

Curso Microservices com Spring Cloud

Curso FJ-33



Conhecendo o Caelum Eats

Peça sua comida com o Caelum Eats

Nesse curso, usaremos como exemplo o Caelum Eats: uma aplicação de entrega de comida nos moldes de soluções conhecidas no mercado.

Há 3 perfis de usuário:

- o cliente, que efetua um pedido
- o dono do restaurante, que mantém os dados do restaurante e muda os status de pedidos pendentes
- o administrador do Caelum Eats, que mantém os dados básicos do sistema e aprova novos restaurantes

Funcionalidades

Cliente

O intuito do cliente é efetuar um pedido, que é um processo de várias etapas. No Caelum Eats, o cliente não precisa fazer login.

Ao acessar a página principal do Caelum Eats, o cliente deve digitar o seu CEP.

Buscar

71503510

70238-500

70238500

71503-510

Depois de digitado o CEP, o Caelum Eats retorna uma lista com os restaurantes mais próximos. Entre as informações mostradas em cada item da lista, está a distância do restaurante ao CEP.

O cliente pode filtrar por tipo de cozinha, se desejar. Então, deve escolher algum restaurante.

Os dados iniciais do Caelum Eats vêm apenas com um restaurante: o Long Fu, de comida chinesa.

Observação: a implementação não calcula de fato a distância do CEP aos restaurantes. O valor exibido é apenas um número randômico.

Tipos de Cozinha

[Árabe](#)[Baiana](#)[Chinesa](#)[Hambúrguer](#)[Italiana](#)[Japonesa](#)[Lanches](#)[Mexicana](#)[Mineira](#)[Variada](#)

Restaurantes próximos a 70238-500

Long Fu

★ 4,1

Chinesa • 40 min - 25

min • 2,2 km

R\$ 6,00

[Escolher](#)**Pamonheria****Goiana**

★ 4,0

Variada • 10 min - 80

min • 1,9 km

R\$ 10,00

[Escolher](#)**Árabe Arabaico**

★ 5,0

Árabe • 9,4 km

R\$ 10,00

[Escolher](#)

Depois de escolhido um restaurante, o cliente vê o cardápio.

Também são exibidas outras informações do restaurante, como a média das avaliações, a descrição, os tempos de espera mínimo e máximo e a distância do CEP digitado pelo cliente.

Há também uma aba de avaliações, em que o cliente pode ver as notas e comentários de pedidos anteriores.

Long Fu

★ 4,1

Chinesa • 40 min - 25 min • 11,7 km

O melhor da China aqui do seu lado.

Cardápio

Avaliações

ENTRADAS

Gyoza Bovino - 6 unidades

Massa fina cozida a vapor recheada com carne temperada com gengibre

R\$ 23,50

Escolhe

Ao escolher um item do cardápio, o cliente deve escolher uma quantidade. É possível fazer observações de preparo.

Gyoza Bovino - 6 unidades

Quantidade

1

Observação

Sem sal

Adicionar

Cancelar

A cada item do cardápio escolhido, o resumo do pedido é atualizado.

Pedido

1x Gyoza Bovino - 6
unidades R\$ 23,50

Sem sal

[Editar](#) [Remover](#)

Taxa de entrega: R\$ 6,00

Total: R\$ 29,50

[Fazer Pedido](#)

Ao clicar no botão "Fazer Pedido", o cliente deve digitar os seus dados pessoais (nome, CPF, email e telefone) e os dados de entrega (CEP, endereço e complemento).

Informações de Entrega

Dados pessoais

Nome:

Joaquim

CPF:

890.898.980-89

Email:

joaquim@cmail.com.br

Telefone:

(61) 9 8123-8799

Local de entrega

CEP:

70238-500

Então, o cliente informa os dados de pagamento. Por enquanto, o Caelum Eats só aceita cartões.

Resumo do pedido

1x Gyoza Bovino - 6 unidades R\$ 23,50

Sem sal

Taxa de entrega:
R\$ 6,00

Total: R\$ 29,50

Dados do pagamento

Forma de pagamento

Ticket Restaurante

Nome no cartão

JOAQUIM

Número do cartão

8989 8098 0989 0890

Data de expiração

February 2022

Código de Segurança

909

Criar pagamento

No próximo passo, o cliente pode confirmar ou cancelar o pagamento criado no anteriormente.

Cartão

8989 XXXX XXXX XXXX

Valor

R\$ 29,50

Confirmar pagamento

Cancelar pagamento

Se o pagamento for confirmado, o pedido será realizado e aparecerá como pedido pendente no restaurante!

Então, o cliente pode acompanhar a situação de seu pedido. Para ver se houve alguma mudança, a página deve ser recarregada.

Acompanhe o Pedido

Pago

Quando o restaurante avisar o Caelum Eats que o pedido foi entregue, o cliente poderá deixar sua avaliação com comentários. A nota da avaliação influenciará na média do restaurante.

Acompanhe o Pedido

Entregue

Avalie o pedido

★★★☆☆

Tava bom, mas veio com sal. :(

Avaliar

Dono do Restaurante

O dono de um restaurante deve efetuar o login para manipular as informações de seu restaurante.

As informações de login do restaurante pré-cadastrado, o Long Fu, são as seguintes:

- usuário: longfu
- senha: 123456

Usuário:

longfu

Senha:

.....

Logar

Depois do login efetuado, o dono do restaurante terá acesso ao menu.



Uma das funcionalidades permite que o dono do restaurante atualize o cadastro, manipulando informações do restaurante como o nome, CNPJ, CEP, endereço, tipo de cozinha, taxa de entrega e tempos mínimo e máximo de entrega.

Além disso, o dono do restaurante pode escolher quais formas de pagamento são aceitas, o horário de funcionamento e cadastrar o cardápio do restaurante.

Endereço:

SQS QUADRA 404 BLOCO D LOJA 17

Taxa de entrega (em R\$)

6

Tempo de entrega mínimo (em minutos):

40

Tempo de entrega máximo (em minutos):

25

Atualizar

O dono do restaurante também pode acessar os pedidos pendentes, que ainda não foram entregues. Cada mudança na situação dos pedidos pode ser informada por meio dessa tela.

Pago

Cliente: Joaquim

Tel: (61) 9 8123-8799

Endereço: CLS 404 BL E AP 306

1x Gyoza Bovino - 6 unidades

Sem sal

Confirmar

O dono de um novo restaurante, que ainda não faz parte do Caelum Eats,

pode registrar-se clicando em "Cadastre seu Restaurante". Depois de cadastrar um usuário e a respectiva senha, poderá preencher as informações do novo restaurante.

O novo restaurante ainda não aparecerá para os usuários. É necessária a aprovação do restaurante pelo administrador do Caelum Eats.

Usuário:

caxambu

Senha:

.....

Confirmação da Senha:

.....

Registrar usuário

Copyright © 2019 - Caelum Eats - Todos os direitos reservados [Cadastre seu Restaurante](#)

Administrador

O administrador do Caelum Eats só terá acesso às suas funcionalidades depois de efetuar o login.

Há um administrador pré-cadastrado, com as seguintes credenciais:

- usuário: admin
- senha: 123456

Não há uma tela de cadastro de novos administradores. Por enquanto, isso deve ser efetuado diretamente no Banco de Dados. Esse cadastro é uma das funcionalidades pendentes!

Usuário:

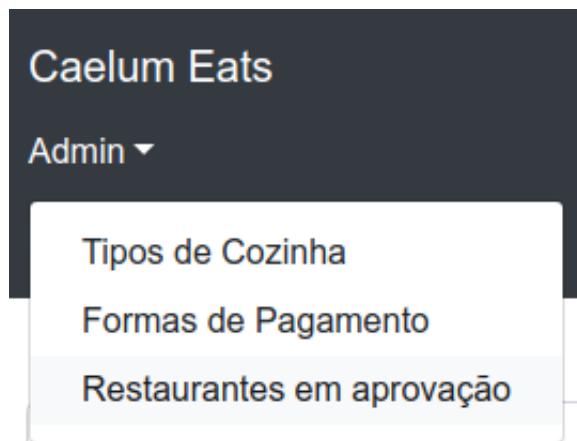
admin

Senha:

.....

Logar

Depois do login efetuado, o administrador verá o menu.



Somente o administrador, depois de logado, pode manter o cadastro dos tipos de cozinha disponíveis no Caelum Eats.

Tipos de Cozinha

[Adicionar](#)

Nome

Árabe

[Editar](#)

[Remover](#)

Baiana

[Editar](#)

[Remover](#)

Chinesa

[Editar](#)

[Remover](#)

Hambúrguer

[Editar](#)

[Remover](#)

Italiana

[Editar](#)

[Remover](#)

Outra funcionalidade disponível apenas do administrador é o cadastro das formas de pagamento que podem ser escolhidas no cadastro de restaurantes.

Formas de Pagamento

[Adicionar](#)

Nome	Tipo		
Alelo	Vale Refeição	Editar	Remover
Amex	Cartão de Crédito	Editar	Remover
MasterCard	Cartão de Crédito	Editar	Remover
MasterCard Maestro	Cartão de Débito	Editar	Remover
Ticket Restaurante	Vale Refeição	Editar	Remover

Também é tarefa do administrador do Caelum Eats revisar o cadastro de novos restaurantes e aprová-los.

Restaurantes em aprovação

Caxambu

Aprovar

Detalhar

Detalhes do restaurante

Nome

Caxambu

Tipo de cozinha

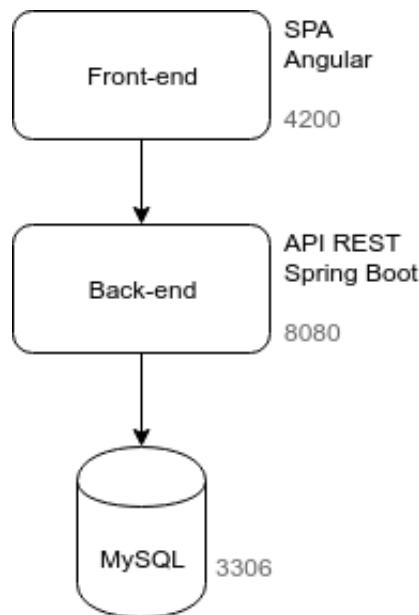
Mineira

CNPJ

89.898.989/8989-89

Descrição

A Arquitetura do Caelum Eats



Back-end

O back-end do Caelum Eats provê uma API REST. A porta usada é a 8080.

O Banco de Dados utilizado é o MySQL, na versão 5.6 e executado na porta 3306.

É implementado com as seguintes tecnologias:

- Spring Boot
- Spring Boot Web
- Spring Boot Validation
- Spring Data JPA
- MySQL Connector/J
- Flyway DB, para migrations
- Lombok, para um Java menos verboso
- Spring Security
- jjwt, para gerar e validar tokens JWT
- Spring Boot Actuator

As migrations do Flyway DB, que ficam no diretório

`src/main/resources/db/migration`, além de criar a estrutura das tabelas, já popula o BD com dados iniciais.

Front-end

O front-end do Caelum Eats é uma SPA (Single Page Application), implementada em Angular 7. A porta usada em desenvolvimento é a 4200.

Para a folha de estilos, é utilizado o Bootstrap 4.

São utilizados alguns componentes open-source:

- ngx-toastr
- angular2-text-mask
- ng-bootstrap

Exercício: Executando o back-end

1. Clone o projeto do back-end para seu Desktop com os seguintes comandos:

```
cd ~/Desktop
```

```
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-mc
```

2. Abra o Eclipse, definindo como workspace /home/<usuario-do-curso>/workspace-monolito. Troque <usuario-do-curso> pelo login utilizado no curso.
3. No Eclipse, acesse *File > Import > Existing Maven Projects* e clique em *Next*. Em *Root Directory*, aponte para o diretório clonado anteriormente.
4. Acesse a classe `EatsApplication` e execute com *CTRL+F11*. O banco de dados será criado automaticamente e alguns dados serão populados.
5. Teste a URL `http://localhost:8080/restaurantes/1` pelo navegador e verifique se um JSON com os dados de um restaurante foi retornado.
6. Analise o código. Veja:
 - as entidades de negócio
 - os recursos e suas respectivas URLs
 - os serviços e suas funcionalidades.

Exercício: Executando o front-end

1. Baixe para o Desktop o projeto do front-end, usando o Git, com os comandos:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-ui
```

2. Abra um Terminal e digite:

```
cd ~/Desktop/fj33-eats-ui
```

3. Instale as dependências do front-end com o comando:
4. Execute a aplicação com o comando:
5. Abra um navegador e teste a URL: `http://localhost:4200`. Explore o projeto, fazendo um pedido, confirmando um pedido efetuado, cadastrando um novo restaurante e aprovando-o. Em caso de dúvidas, peça ajuda ao instrutor.

Um negócio em expansão

No Caelum Eats, a entrega é por conta do restaurantes. Porém, está no *road map* do produto ter entregas por meio de terceiros, como motoboys, ou por funcionários do próprio Caelum Eats.

Atualmente, só são aceitos cartões de débito, crédito e vale refeição. Entre as ideias estão aceitar o pagamento em dinheiro e em formas de pagamentos inovadoras como criptomoedas, soluções de pagamento online como Google Pay e Apple Pay e pagamento com QR Code.

Entre especialistas de negócio, desenvolvedores e operações, a equipe passou a ter algumas dezenas de pessoas, o que complica incrivelmente a comunicação.

Os desenvolvedores passaram a reclamar do código, dizendo que é difícil de entender e de encontrar onde devem ser implementadas manutenções, correções e novas funcionalidades.

Há ainda problemas de performance, especialmente no cálculo dos restaurantes mais próximos ao CEP informado por um cliente. Essa degradação da performance acaba afetando todas as outras partes da aplicação.

Será que esses problemas impedirão a Caelum Eats de expandir os negócios?

Decompondo o monólito

Da bagunça a camadas

Qual é a emoção que a palavra **monólito** traz pra você?

Muitos desenvolvedores associam a palavra monólito a código mal feito, sem estrutura aparente, com muito código repetido e com dados compartilhados entre partes pouco relacionadas. É o que comumente é chamado de código **Spaghetti** ou, [Big Ball of Mud](#) (FOOTE; YODER, 1999).

Porém, é possível criar monólitos bem estruturados. Uma maneira comum de organizar um monólito é usar camadas: cada camada provê um serviço para a camada de cima e é um cliente das camadas de baixo.

Usualmente, o código de uma aplicação é estruturado em 3 camadas:

- *Apresentação*: responsável por prover serviços ao front-end. Em alguns casos, é feita a renderização das telas da UI, como ao utilizar JSPs.
- *Negócio*: responsável pelos cálculos, fluxos e regras de negócio.
- *Persistência*: responsável pelo acesso aos dados armazenados, geralmente, em um Banco de Dados.

Uma maneira de representar essas camadas em um código Java é utilizar pacotes, o que é conhecido como *Package by Layer*.

Na verdade, é preciso distinguir dois tipos de camadas que influenciam a arquitetura do software: as camadas físicas e as camadas lógicas.

Uma camada física, ou *tier* em inglês, descreve a estrutura de implantação do software, ou seja, as máquinas utilizadas.

O que descrevemos no texto anterior é o conceito de camada lógica. Em inglês, a camada lógica é chamada de *layer*. Trata de agrupar o

código que corresponde às camadas descritas anteriormente.

Camadas no Caelum Eats

O código de back-end do Caelum Eats está organizado em camadas (layers). Podemos observar isso estudando a estrutura de pacotes do código Java:

```
eats
└── controller
└── dto
└── exception
└── model
└── repository
└── service
```

Bem organizado, não é mesmo?

Porém, quando a aplicação começa a crescer, o código passa a ficar difícil de entender. Centenas de Controllers juntos, centenas de Repositories misturados. Passa a ser difícil encontrar o código que precisa ser alterado.

A Arquitetura que Grita

Reflita um pouco: dos projetos implementados com a plataforma Java em que você trabalhou, quantos seguiam uma variação da estrutura Package by Layer que estudamos anteriormente? Provavelmente, a maioria!

Veja a estrutura de diretórios abaixo:

```
.
└── assets
└── controllers
└── helpers
└── mailers
```

```
└── models  
└── views
```

Os diretórios anteriores são a estrutura padrão de um framework de aplicações Web muito influente: o Ruby On Rails.

Perceba que interessante: a estrutura básica de diretórios é familiar; em alguns casos, até temos um palpite sobre qual o framework utilizado; mas não temos ideia do **domínio** da aplicação. Qual é o problema que está sendo resolvido?

No post [Screaming Architecture](#) (MARTIN, 2011), Robert "Uncle Bob" Martin diz que a partir das plantas de edifícios, conseguimos saber se trata-se de uma casa ou uma biblioteca: a arquitetura "grita" a finalidade da construção.

Na realidade, a construção civil tem diferentes plantas: a elétrica, a hidráulica, a estrutural, etc. Uncle Bob parece referir-se a um tipo específico de planta: a **planta baixa**. É feita por um arquiteto e é mais próxima do cliente. A partir dela são projetadas outras plantas mais técnicas e específicas. É o tipo de planta que encontramos em lançamentos de imóveis.

Para projetos de software, entretanto, é usual que a estrutura básica de diretórios indique qual o framework ou qual a ferramenta de build utilizados. Porém, para Uncle Bob, o framework é um detalhe; o Banco de Dados é um detalhe; a Web é um mecanismo de entrega da UI, um detalhe.

Uncle Bob cita a ideia de Ivar Jacobson, um dos criadores do UML, descrita no livro [Object Oriented Software Engineering: A Use-Case Driven Approach](#) (JACOBSON, 1992), de que a arquitetura de um software deveria ser *centrada nos casos de uso* e não em detalhes técnicos.

Por arquiteturas centradas no domínio

No livro [Clean Architecture](#) (MARTIN, 2017), Uncle Bob define uma abordagem arquitetural que torna a aplicação:

- **independente de frameworks**: um framework não é sua aplicação. A estrutura de diretórios e as restrições do design do nosso código não deveriam ser determinadas por um framework. Frameworks deveriam ser usados apenas como ferramentas para que a aplicação cumpra suas necessidades.
- **independente da UI**: a UI muda mais frequentemente que o resto do sistema. Além disso, é possível termos diferentes UIs para as mesmas regras de negócio.
- **independente de BD**: as regras de negócio não devem depender de um Banco de Dados específico. Devemos possibilitar a troca, de maneira fácil, de Oracle ou SQL Server para MongoDB, CouchDB, Neo4J ou qualquer outro BD.
- **testável**: deve ser possível testar as regras de negócio diretamente, sem a necessidade de usar uma UI, BD ou servidor Web.

Uncle Bob ainda cita, no mesmo livro, outras arquiteturas semelhantes:

- Hexagonal Architecture, ou Ports & Adapters, descrita por Alistair Cockburn.
- DCI (Data, Context and Interaction), descrita por James Coplien, pioneiro dos Design Patterns, e Trygve Reenskaug, criador do MVC.
- BCE (Boundary-Control-Entity), introduzida por Ivar Jacobson no livro mencionado anteriormente.

O curso [Práticas de Design e Arquitetura de código para aplicações Java](#) (FJ-38) explora, a partir de um gerador de ebooks, tópicos como os princípios SOLID de Orientação a Objetos e alguns Design Patterns, até chegar progressivamente a uma Arquitetura Hexagonal.

Software é massa!

Raymond J. Rubey cita uma carta ao editor da revista CROSSTALK (RUBEY, 1992), em que é feita uma brincadeira que classifica qualidade

de código como se fossem massas:

- **Spaghetti**: complicado, difícil de entender e impossível de manter
- **Lasagna**: simples, fácil de entender, estruturado em camadas, mas monolítico; na teoria, é fácil de mudar uma camada, mas não na prática
- **Ravioli**: componentes pequenos e soltos, que contém "nutrientes" para o sistema; qualquer componente pode ser modificado ou trocado sem afetar significativamente os outros componentes

Explorando o domínio

Até um certo tamanho, uma aplicação estruturada em camadas no estilo Package by Layer, é fácil de entender. Novos desenvolvedores entram no projeto sem muitas dificuldades para entender a organização do código, já que há uma certa familiaridade.

Mas há um momento em que é interessante reorganizar os pacotes ao redor do domínio, o que alguns chamam de *Package by Feature*. O intuito é que fique bem clara qual é a área de negócio de cada parte do código.

Mas qual critério usar para decompor o monólito? Uma funcionalidade é um pacote, como sugere o nome *Package by Feature*? Talvez isso seja muito granular.

Uma fonte de *insights* interessantes em como explorar o domínio de uma aplicação é o livro [Domain Driven Design](#), de Eric Evans (EVANS, 2003).

Caçando entidades importantes

Todo domínio tem entidades importantes, que são um ponto de entrada para o negócio e tem várias outras entidades ligadas. Um objeto que representa um pedido, por exemplo, terá informações dos itens do pedido, do cliente, da entrega e do pagamento. É o que é chamado, nos termos do DDD, de *Agregado*.

| AGREGADO

Agrupamento de objetos associados que são tratados como uma unidade para fins de alterações nos dados. Referências externas são restritas a um único membro do AGREGADO, designado como *raiz*. Um conjunto de regras de consistência se aplicada dentro dos limites do AGREGADO.

[Domain Driven Design](#) (EVANS, 2003)

Referências a objetos de fora desse Agregado devem ser feitas por meio do objeto principal, a raiz do Agregado. Uma relação entre um pedido e um restaurante deve ser feita pelo pedido, e não pela entrega ou pelo pagamento do pedido.

Os dados de um Agregado são alterados em conjunto. Por isso, Banco de Dados Orientados a Documentos, como o MongoDB, em que um documento pode ter um grafo de outros objetos conectados, são interessantes para persistir Agregados.

E no Caelum Eats?

Um objeto muito importante é o `Pedido`. Associado a um `Pedido` temos uma lista de `ItemDoPedido` e uma `Entrega`. Uma `Entrega` está associada a um `Cliente`. Cada `Pedido` também pode ter uma `Avaliacao`. Há, possivelmente, um `Pagamento` para cada `Pedido`. E um `Pagamento` tem uma `FormaDePagamento`. Um `Pedido` também está associado a um `Restaurante`.

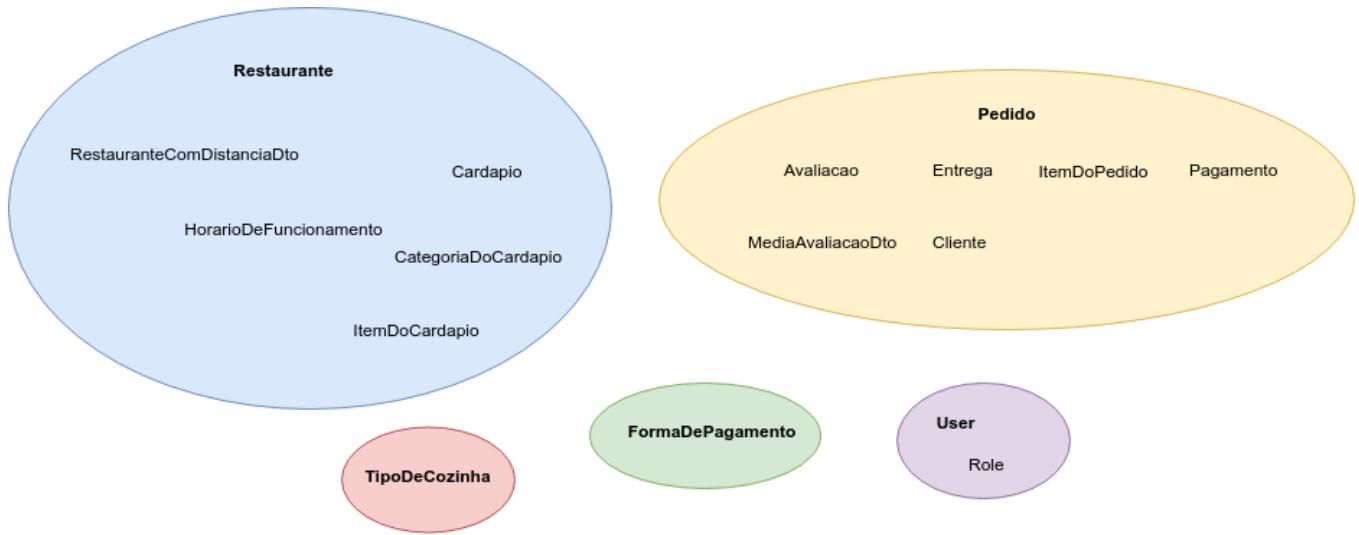
Um `Restaurante` tem seu cadastro mantido por seu dono e é aprovado pelo administrador do Caelum Eats. Um `Restaurante` pode existir sem nenhum pedido, o que acontece logo que foi cadastrado. Um `Restaurante` está relacionado a um `cardapio` que contém uma lista de `CategoriaDoCardapio` que, por sua vez, contém uma lista de `ItemDoCardapio`. Um `Restaurante` possui também uma lista de `HorarioDeFuncionamento` e das `FormaDePagamento` aceitas, assim como um `TipoDeCozinha`.

Um `Restaurante` também possui um `User`, que representa seu dono. O

User também é utilizado pelo administrador do Caelum Eats. Um User tem um ou mais Role.

Uma FormaDePagamento existe independentemente de um Restaurante e os valores possíveis são mantidos pelo administrador. O mesmo ocorre para TipoDeCozinha.

Há algumas classes interessantes como MediaAvaliacoesDto e RestauranteComDistanciaDto, que associam um restaurante à média das notas das avaliações e uma distância a um dado CEP, respectivamente.



Poderíamos alinhar o código do Caelum Eats com o domínio, utilizando os agregados identificados anteriormente.

Das estruturas de comunicação para o código

No artigo [How Do Committees Invent?](#) (CONWAY, 1968), Melvin Conway descreve como sistemas tendem a reproduzir as estruturas de comunicação das empresas/orgãos/corporações que os produziram:

Lei de Conway

Qualquer organização que faz design de sistemas vai inevitavelmente produzir um design cuja estrutura é uma cópia das estruturas de comunicação dessa organização.

No estudo preliminar *Exploring the Duality between Product and*

Organizational Architectures (2008, MACCORMACK et al.) da Harvard Business School, os autores testaram o que chamam de Hipótese do Espelho (em inglês, *Mirroring Hypothesis*): a estrutura dos times influencia na modularidade dos softwares produzidos. Os autores dividem as organizações entre as *altamente acopladas*, em que as visões e os objetivos estão altamente alinhados, e as *baixamente acopladas*, organizadas como comunidades open-source distribuídas geograficamente. Para produtos similares, organizações altamente acopladas tendem a produzir softwares menos modulares.

Um exemplo da Lei de Conway aplicada pode ser encontrada no caso da Amazon, que tem uma regra conhecida como *two pizza team*: um time tem que poder ser alimentado por duas pizzas. Um tanto subjetivo, mas traz a ideia de que os times na Amazon são pequenos, independentes e autônomos, cuidando de todo o ciclo de vida do software, da concepção à operação. E isso influencia na arquitetura do software.

No artigo [Contending with Creaky Platforms CIO](#) (SIMONS; LEROY, 2010), Matthew Simons e Jonny Leroy, argumentam que a Lei de Conway pode ser descrita como: organizações disfuncionais criam aplicações disfuncionais. Por isso, refazer uma aplicação mantendo a mesma estrutura organizacional levaria às mesmas disfunções do software original. Para obter um software mais modular e organizado, poderíamos começar quebrando silos que restringem a habilidade dos times colaborarem de maneira efetiva. Os autores chamam essa ideia de **Manobra Inversa de Conway** (em inglês, *Inverse Conway's Maneuver*).

Pesquisadores da UFMG analisaram se a Lei de Conway se aplica ao kernel do Linux. Para isso, criaram uma métrica que chamaram de DOA (Degree of Authorship), que indica o quanto um determinado desenvolvedor é "autor" de um arquivo do código fonte. O DOA foi estimado por meio da verificação do histórico de uma década dos arquivos no controle de versões. A métrica relaciona um autor a um arquivo e é proporcional a quem criou o arquivo, ao número de mudanças feitas por um determinado autor e é inversamente proporcional ao

número de mudanças feitas por outros desenvolvedores. No artigo de divulgação [Does Conway's Law apply to Linux?](#) (ASERG-UFMG, 2017), é descrita a conclusão de que o kernel do Linux segue uma forma inversa da Lei de Conway, já que a arquitetura definida ao longo dos anos é que influenciou na organização e nas especializações do time de desenvolvimento.

A linguagem do negócio depende do Contexto

Um foco importante do DDD é na linguagem. Os **especialistas de domínio** usam certos termos que devem ser representados nos requisitos, nos testes e, claro, no código de produção. A linguagem do negócio deve estar representada em código, no que chamamos de **Modelo de Domínio** (em inglês, *Domain Model*). Essa linguagem estruturada em torno do domínio e usada por todos os envolvidos no desenvolvimento do software é chamada pelo DDD de **Linguagem Onipresente** (em inglês, *Ubiquitous Language*).

Porém, para uma aplicação de grande porte as coisas não são tão simples. Por exemplo, em uma aplicação de e-commerce, o que é um Produto? Para diferentes especialistas de domínio, um Produto tem diferentes significados:

- para os da Loja Online, um Produto é algo que tem preço, altura, largura e peso.
- para os do Estoque, um Produto é algo que tem uma quantidade em um inventário.
- para os do Financeiro, um Produto é algo que tem um preço e descontos.
- para os de Entrega, um Produto é algo que tem uma altura, largura e peso.

Até atributos de um Produto tem diferentes significados, dependendo do contexto. Para a Loja Online, o que interessa é a altura, largura e peso de um produto fora da caixa, que servem para um cliente saber se o item caberá na pia da sua cozinha ou em seu armário. Já para a Entrega,

esses atributos devem incluir a caixa e não influenciar nos custos de transporte.

Para aplicações maiores, manter apenas um Modelo de Domínio é inviável. A origem do problema está na linguagem utilizada pelos especialistas de domínio: não há só uma Linguagem Onipresente. Nessa situação, tentar unificar o Modelo de Domínio o tornará inconsistente.

A linguagem utilizada pelos especialistas de domínio está atrelada a uma área de negócio. Há um contexto em que um Modelo de Domínio é consistente, porque representa a linguagem de uma área de negócio. No DDD, é importante identificarmos esses **Contextos Delimitados** (em inglês, *Bounded Contexts*), para que não haja degradação dos vários Modelos de Domínio.

CONTEXTO DELIMITADO (BOUNDED CONTEXT)

Aplicabilidade delimitada de um determinado modelo. CONTEXTOS DELIMITADOS dão aos membros de uma equipe um entendimento claro e compartilhado do que deve ser consistente e o que pode se desenvolver independentemente.

[Domain Driven Design](#) (EVANS, 2003)

No fim das contas, ao alinhar as linguagens utilizadas nas áreas de negócio aos modelos representados em código, estamos caminhando na direção de uma velha promessa: *alinhar TI com o Negócio*.

Bounded Contexts no Caelum Eats

No Caelum Eats, podemos verificar como a empresa é organizada para encontrar os Bounded Contexts e influenciar na organização do nosso código.

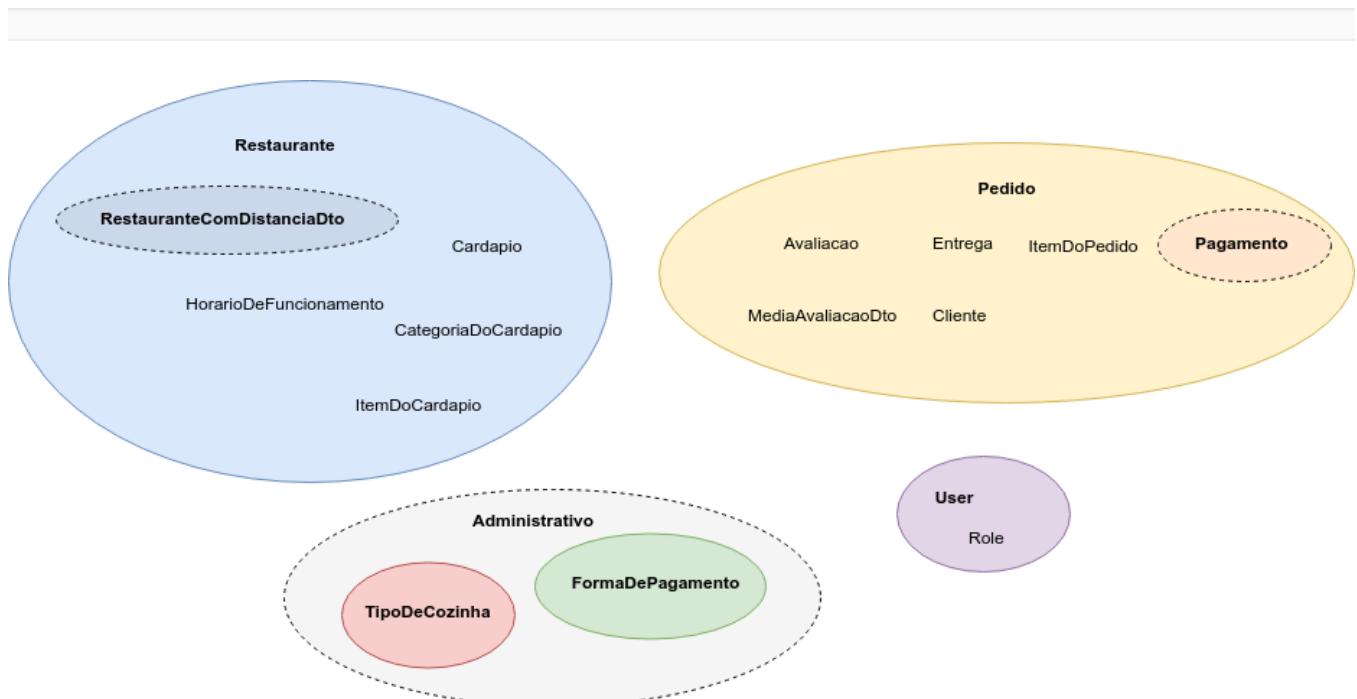
Digamos que a Caelum Eats tem, atualmente, as seguintes áreas de negócio:

- *Administrativo*, que mantém os cadastros básicos como tipos de cozinha e formas de pagamento aceitas, além de aprovar novos restaurantes
- *Pedido*, que acompanha e dá suporte aos pedidos dos clientes
- *Pagamento*, que cuida da parte Financeira e está explorando novos meios de pagamento como criptomoedas e QR Code
- *Distância*, que contém especialistas em geoprocessamento
- *Restaurante*, que lida com tudo que envolve os donos do restaurantes

Esses seriam os Bounded Contexts, que definem uma fronteira para um Modelo de Domínio consistente.

Poderíamos reorganizar o código para que a estrutura básica de pacotes seja parecida com a seguinte:

```
eats
└── administrativo
└── distancia
└── pagamento
└── pedido
└── restaurante
```



Há ainda o código de Segurança, um domínio bastante técnico que cuida

de um requisito transversal (ou não-funcional), utilizado por todas as outras partes do sistema.

Poderíamos incluir um pacote responsável por agrupar o código de segurança:

```
eats
└── administrativo
└── distancia
└── pagamento
└── pedido
└── restaurante
└── seguranca
```

Nem tudo precisa ser público

Em Java, uma classe pode ter dois modificadores de acesso: `public` ou `default`, o padrão, quando não há um modificador de acesso.

Uma classe `public` pode ser acessada por classes de qualquer outro pacote.

Já no caso de classes com modificador de acesso padrão, só podem ser acessadas por outras classes do mesmo pacote. É o que alguns chamam de *package-private*.

No caso de atributos, métodos e construtores, além de `public` e `default`, há o modificador `private`, que restringe acesso a própria classe, e `protected`, que restringe ao mesmo pacote ou classes filhas mesmo se estiverem em outros pacotes.

Pense nos projetos Java em que você trabalhou: quantas vezes você criou uma classe que não é pública?

Provavelmente, é uma implicação de organizarmos o código usando Package by Layer. Como o Controller está em um pacote, a entidade em

outro e o Repository em outro ainda, precisamos tornar as classes públicas.

Uma outra coisa que nos "empurra" na direção de todas as classes serem públicas são as IDEs: o Eclipse, por exemplo, coloca o `public` por padrão.

Porém, se alinharmos os pacotes ao domínio, passamos a ter a opção de deixar as classes acessíveis apenas no pacote em que são definidas.

Por exemplo, podemos agrupar no pacote `br.com.caelum.eats.pagamento` as seguintes classes relacionadas ao contexto de Pagamento:

```
##### br.com.caelum.eats.pagamento
```

```
.  
└── PagamentoController.java  
└── PagamentoDto.java  
└── Pagamento.java  
└── PagamentoRepository.java
```

Algumas classes como `FormaDePagamento` e `Restaurante` são usadas por classes de outros pacotes. Essas devem ser deliberadamente tornadas públicas.

Há o caso da classe `DistanciaService`, que utiliza diretamente `RestauranteRepository`. Ou seja, temos uma classe de um contexto (distância) usando um detalhe de BD de outro contexto (restaurante). É interessante manter `RestauranteRepository` com o modificador padrão e criar uma classe `RestauranteService`, responsável por expôr funcionalidades para outros pacotes.

Exercício opcional: decomposição em pacotes

1. Baixe, via Git, o projeto do monólito decomposto em pacotes:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-pa
```

2. Crie um novo workspace no Eclipse, clicando em *File > Switch Workspace > Other*. Defina o workspace `/home/<usuario-do-curso>/workspace-pacotes`, onde `<usuario-do-curso>` é o login do curso.
3. Acesse *File > Import > Existing Maven Projects* e clique em *Next*. Em *Root Directory*, aponte para o diretório clonado no passo anterior.
4. Acesse a classe `EatsApplication` e a execute com `CTRL+F11`.
5. Certifique-se que o projeto `fj33-eats-ui` esteja sendo executado. Acesse `http://localhost:4200` e teste algumas das funcionalidades. Tudo deve funcionar como antes!
6. Analise o projeto. Veja quais classes e interfaces são públicas e quais são *package private*. Observe as dependências entre os pacotes.

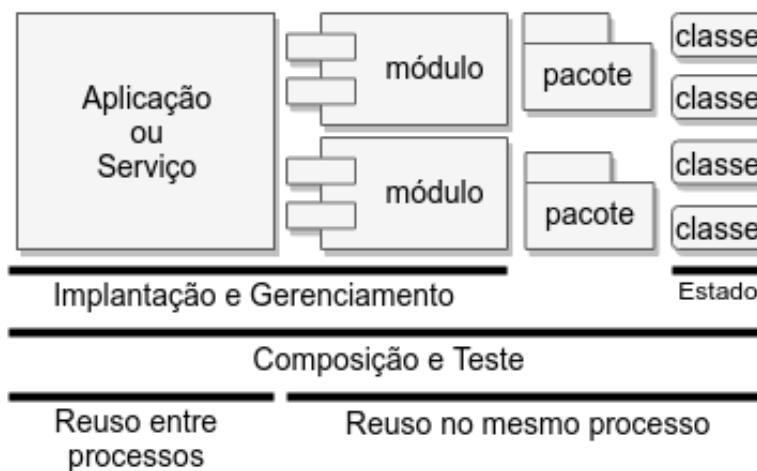
Módulos

No livro [Java Application Architecture: Modularity Patterns](#)

(KNOERNNSCHILD, 2012), Kirk Knoernschild descreve módulos como artefatos que contém as seguintes características:

- **Implantáveis**: são entregáveis que podem ser executados em *runtime*
- **Reusáveis**: são nativamente reusáveis por diferentes aplicações, sem a necessidade de comunicação pela rede. As funcionalidades de um módulo são invocadas diretamente, dentro da mesma JVM e, portanto, do mesmo processo (no Windows, o mesmo `java.exe`).
- **Testáveis**: podem ser testados independentemente, com testes de unidade.
- **Sem Estado**: módulos não mantêm estado, apenas suas classes.

- **Unidades de Composição:** podem se unir a outros módulos para compor uma aplicação.
- **Gerenciáveis:** em um sistema de módulos mais elaborado, como OSGi, podem ser instalados, reinstalados e desinstalados.



Qual será o artefato Java que contém todas essas características?

É o JAR, ou **Java ARchive**.

JARs são arquivos compactados no padrão ZIP que contém pacotes que, por sua vez, contém os `.class` compilados a partir do código fonte das classes.

Um JAR é implantável, reusável, testável, gerenciável, sem estado e é possível compô-lo com outros JARs para formar uma aplicação.

Módulos Maven

Com o Maven, é possível criarmos um **multi-module project**, que permite definir vários módulos em um mesmo projeto. O Maven ficaria responsável por obter as dependências necessárias e o fazer *build* na ordem correta. Os artefatos gerados (JARs, WARs e/ou EARs) teriam a mesma versão.

Devemos definir um módulo pai, ou supermódulo, que contém um ou mais módulos filhos, ou submódulos.

No caso do Caelum Eats, teríamos um supermódulo `eats` e submódulos para cada Bounded Context identificado anteriormente.

Além disso, precisaríamos de um submódulo que depende de todos os outros submódulos e conteria a classe principal `EatsApplication`, que possui o `main` e está anotada com `@SpringBootApplication`. Dentro desse submódulo, que chamaremos de `eats-application`, teríamos em `src/main/resources` o arquivo `application.properties` e as migrations do Flyway.

A estrutura de diretórios seria a seguinte:

```
eats
|
|   pom.xml
|
|   eats-application
|       pom.xml
|       src
|
|   eats-administrativo
|       pom.xml
|       src
|
|   eats-distancia
|       pom.xml
|       src
|
|
|   eats-pagamento
|       pom.xml
|       src
|
|
|   eats-pedido
|       pom.xml
|       src
|
```

```
|   └── eats-restaurante  
|       ├── pom.xml  
|       └── src  
|  
└── eats-seguranca  
    ├── pom.xml  
    └── src
```

O supermódulo `eats` deve definir um `pom.xml`. Nesse arquivo, a propriedade `packaging` deve ter o valor `pom`. Podem ser definidas propriedades, dependências, repositórios e outras configurações comuns a todos os submódulos. No nosso caso, definiríamos no supermódulo a dependência ao Lombok.

Os submódulos disponíveis devem ser declarados da seguinte maneira:

```
<modules>  
  <module>eats-application</module>  
  <module>eats-administrativo</module>  
  <module>eats-distancia</module>  
  <module>eats-pagamento</module>  
  <module>eats-pedido</module>  
  <module>eats-restaurante</module>  
  <module>eats-seguranca</module>  
</modules>
```

Já os submódulos não devem definir um `<groupId>` ou `<version>` próprios, apenas o `<artifactId>`. Devem declarar qual é o seu supermódulo com a tag `<parent>`. Segue o exemplo para o submódulo de restaurante:

```
<parent>  
  <groupId>br.com.caelum</groupId>  
  <artifactId>eats</artifactId>  
  <version>0.0.1-SNAPSHOT</version>
```

```
</parent>
```

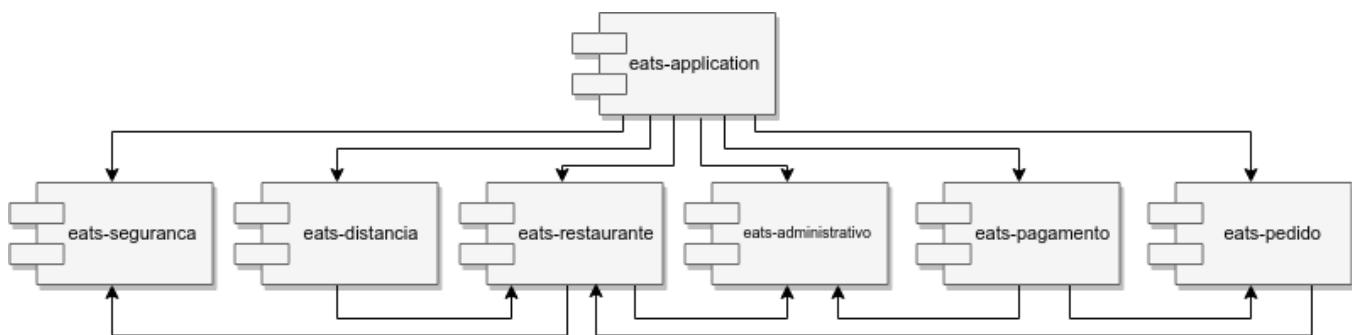
```
<artifactId>eats-restaurante</artifactId>
```

Os arquivos `pom.xml` dos submódulos podem definir em seus `pom.xml` suas próprias configurações, como propriedades, dependências e repositórios.

