestão:

- Produtividade constante de 10 mil solicitações/s.
- Capacidade de lidar com picos de até 50 mil/s sem remover solicitações de cliente nem atingir o tempo limite.
- Latência inferior a 500 ms em 99% do tempo.

O requisito para lidar com picos ocasionais de tráfego apresenta um desafio de design. Em teoria, o sistema pode ser escalado horizontalmente para manipular o tráfego máximo esperado. No entanto, provisionar uma quantidade tão grande de recursos como essa seria muito ineficiente. Na maioria das vezes, o aplicativo não precisará dessa capacidade, portanto haverá núcleos ociosos custando dinheiro sem adicionar valor.

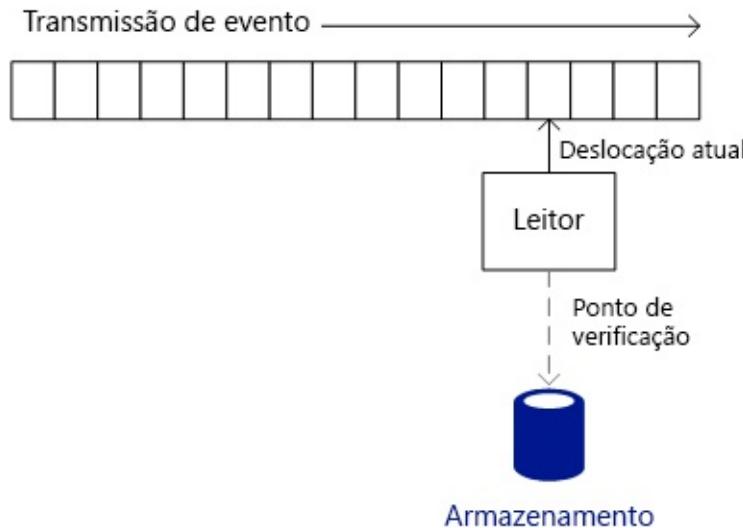
Uma abordagem melhor é colocar as solicitações de entrada em um buffer e deixá-lo atuar como um nivelador de carga. Com esse design, o serviço de Ingestão deve lidar com a taxa máxima de ingestão em curtos períodos, mas os serviços de back-end precisam apenas lidar com a carga constante máxima. Ao armazenar em buffer no front-end, os serviços de back-end não precisarão lidar com grandes picos no tráfego. Os Hubs de Eventos oferecem baixa latência e alta produtividade, além de ser uma solução econômica em altos volumes de ingestão.

É importante entender como os Hubs de Eventos podem alcançar uma produtividade tão alta, porque isso afeta como um cliente deve consumir as mensagens dos Hubs de Eventos. Os Hubs de Eventos não implementam uma fila. Em vez disso, eles implementam um fluxo de eventos. Com uma fila, um consumidor individual poderá remover uma mensagem da fila e o próximo consumidor não verá essa mensagem. As filas, portanto, permitem que você use um Padrão de consumidores concorrentes para processar mensagens em paralelo e melhorar a escalabilidade. Para maior resiliência, o consumidor mantém um bloqueio na mensagem e libera-o quando termina de processá-la. Se o consumidor falhar, por exemplo, por causa de uma falha no nó em que ele é executado, o bloqueio atingirá o tempo limite e a mensagem voltará à fila.



Os Hubs de Eventos, por outro lado, usam a semântica de streaming. Os consumidores leem o fluxo de forma independente em seu próprio ritmo. Cada consumidor é responsável por manter o controle da sua posição atual no fluxo. Um consumidor deve gravar sua posição atual no armazenamento persistente em um intervalo predefinido. Dessa forma, se o consumidor apresentar uma falha (por exemplo, falhas do consumidor ou do host), uma nova instância poderá retomar a leitura do fluxo da última posição gravada. Esse processo é chamado *ponto de verificação*.

Por motivos de desempenho, um consumidor normalmente não realiza o ponto de verificação depois de cada mensagem. Em vez disso, realiza o ponto de verificação em um intervalo fixo, por exemplo, depois de processar n mensagens ou a cada n segundos. Como consequência, se um consumidor falhar, alguns eventos poderão ser processados duas vezes, porque uma nova instância sempre continuará do último ponto de verificação. Há dois lados da mesma moeda: os pontos de verificação frequentes podem prejudicar o desempenho, mas os pontos de verificação esparsos significam que você reproduzirá mais eventos após uma falha.



Os Hubs de Eventos não são projetados para consumidores concorrentes. Embora vários consumidores possam ler um fluxo, cada um deles percorre-o de forma independente. Em vez disso, os Hubs de Eventos usam um padrão de consumidor particionado. Um hub de eventos tem até 32 partições. A escala horizontal é obtida atribuindo um consumidor separado a cada partição.

O que isso significa para o fluxo de trabalho da entrega por drone? Para obter todos os benefícios dos Hubs de Eventos, o Agendador de Entregas não pode esperar o processamento de cada mensagem antes de passar para a próxima. Se fizer isso, passará a maior parte do tempo aguardando a conclusão das chamadas de rede. Em vez disso, ele precisa processar lotes de mensagens em paralelo, usando chamadas assíncronas aos serviços de back-end.

Tratamento de falhas

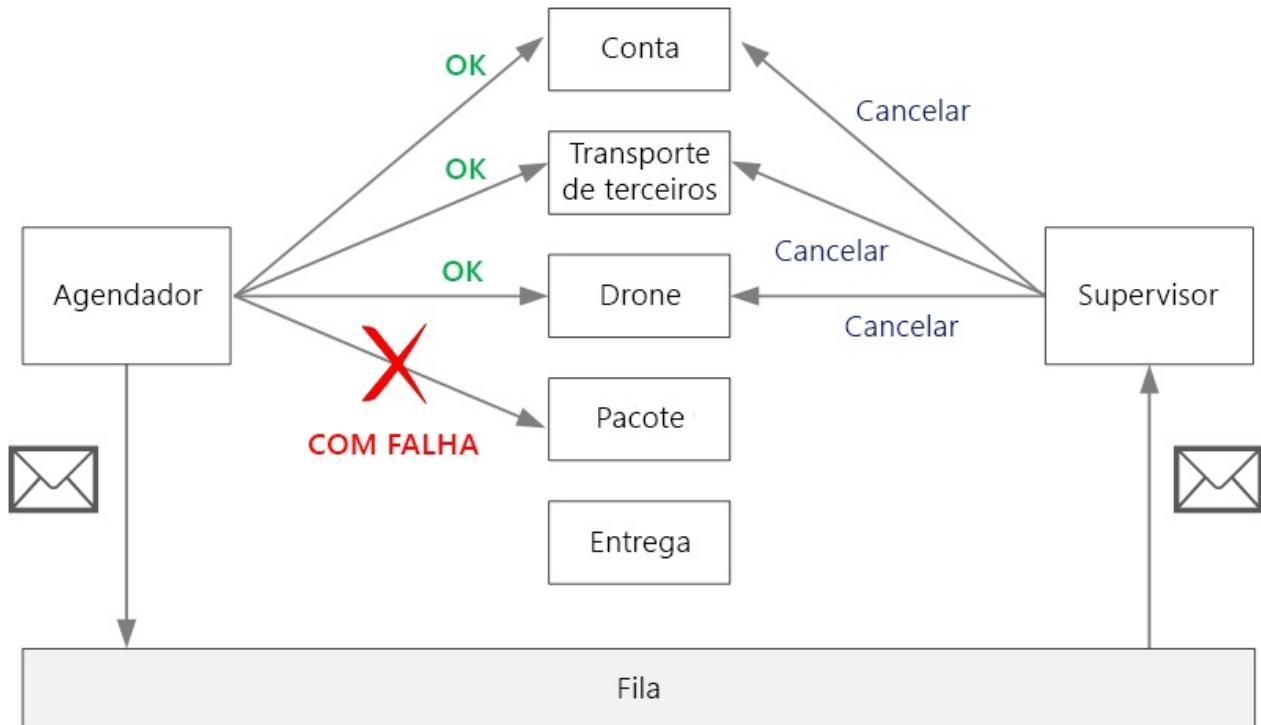
Há três classes gerais de falha a serem consideradas:

1. Um serviço downstream pode ter uma falha não transitória, que é qualquer falha que provavelmente não desapareça por si só. As falhas não transitórias incluem condições de erro normais, como uma entrada inválida para um método. Elas também incluem exceções sem tratamento no código do aplicativo ou uma falha de processo. Se esse tipo de erro ocorrer, a transação comercial inteira deverá ser marcada como uma falha. Será necessário desfazer as outras etapas na mesma transação que já foram concluídas com êxito.
2. Um serviço de downstream pode apresentar uma falha temporária, como tempo limite de rede atingido. Esses erros normalmente podem ser resolvidos simplesmente ao tentar a chamada novamente. Se a operação ainda falhar após um determinado número de tentativas, ela será considerada uma falha não transitória.
3. O serviço de Agendador em si pode falhar (por exemplo, por causa de uma falha no nó). Nesse caso, o Kubernetes abrirá uma nova instância do serviço. No entanto, as transações ainda em andamento deverão ser retomadas.

Transações de compensação

Se ocorrer uma falha não transitória, a transação atual poderá estar em um estado de falha parcial, em que uma ou mais etapas já foram concluídas com êxito. Por exemplo, se o serviço de drone já tiver agendado um drone, ele deverá ser cancelado. Nesse caso, o aplicativo deverá desfazer as etapas concluídas com êxito usando uma transação de compensação. Em alguns casos, isso deve ser feito por um sistema externo ou até mesmo por um processo manual.

Se a lógica das transações de compensação for complexa, considere a criação de um serviço separado responsável por esse processo. No aplicativo de entrega por drone, o serviço de Agendador coloca as operações com falha em uma fila dedicada. Um microserviço separado, chamado Supervisor, lê dessa fila e chama uma API de cancelamento nos serviços que precisam ser compensados. Essa é uma variação do Padrão de Supervisor de Agente do Agendador. O serviço de Supervisor também pode executar outras ações, como notificar o usuário por e-mail ou por SMS ou enviar um alerta para um painel de operações.



Operações idempotentes e não idempotentes

Para evitar a perda das solicitações, o serviço de Agendador deve assegurar que todas as mensagens sejam processadas pelo menos uma vez. Os Hubs de Eventos poderão assegurar a entrega pelo menos uma vez se o cliente realizar o ponto de verificação corretamente.

Se o serviço de Agendador falhar, poderá ser no meio do processamento de uma ou mais solicitações de cliente. Essas mensagens serão retiradas por outra instância do Agendador e reprocessadas. O que acontece se uma solicitação é processada duas vezes? É importante evitar a duplicação de qualquer trabalho. Afinal, não desejamos que o sistema envie dois drones para o mesmo pacote.

Uma abordagem é criar todas as operações para que sejam idempotentes. Uma operação será idempotente se puder ser chamada várias vezes sem produzir efeitos colaterais adicionais após a primeira chamada. Em outras palavras, um cliente poderá invocar a operação uma, duas ou muitas vezes e o resultado será o mesmo. O serviço basicamente deve ignorar chamadas duplicadas. Para um método com efeitos colaterais ser idempotente, o serviço deverá ter a capacidade de detectar chamadas duplicadas. Por exemplo, você pode solicitar que o responsável pela chamada atribua a ID, em vez de o serviço gerar uma nova ID. Com isso, o serviço pode verificar se há IDs duplicadas.

Observação: A especificação do HTTP declara que os métodos GET, PUT e DELETE devem ser idempotentes.

Não há garantia que os métodos POST sejam idempotentes. Se um método POST criar um novo recurso, normalmente não haverá nenhuma garantia de que a operação será idempotente.

Nem sempre é simples gravar um método idempotente. Outra opção é o Agendador acompanhar o andamento de todas as transações de um repositório durável. Sempre que uma mensagem fosse processada, ele consultaria o estado no repositório durável. Depois de cada etapa, ele gravaria o resultado no repositório. Pode haver implicações de desempenho nessa abordagem.

Exemplo: operações idempotentes

A especificação do HTTP declara que os métodos PUT devem ser idempotentes. A especificação define idempotente da seguinte forma:

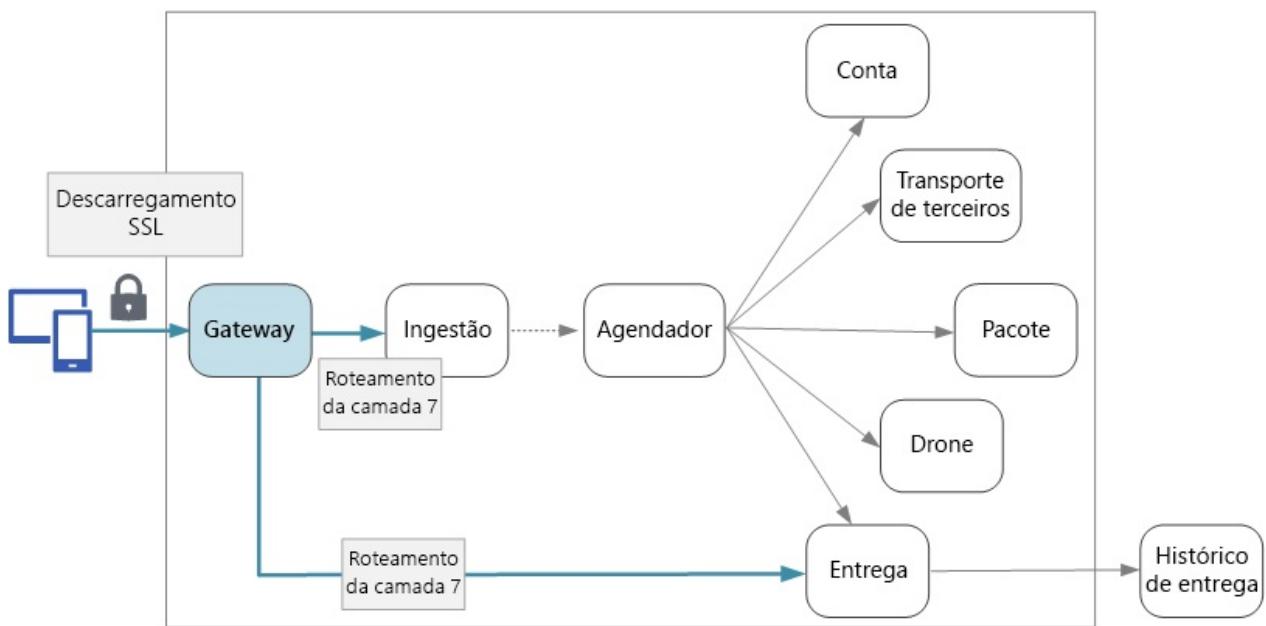
um método de solicitação será considerado "idempotente" se o efeito desejado no servidor de várias solicitações idênticas com esse método for o mesmo efeito de uma única solicitação como essa. (RFC 7231)

É importante entender a diferença entre a semântica de PUT e de POST ao criar uma nova entidade. Em ambos os casos, o cliente envia uma representação de uma entidade no corpo da solicitação. No entanto, o significado do URI é diferente.