Se houver uma dependência a outros submódulos, é possível usar `${project.version}` como versão. Por exemplo, o submódulo `eats-restaurante` deve declarar a dependência ao submódulo `eats-administrativo` da seguinte maneira:

```
<dependency>
  <groupId>br.com.caelum</groupId>
  <artifactId>eats-administrativo</artifactId>
  <version>${project.version}</version>
</dependency>
```

É interessante notar que, nos arquivos `pom.xml`, há uma *materialização em código* da estrutura de dependências entre os módulos do sistema. Observando as dependências declaradas entre os módulos, podemos montar um diagrama parecido com o seguinte:



O submódulo `eats-application` depende de todos os outros e contém a classe principal, que contém o `main`. Ao executarmos o build, com o comando `mvn clean package` no diretório do supermódulo `eats`, o *Fat*

JAR do Spring Boot é gerado no diretório `target` do `eats-application`, contendo o código compilado da aplicação e de todas as bibliotecas utilizadas.

Pragmatismo e (um pouco de) Duplicação

Onde colocar dependências comuns a todos os módulos, como os starters do Web, Validation e Spring Data JPA?

Onde colocar classes comuns à maioria dos módulos, como a `ResourceNotFoundException`, uma exceção que gerará o status HTTP 404 (`Not Found`) quando lançada?

Muitos projetos colocam esses códigos comuns em um módulo (ou pacote) chamado `common` ou `util`. Assim evitariamos duplicação. Afinal de contas, um lema importante no design de código é *Don't Repeat Yourself* (não se repita).

Porém, criar um módulo `common` seria inserir mais uma dependência à maioria dos outros módulos.

Uma eventual extração de um módulo para outro projeto levaria à necessidade de carregar junto o conteúdo do módulo `common`.

E, possivelmente, o módulo `common` acabaria com código útil para apenas alguns módulos específicos e não para outros.

Talvez fosse mais interessante extrair esse código para bibliotecas externas (JARs), bem focadas em necessidades específicas: uma para gráficos, outra para relatórios.

Uma ideia, visando tornar os módulos o mais independentes o possível, é aceitar um pouco de duplicação. Trocaríamos o purismo da qualidade de código por pragmatismo, pensando nos próximos passos do projeto.

Exercício: o monólito decomposto em módulos Maven

1. Clone o projeto com a decomposição do monólito em módulos
Maven:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-mc
```

2. Crie o novo workspace `/home/<usuario-do-curso>/workspace-monolito-modular` no Eclipse, clicando em *File > Switch Workspace > Other*. Troque `<usuario-do-curso>` pelo login do curso.
3. Importe, pelo menu *File > Import > Existing Maven Projects* do Eclipse, o projeto `fj33-eats-monolito-modular`.
4. Para executar a aplicação, acesse o módulo `eats-application` e execute a classe `EatsApplication` com `CTRL+F11`. Certifique-se que as versões anteriores do projeto não estão sendo executadas.
5. Com o projeto `fj33-eats-ui` no ar, teste as funcionalidades por meio de `http://localhost:4200`. Deve funcionar!
6. Observe os diferentes módulos Maven. Note as dependências entre esses módulos, declaradas nos `pom.xml` de cada módulo. Note se há alguma duplicação entre os módulos.

O Monólito Modular

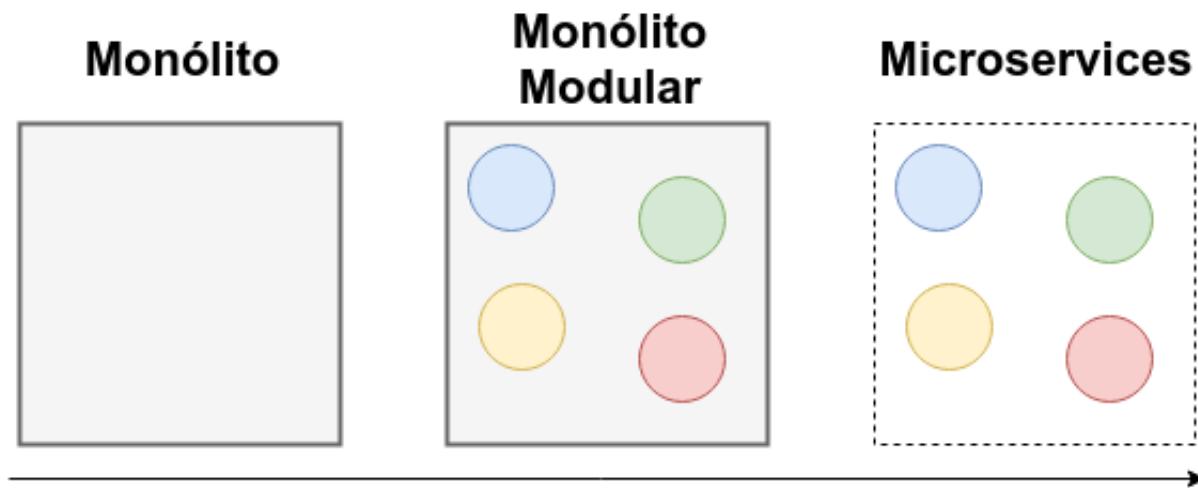
Simon Brown, na sua palestra [Modular monoliths](#) (BROWN, 2015), diz que há uma premissa comum, mas não explícita, no mercado de que todo monólito é um spaghetti e que o único caminho para um código organizado seria a migração para uma arquitetura de microservices. Só que há uma bagagem enorme de conhecimento sobre como componentizar e melhorar a manutenibilidade de um monólito. Estudamos algumas dessas ideias nesse capítulo.

Um **Monólito Modular** poderia ter diversas das características desejáveis em um código:

- alta coesão

- baixo acoplamento
- dados encapsulados
- substituíveis e passíveis de composição
- foco no negócio, inspirados por Agregados ou Bounded Contexts

Além disso, um Monólito Modular seria **um passo na direção de uma Arquitetura de Microservices**, mas sem as complexidades de um sistema distribuído.



Kirsten Westeinde, desenvolvedora sênior da Shopify, no post [Deconstructing the Monolith](#) (WESTEINDE, 2019), conta como a empresa chegou a um ponto em que a base de código monolítica, uma das maiores aplicações Ruby On Rails do mundo, começou a tornar-se tão frágil e difícil de entender, que código novo tinha consequências inesperadas e que novos desenvolvedores precisavam aprender muita coisa antes de entregar pequenas mudanças. Foi avaliada uma Arquitetura de Microservices mas foi identificado que o problema, na verdade, estava na falta de barreiras entre contextos diferentes do código. Então, escolheram um monólito modular.

Alguns times passam para uma migração para Microservices, mas resolvem voltar atrás, para um monólito modular. É o caso da Beep Saúde, como é contado por Luiz Costa, CTO, e outros desenvolvedores da Beep no episódio [Monolitos modulares e vacinas](#) (COSTA et al., 2019) do podcast Hipsters On The Road. A Beep teve que escolher entre focar os desenvolvedores em resolver os problemas técnicos trazidos por uma

Arquitetura de Microservices ou desenvolver novas funcionalidades, algo imprescindível em uma startup. Resolveram voltar para o monólito, mas de maneira modular, inspirados pela Arquitetura Hexagonal, pela Clean Architecture e pelos Bounded Contexts do DDD.

No final da palestra [Modular monoliths](#) (BROWN, 2015), Simon Brown faz uma provocação:

"Se você não consegue construir um Monólito Modular, porque você acha que microservices são a resposta"?

Porque modularizar?

No livro [Modular Java](#) (WALLS, 2009), Craig Walls cita algumas vantagens de modularizar uma aplicação:

- capacidade de trocar um módulo por outro, com uma implementação diferente, desde que a interface pública seja mantida
- facilidade de compreensão de cada módulo individualmente
- possibilidade de desenvolvimento em paralelo, permitindo que tarefas sejam divididas entre diferentes times
- testabilidade melhorada, permitindo um outro nível de testes, que trata um módulo como uma unidade
- flexibilidade, permitindo o reuso de módulos em outras aplicações

Talvez essas vantagens não sejam oferecidas pelo uso de módulos Maven. Nos textos complementares a seguir, discutiremos tecnologias como o Java Module System, a Service Loader API e OSGi, que auxiliariam a atingir essas características. Cada uma dessas tecnologias, porém, traz novas dificuldades de aprendizado, de configurações e de operação.

O que é um monólito, afinal?

Muitas vezes o termo "monólito" é usado como um *antagonista terrível*.

Mas, para uma aplicação pequena, mesmo um monólito clássico organizado em camadas não é um problema. Há diversas vantagens:

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

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

Algumas são complicações no desenvolvimento:

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

Há também complicações na operação:

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

É possível resolver parte dos problemas através de técnicas de modularização, implementando um Monólito Modular.

Um **Monólito Modular** seria composto por **componentes independentes**, que colaboram entre si através de contratos idealmente definidos em abstrações (em Java, interfaces ou classes abstratas). Com um **sistema de módulos** como o Java Module System (Java 9+) ou OSGi, é possível **reforçar as barreiras arquiteturais** mesmo com classes públicas, explicitando e limitando as dependências entre os módulos. Isso levaria a um **melhor gerenciamento das dependências** entre os módulos.

Já através de uma **Arquitetura de Plugins**, a base de código seria “fatiada” em módulos menores. Poderíamos ter **equipes mais enxutas**, focadas em cada módulo, trabalhando em várias **bases de código distintas**. A aplicação seria composta a partir de diferentes módulos em runtime. Cada desenvolvedor passaria a trabalhar com apenas uma parte do código, suavizando a IDE. Porém, subir a aplicação como um todo

ainda seria algo relativamente demorado, já que seria preciso colocar todos os módulos no ar. Os testes poderiam ser agilizados, concentrando-se em cada módulo separadamente, mas teria que ser feito um teste de sistema, que avaliaria a integração entre todos os módulos na aplicação.

Por meio de uma solução como **OSGi**, até seria possível **atualizar um módulo sem parar a aplicação** como um todo.

Monólitos Poliglotas

Um monólito não está restrito a apenas uma tecnologia de persistência, já que pode ter diferentes *data sources* para diferentes paradigmas: BDs relacionais, BDs orientados a documentos, BDs orientados a grafos, etc.

Com tecnologias como a [Graal VM](#), é possível criar monólitos desenvolvidos em múltiplas linguagens de diferentes plataformas: linguagens da JVM como Java, Scala, Kotlin, Groovy e Clojure; linguagens baseadas em LLVM como C e C++; e linguagens como JavaScript, Ruby, Python e R.

Uma definição de monólito

Como estudamos nesse capítulo, um monólito não é sinônimo de código mal feito. É possível implementar um monólito com código organizado, fatiado em pequenos pedaços independentes e alinhados com o negócio. Podemos até compor esses pedaços de diferentes maneiras e implementá-los de maneira poliglotas. Dependendo da tecnologia utilizada, podem ser atualizados parcialmente sem derrubar toda a aplicação. E podemos escalar um monólito, executando múltiplas instâncias por trás de um Load Balancer. Então o que define um monólito?

Para responder, precisamos pensar em qual é o “entregável” de uma aplicação: qual é o artefato que implantamos no ambiente de produção?

Na plataforma Java, há algum tempo, o artefato mais comum era um WAR implantado em um servlet container como Tomcat ou Jetty. Já com frameworks como o Spring Boot, temos um JAR com as dependências embutidas, um *fat JAR*. Em uma Arquitetura de Plugins, seja com Service Loader API ou com OSGi, temos uma coleção de JARs.

Em qualquer uma das maneiras de implantar um monólito, a comunicação entre as "fatias" do código é feita por chamadas de métodos, *in-memory*. E isso só é possível porque cada instância é executada em um mesmo processo.

MONÓLITO

Um monólito, modular ou não, é um sistema em que uma instância do back-end de toda a aplicação é executada em um mesmo processo do sistema operacional.

Para saber mais: Limitações dos Módulos Maven e JARs

Os módulos Maven ajudam a organizar o código de aplicações maiores porque fornecem uma maneira de representar a decomposição modular do domínio. Além disso, as dependências ficam materializadas nas declarações dos `pom.xml` de cada módulo.

Uma desvantagem dos módulos Maven como utilizamos no exemplo do exercício anterior é que **a base de código é uma só**. Diferentes times poderiam estar focados em módulos diferentes mas teriam acesso ao código dos outros módulos. A tentação de alterar algo de um módulo de outro time é muito forte! Para resolver isso, poderíamos implementar os módulos como bibliotecas completamente separadas. Um problema que iria surgir é como controlar a versão de cada módulo.

Outra desvantagem é que, em *runtime*, **a JVM não possui uma maneira de atualizar módulos** (JARs). Para atualizá-los, teríamos que parar a JVM e iniciá-la novamente, o que tornaria a aplicação indisponível. Num

ambiente com diversos times entregando software em taxas diferentes múltiplas vezes por dia/semana, a disponibilidade da aplicação seria terrivelmente afetada.

Ainda outra desvantagem dos módulos Maven é que **um módulo tem acesso às suas dependências transitivas**, ou seja, às dependências de suas dependências. Por exemplo, o módulo `eats-distancia` depende de `eats-restaurante` que, por sua vez, depende de `eats-administrativo`. Portanto, o módulo `eats-distancia` tem como dependência transitiva o módulo `eats-administrativo` e tem acesso a qualquer uma de suas classes públicas, como `FormaDePagamento` e `TipoDeCozinha`. Além disso, em `eats-distancia`, temos acesso a qualquer biblioteca declarada como dependência nos módulos `eats-restaurante` e `eats-administrativo`.

Essa fronteira fraca entre os módulos Maven pode ser problemática, já que leva a dependências indesejadas e não previstas. Se somarmos a isso o fato de que quase sempre definimos classes como públicas, módulos com muitas dependências transitivas terão acesso a boa parte das classes de outros módulos e às bibliotecas utilizadas por esses módulos. O código pode sair do controle.

Uma solução é limitar as classes públicas ao mínimo possível, tornando as classes acessíveis apenas ao pacote em que estão definidas. Mas para módulos mais complexos, teríamos dezenas ou centenas de classes no mesmo pacote! Observe, por exemplo, o módulo `eats-restaurante`: são 26 classes no mesmo pacote. Passa a ficar difícil de entender o código.

Os problemas do Classpath

Na verdade, há uma limitação nos JARs, que são apenas ZIPs com arquivos `.class` e de configuração organizados em diretórios (pacotes).

Uma vez que os JARs disponíveis são vasculhados e uma classe é carregada por um `ClassLoader` na JVM, perde-se o conceito de módulo.

O Classpath da JVM é apenas uma lista de classes, sem qualquer referência a seu JAR de origem ou de quais outros JARs dependem.

A ausência de algo que represente JAR de origem e suas dependências no Classpath enfraquece o encapsulamento em uma aplicação Java modularizada.

Nicolai Parlog, no livro [The Java Module System](#) (PARLOG, 2018), elenca os seguintes problemas no Classpath:

- *Encapsulamento fraco entre JARs*: conforme estudamos, o Classpath é uma grande lista de classes. As classes públicas são visíveis por quaisquer outras. Não é possível criar uma funcionalidade visível dentro de todos os pacotes de um JAR, mas não fora dele.
- *Ausência de representação das dependências entre JARs*: não há como um JAR declarar de quais outros JARs ele depende apenas com o Classpath.
- *Ausência de checagens automáticas de segurança*: o encapsulamento fraco dos JARs permite que código malicioso acesse e manipule funcionalidade crítica. Porém, é possível implementar manualmente checagens de segurança.
- *Sombreamento de classes com o mesmo nome*: no caso de JARs que definem duas classes com o mesmo nome, apenas uma delas é tornada disponível. E não é possível saber qual.
- *Conflitos entre versões diferentes do mesmo JAR*: duas versões do mesmo JAR no Classpath levam a comportamentos imprevisíveis.
- *JRE é rígida*: não é possível disponibilizar no Classpath um subconjunto das bibliotecas padrão do Java. Muitas classes não utilizadas ficam acessíveis.
- *Performance ruim no startup*: apesar dos class loaders serem *lazy*, carregando classes só no seu primeiro uso, muitas classes já são carregadas ao iniciar uma aplicação.
- *Class loading complexo*

Para saber mais: JPMS, um sistema de módulos para o Java

A partir do Java 9, o Java inclui um sistema de módulos bastante poderoso: o Java Platform Module System (JPMS).

Um módulo JPMS é um JAR que define:

- um nome único para o módulo
- dependências a outros módulos
- pacotes exportados, cujos tipos são acessíveis por outros módulos

Com um módulo JPMS, conseguimos definir **encapsulamento no nível de pacotes**, escolhendo quais pacotes são ou não exportados e, em consequência, acessíveis por outros módulos.

A JDK modularizada

Um dos grandes avanços do JPMS, disponível a partir da JDK 9, foi a modularização da própria plataforma Java.

O JPMS é resultado do projeto *Jigsaw*, criado em 2009, no início do desenvolvimento da JDK 7.

Antes do Java 9, todo o código das bibliotecas padrão da JDK ficava apenas no módulo de *runtime*: o `rt.jar`.

O estudo inicial do projeto *Jigsaw* agrupou o código já existente da JDK em diferentes módulos. Por exemplo, foi identificado um módulo base, que conteria pacotes fundamentais como o `java.lang` e `java.io`; um módulo desktop, com as bibliotecas Swing, AWT; além de módulos para APIs como Java Logging, JMX, JNDI.

A análise das dependências entre os módulos identificados na JDK 7 levou à descoberta de ciclos como: o módulo base depende de Logging que depende de JMX que depende de JNDI que depende de desktop que, por sua vez, depende do base.

O código da JDK 8 foi reorganizado para que não houvessem ciclos e dependências indevidas, mesmo que ainda sem um sistema de módulos propriamente dito. Os pacotes que pertenceriam ao módulo base não teriam mais dependências a nenhum outro módulo.

Na JDK 9, foram definidos módulos JPMS para cada parte do código da JDK:

- `java.base`, contendo código de pacotes como `java.lang`, `java.math`, `java.text`, `java.io`, `java.net` e `java.nio`.
- `java.logging`, com a Java Logging API.
- `java.management`, com a Java Managing Extensions (JMX) API.
- `java.naming`, com a Java Naming and Directory Interface (JNDI) API.
- `java.desktop`, contendo código de bibliotecas como Swing, AWT, 2D.

As dependências entre os módulos JPMS da JDK foram organizadas de maneira bem cuidadosa.

Antes da JDK 9, toda aplicação teria disponível todos os pacotes de todas as bibliotecas do Java. Não era possível depender de menos que a totalidade da JDK.

A partir da JDK 9, aplicações modularizadas com JPMS podem escolher, no arquivo `module-info.java`, de quais módulos da JDK dependerão.

Para saber mais: Módulos plugáveis

Antigamente, antes do Java SE 6, para ligar um plugin de uma aplicação a uma implementação era necessário:

- criar uma solução caseira usando a Reflection API
- usar bibliotecas como [JPF](#) ou [PF4J](#)
- usar uma especificação robusta, mas complexa, como [OSGi](#)

Porém, a partir do Java SE 6, a própria JRE contém uma solução: a

Service Loader API.

Na Service Loader API, um ponto de extensão é chamado de *service*.

Para provermos um service precisamos de:

- **Service Provider Interface (SPI)**: interfaces ou classes abstratas que definem a assinatura do ponto de extensão.
- **Service Provider**: uma implementação da SPI.

Para ligar a SPI com seu *service provider*, o JAR do provider precisa definir o *provider configuration file*: um arquivo com o nome da SPI dentro da pasta `META-INF/services`. O conteúdo desse arquivo deve ser o *fully qualified name* da classe de implementação.

No projeto que define a SPI, carregamos as implementações usando a classe `java.util.ServiceLoader`.

A classe `ServiceLoader` possui o método estático `load` que recebe uma SPI como parâmetro e, depois de vasculhar os diretórios `META-INF/services` dos JARs disponíveis no Classpath, retorna uma instância de `ServiceLoader` que contém todas as implementações.

O `ServiceLoader` é um `Iterable` e, por isso, pode ser percorrido com um `for-each`. Caso não haja nenhum service provider para a SPI, o `ServiceLoader` se comporta como uma lista vazia.

Perceba que uma mesma SPI pode ter vários service providers, o que traz bastante flexibilidade.

Uma Arquitetura de Plugins

Com o uso de SPIs e Service Providers, é possível criar uma Arquitetura de Plugins com a plataforma Java.

Com a Service Loader API, a simples presença de um `.jar` que a implemente a abstração do plugin (ou SPI) fará com que o

comportamento da aplicação seja estendido, sem precisarmos modificar nenhuma linha de código.

Em seu artigo [Microservices and Jars](#) (MARTIN, 2014), Uncle Bob escreve:

Não pule para Microservices só porque parece legal. Antes, segregue o sistema em JARs usando uma Arquitetura de Plugins. Se isso não for suficiente, considere a introdução de fronteiras entre serviços (service boundaries) em pontos estratégicos.

Várias bibliotecas das mais usadas por desenvolvedores Java usam SPIs.

Por meio da SPI `javax.persistence.spi.PersistenceProvider`, bibliotecas como o Hibernate e o EclipseLink fornecem implementações para as interfaces do pacote `javax.persistence`.

Do Java SE 6 em diante, a classe [DriverManager](#) carrega automaticamente todas as implementações da SPI `java.sql.Driver`. Por exemplo, o `mysql-connector-java.jar` do MySQL fornece um Service Provider para essa SPI.

O Spring tem a sua própria implementação de algo semelhante a Service Loader API: a classe [SpringFactoriesLoader](#). Essa classe é usada pelas Auto-Configurations do Spring Boot, fazendo com que a simples presença dos `.jar` dos starters adicionem comportamento à aplicação. Ou seja, o próprio Spring Boot é uma Arquitetura de Plugins.

Para saber mais: OSGi

O Java Module System foi disponibilizado a partir de 2017, com o lançamento do Java 9. Porém, a plataforma Java já tinha uma solução mais poderosa desde 1999: a especificação OSGi (Open Services Gateway Initiative). Essa especificação é implementada por frameworks como Apache Felix, Eclipse Equinox, entre outros. IDEs como Eclipse e NetBeans, servidores de aplicação como GlassFish e WebSphere, são

implementadas usando frameworks OSGi.

Por meio de diferentes Class Loaders, um framework OSGi traz a ideia de módulos para o *runtime* da JVM, corrigindo falhas do Classpath. Dessa maneira, provê um nível de encapsulamento além dos pacotes.

Um módulo, no OSGi, é chamado de *bundle*. Bundles são JARs, só que com metadados adicionais no `META-INF/MANIFEST.MF` como o nome do bundle, a versão, de quais outros bundles depende, entre outros detalhes.

Um framework OSGi controla o ciclo de vida de um bundle, fazendo com que seja instalado, iniciado, atualizado, parado e desinstalado. Múltiplas versões de um bundle podem coexistir em runtime e um bundle pode ser trocado sem parar toda a JVM.

O OSGi também especifica o conceito de *service*, análogo à Service Loader API do Java: um bundle define interface pública e outros bundles, uma ou mais implementações. Para ligar as implementações à interface, um framework OSGi provê um *service registry*. Novas implementações podem ter seu registro feito ou cancelado dinamicamente, sem parar a JVM. Os consumidores de um service dependeriam apenas da interface e do service registry, sem ter acesso a detalhes de implementação.

O nível de encapsulamento de um bundle e a possibilidade de **atualizar e registrar implementações dinamicamente** permitiria que diferentes times cuidassem de diferentes bundles alinhados com os Bounded Contexts (e as áreas de negócio) da organização. A implantação de novas versões de um bundle poderia ser feita sem derrubar a aplicação como um todo.

Porém, OSGi traz alguns problemas que limitaram sua adoção em aplicações e restringiram o uso basicamente a criadores de middleware e IDEs. Há quem diga que OSGi aumenta drasticamente o uso de memória, talvez pela implementação de diferentes Class Loaders, mas há soluções com baixo uso de memória, como de IoT, que usam implementações

OSGi. Ross Mason, criador do Mule ESB, escreve no artigo [OSGi? No Thanks](#) (MASON, 2010) que o principal dos problemas é a curva de aprendizado e a complexidade, com a necessidade de diversas configurações cheias de detalhes técnicos, o que afeta negativamente a experiência do desenvolvedor.

Um detalhe interessante é que Craig Walls, no livro [Modular Java](#) (WALLS, 2009), cita o termo *SOA in a JVM* como uma maneira usada para descrever os services do OSGi.

Um post sobre services OSGi de 2010, no blog da OSGi Alliance, usou pela primeira vez o termo [μServices](#) (KRIENS, 2010), com a letra grega mu (μ) que é usada como símbolo do prefixo *micro* pelo Sistema Internacional de Unidades.

No livro de 2012 [Java Application Architecture: Modularity Patterns](#) (KNOERNNSCHILD, 2012), Kirk Knoernschild usa repetidamente o termo μ Services para se referir aos services OSGi.

Decompondo o monólito

Da bagunça a camadas

Qual é a emoção que a palavra **monólito** traz pra você?

Muitos desenvolvedores associam a palavra monólito a código mal feito, sem estrutura aparente, com muito código repetido e com dados compartilhados entre partes pouco relacionadas. É o que comumente é chamado de código **Spaghetti** ou, [Big Ball of Mud](#) (FOOTE; YODER, 1999).

Porém, é possível criar monólitos bem estruturados. Uma maneira comum de organizar um monólito é usar camadas: cada camada provê um serviço para a camada de cima e é um cliente das camadas de baixo.

Usualmente, o código de uma aplicação é estruturado em 3 camadas:

- *Apresentação*: responsável por prover serviços ao front-end. Em alguns casos, é feita a renderização das telas da UI, como ao utilizar JSPs.
- *Negócio*: responsável pelos cálculos, fluxos e regras de negócio.
- *Persistência*: responsável pelo acesso aos dados armazenados, geralmente, em um Banco de Dados.

Uma maneira de representar essas camadas em um código Java é utilizar pacotes, o que é conhecido como *Package by Layer*.

Na verdade, é preciso distinguir dois tipos de camadas que influenciam a arquitetura do software: as camadas físicas e as camadas lógicas.

Uma camada física, ou *tier* em inglês, descreve a estrutura de implantação do software, ou seja, as máquinas utilizadas.

O que descrevemos no texto anterior é o conceito de camada lógica. Em inglês, a camada lógica é chamada de *layer*. Trata de agrupar o

código que corresponde às camadas descritas anteriormente.

Camadas no Caelum Eats

O código de back-end do Caelum Eats está organizado em camadas (layers). Podemos observar isso estudando a estrutura de pacotes do código Java:

```
eats
└── controller
└── dto
└── exception
└── model
└── repository
└── service
```

Bem organizado, não é mesmo?

Porém, quando a aplicação começa a crescer, o código passa a ficar difícil de entender. Centenas de Controllers juntos, centenas de Repositories misturados. Passa a ser difícil encontrar o código que precisa ser alterado.

A Arquitetura que Grita

Reflita um pouco: dos projetos implementados com a plataforma Java em que você trabalhou, quantos seguiam uma variação da estrutura Package by Layer que estudamos anteriormente? Provavelmente, a maioria!

Veja a estrutura de diretórios abaixo:

```
.
└── assets
└── controllers
└── helpers
└── mailers
```

```
└── models  
└── views
```

Os diretórios anteriores são a estrutura padrão de um framework de aplicações Web muito influente: o Ruby On Rails.

Perceba que interessante: a estrutura básica de diretórios é familiar; em alguns casos, até temos um palpite sobre qual o framework utilizado; mas não temos ideia do **domínio** da aplicação. Qual é o problema que está sendo resolvido?

No post [Screaming Architecture](#) (MARTIN, 2011), Robert "Uncle Bob" Martin diz que a partir das plantas de edifícios, conseguimos saber se trata-se de uma casa ou uma biblioteca: a arquitetura "grita" a finalidade da construção.

Na realidade, a construção civil tem diferentes plantas: a elétrica, a hidráulica, a estrutural, etc. Uncle Bob parece referir-se a um tipo específico de planta: a **planta baixa**. É feita por um arquiteto e é mais próxima do cliente. A partir dela são projetadas outras plantas mais técnicas e específicas. É o tipo de planta que encontramos em lançamentos de imóveis.

Para projetos de software, entretanto, é usual que a estrutura básica de diretórios indique qual o framework ou qual a ferramenta de build utilizados. Porém, para Uncle Bob, o framework é um detalhe; o Banco de Dados é um detalhe; a Web é um mecanismo de entrega da UI, um detalhe.

Uncle Bob cita a ideia de Ivar Jacobson, um dos criadores do UML, descrita no livro [Object Oriented Software Engineering: A Use-Case Driven Approach](#) (JACOBSON, 1992), de que a arquitetura de um software deveria ser *centrada nos casos de uso* e não em detalhes técnicos.

Por arquiteturas centradas no domínio

No livro [Clean Architecture](#) (MARTIN, 2017), Uncle Bob define uma abordagem arquitetural que torna a aplicação:

- **independente de frameworks**: um framework não é sua aplicação. A estrutura de diretórios e as restrições do design do nosso código não deveriam ser determinadas por um framework. Frameworks deveriam ser usados apenas como ferramentas para que a aplicação cumpra suas necessidades.
- **independente da UI**: a UI muda mais frequentemente que o resto do sistema. Além disso, é possível termos diferentes UIs para as mesmas regras de negócio.
- **independente de BD**: as regras de negócio não devem depender de um Banco de Dados específico. Devemos possibilitar a troca, de maneira fácil, de Oracle ou SQL Server para MongoDB, CouchDB, Neo4J ou qualquer outro BD.
- **testável**: deve ser possível testar as regras de negócio diretamente, sem a necessidade de usar uma UI, BD ou servidor Web.

Uncle Bob ainda cita, no mesmo livro, outras arquiteturas semelhantes:

- Hexagonal Architecture, ou Ports & Adapters, descrita por Alistair Cockburn.
- DCI (Data, Context and Interaction), descrita por James Coplien, pioneiro dos Design Patterns, e Trygve Reenskaug, criador do MVC.
- BCE (Boundary-Control-Entity), introduzida por Ivar Jacobson no livro mencionado anteriormente.

O curso [Práticas de Design e Arquitetura de código para aplicações Java](#) (FJ-38) explora, a partir de um gerador de ebooks, tópicos como os princípios SOLID de Orientação a Objetos e alguns Design Patterns, até chegar progressivamente a uma Arquitetura Hexagonal.

Software é massa!

Raymond J. Rubey cita uma carta ao editor da revista CROSSTALK (RUBEY, 1992), em que é feita uma brincadeira que classifica qualidade

de código como se fossem massas:

- **Spaghetti**: complicado, difícil de entender e impossível de manter
- **Lasagna**: simples, fácil de entender, estruturado em camadas, mas monolítico; na teoria, é fácil de mudar uma camada, mas não na prática
- **Ravioli**: componentes pequenos e soltos, que contém "nutrientes" para o sistema; qualquer componente pode ser modificado ou trocado sem afetar significativamente os outros componentes

Explorando o domínio

Até um certo tamanho, uma aplicação estruturada em camadas no estilo Package by Layer, é fácil de entender. Novos desenvolvedores entram no projeto sem muitas dificuldades para entender a organização do código, já que há uma certa familiaridade.

Mas há um momento em que é interessante reorganizar os pacotes ao redor do domínio, o que alguns chamam de *Package by Feature*. O intuito é que fique bem clara qual é a área de negócio de cada parte do código.

Mas qual critério usar para decompor o monólito? Uma funcionalidade é um pacote, como sugere o nome *Package by Feature*? Talvez isso seja muito granular.

Uma fonte de *insights* interessantes em como explorar o domínio de uma aplicação é o livro [Domain Driven Design](#), de Eric Evans (EVANS, 2003).

Caçando entidades importantes

Todo domínio tem entidades importantes, que são um ponto de entrada para o negócio e tem várias outras entidades ligadas. Um objeto que representa um pedido, por exemplo, terá informações dos itens do pedido, do cliente, da entrega e do pagamento. É o que é chamado, nos termos do DDD, de *Agregado*.

| AGREGADO

Agrupamento de objetos associados que são tratados como uma unidade para fins de alterações nos dados. Referências externas são restritas a um único membro do AGREGADO, designado como *raiz*. Um conjunto de regras de consistência se aplicada dentro dos limites do AGREGADO.

[Domain Driven Design](#) (EVANS, 2003)

Referências a objetos de fora desse Agregado devem ser feitas por meio do objeto principal, a raiz do Agregado. Uma relação entre um pedido e um restaurante deve ser feita pelo pedido, e não pela entrega ou pelo pagamento do pedido.

Os dados de um Agregado são alterados em conjunto. Por isso, Banco de Dados Orientados a Documentos, como o MongoDB, em que um documento pode ter um grafo de outros objetos conectados, são interessantes para persistir Agregados.

E no Caelum Eats?

Um objeto muito importante é o `Pedido`. Associado a um `Pedido` temos uma lista de `ItemDoPedido` e uma `Entrega`. Uma `Entrega` está associada a um `Cliente`. Cada `Pedido` também pode ter uma `Avaliacao`. Há, possivelmente, um `Pagamento` para cada `Pedido`. E um `Pagamento` tem uma `FormaDePagamento`. Um `Pedido` também está associado a um `Restaurante`.

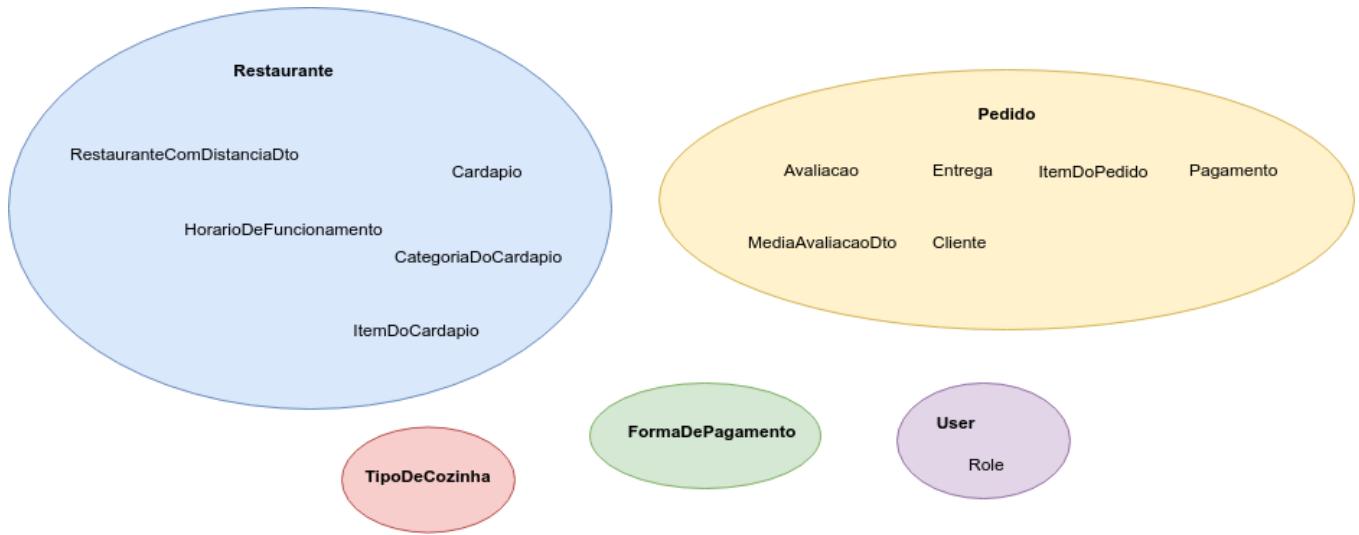
Um `Restaurante` tem seu cadastro mantido por seu dono e é aprovado pelo administrador do Caelum Eats. Um `Restaurante` pode existir sem nenhum pedido, o que acontece logo que foi cadastrado. Um `Restaurante` está relacionado a um `cardapio` que contém uma lista de `CategoriaDoCardapio` que, por sua vez, contém uma lista de `ItemDoCardapio`. Um `Restaurante` possui também uma lista de `HorarioDeFuncionamento` e das `FormaDePagamento` aceitas, assim como um `TipoDeCozinha`.

Um `Restaurante` também possui um `User`, que representa seu dono. O

User também é utilizado pelo administrador do Caelum Eats. Um User tem um ou mais Role.

Uma FormaDePagamento existe independentemente de um Restaurante e os valores possíveis são mantidos pelo administrador. O mesmo ocorre para TipoDeCozinha.

Há algumas classes interessantes como MediaAvaliacoesDto e RestauranteComDistanciaDto, que associam um restaurante à média das notas das avaliações e uma distância a um dado CEP, respectivamente.



Poderíamos alinhar o código do Caelum Eats com o domínio, utilizando os agregados identificados anteriormente.

Das estruturas de comunicação para o código

No artigo [How Do Committees Invent?](#) (CONWAY, 1968), Melvin Conway descreve como sistemas tendem a reproduzir as estruturas de comunicação das empresas/orgãos/corporações que os produziram:

Lei de Conway

Qualquer organização que faz design de sistemas vai inevitavelmente produzir um design cuja estrutura é uma cópia das estruturas de comunicação dessa organização.

No estudo preliminar *Exploring the Duality between Product and*

Organizational Architectures (2008, MACCORMACK et al.) da Harvard Business School, os autores testaram o que chamam de Hipótese do Espelho (em inglês, *Mirroring Hypothesis*): a estrutura dos times influencia na modularidade dos softwares produzidos. Os autores dividem as organizações entre as *altamente acopladas*, em que as visões e os objetivos estão altamente alinhados, e as *baixamente acopladas*, organizadas como comunidades open-source distribuídas geograficamente. Para produtos similares, organizações altamente acopladas tendem a produzir softwares menos modulares.

Um exemplo da Lei de Conway aplicada pode ser encontrada no caso da Amazon, que tem uma regra conhecida como *two pizza team*: um time tem que poder ser alimentado por duas pizzas. Um tanto subjetivo, mas traz a ideia de que os times na Amazon são pequenos, independentes e autônomos, cuidando de todo o ciclo de vida do software, da concepção à operação. E isso influencia na arquitetura do software.

No artigo [Contending with Creaky Platforms CIO](#) (SIMONS; LEROY, 2010), Matthew Simons e Jonny Leroy, argumentam que a Lei de Conway pode ser descrita como: organizações disfuncionais criam aplicações disfuncionais. Por isso, refazer uma aplicação mantendo a mesma estrutura organizacional levaria às mesmas disfunções do software original. Para obter um software mais modular e organizado, poderíamos começar quebrando silos que restringem a habilidade dos times colaborarem de maneira efetiva. Os autores chamam essa ideia de **Manobra Inversa de Conway** (em inglês, *Inverse Conway's Maneuver*).

Pesquisadores da UFMG analisaram se a Lei de Conway se aplica ao kernel do Linux. Para isso, criaram uma métrica que chamaram de DOA (Degree of Authorship), que indica o quanto um determinado desenvolvedor é "autor" de um arquivo do código fonte. O DOA foi estimado por meio da verificação do histórico de uma década dos arquivos no controle de versões. A métrica relaciona um autor a um arquivo e é proporcional a quem criou o arquivo, ao número de mudanças feitas por um determinado autor e é inversamente proporcional ao

número de mudanças feitas por outros desenvolvedores. No artigo de divulgação [Does Conway's Law apply to Linux?](#) (ASERG-UFMG, 2017), é descrita a conclusão de que o kernel do Linux segue uma forma inversa da Lei de Conway, já que a arquitetura definida ao longo dos anos é que influenciou na organização e nas especializações do time de desenvolvimento.

A linguagem do negócio depende do Contexto

Um foco importante do DDD é na linguagem. Os **especialistas de domínio** usam certos termos que devem ser representados nos requisitos, nos testes e, claro, no código de produção. A linguagem do negócio deve estar representada em código, no que chamamos de **Modelo de Domínio** (em inglês, *Domain Model*). Essa linguagem estruturada em torno do domínio e usada por todos os envolvidos no desenvolvimento do software é chamada pelo DDD de **Linguagem Onipresente** (em inglês, *Ubiquitous Language*).

Porém, para uma aplicação de grande porte as coisas não são tão simples. Por exemplo, em uma aplicação de e-commerce, o que é um Produto? Para diferentes especialistas de domínio, um Produto tem diferentes significados:

- para os da Loja Online, um Produto é algo que tem preço, altura, largura e peso.
- para os do Estoque, um Produto é algo que tem uma quantidade em um inventário.
- para os do Financeiro, um Produto é algo que tem um preço e descontos.
- para os de Entrega, um Produto é algo que tem uma altura, largura e peso.

Até atributos de um Produto tem diferentes significados, dependendo do contexto. Para a Loja Online, o que interessa é a altura, largura e peso de um produto fora da caixa, que servem para um cliente saber se o item caberá na pia da sua cozinha ou em seu armário. Já para a Entrega,

esses atributos devem incluir a caixa e não influenciar nos custos de transporte.

Para aplicações maiores, manter apenas um Modelo de Domínio é inviável. A origem do problema está na linguagem utilizada pelos especialistas de domínio: não há só uma Linguagem Onipresente. Nessa situação, tentar unificar o Modelo de Domínio o tornará inconsistente.

A linguagem utilizada pelos especialistas de domínio está atrelada a uma área de negócio. Há um contexto em que um Modelo de Domínio é consistente, porque representa a linguagem de uma área de negócio. No DDD, é importante identificarmos esses **Contextos Delimitados** (em inglês, *Bounded Contexts*), para que não haja degradação dos vários Modelos de Domínio.

CONTEXTO DELIMITADO (BOUNDED CONTEXT)

Aplicabilidade delimitada de um determinado modelo. CONTEXTOS DELIMITADOS dão aos membros de uma equipe um entendimento claro e compartilhado do que deve ser consistente e o que pode se desenvolver independentemente.

[Domain Driven Design](#) (EVANS, 2003)

No fim das contas, ao alinhar as linguagens utilizadas nas áreas de negócio aos modelos representados em código, estamos caminhando na direção de uma velha promessa: *alinhar TI com o Negócio*.

Bounded Contexts no Caelum Eats

No Caelum Eats, podemos verificar como a empresa é organizada para encontrar os Bounded Contexts e influenciar na organização do nosso código.

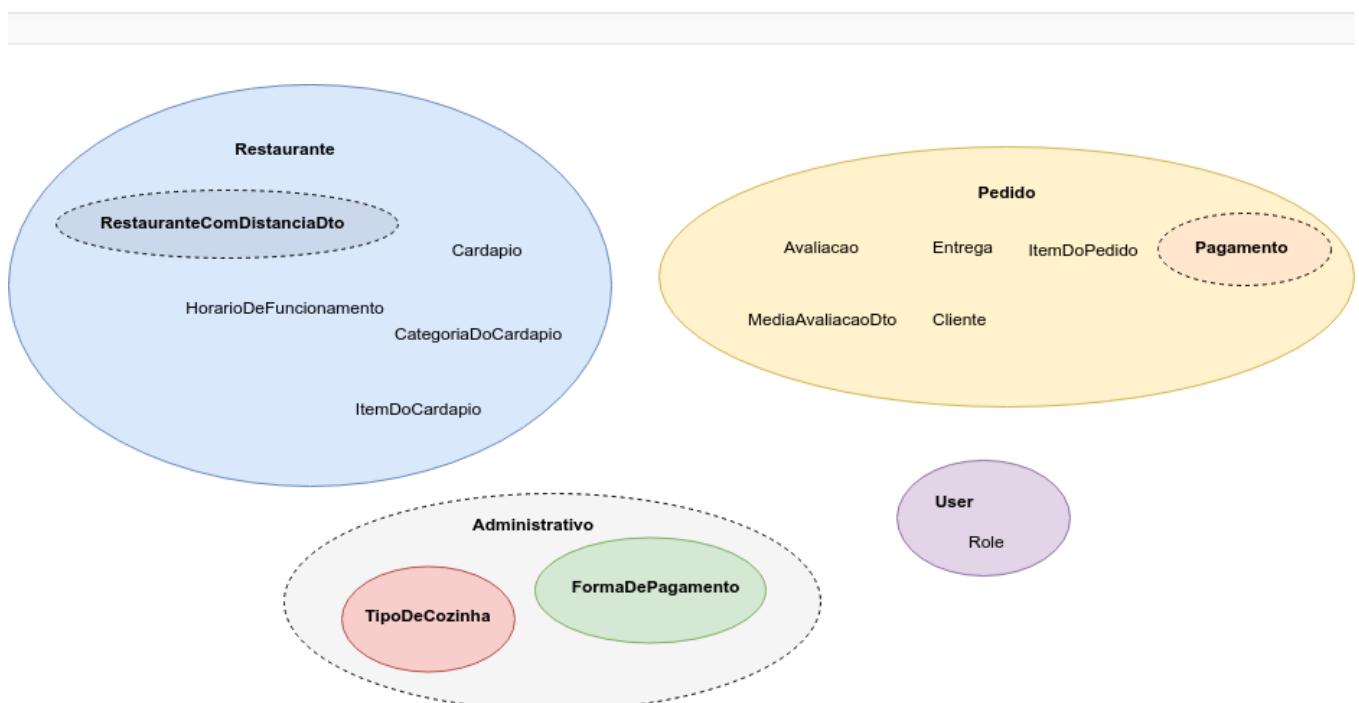
Digamos que a Caelum Eats tem, atualmente, as seguintes áreas de negócio:

- *Administrativo*, que mantém os cadastros básicos como tipos de cozinha e formas de pagamento aceitas, além de aprovar novos restaurantes
- *Pedido*, que acompanha e dá suporte aos pedidos dos clientes
- *Pagamento*, que cuida da parte Financeira e está explorando novos meios de pagamento como criptomoedas e QR Code
- *Distância*, que contém especialistas em geoprocessamento
- *Restaurante*, que lida com tudo que envolve os donos do restaurantes

Esses seriam os Bounded Contexts, que definem uma fronteira para um Modelo de Domínio consistente.

Poderíamos reorganizar o código para que a estrutura básica de pacotes seja parecida com a seguinte:

```
eats
└── administrativo
└── distancia
└── pagamento
└── pedido
└── restaurante
```



Há ainda o código de Segurança, um domínio bastante técnico que cuida

de um requisito transversal (ou não-funcional), utilizado por todas as outras partes do sistema.

Poderíamos incluir um pacote responsável por agrupar o código de segurança:

```
eats
└── administrativo
└── distancia
└── pagamento
└── pedido
└── restaurante
└── seguranca
```

Nem tudo precisa ser público

Em Java, uma classe pode ter dois modificadores de acesso: `public` ou `default`, o padrão, quando não há um modificador de acesso.

Uma classe `public` pode ser acessada por classes de qualquer outro pacote.

Já no caso de classes com modificador de acesso padrão, só podem ser acessadas por outras classes do mesmo pacote. É o que alguns chamam de *package-private*.

No caso de atributos, métodos e construtores, além de `public` e `default`, há o modificador `private`, que restringe acesso a própria classe, e `protected`, que restringe ao mesmo pacote ou classes filhas mesmo se estiverem em outros pacotes.

Pense nos projetos Java em que você trabalhou: quantas vezes você criou uma classe que não é pública?

Provavelmente, é uma implicação de organizarmos o código usando Package by Layer. Como o Controller está em um pacote, a entidade em

outro e o Repository em outro ainda, precisamos tornar as classes públicas.

Uma outra coisa que nos "empurra" na direção de todas as classes serem públicas são as IDEs: o Eclipse, por exemplo, coloca o `public` por padrão.

Porém, se alinharmos os pacotes ao domínio, passamos a ter a opção de deixar as classes acessíveis apenas no pacote em que são definidas.

Por exemplo, podemos agrupar no pacote `br.com.caelum.eats.pagamento` as seguintes classes relacionadas ao contexto de Pagamento:

```
##### br.com.caelum.eats.pagamento
```

```
.  
└── PagamentoController.java  
└── PagamentoDto.java  
└── Pagamento.java  
└── PagamentoRepository.java
```

Algumas classes como `FormaDePagamento` e `Restaurante` são usadas por classes de outros pacotes. Essas devem ser deliberadamente tornadas públicas.

Há o caso da classe `DistanciaService`, que utiliza diretamente `RestauranteRepository`. Ou seja, temos uma classe de um contexto (distância) usando um detalhe de BD de outro contexto (restaurante). É interessante manter `RestauranteRepository` com o modificador padrão e criar uma classe `RestauranteService`, responsável por expôr funcionalidades para outros pacotes.

Exercício opcional: decomposição em pacotes

1. Baixe, via Git, o projeto do monólito decomposto em pacotes:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-pa
```

2. Crie um novo workspace no Eclipse, clicando em *File > Switch Workspace > Other*. Defina o workspace `/home/<usuario-do-curso>/workspace-pacotes`, onde `<usuario-do-curso>` é o login do curso.
3. Acesse *File > Import > Existing Maven Projects* e clique em *Next*. Em *Root Directory*, aponte para o diretório clonado no passo anterior.
4. Acesse a classe `EatsApplication` e a execute com `CTRL+F11`.
5. Certifique-se que o projeto `fj33-eats-ui` esteja sendo executado. Acesse `http://localhost:4200` e teste algumas das funcionalidades. Tudo deve funcionar como antes!
6. Analise o projeto. Veja quais classes e interfaces são públicas e quais são *package private*. Observe as dependências entre os pacotes.

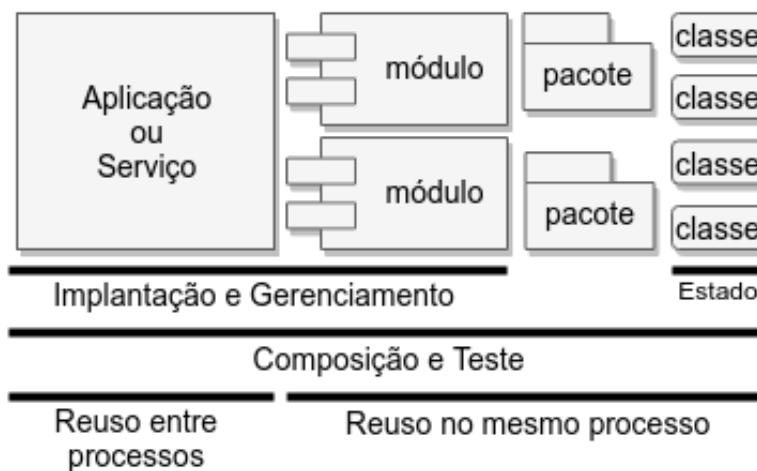
Módulos

No livro [Java Application Architecture: Modularity Patterns](#)

(KNOERNNSCHILD, 2012), Kirk Knoernschild descreve módulos como artefatos que contém as seguintes características:

- **Implantáveis**: são entregáveis que podem ser executados em *runtime*
- **Reusáveis**: são nativamente reusáveis por diferentes aplicações, sem a necessidade de comunicação pela rede. As funcionalidades de um módulo são invocadas diretamente, dentro da mesma JVM e, portanto, do mesmo processo (no Windows, o mesmo `java.exe`).
- **Testáveis**: podem ser testados independentemente, com testes de unidade.
- **Sem Estado**: módulos não mantêm estado, apenas suas classes.

- **Unidades de Composição:** podem se unir a outros módulos para compor uma aplicação.
- **Gerenciáveis:** em um sistema de módulos mais elaborado, como OSGi, podem ser instalados, reinstalados e desinstalados.



Qual será o artefato Java que contém todas essas características?

É o JAR, ou **Java ARchive**.

JARs são arquivos compactados no padrão ZIP que contém pacotes que, por sua vez, contém os `.class` compilados a partir do código fonte das classes.

Um JAR é implantável, reusável, testável, gerenciável, sem estado e é possível compô-lo com outros JARs para formar uma aplicação.

Módulos Maven

Com o Maven, é possível criarmos um **multi-module project**, que permite definir vários módulos em um mesmo projeto. O Maven ficaria responsável por obter as dependências necessárias e o fazer *build* na ordem correta. Os artefatos gerados (JARs, WARs e/ou EARs) teriam a mesma versão.

Devemos definir um módulo pai, ou supermódulo, que contém um ou mais módulos filhos, ou submódulos.

No caso do Caelum Eats, teríamos um supermódulo `eats` e submódulos para cada Bounded Context identificado anteriormente.

Além disso, precisaríamos de um submódulo que depende de todos os outros submódulos e conteria a classe principal `EatsApplication`, que possui o `main` e está anotada com `@SpringBootApplication`. Dentro desse submódulo, que chamaremos de `eats-application`, teríamos em `src/main/resources` o arquivo `application.properties` e as migrations do Flyway.

A estrutura de diretórios seria a seguinte:

```
eats
|
|   pom.xml
|
|   eats-application
|       pom.xml
|       src
|
|   eats-administrativo
|       pom.xml
|       src
|
|   eats-distancia
|       pom.xml
|       src
|
|
|   eats-pagamento
|       pom.xml
|       src
|
|
|   eats-pedido
|       pom.xml
|       src
|
```

```
|   └── eats-restaurante  
|       ├── pom.xml  
|       └── src  
|  
└── eats-seguranca  
    ├── pom.xml  
    └── src
```

O supermódulo `eats` deve definir um `pom.xml`. Nesse arquivo, a propriedade `packaging` deve ter o valor `pom`. Podem ser definidas propriedades, dependências, repositórios e outras configurações comuns a todos os submódulos. No nosso caso, definiríamos no supermódulo a dependência ao Lombok.

Os submódulos disponíveis devem ser declarados da seguinte maneira:

```
<modules>  
  <module>eats-application</module>  
  <module>eats-administrativo</module>  
  <module>eats-distancia</module>  
  <module>eats-pagamento</module>  
  <module>eats-pedido</module>  
  <module>eats-restaurante</module>  
  <module>eats-seguranca</module>  
</modules>
```

Já os submódulos não devem definir um `<groupId>` ou `<version>` próprios, apenas o `<artifactId>`. Devem declarar qual é o seu supermódulo com a tag `<parent>`. Segue o exemplo para o submódulo de restaurante:

```
<parent>  
  <groupId>br.com.caelum</groupId>  
  <artifactId>eats</artifactId>  
  <version>0.0.1-SNAPSHOT</version>
```

```
</parent>
```

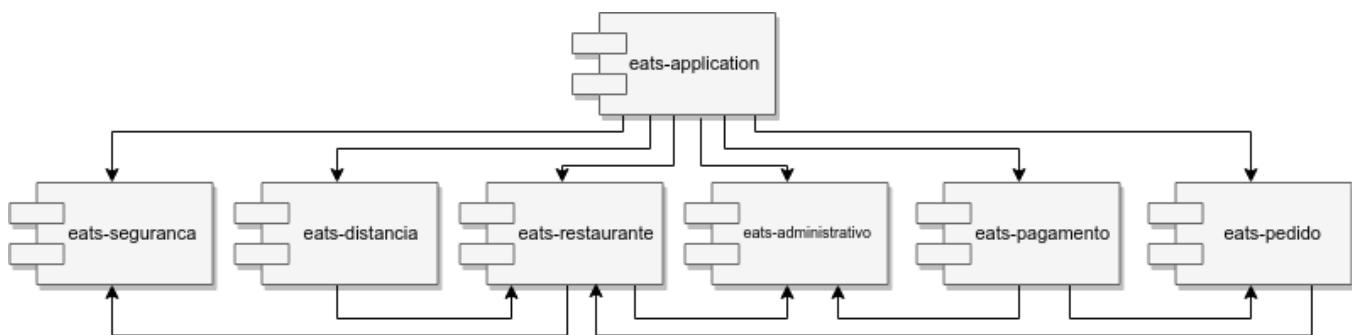
```
<artifactId>eats-restaurante</artifactId>
```

Os arquivos `pom.xml` dos submódulos podem definir em seus `pom.xml` suas próprias configurações, como propriedades, dependências e repositórios.

Se houver uma dependência a outros submódulos, é possível usar `${project.version}` como versão. Por exemplo, o submódulo `eats-restaurante` deve declarar a dependência ao submódulo `eats-administrativo` da seguinte maneira:

```
<dependency>
  <groupId>br.com.caelum</groupId>
  <artifactId>eats-administrativo</artifactId>
  <version>${project.version}</version>
</dependency>
```

É interessante notar que, nos arquivos `pom.xml`, há uma *materialização em código* da estrutura de dependências entre os módulos do sistema. Observando as dependências declaradas entre os módulos, podemos montar um diagrama parecido com o seguinte:



O submódulo `eats-application` depende de todos os outros e contém a classe principal, que contém o `main`. Ao executarmos o build, com o comando `mvn clean package` no diretório do supermódulo `eats`, o *Fat*

JAR do Spring Boot é gerado no diretório `target` do `eats-application`, contendo o código compilado da aplicação e de todas as bibliotecas utilizadas.

Pragmatismo e (um pouco de) Duplicação

Onde colocar dependências comuns a todos os módulos, como os starters do Web, Validation e Spring Data JPA?

Onde colocar classes comuns à maioria dos módulos, como a `ResourceNotFoundException`, uma exceção que gerará o status HTTP 404 (`Not Found`) quando lançada?

Muitos projetos colocam esses códigos comuns em um módulo (ou pacote) chamado `common` ou `util`. Assim evitariamos duplicação. Afinal de contas, um lema importante no design de código é *Don't Repeat Yourself* (não se repita).

Porém, criar um módulo `common` seria inserir mais uma dependência à maioria dos outros módulos.

Uma eventual extração de um módulo para outro projeto levaria à necessidade de carregar junto o conteúdo do módulo `common`.

E, possivelmente, o módulo `common` acabaria com código útil para apenas alguns módulos específicos e não para outros.

Talvez fosse mais interessante extrair esse código para bibliotecas externas (JARs), bem focadas em necessidades específicas: uma para gráficos, outra para relatórios.

Uma ideia, visando tornar os módulos o mais independentes o possível, é aceitar um pouco de duplicação. Trocaríamos o purismo da qualidade de código por pragmatismo, pensando nos próximos passos do projeto.