- Para um método POST, o URI representa um recurso pai da nova entidade, como uma coleção. Por exemplo, para criar uma nova entrega, o URI pode ser `/api/deliveries`. O servidor cria a entidade e atribui a ela um novo URI, como `/api/deliveries/39660`. Esse URI é retornado no cabeçalho de Localização da resposta. Cada vez que o cliente enviar uma solicitação, o servidor criará uma nova entidade com um novo URI.
- Para um método PUT, o URI identifica a entidade. Se já existir uma entidade com esse URI, o servidor substituirá a entidade existente pela versão na solicitação. Se nenhuma entidade existir com esse URI, o servidor criará uma. Por exemplo, suponha que o cliente envie uma solicitação PUT para `api/deliveries/39660`. Pressupondo que não exista nenhuma entrega com esse URI, o servidor criará uma nova. Agora se o cliente enviar a mesma solicitação novamente, o servidor substituirá a entidade existente.

Gateways de API

Em uma arquitetura de microserviços, um cliente pode interagir com mais de um serviço front-end. Em razão disso, como um cliente sabe quais end-points a serem chamados? O que acontece quando são introduzidos novos serviços ou serviços existentes são refatorados? Como os serviços processam terminação SSL, autenticação e outras questões? Um gateway de API pode ajudá-lo a enfrentar esses desafios.



O que é um gateway de API?

Um gateway de API fica entre clientes e serviços. Ele atua como um proxy reverso, encaminhando as solicitações de clientes para serviços. Ele também pode executar várias tarefas detalhadas, como autenticação, terminação de SSL e a limitação de taxa. Se você não implantar um gateway, os clientes deverão enviar solicitações diretamente aos serviços front-end. No entanto, há alguns possíveis problemas com a exposição de serviços diretamente aos clientes:

- Isso pode resultar em código de cliente complexo. O cliente deve manter o controle dos vários end-points e lidar com falhas de forma resiliente.
- Ele cria um acoplamento entre o cliente e o back-end. O cliente precisa saber como os serviços individuais são decompostos. Isso torna mais difícil manter o cliente e também mais difícil refatorar serviços.
- Uma única operação pode exigir chamadas para vários serviços. Isso pode resultar em várias viagem de ida e volta na rede entre o cliente e o servidor, adicionando latência significativa.
- Cada serviço voltado ao público deve lidar com preocupações como autenticação, SSL e limite de taxa do cliente.
- Serviços devem expor um protocolo de cliente amigável, como HTTP ou WebSocket. Isso limita a escolha de protocolos de comunicação.
- Serviços com end-points públicos são uma superfície de ataque potencial e devem ser protegidos.

Um gateway ajuda a resolver esses problemas, separando clientes de serviços. Gateways podem executar várias funções diferentes, e você pode não precisar de todas elas. As funções podem ser agrupadas nos seguintes padrões de design:

- **Roteamento de Gateway.** Use o gateway como um proxy reverso para encaminhar solicitações para um ou mais serviços de back-end, usando o roteamento de 7 camadas. O gateway fornece um end-point único para os clientes e ajuda a separar clientes de serviços.

- **Gateway de Agregação.** Use o gateway para agregar várias solicitações individuais em uma única solicitação. Esse padrão se aplica quando uma única operação exige chamadas para vários serviços de back-end. O cliente envia uma solicitação para o gateway. O gateway envia solicitações para os vários serviços de back-end, agrupa os resultados e os envia de volta ao cliente. Isso ajuda a reduzir conversas entre o cliente e o back-end.

Gateway de Descarregamento. Use o gateway para descarregar funcionalidade de serviços individuais no gateway, particularmente questões transversais. Pode ser útil consolidar essas funções em um único local, em vez de responsabilizar cada serviço por implementá-las. Isso é especialmente verdadeiro para recursos que exigem habilidades especializadas para implementação correta, como autenticação e autorização.

Aqui estão alguns exemplos de funcionalidades que podem ser transferidas para um gateway:

- Terminação SSL
- Autenticação
- Lista de permissões de IP
- Limitação de taxa do cliente
- Log e monitoramento
- Cache de resposta
- Firewall do aplicativo Web
- Compactação GZIP
- Servir conteúdo estático

Escolhendo uma tecnologia de gateway

Existem algumas opções para a implementação de um gateway de API em seu aplicativo.

- **Servidor proxy reverso.** Nginx e HAProxy são servidores de proxy reverso populares compatíveis com recursos como平衡amento de carga, SSL e roteamento de 7 camadas. Os dois são produtos de software livre, com edições pagas que fornecem recursos adicionais e opções de suporte. Nginx e HAProxy são ambos produtos desenvolvidos com conjuntos de recursos avançados e alto desempenho. Você pode estendê-las com módulos de terceiros ou gravar scripts personalizados em Lua. O Nginx também é compatível com um módulo de script baseado em JavaScript, chamado NginScript.
- **Controlador de entrada de malha de serviços.** Se você estiver usando uma malha de serviços como linkerd ou Istio, considere os recursos fornecidos pelo controlador de entrada dessa malha de serviços. Por exemplo, o controlador de entrada Istio é compatível com roteamento de 7 camadas, redirecionamentos de HTTP, repetições e outros recursos.

Ao escolher uma tecnologia de gateway, considere o seguinte:

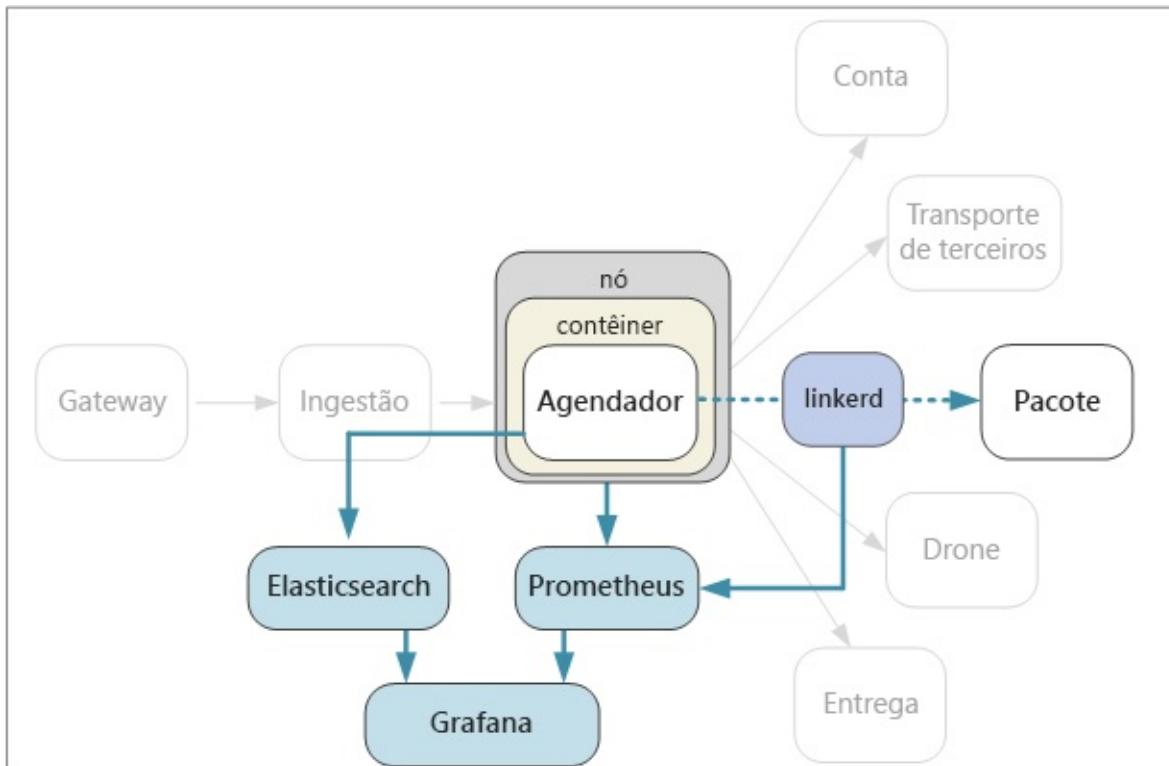
Recursos. As opções listadas acima são compatíveis com roteamento de 7 camadas, mas o suporte a outros recursos vai variar. Dependendo dos recursos que você precisa, poderá implantar mais de um gateway.

Implantação. Gateway de Aplicativo do Azure e Gerenciamento de API são serviços gerenciados. Nginx e HAProxy serão executados normalmente em contêineres dentro do cluster, mas também podem ser implantados em VMs dedicadas fora do cluster. Isso isola o gateway do restante da carga de trabalho, mas resulta em maior sobrecarga de gerenciamento.

Gerenciamento. Talvez seja necessário atualizar as regras de roteamento de gateway quando novos serviços são atualizados ou adicionados. Considere como esse processo será gerenciado. Considerações semelhantes aplicam-se ao gerenciamento de certificados SSL, listas de permissões IP e outros aspectos da configuração.

Log e monitoramento

Em qualquer aplicativo complexo, em algum momento, algo dará errado. Em um aplicativo de microsserviço, você precisa controlar o que está acontecendo em dúzias ou até mesmo centenas de serviços. O monitoramento e o registro em log são extremamente importantes para fornecer uma visão holística do sistema.



Em uma arquitetura de microsserviços, descobrir a causa exata de erros ou gargalos de desempenho pode ser algo especialmente desafiador. Uma única operação de usuário pode abranger vários serviços. Serviços podem atingir os limites de E/S de rede dentro do cluster. Uma cadeia de chamadas entre serviços pode causar pressão de retorno no sistema, resultando em latência alta ou em falhas em cascata. Além disso, geralmente você não sabe em qual nó um contêiner específico será executado. Contêineres colocados no mesmo nó podem estar competindo por CPU ou memória limitada.

Para entender o que está acontecendo, colete a telemetria do aplicativo. A telemetria pode ser dividida em logs e métricas.

Os **Logs** são registros de eventos baseados em texto que ocorrem durante a execução do aplicativo. Eles incluem itens como logs de aplicativos (instruções de rastreamento) ou logs do servidor Web. Os logs são úteis principalmente para análise da causa raiz e análise forense.

Métricas são valores numéricos que podem ser analisados. Você pode usá-los para observar o sistema em tempo real (ou quase em tempo real) ou então para analisar as tendências de desempenho ao longo do tempo. As métricas podem ser subcategorizadas ainda mais da seguinte maneira:

- Métricas **no nível do nó**, incluindo CPU, memória, rede, disco e uso do sistema de arquivos. As métricas do sistema ajudam a compreender a alocação de recurso para cada nó no cluster e exceções de solução de problemas.
- Métricas do **contêiner**. Se os serviços forem executados em contêineres, você precisará coletar métricas no nível do contêiner, não apenas no nível da VM.

- **Métricas de aplicativo.** Isso inclui as métricas que são relevantes para entender o comportamento de um serviço. Exemplos incluem o número de solicitações HTTP de entrada na fila, latência de solicitação, comprimento da fila de mensagens. Os aplicativos também podem criar métricas personalizadas específicas ao domínio, como o número de transações de negócios processadas por minuto.
- **Métricas de serviço dependentes.** Os serviços podem chamar serviços externos ou pontos de extremidade, como serviços de PaaS gerenciados ou serviços SaaS. Os serviços de terceiros podem ou não fornecer alguma métrica. Se não fornecerem, você dependerá de suas próprias métricas de aplicativo para acompanhar as estatísticas de latência e taxa de erros.

Considerações

Aqui estão algumas coisas específicas para se pensar no contexto de uma arquitetura de microserviços:

Configuração e gerenciamento. Você usará um serviço gerenciado para monitoramento e registro em log ou implantará os componentes de registro em log e monitoramento como contêineres dentro do cluster?

Taxa de ingestão. Qual é a taxa de transferência na qual o sistema pode incluir eventos de telemetria? O que acontece se essa taxa é excedida? Por exemplo, o sistema pode limitar os clientes, caso em que os dados de telemetria são perdidos, ou pode reduzir a amostra de dados. Às vezes, você pode atenuar esse problema reduzindo a quantidade de dados coletados:

- Agregue métricas calculando estatísticas como média e desvio padrão e envie esses dados estatísticos para o sistema de monitoramento.
- Reduza a amostra de dados — ou seja, processe apenas um percentual dos eventos.
- Envie os dados em lotes para reduzir o número de chamadas de rede para o serviço de monitoramento.

Custo. O custo de ingerir e armazenar dados de telemetria pode ser alto, especialmente em grandes volumes. Em alguns casos, ele pode até mesmo exceder o custo de executar o aplicativo. Nesse caso, convém reduzir o volume de telemetria agregando os dados, reduzindo sua resolução ou enviando-os em lotes, conforme descrito acima.

Fidelidade de dados. Qual o nível de precisão das métricas? Médias podem ocultar exceções, especialmente em escala. Além disso, se a taxa de amostragem é muito baixa, ela pode suavizar flutuações nos dados. Pode parecer que tem todas as solicitações têm aproximadamente a mesma latência de ponta a ponta, quando na verdade uma parte significativa das solicitações estão demorando muito mais.

Latência. Para habilitar alertas e monitoramento em tempo real, os dados de telemetria devem estar disponíveis rapidamente. Quanto a disponibilização dos dados que aparecem no painel de monitoramento realmente se aproxima de "tempo real"? Com alguns segundos de atraso? Mais de um minuto?

Armazenamento. Para logs, pode ser mais eficiente para gravar os eventos de log em armazenamento efêmero no cluster e configurar um agente para enviar os arquivos de log para armazenamento mais persistente. Dados devem ser movidos eventualmente para o armazenamento de longo prazo para que ele esteja disponível para análise retrospectiva. Uma arquitetura de microserviços pode gerar um grande volume de dados de telemetria, portanto, o custo de armazenar esses dados é uma consideração importante. Além disso, considere como você consultará os dados.

Painel e visualização. Você tem uma visão holística do sistema, em todos os serviços, tanto de dentro do cluster quanto dos serviços externos? Se você estiver gravando dados de telemetria e logs em mais de um local, o painel poderá mostrar todos eles e correlacioná-los? O painel de monitoramento deve mostrar pelo menos as seguintes informações:

- Alocação de recurso geral para crescimento e capacidade. Isso inclui o número de contêineres, as métricas do sistema de arquivos, rede e alocação de núcleos.
- Métricas de contêiner correlacionadas no nível de serviço.

- As métricas do sistema correlacionadas a contêineres.
- Erros de serviço e exceções.

Rastreamento distribuído

Conforme mencionado, um desafio em microserviços é entender o fluxo de eventos entre os serviços. Uma única operação ou transação pode envolver a chamadas para vários serviços. Para reconstruir a toda sequência de etapas, cada serviço deve propagar uma ID de correlação que atua como um identificador exclusivo para essa operação. AID de correlação habilita o rastreamento distribuído entre serviços.

O primeiro serviço que recebe uma solicitação de cliente deve gerar a ID de correlação. Se o serviço faz uma chamada HTTP para outro serviço, ele coloca a ID de correlação em um cabeçalho de solicitação. Da mesma forma, se o serviço envia uma mensagem assíncrona, ele coloca a ID de correlação nessa mensagem. Serviços de downstream continuam a propagar a ID de correlação, de modo que ela flui através de todo o sistema. Além disso, todo o código que grava eventos de log ou métricas do aplicativo deve incluir a ID de correlação.

Quando as chamadas de serviço são correlacionadas, você pode calcular métricas operacionais, como a latência de ponta a ponta para uma transação completa, o número de transações bem-sucedidas por segundo e o percentual de transações com falha. Incluir IDs de correlação nos logs de aplicativo possibilita realizar a análise da causa raiz. Se uma operação falhar, você poderá encontrar as instruções de log para todas as chamadas de serviço que fizerem parte da mesma operação.

Aqui estão algumas considerações ao implementar o rastreamento distribuído:

- Atualmente, não há nenhum cabeçalho HTTP padrão para IDs de correlação. Sua equipe deve padronizar em um valor de cabeçalho personalizado. Essa escolha pode ser decidida por sua estrutura de monitoramento/registro em log ou sua escolha de malha de serviço.
- Para mensagens assíncronas, se sua infraestrutura de mensagens é compatível com a adição de metadados a mensagens, você deve incluir a ID de correlação como metadados. Caso contrário, inclua-a como parte do esquema de mensagem.
- Em vez de um único identificador opaco, você pode enviar um contexto de correlação que inclui informações mais avançadas, tais como relações chamador/receptor.
- Considere o modo como você agregará os logs. Você talvez queira que o modo como as IDs de correlação são incluídas nos logs seja padronizado entre as equipes. Use um formato estruturado ou semiestruturado, tal como JSON, e defina um campo comum para conter a ID de correlação.

Opções de tecnologia

Para métricas de sistema e de contêiner, considere exportar métricas para um banco de dados de série temporal como Prometheus ou InfluxDB em execução no cluster.

- InfluxDB é um sistema baseado em push. É necessário que um agente envie as métricas por push. Você pode usar o Heapster, que é um serviço que coleta métricas de todo o cluster do kubelet, agrupa os dados e envia-os por push para InfluxDB ou outra solução de armazenamento de série temporal.
- O Prometheus é um sistema baseado em pull. Ele extrai periodicamente as métricas de locais configurados. O Prometheus pode extrair métricas geradas por cAdvisor ou métricas de estado kube. Métricas de estado kube é um serviço que coleta métricas do servidor de API do Kubernetes e as disponibiliza para o Prometheus (ou um extrator que seja compatível com um ponto de extremidade de cliente do Prometheus). Embora o Heapster agregue as métricas geradas pelo Kubernetes e encaminhe-as para um coletor, o serviço métricas de estado kube gera suas próprias métricas e disponibiliza-as por meio de um ponto de extremidade para extração. Para

métricas do sistema, use o Exportador de nó, que é um exportador Prometheus para métricas do sistema. O Prometheus é compatível com os dados de ponto flutuante, mas não com os dados de cadeia de caracteres, portanto, ele é apropriado para as métricas do sistema, mas não para logs.

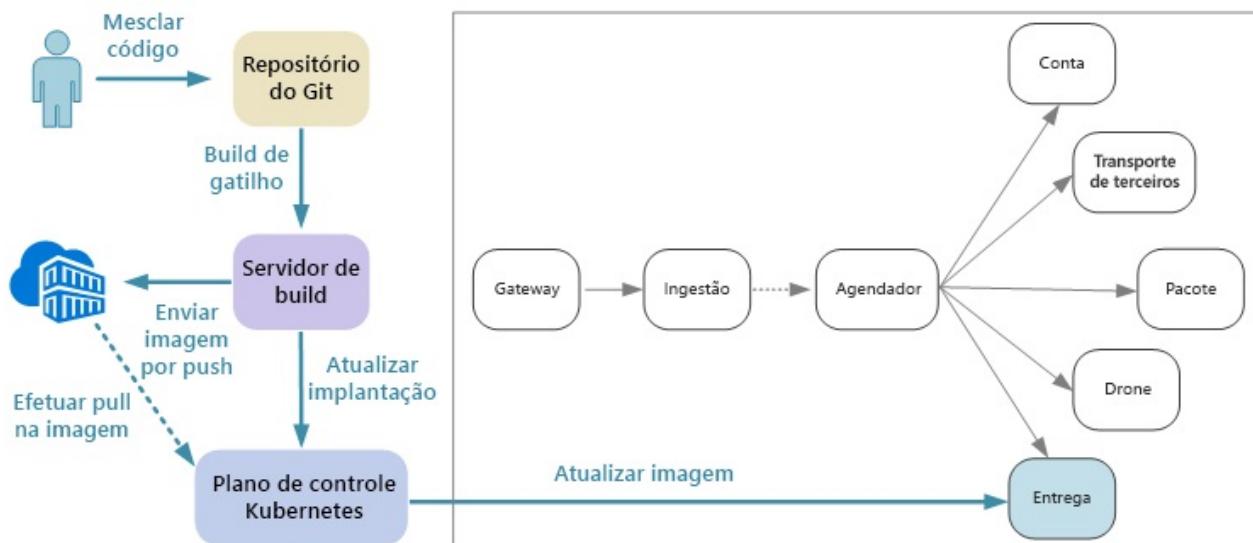
- Usar uma ferramenta de painel, tal como Kibana ou Grafana, para visualizar e monitorar os dados. O serviço de painel também pode ser executado dentro de um contêiner no cluster.

Para logs de aplicativo, considere o uso de Fluentd e Elasticsearch. Fluentd é um coletor de dados de software livre e Elasticsearch é um banco de dados de documento que é otimizado para atuar como um mecanismo de pesquisa. Usando essa abordagem, cada serviço envia logs para stdout e stderr e o Kubernetes grava esses fluxos para o sistema de arquivos local. O Fluentd coleta os logs, opcionalmente enriquece-os com metadados adicionais do Kubernetes e envia os logs para o Elasticsearch. Use o Kibana, o Grafana ou uma ferramenta semelhante para criar um painel para o Elasticsearch. O Fluentd é executado como um conjunto de daemons no cluster, o que garante que um pod do Fluentd é atribuído a cada nó. Você pode configurar o Fluentd para coletar logs do kubelet, bem como os logs de contêiner. Em grandes volumes, gravar logs no sistema de arquivos local pode se tornar um gargalo de desempenho, especialmente quando vários serviços estão em execução no mesmo nó. Monitore a latência do disco e a utilização do sistema de arquivos na produção.

Uma vantagem de usar Fluentd com Elasticsearch para logs é que os serviços não exigem nenhuma dependência de biblioteca adicional. Cada serviço grava apenas para stdout e stderr e o Fluentd fica responsável pela exportação dos logs para o Elasticsearch. Além disso, as equipes que escrevem código de serviços não precisam entender como configurar a infraestrutura de log. Um desafio é configurar o cluster Elasticsearch para uma implantação de produção de modo que ele seja dimensionado para lidar com o tráfego.

Integração Contínua

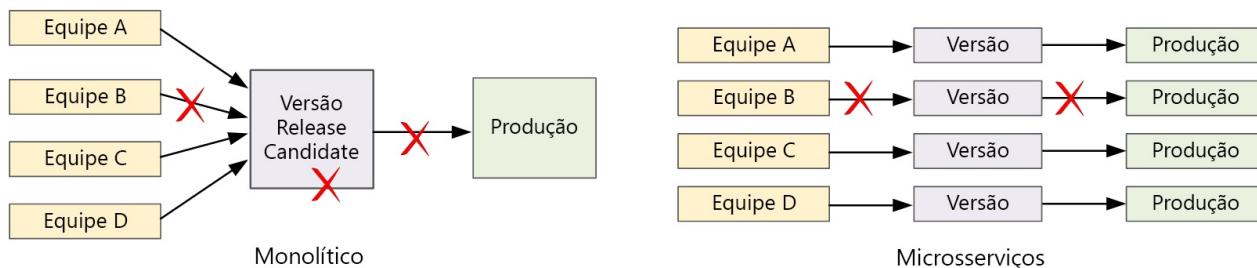
CI/CD (integração contínua e entrega contínua) são um requisito fundamental para alcançar êxito com microsserviços. Sem um bom processo de CI/CD, você não obterá a agilidade que os microsserviços prometem. Alguns dos desafios de CI/CD para microsserviços são oriundos de se ter várias bases de código e ambientes de build heterogêneos para os diversos serviços.



Ciclos de lançamento mais rápidos são um dos maiores motivos para se adotar uma arquitetura de microsserviços.

Em um aplicativo totalmente monolítico, há um único pipeline de build cuja saída é o executável do aplicativo. Todo o trabalho de desenvolvimento alimenta este pipeline. Se for encontrado um bug de alta prioridade, uma correção deverá ser integrada, testada e publicada, o que poderá atrasar o lançamento dos novos recursos. É verdade que você pode atenuar esses problemas com módulos bem fatorados e usando ramificações de recursos para minimizar o impacto de alterações de código. Mas conforme o aplicativo se torna mais complexo e mais recursos são adicionados, o processo de liberação para um monolito tende a se tornar mais frágil e suscetível a interrupção.

Seguindo a filosofia de microsserviços, nunca deverá haver uma *release train* longa em que todas as equipes precisam se mobilizar. A equipe que cria o serviço "A" pode lançar uma atualização a qualquer momento, sem esperar que as alterações no serviço "B" sejam mescladas, testadas e implantadas. O processo de CI/CD é essencial para tornar isso possível. O pipeline de lançamento deve ser automatizado e altamente confiável, para que os riscos de implantação de atualizações sejam minimizados. Se você estiver liberando para produção diariamente ou várias vezes ao dia, as regressões ou as interrupções de serviço deverão ser muito raras. Ao mesmo tempo, se uma atualização inválida é implantada, você deve ter uma maneira confiável de reverter ou efetuar roll forward rapidamente para uma versão anterior de um serviço.



Quando falamos de CI/CD, na verdade estamos falando de processos relacionados: Integração contínua, entrega contínua e implantação contínua.

- **Integração contínua** significa que as alterações de código são frequentemente mescladas à ramificação principal, usando processos de build e de teste automatizados para garantir que o código na ramificação principal tenha sempre qualidade em nível de produção.
- **Entrega contínua** significa que as alterações de código que passam pelo processo de CI são publicadas automaticamente em um ambiente similar ao de produção. A implantação no ambiente de produção dinâmico pode exigir aprovação manual, mas caso contrário, é automatizada. A meta é que o seu código esteja sempre pronto para implantação na produção.
- **Implantação contínua** significa que as alterações de código que passam pelo processo de CI/CD são automaticamente implantadas em produção.

No contexto de Kubernetes e microsserviços, o estágio de CI trata de compilar e testar as imagens de contêiner e enviá-las por push para um registro de contêiner. No estágio de implantação, especificações de pod são atualizadas para que a imagem de produção mais recente seja escolhida.