Exercício: o monólito decomposto em módulos Maven

1. Clone o projeto com a decomposição do monólito em módulos
Maven:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-mc
```

2. Crie o novo workspace `/home/<usuario-do-curso>/workspace-monolito-modular` no Eclipse, clicando em *File > Switch Workspace > Other*. Troque `<usuario-do-curso>` pelo login do curso.
3. Importe, pelo menu *File > Import > Existing Maven Projects* do Eclipse, o projeto `fj33-eats-monolito-modular`.
4. Para executar a aplicação, acesse o módulo `eats-application` e execute a classe `EatsApplication` com `CTRL+F11`. Certifique-se que as versões anteriores do projeto não estão sendo executadas.
5. Com o projeto `fj33-eats-ui` no ar, teste as funcionalidades por meio de `http://localhost:4200`. Deve funcionar!
6. Observe os diferentes módulos Maven. Note as dependências entre esses módulos, declaradas nos `pom.xml` de cada módulo. Note se há alguma duplicação entre os módulos.

O Monólito Modular

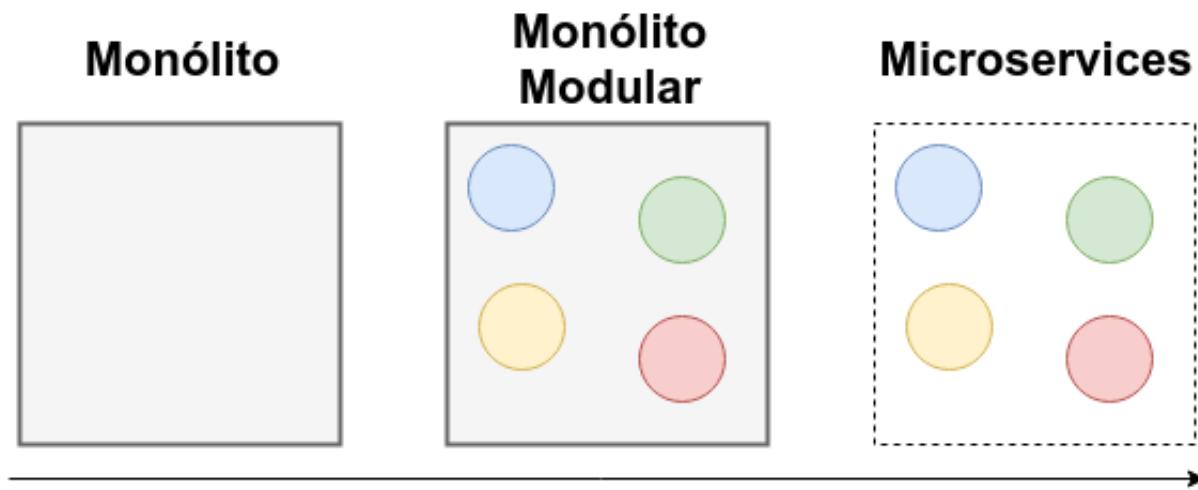
Simon Brown, na sua palestra [Modular monoliths](#) (BROWN, 2015), diz que há uma premissa comum, mas não explícita, no mercado de que todo monólito é um spaghetti e que o único caminho para um código organizado seria a migração para uma arquitetura de microservices. Só que há uma bagagem enorme de conhecimento sobre como componentizar e melhorar a manutenibilidade de um monólito. Estudamos algumas dessas ideias nesse capítulo.

Um **Monólito Modular** poderia ter diversas das características desejáveis em um código:

- alta coesão

- baixo acoplamento
- dados encapsulados
- substituíveis e passíveis de composição
- foco no negócio, inspirados por Agregados ou Bounded Contexts

Além disso, um Monólito Modular seria **um passo na direção de uma Arquitetura de Microservices**, mas sem as complexidades de um sistema distribuído.



Kirsten Westeinde, desenvolvedora sênior da Shopify, no post [Deconstructing the Monolith](#) (WESTEINDE, 2019), conta como a empresa chegou a um ponto em que a base de código monolítica, uma das maiores aplicações Ruby On Rails do mundo, começou a tornar-se tão frágil e difícil de entender, que código novo tinha consequências inesperadas e que novos desenvolvedores precisavam aprender muita coisa antes de entregar pequenas mudanças. Foi avaliada uma Arquitetura de Microservices mas foi identificado que o problema, na verdade, estava na falta de barreiras entre contextos diferentes do código. Então, escolheram um monólito modular.

Alguns times passam para uma migração para Microservices, mas resolvem voltar atrás, para um monólito modular. É o caso da Beep Saúde, como é contado por Luiz Costa, CTO, e outros desenvolvedores da Beep no episódio [Monolitos modulares e vacinas](#) (COSTA et al., 2019) do podcast Hipsters On The Road. A Beep teve que escolher entre focar os desenvolvedores em resolver os problemas técnicos trazidos por uma

Arquitetura de Microservices ou desenvolver novas funcionalidades, algo imprescindível em uma startup. Resolveram voltar para o monólito, mas de maneira modular, inspirados pela Arquitetura Hexagonal, pela Clean Architecture e pelos Bounded Contexts do DDD.

No final da palestra [Modular monoliths](#) (BROWN, 2015), Simon Brown faz uma provocação:

"Se você não consegue construir um Monólito Modular, porque você acha que microservices são a resposta"?

Porque modularizar?

No livro [Modular Java](#) (WALLS, 2009), Craig Walls cita algumas vantagens de modularizar uma aplicação:

- capacidade de trocar um módulo por outro, com uma implementação diferente, desde que a interface pública seja mantida
- facilidade de compreensão de cada módulo individualmente
- possibilidade de desenvolvimento em paralelo, permitindo que tarefas sejam divididas entre diferentes times
- testabilidade melhorada, permitindo um outro nível de testes, que trata um módulo como uma unidade
- flexibilidade, permitindo o reuso de módulos em outras aplicações

Talvez essas vantagens não sejam oferecidas pelo uso de módulos Maven. Nos textos complementares a seguir, discutiremos tecnologias como o Java Module System, a Service Loader API e OSGi, que auxiliariam a atingir essas características. Cada uma dessas tecnologias, porém, traz novas dificuldades de aprendizado, de configurações e de operação.

O que é um monólito, afinal?

Muitas vezes o termo "monólito" é usado como um *antagonista terrível*.

Mas, para uma aplicação pequena, mesmo um monólito clássico organizado em camadas não é um problema. Há diversas vantagens:

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

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

Algumas são complicações no desenvolvimento:

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

Há também complicações na operação:

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

É possível resolver parte dos problemas através de técnicas de modularização, implementando um Monólito Modular.

Um **Monólito Modular** seria composto por **componentes independentes**, que colaboram entre si através de contratos idealmente definidos em abstrações (em Java, interfaces ou classes abstratas). Com um **sistema de módulos** como o Java Module System (Java 9+) ou OSGi, é possível **reforçar as barreiras arquiteturais** mesmo com classes públicas, explicitando e limitando as dependências entre os módulos. Isso levaria a um **melhor gerenciamento das dependências** entre os módulos.

Já através de uma **Arquitetura de Plugins**, a base de código seria “fatiada” em módulos menores. Poderíamos ter **equipes mais enxutas**, focadas em cada módulo, trabalhando em várias **bases de código distintas**. A aplicação seria composta a partir de diferentes módulos em runtime. Cada desenvolvedor passaria a trabalhar com apenas uma parte do código, suavizando a IDE. Porém, subir a aplicação como um todo

ainda seria algo relativamente demorado, já que seria preciso colocar todos os módulos no ar. Os testes poderiam ser agilizados, concentrando-se em cada módulo separadamente, mas teria que ser feito um teste de sistema, que avaliaria a integração entre todos os módulos na aplicação.

Por meio de uma solução como **OSGi**, até seria possível **atualizar um módulo sem parar a aplicação** como um todo.

Monólitos Poliglotas

Um monólito não está restrito a apenas uma tecnologia de persistência, já que pode ter diferentes *data sources* para diferentes paradigmas: BDs relacionais, BDs orientados a documentos, BDs orientados a grafos, etc.

Com tecnologias como a [Graal VM](#), é possível criar monólitos desenvolvidos em múltiplas linguagens de diferentes plataformas: linguagens da JVM como Java, Scala, Kotlin, Groovy e Clojure; linguagens baseadas em LLVM como C e C++; e linguagens como JavaScript, Ruby, Python e R.

Uma definição de monólito

Como estudamos nesse capítulo, um monólito não é sinônimo de código mal feito. É possível implementar um monólito com código organizado, fatiado em pequenos pedaços independentes e alinhados com o negócio. Podemos até compor esses pedaços de diferentes maneiras e implementá-los de maneira poliglotas. Dependendo da tecnologia utilizada, podem ser atualizados parcialmente sem derrubar toda a aplicação. E podemos escalar um monólito, executando múltiplas instâncias por trás de um Load Balancer. Então o que define um monólito?

Para responder, precisamos pensar em qual é o “entregável” de uma aplicação: qual é o artefato que implantamos no ambiente de produção?

Na plataforma Java, há algum tempo, o artefato mais comum era um WAR implantado em um servlet container como Tomcat ou Jetty. Já com frameworks como o Spring Boot, temos um JAR com as dependências embutidas, um *fat JAR*. Em uma Arquitetura de Plugins, seja com Service Loader API ou com OSGi, temos uma coleção de JARs.

Em qualquer uma das maneiras de implantar um monólito, a comunicação entre as "fatias" do código é feita por chamadas de métodos, *in-memory*. E isso só é possível porque cada instância é executada em um mesmo processo.

MONÓLITO

Um monólito, modular ou não, é um sistema em que uma instância do back-end de toda a aplicação é executada em um mesmo processo do sistema operacional.

Para saber mais: Limitações dos Módulos Maven e JARs

Os módulos Maven ajudam a organizar o código de aplicações maiores porque fornecem uma maneira de representar a decomposição modular do domínio. Além disso, as dependências ficam materializadas nas declarações dos `pom.xml` de cada módulo.

Uma desvantagem dos módulos Maven como utilizamos no exemplo do exercício anterior é que **a base de código é uma só**. Diferentes times poderiam estar focados em módulos diferentes mas teriam acesso ao código dos outros módulos. A tentação de alterar algo de um módulo de outro time é muito forte! Para resolver isso, poderíamos implementar os módulos como bibliotecas completamente separadas. Um problema que iria surgir é como controlar a versão de cada módulo.

Outra desvantagem é que, em *runtime*, **a JVM não possui uma maneira de atualizar módulos** (JARs). Para atualizá-los, teríamos que parar a JVM e iniciá-la novamente, o que tornaria a aplicação indisponível. Num

ambiente com diversos times entregando software em taxas diferentes múltiplas vezes por dia/semana, a disponibilidade da aplicação seria terrivelmente afetada.

Ainda outra desvantagem dos módulos Maven é que **um módulo tem acesso às suas dependências transitivas**, ou seja, às dependências de suas dependências. Por exemplo, o módulo `eats-distancia` depende de `eats-restaurante` que, por sua vez, depende de `eats-administrativo`. Portanto, o módulo `eats-distancia` tem como dependência transitiva o módulo `eats-administrativo` e tem acesso a qualquer uma de suas classes públicas, como `FormaDePagamento` e `TipoDeCozinha`. Além disso, em `eats-distancia`, temos acesso a qualquer biblioteca declarada como dependência nos módulos `eats-restaurante` e `eats-administrativo`.

Essa fronteira fraca entre os módulos Maven pode ser problemática, já que leva a dependências indesejadas e não previstas. Se somarmos a isso o fato de que quase sempre definimos classes como públicas, módulos com muitas dependências transitivas terão acesso a boa parte das classes de outros módulos e às bibliotecas utilizadas por esses módulos. O código pode sair do controle.

Uma solução é limitar as classes públicas ao mínimo possível, tornando as classes acessíveis apenas ao pacote em que estão definidas. Mas para módulos mais complexos, teríamos dezenas ou centenas de classes no mesmo pacote! Observe, por exemplo, o módulo `eats-restaurante`: são 26 classes no mesmo pacote. Passa a ficar difícil de entender o código.

Os problemas do Classpath

Na verdade, há uma limitação nos JARs, que são apenas ZIPs com arquivos `.class` e de configuração organizados em diretórios (pacotes).

Uma vez que os JARs disponíveis são vasculhados e uma classe é carregada por um `ClassLoader` na JVM, perde-se o conceito de módulo.

O Classpath da JVM é apenas uma lista de classes, sem qualquer referência a seu JAR de origem ou de quais outros JARs dependem.

A ausência de algo que represente JAR de origem e suas dependências no Classpath enfraquece o encapsulamento em uma aplicação Java modularizada.

Nicolai Parlog, no livro [The Java Module System](#) (PARLOG, 2018), elenca os seguintes problemas no Classpath:

- *Encapsulamento fraco entre JARs*: conforme estudamos, o Classpath é uma grande lista de classes. As classes públicas são visíveis por quaisquer outras. Não é possível criar uma funcionalidade visível dentro de todos os pacotes de um JAR, mas não fora dele.
- *Ausência de representação das dependências entre JARs*: não há como um JAR declarar de quais outros JARs ele depende apenas com o Classpath.
- *Ausência de checagens automáticas de segurança*: o encapsulamento fraco dos JARs permite que código malicioso acesse e manipule funcionalidade crítica. Porém, é possível implementar manualmente checagens de segurança.
- *Sombreamento de classes com o mesmo nome*: no caso de JARs que definem duas classes com o mesmo nome, apenas uma delas é tornada disponível. E não é possível saber qual.
- *Conflitos entre versões diferentes do mesmo JAR*: duas versões do mesmo JAR no Classpath levam a comportamentos imprevisíveis.
- *JRE é rígida*: não é possível disponibilizar no Classpath um subconjunto das bibliotecas padrão do Java. Muitas classes não utilizadas ficam acessíveis.
- *Performance ruim no startup*: apesar dos class loaders serem *lazy*, carregando classes só no seu primeiro uso, muitas classes já são carregadas ao iniciar uma aplicação.
- *Class loading complexo*

Para saber mais: JPMS, um sistema de módulos para o Java

A partir do Java 9, o Java inclui um sistema de módulos bastante poderoso: o Java Platform Module System (JPMS).

Um módulo JPMS é um JAR que define:

- um nome único para o módulo
- dependências a outros módulos
- pacotes exportados, cujos tipos são acessíveis por outros módulos

Com um módulo JPMS, conseguimos definir **encapsulamento no nível de pacotes**, escolhendo quais pacotes são ou não exportados e, em consequência, acessíveis por outros módulos.

A JDK modularizada

Um dos grandes avanços do JPMS, disponível a partir da JDK 9, foi a modularização da própria plataforma Java.

O JPMS é resultado do projeto *Jigsaw*, criado em 2009, no início do desenvolvimento da JDK 7.

Antes do Java 9, todo o código das bibliotecas padrão da JDK ficava apenas no módulo de *runtime*: o `rt.jar`.

O estudo inicial do projeto *Jigsaw* agrupou o código já existente da JDK em diferentes módulos. Por exemplo, foi identificado um módulo base, que conteria pacotes fundamentais como o `java.lang` e `java.io`; um módulo desktop, com as bibliotecas Swing, AWT; além de módulos para APIs como Java Logging, JMX, JNDI.

A análise das dependências entre os módulos identificados na JDK 7 levou à descoberta de ciclos como: o módulo base depende de Logging que depende de JMX que depende de JNDI que depende de desktop que, por sua vez, depende do base.

O código da JDK 8 foi reorganizado para que não houvessem ciclos e dependências indevidas, mesmo que ainda sem um sistema de módulos propriamente dito. Os pacotes que pertenceriam ao módulo base não teriam mais dependências a nenhum outro módulo.

Na JDK 9, foram definidos módulos JPMS para cada parte do código da JDK:

- `java.base`, contendo código de pacotes como `java.lang`, `java.math`, `java.text`, `java.io`, `java.net` e `java.nio`.
- `java.logging`, com a Java Logging API.
- `java.management`, com a Java Managing Extensions (JMX) API.
- `java.naming`, com a Java Naming and Directory Interface (JNDI) API.
- `java.desktop`, contendo código de bibliotecas como Swing, AWT, 2D.

As dependências entre os módulos JPMS da JDK foram organizadas de maneira bem cuidadosa.

Antes da JDK 9, toda aplicação teria disponível todos os pacotes de todas as bibliotecas do Java. Não era possível depender de menos que a totalidade da JDK.

A partir da JDK 9, aplicações modularizadas com JPMS podem escolher, no arquivo `module-info.java`, de quais módulos da JDK dependerão.

Para saber mais: Módulos plugáveis

Antigamente, antes do Java SE 6, para ligar um plugin de uma aplicação a uma implementação era necessário:

- criar uma solução caseira usando a Reflection API
- usar bibliotecas como [JPF](#) ou [PF4J](#)
- usar uma especificação robusta, mas complexa, como [OSGi](#)

Porém, a partir do Java SE 6, a própria JRE contém uma solução: a

Service Loader API.

Na Service Loader API, um ponto de extensão é chamado de *service*.

Para provermos um service precisamos de:

- **Service Provider Interface (SPI)**: interfaces ou classes abstratas que definem a assinatura do ponto de extensão.
- **Service Provider**: uma implementação da SPI.

Para ligar a SPI com seu *service provider*, o JAR do provider precisa definir o *provider configuration file*: um arquivo com o nome da SPI dentro da pasta `META-INF/services`. O conteúdo desse arquivo deve ser o *fully qualified name* da classe de implementação.

No projeto que define a SPI, carregamos as implementações usando a classe `java.util.ServiceLoader`.

A classe `ServiceLoader` possui o método estático `load` que recebe uma SPI como parâmetro e, depois de vasculhar os diretórios `META-INF/services` dos JARs disponíveis no Classpath, retorna uma instância de `ServiceLoader` que contém todas as implementações.

O `ServiceLoader` é um `Iterable` e, por isso, pode ser percorrido com um `for-each`. Caso não haja nenhum service provider para a SPI, o `ServiceLoader` se comporta como uma lista vazia.

Perceba que uma mesma SPI pode ter vários service providers, o que traz bastante flexibilidade.

Uma Arquitetura de Plugins

Com o uso de SPIs e Service Providers, é possível criar uma Arquitetura de Plugins com a plataforma Java.

Com a Service Loader API, a simples presença de um `.jar` que a implemente a abstração do plugin (ou SPI) fará com que o

comportamento da aplicação seja estendido, sem precisarmos modificar nenhuma linha de código.

Em seu artigo [Microservices and Jars](#) (MARTIN, 2014), Uncle Bob escreve:

Não pule para Microservices só porque parece legal. Antes, segregue o sistema em JARs usando uma Arquitetura de Plugins. Se isso não for suficiente, considere a introdução de fronteiras entre serviços (service boundaries) em pontos estratégicos.

Várias bibliotecas das mais usadas por desenvolvedores Java usam SPIs.

Por meio da SPI `javax.persistence.spi.PersistenceProvider`, bibliotecas como o Hibernate e o EclipseLink fornecem implementações para as interfaces do pacote `javax.persistence`.

Do Java SE 6 em diante, a classe [DriverManager](#) carrega automaticamente todas as implementações da SPI `java.sql.Driver`. Por exemplo, o `mysql-connector-java.jar` do MySQL fornece um Service Provider para essa SPI.

O Spring tem a sua própria implementação de algo semelhante a Service Loader API: a classe [SpringFactoriesLoader](#). Essa classe é usada pelas Auto-Configurations do Spring Boot, fazendo com que a simples presença dos `.jar` dos starters adicionem comportamento à aplicação. Ou seja, o próprio Spring Boot é uma Arquitetura de Plugins.

Para saber mais: OSGi

O Java Module System foi disponibilizado a partir de 2017, com o lançamento do Java 9. Porém, a plataforma Java já tinha uma solução mais poderosa desde 1999: a especificação OSGi (Open Services Gateway Initiative). Essa especificação é implementada por frameworks como Apache Felix, Eclipse Equinox, entre outros. IDEs como Eclipse e NetBeans, servidores de aplicação como GlassFish e WebSphere, são

implementadas usando frameworks OSGi.

Por meio de diferentes Class Loaders, um framework OSGi traz a ideia de módulos para o *runtime* da JVM, corrigindo falhas do Classpath. Dessa maneira, provê um nível de encapsulamento além dos pacotes.

Um módulo, no OSGi, é chamado de *bundle*. Bundles são JARs, só que com metadados adicionais no `META-INF/MANIFEST.MF` como o nome do bundle, a versão, de quais outros bundles depende, entre outros detalhes.

Um framework OSGi controla o ciclo de vida de um bundle, fazendo com que seja instalado, iniciado, atualizado, parado e desinstalado. Múltiplas versões de um bundle podem coexistir em runtime e um bundle pode ser trocado sem parar toda a JVM.

O OSGi também especifica o conceito de *service*, análogo à Service Loader API do Java: um bundle define interface pública e outros bundles, uma ou mais implementações. Para ligar as implementações à interface, um framework OSGi provê um *service registry*. Novas implementações podem ter seu registro feito ou cancelado dinamicamente, sem parar a JVM. Os consumidores de um service dependeriam apenas da interface e do service registry, sem ter acesso a detalhes de implementação.

O nível de encapsulamento de um bundle e a possibilidade de **atualizar e registrar implementações dinamicamente** permitiria que diferentes times cuidassem de diferentes bundles alinhados com os Bounded Contexts (e as áreas de negócio) da organização. A implantação de novas versões de um bundle poderia ser feita sem derrubar a aplicação como um todo.

Porém, OSGi traz alguns problemas que limitaram sua adoção em aplicações e restringiram o uso basicamente a criadores de middleware e IDEs. Há quem diga que OSGi aumenta drasticamente o uso de memória, talvez pela implementação de diferentes Class Loaders, mas há soluções com baixo uso de memória, como de IoT, que usam implementações

OSGi. Ross Mason, criador do Mule ESB, escreve no artigo [OSGi? No Thanks](#) (MASON, 2010) que o principal dos problemas é a curva de aprendizado e a complexidade, com a necessidade de diversas configurações cheias de detalhes técnicos, o que afeta negativamente a experiência do desenvolvedor.

Um detalhe interessante é que Craig Walls, no livro [Modular Java](#) (WALLS, 2009), cita o termo *SOA in a JVM* como uma maneira usada para descrever os services do OSGi.

Um post sobre services OSGi de 2010, no blog da OSGi Alliance, usou pela primeira vez o termo [μServices](#) (KRIENS, 2010), com a letra grega mu (μ) que é usada como símbolo do prefixo *micro* pelo Sistema Internacional de Unidades.

No livro de 2012 [Java Application Architecture: Modularity Patterns](#) (KNOERNNSCHILD, 2012), Kirk Knoernschild usa repetidamente o termo μ Services para se referir aos services OSGi.

Extraindo serviços

O contexto que levou aos Microservices

Na década de 90, aplicações Desktop deram lugar à Web. A arquitetura Web, do estilo Cliente/Servidor com “telas” geradas pelo Servidor (*thin clients*), influenciou na maneira como o código é implantado. Com controle total dos servidores, publicações de novas versões da aplicação foram facilitadas.

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

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

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

As tecnologias de criação de máquinas virtuais permitiram o provisionamento (configuração) de máquinas virtuais por meio de

scripts, o que ficou conhecido como **infrastructure as code**. A partir de 2005, surgiram várias soluções do tipo como Puppet, Chef, Vagrant, Salt e Ansible.

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

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

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

Todo esse contexto é resumido por Sam Newman no início do livro [Building Microservices](#) (NEWMAN, 2015):

Domain-driven design. Continuous delivery. Virtualização sob demanda. Automação de infraestrutura. Times pequenos e autônomos. Sistemas em larga escala. Microservices emergiram desse mundo.

Do monólito (modular) aos Microservices

Um monólito comum, organizado com Package by Layer, pode trazer problemas para aplicações maiores: código progressivamente mais complexo, dependências indevidas, times cada vez maiores, impossibilidade de deploys sem parar a aplicação, entre outros.

Com uma estratégia de componentização baseada em módulos, a manutenção e evolução podem ser melhoradas. Com uma Arquitetura de Plugins, podemos ter pequenos times autônomos. Com runtimes como OSGi, podemos até fazer hot-deploy, atualizando módulos com a aplicação no ar. Com diferentes datasources, podemos explorar diferentes tecnologias de persistência. Com plataformas como a JVM, é possível o uso de linguagens de diversos paradigmas diferentes.

Mas, ainda assim, o monólito é executado como um único processo. Se houver alguma falha na memória ou bug que cause uso massivo de CPU ou um loop infinito, a aplicação toda sairá do ar.

Se houver um aumento na carga, é possível sim escalar um monólito: basta colocarmos um cluster de instâncias, com requests alternados por um Load Balancer. Porém, pode haver uma subutilização de recursos, já que a replicação será de toda a aplicação e não daquelas partes em que há mais necessidade de CPU ou memória. E, como o código será o mesmo, bugs que derrubam a aplicação poderão ser replicados por todos os nós do cluster.

Na palestra [Evoluindo uma Arquitetura inteiramente sobre APIs](#) (CALÇADO, 2013), Phil Calçado diz de maneira bem clara o grande problema de um monólito:

Quando você tem uma base de código só, você é tão estável quanto a sua parte menos estável.

E, convenhamos, um monólito modular é algo raríssimo no mercado. Então, para a maioria dos monólitos há problemas com times grandes, deploys e complexidade do código. Mas um monólito modular, com complexidade e dependências gerenciadas, facilitará uma possível migração para serviços.

Simon Brown diz na palestra [Modular monoliths](#) (BROWN, 2015):

Escolha Microservices por seus benefícios, não por que sua base de código monolítica é uma bagunça.

Componentização em serviços

Uma **Arquitetura de Microservices** traz uma abordagem diferente de componentização: a aplicação é decomposta em diversos **serviços**.

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

Microservices:

- são executados em diferentes processos em máquinas (ou containers) distintos
- a comunicação é interprocessos, pela rede
- o deploy é independente, por definição
- podem usar múltiplos mecanismos de persistência em paradigmas diversos (Relacional, NoSQL, etc), desde que não compartilhados com outros serviços
- há diversidade tecnológica, sendo a única restrição a possibilidade de prover uma API em um protocolo padrão (HTTP, AMQP, etc)
- há uma fronteira fortíssima entre as bases de código

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), define Microservices da seguinte maneira:

Microservices são serviços pequenos e autônomos que trabalham juntos.

Prós e contras de uma Arquitetura de Microservices

PRÓ: Times Pequenos e Autônomos

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

afeta positivamente a velocidade de desenvolvimento de novas funcionalidades.

Uma monólito com uma Arquitetura de Plugins talvez permita o desenvolvimento de diferentes módulos por times diversos em torno de um núcleo comum. Porém, é uma raridade no mercado.

PRÓ: “Trocabilidade”

É comum termos aquele sistema legado em que ninguém toca e que ninguém sabe dar manutenção. E isso acontece porque qualquer mudança ficou muito arriscada.

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

É algo que um monólito modular também permitiria.

PRÓ: Reuso e composibilidade

Diferentes serviços podem ser compostos em novos serviços, atendendo com agilidade às demandas do negócio. É uma velha promessa do SOA (Service-Oriented Architecture).

A Uber é um exemplo disso: provê um serviço de transporte urbano, mas lançou há algum tempo um serviço de fretes de caminhões e outro de entrega de comida. Provavelmente, reutilizaram serviços de pagamentos, de geolocalização, entre outros.

Um monólito modular é uma outra maneira de atingir isso, sem termos um Sistema Distribuído.

PRÓ: Fronteiras fortes entre componentes

Se usarmos serviços como estratégia de componentização, ao invés de simples pacotes ou módulos, teremos uma separação fortíssima entre o código de cada componente.

Em um monólito não modular é tentador tomar atalhos para entregar funcionalidades mais rápido, esquecendo das fronteiras entre componentes. Mesmo em monólitos modulares, dependendo do sistema de módulos utilizado, é possível acessar em *runtime* (e até via código) funcionalidades de outros módulos, talvez por meio de *workarounds* (as famosas gambiarras). Porém, há sistemas de módulos como o Java Module System do Java 9+ e o OSGi, que reforçaram as barreiras entre código de módulos diferentes tanto em desenvolvimento como em *runtime*.

Uma vantagem de uma Arquitetura de Microservices é que temos esse fronteira fortes entre componentes mantendo a estrutura de cada serviço parecida com a que estamos acostumados. É possível usar o bom e velho *Package by Layer*, já que o código de cada serviço é focado em um conjunto específico de funcionalidades.

Um ponto de atenção é o acesso a dados. Em uma Arquitetura de Microservices, cada serviço tem o seu BD separado. E isso reforça bastante a componentização dos dados, eliminando os perigos de uma integração pelo BD que pode levar a um acomplamento indesejado. Por outro lado, ao separar os BDs, perdemos muitas coisas, que discutiremos adiante.

PRÓ: Diversidade tecnológica e experimentação

Em uma aplicação monolítica, as escolhas tecnológicas iniciais restringem as linguagens e frameworks que podem ser usados.

Com Microservices, partes do sistema podem ser implementadas em tecnologias que estejam mais de acordo com o problema a ser resolvido. Os protocolos de integração entre os serviços passam a ser as partes mais importantes das escolhas tecnológicas.

Essa heterogeneidade tecnológica permite que soluções performáticas e com mais funcionalidades sejam utilizadas.

Em seu artigo [Microservice Trade-Offs](#) (FOWLER, 2015a), Martin Fowler diz que coisas prosaicas como atualizar a versão de uma biblioteca podem ser facilitadas. Em um monólito, temos que usar a mesma versão para todo a aplicação e os upgrades podem ser problemáticos. Ou tudo é atualizado, ou nada. Quanto maior a base de código, maior é o problema nas atualizações de bibliotecas.

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), argumenta: *Uma das grandes barreiras para testar e adotar novas tecnologias são os riscos associados. Em um monólito, qualquer mudança impactará uma quantia grande do sistema. Com um sistema que consiste de múltiplos serviços, há múltiplos lugares para testar novas tecnologias. Um serviço de baixo risco pode ser usado para minimizar os riscos e limitar os possíveis impactos negativos. A habilidade de absorver novas tecnologias traz vantagens competitivas para as organizações.*

Porém, é importante ressaltar o risco de adotar muitas stacks de tecnologias completamente distintas: é difícil de entender as características de performance, confiabilidade, operações e monitoramento. Por isso, no livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman diz que empresas como a Netflix e Twitter focam boa parte dos seus esforços em usar a JVM como uma plataforma para diferentes linguagens e tecnologias. Para empresas menores, o risco de adotar tecnologias "esotéricas" é aumentado, já que pode ser difícil de contratar pessoas experientes e familiarizadas com algumas tecnologias.

É possível implementar um monólito poliglota. Mas é algo raro. No caso da JVM, podemos usar linguagens como Java, Kotlin, Scala e Clojure. Com a GraalVM, podemos até mesclar plataformas, usando algumas linguagens da JVM com algumas da LLVM, entre outras. E múltiplos datasources podem permitir o uso de mecanismos de persistências diferentes, como um BD orientado a Grafos, como o Neo4J, junto a um

BD relacional, como o PostgreSQL.

PRÓ: Deploy independente

A mudança de uma linha de código em uma aplicação monolítica de um milhão de linhas requer que toda a aplicação seja implantada para que seja feito o *release* da pequena alteração. Isso leva a deploys de alto risco e alto impacto, o que leva a medo de que alguma coisa dê errado. E esse medo leva a diminuir a frequência dos deploys, o que leva ao acúmulo de mudanças em cada deploy. E quanto mais alterações em um mesmo deploy, maior o risco de que algo dê errado. Esse *ciclo vicioso dos deploys* é demonstrado por Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015).

Com Microservices, só um pedaço do sistema fica fora do ar ao implantarmos novas versões. Isso minimiza o risco e o impacto de cada deploy, já que uma falha de um serviço e diminuiu a indisponibilidade da aplicação. A consequência é que podemos passar a fazer mais deploys em produção, talvez várias vezes por dia. Ou seja, serviços habilitam a entrega rápida, frequente, confiável de aplicações complexas.

Essa Entrega Contínua (*Continuous Delivery*, em inglês) permite que os negócios da organização reajam rápido ao feedback do cliente, a novas oportunidades e a concorrentes. A mudança cultural e organizacional para permitir isso é um dos temas do DevOps.

No artigo [Microservice Trade-Offs](#) (FOWLER, 2015a), Martin Fowler argumenta que a relação entre Microservices e Continuous Delivery/DevOps é de duas vias: para ter vários Microservices, provisionar máquinas e implantar aplicações rapidamente são pré-requisitos. Fowler ainda cita Neal Ford, que relaciona uma Arquitetura de Microservices e DevOps: *Microservices são a primeira arquitetura depois da revolução trazida pelo DevOps.*

Atingir algo parecido com um monólito é até possível com algumas tecnologias como OSGi, mas incomum. Martin Fowler diz, ainda no artigo

[Microservice Trade-Offs](#) (FOWLER, 2015a), que Facebook e Etsy são dois casos de empresas cujos monólitos têm Continuous Delivery. O autor ainda diz que entregas rápidas, confiáveis e frequentes são mais relacionadas com o uso prático de Modularidade do que necessariamente com Microservices.

PRÓ: Maior isolamento de falhas

Em um monólito, se uma parte da aplicação apresentar um vazamento de memória, pode ser que o todo seja interrompido.

Quando há uma falha ou indisponibilidade em um serviço, os outros serviços continuam no ar e, portanto, parte da aplicação ainda permanece disponível e utilizável, o que alguns chamam de *graceful degradation*.

PRÓ: Escalabilidade independente

Em um monólito, componentes que tem necessidades de recursos computacionais completamente diferentes devem ser implantados em conjunto, isso leva a um desperdício de recursos. Se uma pequena parte usa muita CPU, por exemplo, estamos restritos a escalar o todo para atender às demandas de processamento.

Com Microservices, necessidades diferentes em termos computacionais, como processamentos intensivos em termos de memória e/ou CPU, podem ter recursos específicos. Isso minimiza o impacto em outras partes da aplicação, otimiza recursos e diminui custos de operação. Quando são usados provedores de Cloud como AWS, Azure ou Google Cloud, isso levará a um corte de custos quase imediato.

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), cita o caso da Gilt, uma loja online de roupas. Começaram com um monólito Rails em 2007 que, já em 2009, não estava suportando a carga. Ao quebrar partes do sistema, a Gilt conseguiu lidar melhor com picos de tráfego.

CONTRA: Dificuldades inerentes a um Sistema Distribuído

Uma chamada entre dois Microservices envolve a rede. Uma Arquitetura de Microservices é um Sistema Distribuído.

A comunicação intraprocesso, com as chamadas em memória, é centenas de milhares de vezes mais rápida que uma chamada interprocessos dentro de um mesmo data center. Algumas das latências mostradas pelo pesquisador da Google Jeffrey Dean na palestra [Designs, Lessons and Advice from Building Large Distributed Systems](#) (DEAN, 2009):

- Referência Cache L1: 0.5 ns
- Referência Cache L2: 7 ns
- Referência à Memória Principal: 100 ns
- Round trip dentro do mesmo data center: 500 000 ns (0,5 ms)
- Roud trip Califórnia-Holanda: 150 000 000 ns (150 ms)

Uma versão mais atualizada e interativa dessa tabela pode ser encontrada em:

https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Leslie Lamport, pioneiro da teoria de Sistemas Distribuídos, brincou com a definição de Sistemas Distribuídas em uma [lista interna](#) (LAMPORT, 1987) da DEC Systems Research Center: *Um sistema distribuído é um sistema em que uma falha em um computador que você nem sabia da existência torna o seu próprio computador inutilizável.*

A performance é afetada negativamente. É preciso tomar cuidado com latência, limites de banda, falhas na rede, indisponibilidade de outros serviços, entre outros problemas. Além disso, transações distribuídas são um problema muito complexo.

A rede é lenta e instável e temos que lidar com as consequências disso.

As faláciais da Computação Distribuída

Peter Deustch e seus colegas da Sun Microsystems definiram algumas premissas falsas que desenvolvedores assumem quando tratam de aplicações distribuídas:

1. A rede é confiável
2. A latência é zero
3. A banda é infinita
4. A rede é segura
5. A topologia não muda
6. Só há um administrador
7. O custo de transporte é zero
8. A rede é homogênea

E a Primeira Lei do Design de Objetos Distribuídos?

No livro [Patterns of Enterprise Application Architecture](#) (FOWLER, 2002), Martin Fowler cunhou a *Primeira Lei do Design de Objetos Distribuídos*: não distribua seus objetos.

Isso valeria para uma Arquitetura de Microservices? Fowler responde a essa pergunta no artigo [Microservices and the First Law of Distributed Objects](#) (FOWLER, 2014a). O autor explica que o contexto da "lei" era a ideia, em voga no final dos anos 90 e no início dos anos 2000, de era possível tratar objetos intraprocesso e remotos de maneira transparente. Com o uso de CORBA, DCOM ou RMI, bastaria rodar objetos em outras máquinas, sem a necessidade de estratégias elaboradas de decomposição e migração. Considerando que a rede é lenta e instável, buscar preços de 100 produtos não teria a mesma performance e confiabilidade comparando uma chamada em memória e uma pela rede. É preciso tomar cuidado com a granularidade das chamadas e tolerância a falhas. A suposta transparência entre chamadas em memória e remotas é uma falácia tardia. Por isso, a lei proposta no livro mencionado.

Fowler, no mesmo artigo, diz que os defensores dos Microservices com que teve contato estão cientes da distinção entre chamadas em memória e pela rede e desconsideraram sua suposta transparência. As interações

entre Microservices, portanto, seriam de granularidade mais grossa e usariam técnicas como Mensageria.

Apesar de ter uma inclinação para Monólitos, Fowler diz que sua natureza de empiricista o fez aceitar que uma abordagem de Microservices teve sucesso em vários times com que trabalhou. Porém, enquanto é mais fácil pensar sobre Microservices pequenos, o autor diz preocupar-se que a complexidade é empurrada para as interações entre os serviços, onde é menos explícita, o que torna mais difícil descobrir quando algo dá errado.

Essa preocupação ressoa em Michael Feathers que, no post [Microservices Until Macro Complexity](#) (FEATHERS, 2014), diz que parece haver uma Lei da Conservação da Complexidade no software:

"Quando quebramos coisas grandes em pequenos pedaços nós passamos a complexidade para a interação entre elas."

CONTRA: Complexidade ao operar e monitorar

Configurar, fazer deploy e monitorar um monólito é fácil. Depois de gerar o entregável (WAR, JAR, etc) e configurar portas e endereços de BDs, basta replicar o artefato em diferentes servidores. O sistema está ou não fora do ar, os logs ficam apenas em uma máquina e sabemos claramente por onde uma requisição passou.

Em uma Arquitetura de Microservices, precisamos:

- agregar logs que ficam espalhados pelos diversos Microservices
- saber da “saúde” de cada um dos Microservices
- rastrear por quais Microservices passa uma requisição
- ter uma maneira de facilitar a configuração de portas e endereços de BDs e de outros Microservices
- fazer deploy dos diferentes Microservices

Já no post [Microservice Prerequisites](#) (FOWLER, 2014b), Martin Fowler descrever alguns pré-requisitos para a adoção de uma Arquitetura de

Microservices:

- **provisionamento rápido:** preparar novos servidores com os softwares, dados e configurações necessários deve ser rápido e o mais automatizado o possível. Provedores e ferramentas de Cloud ajudam muito nessa tarefa.
- **deploy rápido:** fazer o deploy da aplicação em ambientes de teste e produção deve ser algo rápido e automatizado.
- **monitoramento básico:** detectar indisponibilidade de serviços, erros e acompanhar métricas de negócio é essencial
- **cultura DevOps:** é necessária uma mudança cultural em direção a uma maior colaboração entre desenvolvedores e pessoal de infra

Se Continuous Delivery é uma prática importante para monólitos, torna-se essencial para uma Arquitetura de Microservices. Ferramentas de automação de infra-estrutura são imprescindíveis.

James Lewis diz, no [podcast SE Radio](#) (LEWIS, 2014), que:

"Nós estamos mudando a COMPLEXIDADE ACIDENTAL de dentro da aplicação para a infraestrutura. AGORA é uma boa hora para isso porque nós temos mais maneiras de gerenciar a complexidade. Infraestrutura programável, automação, tudo indo pra cloud. Nós temos ferramentas melhores para resolver esse problemas AGORA."

Complexidade Essencial x Complexidade Acidental

No clássico artigo [No Silver Bullets](#) (BROOKS, 1986), Fred Brooks separa complexidades essenciais do software, que tem a ver com o problema que está sendo resolvido (e, poderíamos dizer, com o domínio) de complexidades acidentais, que são reflexos das escolhas tecnológicas. O autor argumenta que mesmo que as complexidades acidentais fossem zero, ainda não teríamos um ganho significativo no esforço de produzir um software. Por isso, **não existe bala de prata**.

CONTRA: Perda da consistência dos dados e transações

Manter uma consistência forte dos dados em um Sistema Distribuído é extremamente difícil.

De acordo com o Teorema CAP, cunhado por Eric Brewer na publicação [Towards Robust Distributed Systems](#) (BREWER, 2000), não é possível termos simultaneamente mais que duas das seguintes características: Consistência dos dados, Disponibilidade (*Availability*, em inglês) e tolerância a Partições de rede. Ou seja, se a rede falhar, temos que escolher entre Consistência e Disponibilidade. Se escolhermos Consistência, o sistema ficará indisponível até a falha na rede ser resolvida. Se escolhermos Disponibilidade, a Consistência será sacrificada. Portanto, em um Sistema Distribuído, não temos garantias ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Em um esperto jogo de palavras com os conceitos da Química de ácido e base, Brewer diz que poderíamos ter garantias BASE (Basically Available, Soft-state, Eventually consistent): para manter um Sistema Distribuído disponível, teríamos respostas aproximadas que eventualmente ficariam consistentes.

Daniel Abadi, no paper [Consistency Tradeoffs in Modern Distributed Database System Design](#) (ABADI, 2012), cunhou o Teorema PACELC, incluindo alta latência de rede como uma forma de indisponibilidade.

No artigo [Microservice Trade-Offs](#) (FOWLER, 2015a), Martin Fowler descreve o seguinte cenário: você faz uma atualização de algo e, ao recarregar a página, a atualização não está lá. Depois de alguns minutos, você dá refresh novamente a atualização aparece. Talvez isso acontece porque a atualização foi feita em um nó do cluster mas o segundo request obteve os dados de outro nó. Eventualmente, os nós ficam consistentes, com os mesmos dados. O autor pondera que inconsistências como essa são irritantes, mas podem ser catastróficas para a Organização quando decisões de negócios são feitas com base em dados inconsistentes. E o pior: é muito difícil de reproduzir e debugar!

Em um monólito, é possível alterar vários dados no BD de maneira

consistente, usando apenas uma transação. Como cada Microservice tem o seu BD, as transações teriam que ser distribuídas, o que iria na direção da Consistência em detrimento da Disponibilidade. O mundo dos Microservices abraça a consistência eventual (em inglês, *eventual consistency*). Processos de negócio são relativamente tolerantes a pequenas inconsistências momentâneas.

CONTRA: Saber o momento correto de adoção é difícil

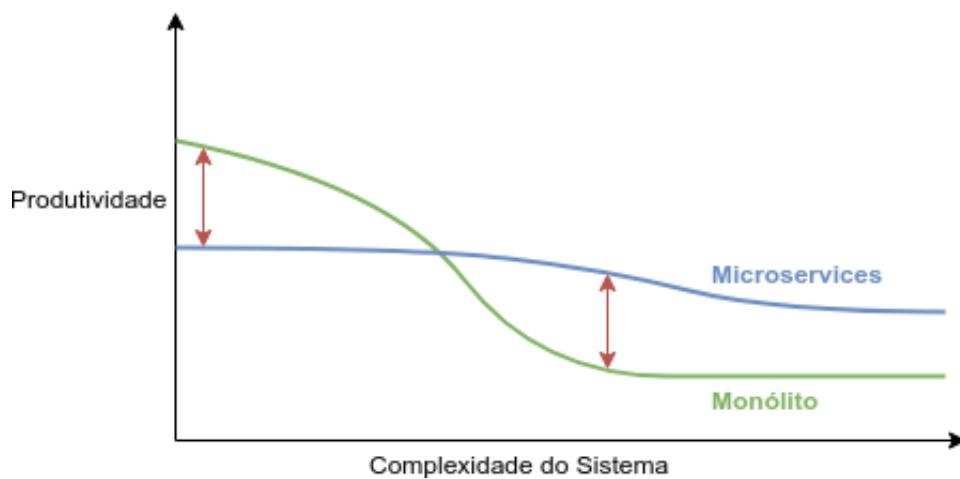
Encontrar fatias pequenas e independentes do domínio, criando fronteiras arquiteturais alinhadas com os Bounded Contexts, é difícil no começo do projeto, quando não se conhece claramente o Negócio ou as possíveis alterações. Isso é especialmente difícil para startups, que ainda estão validando o Modelo de Negócio e fazem mudanças drásticas com frequência. Aliando-se a isso as complexidade de operação, monitoramento e os desafios de um Sistema Distribuído, uma Arquitetura de Microservices pode ser uma escolha ruim para uma startup que tem uma base de usuário limitada, uma equipe reduzida e pouco financiamento.

Do ponto de vista Lean, usar uma Arquitetura de Microservices para um projeto simples ou em fase inicial pode ser considerado *overengineering*, um tipo de desperdício (Muda).

No artigo [Microservice Premium](#) (FOWLER, 2015b), Martin Fowler argumenta que uma Arquitetura de Microservices introduz uma complexidade que pode elevar os custos e riscos do projeto, como se fosse adicionado um ágio (em inglês, *premium*). O autor diz que, para projetos de baixa complexidade (essenciais, poderíamos dizer), uma Arquitetura de Microservices adiciona uma série de complicações no monitoramento, em como lidar com falhas e consistência eventual, entre outras. Já para projetos mais complexos, com um time muito grande, muitos modelos de interação com o usuário, dificuldade em escalar, partes do negócio que evoluem independentemente ou uma base de código gigantesca, vale pensar em uma Arquitetura de Microservices.

Martin Fowler conclui:

Minha principal orientação seria nem considerar Microservices, a menos que você tenha um sistema complexo demais para gerenciar como um Monólito. A maioria dos sistemas de software deve ser construída como uma única aplicação monolítica. Atenção deve ser prestada à uma boa modularização do Monólito (...) Se você puder manter o seu sistema simples o suficiente para evitar a necessidade de Microservices: faça.



Começar com um Monólito ou com Microservices?

No artigo [Monolith First](#) (FOWLER, 2015c), Martin Fowler argumenta que devemos começar com um Monólito, mesmo se você tiver certeza que a aplicação será grande e complexa o bastante para compensar o uso de Microservices. Fowler baseia o argumento em sua experiência:

Quase todas as histórias de sucesso de Microservices começaram com um Monólito que ficou muito grande e foi decomposto. Quase todos os casos de sistemas que começaram com Microservices do zero, terminaram em sérios apuros. (...) Até arquitetos experientes trabalhando em domínios familiares tem grandes dificuldades em acertar quais são as fronteiras estáveis entre serviços.

Stefan Tilkov publicou o artigo [Don't start with a monolith](#) (TILKOV, 2015) no próprio site de Martin Fowler, argumentando o contrário: é incrivelmente difícil, senão impossível, fatiar um monólito. Pra Tilkov, o que é necessário na verdade é um bom conhecimento sobre o domínio

da aplicação antes começar a particioná-lo. Outro argumento é que um Monólito bem componentizado e com baixo acoplamento é raríssimo e que uma das maiores vantagens dos Microservices é a fronteira fortíssima entre o código de cada serviço, evitando um emaranhado nas dependências. Partes do monólito comunicam entre si usando as mesmas bibliotecas, usam o mesmo modelo de persistência, podem usar transações no BD e muitas vezes compartilham objetos de domínio. Tudo isso dificulta imensamente uma possível migração posterior para uma Arquitetura de Microservices. Para o autor, em sistemas que sabe-se que serão grandes, complexos e em que o domínio é familiar, vale a pena começar a construí-los em subsistemas da maneira mais independente o possível.

Um outro argumento a favor do uso inicial de uma Arquitetura de Microservices é que, se as ferramentas de deploy, configuração e monitoramento são complexas, devemos começar a dominá-las o mais cedo o possível. Claro, se a visão é que o projeto crescerá em tamanho e complexidade.

Quão micro deve ser um Microservice?

Os serviços em uma Arquitetura de Microservices devem ser pequenos. Por isso, o “micro” no nome. Mas o que deve ser considerado “micro”? Algo menor que um miliservice ou maior que um nanoservice (termo infelizmente usado pelo mercado)? Não! O tamanho não é importante! O termo “micro” é enganoso.

Chris Richardson, no livro [Microservice Patterns](#) (RICHARDSON, 2018a) diz:

Um problema com o termo Microservice é que a primeira coisa que você ouve é micro. Isso sugere que um serviço deve ser muito pequeno. (...) Na realidade, tamanho não é uma métrica útil. Um objetivo melhor é definir um serviço bem modelado como um serviço capaz de ser desenvolvido por um time pequeno com um lead time mínimo e com mínima colaboração com outros times. Na teoria, um time deve ser

responsável somente por um serviço (...) Por outro lado, se um serviço requer um time grande ou leva muito tempo para ser testado, provavelmente faz sentido dividir o time e o serviço.

O critério para decomposição deve ser, em geral, algo alinhado com o negócio da organização. No fim das contas, o objetivo principal é alinhar negócio à TI. Um serviço pequeno é um serviço que embarca uma capacidade de negócio.

Os conceitos de Agregado e Contexto Delimitado do DDD, que vimos no capítulo anterior, vêm à nossa ajuda!

Um Microservice pode ser modelado como um Agregado ou, preferencialmente, como um Contexto Delimitado (Bounded Context) em que a linguagem do especialista de domínio será representada no código (Domain Model) sem apresentar inconsistências.

No livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman diz que devemos focar as fronteiras entre os serviços nas fronteiras do negócio. Dessa maneira, saberemos onde estará o código de uma determinada funcionalidade e evitaremos a tentação de deixar um determinado serviço crescer demais. Ao modelar de acordo com o negócio, as fronteiras ficam claras.

Phil Calçado, em [um tweet](#) (CALÇADO, 2018), diz que o critério de decomposição de uma Arquitetura de Microservices deve ser parecido com o de um monólito modular:

Eu sempre descrevo Microservices como a aplicação da mesma maneira de agrupar que você teria em uma aplicação maior, só que através de seus componentes distribuídos.

Cuidado com o Monólito Distribuído

No livro [Monolith to Microservices](#) (NEWMAN, 2019), Sam Newman define um Monólito Distribuído como um sistema que consiste de múltiplos serviços mas cujos deploys devem ser feitos ao mesmo tempo.

Na experiência do autor, um Monólito Distribuído tem todas as desvantagens de um Sistema Distribuído e todas as desvantagens de um Monólito. Para Newman, um Monólito Distribuído emerge de um ambiente em que não houve foco o suficiente em conceitos como *Information Hiding* e coesão das funcionalidades de negócio, levando a arquiteturas altamente acopladas em que mudanças se propagam através dos limites de serviço e onde mudanças aparentemente inocentes, que parecem ter escopo local, quebram outras partes do sistema.

Uma maneira comum de chegar a um Monólito Distribuído é ter serviços extremamente pequenos, chegando a um serviço por Entidade de Negócio (objetos que tem continuidade, identidade e estão representados em algum mecanismo de persistência).

Chris Richardson, no livro [Microservice Patterns](#) (RICHARDSON, 2018a), chega uma conclusão semelhante: um Monólito Distribuído é o resultado de uma decomposição incorreta dos componentes. Para Richardson, o antídoto aos Monólitos Distribuídos é seguir, só que no nível de serviços, o Common Closure Principle definido por Robert "Uncle Bob" Martin: *Agregue, em componentes, classes que mudam ao mesmo tempo e pelos mesmos motivos. Separe em componentes diferentes classes que mudam em momentos diferentes e por razões distintas.*

Microservices e SOA

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

Chris Richardson, em seu livro [Microservices Patterns](#) (RICHARDSON, 2018a), descreve SOA como sendo uma arquitetura que usa *smart pipes* como ESB, protocolos pesados como SOAP e WS-*, Persistência

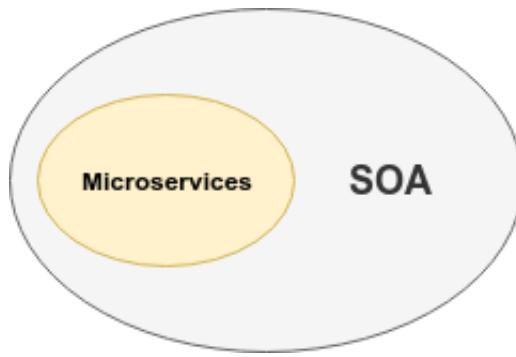
centralizada em BDs corporativos e serviços de granularidade grossa. Talvez seja a versão mais comum de SOA que vemos implementada nas organizações.

Martin Fowler e James Lewis dizem em seu artigo sobre [Microservices](#) (FOWLER; LEWIS, 2014), que há uma grande ambiguidade sobre o que SOA realmente é e, dependendo da definição, uma Arquitetura de Microservices é SOA, mas pode não ser. Talvez seria “SOA do jeito certo”. Algumas características a distinguem do SOA implementado em grandes organizações: governança e gerenciamento de dados descentralizado; mais inteligência nos Microservices (*smart endpoints*) e menos nos canais de comunicação (*dumb pipes*). Um ESB seria um *smart pipe*, já que faz roteamento de mensagens, transformações, orquestração e até algumas regras de negócio. Ainda citam em um rodapé que a Netflix, uma das referências em Microservices, inicialmente chamava sua abordagem de *fine-grained SOA*.

Henrique Lobo mostra em seu artigo [Repensando micro serviços](#) que SOA como descrito pelo consórcio de padrões abertos [OASIS] ([https://en.wikipedia.org/wiki/OASIS_\(organization\)](https://en.wikipedia.org/wiki/OASIS_(organization))) é muito parecido com o espírito dos Microservices.

Sam Newman, em seu livro [Building Microservices](#) (NEWMAN, 2015), reconhece que SOA trouxe boas ideias, mas que houve uma falta de consenso em como fazer SOA bem e dificuldade em ter uma narrativa alternativa à dos vendedores. SOA passou a ser visto como uma coleção de ferramentas e não como uma abordagem arquitetural. Ainda fala que uma Arquitetura de Microservices está para SOA assim como XP e Scrum estão para Agile: uma abordagem específica que veio de projetos reais.

Microservices são, então, uma abordagem para SOA.



Microservices e a Cloud

Os 12 fatores do Heroku

Um dos fundadores da plataforma de Cloud Heroku, Adam Wiggins, escreveu em 2011 um texto em que descreve soluções comuns para aplicações Web do tipo SaaS (Software as a Service) que rodam em plataformas de Cloud Computing. Essas soluções foram coletadas a partir da experiência em desenvolver, operar e escalar milhares de aplicações. É o que o autor chamou de [Os 12 Fatores](#) (WIGGINS, 2011):

1. *Base de Código*: há só uma base de código para a aplicação, rastreada em um sistema de controle de versão como Git e que pode gerar diferentes deploys (desenvolvimento, testes, produção).
2. *Dependências*: todas as bibliotecas e frameworks usados pela aplicação devem ser declarados explicitamente como dependências. As dependências são isoladas, impedindo que dependências implícitas vazem a partir do ambiente de execução.
3. *Configurações*: tudo que varia entre deploys, como credenciais de BDs e de serviços externos como Amazon S3, deve estar separado do código da aplicação. Essas configurações devem ser armazenadas em variáveis de ambiente, que são uma maneira multiplataforma e não ficarão na base de código.
4. *Backing services*: não há distinção entre um BD local ou serviço externo como New Relic (usado para métricas). Todos são recursos acessíveis pela rede e cujas URL e credenciais estão nas configurações, e não no código.
5. *Build, release, run*: há 3 estágios distintos para transformar código

em um deploy. O estágio de *build* converte um repositório de código em um executável, contendo todas as dependências. O estágio de *release* combina um executável com as configurações de um deploy, gerando um release imutável e com um timestamp. O estágio de *run* executa um release em um ou mais processos. O build é iniciado quando há novo código. O run pode ser iniciado automaticamente, por exemplo, em um reboot do servidor.

6. *Processos*: devem ser *stateless*. Dados de sessão deve estar em um *datastore* que tem expiração, como o Redis. Nunca deve ser assumido que a memória ou o disco estarão disponíveis em um próximo *request*.
7. *Port binding*: a aplicação é auto-contida e expõe a si mesma por meio de uma porta HTTP. Não há a necessidade de um servidor Web ou servidor de aplicação. Uma camada de roteamento repassa um *hostname* público para a porta HTTP da aplicação.
8. *Concorrência*: deve ser possível escalar a aplicação horizontalmente, replicando múltiplas instâncias idênticas que recebem requests de um *load balancer*. Podem existir processos web, que tratam de um request HTTP e processos *worker*, que cuidam de tarefas que demoram mais.
9. *Descartabilidade*: processos são descartáveis e são iniciados e parados a qualquer momento. O tempo de *startup* deve ser minimizado. Para um processo web, todos requests HTTP devem ser finalizados antes de parar. Para um processo worker, o job deve ser retornado à fila. Os processos devem considerar falhas de hardware e lidar com paradas inesperadas.
10. *Paridade dev/prod*: não devem ser acumuladas semanas de trabalho entre deploys em produção. Os desenvolvedores que escrevem código devem estar envolvidos na implantação e monitoramento em produção. As ferramentas de desenvolvimento devem ser semelhantes às de produção.
11. *Logs*: devem ser tratados como eventos que fluem continuamente enquanto a aplicação estiver no ar. Não devem ser armazenados em arquivos. O ambiente de execução cuida de rotear os logs para

ferramentas de análise como Splunk.

12. *Processos de Admin*: tarefas de administração, como scripts que são executados apenas uma vez, devem ser executados em um ambiente idêntico aos processos worker. O código dessas tarefas pontuais deve estar junto ao código da aplicação.

Cloud Native

No workshop [Patterns for Continuous Delivery, High Availability, DevOps & Cloud Native Open Source with NetflixOSS](#) (COCKCROFT, 2013), Adrian Cockcroft, um dos responsáveis pela migração da Netflix para Cloud iniciada em 2009, conta como seu time uniu patterns de sucesso no que começou a ser referida como arquitetura **Cloud Native**. A agilidade nos negócios, produtividade dos desenvolvedores e melhora na Escalabilidade e Disponibilidade são resultados da adoção de Continuous Delivery, DevOps, Open Source, Microservices, dados desnormalizados (NoSQL) e Cloud Computing com data centers globais. Isso permitiu que a Netflix atendesse a um crescimento exponencial no número de usuários. Algumas das ferramentas desenvolvidas na Netflix tiveram seu código aberto, numa plataforma chamada [Netflix OSS](#).

Em 2015, foi criada a [Cloud Native Computing Foundation](#) (CNCF) a partir da Linux Foundation, visando manter projetos open-source de ferramentas Cloud Native de maneira a evitar *vendor lock-in*. Entre os projetos mantidos pela CNCF estão orquestradores de containers como Kubernetes, proxys como o Envoy e ferramentas de monitoramento como o Prometheus. Entre as empresas que participam da CNCF estão Google, Amazon, Microsoft, Alibaba, Baidu, totalizando US\$ 13 trilhões de valor de mercado, em números de 2019.

Segundo a [definição da CNCF](#) (CNCF TOC, 2018):

Tecnologias Cloud Native empoderam organizações a construir e rodar aplicações escaláveis em ambientes dinâmicos e modernos como Clouds públicas, privadas ou híbridas. Containers, Service Meshes, Microservices, infraestrutura imutável e APIs declarativas são exemplos

dessa abordagem. Essas técnicas permitem sistemas baixamente acoplados, que são resilientes, gerenciáveis e observáveis. Combinadas a automação robusta, permitem que os engenheiros façam mudanças de grande impacto frequentemente e de maneira previsível, com o mínimo de esforço.

Microservices chassis

Há preocupações comuns em uma Arquitetura de Microservices:

- Configuração externalizada
- Health checks
- Métricas de aplicação
- Service discovery
- Circuit breakers
- Distributed tracing

Observação: várias dessas necessidades serão abordadas nos próximos capítulos.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a), chama de **Microservices chassis** um framework ou conjunto de bibliotecas que tratam dessas questões transversais.

Pattern: Microservice chassis

Construa serviços usando um framework ou coleção de frameworks que lida com questões transversais como rastreamento de exceções, logging, health checks, configuração externalizada e distributed tracing.

Frameworks Java como [Dropwizard](#) e [Spring Boot](#) são exemplos de Microservices chassis.

O [Spring Cloud](#) é um conjunto de ferramentas que expandem as capacidades do Spring Boot e oferecem implementações para patterns comuns de sistemas distribuídos. Há, por exemplo, o [Spring Cloud](#)

[Netflix](#), que integra vários componentes da Netflix OSS com o ecossistema do Spring. Estudaremos várias das ferramentas do Spring Cloud durante o curso.

No Java EE (ou Jakarta EE), foi criada a especificação MicroProfile, um conjunto de especificações com foco que servem como um Microservices chassis. Entre as implementações estão: [KumuluzEE](#), [Wildfly Swarm/Thorntail](#), baseado no Wildfly da JBoss/Red Hat, [Open Liberty](#), baseado no WebSphere Liberty da IBM, [Payara Micro](#), baseado no Payara, um fork do GlassFish, e [Apache TomEE](#), baseado no Tomcat.

Decidindo por uma Arquitetura de Microservices no Caelum Eats

Por enquanto, no Caelum Eats, temos uma aplicação monolítica.

Há times separados pelos Bounded Contexts identificados: Pagamentos, Distância, Pedidos, Restaurantes, Administrativo. O código está organizado, alinhados com os Negócios, mas a base de código é uma só. Poderíamos fazer algo mais independente se usássemos uma Arquitetura de Plugins e/ou um Module System diferente, mas os módulos Maven requerem todos os times trabalhando no mesmo projeto.

Há maior controle das dependências entre os módulos do que um projeto não modularizado. Mas ainda é possível, com os módulos Maven, usar classes das dependências transitivas: por exemplo, o módulo de Distância, que depende do módulo de Restaurantes, pode usar classes do módulo Administrativo. A fronteira entre módulos Maven não é forte o bastante.

O cenário de Negócios requer uma evolução rápida de algumas partes da aplicação, como o módulo de Pagamentos, que deseja explorar novos meios de pagamento como criptomoedas e QR Code, além de permitir formas de pagamento mais antigas, como dinheiro.

Já em termos tecnológicos, algumas partes da aplicação da requerem

experimentação, como o módulo de Distância tem a necessidade de explorar novas tecnologias seja de geoprocessamento e até de plataformas de programação diferentes da JVM.

Quanto às operações, há partes da aplicação que apresentam uso intenso de CPU, como o módulo de Distância, levando a necessidade de escalar toda a aplicação em diferentes instâncias para atender à necessidade de processamento de um dos módulos. O mesmo módulo de Distância apresenta uso elevado de memória e eventuais estouros de memória, os famigerados `OutOfMemoryError`, param a instância da aplicação como um todo.

O deploy deve ser feito em conjunto, já que o entregável da aplicação é o fat JAR do Spring Boot gerado pelo módulo `eats-application`. Como o módulo de Pagamentos tem uma taxa de mudança mais frequente que o resto da aplicação, o deploy do módulo de Pagamentos é um deploy de toda a aplicação. O time de pagamentos precisa coordenar a atividade com os outros times antes de lançar uma nova versão.

Nesse cenário, poderíamos aproveitar alguns dos prós de uma Arquitetura de Microservices:

- **Deploy independente:** o time de Pagamentos poderia publicar novas versões com a frequência desejada, minimizando a necessidade de sincronizar os trabalhos com outras equipes, deixando essa coordenação para mudanças nos contratos com outras equipes
- **Escalabilidade independente:** se separarmos o módulo de Distância em um serviço, podemos alocar mais recursos de memória e processamento, sem a necessidade de investir em poder computacional para os outros módulos
- **Maior isolamento de falhas:** um estouro de memória em um serviço de Distância ficaria isolado a instâncias desse serviço, sem derrubar outras funcionalidades
- **Experimentação tecnológica:** o time de Distância poderia explorar o uso de novas tecnologias com maior independência

- **Fronteiras fortes entre componentes:** acabaríamos com o uso indevido de dependências transitivas e as bases de código ficariam completamente isoladas; as dependências entre os serviços seriam por meio de suas APIs

Uma vez que decidimos ir em direção a uma Arquitetura de Microservices, temos que ter consciência das dificuldades que enfrentaremos. Teremos que lidar:

- com as consequências de termos um Sistema Distribuído
- com uma maior complexidade no deploy e monitoramento
- com a possível perda de consistência dos dados
- com a ausência de garantias transacionais

Como falhar numa migração: o Big Bang Rewrite

No artigo [Things You Should Never Do, Part I](#) (SPOLSKY, 2000), Joel Spolsky conta o caso da Netscape, que passou 3 anos sem lançar uma nova versão e, nesse tempo, viu sua fatia de mercado cair drasticamente. Spolsky diz que o motivo para o fracasso é o pior erro estratégico para uma companhia que depende de software: *reescrever código do zero*. Segundo o autor, é comum que programadores querem jogar código antigo fora e escrever tudo do zero e o motivo para isso é que é *mais difícil ler código do que escrever*. Reescrever todo o código, para Joel, é jogar conhecimento fora, dar vantagem competitiva para os concorrentes e gastar dinheiro com código que já existe. Uma refatoração cuidadosa e reescrita pontual de trechos de código seria uma abordagem melhor.

O ocaso da Netscape e o nascimento do Mozilla são assunto do documentário [Code Rush](#) (WINTON, 2000).

Robert "Uncle Bob" Martin chama essa ideia de reescrever todo o projeto de *Grand Redesign in the Sky*, no livro [Clean Code](#) (MARTIN, 2009). Uncle Bob descreve o cenário em que um time dos sonhos é escolhido para reescrever um projeto do zero, enquanto outro time continua a

manter o sistema atual. O novo time passa a ter que fazer tudo o que o software antigo faz, mantendo-se atualizado com as mudanças que são continuamente realizadas.

Esse cenário de manter um sistema antigo enquanto um novo é reescrito do zero lembra um dos paradoxos do filósofo pré-socrático Zenão de Eleia: o paradoxo de Aquiles e a tartaruga. Nesse paradoxo, o herói grego Aquiles e uma tartaruga resolvem apostar uma corrida, com uma vantagem inicial para a tartaruga. Mesmo a velocidade de Aquiles sendo maior que a da tartaruga, quando Aquiles chega à posição inicial A do animal, este move-se até uma posição B. Quando Aquiles chega à posição B, a tartaruga já teria avançado para uma posição C e assim sucessivamente, *ad infinitum*.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a), chama essa ideia de desenvolver uma nova versão do zero de *Big Bang Rewrite* e diz que é algo extremamente arriscado e que provavelmente resultará em fracasso. Duas aplicações teria que ser mantidas, funcionalidades teriam que ser duplicadas e existiria o risco de parte das funcionalidades reescritas não serem necessárias para o Negócio em um futuro próximo. Para Richardson, o sistema legado seria um alvo em constante movimento.

Estrangulando o monólito

Como fazer a migração para uma Arquitetura de Microservices, já que um Big Bang Rewrite não é uma boa ideia?

Uma ideia eficaz é refatorar a aplicação monolítica incrementalmente, removendo funcionalidades e criando novos serviços ao redor do monólito. Com o decorrer do tempo, o Monólito vai encolhendo até, eventualmente, ser reduzido a pó.

Martin Fowler, no artigo [Strangler Fig Application](#) (FOWLER, 2004), faz uma metáfora dessa redução progressiva do Monólito com um tipo de figueira que cresce em volta de uma árvore hospedeira, eventualmente

matando a árvore original e tornando-se uma coluna com o núcleo oco.



Pattern: STRANGLER APPLICATION

Modernize uma aplicação desenvolvendo incrementalmente uma nova aplicação ao redor do legado.

No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson dá algumas dicas de como lidar com uma Strangler Application:

- **Demonstre valor frequentemente e desde cedo:** a ideia é que sejam usados ciclos curtos e frequentes de entrega. Dessa maneira, a Strangler Application reduz o risco e oferece alto retorno sobre o investimento (ROI), ainda que coexistindo com o sistema original. Devem ser priorizadas novas funcionalidades ou migrações de alto impacto e valor de negócio. Assim, os financiadores da migração oferecerão o suporte necessário, justificando o investimento técnico na nova arquitetura.
- **Minimize as mudanças no Monólito:** será necessário alterar o monólito durante a migração para serviços. Mas essas mudanças

tem que ser gerenciadas, evitando que sejam de alto custo, arriscada e consumam muito tempo.

- **Simplifique a infraestrutura:** pode haver o desejo de explorar novas e sofisticadas plataformas de operações como Kubernetes ou AWS Lambda. A única coisa mandatória é um deployment pipeline com testes automatizados. Com um número reduzido de serviços, não há a necessidade de deploy ou monitoramento sofisticados. Adie o investimento até que haja experiência com uma Arquitetura de Microservices.

Richardson ainda cita que a migração para Microservices pode durar alguns anos, como foi o caso da Amazon. E pode ser que a organização dê prioridade a funcionalidades que geram receita, em detrimento daquebra do Monólito, principalmente, se não oferecer obstáculos.

Fowler, em seu artigo [Strangler Fig Application](#) (FOWLER, 2004), argumenta que novas aplicações deveriam ser arquitetadas de maneira a facilitar uma possível estrangulação no futuro. Afinal, o código que escrevemos hoje será o legado de amanhã.

Começar a migração pelo BD ou pela aplicação?

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), recomenda uma progressão que começa pelo BD:

1. Encontrar as linhas de costura (em inglês, *seams*) da aplicação, agrupando o código do Monólito em pacotes.
2. Identificar as costuras no BD, tentando já quebrar dependências.
3. Dividir o schema do BD, mantendo o código no monólito. Nesse momento, joins, foreign keys e integridade transacional já seriam perdidas. Para Newman, seria uma boa maneira de explorar a decomposição e ajustar detalhes.
4. Só então o código seria dividido em serviços, poucos de cada vez, progressivamente

Já no livro [Microservices AntiPatterns and Pitfalls](#) (RICHARDS, 2016),

Mark Richards discorda diametralmente. Richards argumenta que são muitos comuns ajustes na granularidade dos serviços no começo da migração. Podemos ter quebrado demais os serviços, ou de menos. E reagrupar os dados no BD é muito mais difícil, custoso e propenso a erros que reagrupar o código da aplicação. Para o autor, uma migração para Microservices deveria começar com o código. Assim que haja uma garantia que a granularidade do serviço está correta, os dados podem ser migrados. É importante ressaltar que manter o BD monolítico é uma solução paliativa. Richards deixa claro o risco dessa abordagem: acoplamento dos serviços pelo BD. Discutiremos essa ideia mais adiante no curso.

Já em seu novo livro [Monolith to Microservices](#) (NEWMAN, 2019), Sam Newman explora diferentes abordagens para extração de serviços: pelo BD primeiro, pelo código primeiro e BD e código juntos.

Newman diz que começaria pelo BD nos casos em que a performance ou consistência dos dados são preocupações especiais, de maneira a antecipar problemas. Uma desvantagem de começar pelo BD seria o fato de não trazer benefícios claros no curto prazo.

Começar a extração de serviços pelo código, para Newman, traz a vantagem de facilitar o entendimento de qual é o código necessário para o serviço a ser extraído. Além disso, desde cedo há um artefato de código cujo deploy é independente. Uma grande desvantagem é que, na experiência de Newman, é muito comum parar a migração e manter um Shared Database. Outra preocupação é que desafios de performance e consistência são deixados para o futuro, o que pode trazer surpresas nefastas.

Fazer a extração simultânea do BD e do código deve ser evitado, na opinião de Newman. É um passo muito grande, de alto risco e impacto.

No fim das contas, Newman conclui com "Depende". Cada situação é diferente e prós e contras tem que ser discutidos com o contexto específico em mente.

O que extraír do monólito?

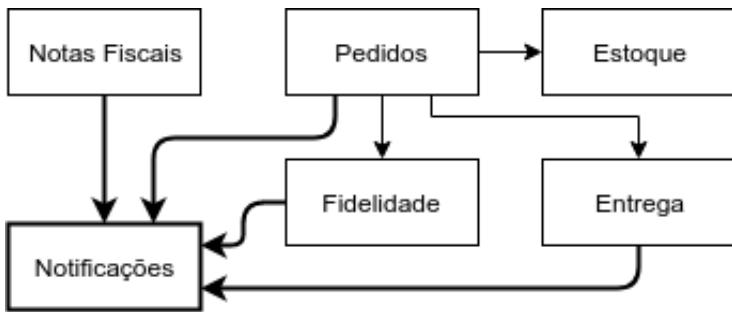
Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a), diz que os módulos devem ser classificados de acordo com critérios que ajudem a visualizar antecipadamente os benefícios de extraí-los do Monólito. Entre os critérios:

- O desenvolvimento será acelerado pela extração: se uma parte específica da aplicação sofrerá uma grande evolução em um futuro próximo, convertê-la para um serviço pode acelerar as entregas.
- Um problema de performance, escalabilidade ou confiabilidade será resolvido: se uma fatia da aplicação apresenta problemas nesse requisitos não-funcionais, afetando o Monólito como um todo, pode ser uma boa ideia extraí-la para um serviço.
- Permitirá a extração de outros serviços: às vezes, as dependências entre os módulos fazem com que fique mais fácil extraír um serviço depois de algum outro ter sido extraído

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), traz reflexões semelhantes. Seriam bons candidatos a serem extraídos, partes do Monólito:

- que mudam com uma frequência muito maior
- que tem times separados, às vezes geograficamente
- que possuem necessidades de segurança e proteção da informação mais restritas
- que teriam vantagens no uso de uma tecnologia diferente

No livro [Monolith to Microservices](#) (NEWMAN, 2019), Sam Newman argumenta que, na priorização de novos serviços a serem extraídos do Monólito, devem ser levadas em conta as dependências entre os grupos de funcionalidade (ou componentes). Partes do código com muitas dependências aferentes (que chegam) dariam muito trabalho para serem extraídas, já que iriam requerer mudanças no código de todas as outras partes que a usam. Já partes do código com poucas, ou nenhuma, dependência aferente seriam bem mais fácil de serem extraídas.

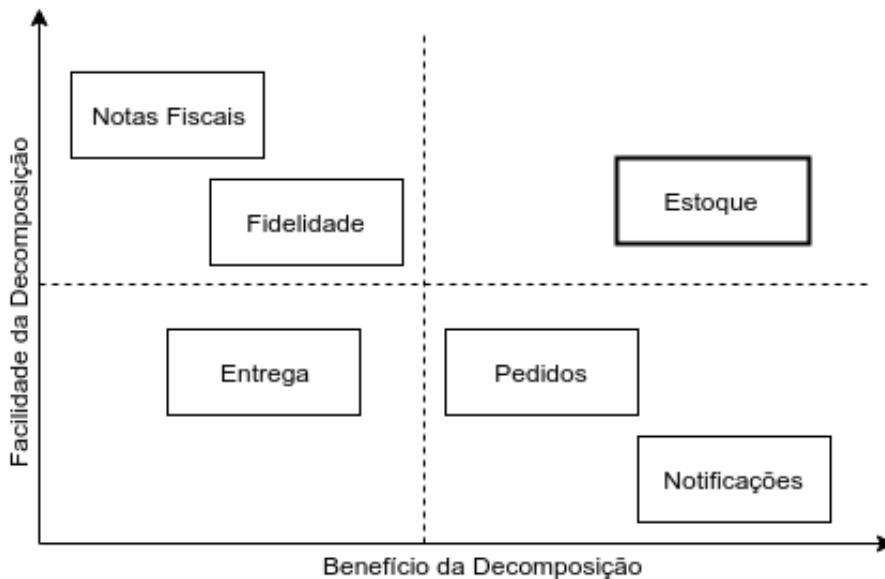


Na imagem anterior, Notificação é um componente difícil de ser extraído, porque tem muitas dependências que chegam. Já um componente como Notas Fiscais seria mais fácil de extrair, porque ninguém depende dele.

Porém, Newman discute que uma visão parecida com a da imagem anterior é uma visão lógica do domínio. Não necessariamente essa visão estará refletida no código. Por exemplo, o BD é um ponto de acoplamento muitas vezes negligenciado.

Ainda no livro [Monolith to Microservices](#) (NEWMAN, 2019), Newman discute que a facilidade de decomposição deve ser ponderada com o benefício trazido para os Negócios. Se o módulo de Notas Fiscais muda muito pouco e não melhora o *time to market*, talvez não seja um bom uso do tempo do time de desenvolvimento e, consequentemente, dos recursos financeiros da organização.

Newman, então, argumenta em favor de uma visão multidimensional, considerando tanto a facilidade como o benefício trazido pelas decomposições. Os componentes podem ser posicionados em um quadrante considerando essas duas dimensões. Ainda que subjetivas, as posições relativas ajudam na priorização. Os componentes que estiverem no canto superior direito do quadrante são bons candidatos à decomposição.



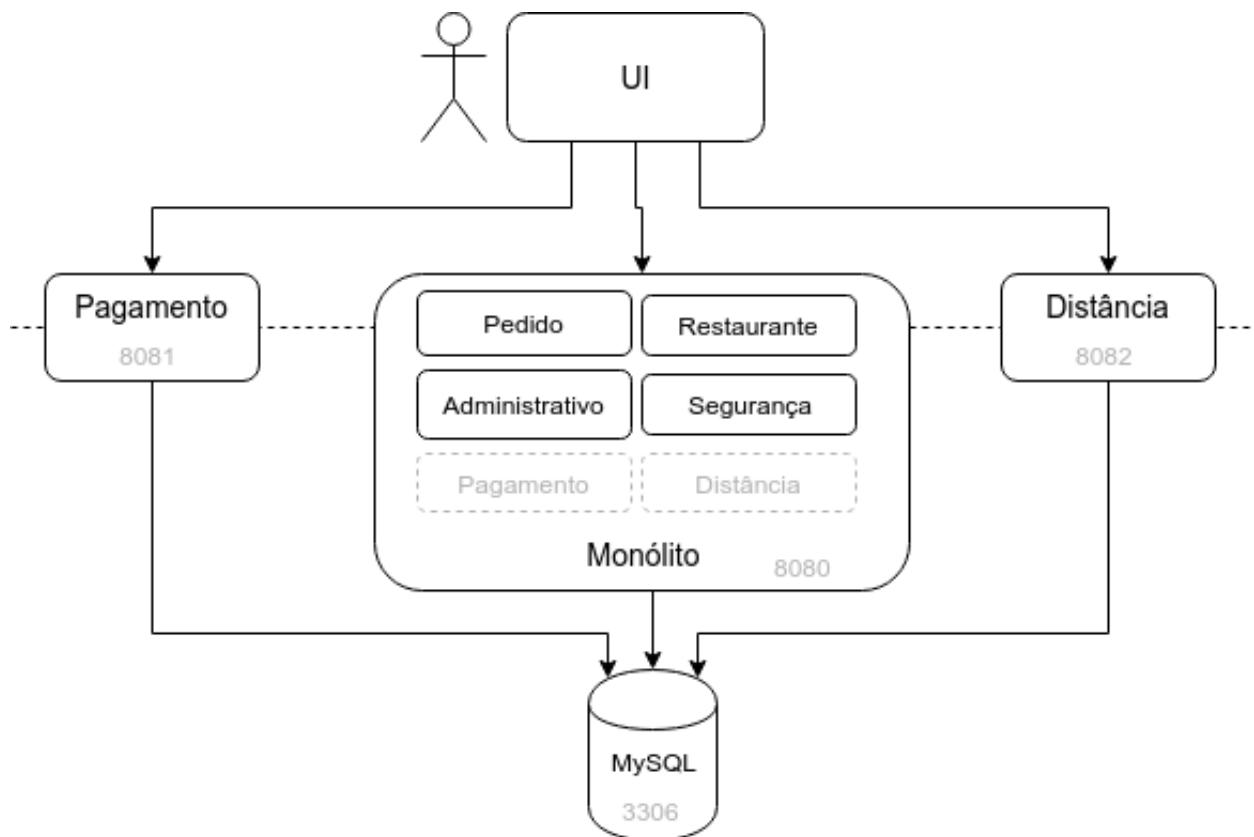
Extraindo serviços do Monólito do Caelum Eats

Como vimos anteriormente, o módulo de Pagamentos do Monólito do Caelum Eats precisa evoluir mais rápido que os demais e, portanto, seria interessante que o deploy fosse independente.

Já para o módulo de Distância, há a necessidade de experimentação tecnológica. Em termos de operações, há maior uso de recursos computacionais e, então, seria interessante escalar esse módulo de maneira independente. Além disso, o módulo de Distância falha mais frequentemente que os outros módulos, tirando toda a aplicação do ar. Seria interessante que as falhas fossem isoladas.

Podemos pensar numa estratégia em que os módulos de Pagamentos e Distância seriam extraídos por seus respectivos times, em paralelo. Esses times trabalhariam em bases de código separadas e, no melhor estilo DevOps, cuidariam de sua infraestrutura.

Ainda há um empecilho: o Banco de Dados. Por enquanto, deixaremos um só BD. Mais adiante, discutiremos se essa será a decisão final.



Quebrando o Domínio

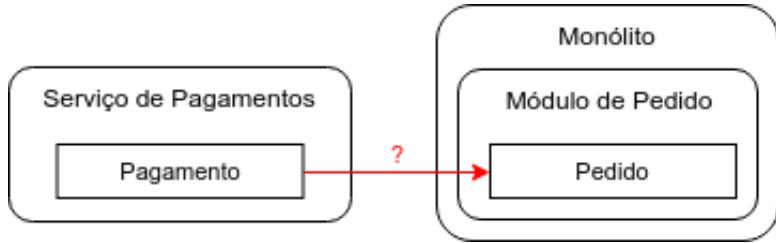
No capítulo que discute sobre como refatorar um monólito em direção a Microservices, do livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson diz que é necessário extrair o Modelo de Domínio específico para um novo serviço do Modelo de Domínio já existente no Monólito. E um dos principais desafios é eliminar referências a objetos de outros domínios, que vão além das fronteiras de um serviço.

No Caelum Eats, por exemplo, o novo serviço de Pagamentos teria um objeto `Pagamento` que está relacionado a um `Pedido`, que continuará no módulo de Pedido do Monólito.