Desafios

- **Várias bases de código pequenas independentes.** Cada equipe é responsável por criar seu próprio serviço, com seu próprio pipeline de build. Em algumas organizações, as equipes podem usar repositórios de código separados. Isso pode levar a uma situação em que o conhecimento de como compilar o sistema é disseminado entre as equipes e ninguém na organização sabe como implantar o aplicativo inteiro. Por exemplo, o que acontecerá em um cenário de recuperação de desastre, se você precisar implantar rapidamente para um novo cluster?
- **Múltiplas linguagens de programação e frameworks.** Com cada equipe usando seu próprio conjunto de tecnologias, pode ser difícil criar um processo de build único que funcione em toda a organização. O processo de build deve ser flexível o suficiente para que cada equipe possa adaptá-lo para sua linguagem de programação ou estrutura de preferência.
- **Integração e teste de carga.** Com as equipes de liberação de atualizações em seu próprio ritmo, pode ser difícil projetar testes de ponta a ponta robustos, especialmente quando os serviços têm dependências em outros serviços. Além disso, o processo de execução de um cluster de produção completo pode ser cara, portanto, é improvável que cada equipe possa executar seu próprio cluster completo em escalas de produção, apenas para teste.
- **Gerenciamento de versão.** Cada equipe deve ter a capacidade de implantar uma atualização em produção. Isso não significa que cada membro da equipe tem permissões para fazer isso. Mas ter uma função de Gerenciador de Versão centralizada pode reduzir a velocidade das implantações. Quanto mais o processo de CI/CD for automatizado e confiável, menos deverá haver necessidade de uma autoridade central. Dito isso, você pode ter diferentes políticas para liberar atualizações dos principais recursos e correções de bugs secundários. Ser descentralizado não significa que não deve haver governança.
- **Controle de versão de imagem de contêiner.** Durante o ciclo de desenvolvimento e teste, o processo de CI/CD cria muitas imagens de contêiner. Somente alguns deles são candidatos a lançamento, e apenas alguns desses serão enviados por push para produção. É preciso ter uma estratégia de controle de versão clara para que você saiba quais imagens estão implantadas atualmente para produção e possa reverter para uma versão anterior, se necessário.
- **Atualizações de serviço.** Quando você atualizar um serviço para uma nova versão, ele não deverá interromper outros serviços que dependem dele. Se você fizer uma atualização sem interrupção, haverá um período de tempo quando uma mistura de versões estará em execução.

Esses desafios refletem uma tensão fundamental. Por outro lado, as equipes precisam trabalhar de modo tão independente quanto possível. Por outro lado, alguma coordenação é necessária para que uma única pessoa possa realizar tarefas como executar um teste de integração, reimplantar a solução inteira para um novo cluster ou reverter uma atualização inválida.

Abordagens de CI/CD para microserviços

É uma prática recomendada que cada equipe de serviço coloque o respectivo ambiente de build em um contêiner. Esse contêiner deverá ter todas as ferramentas de build necessárias para compilar os artefatos de código para o serviço dessa equipe. Geralmente, você pode encontrar uma imagem do Docker oficial para sua linguagem de programação e framework. Em seguida, você pode usar `docker run` ou o Docker Compose para executar o build.

Com essa abordagem, é simples configurar um novo ambiente de build. Um desenvolvedor que deseja compilar seu código não precisa instalar um conjunto de ferramentas de build, apenas executar a imagem de contêiner. E o que é talvez ainda mais importante, o servidor de build pode ser configurado para fazer a mesma coisa. Dessa forma, você não precisa instalar essas ferramentas no servidor de build nem gerenciar versões conflitantes de ferramentas.

Para desenvolvimento e teste locais, use o Docker para executar o serviço dentro de um contêiner. Como parte desse processo, talvez seja necessário executar outros contêineres com serviços simulados ou bancos de dados de teste necessários para teste local. Você pode usar o Docker Compose para coordenar esses contêineres ou usar o Minikube para executar o Kubernetes localmente.

Quando o código estiver pronto, abra uma solicitação pull e realize a merge no master. Isso iniciará um trabalho no servidor de build:

1. Compile os ativos de código.
2. Execute testes de unidade no código.
3. Compile a imagem de contêiner.
4. Teste a imagem de contêiner executando testes funcionais em um contêiner em execução. Essa etapa pode detectar erros no arquivo de Docker como um ponto de entrada inválido.
5. Envie a imagem por push para um registro de contêiner.
6. Atualize o cluster de teste com a nova imagem para executar testes de integração.

Quando a imagem estiver pronta para entrar em produção, atualize os arquivos de implantação necessários para especificar a imagem mais recente, inclusive eventuais arquivos de configuração do Kubernetes. Em seguida, aplique a atualização ao cluster de produção.

Aqui estão algumas recomendações para tornar as implantações mais confiáveis:

- Defina as convenções de toda a organização para marcações de contêiner, controle de versão e convenções de nomenclatura de recursos implantados para o cluster (pods, serviços e assim por diante). Isso pode facilitar o diagnóstico de problemas de implantação.
- Crie dois registros de contêiner separados, um para desenvolvimento/teste e outro para produção. Não envie uma imagem por push para o registro de produção até que você esteja pronto para implantá-lo em produção. Se você combinar essa prática com controle de versão semântico de imagens de contêiner, isso poderá reduzir a chance de acidentalmente implantar uma versão não aprovada para lançamento.

Atualizando serviços

Há várias estratégias para atualizar um serviço que já está em produção. Aqui, abordamos três opções comuns: Atualização sem interrupção, implantação "blue-green" e versão canário.

Atualização sem interrupção (Rolling update)

Em uma atualização sem interrupção, você implanta novas instâncias de um serviço e as novas instâncias começam a receber solicitações imediatamente. À medida que as novas instâncias chegam, as anteriores são removidas.

Atualizações sem interrupção são o comportamento padrão no Kubernetes quando você atualiza a especificação de pod para uma implantação. O controlador de implantação cria um novo ReplicaSet para os pods atualizados. Em seguida, ele expande o novo ReplicaSet e reduz simultaneamente o antigo, para manter a contagem de réplicas.

desejada. Ela não exclui os pods antigos até que os novos estejam prontos. O Kubernetes mantém um histórico da atualização, de modo que você pode usar `kubectl` para reverter uma atualização se necessário.

Se o seu serviço executa uma tarefa de inicialização longa, você pode definir um teste de preparação. A investigação de preparação relata quando o contêiner está pronto para começar a receber tráfego. O Kubernetes não enviará tráfego para o pod até que a investigação relate êxito.

Um desafio de reverter atualizações é que durante o processo de atualização uma mistura das versões antiga e nova estão em execução e recebendo tráfego. Durante esse período, qualquer solicitação poderia ser roteada para qualquer uma das duas versões. Isso pode causar ou não problemas, dependendo do escopo das alterações entre as duas versões.

Implantação "blue-green"

Em uma implantação "blue-green", você deve implantar a nova versão juntamente com a versão anterior. Depois de validar a nova versão, você pode mudar todo o tráfego da versão anterior para a nova versão, de uma só vez. Após a troca, você deve monitorar o aplicativo para quaisquer problemas. Se algo der errado, você poderá retornar à versão antiga. Se nenhum problema ocorrer, você poderá excluir a versão antiga.

Com um aplicativo monolítico ou de N camadas mais tradicional, a implantação "blue-green" geralmente significa provisionar dois ambientes idênticos. Você implantaria a nova versão em um ambiente de preparo e então redirecionaria o tráfego de cliente para o ambiente de preparo — por exemplo, alternando VIPs.

No Kubernetes, você não precisa provisionar um cluster separado para fazer implantações "blue-green". Em vez disso, você pode tirar proveito de seletores. Crie um novo recurso de implantação com uma nova especificação de pod e um conjunto diferente de rótulos. Crie essa implantação sem excluir a implantação anterior nem modificar o serviço que aponta para ela. Quando os novo pods estiverem em execução, você poderá atualizar o seletor do serviço para corresponder à nova implantação.

Uma vantagem de implantações "blue-green" é que o serviço muda todos os pods simultaneamente. Depois que o serviço for atualizado, todas as novas solicitações serão roteadas para a nova versão. Uma desvantagem é que durante a atualização você executa o dobro de pods para o serviço (os atuais e os próximos). Se os pods exigirem muitos recursos de CPU ou de memória, talvez você precisará expandir o cluster temporariamente para dar conta do consumo de recursos.

Versão canário

Em uma versão canário, você distribui uma versão atualizada para um número pequeno de clientes. Em seguida, você monitora o comportamento do novo serviço antes de implantá-lo em todos os clientes. Isso lhe permite fazer uma distribuição lenta de forma controlada, observar dados reais e identificar problemas antes que todos os clientes sejam afetados.

Uma versão canário é mais complexa de gerenciar do que a atualização sem interrupção ou "blue-green", porque você deve rotear solicitações dinamicamente para diferentes versões do serviço. No Kubernetes, você pode configurar um serviço para abranger dois conjuntos de réplicas (um para cada versão) e ajustar as contagens de réplicas manualmente. No entanto, essa abordagem tem uma granularidade bastante alta devido ao modo como o Kubernetes balanceia a carga entre os pods. Por exemplo, se você tiver um total de dez réplicas, só poderá realizar deslocamentos de tráfego em incrementos de 10%. Se você estiver usando uma malha de serviço, poderá usar as regras de roteamento de malha do serviço para implementar uma estratégia de versão canário mais sofisticada.

Conclusão

Nos últimos anos, houve uma mudança radical na indústria, um movimento saindo da criação de *sistemas de registro* em direção à criação de *sistemas de engajamento*.

Sistemas de registro são aplicativos de gerenciamento de dados de back office tradicionais. No núcleo desses sistemas geralmente há um RDBMS, que é a única fonte de verdade. O termo "sistema de engajamento" é creditado a Geoffrey Moore em seu artigo de 2011, Systems of Engagement and the Future of Enterprise IT (sistemas de engajamento e o futuro do TI empresarial). **Sistemas de engajamento** são aplicativos voltados à comunicação e colaboração. Eles conectam pessoas em tempo real. Eles devem estar disponíveis 24 horas por dia, 7 dias por semana. Novos recursos são introduzidos regularmente sem que o aplicativo fique offline. Os usuários esperam mais e são menos pacientes com relação a tempo de inatividade ou atrasos inesperados.

No espaço do consumidor, uma melhor experiência de usuário pode ter valor comercial mensurável. A quantidade de tempo durante o qual um usuário se envolve com um aplicativo pode ser convertida diretamente em receita. E no universo dos sistemas comerciais, as expectativas dos usuários mudaram. Se esses sistemas visam promover comunicação e colaboração, eles devem usar como referência os aplicativos voltados para o consumidor.

Microsserviços são uma resposta a essa paisagem em mudança. Ao decompor um aplicativo monolítico em um grupo de serviços acoplados de forma flexível, podemos controlar o ciclo de versão de cada serviço e habilitar atualizações frequentes, sem tempo de inatividade nem alterações significativas. Os microsserviços também ajudam com a escalabilidade, o isolamento de falhas e a resiliência. Enquanto isso, plataformas de nuvem estão facilitando o build e a execução de microsserviços, com o provisionamento automatizado de recursos de computação, orquestradores de contêiner como um serviço e ambientes sem servidor controlados por eventos.

Mas, como vimos, arquiteturas de microsserviços também enfrentam muitos desafios. Para ter êxito, você deve iniciar de um design sólido. Você deve pensar cuidadosamente ao analisar o domínio, escolher as tecnologias, modelar dados, projetar APIs e criar uma cultura de DevOps madura. Esperamos que este guia e que a implementação de referência que o acompanha tenham ajudado a iluminar a jornada.

Implantação e Monitoramento

Em nosso último capítulo, focaremos no monitoramento, resistência a falhas e testes dos microsserviços.

Monitoramento

Prometheus

O Prometheus é um kit open source de ferramentas de monitoramento e alerta originalmente criado no SoundCloud. Desde a sua criação em 2012, muitas empresas e organizações adotaram o Prometheus, e o projeto tem uma comunidade de desenvolvedores e usuários muito ativa. Agora é um projeto de código aberto independente e mantido independentemente de qualquer empresa.

Características

As principais características do Prometheus são:

- Um modelo de dados multidimensional com dados de série temporal identificados por pares de nome de métrica e chave/valor;
- PromQL, uma linguagem de consulta flexível para se aproveitar essa dimensionalidade;
- Nenhuma dependência de armazenamento distribuído; nós de servidor único são autônomos;
- Coleção de séries temporais por meio de um modelo pull sobre HTTP;
- A série temporal de envio é suportada através de um gateway intermediário;
- Os destinos são descobertos por meio da descoberta de serviço ou da configuração estática;
- Vários modos de suporte a gráficos e painéis;

Componentes

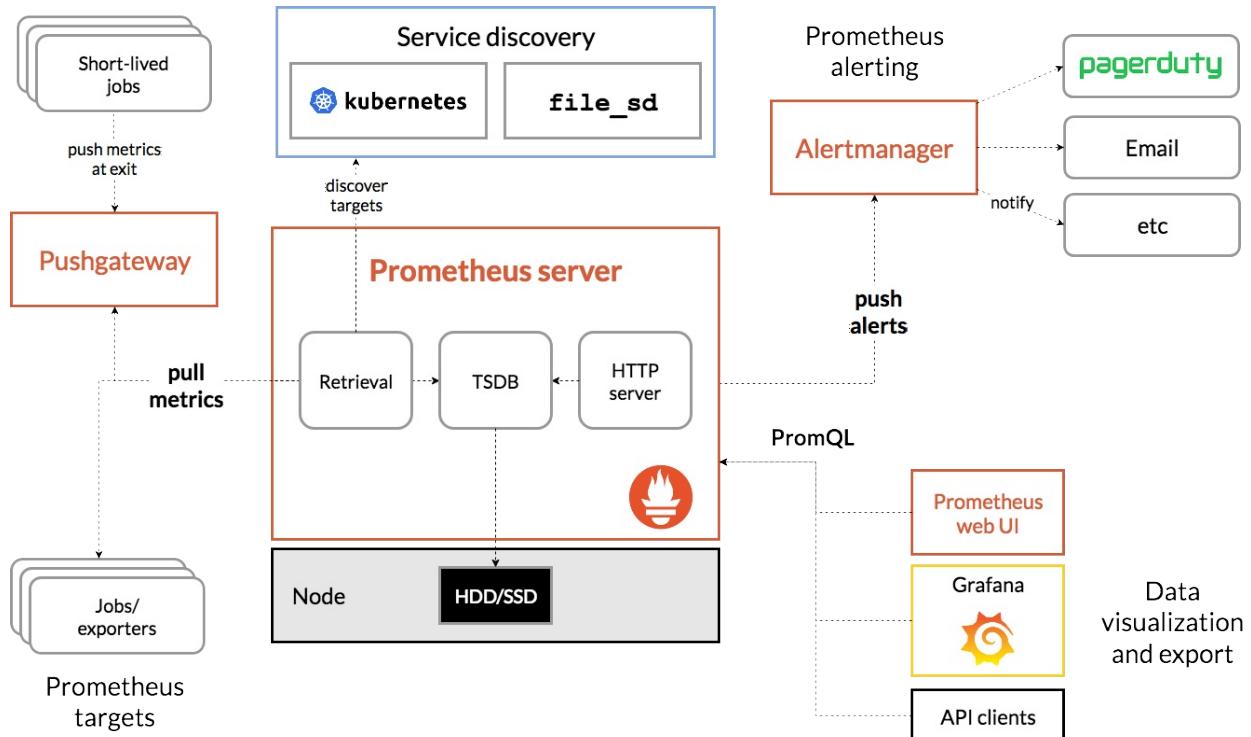
O ecossistema Prometheus consiste em vários componentes, muitos dos quais são opcionais:

- O servidor principal Prometheus que busca e armazena dados de séries temporais;
- Bibliotecas do cliente para instrumentar o código do aplicativo;
- Um gateway de envio para apoiar trabalhos de curta duração
- Exportadores especiais para serviços como HAProxy, StatsD, Graphite, etc;
- Um gerenciador de alertas para lidar com alertas
- Várias ferramentas de suporte

A maioria dos componentes do Prometheus é escrita em Go, facilitando sua criação e implementação como binários estáticos.

Arquitetura

Este diagrama ilustra a arquitetura do Prometheus e alguns de seus componentes do ecossistema:



O Prometheus obtém as métricas de trabalhos instrumentados, diretamente ou através de um gateway de envio intermediário para trabalhos de curta duração. Ele armazena todos os exemplos obtidos localmente e executa regras sobre esses dados para agregar e registrar novas séries temporais a partir de dados existentes ou gerar alertas. Grafana ou outros consumidores de API podem ser usados para visualizar os dados coletados.

Quando utilizá-lo?

O Prometheus funciona bem para gravar qualquer série temporal puramente numérica. Ele se ajusta tanto ao monitoramento centrado na máquina quanto ao monitoramento de arquiteturas altamente dinâmicas orientadas a serviços. Em um mundo de microserviços, seu suporte para coleta e consulta de dados multidimensionais é uma força especial.

Prometheus é projetado para confiabilidade, para ser o sistema que você utiliza durante uma interrupção para permitir que você rapidamente diagnostique problemas. Cada servidor Prometheus é independente, não dependendo do armazenamento de rede ou de outros serviços remotos. Você pode confiar nele quando outras partes de sua infraestrutura estiverem quebradas e não precisar configurar uma infraestrutura extensiva para usá-lo.

Quando não utilizá-lo?

Prometheus valoriza a confiabilidade. Você sempre pode ver quais estatísticas estão disponíveis sobre o seu sistema, mesmo em condições de falha. Se você precisar de 100% de precisão, como por faturamento por solicitação, o Prometheus não é uma boa escolha, pois os dados coletados provavelmente não serão detalhados e completos o suficiente. Nesse caso, seria melhor usar algum outro sistema para coletar e analisar os dados para faturamento e a Prometheus para o restante de seu monitoramento.

Grafana

O Grafana permite consultar, visualizar, alertar e entender suas métricas, independentemente de onde elas estejam armazenadas. Criar, explorar e compartilhar painéis com sua equipe e promover uma cultura orientada por dados.

Micrometer

O Micrometer fornece uma fachada simples sobre os clientes de instrumentação para os sistemas de monitoramento mais populares, permitindo instrumentar o código do aplicativo baseado em JVM sem vendor lock-in. Pense SLF4J, mas para métricas.

Utilizando em nosso projeto

Descompactar o Prometheus e executá-lo com arquivo de configuração:

prometheus.yml

```
global:
  scrape_interval:      15s # By default, scrape targets every 15 seconds.

  # Attach these labels to any time series or alerts when communicating with
  # external systems (federation, remote storage, Alertmanager).
  external_labels:
    monitor: 'codelab-monitor'

  # A scrape configuration containing exactly one endpoint to scrape:
  # Here it's Prometheus itself.
  scrape_configs:
    # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
    - job_name: 'prometheus'

      # Override the global default and scrape targets from this job every 5 seconds.
      scrape_interval: 5s

      static_configs:
        - targets: ['localhost:9091']
```

```
> prometheus-2.7.0.windows-amd64\prometheus.exe --config.file=prometheus.yml --web.enable-admin-api --web.listen-address=:9091
```

Para testar a execução, acesse o endereço <http://localhost:9091>.

Agora, vamos iniciar o Grafana e conectá-lo ao Prometheus, para isso, descompacte o Grafana e no diretório `conf`, crie uma cópia do arquivo `sample.ini` com o nome `custom.ini`, execute o Grafana:

```
> grafana-5.4.3\bin\grafana-server.exe
```

Para testar a execução, acesse o endereço <http://localhost:3000>.

Criando uma fonte de dados Prometheus

Para criar uma fonte de dados do Prometheus:

1. Clique no logotipo do Grafana para abrir o menu da barra lateral.
2. Clique em "Data Sources" na barra lateral.
3. Clique em "Add New".
4. Selecione "Prometheus" como o tipo.
5. Definir a URL do servidor Prometheus apropriada (por exemplo, <http://localhost:9091>)
6. Clique em "Adicionar" para salvar a nova fonte de dados.

Importando dashboards pré-criados do Grafana.com

O Grafana.com mantém uma coleção de dashboards compartilhados que podem ser baixados e usados com instâncias autônomas do Grafana. Use a opção "Filtro" do Grafana.com para procurar dashboards somente pela fonte de dados "Prometheus".

O link direto é <https://grafana.com/dashboards/2>

Clique em "+" e "Import" e cole o conteúdo do JSON, em seguida aponte para a fonte de dados correta.

Configurando a aplicação

Incluir as dependencias do Micrometer na aplicação livro-service:

pom.xml

```
<!-- Micrometer core dependency -->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-core</artifactId>
</dependency>
<!-- Micrometer Prometheus registry -->
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Consultar a url <http://localhost:8080/actuator>, deve retornar um end-point do prometheus

O end-point <http://localhost:8080/actuator/prometheus> deve retornar as métricas do prometheus.

Nosso próximo passo é configurar o Prometheus para receber as métricas de nossa aplicação:

```
global:
  # Configuração atual omitida

  # Novidade aqui
  - job_name: 'livro-service'
    scrape_interval: 5s
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['localhost:8080']
```

Fontes

- <https://prometheus.io/docs/introduction/overview/>
- <https://dzone.com/articles/prometheus-monitoring-with-grafana>
- <https://dzone.com/articles/monitoring-using-spring-boot-2-prometheus-and-graf>
- <https://dzone.com/articles/monitoring-using-spring-boot-20-prometheus-and-gra>
- <https://thepracticalsysadmin.com/introduction-to-grafana/>

Tolerância a Falhas

Um sistema distribuído típico consiste em vários serviços que colaboram juntos.

Esses serviços são propensos a falhas ou respostas atrasadas. Se um serviço falhar, isso pode afetar outros serviços que afetam o desempenho e possivelmente tornar outras partes do aplicativo inacessíveis ou, no pior dos casos, derrubar todo o aplicativo.

Naturalmente, existem soluções disponíveis que ajudam a tornar os aplicativos resilientes e tolerantes a falhas - uma dessas estruturas é a **Hystrix**.

A biblioteca de framework Hystrix ajuda a controlar a interação entre os serviços, fornecendo tolerância a falhas e tolerância a latência. Ele melhora a resiliência geral do sistema, isolando os serviços com falha e interrompendo o efeito de falhas em cascata.

Analisaremos como a Hystrix auxilia quando um serviço ou sistema falha e o que a Hystrix pode realizar nessas circunstâncias.

Usaremos a biblioteca e implementaremos o padrão corporativo "Circuit Breaker", que descreve uma estratégia contra a falha em cascata em diferentes níveis em um aplicativo.

O princípio é análogo à eletrônica: a Hystrix observa métodos para falhas nas chamadas de serviços relacionados. Se houver essa falha, ele abrirá o circuito e encaminhará a chamada para um método de fallback.

A biblioteca tolerará falhas até um limite. Além disso, deixa o circuito aberto. O que significa que ele encaminhará todas as chamadas subsequentes para o método de fallback, para evitar futuras falhas. Isso cria um buffer de tempo para o serviço relacionado recuperar de seu estado com falha.

Hystrix

O Hystrix é uma biblioteca de tolerância a falhas e latência projetada para isolar pontos de acesso em sistemas remotos, serviços e bibliotecas de terceiros, parar falhas em cascata e habilitar a resiliência em sistemas distribuídos complexos onde a falha é inevitável.

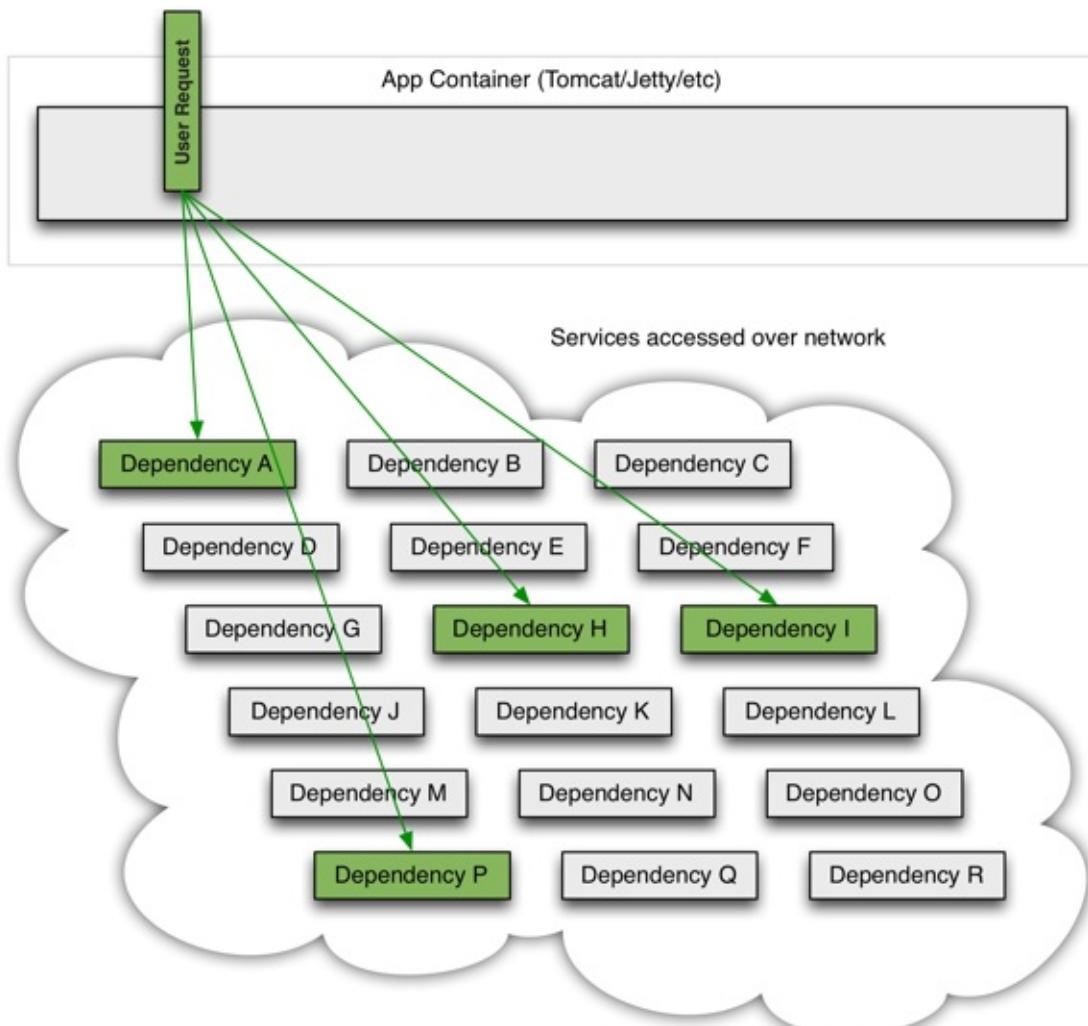
Quais tipos de problemas o Hystrix resolve?

Aplicações em arquiteturas complexas contém muitas dependências, algumas inevitavelmente podem falhar em algum ponto.

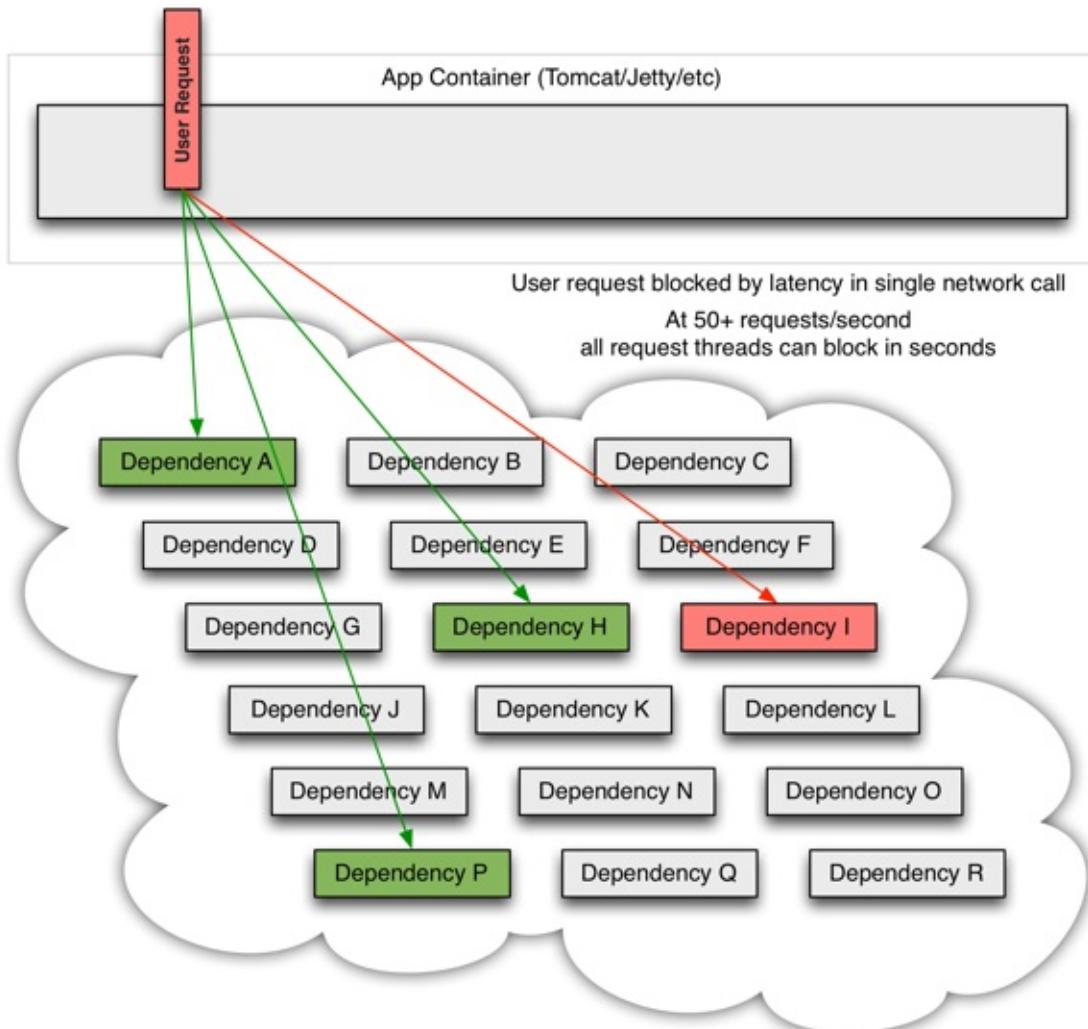
Imagine o cenário onde você tem 30 serviços e cada um tem 99.99% de uptime, portanto você pode esperar que:

$99.99^{30} = 99.7\%$ uptime
0.3% de 1 bilhão de requests = 3.000.000 falhas 2+ horas de downtime por mês,
mesmo que todas as dependências possuam um excelente uptime.

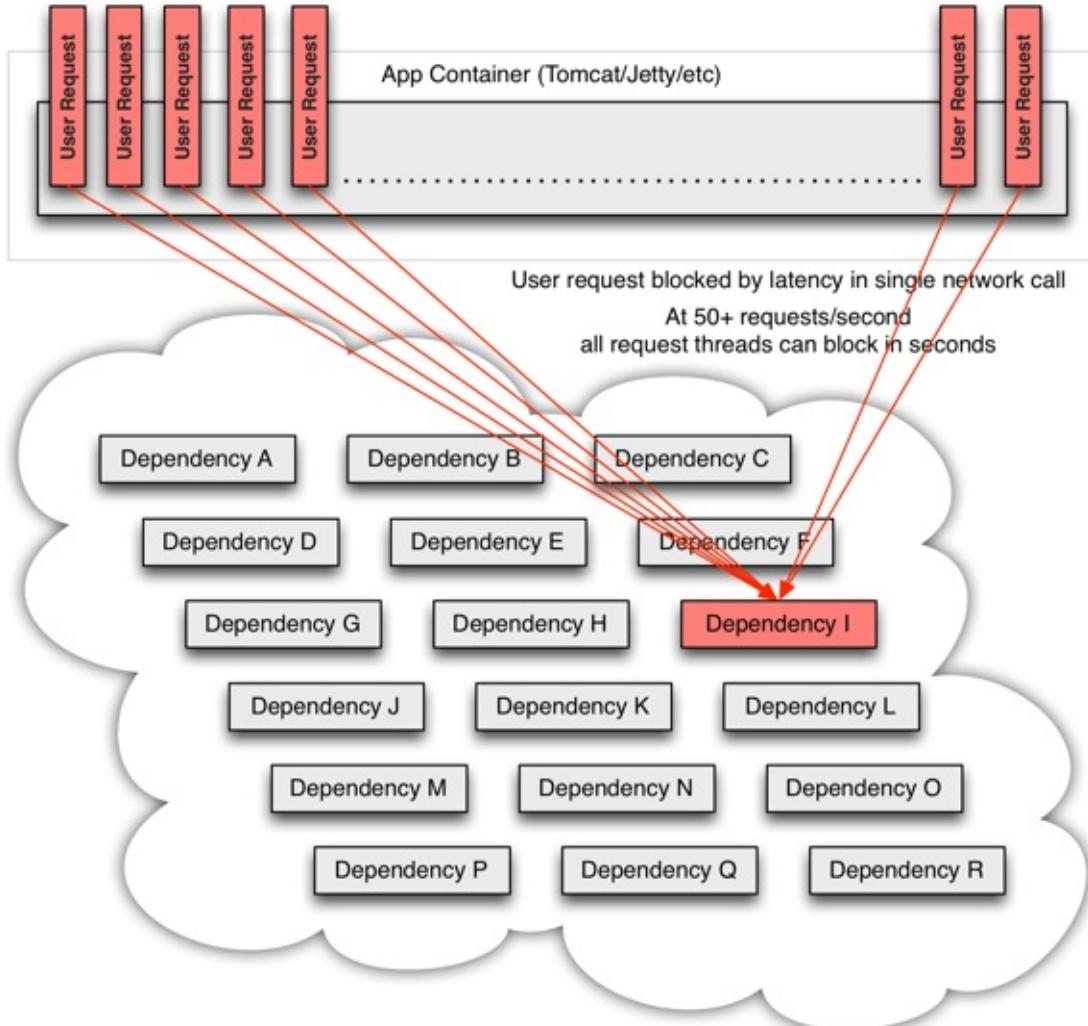
Em um cenário ideal o sistema funciona da seguinte forma:



Portanto algum serviço pode começar a falhar:



Isso significa que o UserRequest fica bloqueado porque a dependência I ilustrada em vermelho está em falha, seja um timeout, um problema de infra, base de dados etc.. resultando para o usuário um possível **Server Error 500** e todos os futuros requests irão travar devido à esse serviço que não está saudável.



Hystrix foi projetado para fazer o seguinte:

- Dar proteção e controle sobre a latência e a falha nas dependências acessadas (normalmente através da rede) através de bibliotecas de clientes de terceiros.
- Parar falhas em cascata em um sistema distribuído complexo.
- Falhar rapidamente e permitir rápida recuperação.
- Fallback e interromper processos quando possível.
- Permite monitoramento, alerta e controle operacional quase em tempo real.

Tornando a inclusão de Avaliações tolerante a falhas

O serviço de inclusão de avaliações realiza uma requisição para o serviço de consulta de livros antes de efetuar a inclusão da avaliação, porém, um novo requisito que temos é tornar esta operação tolerante a falhas no serviço de consulta de livros, ou seja, mesmo que o serviço de consulta de livros falhe a inclusão da avaliação deverá proceder com um valor default para o nome do livro.

Inicialmente, iremos incluir a dependência do Hystrix no projeto avaliacao-service:

pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

A anotação `@EnableCircuitBreaker` examinará o caminho de classe para qualquer implementação de circuit-breaker compatível.

Para usar o Hystrix explicitamente, você deve anotar `AvaliacaoServiceApplication` com `@EnableHystrix`:

AvaliacaoServiceApplication