```

class Pagamento {
    // outros atributos...
    @ManyToOne(optional=false)
    private Pedido pedido;
}

```



Observação: além de depender de Pedido, um Pagamento também depende de FormaDePagamento que está presente no módulo Administrativo do Monólito.

Essa dependência a objetos além dos limites do serviço é problemática porque haveria um acoplamento indesejado entre os Modelos de Domínio. Não há a necessidade do serviço de Pagamentos conhecer todos os detalhes de um pedido. Além disso, como tanto Pagamento como Pedido são entidades, haveria uma dependência pelo BD, que queremos evitar, pensando nos próximos passos da extração do serviço de Pagamentos.

Uma ideia seria usar bibliotecas com o Modelo de Domínio de outro serviço, o que é comumente chamado de *Shared Libs*. Por exemplo, no Caelum Eats, poderíamos ter um JAR apenas com a classe Pedido e classes associadas, como ItemDoPedido, Entrega e Avaliacao.

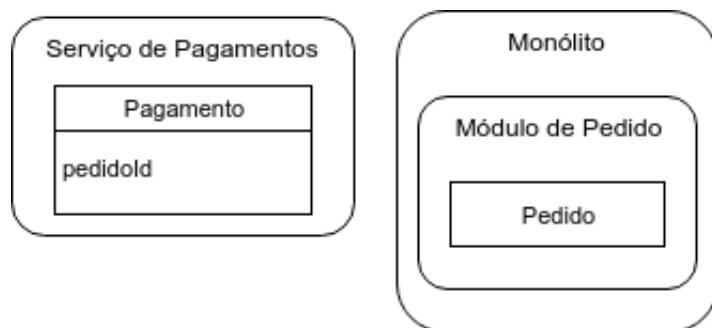
Porém, usar *Shared Libs* para classes de Domínio traz um acoplamento entre os serviços ao redor da API. A cada mudança do Modelo de Domínio de um serviço, todos os seus clientes devem receber uma nova versão do JAR e deve ser feito um novo deploy em cada um deles.

Para Richardson, porém, há lugar para *Shared Libs*: para funcionalidades que são improváveis de serem modificadas, como uma classe Moeda. Bibliotecas técnicas, como frameworks, drivers e ferramentas para tarefas mais específicas, não são um problema.

Richardson argumenta que uma boa solução é pensar em termos de Agregados do DDD, que referenciam outros Agregados por meio da identidade de sua raiz.

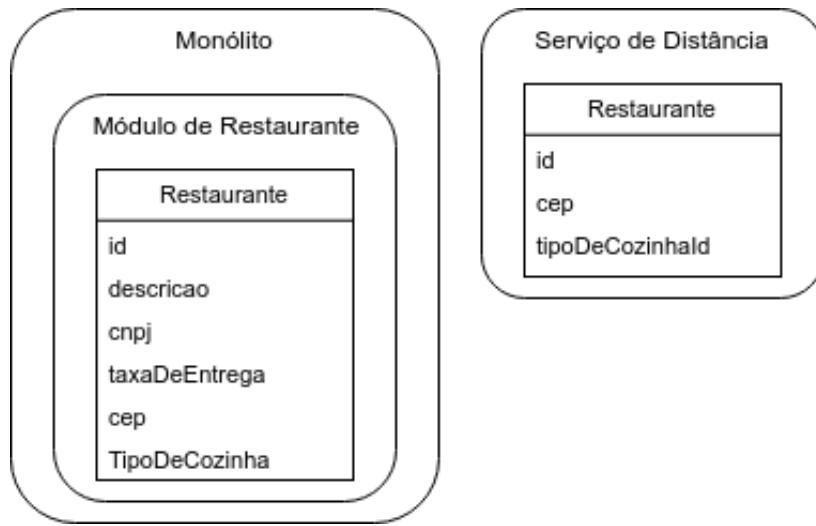
Traduzindo isso para o cenário do Caelum Eats, teríamos uma referência, em `Pagamento`, apenas ao `id` do `Pedido`:

```
class Pagamento {  
    // outros atributos...  
  
    @ManyToOne(optional=false)  
    private Pedido pedido;  
  
    @Column(nullable=false)  
    private Long pedidoId;  
}
```



Richardson discute que uma pequena mudança como a feita anteriormente pode trazer um grande impacto para outras classes, que esperavam uma referência a um objeto. Além disso, há desafios maiores como extrair lógica de negócio de classes que tem mais de uma responsabilidade.

Um outro caso interessante são serviços que tem visões diferentes de uma mesma entidade. Por exemplo, a classe `Restaurante` é utilizada tanto pelo módulo de Restaurante do Monólito como pelo serviço de Distância. Mas, enquanto o módulo Restaurante tem a necessidade de manter o CNPJ, taxa de entrega e outros detalhes, o serviço de Distância só está interessado no CEP e Tipo de Cozinha.



Criando um Microservice de Pagamentos

Vamos iniciar criando um projeto para o serviço de pagamentos.

Pelo navegador, abra <https://start.spring.io/>. Em *Project*, mantenha *Maven Project*. Em *Language*, mantenha Java. Em *Spring Boot*, mantenha a versão padrão. No trecho de *Project Metadata*, defina:

- br.com.caelum em *Group*
- eats-pagamento-service em *Artifact*

Clique em *More options*. Mantenha o valor em *Name*. Apague a *Description*, deixando-a em branco. Em *Package Name*, mude para `br.com.caelum.eats.pagamento`.

Mantenha o *Packaging* como *Jar*. Mantenha a *Java Version* em 8.

Em *Dependencies*, adicione:

- Web
- DevTools
- Lombok
- JPA
- MySQL

Clique em *Generate Project*.

Extraia o eats-pagamento-service.zip.

No arquivo `src/main/resources/application.properties`, modifique a porta para 8081 e, por enquanto, aponte para o mesmo BD do monólito. Defina também algumas outras configurações do JPA e de serialização de JSON.

```
##### fj33-eats-pagamento-
service/src/main/resources/application.properties

server.port = 8081

#DATASOURCE CONFIGS
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true
spring.datasource.username=<SEU USUARIO>
spring.datasource.password=<SU A SENHA>

#JPA CONFIGS
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true

spring.jackson.serialization.fail-on-empty-beans=false
```

Observação: <SEU USUARIO> e <SU A SENHA> devem ser trocados pelos valores do MySQL do monólito.

Extraindo código de pagamentos do monólito

Copie do módulo eats-pagamento do monólito, as seguintes classes, colando-as no pacote `br.com.caelum.eats.pagamento` do eats-pagamento-service:

- Pagamento
- PagamentoController
- PagamentoDto

- PagamentoRepository
- ResourceNotFoundException

Dica: você pode copiar e colar pelo próprio Eclipse.

Há alguns erros de compilação. Os corrigiremos nos próximos passos.

Na classe `Pagamento`, há erros de compilação nas referências às classes `Pedido` e `FormaDePagamento` que são, respectivamente, dos módulos `eats-pedido` e `eats-administrativo` do monólito.

Será que devemos colocar dependências Maven a esses módulos? Não parece uma boa, não é mesmo?

Vamos, então, trocar as referências a essas classes pelos respectivos ids, de maneira a referenciar as raízes dos agregados `Pedido` e `FormaDePagamento`:

```
##### fj33-eats-pagamento-
service/src/main/java/br/com/caelum/eats/pagamento/Pagamento.java
```

```
// anotações ...
class Pagamento {

    // código omitido...

    @ManyToOne(optional=false)
    private Pedido pedido;

    @Column(nullable=false)
    private Long pedidoId;

    @ManyToOne(optional=false)
    private FormaDePagamento formaDePagamento;

    @Column(nullable=false)
    private Long formaDePagamentoId;
```

```
}
```

Ajuste os imports, removendo os desnecessários e adicionando novos:

```
import br.com.caelum.eats.admin.FormaDePagamento;
import br.com.caelum.eats.pedido.Pedido;
import javax.persistence.ManyToOne;

import javax.persistence.Column; // adicionado ...

// outros imports ...
```

A mesma mudança deve ser feita para a classe PagamentoDto, referenciando apenas os ids das classes PedidoDto e FormaDePagamento:

```
##### fj33-eats-pagamento-
service/src/main/java(br/com/caelum/eats/pagamento/PagamentoDto.jav
a
```

```
// anotações ...
class PagamentoDto {

    // outros atributos...

    private FormaDePagamentoDto formaDePagamento;
    private Long formaDePagamentoId;

    private PedidoDto pedido;
    private Long pedidoId;

    public PagamentoDto(Pagamento p) {
        this(p.getId(), p.getValor(), p.getNome(), p.getNumero(),
            new FormaDePagamentoDto(p.getFormaDePagamento()),
            p.getFormaDePagamentoId(),
            new PedidoDto(p.getPedido()), p.getPedidoId());
```

```
}
```

```
}
```

Remova os imports desnecessários:

```
import br.com.caelum.eats.administrative.FormaDeP...
import br.com.caelum.eats.pedido.PedidoDto;
```

Ao confirmar um pagamento, a classe `PagamentoController` atualiza o status do pedido.

Por enquanto, vamos simplificar a confirmação de pagamento, que ficará semelhante a criação e cancelamento: apenas o status do pagamento será atualizado.

Depois voltaremos com a atualização do pedido.

```
##### fj33-eats-pagamento-
service/src/main/java(br/com/caelum/eats/pagamento/PagamentoController.java)
```

```
// anotações ...
class PagamentoController {

    private PagamentoRepository pagamentoRepo;
    private PedidoService pedidos;

    // demais métodos...

    @PutMapping("/{id}")
    public PagamentoDto confirma(@PathVariable Long id) {
        Pagamento pagamento = pagamentoRepo.findById(id).orElseTh
        pagamento.setStatus(Pagamento.Status.CONFIRMADO);
        pagamentoRepo.save(pagamento);
        Long pedidoId = pagamento.getPedido().getId();
```

```
-P-edido-pedido==pedidos.-perI-dC-omI-tens+pedidoI-d  
-pedido.-setS-tatus+P-edido.-S-tatus.-P-A-G-0-> ;  
-pedidos.-atualizaS-tatus+P-edido.-S-tatus.-P-A-G-0,-  
    return new PagamentoDto(pagamento);  
}  
}
```

Ah! Limpe os imports:

```
import br.com.caetum.eats.pedido.P-edido;  
import br.com.caetum.eats.pedido.P-edidos-service;
```

Fazendo a UI chamar novo serviço de pagamentos

Adicione uma propriedade `pagamentoUrl`, que aponta para o endereço do novo serviço de pagamentos, no arquivo `environment.ts`:

```
##### fj33-eats-ui/src/environments/environment.ts
```

```
export const environment = {  
  production: false,  
  baseUrl: '//localhost:8080'  
 , pagamentoUrl: '//localhost:8081' //adicionado  
};
```

Use a nova propriedade `pagamentoUrl` na classe `PagamentoService`:

```
##### fj33-eats-ui/src/app/services/pagamento.service.ts
```

```
export class PagamentoService {  
  
  private API == environment.baseUrl + '/pagamentos'  
  private API = environment.pagamentoUrl + '/pagamentos';
```

```
// restante do código ...  
}
```

No eats-pagamento-service, trocamos referências às entidades Pedido e FormaDePagamento pelos respectivos ids. Essa mudança afeta o código do front-end. Faça o ajuste dos ids na classe PagamentoService:

```
##### fj33-eats-ui/src/app/services/pagamento.service.ts
```

```
export class PagamentoService {  
  
    // código omitido ...  
  
    cria(pagamento): Observable<any> {  
        this.ajustaIds(pagamento); // adicionado  
        return this.http.post(`.${this.API}`, pagamento);  
    }  
  
    confirma(pagamento): Observable<any> {  
        this.ajustaIds(pagamento); // adicionado  
        return this.http.put(`.${this.API}/${pagamento.id}`, null);  
    }  
  
    cancela(pagamento): Observable<any> {  
        this.ajustaIds(pagamento); // adicionado  
        return this.http.delete(`.${this.API}/${pagamento.id}`);  
    }  
  
    // adicionado  
    private ajustaIds(pagamento) {  
        pagamento.formaDePagamentoId = pagamento.formaDePagamentoId || pagamento.formaDePagamentoId;  
        pagamento.pedidoId = pagamento.pedidoId || pagamento.pedidoId;  
    }  
}
```

O código do método privado `ajustaIds` define as propriedades

`formaDePagamentoId` e `pedidoId`, caso ainda não estejam presentes.

No componente `PagamentoPedidoComponent`, precisamos fazer ajustes para usar o atributo `pedidoId` do pagamento:

```
##### fj33-eats-ui/src/app/pedido/pagamento/pagamento-
pedido.component.ts
```

```
export class PagamentoPedidoComponent implements OnInit {

    // código omitido ...

    confirmaPagamento() {
        this.pagamentoService.confirma(this.pagamento)
            .subscribe(pagamento => this.router.navigate([
                'pagamento',
                pagamento.id
            ]))
            .subscribe(pagamento => this.router.navigateByUrl(`pedidos/${pagamento.id}`))
    }

    // restante do código ...
}
```

Com o monólito e o serviço de pagamentos sendo executados, podemos testar o pagamento de um novo pedido.

Deve ocorrer um *Erro no Servidor*. O Console do navegador, acessível com F12, deve ter um erro parecido com:

```
Access to XMLHttpRequest at 'http://localhost:8081/pagamentos' from
origin 'http://localhost:4200' has been blocked by CORS policy:
Response to preflight request doesn't pass access control check: No
'Access-Control-Allow-Origin' header is present on the requested
resource.
```

Isso acontece porque precisamos habilitar o CORS no serviço de pagamentos, que está sendo invocado diretamente pelo navegador.

Habilitando CORS no serviço de pagamentos

Para habilitar o Cross-Origin Resource Sharing (CORS) no serviço de pagamento, é necessário definir uma classe `CorsConfig` no pacote `br.com.caelum.eats.pagamento`, semelhante à do módulo `eats-application` do monólito:

```
@Configuration
class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**").allowedMethods("*").allowCred
    }

}
```

Faça um novo pedido, crie e confirme um pagamento. Deve funcionar!

Note apenas um detalhe: o status do pedido, exibido na tela após a confirmação do pagamento, **está REALIZADO e não PAGO**. Isso ocorre porque removemos a chamada à classe `PedidoService`, que ainda está no módulo `eats-pedido` do monólito. Corrigiremos esse detalhe mais adiante no curso.

Apagando código de pagamentos do monólito

Remova a dependência a `eats-pagamento` do `pom.xml` do módulo `eats-application` do monólito:

```
#####
fj33-eats-monolito-modular/eats/eats-application/pom.xml

<dependency>
    <groupId>br.com/caelum</groupId>
    <artifactId>eats-pagamento</artifactId>
```

```
—<version>${project.version}</version>
-</dependency>
```

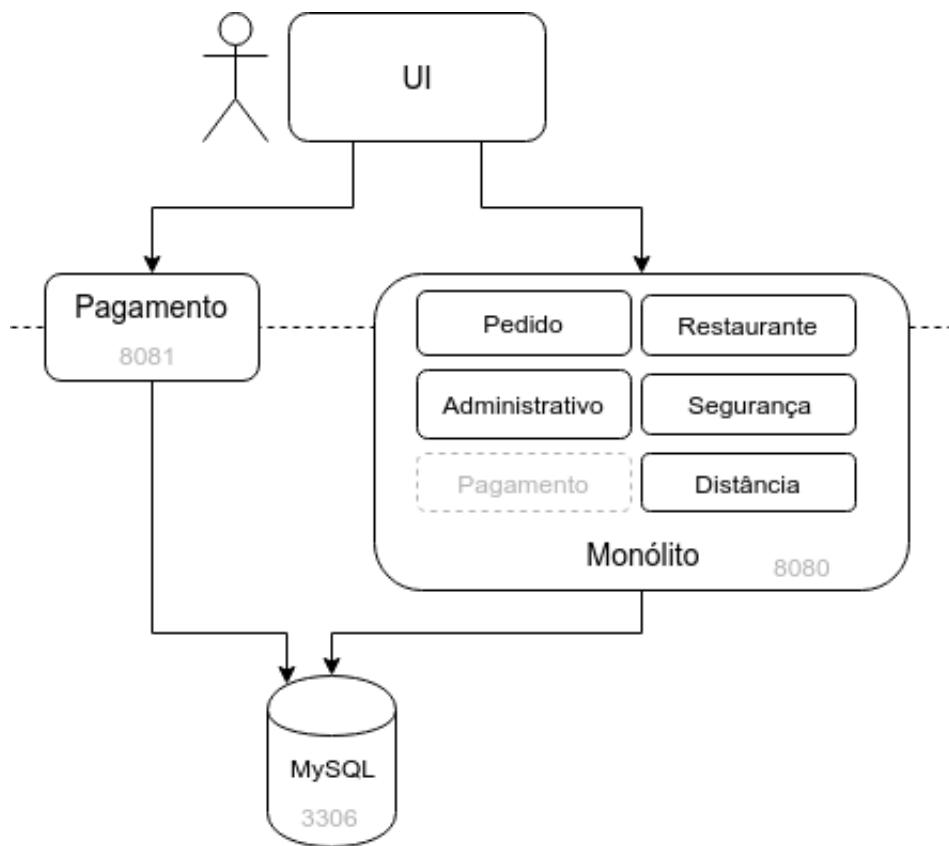
No projeto pai dos módulos, o projeto eats, remova o módulo eats-pagamento do pom.xml:

```
##### fj33-eats-monolito-modular/eats/pom.xml
```

```
<modules>
  <module>eats-administrativo</module>
  <module>eats-pagamento</module>
  <module>eats-restaurante</module>
  <module>eats-pedido</module>
  <module>eats-distancia</module>
  <module>eats-seguranca</module>
  <module>eats-application</module>
</modules>
```

Apague o módulo eats-pagamento do monólito. Pelo Eclipse, tecle *Delete* em cima do módulo, selecione a opção *Delete project contents on disk (cannot be undone)* e clique em *OK*. O diretório com o código do módulo eats-pagamento será removido do disco.

Extraímos nosso primeiro serviço do monólito. A evolução do código de pagamento, incluindo a exploração de novos meios de pagamento, pode ser feita em uma base de código separada do monólito. Porém, ainda mantivemos o mesmo BD, que será migrado em capítulos posteriores.



Exercício: Executando o novo serviço de pagamentos

1. Abra um Terminal e, no Desktop, clone o projeto com o código do serviço de pagamentos:

```
cd ~/Desktop
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-pa
```

Vamos criar um workspace do Eclipse separado para os Microservices, mantendo aberto o workspace com o monólito. Para isso, clique no ícone do Eclipse da área de trabalho. Em *Workspace*, defina `/home/<usuario-do-curso>/workspace-microservices`, onde `<usuario-do-curso>` é o login do curso.

No Eclipse, importe o projeto `fj33-eats-pagamento-service`, usando o menu *File > Import > Existing Maven Projects*.

Então, execute a classe `EatsPagamentoServiceApplication`.

Teste a criação de um pagamento com o cURL:

```
curl -X POST  
-i  
-H 'Content-Type: application/json'  
-d '{ "valor": 51.8, "nome": "JOÃO DA SILVA", "numero": "1111  
http://localhost:8081/pagamentos
```

Para que você não precise digitar muito, o comando acima está disponível em: <https://gitlab.com/snippets/1859389>

No comando acima, usamos as seguintes opções do cURL:

- `-x` define o método HTTP a ser utilizado
- `-i` inclui informações detalhadas da resposta
- `-H` define um cabeçalho HTTP
- `-d` define uma representação do recurso a ser enviado ao serviço

A resposta deve ser algo parecido com:

```
HTTP/1.1 200  
Content-Type: application/json; charset=UTF-8  
Transfer-Encoding: chunked  
Date: Tue, 21 May 2019 20:27:10 GMT
```

```
{ "id":7, "valor":51.8, "nome":"JOÃO DA SILVA",  
"numero":"1111 2222 3333 4444", "expiracao":"2022-07", "codi  
"status":"CRIADO", "formaDePagamentoId":2, "pedidoId":1}
```

Observação: há outros clientes para testar APIs RESTful, como o Postman. Fique à vontade para usá-los. Peça ajuda ao instrutor para instalá-los.

Usando o id retornado no passo anterior, teste a confirmação do

pagamento pelo cURL, com o seguinte comando:

```
curl -X PUT -i http://localhost:8081/pagamentos/7
```

Você deve obter uma resposta semelhante a:

```
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Tue, 21 May 2019 20:31:08 GMT
```

```
{ "id":7, "valor":51.8, "nome":"JOÃO DA SILVA",
  "numero":"1111 2222 3333 4444", "expiracao":"2022-07", "codi
  "status":"CONFIRMADO","formaDePagamentoId":2,"pedidoId":1}
```

Observe que o status foi modificado para **CONFIRMADO**.

2. Pare a execução do monólito, caso esteja no ar.

Vá até o diretório do monólito. Obtenha o código da branch `cap3-extrai-pagamento-service`, que já tem o serviço de pagamentos extraído.

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap3-extrai-pagamento-service
```

Execute novamente a classe `EatsApplication` do módulo `eats-application` do monólito.

3. Pare a execução da UI.

No diretório da UI, mude a branch para `cap3-extrai-pagamento-service`, que contém as alterações necessárias para invocar o novo serviço de pagamentos.

```
cd ~/Desktop/fj33-eats-ui  
git checkout -f cap3-extrai-pagamento-service
```

Execute novamente a UI com o comando `ng serve`.

Acesse `http://localhost:4200` e realize um pedido. Tente criar um pagamento.

Observe que, após a confirmação do pagamento, o status do pedido **está REALIZADO e não PAGO**. Isso ocorre porque removemos a chamada à classe `PedidoService`, cujo código ainda está no monólito. Corrigiremos esse detalhe mais adiante no curso.

Criando um Microservice de distância

Abra `https://start.spring.io/` no navegador. Em *Project*, mantenha *Maven Project*. Em *Language*, mantenha Java. Em *Spring Boot*, mantenha a versão padrão. No trecho de *Project Metadata*, defina:

- `br.com.caelum` em *Group*
- `eats-distancia-service` em *Artifact*

Clique em *More options*. Mantenha o valor em *Name*. Apague a *Description*, deixando-a em branco. Em *Package Name*, mude para `br.com.caelum.eats.distancia`.

Mantenha o *Packaging* como `Jar`. Mantenha a *Java Version* em 8.

Em *Dependencies*, adicione:

- Web
- DevTools
- Lombok
- JPA
- MySQL

Clique em *Generate Project*.

Descompacte o `eats-distancia-service.zip` para seu Desktop.

Edite o arquivo `src/main/resources/application.properties`, modificando a porta para 8082, apontando para o BD do monólito, além de definir configurações do JPA e de serialização de JSON:

```
##### fj33-eats-distancia-
service/src/main/resources/application.properties

server.port = 8082

#DATASOURCE CONFIGS
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true
spring.datasource.username=<SEU USUARIO>
spring.datasource.password=<SUAS SENHAS>

#JPA CONFIGS
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true

spring.jackson.serialization.fail-on-empty-beans=false
```

Troque `<SEU USUARIO>` e `<SUAS SENHAS>` pelos valores do BD.

Extraindo código de distância do monólito

Copie para o pacote `br.com.caelum.eats.distancia` do serviço `eats-distancia-service`, as seguintes classes do módulo `eats-distancia` do monólito:

- `DistanciaService`
- `RestauranteComDistanciaDto`
- `RestaurantesMaisProximosController`
- `ResourceNotFoundException`

Além disso, já antecipando problemas com CORS no front-end, copie do módulo eats-application do monólito, para o pacote `br.com.caelum.eats.distancia` do serviço de distância, a classe:

- `CorsConfig`

Há alguns erros de compilação na classe `DistanciaService`, que corrigiremos nos passos seguintes.

O motivo de um dos erros de compilação é uma referência à classe `Restaurante` do módulo eats-restaurante do monólito.

Copie essa classe para o pacote `br.com.caelum.eats.distancia` do serviço de distância. Ajuste o pacote, caso seja necessário.

Remova, na classe `Restaurante` copiada, a referência à entidade `TipoDeCozinha`, trocando-a pelo id.

Remova por completo a referência à classe `User`.

```
##### fj33-eats-distancia-
service/src/main/java;br/com/caelum/eats/distancia/Restaurante.java
```

```
// anotações
public class Restaurante {

    // código omitido ...

    @ManyToOne(optional=false)
    private TipoDeCozinha tipoDeCozinha;

    @Column(nullable=false)
    private Long tipoDeCozinhaId;

    @OneToOne
    private User user;

}
```

Ajuste os imports:

```
import javax.persistence.ManyToOne;
import javax.persistence.OneToOne;

import br.com.caelum.eats.administrative.TipoDeCozinha;
import br.com.caelum.eats.seguranca.Usuario;

import javax.persistence.Column; // adicionado ...
```

Na classe `DistanciaService` de `eats-distancia-service`, remova os imports que referenciam as classes `Restaurante` e `TipoDeCozinha`:

```
##### fj33-eats-distancia-
service/src/main/java(br/com/caelum/eats/distancia/DistanciaService.java
```

```
import br.com.caelum.eats.administrative.TipoDeCozinha;
import br.com.caelum.eats.restaurante.Restaurante;
```

Como a classe `Restaurante` foi copiada para o mesmo pacote de `DistanciaService`, não há a necessidade de importá-la.

Mas e para `TipoDeCozinha`? Utilizaremos apenas o id. Por isso, modifique o método `restaurantesDoTipoDeCozinhaMaisProximosAoCep` de `DistanciaService`:

```
##### fj33-eats-distancia-
service/src/main/java(br/com/caelum/eats/distancia/DistanciaService.java
```

```
public List<RestauranteComDistanciaDto> restaurantesDoTipoDeCozinha(TipoDeCozinha tipo) {
    tipo.setId(new TipoDeCozinha());
    tipo.setI_d(tipo.getId());
```

```
List<Restaurante> aprovadosDoTipoDeCozinha = restaurantes.findAll();
List<Restaurante> aprovadosDoTipoDeCozinha = restaurantes.findAll();
return calculaDistanciaParaOsRestaurantes(aprovadosDoTipoDeCozinha);
}
```

Ainda resta um erro de compilação na classe `DistanciaService`: o uso da classe `RestauranteService`. Poderíamos fazer uma chamada remota, por meio de um cliente REST, ao monólito para obter os dados necessários. Porém, para esse serviço, acessaremos diretamente o BD.

Por isso, crie uma interface `RestauranteRepository` no pacote `br.com.caelum.eats.distancia` de `eats-distancia-service`, que estende `JpaRepository` do Spring Data Jpa e possui os métodos usados por `DistanciaService`:

```
#####
fj33-eats-distancia-
service/src/main/java;br/com/caelum/eats/distancia/RestauranteReposito-
ry.java
```

```
package br.com.caelum.eats.distancia;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;

interface RestauranteRepository extends JpaRepository<Restaurante, Long> {
    Page<Restaurante> findAllByAprovadoAndTipoDeCozinhaId(boolean aprovado, Long id);
    Page<Restaurante> findAllByAprovado(boolean aprovado, Pageable pageable);
}
```

Em `DistanciaService`, use `RestauranteRepository` ao invés de

RestauranteService:

```
##### fj33-eats-distancia-
service/src/main/java(br/com/caelum/eats/distancia/DistanciaService.java

// anotações ...
class DistanciaService {

    // código omitido ...

    private RestauranteService restaurantes;
    private RestauranteRepository restaurantes;

    // restante do código ...

}
```

Limpe o import:

```
import br.com.caelum.eats.restaurante.Restaurante
```

Simplificando o restaurante do serviço de distância

O eats-distancia-service necessita apenas de um subconjunto das informações do restaurante: o `id`, o `cep`, se o restaurante está aprovado e o `tipoDeCozinhaId`.

Enxugue a classe Restaurante do pacote `br.com.caelum.eats.distancia`, deixando apenas as informações realmente necessárias:

```
##### fj33-eats-distancia-
service/src/main/java(br/com/caelum/eats/distancia/Restaurante.java

// anotações ...
```

```
public class Restaurante {  
  
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private Long id;  
  
    @NotNull @Size(max=18)  
    private String cep;  
  
    @NotNull @Size(max=255)  
    private String nome;  
  
    @Size(max=1000)  
    private String descricao;  
  
    @NotNull @Size(max=9)  
    private String cep;  
  
    @NotNull @Size(max=300)  
    private String endereco;  
  
    @Positive  
    private BigDecimal taxaEntregaMReais;  
  
    @Positive @Min(10) @Max(180)  
    private Integer tempoDeEntregaMinimoEmMinutos;  
  
    @Positive @Min(10) @Max(180)  
    private Integer tempoDeEntregaMaximoEmMinutos;  
  
    private Boolean aprovado;  
  
    @Column(nullable=false)  
    private Long tipoDeCozinhaId;  
}
```

fj33-eats-distancia-
service/src/main/java;br/com/caelum/eats/distancia/Restaurante.java

O conteúdo da classe `Restaurante` do serviço de distância ficará da seguinte maneira:

```
// anotações ...
public class Restaurante {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String cep;

    private Boolean aprovado;

    private Long tipoDeCozinhaId;

}
```

Alguns dos imports podem ser removidos:

```
import java.math.BigDecimal;
import javax.persistence.Table;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Positive;
import javax.validation.constraints.Size;
```

Fazendo a UI chamar serviço de distância

Abra o projeto `fj33-eats-ui` e defina uma nova propriedade `distanciaUrl` no arquivo `environment.ts`:

```
##### fj33-eats-ui/src/environments/environment.ts
```

```
export const environment = {
  production: false,
  baseUrl: '//localhost:8080'
, pagamentoUrl: '//localhost:8081'
, distanciaUrl: '//localhost:8082'
};
```

Modifique a classe `RestauranteService` para que use `distanciaUrl` nos métodos `maisProximosPorCep`, `maisProximosPorCepETipoDeCozinha` e `distanciaPorCepEId`:

```
##### f33-eats-ui/src/app/services/restaurante.service.ts
```

```
export class RestauranteService {

  private API = environment.baseUrl;
  private DISTANCIA_API = environment.distanciaUrl; // adicionei

  // código omitido ...

  maisProximosPorCep(cep: string): Observable<any> {
    return -this.-http.-get(`-${this.API}/restaurantes/
      return this.http.get(` ${this.DISTANCIA_API}/restaurantes/
    }

  maisProximosPorCepETipoDeCozinha(cep: string, tipoDeCozinha]: Observable<any> {
    return -this.-http.-get(`-${this.API}/restaurantes/
      return this.http.get(` ${this.DISTANCIA_API}/restaurantes/
    }

  distanciaPorCepEId(cep: string, restauranteId: string): Observable<any> {
    return -this.-http.-get(`-${this.API}/restaurantes/
      return this.http.get(` ${this.DISTANCIA_API}/restaurantes/
    }

  // restante do código ...
}
```

}

Removendo código de distância do monólito

Remova a dependência a eats-distancia do pom.xml do módulo eats-application:

```
##### fj33-eats-monolito-modular/eats/eats-application/pom.xml
```

```
<dependency>
  <groupId>br.com.caetum</groupId>
  <artifactId>eats-distancia</artifactId>
  <version>${project.version}</version>
</dependency>
```

No pom.xml do projeto eats, o módulo pai, remova a declaração do módulo eats-distancia:

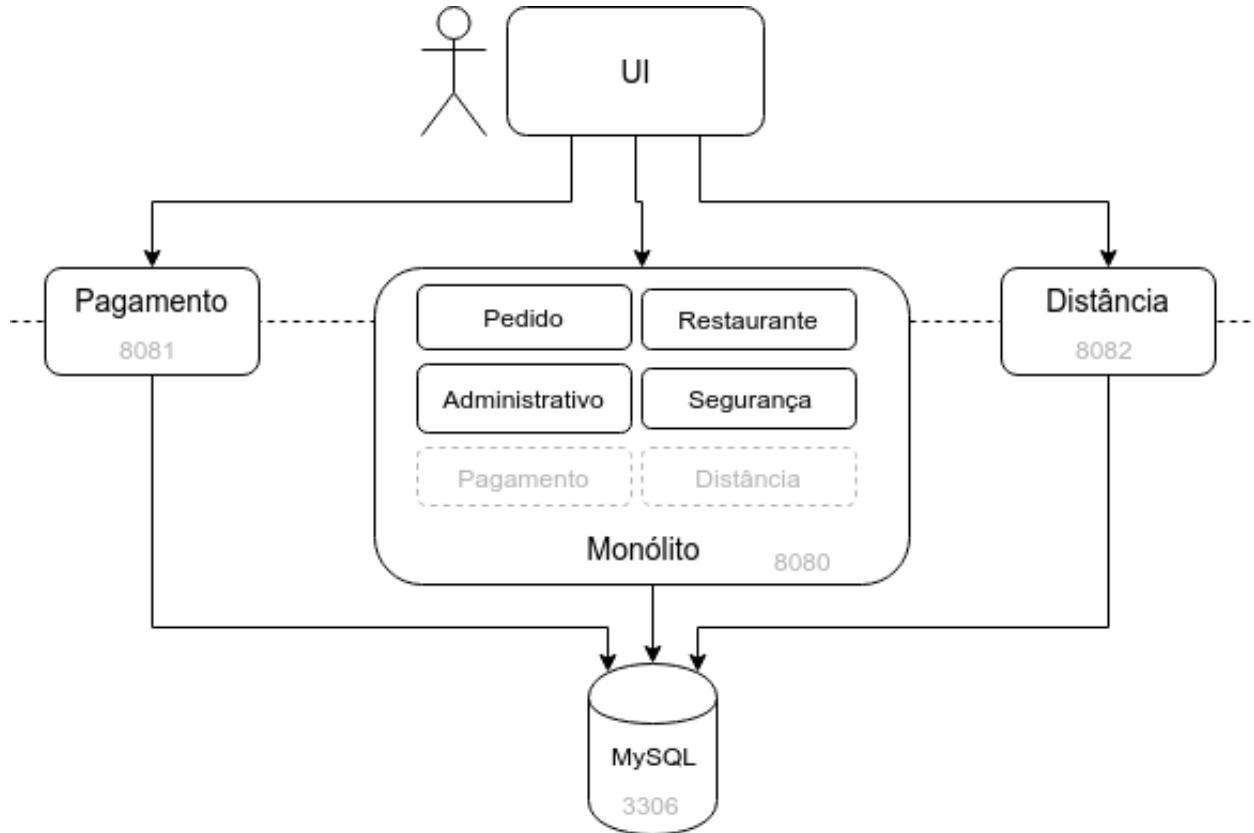
```
##### fj33-eats-monolito-modular/eats/pom.xml
```

```
<modules>
  <module>eats-administrativo</module>
  <module>eats-restaurante</module>
  <module>eats-pedido</module>
  <module>eats-distancia</module>
  <module>eats-seguranca</module>
  <module>eats-application</module>
</modules>
```

Apague o código do módulo eats-distancia do monólito. Pelo Eclipse, tecle *Delete* em cima do módulo, selecione a opção *Delete project contents on disk (cannot be undone)* e clique em *OK*.