```
package com.acme.avaliacaoservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication
@EnableDiscoveryClient
// Novidade aqui
@EnableCircuitBreaker
public class AvaliacaoServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(AvaliacaoServiceApplication.class, args);
    }
}
```

Agora, criaremos um serviço para retornar o título do livro, isolando esta funcionalidade:

TituloLivroService

```
@Service
public class TituloLivroService {

    @HystrixCommand(fallbackMethod = "getTituloDefault")
    public String getTitulo(Long livroId) {

        RestTemplate restTemplate = new RestTemplate();
        String livroResourceUrl = "http://localhost:8080/livros/";

        ResponseEntity<Livro> responseLivro = restTemplate.getEntity(livroResourceUrl + livroId, Livro.class);

        return responseLivro.getBody().getTitulo();
    }

    @SuppressWarnings("unused")
    private String getTituloDefault(Long livroId) {
        return "Erro ao consultar o título: " + livroId;
    }
}
```

O último passo é alterar `AvalicacoesController` para que utilize o serviço de consulta do nome do livro:

AvalicacoesController

```
// Código atual omitido
private final AvaliacaoRepository repository;

// Novidades aqui
private final TituloLivroService tituloLivroService;

AvaliacoesController(AvaliacaoRepository repository, TituloLivroService tituloLivroService) {
    this.repository = repository;
    this.tituloLivroService = tituloLivroService;
}

// Código atual omitido

// Novidades aqui
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Avaliacao adicionarAvaliacao(@RequestBody Avaliacao avaliacao) throws IOException {
    logger.info("adicionarAvaliacao: " + avaliacao);

    String tituloLivro = tituloLivroService.getTitulo(avaliacao.getLivroId());
    logger.info("Título do livro avaliado: " + tituloLivro);

    return repository.save(avaliacao);
}
```

Para testar, tente incluir uma avaliação com o serviço de livros ativo e em seguida desligue o servidor de livros-service.

Mas agora, se o serviço de consulta de livros estiver ativo e você tentar incluir uma avaliação para um livro inexistente, vai ter um resultado indesejado, a função de fallback será acionada.

Para evitar isso, temos que ajustar o serviço de consulta de nomes de livros para não emitir exceção somente no caso do serviço estar ativo e retornar 404 para o determinado livro.

```

package com.acme.avaliacaoservice;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.RestTemplate;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

@Service
public class TituloLivroService {

    Logger logger = LoggerFactory.getLogger(TituloLivroService.class);

    @HystrixCommand(fallbackMethod = "getTituloDefault")
    public String getTitulo(Long livroId) {
        RestTemplate restTemplate = new RestTemplate();
        String livroResourceUrl = "http://localhost:8080/livros/";
        try {
            ResponseEntity<Livro> responseLivro = restTemplate.getForEntity(livroResourceUrl + livroId, Livro.class);
            return responseLivro.getBody().getTitulo();
        } catch (HttpClientErrorException ex) {
            if (ex.getRawStatusCode() == HttpStatus.NOT_FOUND.value()) {
                return null;
            } else {
                logger.error("Ocorreu um erro na comunicação com o serviço de livros", ex);
                throw ex;
            }
        }
    }

    @SuppressWarnings("unused")
    private String getTituloDefault(Long livroId) {
        return "Erro ao consultar o título: " + livroId;
    }
}

```

A versão final em AvaliacaoController fica assim:

```

@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Avaliacao adicionarAvaliacao(@RequestBody Avaliacao avaliacao) throws IOException {
    logger.info("adicionarAvaliacao: " + avaliacao);

    String tituloLivro = tituloLivroService.getTitulo(avaliacao.getLivroId());
    if (tituloLivro != null) {
        logger.info("Titulo do livro avaliado: " + tituloLivro);
    } else {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Livro não existe: " + avaliacao.getLivroId());
    }

    return repository.save(avaliacao);
}

```

Feign

Feign é um cliente de serviço da web declarativo. Facilita a criação de clientes de serviços da Web. Para usar Feign, crie uma interface e anote-a. Ele possui suporte a anotações plugáveis, incluindo anotações Feign e anotações JAX-RS. O Feign também suporta codificadores e decodificadores plugáveis. O Spring Cloud adiciona suporte para anotações do Spring MVC e para usar os mesmos HttpMessageConverters usados por padrão no Spring Web. O Spring Cloud integra o Ribbon e o Eureka para fornecer um cliente http de carga balanceada ao usar o Feign.

Utilizando o Feign para as chamadas entre serviços

Para utilizar o Feign devemos incluir sua dependência:

pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Devemos também anotar `AvaliacaoServiceApplication` com `@EnableFeignClients`:

AvaliacaoServiceApplication

```
package com.acme.avaliacaoservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
// Novidade aqui
@EnableFeignClients
public class AvaliacaoServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(AvaliacaoServiceApplication.class, args);
    }
}
```

Vamos criar uma classe cliente para realizar as requisições REST, é algo parecido com o conceito do repository.

LivroClient

```
package com.acme.avaliacaoservice;

import java.util.List;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@FeignClient(name = "livros", url = "http://localhost:8080")
public interface LivroClient {
    @RequestMapping(method = RequestMethod.GET, value = "/livros")
    List<Livro> getLivros();

    @RequestMapping(method = RequestMethod.GET, value = "/livros/{livroId}")
    Livro getLivroPorId(@PathVariable("livroId") Long livroId);
}
```

O último passo é ajustar nosso controller para utilize o novo método para recuperar o título do livro:

```
// Código atual omitido

@RestController
@RequestMapping("/avaliacoes")
public class AvaliacoesController {

    Logger logger = LoggerFactory.getLogger(AvaliacoesController.class);

    private final AvaliacaoRepository repository;

    // Novidades aqui
    private final LivroClient livroClient;

    AvaliacoesController(AvaliacaoRepository repository, LivroClient livroClient) {
        this.repository = repository;
        this.livroClient = livroClient;
    }

    // Código atual omitido

    // Novidades aqui
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Avaliacao adicionarAvaliacao(@RequestBody Avaliacao avaliacao) throws IOException {
        logger.info("adicionarAvaliacao: " + avaliacao);

        try {
            String tituloLivro = livroClient.getLivroPorId(avaliacao.getLivroId()).getTitulo();
            logger.info("Titulo do livro avaliado: " + tituloLivro);
        } catch (FeignException ex) {
            if (ex.status() == HttpStatus.NOT_FOUND.value()) {
                throw new ResponseStatusException(HttpStatus.BAD_REQUEST,
                    "Livro não existe: " + avaliacao.getLivroId());
            } else {
                throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE,
                    "Ocorreu um erro ao consultar o título do livro: " + ex.getMessage());
            }
        }
        return repository.save(avaliacao);
    }
}
```

Compile e teste a aplicação, tudo deve estar funcionando normalmente, mas perceba que perdemos a funcionalidade do circuit-breaker neste ajuste, vamos recuperá-la.

Hystrix e Feign

Agora, vamos combinar a utilização do Hystrix e do Feign, para isso, primeiro temos que ativar uma propriedade de integração das duas bibliotecas no arquivo de propriedades da aplicação:

bootstrap.properties

```
...
feign.hystrix.enabled=true
```

Ajustamos agora nossa classe cliente para que tenha uma classe de fallback configurada:

LivroClient

```

package com.acme.avaliacaoservice;

import java.util.ArrayList;
import java.util.List;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.stereotype.Component;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@FeignClient(name = "livros", url = "http://localhost:8080", decode404 = true, fallback = LivroClient.LivroClientFallback.class)
public interface LivroClient {

    @RequestMapping(method = RequestMethod.GET, value = "/livros")
    List<Livro> getLivros();

    @RequestMapping(method = RequestMethod.GET, value = "/livros/{livroId}")
    Livro getLivroPorId(@PathVariable("livroId") Long livroId);

    @Component
    public static class LivroClientFallback implements LivroClient {

        @Override
        public List<Livro> getLivros() {
            return new ArrayList<Livro>();
        }

        @Override
        public Livro getLivroPorId(Long livroId) {
            return new Livro("Desconhecido", "Desconhecido", 0d);
        }
    }
}

```

O último passo é ajustar nosso controller para que identifique quando um livro não foi encontrado.

```

// Código atual omitido

@RestController
@RequestMapping("/avaliacoes")
public class AvaliacoesController {

    // Código atual omitido

    // Novidades aqui
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Avaliacao adicionarAvaliacao(@RequestBody Avaliacao avaliacao) throws IOException {
        logger.info("adicionarAvaliacao: " + avaliacao);
        String tituloLivro = livroClient.getLivroPorId(avaliacao.getLivroId()).getTitulo();
        if (tituloLivro == null) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Livro não existe: " + avaliacao.getLivroId());
        }
        logger.info("Título do livro avaliado: " + tituloLivro);
        return repository.save(avaliacao);
    }
}

```

Exercício opcional

Utilizar o Feign e o Hystrix nas chamadas do lado do livro-service

Hystrix Dashboard

Um bom recurso do Hystrix é a capacidade de monitorar seu status em um painel.

Para ativá-lo, colocaremos uma nova dependência em nosso projeto:

pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
```

Em seguida, o ativamos via anotação `@EnableHystrixDashboard`:

AvaliacaoServiceApplication

```
package com.acme.avaliacaoservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;
import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
@EnableFeignClients
// Novidade aqui
@EnableHystrixDashboard
public class AvaliacaoServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(AvaliacaoServiceApplication.class, args);
    }
}
```

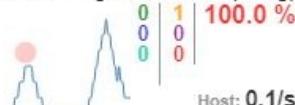
Depois que reiniciarmos o aplicativo, direcionaremos um navegador para <http://localhost:8081/hystrix>, inseriremos o URL de métricas de um "hystrix.stream" (<http://localhost:8081/actuator/hystrix.stream>) e começaremos o monitoramento.

Finalmente, devemos ver algo assim:

Hystrix Stream: <http://localhost:8081/actuator/hystrix.stream>

Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)

LivroClient#getLivroPorId(Long)



Host: 0.1/s

Cluster: 0.1/s

Circuit Closed

Hosts	1	90th	1006ms
Median	1003ms	99th	1044ms
Mean	1006ms	99.5th	1044ms

Thread Pools Sort: [Alphabetical](#) | [Volume](#) |

livros

Host: 0.1/s

Cluster: 0.1/s

Active	0	Max Active	1
Queued	0	Executions	1
Pool Size	10	Queue Size	5

Monitorar um "hystrix.stream" é algo bom, mas se você tiver que assistir a vários aplicativos habilitados para o Hystrix, ele se tornará inconveniente. Para essa finalidade, o Spring Cloud fornece uma ferramenta chamada Turbine, que pode agregar fluxos para apresentar em um painel Hystrix.

Consul e Feign

Uma das vantagens de se utilizar o Feign é sua fácil integração com serviços de discovery como o Consul e Eureka.

Vamos retirar as URLs fixas em nosso código e utilizar o serviço de discovery para identificar os hosts a serem acionados pelos clientes Feign, para isso, basta um pequeno ajuste:

LivroClient

```
// Novidade aqui
@FeignClient(name = "livro-service", decode404 = true, fallback = LivroClient.LivroClientFallback.class)
public interface LivroClient {
    @RequestMapping(method = RequestMethod.GET, value = "/livros")
    List<Livro> getLivros();

    @RequestMapping(method = RequestMethod.GET, value = "/livros/{livroId}")
    Livro getLivroPorId(@PathVariable("livroId") Long livroId);

    @Component
    public static class LivroClientFallback implements LivroClient {

        @Override
        public List<Livro> getLivros() {
            return new ArrayList<Livro>();
        }

        @Override
        public Livro getLivroPorId(Long livroId) {
            return new Livro("Desconhecido", "Desconhecido", 0d);
        }
    }
}
```

Fontes

- <https://www.baeldung.com/spring-cloud-netflix-hystrix>
- <https://medium.com/@thiagoloureiro/conhecendo-o-hystrix-2c72e2cbe5ef>

Testes

Uma confusão bastante comum na comunidade de desenvolvimento é justamente sobre qual nome dar para o tipo de teste. **Esse é um teste de unidade, integração ou sistema?** Apesar de parecer uma discussão boba, é importante que desenvolvedores usem os mesmos termos para se comunicar; isso facilita e acelera o entendimento.

Um **teste de unidade** é aquele que testa uma única unidade do sistema. Ele a testa de maneira isolada, geralmente simulando as prováveis dependências que aquela unidade tem. Em sistemas orientados a objetos, é comum que a unidade seja uma classe. Ou seja, quando queremos escrever testes de unidade para a classe Pedido, essa bateria de testes testará o funcionamento da classe Pedido, isolada, sem interações com outras classes.

Um **teste de integração** é aquele que testa a integração entre duas partes do seu sistema. Os testes que você escreve para a sua classe *PedidoDao*, por exemplo, onde seu teste vai até o banco de dados, é um teste de integração. Afinal, você está testando a integração do seu sistema com o sistema externo, que é o banco de dados. Testes que garantem que suas classes comunicam-se bem com serviços web, escrevem arquivos texto, ou mesmo mandam mensagens via socket são considerados testes de integração.

Já um **teste de sistema** garante que o sistema funciona como um todo. Este nível de teste está interessado se o sistema funciona como um todo, com todas as unidades trabalhando juntas. Ele é comumente chamado de teste de caixa preta, já que o sistema é testado “com tudo ligado”: banco de dados, serviços web, batch jobs, e etc. Os **testes de aceitação**, famosos com a onda ágil, são, no fim, testes de sistema. Testes de aceitação são aqueles onde as equipes ágeis dizem se uma determinada funcionalidade está “aceita” ou não.

Independente do nível do teste, todos eles têm vantagens e desvantagens. Um teste de unidade, por exemplo, é bastante fácil de ser e roda muito rápido; mas não é um teste que simula bem o mundo real. Por outro lado, um teste de sistema faz uma simulação bastante real, mas é muito mais difícil de ser escrito, dá mais trabalho de manutenção e leva mais tempo para executar.

Mas qual nível de teste usar então? A ideia é que você escolha o nível de teste certo para aquele problema. Uma classe de negócio pode ser testada de maneira isolada; já um DAO precisa ser testado junto a um banco de dados. Lembre-se: o teste deve dar feedback rico; um teste que nunca quebra não serve de nada.

Introdução

Abordaremos aqui o suporte do Spring para testes de integração e práticas recomendadas para testes de unidade. O time do Spring defende o desenvolvimento orientado a testes (TDD). A equipe do Spring descobriu que o uso correto da inversão de controle (IoC) certamente torna os testes de unidade e integração mais fáceis (na medida em que a presença de métodos setter e construtores apropriados em classes facilita a conexão em um teste sem ter que configurar registros de localizadores de serviço e estruturas semelhantes).

O teste é uma parte integrante do desenvolvimento de software corporativo. Iremos enfocar o valor agregado pelo princípio IoC ao teste de unidade e aos benefícios do suporte do Spring Framework para testes de integração.

Testes Unitários no Spring

Ainjeção de dependência deve tornar seu código menos dependente do contêiner do que seria com o desenvolvimento tradicional do Java EE. Os POJOs que compõem seu aplicativo devem ser testáveis nos testes JUnit ou TestNG, com objetos instanciados usando o novo operador, sem o Spring ou qualquer outro contêiner. Você pode usar objetos simulados (em conjunto com outras valiosas técnicas de teste) para testar seu código isoladamente. Se você seguir as recomendações de arquitetura do Spring, a estratificação e a componentização limpas resultantes da base de código facilitarão o teste da unidade. Por exemplo, você pode testar objetos da camada de serviço por meio do stub ou do escaneamento de interfaces DAO ou de repositório, sem precisar acessar dados persistentes durante a execução de testes de unidade.

Os testes unitários verdadeiros geralmente são executados com extrema rapidez, pois não há infraestrutura de tempo de execução para serem configurados. Enfatizar os verdadeiros testes unitários como parte de sua metodologia de desenvolvimento pode aumentar sua produtividade.

Teste de integração no Spring

É importante poder realizar alguns testes de integração sem exigir a implantação no servidor de aplicativos ou a conexão com outra infraestrutura corporativa. Isso permite testar coisas como:

- Aligação correta dos contextos de contêiner do Spring IoC.
- Acesso de dados usando JDBC ou uma ferramenta ORM. Isso pode incluir coisas como a correção de instruções SQL, consultas Hibernate, mapeamentos de entidades JPA e assim por diante.

O Spring Framework fornece suporte de primeira classe para testes de integração no módulo `spring-test`. O nome do arquivo JAR real pode incluir a versão de liberação e também pode estar no formato longo `org.springframework.test`. Essa biblioteca inclui o pacote `org.springframework.test`, que contém classes valiosas para teste de integração com um contêiner Spring. Esse teste não depende de um servidor de aplicativos ou outro ambiente de implementação. Esses testes são mais lentos para serem executados do que os testes de unidade, mas muito mais rápidos que os testes Selenium equivalentes ou testes remotos que dependem da implantação em um servidor de aplicativos.

O suporte a testes de unidade e integração é fornecido na forma de Spring TestContext Framework orientada por anotações. O framework TestContext é agnóstico da estrutura de teste em uso, que permite a instrumentação de testes em vários ambientes, incluindo JUnit, TestNG e outros.

Objetivos do Teste de Integração

O suporte de testes de integração do Spring tem os seguintes objetivos principais:

- Gerenciar o armazenamento em cache do contêiner do Spring IoC entre os testes;
- Fornecer Injeção de Dependência de instâncias de fixtures de teste;
- Fornecer gerenciamento de transações apropriado para testes de integração;
- Fornecer classes base específicas de Spring que auxiliem os desenvolvedores a escrever testes de integração;

Preparando nosso projeto para testes unitários

Primeiramente, vamos adequar nosso projeto, criando uma classe de serviços

LivrosController

```
package com.acme.livroservice;

import java.util.List;
import java.util.Optional;
import java.util.concurrent.TimeUnit;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
```

```

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

@RestController
@RequestMapping("/livros")
public class LivrosController {

    Logger logger = LoggerFactory.getLogger(LivrosController.class);

    private final RabbitTemplate rabbitTemplate;

    private final LivroService livroService;

    LivrosController(RabbitTemplate rabbitTemplate, LivroService livroService) {
        this.rabbitTemplate = rabbitTemplate;
        this.livroService = livroService;
    }

    @GetMapping
    public List<Livro> getLivros(@RequestParam("autor") Optional<String> autor,
                                  @RequestParam("titulo") Optional<String> titulo) {
        logger.info("getLivros - autor: " + autor.orElse("Não informado") + " titulo: " + titulo.orElse("Não informado"));

        return livroService.getLivros(autor, titulo);
    }

    @GetMapping("/{id}")
    @Cacheable(value = "livros", key = "#id")
    public Livro getLivroPorId(@PathVariable Long id) {
        logger.info("getLivroPorId: " + id);
        return livroService.getLivroPorId(id);
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Livro adicionarLivro(@RequestBody Livro livro) {
        logger.info("adicionarLivro: " + livro);
        return livroService.adicionarLivro(livro);
    }

    @PostMapping("/demorado")
    @ResponseStatus(HttpStatus.CREATED)
    public Livro adicionarLivroDemorado(@RequestBody Livro livro) throws InterruptedException {
        logger.info("adicionarLivroDemorado iniciou: " + livro);
        TimeUnit.SECONDS.sleep(3);
        Livro livroSalvo = livroService.adicionarLivro(livro);
        logger.info("adicionarLivroDemorado terminou: " + livroSalvo);
        return livroSalvo;
    }

    @PostMapping("/assincrono")
    @ResponseStatus(HttpStatus.CREATED)
    public void adicionarLivroAssincrono(@RequestBody Livro livro) throws InterruptedException {
        logger.info("adicionarLivroAssincrono iniciou: " + livro);
        rabbitTemplate.convertAndSend(LivroServiceApplication.TOPIC_EXCHANGE_NAME, LivroServiceApplication.ROUTING_KEY,
                                      livro);
    }

    @PutMapping("/{id}")
    @CachePut(value = "livros", key = "#livro.id")
    public Livro atualizarLivro(@RequestBody Livro livro, @PathVariable Long id) {
        logger.info("atualizarLivro: " + livro + " id: " + id);
        return livroService.atualizarLivro(livro, id);
    }

    @DeleteMapping("/{id}")
    @CacheEvict(value = "livros", allEntries = true)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void excluirLivro(@PathVariable Long id) {
        logger.info("excluirLivro: " + id);
        livroService.excluirLivro(id);
    }
}