Ufa! Mais um serviço extraído do monólito. Em um projeto real, isso seria

feito em paralelo com a extração do serviço de pagamentos, por times independentes. A exploração de novas tecnologias, afim de melhorar o desempenho da busca de restaurantes próximos a um dado CEP, poderia ser feita de maneira separada do monólito. Contudo, o BD continua monolítico.



Exercício: Executando o novo serviço de distância

1. Em um Terminal, clone o projeto do serviço de distância para o seu Desktop:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-di
```

No workspace de Microservices do Eclipse, use o menu *File > Import > Existing Maven Projects* para importar o projeto `fj33-eats-distancia-service`.

Execute a classe `EatsDistanciaServiceApplication`.

Use o cURL para disparar chamadas ao serviço de distância.

Para buscar os restaurantes mais próximos ao CEP 71503-510:

```
curl -i http://localhost:8082/restaurantes/mais-proximos/71503
```

A resposta será algo como:

```
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 22 May 2019 18:44:13 GMT
```

```
[{"restauranteId": 1, "distancia": 8.357388557756333824499961}, {"restauranteId": 2, "distancia": 8.170183211279926638326287}]
[
```

Para buscar os restaurantes mais próximos ao CEP 71503-510 com o tipo de cozinha *Chinesa* (que tem o id 1):

```
curl -i http://localhost:8082/restaurantes/mais-proximos/71503?cozinha=1
```

A resposta será semelhante a:

```
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 22 May 2019 18:44:13 GMT
```

```
[{"restauranteId": 1, "distancia": 18.382449996133800595998764}]
```

Para descobrir a distância de um dado CEP a um restaurante específico:

```
curl -i http://localhost:8082/restaurantes/71503510/restaurant
```

Teremos um resultado parecido com:

```
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 22 May 2019 18:44:13 GMT
```

```
{ "restauranteId": 1, "distancia":13.9599876403835738853824495 }
```

2. Interrompa o monólito, caso esteja sendo executado.

No diretório do monólito, vá até a branch `cap3-extrai-distancia-service`, que tem as alterações no monólito logo após da extração do serviço de distância:

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap3-extrai-distancia-service
```

Execute novamente a classe `EatsApplication` do módulo `eats-application` do monólito.

3. Interrompa a UI, se estiver sendo executada.

No diretório da UI, altere a branch para `cap3-extrai-distancia-service`, que contém as mudanças para chamar o novo serviço de distância:

```
cd ~/Desktop/fj33-eats-ui
```

```
git checkout -f cap3-extrai-distancia-service
```

Com o comando `ng serve`, garanta que o front-end esteja rodando.

Acesse `http://localhost:4200`. Busque os restaurantes de um dado CEP, escolha um dos restaurantes retornados e, na tela de detalhes do restaurante, verifique que a distância aparece logo acima da descrição. Deve funcionar!

Decompondo o monólito

Da bagunça a camadas

Qual é a emoção que a palavra **monólito** traz pra você?

Muitos desenvolvedores associam a palavra monólito a código mal feito, sem estrutura aparente, com muito código repetido e com dados compartilhados entre partes pouco relacionadas. É o que comumente é chamado de código **Spaghetti** ou, [Big Ball of Mud](#) (FOOTE; YODER, 1999).

Porém, é possível criar monólitos bem estruturados. Uma maneira comum de organizar um monólito é usar camadas: cada camada provê um serviço para a camada de cima e é um cliente das camadas de baixo.

Usualmente, o código de uma aplicação é estruturado em 3 camadas:

- *Apresentação*: responsável por prover serviços ao front-end. Em alguns casos, é feita a renderização das telas da UI, como ao utilizar JSPs.
- *Negócio*: responsável pelos cálculos, fluxos e regras de negócio.
- *Persistência*: responsável pelo acesso aos dados armazenados, geralmente, em um Banco de Dados.

Uma maneira de representar essas camadas em um código Java é utilizar pacotes, o que é conhecido como *Package by Layer*.

Na verdade, é preciso distinguir dois tipos de camadas que influenciam a arquitetura do software: as camadas físicas e as camadas lógicas.

Uma camada física, ou *tier* em inglês, descreve a estrutura de implantação do software, ou seja, as máquinas utilizadas.

O que descrevemos no texto anterior é o conceito de camada lógica. Em inglês, a camada lógica é chamada de *layer*. Trata de agrupar o

código que corresponde às camadas descritas anteriormente.

Camadas no Caelum Eats

O código de back-end do Caelum Eats está organizado em camadas (layers). Podemos observar isso estudando a estrutura de pacotes do código Java:

```
eats
└── controller
└── dto
└── exception
└── model
└── repository
└── service
```

Bem organizado, não é mesmo?

Porém, quando a aplicação começa a crescer, o código passa a ficar difícil de entender. Centenas de Controllers juntos, centenas de Repositories misturados. Passa a ser difícil encontrar o código que precisa ser alterado.

A Arquitetura que Grita

Reflita um pouco: dos projetos implementados com a plataforma Java em que você trabalhou, quantos seguiam uma variação da estrutura Package by Layer que estudamos anteriormente? Provavelmente, a maioria!

Veja a estrutura de diretórios abaixo:

```
.
└── assets
└── controllers
└── helpers
└── mailers
```

```
└── models  
└── views
```

Os diretórios anteriores são a estrutura padrão de um framework de aplicações Web muito influente: o Ruby On Rails.

Perceba que interessante: a estrutura básica de diretórios é familiar; em alguns casos, até temos um palpite sobre qual o framework utilizado; mas não temos ideia do **domínio** da aplicação. Qual é o problema que está sendo resolvido?

No post [Screaming Architecture](#) (MARTIN, 2011), Robert "Uncle Bob" Martin diz que a partir das plantas de edifícios, conseguimos saber se trata-se de uma casa ou uma biblioteca: a arquitetura "grita" a finalidade da construção.

Na realidade, a construção civil tem diferentes plantas: a elétrica, a hidráulica, a estrutural, etc. Uncle Bob parece referir-se a um tipo específico de planta: a **planta baixa**. É feita por um arquiteto e é mais próxima do cliente. A partir dela são projetadas outras plantas mais técnicas e específicas. É o tipo de planta que encontramos em lançamentos de imóveis.

Para projetos de software, entretanto, é usual que a estrutura básica de diretórios indique qual o framework ou qual a ferramenta de build utilizados. Porém, para Uncle Bob, o framework é um detalhe; o Banco de Dados é um detalhe; a Web é um mecanismo de entrega da UI, um detalhe.

Uncle Bob cita a ideia de Ivar Jacobson, um dos criadores do UML, descrita no livro [Object Oriented Software Engineering: A Use-Case Driven Approach](#) (JACOBSON, 1992), de que a arquitetura de um software deveria ser *centrada nos casos de uso* e não em detalhes técnicos.

Por arquiteturas centradas no domínio

No livro [Clean Architecture](#) (MARTIN, 2017), Uncle Bob define uma abordagem arquitetural que torna a aplicação:

- **independente de frameworks:** um framework não é sua aplicação. A estrutura de diretórios e as restrições do design do nosso código não deveriam ser determinadas por um framework. Frameworks deveriam ser usados apenas como ferramentas para que a aplicação cumpra suas necessidades.
- **independente da UI:** a UI muda mais frequentemente que o resto do sistema. Além disso, é possível termos diferentes UIs para as mesmas regras de negócio.
- **independente de BD:** as regras de negócio não devem depender de um Banco de Dados específico. Devemos possibilitar a troca, de maneira fácil, de Oracle ou SQL Server para MongoDB, CouchDB, Neo4J ou qualquer outro BD.
- **testável:** deve ser possível testar as regras de negócio diretamente, sem a necessidade de usar uma UI, BD ou servidor Web.

Uncle Bob ainda cita, no mesmo livro, outras arquiteturas semelhantes:

- Hexagonal Architecture, ou Ports & Adapters, descrita por Alistair Cockburn.
- DCI (Data, Context and Interaction), descrita por James Coplien, pioneiro dos Design Patterns, e Trygve Reenskaug, criador do MVC.
- BCE (Boundary-Control-Entity), introduzida por Ivar Jacobson no livro mencionado anteriormente.

O curso [Práticas de Design e Arquitetura de código para aplicações Java](#) (FJ-38) explora, a partir de um gerador de ebooks, tópicos como os princípios SOLID de Orientação a Objetos e alguns Design Patterns, até chegar progressivamente a uma Arquitetura Hexagonal.

Software é massa!

Raymond J. Rubey cita uma carta ao editor da revista CROSSTALK (RUBEY, 1992), em que é feita uma brincadeira que classifica qualidade

de código como se fossem massas:

- **Spaghetti**: complicado, difícil de entender e impossível de manter
- **Lasagna**: simples, fácil de entender, estruturado em camadas, mas monolítico; na teoria, é fácil de mudar uma camada, mas não na prática
- **Ravioli**: componentes pequenos e soltos, que contém "nutrientes" para o sistema; qualquer componente pode ser modificado ou trocado sem afetar significativamente os outros componentes

Explorando o domínio

Até um certo tamanho, uma aplicação estruturada em camadas no estilo Package by Layer, é fácil de entender. Novos desenvolvedores entram no projeto sem muitas dificuldades para entender a organização do código, já que há uma certa familiaridade.

Mas há um momento em que é interessante reorganizar os pacotes ao redor do domínio, o que alguns chamam de *Package by Feature*. O intuito é que fique bem clara qual é a área de negócio de cada parte do código.

Mas qual critério usar para decompor o monólito? Uma funcionalidade é um pacote, como sugere o nome *Package by Feature*? Talvez isso seja muito granular.

Uma fonte de *insights* interessantes em como explorar o domínio de uma aplicação é o livro [Domain Driven Design](#), de Eric Evans (EVANS, 2003).

Caçando entidades importantes

Todo domínio tem entidades importantes, que são um ponto de entrada para o negócio e tem várias outras entidades ligadas. Um objeto que representa um pedido, por exemplo, terá informações dos itens do pedido, do cliente, da entrega e do pagamento. É o que é chamado, nos termos do DDD, de *Agregado*.

| AGREGADO

Agrupamento de objetos associados que são tratados como uma unidade para fins de alterações nos dados. Referências externas são restritas a um único membro do AGREGADO, designado como *raiz*. Um conjunto de regras de consistência se aplicada dentro dos limites do AGREGADO.

[Domain Driven Design](#) (EVANS, 2003)

Referências a objetos de fora desse Agregado devem ser feitas por meio do objeto principal, a raiz do Agregado. Uma relação entre um pedido e um restaurante deve ser feita pelo pedido, e não pela entrega ou pelo pagamento do pedido.

Os dados de um Agregado são alterados em conjunto. Por isso, Banco de Dados Orientados a Documentos, como o MongoDB, em que um documento pode ter um grafo de outros objetos conectados, são interessantes para persistir Agregados.

E no Caelum Eats?

Um objeto muito importante é o `Pedido`. Associado a um `Pedido` temos uma lista de `ItemDoPedido` e uma `Entrega`. Uma `Entrega` está associada a um `Cliente`. Cada `Pedido` também pode ter uma `Avaliacao`. Há, possivelmente, um `Pagamento` para cada `Pedido`. E um `Pagamento` tem uma `FormaDePagamento`. Um `Pedido` também está associado a um `Restaurante`.

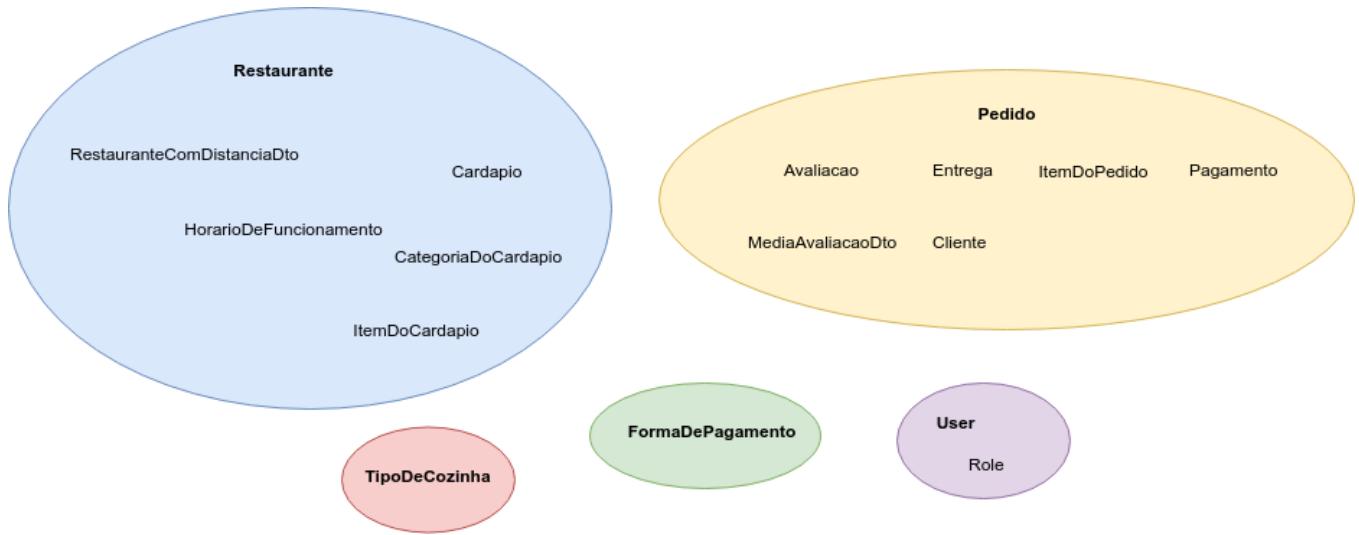
Um `Restaurante` tem seu cadastro mantido por seu dono e é aprovado pelo administrador do Caelum Eats. Um `Restaurante` pode existir sem nenhum pedido, o que acontece logo que foi cadastrado. Um `Restaurante` está relacionado a um `cardapio` que contém uma lista de `CategoriaDoCardapio` que, por sua vez, contém uma lista de `ItemDoCardapio`. Um `Restaurante` possui também uma lista de `HorarioDeFuncionamento` e das `FormaDePagamento` aceitas, assim como um `TipoDeCozinha`.

Um `Restaurante` também possui um `User`, que representa seu dono. O

User também é utilizado pelo administrador do Caelum Eats. Um User tem um ou mais Role.

Uma FormaDePagamento existe independentemente de um Restaurante e os valores possíveis são mantidos pelo administrador. O mesmo ocorre para TipoDeCozinha.

Há algumas classes interessantes como MediaAvaliacoesDto e RestauranteComDistanciaDto, que associam um restaurante à média das notas das avaliações e uma distância a um dado CEP, respectivamente.



Poderíamos alinhar o código do Caelum Eats com o domínio, utilizando os agregados identificados anteriormente.

Das estruturas de comunicação para o código

No artigo [How Do Committees Invent?](#) (CONWAY, 1968), Melvin Conway descreve como sistemas tendem a reproduzir as estruturas de comunicação das empresas/orgãos/corporações que os produziram:

Lei de Conway

Qualquer organização que faz design de sistemas vai inevitavelmente produzir um design cuja estrutura é uma cópia das estruturas de comunicação dessa organização.

No estudo preliminar *Exploring the Duality between Product and*

Organizational Architectures (2008, MACCORMACK et al.) da Harvard Business School, os autores testaram o que chamam de Hipótese do Espelho (em inglês, *Mirroring Hypothesis*): a estrutura dos times influencia na modularidade dos softwares produzidos. Os autores dividem as organizações entre as *altamente acopladas*, em que as visões e os objetivos estão altamente alinhados, e as *baixamente acopladas*, organizadas como comunidades open-source distribuídas geograficamente. Para produtos similares, organizações altamente acopladas tendem a produzir softwares menos modulares.

Um exemplo da Lei de Conway aplicada pode ser encontrada no caso da Amazon, que tem uma regra conhecida como *two pizza team*: um time tem que poder ser alimentado por duas pizzas. Um tanto subjetivo, mas traz a ideia de que os times na Amazon são pequenos, independentes e autônomos, cuidando de todo o ciclo de vida do software, da concepção à operação. E isso influencia na arquitetura do software.

No artigo [Contending with Creaky Platforms CIO](#) (SIMONS; LEROY, 2010), Matthew Simons e Jonny Leroy, argumentam que a Lei de Conway pode ser descrita como: organizações disfuncionais criam aplicações disfuncionais. Por isso, refazer uma aplicação mantendo a mesma estrutura organizacional levaria às mesmas disfunções do software original. Para obter um software mais modular e organizado, poderíamos começar quebrando silos que restringem a habilidade dos times colaborarem de maneira efetiva. Os autores chamam essa ideia de **Manobra Inversa de Conway** (em inglês, *Inverse Conway's Maneuver*).

Pesquisadores da UFMG analisaram se a Lei de Conway se aplica ao kernel do Linux. Para isso, criaram uma métrica que chamaram de DOA (Degree of Authorship), que indica o quanto um determinado desenvolvedor é "autor" de um arquivo do código fonte. O DOA foi estimado por meio da verificação do histórico de uma década dos arquivos no controle de versões. A métrica relaciona um autor a um arquivo e é proporcional a quem criou o arquivo, ao número de mudanças feitas por um determinado autor e é inversamente proporcional ao

número de mudanças feitas por outros desenvolvedores. No artigo de divulgação [Does Conway's Law apply to Linux?](#) (ASERG-UFMG, 2017), é descrita a conclusão de que o kernel do Linux segue uma forma inversa da Lei de Conway, já que a arquitetura definida ao longo dos anos é que influenciou na organização e nas especializações do time de desenvolvimento.

A linguagem do negócio depende do Contexto

Um foco importante do DDD é na linguagem. Os **especialistas de domínio** usam certos termos que devem ser representados nos requisitos, nos testes e, claro, no código de produção. A linguagem do negócio deve estar representada em código, no que chamamos de **Modelo de Domínio** (em inglês, *Domain Model*). Essa linguagem estruturada em torno do domínio e usada por todos os envolvidos no desenvolvimento do software é chamada pelo DDD de **Linguagem Onipresente** (em inglês, *Ubiquitous Language*).

Porém, para uma aplicação de grande porte as coisas não são tão simples. Por exemplo, em uma aplicação de e-commerce, o que é um Produto? Para diferentes especialistas de domínio, um Produto tem diferentes significados:

- para os da Loja Online, um Produto é algo que tem preço, altura, largura e peso.
- para os do Estoque, um Produto é algo que tem uma quantidade em um inventário.
- para os do Financeiro, um Produto é algo que tem um preço e descontos.
- para os de Entrega, um Produto é algo que tem uma altura, largura e peso.

Até atributos de um Produto tem diferentes significados, dependendo do contexto. Para a Loja Online, o que interessa é a altura, largura e peso de um produto fora da caixa, que servem para um cliente saber se o item caberá na pia da sua cozinha ou em seu armário. Já para a Entrega,

esses atributos devem incluir a caixa e não influenciar nos custos de transporte.

Para aplicações maiores, manter apenas um Modelo de Domínio é inviável. A origem do problema está na linguagem utilizada pelos especialistas de domínio: não há só uma Linguagem Onipresente. Nessa situação, tentar unificar o Modelo de Domínio o tornará inconsistente.

A linguagem utilizada pelos especialistas de domínio está atrelada a uma área de negócio. Há um contexto em que um Modelo de Domínio é consistente, porque representa a linguagem de uma área de negócio. No DDD, é importante identificarmos esses **Contextos Delimitados** (em inglês, *Bounded Contexts*), para que não haja degradação dos vários Modelos de Domínio.

CONTEXTO DELIMITADO (BOUNDED CONTEXT)

Aplicabilidade delimitada de um determinado modelo. CONTEXTOS DELIMITADOS dão aos membros de uma equipe um entendimento claro e compartilhado do que deve ser consistente e o que pode se desenvolver independentemente.

[Domain Driven Design](#) (EVANS, 2003)

No fim das contas, ao alinhar as linguagens utilizadas nas áreas de negócio aos modelos representados em código, estamos caminhando na direção de uma velha promessa: *alinhar TI com o Negócio*.

Bounded Contexts no Caelum Eats

No Caelum Eats, podemos verificar como a empresa é organizada para encontrar os Bounded Contexts e influenciar na organização do nosso código.

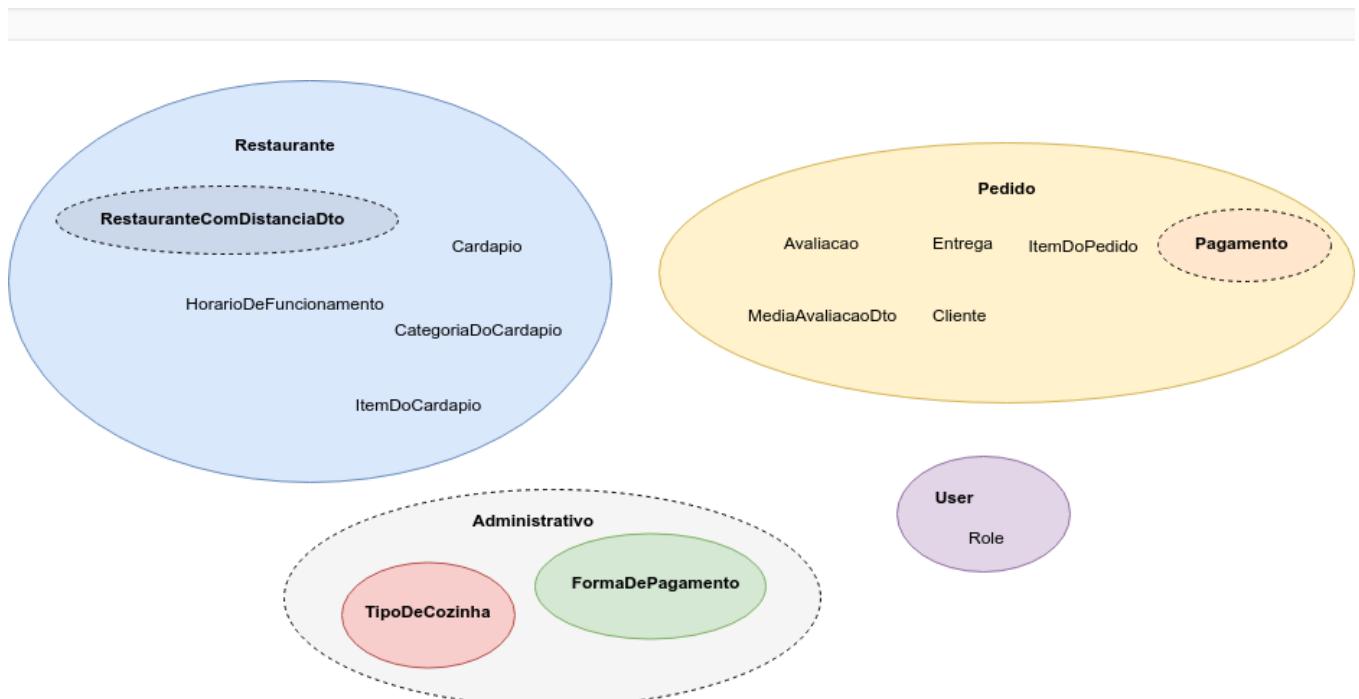
Digamos que a Caelum Eats tem, atualmente, as seguintes áreas de negócio:

- *Administrativo*, que mantém os cadastros básicos como tipos de cozinha e formas de pagamento aceitas, além de aprovar novos restaurantes
- *Pedido*, que acompanha e dá suporte aos pedidos dos clientes
- *Pagamento*, que cuida da parte Financeira e está explorando novos meios de pagamento como criptomoedas e QR Code
- *Distância*, que contém especialistas em geoprocessamento
- *Restaurante*, que lida com tudo que envolve os donos do restaurantes

Esses seriam os Bounded Contexts, que definem uma fronteira para um Modelo de Domínio consistente.

Poderíamos reorganizar o código para que a estrutura básica de pacotes seja parecida com a seguinte:

```
eats
└── administrativo
└── distancia
└── pagamento
└── pedido
└── restaurante
```



Há ainda o código de Segurança, um domínio bastante técnico que cuida

de um requisito transversal (ou não-funcional), utilizado por todas as outras partes do sistema.

Poderíamos incluir um pacote responsável por agrupar o código de segurança:

```
eats
└── administrativo
└── distancia
└── pagamento
└── pedido
└── restaurante
└── seguranca
```

Nem tudo precisa ser público

Em Java, uma classe pode ter dois modificadores de acesso: `public` ou `default`, o padrão, quando não há um modificador de acesso.

Uma classe `public` pode ser acessada por classes de qualquer outro pacote.

Já no caso de classes com modificador de acesso padrão, só podem ser acessadas por outras classes do mesmo pacote. É o que alguns chamam de *package-private*.

No caso de atributos, métodos e construtores, além de `public` e `default`, há o modificador `private`, que restringe acesso a própria classe, e `protected`, que restringe ao mesmo pacote ou classes filhas mesmo se estiverem em outros pacotes.

Pense nos projetos Java em que você trabalhou: quantas vezes você criou uma classe que não é pública?

Provavelmente, é uma implicação de organizarmos o código usando Package by Layer. Como o Controller está em um pacote, a entidade em

outro e o Repository em outro ainda, precisamos tornar as classes públicas.

Uma outra coisa que nos "empurra" na direção de todas as classes serem públicas são as IDEs: o Eclipse, por exemplo, coloca o `public` por padrão.

Porém, se alinharmos os pacotes ao domínio, passamos a ter a opção de deixar as classes acessíveis apenas no pacote em que são definidas.

Por exemplo, podemos agrupar no pacote `br.com.caelum.eats.pagamento` as seguintes classes relacionadas ao contexto de Pagamento:

```
##### br.com.caelum.eats.pagamento
```

```
.  
└── PagamentoController.java  
└── PagamentoDto.java  
└── Pagamento.java  
└── PagamentoRepository.java
```

Algumas classes como `FormaDePagamento` e `Restaurante` são usadas por classes de outros pacotes. Essas devem ser deliberadamente tornadas públicas.

Há o caso da classe `DistanciaService`, que utiliza diretamente `RestauranteRepository`. Ou seja, temos uma classe de um contexto (distância) usando um detalhe de BD de outro contexto (restaurante). É interessante manter `RestauranteRepository` com o modificador padrão e criar uma classe `RestauranteService`, responsável por expôr funcionalidades para outros pacotes.

Exercício opcional: decomposição em pacotes

1. Baixe, via Git, o projeto do monólito decomposto em pacotes:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-pa
```

2. Crie um novo workspace no Eclipse, clicando em *File > Switch Workspace > Other*. Defina o workspace `/home/<usuario-do-curso>/workspace-pacotes`, onde `<usuario-do-curso>` é o login do curso.
3. Acesse *File > Import > Existing Maven Projects* e clique em *Next*. Em *Root Directory*, aponte para o diretório clonado no passo anterior.
4. Acesse a classe `EatsApplication` e a execute com `CTRL+F11`.
5. Certifique-se que o projeto `fj33-eats-ui` esteja sendo executado. Acesse `http://localhost:4200` e teste algumas das funcionalidades. Tudo deve funcionar como antes!
6. Analise o projeto. Veja quais classes e interfaces são públicas e quais são *package private*. Observe as dependências entre os pacotes.

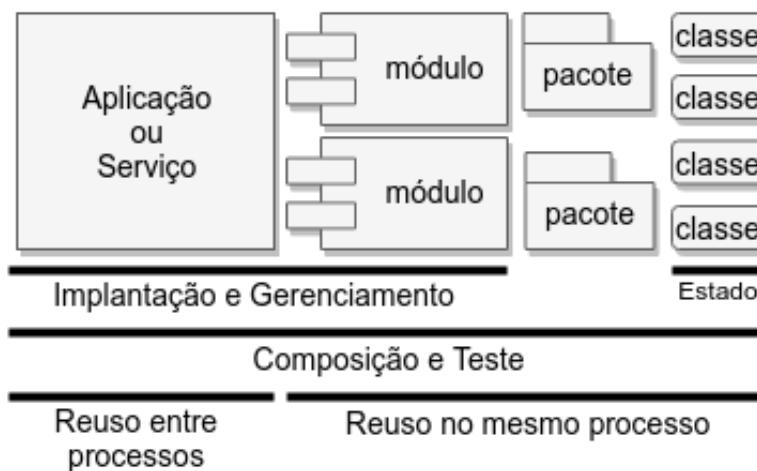
Módulos

No livro [Java Application Architecture: Modularity Patterns](#)

(KNOERNNSCHILD, 2012), Kirk Knoernschild descreve módulos como artefatos que contém as seguintes características:

- **Implantáveis**: são entregáveis que podem ser executados em *runtime*
- **Reusáveis**: são nativamente reusáveis por diferentes aplicações, sem a necessidade de comunicação pela rede. As funcionalidades de um módulo são invocadas diretamente, dentro da mesma JVM e, portanto, do mesmo processo (no Windows, o mesmo `java.exe`).
- **Testáveis**: podem ser testados independentemente, com testes de unidade.
- **Sem Estado**: módulos não mantêm estado, apenas suas classes.

- **Unidades de Composição:** podem se unir a outros módulos para compor uma aplicação.
- **Gerenciáveis:** em um sistema de módulos mais elaborado, como OSGi, podem ser instalados, reinstalados e desinstalados.



Qual será o artefato Java que contém todas essas características?

É o JAR, ou **Java ARchive**.

JARs são arquivos compactados no padrão ZIP que contém pacotes que, por sua vez, contém os `.class` compilados a partir do código fonte das classes.

Um JAR é implantável, reusável, testável, gerenciável, sem estado e é possível compô-lo com outros JARs para formar uma aplicação.

Módulos Maven

Com o Maven, é possível criarmos um **multi-module project**, que permite definir vários módulos em um mesmo projeto. O Maven ficaria responsável por obter as dependências necessárias e o fazer *build* na ordem correta. Os artefatos gerados (JARs, WARs e/ou EARs) teriam a mesma versão.

Devemos definir um módulo pai, ou supermódulo, que contém um ou mais módulos filhos, ou submódulos.

No caso do Caelum Eats, teríamos um supermódulo `eats` e submódulos para cada Bounded Context identificado anteriormente.

Além disso, precisaríamos de um submódulo que depende de todos os outros submódulos e conteria a classe principal `EatsApplication`, que possui o `main` e está anotada com `@SpringBootApplication`. Dentro desse submódulo, que chamaremos de `eats-application`, teríamos em `src/main/resources` o arquivo `application.properties` e as migrations do Flyway.

A estrutura de diretórios seria a seguinte:

```
eats
|
|   pom.xml
|
|   eats-application
|       pom.xml
|       src
|
|   eats-administrativo
|       pom.xml
|       src
|
|   eats-distancia
|       pom.xml
|       src
|
|
|   eats-pagamento
|       pom.xml
|       src
|
|
|   eats-pedido
|       pom.xml
|       src
|
```

```
|   └── eats-restaurante  
|       ├── pom.xml  
|       └── src  
|  
└── eats-seguranca  
    ├── pom.xml  
    └── src
```

O supermódulo `eats` deve definir um `pom.xml`. Nesse arquivo, a propriedade `packaging` deve ter o valor `pom`. Podem ser definidas propriedades, dependências, repositórios e outras configurações comuns a todos os submódulos. No nosso caso, definiríamos no supermódulo a dependência ao Lombok.

Os submódulos disponíveis devem ser declarados da seguinte maneira:

```
<modules>  
  <module>eats-application</module>  
  <module>eats-administrativo</module>  
  <module>eats-distancia</module>  
  <module>eats-pagamento</module>  
  <module>eats-pedido</module>  
  <module>eats-restaurante</module>  
  <module>eats-seguranca</module>  
</modules>
```

Já os submódulos não devem definir um `<groupId>` ou `<version>` próprios, apenas o `<artifactId>`. Devem declarar qual é o seu supermódulo com a tag `<parent>`. Segue o exemplo para o submódulo de restaurante:

```
<parent>  
  <groupId>br.com.caelum</groupId>  
  <artifactId>eats</artifactId>  
  <version>0.0.1-SNAPSHOT</version>
```

```
</parent>
```

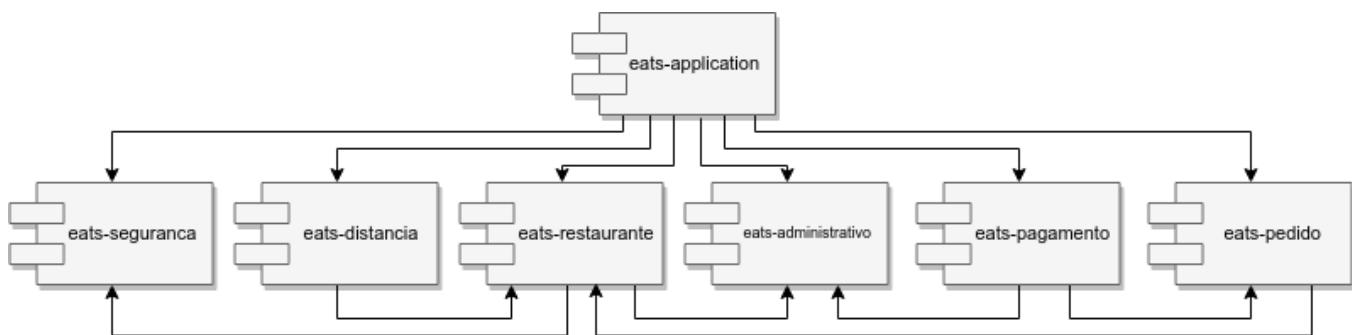
```
<artifactId>eats-restaurante</artifactId>
```

Os arquivos `pom.xml` dos submódulos podem definir em seus `pom.xml` suas próprias configurações, como propriedades, dependências e repositórios.

Se houver uma dependência a outros submódulos, é possível usar `${project.version}` como versão. Por exemplo, o submódulo `eats-restaurante` deve declarar a dependência ao submódulo `eats-administrativo` da seguinte maneira:

```
<dependency>
  <groupId>br.com.caelum</groupId>
  <artifactId>eats-administrativo</artifactId>
  <version>${project.version}</version>
</dependency>
```

É interessante notar que, nos arquivos `pom.xml`, há uma *materialização em código* da estrutura de dependências entre os módulos do sistema. Observando as dependências declaradas entre os módulos, podemos montar um diagrama parecido com o seguinte:



O submódulo `eats-application` depende de todos os outros e contém a classe principal, que contém o `main`. Ao executarmos o build, com o comando `mvn clean package` no diretório do supermódulo `eats`, o *Fat*

JAR do Spring Boot é gerado no diretório `target` do `eats-application`, contendo o código compilado da aplicação e de todas as bibliotecas utilizadas.

Pragmatismo e (um pouco de) Duplicação

Onde colocar dependências comuns a todos os módulos, como os starters do Web, Validation e Spring Data JPA?

Onde colocar classes comuns à maioria dos módulos, como a `ResourceNotFoundException`, uma exceção que gerará o status HTTP 404 (`Not Found`) quando lançada?

Muitos projetos colocam esses códigos comuns em um módulo (ou pacote) chamado `common` ou `util`. Assim evitariíamos duplicação. Afinal de contas, um lema importante no design de código é *Don't Repeat Yourself* (não se repita).

Porém, criar um módulo `common` seria inserir mais uma dependência à maioria dos outros módulos.

Uma eventual extração de um módulo para outro projeto levaria à necessidade de carregar junto o conteúdo do módulo `common`.

E, possivelmente, o módulo `common` acabaria com código útil para apenas alguns módulos específicos e não para outros.

Talvez fosse mais interessante extrair esse código para bibliotecas externas (JARs), bem focadas em necessidades específicas: uma para gráficos, outra para relatórios.

Uma ideia, visando tornar os módulos o mais independentes o possível, é aceitar um pouco de duplicação. Trocaríamos o purismo da qualidade de código por pragmatismo, pensando nos próximos passos do projeto.

Exercício: o monólito decomposto em módulos Maven

1. Clone o projeto com a decomposição do monólito em módulos
Maven:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-mc
```

2. Crie o novo workspace `/home/<usuario-do-curso>/workspace-monolito-modular` no Eclipse, clicando em *File > Switch Workspace > Other*. Troque `<usuario-do-curso>` pelo login do curso.
3. Importe, pelo menu *File > Import > Existing Maven Projects* do Eclipse, o projeto `fj33-eats-monolito-modular`.
4. Para executar a aplicação, acesse o módulo `eats-application` e execute a classe `EatsApplication` com `CTRL+F11`. Certifique-se que as versões anteriores do projeto não estão sendo executadas.
5. Com o projeto `fj33-eats-ui` no ar, teste as funcionalidades por meio de `http://localhost:4200`. Deve funcionar!
6. Observe os diferentes módulos Maven. Note as dependências entre esses módulos, declaradas nos `pom.xml` de cada módulo. Note se há alguma duplicação entre os módulos.

O Monólito Modular

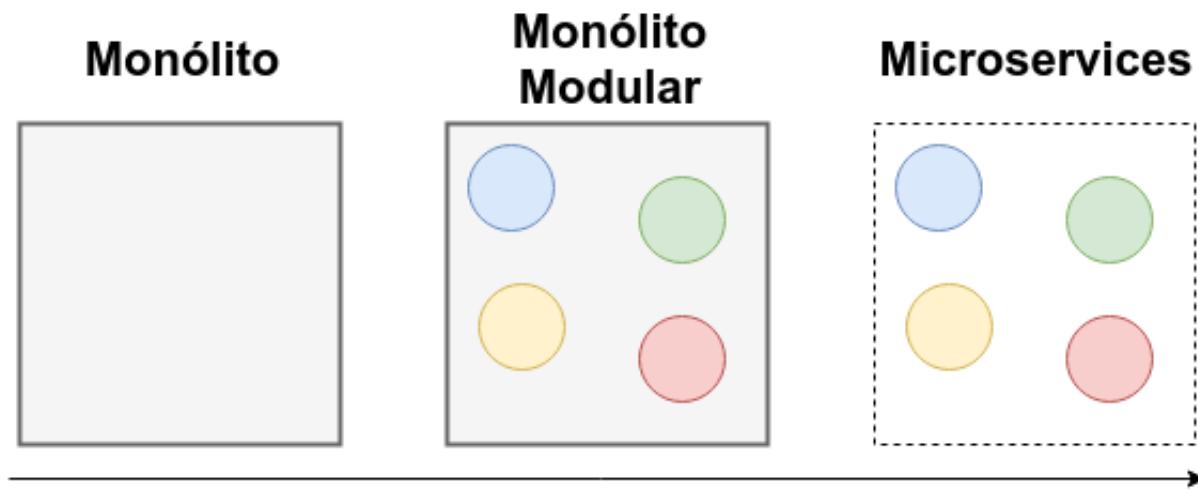
Simon Brown, na sua palestra [Modular monoliths](#) (BROWN, 2015), diz que há uma premissa comum, mas não explícita, no mercado de que todo monólito é um spaghetti e que o único caminho para um código organizado seria a migração para uma arquitetura de microservices. Só que há uma bagagem enorme de conhecimento sobre como componentizar e melhorar a manutenibilidade de um monólito. Estudamos algumas dessas ideias nesse capítulo.

Um **Monólito Modular** poderia ter diversas das características desejáveis em um código:

- alta coesão

- baixo acoplamento
- dados encapsulados
- substituíveis e passíveis de composição
- foco no negócio, inspirados por Agregados ou Bounded Contexts

Além disso, um Monólito Modular seria **um passo na direção de uma Arquitetura de Microservices**, mas sem as complexidades de um sistema distribuído.



Kirsten Westeinde, desenvolvedora sênior da Shopify, no post [Deconstructing the Monolith](#) (WESTEINDE, 2019), conta como a empresa chegou a um ponto em que a base de código monolítica, uma das maiores aplicações Ruby On Rails do mundo, começou a tornar-se tão frágil e difícil de entender, que código novo tinha consequências inesperadas e que novos desenvolvedores precisavam aprender muita coisa antes de entregar pequenas mudanças. Foi avaliada uma Arquitetura de Microservices mas foi identificado que o problema, na verdade, estava na falta de barreiras entre contextos diferentes do código. Então, escolheram um monólito modular.