```

```

    }
}
```

LivroService

```

package com.acme.livroservice;

import java.util.List;
import java.util.Optional;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.jpa.domain.Specification;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Service;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.ResourceAccessException;
import org.springframework.web.server.ResponseStatusException;

@Service
public class LivroService {
    Logger logger = LoggerFactory.getLogger(LivroService.class);

    private final LivroRepository repository;

    private final AvaliacaoService avaliacaoService;

    LivroService(LivroRepository repository, AvaliacaoService avaliacaoService) {
        this.repository = repository;
        this.avaliacaoService = avaliacaoService;
    }

    public List<Livro> getLivros(Optional<String> autor, Optional<String> titulo) {
        logger.info("getLivros - autor: " + autor.orElse("Não informado") + " titulo: " + titulo.orElse("Não informado"));

        if (autor.isPresent()) {
            return repository.findAll(LivroRepository.autorContem(autor.get()));
        } else if (titulo.isPresent()) {
            return repository.findAll(LivroRepository.tituloContem(titulo.get()));
        } else if (autor.isPresent() && titulo.isPresent()) {
            return repository.findAll(Specification.where(LivroRepository.autorContem(autor.get()))
                .and(LivroRepository.tituloContem(titulo.get())));
        } else {
            return repository.findAll();
        }
    }

    public Livro getLivroPorId(Long id) {
        logger.info("getLivroPorId: " + id);
        return repository.findById(id)
            .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
    }

    public Livro adicionarLivro(@RequestBody Livro livro) {
        logger.info("adicionarLivro: " + livro);
        return repository.save(livro);
    }

    public Livro atualizarLivro(Livro livro, Long id) {
        logger.info("atualizarLivro: " + livro + " id: " + id);
        return repository.findById(id).map(livroSalvo -> {
            livroSalvo.setAutor(livro.getAutor());
            livroSalvo.setTitulo(livro.getTitulo());
            livroSalvo.setPreco(livro.getPreco());
            return repository.save(livroSalvo);
        }).orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Livro não encontrado: " + id));
    }

    public void excluirLivro(Long livroId) {
```

```

logger.info("excluirLivro: " + livroId);

try {
    this.avaliacaoService.apagarAvaliacoesPorLivroId(livroId);
    logger.info("Avaliações vinculadas excluídas com sucesso");
} catch (ResourceAccessException | HttpClientErrorException ex) {
    logger.error("Ocorreu um erro na comunicação com o serviço de avaliações", ex);
    throw new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE,
        "Ocorreu um erro não esperado na comunicação com o serviço de livros: " + ex.getMessage());
}

repository.deleteById(livroId);
}
}

```

AvaliacaoService

```

package com.acme.livroservice;

import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class AvaliacaoService {

    private final RestTemplate restTemplate;

    public AvaliacaoService(RestTemplateBuilder restTemplateBuilder) {
        restTemplate = restTemplateBuilder.build();
    }

    public void apagarAvaliacoesPorLivroId(Long livroId) {
        String avaliacaoResourceUrl = "http://localhost:8081/avaliacoes/livro/";
        restTemplate.delete(avaliacaoResourceUrl + livroId);
    }
}

```

Outra alteração importante é implementar um método `equals` na classe `Livro`:

```

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Livro other = (Livro) obj;
    if (autor == null) {
        if (other.autor != null)
            return false;
    } else if (!autor.equals(other.autor))
        return false;
    if (id == null) {
        if (other.id != null)
            return false;
    } else if (!id.equals(other.id))
        return false;
    if (preco == null) {
        if (other.preco != null)
            return false;
    } else if (!preco.equals(other.preco))
        return false;
    if (titulo == null) {
        if (other.titulo != null)
            return false;
    } else if (!titulo.equals(other.titulo))
        return false;
    return true;
}

```

Testes Unitários com Spring Boot

O boot Spring oferece uma ótima classe para facilitar o teste: anotação `@SpringBootTest`.

Esta anotação pode ser especificada em uma classe de teste que executa testes baseados no Spring Boot. Fornece os seguintes recursos além do Spring TestContext Framework regular:

- Usa `SpringBootContextLoader` como o `ContextLoader` padrão quando nenhum `@ContextConfiguration(loader=...)` específico é definido;
- Procura automaticamente por um `@SpringBootConfiguration` quando `@Configuration` aninhado não é usado, e nenhuma classe explícita é especificada;
- Permite que propriedades de ambiente personalizadas sejam definidas usando o atributo `properties`;
- Fornece suporte para diferentes modos de ambiente da Web, incluindo a capacidade de iniciar um servidor da Web totalmente em execução, ouvindo em uma porta definida ou aleatória;
- Registra um bean `TestRestTemplate` e/ou `WebTestClient` para uso em testes da web que estão usando um servidor da Web totalmente em execução;

O primeiro passo é nos certificarmos que a dependência `spring-boot-starter-test` está presente no `pom.xml` de nosso projeto.

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

```

Nós basicamente temos dois componentes para testar aqui: `LivroService` e `LivrosController`.

Testes Unitários de `LivroService`

LivroServiceUnitTest

```

package com.acme.livroservice;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.Mockito.times;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class LivroServiceUnitTest {
    @Autowired
    private LivroService livroService;

    @MockBean
    private LivroRepository repository;

    @MockBean
    private AvaliacaoService avaliacaoService;

    public void criaListaLivrosVazia() {
        when(repository.findAll()).thenReturn(new ArrayList<Livro>());
    }

    public void criaListaLivrosDefault() {
        List<Livro> livros = new ArrayList<Livro>();
        livros.add(new Livro(1L, "Miguel de Cervantes", "Don Quixote", 144.0));
        livros.add(new Livro(2L, "J. R. R. Tolkien", "O Senhor dos Anéis", 123.0));
        livros.add(new Livro(3L, "Antoine de Saint-Exupéry", "O Pequeno Príncipe", 152.0));
        livros.add(new Livro(4L, "Charles Dickens", "Um Conto de Duas Cidades", 35.0));

        when(repository.findAll()).thenReturn(livros);
    }

    @Test
    public void getLivrosVazio() {
        criaListaLivrosVazia();
        List<Livro> livros = livroService.getLivros(Optional.empty(), Optional.empty());
        assertThat(livros).isEmpty();
    }

    @Test
    public void getLivrosComLivros() {
        criaListaLivrosDefault();
        List<Livro> livros = livroService.getLivros(Optional.empty(), Optional.empty());
        assertThat(livros.size()).isEqualTo(4);
    }

    @Test
    public void getLivroPorId() {
        when(repository.findById(1L))
            .thenReturn(Optional.of(new Livro(1L, "Miguel de Cervantes", "Don Quixote", 144.0)));
        Livro livro = livroService.getLivroPorId(1L);
        assertThat(livro).isEqualTo(new Livro(1L, "Miguel de Cervantes", "Don Quixote", 144.0));
    }

    @Test
    public void adicionarLivro() {
        when(repository.save(new Livro("Miguel de Cervantes", "Don Quixote", 144.0)))
    }
}

```

```

        .thenReturn(new Livro(1L, "Miguel de Cervantes", "Don Quixote", 144.0));
    Livro livro = livroService.adicionarLivro(new Livro("Miguel de Cervantes", "Don Quixote", 144.0));
    assertThat(livro).isEqualTo(new Livro(1L, "Miguel de Cervantes", "Don Quixote", 144.0));
}

@Test
public void atualizarLivro() {
    when(repository.findById(1L))
        .thenReturn(Optional.of(new Livro(1L, "Miguel de Cervantes", "Don Quixote", 144.0)));
    when(repository.save(new Livro(1L, "Foo Bar", "Don Quixote", 144.0)))
        .thenReturn(new Livro(1L, "Foo Bar", "Don Quixote", 144.0));
    Livro livro = livroService.atualizarLivro(new Livro(1L, "Foo Bar", "Don Quixote", 144.0), 1L);
    assertThat(livro).isEqualTo(new Livro(1L, "Foo Bar", "Don Quixote", 144.0));
}

// É correto testar o funcionamento interno da função?
@Test
public void excluirLivro() {
    livroService.excluirLivro(1L);
    verify(avaliacaoService, times(1)).apagarAvaliacoesPorLivroId(1L);
    verify(repository, times(1)).deleteById(1L);
}
}

```

Testes Unitários de LivrosController

LivrosControllerUnitTest

```

package com.acme.livroservice;

import static org.hamcrest.Matchers.is;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.put;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.delete;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.setup.MockMvcBuilders.standaloneSetup;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import com.fasterxml.jackson.databind.ObjectMapper;

@RunWith(SpringRunner.class)
@SpringBootTest
public class LivrosControllerUnitTest {

    MockMvc mockMvc;

    @Autowired
    private LivrosController livrosController;

    @MockBean
    private RabbitTemplate rabbitTemplate;

    @MockBean

```

```

private LivroService livroService;

private List<Livro> livros;

@Before
public void setup() throws Exception {
    this.mockMvc = standaloneSetup(this.livrosController).build();

    livros = new ArrayList<Livro>();
    livros.add(new Livro(1L, "Miguel de Cervantes", "Don Quixote", 144.0));
    livros.add(new Livro(2L, "J. R. R. Tolkien", "O Senhor dos Anéis", 123.0));
    livros.add(new Livro(3L, "Antoine de Saint-Exupéry", "O Pequeno Príncipe", 152.0));
    livros.add(new Livro(4L, "Charles Dickens", "Um Conto de Duas Cidades", 35.0));
}

@Test
public void getLivros() throws Exception {
    when(livroService.getLivros(Optional.empty(), Optional.empty())).thenReturn(livros);
    mockMvc.perform(get("/livros").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
        .andExpect(jsonPath("$.length()", is(livros.size())))
        .andExpect(jsonPath("$.[0].titulo", is("Don Quixote")))
        .andExpect(jsonPath("$.[1].titulo", is("O Senhor dos Anéis")))
        .andExpect(jsonPath("$.[2].titulo", is("O Pequeno Príncipe")))
        .andExpect(jsonPath("$.[3].titulo", is("Um Conto de Duas Cidades")));
}

@Test
public void getLivro() throws Exception {
    when(livroService.getLivroPorId(1L)).thenReturn(new Livro(1L, "Miguel de Cervantes", "Don Quixote", 144.0));
    mockMvc.perform(get("/livros/1").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
        .andExpect(jsonPath("$.titulo", is("Don Quixote")));
}

@Test
public void adicionarLivro() throws Exception {
    when(livroService.adicionarLivro(new Livro("Miguel de Cervantes", "Don Quixote", 144.0)))
        .thenReturn(new Livro(1L, "Miguel de Cervantes", "Don Quixote", 144.0));
    mockMvc.perform(post("/livros").contentType(MediaType.APPLICATION_JSON)
        .content(asJsonString(new Livro("Miguel de Cervantes", "Don Quixote", 144.0)))
        .accept(MediaType.APPLICATION_JSON)).andExpect(status().isCreated())
        .andExpect(jsonPath("$.titulo", is("Don Quixote")));
}

public static String asJsonString(final Object obj) {
    try {
        final ObjectMapper mapper = new ObjectMapper();
        final String jsonContent = mapper.writeValueAsString(obj);
        return jsonContent;
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

@Test
public void atualizarLivro() throws Exception {
    when(livroService.atualizarLivro(new Livro(1L, "Foo Bar", "Don Quixote", 144.0), 1L))
        .thenReturn(new Livro(1L, "Foo Bar", "Don Quixote", 144.0));
    mockMvc.perform(put("/livros/1").contentType(MediaType.APPLICATION_JSON)
        .content(asJsonString(new Livro(1L, "Foo Bar", "Don Quixote", 144.0)))
        .accept(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
        .andExpect(jsonPath("$.autor", is("Foo Bar")));
}

@Test
public void excluirLivro() throws Exception {
    mockMvc.perform(delete("/livros/1")).andExpect(status().isNoContent());
}
}

```

Testes de Integração com Spring Boot

Para os testes de integração, queremos verificar nossos principais componentes com a comunicação downstream.

Testes Unitários de LivroService

Este teste é muito simples. Não precisamos mocar nada.

LivroServiceIntegrationTest

```
package com.acme.livroservice;

import static org.assertj.core.api.Assertions.assertThat;

import java.util.List;
import java.util.Optional;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class LivroServiceIntegrationTest {
    @Autowired
    private LivroService livroService;

    @Test
    public void getLivros() {
        List<Livro> livros = livroService.getLivros(Optional.empty(), Optional.empty());
        assertThat(livros).isNotEmpty();
    }
}
```

Testes Unitários de LivrosController

LivrosControllerIntegrationTest

```

package com.acme.livroservice;

import static org.hamcrest.Matchers.is;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.setup.MockMvcBuilders.standaloneSetup;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

@RunWith(SpringRunner.class)
@SpringBootTest
public class LivrosControllerIntegrationTest {

    MockMvc mockMvc;

    @Autowired
    private LivrosController livrosController;

    @Before
    public void setup() throws Exception {
        this.mockMvc = standaloneSetup(this.livrosController).build();
    }

    @Test
    public void getLivros() throws Exception {
        mockMvc.perform(get("/livros").contentType(MediaType.APPLICATION_JSON)).andExpect(status().isOk())
            .andExpect(jsonPath("$.titles", is("Don Quixote")))
            .andExpect(jsonPath("$.titles", is("O Senhor dos Anéis")))
            .andExpect(jsonPath("$.titles", is("O Pequeno Príncipe")))
            .andExpect(jsonPath("$.titles", is("Um Conto de Duas Cidades")));
    }

}

```

Fontes

- <http://blog.caelum.com.br/unidade-integracao-ou-sistema-qual-teste-fazer/>
- <https://www.baeldung.com/spring-boot-testing>
- <https://dzone.com/articles/unit-and-integration-tests-in-spring-boot-2>
- <https://thepracticaldeveloper.com/2017/07/31/guide-spring-boot-controller-tests/>