Alguns times passam para uma migração para Microservices, mas resolvem voltar atrás, para um monólito modular. É o caso da Beep Saúde, como é contado por Luiz Costa, CTO, e outros desenvolvedores da Beep no episódio [Monolitos modulares e vacinas](#) (COSTA et al., 2019) do podcast Hipsters On The Road. A Beep teve que escolher entre focar os desenvolvedores em resolver os problemas técnicos trazidos por uma

Arquitetura de Microservices ou desenvolver novas funcionalidades, algo imprescindível em uma startup. Resolveram voltar para o monólito, mas de maneira modular, inspirados pela Arquitetura Hexagonal, pela Clean Architecture e pelos Bounded Contexts do DDD.

No final da palestra [Modular monoliths](#) (BROWN, 2015), Simon Brown faz uma provocação:

"Se você não consegue construir um Monólito Modular, porque você acha que microservices são a resposta"?

Porque modularizar?

No livro [Modular Java](#) (WALLS, 2009), Craig Walls cita algumas vantagens de modularizar uma aplicação:

- capacidade de trocar um módulo por outro, com uma implementação diferente, desde que a interface pública seja mantida
- facilidade de compreensão de cada módulo individualmente
- possibilidade de desenvolvimento em paralelo, permitindo que tarefas sejam divididas entre diferentes times
- testabilidade melhorada, permitindo um outro nível de testes, que trata um módulo como uma unidade
- flexibilidade, permitindo o reuso de módulos em outras aplicações

Talvez essas vantagens não sejam oferecidas pelo uso de módulos Maven. Nos textos complementares a seguir, discutiremos tecnologias como o Java Module System, a Service Loader API e OSGi, que auxiliariam a atingir essas características. Cada uma dessas tecnologias, porém, traz novas dificuldades de aprendizado, de configurações e de operação.

O que é um monólito, afinal?

Muitas vezes o termo "monólito" é usado como um *antagonista terrível*.

Mas, para uma aplicação pequena, mesmo um monólito clássico organizado em camadas não é um problema. Há diversas vantagens:

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

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

Algumas são complicações no desenvolvimento:

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

Há também complicações na operação:

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

É possível resolver parte dos problemas através de técnicas de modularização, implementando um Monólito Modular.

Um **Monólito Modular** seria composto por **componentes independentes**, que colaboram entre si através de contratos idealmente definidos em abstrações (em Java, interfaces ou classes abstratas). Com um **sistema de módulos** como o Java Module System (Java 9+) ou OSGi, é possível **reforçar as barreiras arquiteturais** mesmo com classes públicas, explicitando e limitando as dependências entre os módulos. Isso levaria a um **melhor gerenciamento das dependências** entre os módulos.

Já através de uma **Arquitetura de Plugins**, a base de código seria “fatiada” em módulos menores. Poderíamos ter **equipes mais enxutas**, focadas em cada módulo, trabalhando em várias **bases de código distintas**. A aplicação seria composta a partir de diferentes módulos em runtime. Cada desenvolvedor passaria a trabalhar com apenas uma parte do código, suavizando a IDE. Porém, subir a aplicação como um todo

ainda seria algo relativamente demorado, já que seria preciso colocar todos os módulos no ar. Os testes poderiam ser agilizados, concentrando-se em cada módulo separadamente, mas teria que ser feito um teste de sistema, que avaliaria a integração entre todos os módulos na aplicação.

Por meio de uma solução como **OSGi**, até seria possível **atualizar um módulo sem parar a aplicação** como um todo.

Monólitos Poliglotas

Um monólito não está restrito a apenas uma tecnologia de persistência, já que pode ter diferentes *data sources* para diferentes paradigmas: BDs relacionais, BDs orientados a documentos, BDs orientados a grafos, etc.

Com tecnologias como a [Graal VM](#), é possível criar monólitos desenvolvidos em múltiplas linguagens de diferentes plataformas: linguagens da JVM como Java, Scala, Kotlin, Groovy e Clojure; linguagens baseadas em LLVM como C e C++; e linguagens como JavaScript, Ruby, Python e R.

Uma definição de monólito

Como estudamos nesse capítulo, um monólito não é sinônimo de código mal feito. É possível implementar um monólito com código organizado, fatiado em pequenos pedaços independentes e alinhados com o negócio. Podemos até compor esses pedaços de diferentes maneiras e implementá-los de maneira poliglotas. Dependendo da tecnologia utilizada, podem ser atualizados parcialmente sem derrubar toda a aplicação. E podemos escalar um monólito, executando múltiplas instâncias por trás de um Load Balancer. Então o que define um monólito?

Para responder, precisamos pensar em qual é o “entregável” de uma aplicação: qual é o artefato que implantamos no ambiente de produção?

Na plataforma Java, há algum tempo, o artefato mais comum era um WAR implantado em um servlet container como Tomcat ou Jetty. Já com frameworks como o Spring Boot, temos um JAR com as dependências embutidas, um *fat JAR*. Em uma Arquitetura de Plugins, seja com Service Loader API ou com OSGi, temos uma coleção de JARs.

Em qualquer uma das maneiras de implantar um monólito, a comunicação entre as "fatias" do código é feita por chamadas de métodos, *in-memory*. E isso só é possível porque cada instância é executada em um mesmo processo.

MONÓLITO

Um monólito, modular ou não, é um sistema em que uma instância do back-end de toda a aplicação é executada em um mesmo processo do sistema operacional.

Para saber mais: Limitações dos Módulos Maven e JARs

Os módulos Maven ajudam a organizar o código de aplicações maiores porque fornecem uma maneira de representar a decomposição modular do domínio. Além disso, as dependências ficam materializadas nas declarações dos `pom.xml` de cada módulo.

Uma desvantagem dos módulos Maven como utilizamos no exemplo do exercício anterior é que **a base de código é uma só**. Diferentes times poderiam estar focados em módulos diferentes mas teriam acesso ao código dos outros módulos. A tentação de alterar algo de um módulo de outro time é muito forte! Para resolver isso, poderíamos implementar os módulos como bibliotecas completamente separadas. Um problema que iria surgir é como controlar a versão de cada módulo.

Outra desvantagem é que, em *runtime*, **a JVM não possui uma maneira de atualizar módulos** (JARs). Para atualizá-los, teríamos que parar a JVM e iniciá-la novamente, o que tornaria a aplicação indisponível. Num

ambiente com diversos times entregando software em taxas diferentes múltiplas vezes por dia/semana, a disponibilidade da aplicação seria terrivelmente afetada.

Ainda outra desvantagem dos módulos Maven é que **um módulo tem acesso às suas dependências transitivas**, ou seja, às dependências de suas dependências. Por exemplo, o módulo `eats-distancia` depende de `eats-restaurante` que, por sua vez, depende de `eats-administrativo`. Portanto, o módulo `eats-distancia` tem como dependência transitiva o módulo `eats-administrativo` e tem acesso a qualquer uma de suas classes públicas, como `FormaDePagamento` e `TipoDeCozinha`. Além disso, em `eats-distancia`, temos acesso a qualquer biblioteca declarada como dependência nos módulos `eats-restaurante` e `eats-administrativo`.

Essa fronteira fraca entre os módulos Maven pode ser problemática, já que leva a dependências indesejadas e não previstas. Se somarmos a isso o fato de que quase sempre definimos classes como públicas, módulos com muitas dependências transitivas terão acesso a boa parte das classes de outros módulos e às bibliotecas utilizadas por esses módulos. O código pode sair do controle.

Uma solução é limitar as classes públicas ao mínimo possível, tornando as classes acessíveis apenas ao pacote em que estão definidas. Mas para módulos mais complexos, teríamos dezenas ou centenas de classes no mesmo pacote! Observe, por exemplo, o módulo `eats-restaurante`: são 26 classes no mesmo pacote. Passa a ficar difícil de entender o código.

Os problemas do Classpath

Na verdade, há uma limitação nos JARs, que são apenas ZIPs com arquivos `.class` e de configuração organizados em diretórios (pacotes).

Uma vez que os JARs disponíveis são vasculhados e uma classe é carregada por um `ClassLoader` na JVM, perde-se o conceito de módulo.

O Classpath da JVM é apenas uma lista de classes, sem qualquer referência a seu JAR de origem ou de quais outros JARs dependem.

A ausência de algo que represente JAR de origem e suas dependências no Classpath enfraquece o encapsulamento em uma aplicação Java modularizada.

Nicolai Parlog, no livro [The Java Module System](#) (PARLOG, 2018), elenca os seguintes problemas no Classpath:

- *Encapsulamento fraco entre JARs*: conforme estudamos, o Classpath é uma grande lista de classes. As classes públicas são visíveis por quaisquer outras. Não é possível criar uma funcionalidade visível dentro de todos os pacotes de um JAR, mas não fora dele.
- *Ausência de representação das dependências entre JARs*: não há como um JAR declarar de quais outros JARs ele depende apenas com o Classpath.
- *Ausência de checagens automáticas de segurança*: o encapsulamento fraco dos JARs permite que código malicioso acesse e manipule funcionalidade crítica. Porém, é possível implementar manualmente checagens de segurança.
- *Sombreamento de classes com o mesmo nome*: no caso de JARs que definem duas classes com o mesmo nome, apenas uma delas é tornada disponível. E não é possível saber qual.
- *Conflitos entre versões diferentes do mesmo JAR*: duas versões do mesmo JAR no Classpath levam a comportamentos imprevisíveis.
- *JRE é rígida*: não é possível disponibilizar no Classpath um subconjunto das bibliotecas padrão do Java. Muitas classes não utilizadas ficam acessíveis.
- *Performance ruim no startup*: apesar dos class loaders serem *lazy*, carregando classes só no seu primeiro uso, muitas classes já são carregadas ao iniciar uma aplicação.
- *Class loading complexo*

Para saber mais: JPMS, um sistema de módulos para o Java

A partir do Java 9, o Java inclui um sistema de módulos bastante poderoso: o Java Platform Module System (JPMS).

Um módulo JPMS é um JAR que define:

- um nome único para o módulo
- dependências a outros módulos
- pacotes exportados, cujos tipos são acessíveis por outros módulos

Com um módulo JPMS, conseguimos definir **encapsulamento no nível de pacotes**, escolhendo quais pacotes são ou não exportados e, em consequência, acessíveis por outros módulos.

A JDK modularizada

Um dos grandes avanços do JPMS, disponível a partir da JDK 9, foi a modularização da própria plataforma Java.

O JPMS é resultado do projeto *Jigsaw*, criado em 2009, no início do desenvolvimento da JDK 7.

Antes do Java 9, todo o código das bibliotecas padrão da JDK ficava apenas no módulo de *runtime*: o `rt.jar`.

O estudo inicial do projeto *Jigsaw* agrupou o código já existente da JDK em diferentes módulos. Por exemplo, foi identificado um módulo base, que conteria pacotes fundamentais como o `java.lang` e `java.io`; um módulo desktop, com as bibliotecas Swing, AWT; além de módulos para APIs como Java Logging, JMX, JNDI.

A análise das dependências entre os módulos identificados na JDK 7 levou à descoberta de ciclos como: o módulo base depende de Logging que depende de JMX que depende de JNDI que depende de desktop que, por sua vez, depende do base.

O código da JDK 8 foi reorganizado para que não houvessem ciclos e dependências indevidas, mesmo que ainda sem um sistema de módulos propriamente dito. Os pacotes que pertenceriam ao módulo base não teriam mais dependências a nenhum outro módulo.

Na JDK 9, foram definidos módulos JPMS para cada parte do código da JDK:

- `java.base`, contendo código de pacotes como `java.lang`, `java.math`, `java.text`, `java.io`, `java.net` e `java.nio`.
- `java.logging`, com a Java Logging API.
- `java.management`, com a Java Managing Extensions (JMX) API.
- `java.naming`, com a Java Naming and Directory Interface (JNDI) API.
- `java.desktop`, contendo código de bibliotecas como Swing, AWT, 2D.

As dependências entre os módulos JPMS da JDK foram organizadas de maneira bem cuidadosa.

Antes da JDK 9, toda aplicação teria disponível todos os pacotes de todas as bibliotecas do Java. Não era possível depender de menos que a totalidade da JDK.

A partir da JDK 9, aplicações modularizadas com JPMS podem escolher, no arquivo `module-info.java`, de quais módulos da JDK dependerão.

Para saber mais: Módulos plugáveis

Antigamente, antes do Java SE 6, para ligar um plugin de uma aplicação a uma implementação era necessário:

- criar uma solução caseira usando a Reflection API
- usar bibliotecas como [JPF](#) ou [PF4J](#)
- usar uma especificação robusta, mas complexa, como [OSGi](#)

Porém, a partir do Java SE 6, a própria JRE contém uma solução: a

Service Loader API.

Na Service Loader API, um ponto de extensão é chamado de *service*.

Para provermos um service precisamos de:

- **Service Provider Interface (SPI)**: interfaces ou classes abstratas que definem a assinatura do ponto de extensão.
- **Service Provider**: uma implementação da SPI.

Para ligar a SPI com seu *service provider*, o JAR do provider precisa definir o *provider configuration file*: um arquivo com o nome da SPI dentro da pasta `META-INF/services`. O conteúdo desse arquivo deve ser o *fully qualified name* da classe de implementação.

No projeto que define a SPI, carregamos as implementações usando a classe `java.util.ServiceLoader`.

A classe `ServiceLoader` possui o método estático `load` que recebe uma SPI como parâmetro e, depois de vasculhar os diretórios `META-INF/services` dos JARs disponíveis no Classpath, retorna uma instância de `ServiceLoader` que contém todas as implementações.

O `ServiceLoader` é um `Iterable` e, por isso, pode ser percorrido com um `for-each`. Caso não haja nenhum service provider para a SPI, o `ServiceLoader` se comporta como uma lista vazia.

Perceba que uma mesma SPI pode ter vários service providers, o que traz bastante flexibilidade.

Uma Arquitetura de Plugins

Com o uso de SPIs e Service Providers, é possível criar uma Arquitetura de Plugins com a plataforma Java.

Com a Service Loader API, a simples presença de um `.jar` que a implemente a abstração do plugin (ou SPI) fará com que o

comportamento da aplicação seja estendido, sem precisarmos modificar nenhuma linha de código.

Em seu artigo [Microservices and Jars](#) (MARTIN, 2014), Uncle Bob escreve:

Não pule para Microservices só porque parece legal. Antes, segregue o sistema em JARs usando uma Arquitetura de Plugins. Se isso não for suficiente, considere a introdução de fronteiras entre serviços (service boundaries) em pontos estratégicos.

Várias bibliotecas das mais usadas por desenvolvedores Java usam SPIs.

Por meio da SPI `javax.persistence.spi.PersistenceProvider`, bibliotecas como o Hibernate e o EclipseLink fornecem implementações para as interfaces do pacote `javax.persistence`.

Do Java SE 6 em diante, a classe [DriverManager](#) carrega automaticamente todas as implementações da SPI `java.sql.Driver`. Por exemplo, o `mysql-connector-java.jar` do MySQL fornece um Service Provider para essa SPI.

O Spring tem a sua própria implementação de algo semelhante a Service Loader API: a classe [SpringFactoriesLoader](#). Essa classe é usada pelas Auto-Configurations do Spring Boot, fazendo com que a simples presença dos `.jar` dos starters adicionem comportamento à aplicação. Ou seja, o próprio Spring Boot é uma Arquitetura de Plugins.

Para saber mais: OSGi

O Java Module System foi disponibilizado a partir de 2017, com o lançamento do Java 9. Porém, a plataforma Java já tinha uma solução mais poderosa desde 1999: a especificação OSGi (Open Services Gateway Initiative). Essa especificação é implementada por frameworks como Apache Felix, Eclipse Equinox, entre outros. IDEs como Eclipse e NetBeans, servidores de aplicação como GlassFish e WebSphere, são

implementadas usando frameworks OSGi.

Por meio de diferentes Class Loaders, um framework OSGi traz a ideia de módulos para o *runtime* da JVM, corrigindo falhas do Classpath. Dessa maneira, provê um nível de encapsulamento além dos pacotes.

Um módulo, no OSGi, é chamado de *bundle*. Bundles são JARs, só que com metadados adicionais no `META-INF/MANIFEST.MF` como o nome do bundle, a versão, de quais outros bundles depende, entre outros detalhes.

Um framework OSGi controla o ciclo de vida de um bundle, fazendo com que seja instalado, iniciado, atualizado, parado e desinstalado. Múltiplas versões de um bundle podem coexistir em runtime e um bundle pode ser trocado sem parar toda a JVM.

O OSGi também especifica o conceito de *service*, análogo à Service Loader API do Java: um bundle define interface pública e outros bundles, uma ou mais implementações. Para ligar as implementações à interface, um framework OSGi provê um *service registry*. Novas implementações podem ter seu registro feito ou cancelado dinamicamente, sem parar a JVM. Os consumidores de um service dependeriam apenas da interface e do service registry, sem ter acesso a detalhes de implementação.

O nível de encapsulamento de um bundle e a possibilidade de **atualizar e registrar implementações dinamicamente** permitiria que diferentes times cuidassem de diferentes bundles alinhados com os Bounded Contexts (e as áreas de negócio) da organização. A implantação de novas versões de um bundle poderia ser feita sem derrubar a aplicação como um todo.

Porém, OSGi traz alguns problemas que limitaram sua adoção em aplicações e restringiram o uso basicamente a criadores de middleware e IDEs. Há quem diga que OSGi aumenta drasticamente o uso de memória, talvez pela implementação de diferentes Class Loaders, mas há soluções com baixo uso de memória, como de IoT, que usam implementações

OSGi. Ross Mason, criador do Mule ESB, escreve no artigo [OSGi? No Thanks](#) (MASON, 2010) que o principal dos problemas é a curva de aprendizado e a complexidade, com a necessidade de diversas configurações cheias de detalhes técnicos, o que afeta negativamente a experiência do desenvolvedor.

Um detalhe interessante é que Craig Walls, no livro [Modular Java](#) (WALLS, 2009), cita o termo *SOA in a JVM* como uma maneira usada para descrever os services do OSGi.

Um post sobre services OSGi de 2010, no blog da OSGi Alliance, usou pela primeira vez o termo [μServices](#) (KRIENS, 2010), com a letra grega mu (μ) que é usada como símbolo do prefixo *micro* pelo Sistema Internacional de Unidades.

No livro de 2012 [Java Application Architecture: Modularity Patterns](#) (KNOERNNSCHILD, 2012), Kirk Knoernschild usa repetidamente o termo μ Services para se referir aos services OSGi.

Extraindo serviços

O contexto que levou aos Microservices

Na década de 90, aplicações Desktop deram lugar à Web. A arquitetura Web, do estilo Cliente/Servidor com “telas” geradas pelo Servidor (*thin clients*), influenciou na maneira como o código é implantado. Com controle total dos servidores, publicações de novas versões da aplicação foram facilitadas.

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

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

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

As tecnologias de criação de máquinas virtuais permitiram o provisionamento (configuração) de máquinas virtuais por meio de

scripts, o que ficou conhecido como **infrastructure as code**. A partir de 2005, surgiram várias soluções do tipo como Puppet, Chef, Vagrant, Salt e Ansible.

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

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

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

Todo esse contexto é resumido por Sam Newman no início do livro [Building Microservices](#) (NEWMAN, 2015):

Domain-driven design. Continuous delivery. Virtualização sob demanda. Automação de infraestrutura. Times pequenos e autônomos. Sistemas em larga escala. Microservices emergiram desse mundo.

Do monólito (modular) aos Microservices

Um monólito comum, organizado com Package by Layer, pode trazer problemas para aplicações maiores: código progressivamente mais complexo, dependências indevidas, times cada vez maiores, impossibilidade de deploys sem parar a aplicação, entre outros.

Com uma estratégia de componentização baseada em módulos, a manutenção e evolução podem ser melhoradas. Com uma Arquitetura de Plugins, podemos ter pequenos times autônomos. Com runtimes como OSGi, podemos até fazer hot-deploy, atualizando módulos com a aplicação no ar. Com diferentes datasources, podemos explorar diferentes tecnologias de persistência. Com plataformas como a JVM, é possível o uso de linguagens de diversos paradigmas diferentes.

Mas, ainda assim, o monólito é executado como um único processo. Se houver alguma falha na memória ou bug que cause uso massivo de CPU ou um loop infinito, a aplicação toda sairá do ar.

Se houver um aumento na carga, é possível sim escalar um monólito: basta colocarmos um cluster de instâncias, com requests alternados por um Load Balancer. Porém, pode haver uma subutilização de recursos, já que a replicação será de toda a aplicação e não daquelas partes em que há mais necessidade de CPU ou memória. E, como o código será o mesmo, bugs que derrubam a aplicação poderão ser replicados por todos os nós do cluster.

Na palestra [Evoluindo uma Arquitetura inteiramente sobre APIs](#) (CALÇADO, 2013), Phil Calçado diz de maneira bem clara o grande problema de um monólito:

Quando você tem uma base de código só, você é tão estável quanto a sua parte menos estável.

E, convenhamos, um monólito modular é algo raríssimo no mercado. Então, para a maioria dos monólitos há problemas com times grandes, deploys e complexidade do código. Mas um monólito modular, com complexidade e dependências gerenciadas, facilitará uma possível migração para serviços.

Simon Brown diz na palestra [Modular monoliths](#) (BROWN, 2015):

Escolha Microservices por seus benefícios, não por que sua base de código monolítica é uma bagunça.

Componentização em serviços

Uma **Arquitetura de Microservices** traz uma abordagem diferente de componentização: a aplicação é decomposta em diversos **serviços**.

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

Microservices:

- são executados em diferentes processos em máquinas (ou containers) distintos
- a comunicação é interprocessos, pela rede
- o deploy é independente, por definição
- podem usar múltiplos mecanismos de persistência em paradigmas diversos (Relacional, NoSQL, etc), desde que não compartilhados com outros serviços
- há diversidade tecnológica, sendo a única restrição a possibilidade de prover uma API em um protocolo padrão (HTTP, AMQP, etc)
- há uma fronteira fortíssima entre as bases de código

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), define Microservices da seguinte maneira:

Microservices são serviços pequenos e autônomos que trabalham juntos.

Prós e contras de uma Arquitetura de Microservices

PRÓ: Times Pequenos e Autônomos

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

afeta positivamente a velocidade de desenvolvimento de novas funcionalidades.

Uma monólito com uma Arquitetura de Plugins talvez permita o desenvolvimento de diferentes módulos por times diversos em torno de um núcleo comum. Porém, é uma raridade no mercado.

PRÓ: “Trocabilidade”

É comum termos aquele sistema legado em que ninguém toca e que ninguém sabe dar manutenção. E isso acontece porque qualquer mudança ficou muito arriscada.

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

É algo que um monólito modular também permitiria.

PRÓ: Reuso e composibilidade

Diferentes serviços podem ser compostos em novos serviços, atendendo com agilidade às demandas do negócio. É uma velha promessa do SOA (Service-Oriented Architecture).

A Uber é um exemplo disso: provê um serviço de transporte urbano, mas lançou há algum tempo um serviço de fretes de caminhões e outro de entrega de comida. Provavelmente, reutilizaram serviços de pagamentos, de geolocalização, entre outros.

Um monólito modular é uma outra maneira de atingir isso, sem termos um Sistema Distribuído.

PRÓ: Fronteiras fortes entre componentes

Se usarmos serviços como estratégia de componentização, ao invés de simples pacotes ou módulos, teremos uma separação fortíssima entre o código de cada componente.

Em um monólito não modular é tentador tomar atalhos para entregar funcionalidades mais rápido, esquecendo das fronteiras entre componentes. Mesmo em monólitos modulares, dependendo do sistema de módulos utilizado, é possível acessar em *runtime* (e até via código) funcionalidades de outros módulos, talvez por meio de *workarounds* (as famosas gambiarras). Porém, há sistemas de módulos como o Java Module System do Java 9+ e o OSGi, que reforçaram as barreiras entre código de módulos diferentes tanto em desenvolvimento como em *runtime*.

Uma vantagem de uma Arquitetura de Microservices é que temos esse fronteira fortes entre componentes mantendo a estrutura de cada serviço parecida com a que estamos acostumados. É possível usar o bom e velho *Package by Layer*, já que o código de cada serviço é focado em um conjunto específico de funcionalidades.

Um ponto de atenção é o acesso a dados. Em uma Arquitetura de Microservices, cada serviço tem o seu BD separado. E isso reforça bastante a componentização dos dados, eliminando os perigos de uma integração pelo BD que pode levar a um acomplamento indesejado. Por outro lado, ao separar os BDs, perdemos muitas coisas, que discutiremos adiante.

PRÓ: Diversidade tecnológica e experimentação

Em uma aplicação monolítica, as escolhas tecnológicas iniciais restringem as linguagens e frameworks que podem ser usados.

Com Microservices, partes do sistema podem ser implementadas em tecnologias que estejam mais de acordo com o problema a ser resolvido. Os protocolos de integração entre os serviços passam a ser as partes mais importantes das escolhas tecnológicas.

Essa heterogeneidade tecnológica permite que soluções performáticas e com mais funcionalidades sejam utilizadas.

Em seu artigo [Microservice Trade-Offs](#) (FOWLER, 2015a), Martin Fowler diz que coisas prosaicas como atualizar a versão de uma biblioteca podem ser facilitadas. Em um monólito, temos que usar a mesma versão para todo a aplicação e os upgrades podem ser problemáticos. Ou tudo é atualizado, ou nada. Quanto maior a base de código, maior é o problema nas atualizações de bibliotecas.

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), argumenta: *Uma das grandes barreiras para testar e adotar novas tecnologias são os riscos associados. Em um monólito, qualquer mudança impactará uma quantia grande do sistema. Com um sistema que consiste de múltiplos serviços, há múltiplos lugares para testar novas tecnologias. Um serviço de baixo risco pode ser usado para minimizar os riscos e limitar os possíveis impactos negativos. A habilidade de absorver novas tecnologias traz vantagens competitivas para as organizações.*

Porém, é importante ressaltar o risco de adotar muitas stacks de tecnologias completamente distintas: é difícil de entender as características de performance, confiabilidade, operações e monitoramento. Por isso, no livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman diz que empresas como a Netflix e Twitter focam boa parte dos seus esforços em usar a JVM como uma plataforma para diferentes linguagens e tecnologias. Para empresas menores, o risco de adotar tecnologias "esotéricas" é aumentado, já que pode ser difícil de contratar pessoas experientes e familiarizadas com algumas tecnologias.

É possível implementar um monólito poliglota. Mas é algo raro. No caso da JVM, podemos usar linguagens como Java, Kotlin, Scala e Clojure. Com a GraalVM, podemos até mesclar plataformas, usando algumas linguagens da JVM com algumas da LLVM, entre outras. E múltiplos datasources podem permitir o uso de mecanismos de persistências diferentes, como um BD orientado a Grafos, como o Neo4J, junto a um

BD relacional, como o PostgreSQL.

PRÓ: Deploy independente

A mudança de uma linha de código em uma aplicação monolítica de um milhão de linhas requer que toda a aplicação seja implantada para que seja feito o *release* da pequena alteração. Isso leva a deploys de alto risco e alto impacto, o que leva a medo de que alguma coisa dê errado. E esse medo leva a diminuir a frequência dos deploys, o que leva ao acúmulo de mudanças em cada deploy. E quanto mais alterações em um mesmo deploy, maior o risco de que algo dê errado. Esse *ciclo vicioso dos deploys* é demonstrado por Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015).

Com Microservices, só um pedaço do sistema fica fora do ar ao implantarmos novas versões. Isso minimiza o risco e o impacto de cada deploy, já que uma falha de um serviço e diminuiu a indisponibilidade da aplicação. A consequência é que podemos passar a fazer mais deploys em produção, talvez várias vezes por dia. Ou seja, serviços habilitam a entrega rápida, frequente, confiável de aplicações complexas.

Essa Entrega Contínua (*Continuous Delivery*, em inglês) permite que os negócios da organização reajam rápido ao feedback do cliente, a novas oportunidades e a concorrentes. A mudança cultural e organizacional para permitir isso é um dos temas do DevOps.

No artigo [Microservice Trade-Offs](#) (FOWLER, 2015a), Martin Fowler argumenta que a relação entre Microservices e Continuous Delivery/DevOps é de duas vias: para ter vários Microservices, provisionar máquinas e implantar aplicações rapidamente são pré-requisitos. Fowler ainda cita Neal Ford, que relaciona uma Arquitetura de Microservices e DevOps: *Microservices são a primeira arquitetura depois da revolução trazida pelo DevOps.*

Atingir algo parecido com um monólito é até possível com algumas tecnologias como OSGi, mas incomum. Martin Fowler diz, ainda no artigo

[Microservice Trade-Offs](#) (FOWLER, 2015a), que Facebook e Etsy são dois casos de empresas cujos monólitos têm Continuous Delivery. O autor ainda diz que entregas rápidas, confiáveis e frequentes são mais relacionadas com o uso prático de Modularidade do que necessariamente com Microservices.

PRÓ: Maior isolamento de falhas

Em um monólito, se uma parte da aplicação apresentar um vazamento de memória, pode ser que o todo seja interrompido.

Quando há uma falha ou indisponibilidade em um serviço, os outros serviços continuam no ar e, portanto, parte da aplicação ainda permanece disponível e utilizável, o que alguns chamam de *graceful degradation*.

PRÓ: Escalabilidade independente

Em um monólito, componentes que tem necessidades de recursos computacionais completamente diferentes devem ser implantados em conjunto, isso leva a um desperdício de recursos. Se uma pequena parte usa muita CPU, por exemplo, estamos restritos a escalar o todo para atender às demandas de processamento.

Com Microservices, necessidades diferentes em termos computacionais, como processamentos intensivos em termos de memória e/ou CPU, podem ter recursos específicos. Isso minimiza o impacto em outras partes da aplicação, otimiza recursos e diminui custos de operação. Quando são usados provedores de Cloud como AWS, Azure ou Google Cloud, isso levará a um corte de custos quase imediato.

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), cita o caso da Gilt, uma loja online de roupas. Começaram com um monólito Rails em 2007 que, já em 2009, não estava suportando a carga. Ao quebrar partes do sistema, a Gilt conseguiu lidar melhor com picos de tráfego.

CONTRA: Dificuldades inerentes a um Sistema Distribuído

Uma chamada entre dois Microservices envolve a rede. Uma Arquitetura de Microservices é um Sistema Distribuído.

A comunicação intraprocesso, com as chamadas em memória, é centenas de milhares de vezes mais rápida que uma chamada interprocessos dentro de um mesmo data center. Algumas das latências mostradas pelo pesquisador da Google Jeffrey Dean na palestra [Designs, Lessons and Advice from Building Large Distributed Systems](#) (DEAN, 2009):

- Referência Cache L1: 0.5 ns
- Referência Cache L2: 7 ns
- Referência à Memória Principal: 100 ns
- Round trip dentro do mesmo data center: 500 000 ns (0,5 ms)
- Roud trip Califórnia-Holanda: 150 000 000 ns (150 ms)

Uma versão mais atualizada e interativa dessa tabela pode ser encontrada em:

https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Leslie Lamport, pioneiro da teoria de Sistemas Distribuídos, brincou com a definição de Sistemas Distribuídas em uma [lista interna](#) (LAMPORT, 1987) da DEC Systems Research Center: *Um sistema distribuído é um sistema em que uma falha em um computador que você nem sabia da existência torna o seu próprio computador inutilizável.*

A performance é afetada negativamente. É preciso tomar cuidado com latência, limites de banda, falhas na rede, indisponibilidade de outros serviços, entre outros problemas. Além disso, transações distribuídas são um problema muito complexo.

A rede é lenta e instável e temos que lidar com as consequências disso.

As faláciais da Computação Distribuída

Peter Deustch e seus colegas da Sun Microsystems definiram algumas premissas falsas que desenvolvedores assumem quando tratam de aplicações distribuídas:

1. A rede é confiável
2. A latência é zero
3. A banda é infinita
4. A rede é segura
5. A topologia não muda
6. Só há um administrador
7. O custo de transporte é zero
8. A rede é homogênea

E a Primeira Lei do Design de Objetos Distribuídos?

No livro [Patterns of Enterprise Application Architecture](#) (FOWLER, 2002), Martin Fowler cunhou a *Primeira Lei do Design de Objetos Distribuídos*: não distribua seus objetos.

Isso valeria para uma Arquitetura de Microservices? Fowler responde a essa pergunta no artigo [Microservices and the First Law of Distributed Objects](#) (FOWLER, 2014a). O autor explica que o contexto da "lei" era a ideia, em voga no final dos anos 90 e no início dos anos 2000, de era possível tratar objetos intraprocesso e remotos de maneira transparente. Com o uso de CORBA, DCOM ou RMI, bastaria rodar objetos em outras máquinas, sem a necessidade de estratégias elaboradas de decomposição e migração. Considerando que a rede é lenta e instável, buscar preços de 100 produtos não teria a mesma performance e confiabilidade comparando uma chamada em memória e uma pela rede. É preciso tomar cuidado com a granularidade das chamadas e tolerância a falhas. A suposta transparência entre chamadas em memória e remotas é uma falácia tardia. Por isso, a lei proposta no livro mencionado.

Fowler, no mesmo artigo, diz que os defensores dos Microservices com que teve contato estão cientes da distinção entre chamadas em memória e pela rede e desconsideraram sua suposta transparência. As interações

entre Microservices, portanto, seriam de granularidade mais grossa e usariam técnicas como Mensageria.

Apesar de ter uma inclinação para Monólitos, Fowler diz que sua natureza de empiricista o fez aceitar que uma abordagem de Microservices teve sucesso em vários times com que trabalhou. Porém, enquanto é mais fácil pensar sobre Microservices pequenos, o autor diz preocupar-se que a complexidade é empurrada para as interações entre os serviços, onde é menos explícita, o que torna mais difícil descobrir quando algo dá errado.

Essa preocupação ressoa em Michael Feathers que, no post [Microservices Until Macro Complexity](#) (FEATHERS, 2014), diz que parece haver uma Lei da Conservação da Complexidade no software:

"Quando quebramos coisas grandes em pequenos pedaços nós passamos a complexidade para a interação entre elas."

CONTRA: Complexidade ao operar e monitorar

Configurar, fazer deploy e monitorar um monólito é fácil. Depois de gerar o entregável (WAR, JAR, etc) e configurar portas e endereços de BDs, basta replicar o artefato em diferentes servidores. O sistema está ou não fora do ar, os logs ficam apenas em uma máquina e sabemos claramente por onde uma requisição passou.

Em uma Arquitetura de Microservices, precisamos:

- agregar logs que ficam espalhados pelos diversos Microservices
- saber da “saúde” de cada um dos Microservices
- rastrear por quais Microservices passa uma requisição
- ter uma maneira de facilitar a configuração de portas e endereços de BDs e de outros Microservices
- fazer deploy dos diferentes Microservices

Já no post [Microservice Prerequisites](#) (FOWLER, 2014b), Martin Fowler descrever alguns pré-requisitos para a adoção de uma Arquitetura de

Microservices:

- **provisionamento rápido:** preparar novos servidores com os softwares, dados e configurações necessários deve ser rápido e o mais automatizado o possível. Provedores e ferramentas de Cloud ajudam muito nessa tarefa.
- **deploy rápido:** fazer o deploy da aplicação em ambientes de teste e produção deve ser algo rápido e automatizado.
- **monitoramento básico:** detectar indisponibilidade de serviços, erros e acompanhar métricas de negócio é essencial
- **cultura DevOps:** é necessária uma mudança cultural em direção a uma maior colaboração entre desenvolvedores e pessoal de infra

Se Continuous Delivery é uma prática importante para monólitos, torna-se essencial para uma Arquitetura de Microservices. Ferramentas de automação de infra-estrutura são imprescindíveis.

James Lewis diz, no [podcast SE Radio](#) (LEWIS, 2014), que:

"Nós estamos mudando a COMPLEXIDADE ACIDENTAL de dentro da aplicação para a infraestrutura. AGORA é uma boa hora para isso porque nós temos mais maneiras de gerenciar a complexidade. Infraestrutura programável, automação, tudo indo pra cloud. Nós temos ferramentas melhores para resolver esse problemas AGORA."

Complexidade Essencial x Complexidade Acidental

No clássico artigo [No Silver Bullets](#) (BROOKS, 1986), Fred Brooks separa complexidades essenciais do software, que tem a ver com o problema que está sendo resolvido (e, poderíamos dizer, com o domínio) de complexidades acidentais, que são reflexos das escolhas tecnológicas. O autor argumenta que mesmo que as complexidades acidentais fossem zero, ainda não teríamos um ganho significativo no esforço de produzir um software. Por isso, **não existe bala de prata**.

CONTRA: Perda da consistência dos dados e transações

Manter uma consistência forte dos dados em um Sistema Distribuído é extremamente difícil.

De acordo com o Teorema CAP, cunhado por Eric Brewer na publicação [Towards Robust Distributed Systems](#) (BREWER, 2000), não é possível termos simultaneamente mais que duas das seguintes características: Consistência dos dados, Disponibilidade (*Availability*, em inglês) e tolerância a Partições de rede. Ou seja, se a rede falhar, temos que escolher entre Consistência e Disponibilidade. Se escolhermos Consistência, o sistema ficará indisponível até a falha na rede ser resolvida. Se escolhermos Disponibilidade, a Consistência será sacrificada. Portanto, em um Sistema Distribuído, não temos garantias ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Em um esperto jogo de palavras com os conceitos da Química de ácido e base, Brewer diz que poderíamos ter garantias BASE (Basically Available, Soft-state, Eventually consistent): para manter um Sistema Distribuído disponível, teríamos respostas aproximadas que eventualmente ficariam consistentes.

Daniel Abadi, no paper [Consistency Tradeoffs in Modern Distributed Database System Design](#) (ABADI, 2012), cunhou o Teorema PACELC, incluindo alta latência de rede como uma forma de indisponibilidade.

No artigo [Microservice Trade-Offs](#) (FOWLER, 2015a), Martin Fowler descreve o seguinte cenário: você faz uma atualização de algo e, ao recarregar a página, a atualização não está lá. Depois de alguns minutos, você dá refresh novamente a atualização aparece. Talvez isso acontece porque a atualização foi feita em um nó do cluster mas o segundo request obteve os dados de outro nó. Eventualmente, os nós ficam consistentes, com os mesmos dados. O autor pondera que inconsistências como essa são irritantes, mas podem ser catastróficas para a Organização quando decisões de negócios são feitas com base em dados inconsistentes. E o pior: é muito difícil de reproduzir e debugar!

Em um monólito, é possível alterar vários dados no BD de maneira

consistente, usando apenas uma transação. Como cada Microservice tem o seu BD, as transações teriam que ser distribuídas, o que iria na direção da Consistência em detrimento da Disponibilidade. O mundo dos Microservices abraça a consistência eventual (em inglês, *eventual consistency*). Processos de negócio são relativamente tolerantes a pequenas inconsistências momentâneas.

CONTRA: Saber o momento correto de adoção é difícil

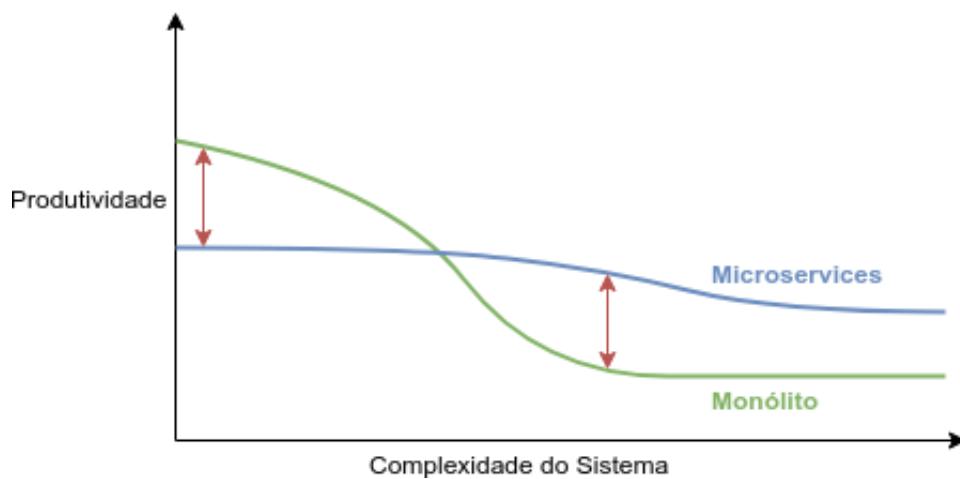
Encontrar fatias pequenas e independentes do domínio, criando fronteiras arquiteturais alinhadas com os Bounded Contexts, é difícil no começo do projeto, quando não se conhece claramente o Negócio ou as possíveis alterações. Isso é especialmente difícil para startups, que ainda estão validando o Modelo de Negócio e fazem mudanças drásticas com frequência. Aliando-se a isso as complexidade de operação, monitoramento e os desafios de um Sistema Distribuído, uma Arquitetura de Microservices pode ser uma escolha ruim para uma startup que tem uma base de usuário limitada, uma equipe reduzida e pouco financiamento.

Do ponto de vista Lean, usar uma Arquitetura de Microservices para um projeto simples ou em fase inicial pode ser considerado *overengineering*, um tipo de desperdício (Muda).

No artigo [Microservice Premium](#) (FOWLER, 2015b), Martin Fowler argumenta que uma Arquitetura de Microservices introduz uma complexidade que pode elevar os custos e riscos do projeto, como se fosse adicionado um ágio (em inglês, *premium*). O autor diz que, para projetos de baixa complexidade (essenciais, poderíamos dizer), uma Arquitetura de Microservices adiciona uma série de complicações no monitoramento, em como lidar com falhas e consistência eventual, entre outras. Já para projetos mais complexos, com um time muito grande, muitos modelos de interação com o usuário, dificuldade em escalar, partes do negócio que evoluem independentemente ou uma base de código gigantesca, vale pensar em uma Arquitetura de Microservices.

Martin Fowler conclui:

Minha principal orientação seria nem considerar Microservices, a menos que você tenha um sistema complexo demais para gerenciar como um Monólito. A maioria dos sistemas de software deve ser construída como uma única aplicação monolítica. Atenção deve ser prestada à uma boa modularização do Monólito (...) Se você puder manter o seu sistema simples o suficiente para evitar a necessidade de Microservices: faça.



Começar com um Monólito ou com Microservices?

No artigo [Monolith First](#) (FOWLER, 2015c), Martin Fowler argumenta que devemos começar com um Monólito, mesmo se você tiver certeza que a aplicação será grande e complexa o bastante para compensar o uso de Microservices. Fowler baseia o argumento em sua experiência:

Quase todas as histórias de sucesso de Microservices começaram com um Monólito que ficou muito grande e foi decomposto. Quase todos os casos de sistemas que começaram com Microservices do zero, terminaram em sérios apuros. (...) Até arquitetos experientes trabalhando em domínios familiares tem grandes dificuldades em acertar quais são as fronteiras estáveis entre serviços.

Stefan Tilkov publicou o artigo [Don't start with a monolith](#) (TILKOV, 2015) no próprio site de Martin Fowler, argumentando o contrário: é incrivelmente difícil, senão impossível, fatiar um monólito. Pra Tilkov, o que é necessário na verdade é um bom conhecimento sobre o domínio

da aplicação antes começar a particioná-lo. Outro argumento é que um Monólito bem componentizado e com baixo acoplamento é raríssimo e que uma das maiores vantagens dos Microservices é a fronteira fortíssima entre o código de cada serviço, evitando um emaranhado nas dependências. Partes do monólito comunicam entre si usando as mesmas bibliotecas, usam o mesmo modelo de persistência, podem usar transações no BD e muitas vezes compartilham objetos de domínio. Tudo isso dificulta imensamente uma possível migração posterior para uma Arquitetura de Microservices. Para o autor, em sistemas que sabe-se que serão grandes, complexos e em que o domínio é familiar, vale a pena começar a construí-los em subsistemas da maneira mais independente o possível.

Um outro argumento a favor do uso inicial de uma Arquitetura de Microservices é que, se as ferramentas de deploy, configuração e monitoramento são complexas, devemos começar a dominá-las o mais cedo o possível. Claro, se a visão é que o projeto crescerá em tamanho e complexidade.

Quão micro deve ser um Microservice?

Os serviços em uma Arquitetura de Microservices devem ser pequenos. Por isso, o “micro” no nome. Mas o que deve ser considerado “micro”? Algo menor que um miliservice ou maior que um nanoservice (termo infelizmente usado pelo mercado)? Não! O tamanho não é importante! O termo “micro” é enganoso.

Chris Richardson, no livro [Microservice Patterns](#) (RICHARDSON, 2018a) diz:

Um problema com o termo Microservice é que a primeira coisa que você ouve é micro. Isso sugere que um serviço deve ser muito pequeno. (...) Na realidade, tamanho não é uma métrica útil. Um objetivo melhor é definir um serviço bem modelado como um serviço capaz de ser desenvolvido por um time pequeno com um lead time mínimo e com mínima colaboração com outros times. Na teoria, um time deve ser

responsável somente por um serviço (...) Por outro lado, se um serviço requer um time grande ou leva muito tempo para ser testado, provavelmente faz sentido dividir o time e o serviço.

O critério para decomposição deve ser, em geral, algo alinhado com o negócio da organização. No fim das contas, o objetivo principal é alinhar negócio à TI. Um serviço pequeno é um serviço que embarca uma capacidade de negócio.

Os conceitos de Agregado e Contexto Delimitado do DDD, que vimos no capítulo anterior, vêm à nossa ajuda!

Um Microservice pode ser modelado como um Agregado ou, preferencialmente, como um Contexto Delimitado (Bounded Context) em que a linguagem do especialista de domínio será representada no código (Domain Model) sem apresentar inconsistências.

No livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman diz que devemos focar as fronteiras entre os serviços nas fronteiras do negócio. Dessa maneira, saberemos onde estará o código de uma determinada funcionalidade e evitaremos a tentação de deixar um determinado serviço crescer demais. Ao modelar de acordo com o negócio, as fronteiras ficam claras.

Phil Calçado, em [um tweet](#) (CALÇADO, 2018), diz que o critério de decomposição de uma Arquitetura de Microservices deve ser parecido com o de um monólito modular:

Eu sempre descrevo Microservices como a aplicação da mesma maneira de agrupar que você teria em uma aplicação maior, só que através de seus componentes distribuídos.

Cuidado com o Monólito Distribuído

No livro [Monolith to Microservices](#) (NEWMAN, 2019), Sam Newman define um Monólito Distribuído como um sistema que consiste de múltiplos serviços mas cujos deploys devem ser feitos ao mesmo tempo.

Na experiência do autor, um Monólito Distribuído tem todas as desvantagens de um Sistema Distribuído e todas as desvantagens de um Monólito. Para Newman, um Monólito Distribuído emerge de um ambiente em que não houve foco o suficiente em conceitos como *Information Hiding* e coesão das funcionalidades de negócio, levando a arquiteturas altamente acopladas em que mudanças se propagam através dos limites de serviço e onde mudanças aparentemente inocentes, que parecem ter escopo local, quebram outras partes do sistema.

Uma maneira comum de chegar a um Monólito Distribuído é ter serviços extremamente pequenos, chegando a um serviço por Entidade de Negócio (objetos que tem continuidade, identidade e estão representados em algum mecanismo de persistência).

Chris Richardson, no livro [Microservice Patterns](#) (RICHARDSON, 2018a), chega uma conclusão semelhante: um Monólito Distribuído é o resultado de uma decomposição incorreta dos componentes. Para Richardson, o antídoto aos Monólitos Distribuídos é seguir, só que no nível de serviços, o Common Closure Principle definido por Robert "Uncle Bob" Martin: *Agregue, em componentes, classes que mudam ao mesmo tempo e pelos mesmos motivos. Separe em componentes diferentes classes que mudam em momentos diferentes e por razões distintas.*

Microservices e SOA

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

Chris Richardson, em seu livro [Microservices Patterns](#) (RICHARDSON, 2018a), descreve SOA como sendo uma arquitetura que usa *smart pipes* como ESB, protocolos pesados como SOAP e WS-*, Persistência

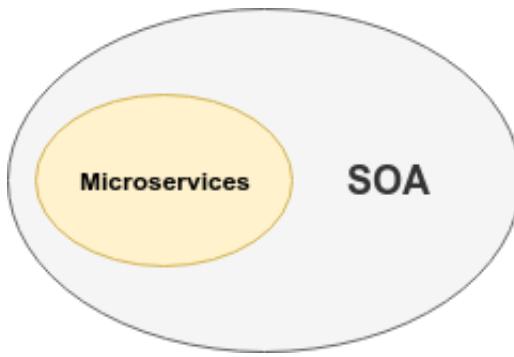
centralizada em BDs corporativos e serviços de granularidade grossa. Talvez seja a versão mais comum de SOA que vemos implementada nas organizações.

Martin Fowler e James Lewis dizem em seu artigo sobre [Microservices](#) (FOWLER; LEWIS, 2014), que há uma grande ambiguidade sobre o que SOA realmente é e, dependendo da definição, uma Arquitetura de Microservices é SOA, mas pode não ser. Talvez seria “SOA do jeito certo”. Algumas características a distinguem do SOA implementado em grandes organizações: governança e gerenciamento de dados descentralizado; mais inteligência nos Microservices (*smart endpoints*) e menos nos canais de comunicação (*dumb pipes*). Um ESB seria um *smart pipe*, já que faz roteamento de mensagens, transformações, orquestração e até algumas regras de negócio. Ainda citam em um rodapé que a Netflix, uma das referências em Microservices, inicialmente chamava sua abordagem de *fine-grained SOA*.

Henrique Lobo mostra em seu artigo [Repensando micro serviços](#) que SOA como descrito pelo consórcio de padrões abertos [OASIS] ([https://en.wikipedia.org/wiki/OASIS_\(organization\)](https://en.wikipedia.org/wiki/OASIS_(organization))) é muito parecido com o espírito dos Microservices.

Sam Newman, em seu livro [Building Microservices](#) (NEWMAN, 2015), reconhece que SOA trouxe boas ideias, mas que houve uma falta de consenso em como fazer SOA bem e dificuldade em ter uma narrativa alternativa à dos vendedores. SOA passou a ser visto como uma coleção de ferramentas e não como uma abordagem arquitetural. Ainda fala que uma Arquitetura de Microservices está para SOA assim como XP e Scrum estão para Agile: uma abordagem específica que veio de projetos reais.

Microservices são, então, uma abordagem para SOA.



Microservices e a Cloud

Os 12 fatores do Heroku

Um dos fundadores da plataforma de Cloud Heroku, Adam Wiggins, escreveu em 2011 um texto em que descreve soluções comuns para aplicações Web do tipo SaaS (Software as a Service) que rodam em plataformas de Cloud Computing. Essas soluções foram coletadas a partir da experiência em desenvolver, operar e escalar milhares de aplicações. É o que o autor chamou de [Os 12 Fatores](#) (WIGGINS, 2011):

1. *Base de Código*: há só uma base de código para a aplicação, rastreada em um sistema de controle de versão como Git e que pode gerar diferentes deploys (desenvolvimento, testes, produção).
2. *Dependências*: todas as bibliotecas e frameworks usados pela aplicação devem ser declarados explicitamente como dependências. As dependências são isoladas, impedindo que dependências implícitas vazem a partir do ambiente de execução.
3. *Configurações*: tudo que varia entre deploys, como credenciais de BDs e de serviços externos como Amazon S3, deve estar separado do código da aplicação. Essas configurações devem ser armazenadas em variáveis de ambiente, que são uma maneira multiplataforma e não ficarão na base de código.
4. *Backing services*: não há distinção entre um BD local ou serviço externo como New Relic (usado para métricas). Todos são recursos acessíveis pela rede e cujas URL e credenciais estão nas configurações, e não no código.
5. *Build, release, run*: há 3 estágios distintos para transformar código

em um deploy. O estágio de *build* converte um repositório de código em um executável, contendo todas as dependências. O estágio de *release* combina um executável com as configurações de um deploy, gerando um release imutável e com um timestamp. O estágio de *run* executa um release em um ou mais processos. O build é iniciado quando há novo código. O run pode ser iniciado automaticamente, por exemplo, em um reboot do servidor.

6. *Processos*: devem ser *stateless*. Dados de sessão deve estar em um *datastore* que tem expiração, como o Redis. Nunca deve ser assumido que a memória ou o disco estarão disponíveis em um próximo *request*.
7. *Port binding*: a aplicação é auto-contida e expõe a si mesma por meio de uma porta HTTP. Não há a necessidade de um servidor Web ou servidor de aplicação. Uma camada de roteamento repassa um *hostname* público para a porta HTTP da aplicação.
8. *Concorrência*: deve ser possível escalar a aplicação horizontalmente, replicando múltiplas instâncias idênticas que recebem requests de um *load balancer*. Podem existir processos web, que tratam de um request HTTP e processos *worker*, que cuidam de tarefas que demoram mais.
9. *Descartabilidade*: processos são descartáveis e são iniciados e parados a qualquer momento. O tempo de *startup* deve ser minimizado. Para um processo web, todos requests HTTP devem ser finalizados antes de parar. Para um processo worker, o job deve ser retornado à fila. Os processos devem considerar falhas de hardware e lidar com paradas inesperadas.
10. *Paridade dev/prod*: não devem ser acumuladas semanas de trabalho entre deploys em produção. Os desenvolvedores que escrevem código devem estar envolvidos na implantação e monitoramento em produção. As ferramentas de desenvolvimento devem ser semelhantes às de produção.
11. *Logs*: devem ser tratados como eventos que fluem continuamente enquanto a aplicação estiver no ar. Não devem ser armazenados em arquivos. O ambiente de execução cuida de rotear os logs para

ferramentas de análise como Splunk.

12. *Processos de Admin*: tarefas de administração, como scripts que são executados apenas uma vez, devem ser executados em um ambiente idêntico aos processos worker. O código dessas tarefas pontuais deve estar junto ao código da aplicação.

Cloud Native

No workshop [Patterns for Continuous Delivery, High Availability, DevOps & Cloud Native Open Source with NetflixOSS](#) (COCKCROFT, 2013), Adrian Cockcroft, um dos responsáveis pela migração da Netflix para Cloud iniciada em 2009, conta como seu time uniu patterns de sucesso no que começou a ser referida como arquitetura **Cloud Native**. A agilidade nos negócios, produtividade dos desenvolvedores e melhora na Escalabilidade e Disponibilidade são resultados da adoção de Continuous Delivery, DevOps, Open Source, Microservices, dados desnormalizados (NoSQL) e Cloud Computing com data centers globais. Isso permitiu que a Netflix atendesse a um crescimento exponencial no número de usuários. Algumas das ferramentas desenvolvidas na Netflix tiveram seu código aberto, numa plataforma chamada [Netflix OSS](#).

Em 2015, foi criada a [Cloud Native Computing Foundation](#) (CNCF) a partir da Linux Foundation, visando manter projetos open-source de ferramentas Cloud Native de maneira a evitar *vendor lock-in*. Entre os projetos mantidos pela CNCF estão orquestradores de containers como Kubernetes, proxys como o Envoy e ferramentas de monitoramento como o Prometheus. Entre as empresas que participam da CNCF estão Google, Amazon, Microsoft, Alibaba, Baidu, totalizando US\$ 13 trilhões de valor de mercado, em números de 2019.

Segundo a [definição da CNCF](#) (CNCF TOC, 2018):

Tecnologias Cloud Native empoderam organizações a construir e rodar aplicações escaláveis em ambientes dinâmicos e modernos como Clouds públicas, privadas ou híbridas. Containers, Service Meshes, Microservices, infraestrutura imutável e APIs declarativas são exemplos

dessa abordagem. Essas técnicas permitem sistemas baixamente acoplados, que são resilientes, gerenciáveis e observáveis. Combinadas a automação robusta, permitem que os engenheiros façam mudanças de grande impacto frequentemente e de maneira previsível, com o mínimo de esforço.

Microservices chassis

Há preocupações comuns em uma Arquitetura de Microservices:

- Configuração externalizada
- Health checks
- Métricas de aplicação
- Service discovery
- Circuit breakers
- Distributed tracing

Observação: várias dessas necessidades serão abordadas nos próximos capítulos.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a), chama de **Microservices chassis** um framework ou conjunto de bibliotecas que tratam dessas questões transversais.

Pattern: Microservice chassis

Construa serviços usando um framework ou coleção de frameworks que lida com questões transversais como rastreamento de exceções, logging, health checks, configuração externalizada e distributed tracing.

Frameworks Java como [Dropwizard](#) e [Spring Boot](#) são exemplos de Microservices chassis.

O [Spring Cloud](#) é um conjunto de ferramentas que expandem as capacidades do Spring Boot e oferecem implementações para patterns comuns de sistemas distribuídos. Há, por exemplo, o [Spring Cloud](#)

[Netflix](#), que integra vários componentes da Netflix OSS com o ecossistema do Spring. Estudaremos várias das ferramentas do Spring Cloud durante o curso.

No Java EE (ou Jakarta EE), foi criada a especificação MicroProfile, um conjunto de especificações com foco que servem como um Microservices chassis. Entre as implementações estão: [KumuluzEE](#), [Wildfly Swarm/Thorntail](#), baseado no Wildfly da JBoss/Red Hat, [Open Liberty](#), baseado no WebSphere Liberty da IBM, [Payara Micro](#), baseado no Payara, um fork do GlassFish, e [Apache TomEE](#), baseado no Tomcat.

Decidindo por uma Arquitetura de Microservices no Caelum Eats

Por enquanto, no Caelum Eats, temos uma aplicação monolítica.

Há times separados pelos Bounded Contexts identificados: Pagamentos, Distância, Pedidos, Restaurantes, Administrativo. O código está organizado, alinhados com os Negócios, mas a base de código é uma só. Poderíamos fazer algo mais independente se usássemos uma Arquitetura de Plugins e/ou um Module System diferente, mas os módulos Maven requerem todos os times trabalhando no mesmo projeto.

Há maior controle das dependências entre os módulos do que um projeto não modularizado. Mas ainda é possível, com os módulos Maven, usar classes das dependências transitivas: por exemplo, o módulo de Distância, que depende do módulo de Restaurantes, pode usar classes do módulo Administrativo. A fronteira entre módulos Maven não é forte o bastante.

O cenário de Negócios requer uma evolução rápida de algumas partes da aplicação, como o módulo de Pagamentos, que deseja explorar novos meios de pagamento como criptomoedas e QR Code, além de permitir formas de pagamento mais antigas, como dinheiro.

Já em termos tecnológicos, algumas partes da aplicação da requerem

experimentação, como o módulo de Distância tem a necessidade de explorar novas tecnologias seja de geoprocessamento e até de plataformas de programação diferentes da JVM.

Quanto às operações, há partes da aplicação que apresentam uso intenso de CPU, como o módulo de Distância, levando a necessidade de escalar toda a aplicação em diferentes instâncias para atender à necessidade de processamento de um dos módulos. O mesmo módulo de Distância apresenta uso elevado de memória e eventuais estouros de memória, os famigerados `OutOfMemoryError`, param a instância da aplicação como um todo.

O deploy deve ser feito em conjunto, já que o entregável da aplicação é o fat JAR do Spring Boot gerado pelo módulo `eats-application`. Como o módulo de Pagamentos tem uma taxa de mudança mais frequente que o resto da aplicação, o deploy do módulo de Pagamentos é um deploy de toda a aplicação. O time de pagamentos precisa coordenar a atividade com os outros times antes de lançar uma nova versão.

Nesse cenário, poderíamos aproveitar alguns dos prós de uma Arquitetura de Microservices:

- **Deploy independente:** o time de Pagamentos poderia publicar novas versões com a frequência desejada, minimizando a necessidade de sincronizar os trabalhos com outras equipes, deixando essa coordenação para mudanças nos contratos com outras equipes
- **Escalabilidade independente:** se separarmos o módulo de Distância em um serviço, podemos alocar mais recursos de memória e processamento, sem a necessidade de investir em poder computacional para os outros módulos
- **Maior isolamento de falhas:** um estouro de memória em um serviço de Distância ficaria isolado a instâncias desse serviço, sem derrubar outras funcionalidades
- **Experimentação tecnológica:** o time de Distância poderia explorar o uso de novas tecnologias com maior independência

- **Fronteiras fortes entre componentes:** acabaríamos com o uso indevido de dependências transitivas e as bases de código ficariam completamente isoladas; as dependências entre os serviços seriam por meio de suas APIs

Uma vez que decidimos ir em direção a uma Arquitetura de Microservices, temos que ter consciência das dificuldades que enfrentaremos. Teremos que lidar:

- com as consequências de termos um Sistema Distribuído
- com uma maior complexidade no deploy e monitoramento
- com a possível perda de consistência dos dados
- com a ausência de garantias transacionais

Como falhar numa migração: o Big Bang Rewrite

No artigo [Things You Should Never Do, Part I](#) (SPOLSKY, 2000), Joel Spolsky conta o caso da Netscape, que passou 3 anos sem lançar uma nova versão e, nesse tempo, viu sua fatia de mercado cair drasticamente. Spolsky diz que o motivo para o fracasso é o pior erro estratégico para uma companhia que depende de software: *reescrever código do zero*. Segundo o autor, é comum que programadores querem jogar código antigo fora e escrever tudo do zero e o motivo para isso é que é *mais difícil ler código do que escrever*. Reescrever todo o código, para Joel, é jogar conhecimento fora, dar vantagem competitiva para os concorrentes e gastar dinheiro com código que já existe. Uma refatoração cuidadosa e reescrita pontual de trechos de código seria uma abordagem melhor.

O ocaso da Netscape e o nascimento do Mozilla são assunto do documentário [Code Rush](#) (WINTON, 2000).

Robert "Uncle Bob" Martin chama essa ideia de reescrever todo o projeto de *Grand Redesign in the Sky*, no livro [Clean Code](#) (MARTIN, 2009). Uncle Bob descreve o cenário em que um time dos sonhos é escolhido para reescrever um projeto do zero, enquanto outro time continua a

manter o sistema atual. O novo time passa a ter que fazer tudo o que o software antigo faz, mantendo-se atualizado com as mudanças que são continuamente realizadas.

Esse cenário de manter um sistema antigo enquanto um novo é reescrito do zero lembra um dos paradoxos do filósofo pré-socrático Zenão de Eleia: o paradoxo de Aquiles e a tartaruga. Nesse paradoxo, o herói grego Aquiles e uma tartaruga resolvem apostar uma corrida, com uma vantagem inicial para a tartaruga. Mesmo a velocidade de Aquiles sendo maior que a da tartaruga, quando Aquiles chega à posição inicial A do animal, este move-se até uma posição B. Quando Aquiles chega à posição B, a tartaruga já teria avançado para uma posição C e assim sucessivamente, *ad infinitum*.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a), chama essa ideia de desenvolver uma nova versão do zero de *Big Bang Rewrite* e diz que é algo extremamente arriscado e que provavelmente resultará em fracasso. Duas aplicações teria que ser mantidas, funcionalidades teriam que ser duplicadas e existiria o risco de parte das funcionalidades reescritas não serem necessárias para o Negócio em um futuro próximo. Para Richardson, o sistema legado seria um alvo em constante movimento.

Estrangulando o monólito

Como fazer a migração para uma Arquitetura de Microservices, já que um Big Bang Rewrite não é uma boa ideia?

Uma ideia eficaz é refatorar a aplicação monolítica incrementalmente, removendo funcionalidades e criando novos serviços ao redor do monólito. Com o decorrer do tempo, o Monólito vai encolhendo até, eventualmente, ser reduzido a pó.

Martin Fowler, no artigo [Strangler Fig Application](#) (FOWLER, 2004), faz uma metáfora dessa redução progressiva do Monólito com um tipo de figueira que cresce em volta de uma árvore hospedeira, eventualmente

matando a árvore original e tornando-se uma coluna com o núcleo oco.



Pattern: STRANGLER APPLICATION

Modernize uma aplicação desenvolvendo incrementalmente uma nova aplicação ao redor do legado.

No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson dá algumas dicas de como lidar com uma Strangler Application:

- **Demonstre valor frequentemente e desde cedo:** a ideia é que sejam usados ciclos curtos e frequentes de entrega. Dessa maneira, a Strangler Application reduz o risco e oferece alto retorno sobre o investimento (ROI), ainda que coexistindo com o sistema original. Devem ser priorizadas novas funcionalidades ou migrações de alto impacto e valor de negócio. Assim, os financiadores da migração oferecerão o suporte necessário, justificando o investimento técnico na nova arquitetura.
- **Minimize as mudanças no Monólito:** será necessário alterar o monólito durante a migração para serviços. Mas essas mudanças

tem que ser gerenciadas, evitando que sejam de alto custo, arriscada e consumam muito tempo.

- **Simplifique a infraestrutura:** pode haver o desejo de explorar novas e sofisticadas plataformas de operações como Kubernetes ou AWS Lambda. A única coisa mandatória é um deployment pipeline com testes automatizados. Com um número reduzido de serviços, não há a necessidade de deploy ou monitoramento sofisticados. Adie o investimento até que haja experiência com uma Arquitetura de Microservices.

Richardson ainda cita que a migração para Microservices pode durar alguns anos, como foi o caso da Amazon. E pode ser que a organização dê prioridade a funcionalidades que geram receita, em detrimento daquebra do Monólito, principalmente, se não oferecer obstáculos.

Fowler, em seu artigo [Strangler Fig Application](#) (FOWLER, 2004), argumenta que novas aplicações deveriam ser arquitetadas de maneira a facilitar uma possível estrangulação no futuro. Afinal, o código que escrevemos hoje será o legado de amanhã.

Começar a migração pelo BD ou pela aplicação?

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), recomenda uma progressão que começa pelo BD:

1. Encontrar as linhas de costura (em inglês, *seams*) da aplicação, agrupando o código do Monólito em pacotes.
2. Identificar as costuras no BD, tentando já quebrar dependências.
3. Dividir o schema do BD, mantendo o código no monólito. Nesse momento, joins, foreign keys e integridade transacional já seriam perdidas. Para Newman, seria uma boa maneira de explorar a decomposição e ajustar detalhes.
4. Só então o código seria dividido em serviços, poucos de cada vez, progressivamente

Já no livro [Microservices AntiPatterns and Pitfalls](#) (RICHARDS, 2016),

Mark Richards discorda diametralmente. Richards argumenta que são muitos comuns ajustes na granularidade dos serviços no começo da migração. Podemos ter quebrado demais os serviços, ou de menos. E reagrupar os dados no BD é muito mais difícil, custoso e propenso a erros que reagrupar o código da aplicação. Para o autor, uma migração para Microservices deveria começar com o código. Assim que haja uma garantia que a granularidade do serviço está correta, os dados podem ser migrados. É importante ressaltar que manter o BD monolítico é uma solução paliativa. Richards deixa claro o risco dessa abordagem: acoplamento dos serviços pelo BD. Discutiremos essa ideia mais adiante no curso.

Já em seu novo livro [Monolith to Microservices](#) (NEWMAN, 2019), Sam Newman explora diferentes abordagens para extração de serviços: pelo BD primeiro, pelo código primeiro e BD e código juntos.

Newman diz que começaria pelo BD nos casos em que a performance ou consistência dos dados são preocupações especiais, de maneira a antecipar problemas. Uma desvantagem de começar pelo BD seria o fato de não trazer benefícios claros no curto prazo.

Começar a extração de serviços pelo código, para Newman, traz a vantagem de facilitar o entendimento de qual é o código necessário para o serviço a ser extraído. Além disso, desde cedo há um artefato de código cujo deploy é independente. Uma grande desvantagem é que, na experiência de Newman, é muito comum parar a migração e manter um Shared Database. Outra preocupação é que desafios de performance e consistência são deixados para o futuro, o que pode trazer surpresas nefastas.

Fazer a extração simultânea do BD e do código deve ser evitado, na opinião de Newman. É um passo muito grande, de alto risco e impacto.

No fim das contas, Newman conclui com "Depende". Cada situação é diferente e prós e contras tem que ser discutidos com o contexto específico em mente.

O que extrair do monólito?

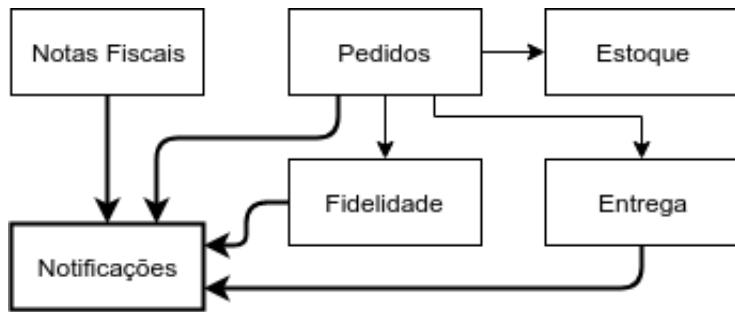
Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a), diz que os módulos devem ser classificados de acordo com critérios que ajudem a visualizar antecipadamente os benefícios de extraí-los do Monólito. Entre os critérios:

- O desenvolvimento será acelerado pela extração: se uma parte específica da aplicação sofrerá uma grande evolução em um futuro próximo, convertê-la para um serviço pode acelerar as entregas.
- Um problema de performance, escalabilidade ou confiabilidade será resolvido: se uma fatia da aplicação apresenta problemas nesse requisitos não-funcionais, afetando o Monólito como um todo, pode ser uma boa ideia extraí-la para um serviço.
- Permitirá a extração de outros serviços: às vezes, as dependências entre os módulos fazem com que fique mais fácil extraír um serviço depois de algum outro ter sido extraído

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), traz reflexões semelhantes. Seriam bons candidatos a serem extraídos, partes do Monólito:

- que mudam com uma frequência muito maior
- que tem times separados, às vezes geograficamente
- que possuem necessidades de segurança e proteção da informação mais restritas
- que teriam vantagens no uso de uma tecnologia diferente

No livro [Monolith to Microservices](#) (NEWMAN, 2019), Sam Newman argumenta que, na priorização de novos serviços a serem extraídos do Monólito, devem ser levadas em conta as dependências entre os grupos de funcionalidade (ou componentes). Partes do código com muitas dependências aferentes (que chegam) dariam muito trabalho para serem extraídas, já que iriam requerer mudanças no código de todas as outras partes que a usam. Já partes do código com poucas, ou nenhuma, dependência aferente seriam bem mais fácil de serem extraídas.

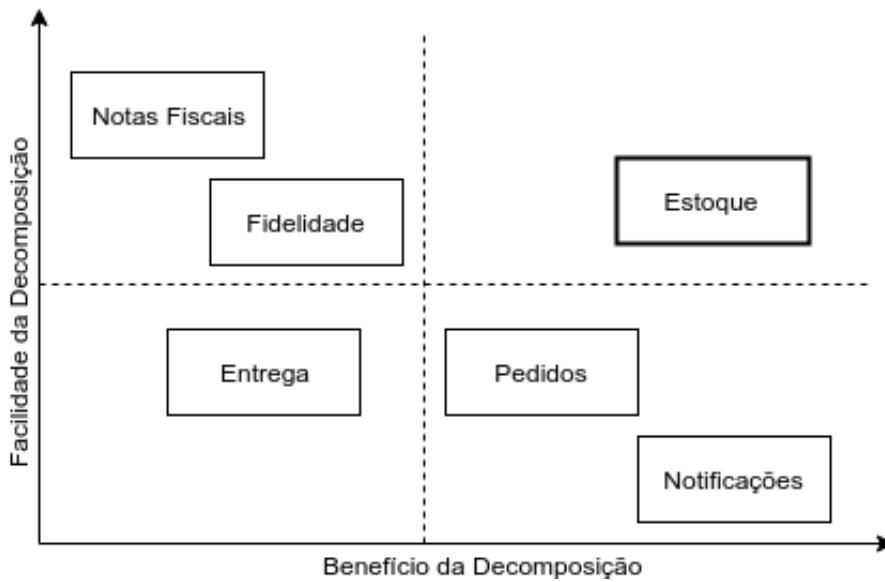


Na imagem anterior, Notificação é um componente difícil de ser extraído, porque tem muitas dependências que chegam. Já um componente como Notas Fiscais seria mais fácil de extrair, porque ninguém depende dele.

Porém, Newman discute que uma visão parecida com a da imagem anterior é uma visão lógica do domínio. Não necessariamente essa visão estará refletida no código. Por exemplo, o BD é um ponto de acoplamento muitas vezes negligenciado.

Ainda no livro [Monolith to Microservices](#) (NEWMAN, 2019), Newman discute que a facilidade de decomposição deve ser ponderada com o benefício trazido para os Negócios. Se o módulo de Notas Fiscais muda muito pouco e não melhora o *time to market*, talvez não seja um bom uso do tempo do time de desenvolvimento e, consequentemente, dos recursos financeiros da organização.

Newman, então, argumenta em favor de uma visão multidimensional, considerando tanto a facilidade como o benefício trazido pelas decomposições. Os componentes podem ser posicionados em um quadrante considerando essas duas dimensões. Ainda que subjetivas, as posições relativas ajudam na priorização. Os componentes que estiverem no canto superior direito do quadrante são bons candidatos à decomposição.



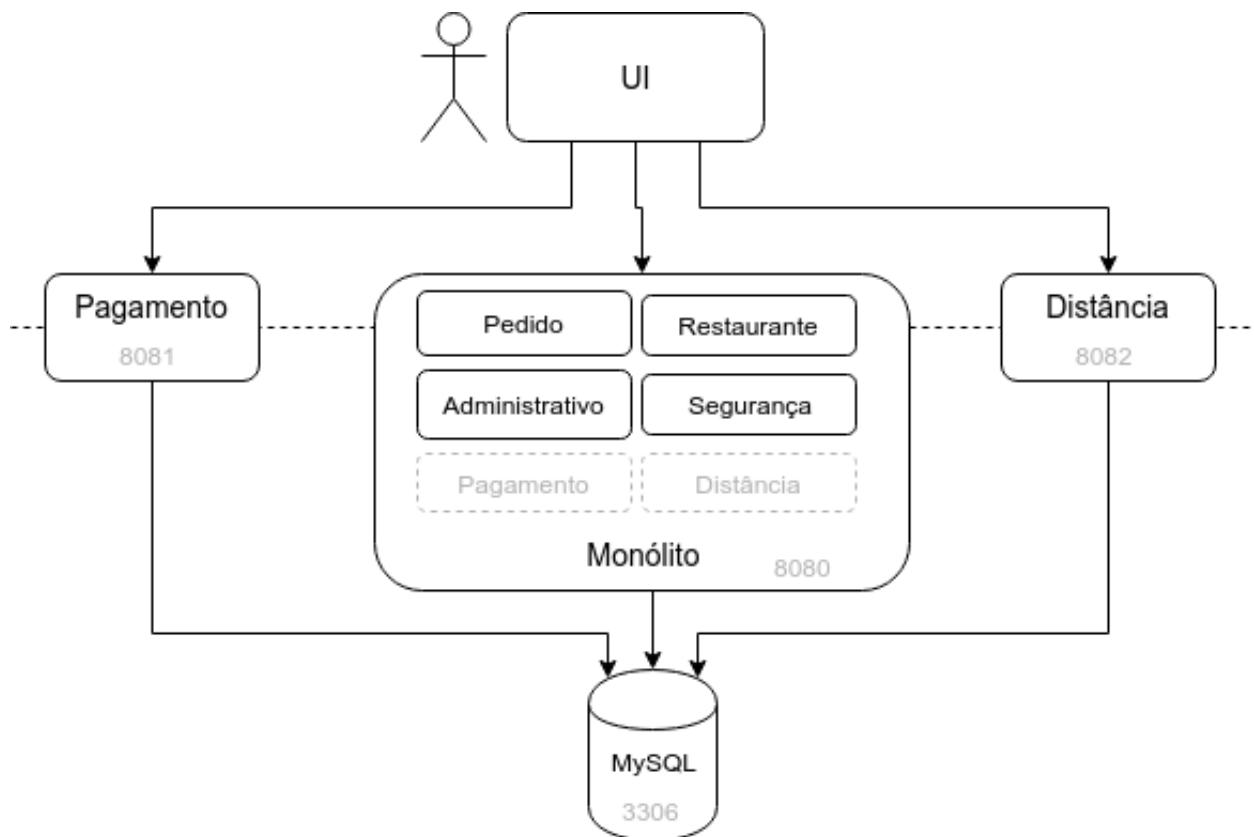
Extraindo serviços do Monólito do Caelum Eats

Como vimos anteriormente, o módulo de Pagamentos do Monólito do Caelum Eats precisa evoluir mais rápido que os demais e, portanto, seria interessante que o deploy fosse independente.

Já para o módulo de Distância, há a necessidade de experimentação tecnológica. Em termos de operações, há maior uso de recursos computacionais e, então, seria interessante escalar esse módulo de maneira independente. Além disso, o módulo de Distância falha mais frequentemente que os outros módulos, tirando toda a aplicação do ar. Seria interessante que as falhas fossem isoladas.

Podemos pensar numa estratégia em que os módulos de Pagamentos e Distância seriam extraídos por seus respectivos times, em paralelo. Esses times trabalhariam em bases de código separadas e, no melhor estilo DevOps, cuidariam de sua infraestrutura.

Ainda há um empecilho: o Banco de Dados. Por enquanto, deixaremos um só BD. Mais adiante, discutiremos se essa será a decisão final.

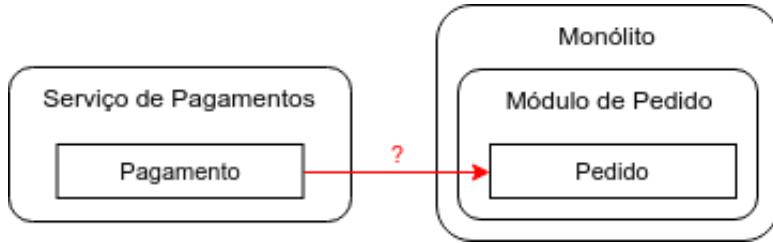


Quebrando o Domínio

No capítulo que discute sobre como refatorar um monólito em direção a Microservices, do livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson diz que é necessário extrair o Modelo de Domínio específico para um novo serviço do Modelo de Domínio já existente no Monólito. E um dos principais desafios é eliminar referências a objetos de outros domínios, que vão além das fronteiras de um serviço.

No Caelum Eats, por exemplo, o novo serviço de Pagamentos teria um objeto `Pagamento` que está relacionado a um `Pedido`, que continuará no módulo de Pedido do Monólito.

```
class Pagamento {
    // outros atributos...
    @ManyToOne(optional=false)
    private Pedido pedido;
}
```



Observação: além de depender de Pedido, um Pagamento também depende de FormaDePagamento que está presente no módulo Administrativo do Monólito.

Essa dependência a objetos além dos limites do serviço é problemática porque haveria um acoplamento indesejado entre os Modelos de Domínio. Não há a necessidade do serviço de Pagamentos conhecer todos os detalhes de um pedido. Além disso, como tanto Pagamento como Pedido são entidades, haveria uma dependência pelo BD, que queremos evitar, pensando nos próximos passos da extração do serviço de Pagamentos.

Uma ideia seria usar bibliotecas com o Modelo de Domínio de outro serviço, o que é comumente chamado de *Shared Libs*. Por exemplo, no Caelum Eats, poderíamos ter um JAR apenas com a classe Pedido e classes associadas, como ItemDoPedido, Entrega e Avaliacao.

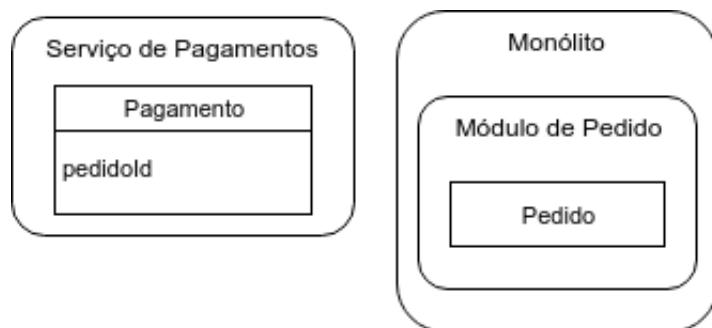
Porém, usar *Shared Libs* para classes de Domínio traz um acoplamento entre os serviços ao redor da API. A cada mudança do Modelo de Domínio de um serviço, todos os seus clientes devem receber uma nova versão do JAR e deve ser feito um novo deploy em cada um deles.

Para Richardson, porém, há lugar para *Shared Libs*: para funcionalidades que são improváveis de serem modificadas, como uma classe Moeda. Bibliotecas técnicas, como frameworks, drivers e ferramentas para tarefas mais específicas, não são um problema.

Richardson argumenta que uma boa solução é pensar em termos de Agregados do DDD, que referenciam outros Agregados por meio da identidade de sua raiz.

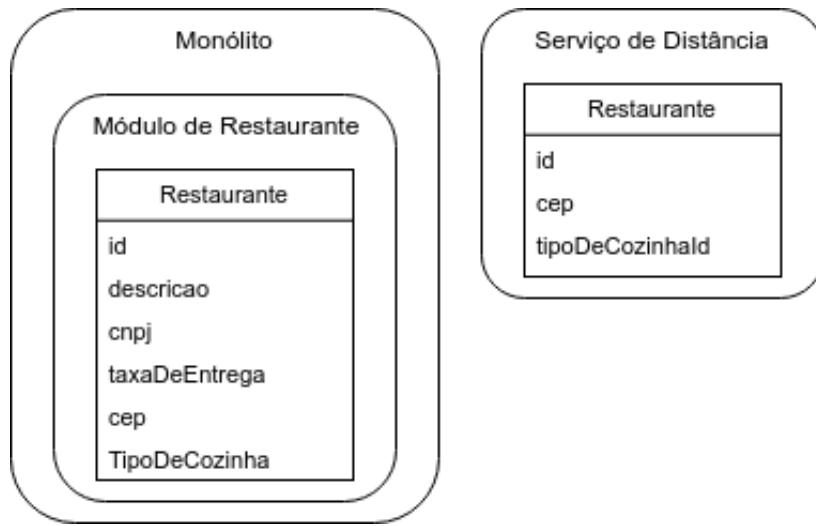
Traduzindo isso para o cenário do Caelum Eats, teríamos uma referência, em `Pagamento`, apenas ao `id` do `Pedido`:

```
class Pagamento {  
    // outros atributos...  
  
    @ManyToOne(optional=false)  
    private Pedido pedido;  
  
    @Column(nullable=false)  
    private Long pedidoId;  
}
```



Richardson discute que uma pequena mudança como a feita anteriormente pode trazer um grande impacto para outras classes, que esperavam uma referência a um objeto. Além disso, há desafios maiores como extrair lógica de negócio de classes que tem mais de uma responsabilidade.

Um outro caso interessante são serviços que tem visões diferentes de uma mesma entidade. Por exemplo, a classe `Restaurante` é utilizada tanto pelo módulo de Restaurante do Monólito como pelo serviço de Distância. Mas, enquanto o módulo Restaurante tem a necessidade de manter o CNPJ, taxa de entrega e outros detalhes, o serviço de Distância só está interessado no CEP e Tipo de Cozinha.



Criando um Microservice de Pagamentos

Vamos iniciar criando um projeto para o serviço de pagamentos.

Pelo navegador, abra <https://start.spring.io/>. Em *Project*, mantenha *Maven Project*. Em *Language*, mantenha Java. Em *Spring Boot*, mantenha a versão padrão. No trecho de *Project Metadata*, defina:

- br.com.caelum em *Group*
- eats-pagamento-service em *Artifact*

Clique em *More options*. Mantenha o valor em *Name*. Apague a *Description*, deixando-a em branco. Em *Package Name*, mude para `br.com.caelum.eats.pagamento`.

Mantenha o *Packaging* como *Jar*. Mantenha a *Java Version* em 8.

Em *Dependencies*, adicione:

- Web
- DevTools
- Lombok
- JPA
- MySQL

Clique em *Generate Project*.

Extraia o eats-pagamento-service.zip.

No arquivo `src/main/resources/application.properties`, modifique a porta para 8081 e, por enquanto, aponte para o mesmo BD do monólito. Defina também algumas outras configurações do JPA e de serialização de JSON.

```
##### fj33-eats-pagamento-
service/src/main/resources/application.properties

server.port = 8081

#DATASOURCE CONFIGS
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true
spring.datasource.username=<SEU USUARIO>
spring.datasource.password=<SU A SENHA>

#JPA CONFIGS
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true

spring.jackson.serialization.fail-on-empty-beans=false
```

Observação: <SEU USUARIO> e <SU A SENHA> devem ser trocados pelos valores do MySQL do monólito.

Extraindo código de pagamentos do monólito

Copie do módulo eats-pagamento do monólito, as seguintes classes, colando-as no pacote `br.com.caelum.eats.pagamento` do eats-pagamento-service:

- Pagamento
- PagamentoController
- PagamentoDto

- PagamentoRepository
- ResourceNotFoundException

Dica: você pode copiar e colar pelo próprio Eclipse.

Há alguns erros de compilação. Os corrigiremos nos próximos passos.

Na classe `Pagamento`, há erros de compilação nas referências às classes `Pedido` e `FormaDePagamento` que são, respectivamente, dos módulos `eats-pedido` e `eats-administrativo` do monólito.

Será que devemos colocar dependências Maven a esses módulos? Não parece uma boa, não é mesmo?

Vamos, então, trocar as referências a essas classes pelos respectivos ids, de maneira a referenciar as raízes dos agregados `Pedido` e `FormaDePagamento`:

```
##### fj33-eats-pagamento-
service/src/main/java/br/com/caelum/eats/pagamento/Pagamento.java
```

```
// anotações ...
class Pagamento {

    // código omitido...

    @ManyToOne(optional=false)
    private Pedido pedido;

    @Column(nullable=false)
    private Long pedidoId;

    @ManyToOne(optional=false)
    private FormaDePagamento formaDePagamento;

    @Column(nullable=false)
    private Long formaDePagamentoId;
```

```
}
```

Ajuste os imports, removendo os desnecessários e adicionando novos:

```
import br.com.caelum.eats.admin.FormaDePagamento;
import br.com.caelum.eats.pedido.Pedido;
import javax.persistence.ManyToOne;

import javax.persistence.Column; // adicionado ...

// outros imports ...
```

A mesma mudança deve ser feita para a classe PagamentoDto, referenciando apenas os ids das classes PedidoDto e FormaDePagamento:

```
##### fj33-eats-pagamento-
service/src/main/java(br/com/caelum/eats/pagamento/PagamentoDto.jav
a
```

```
// anotações ...
class PagamentoDto {

    // outros atributos...

    private FormaDePagamentoDto formaDePagamento;
    private Long formaDePagamentoId;

    private PedidoDto pedido;
    private Long pedidoId;

    public PagamentoDto(Pagamento p) {
        this(p.getId(), p.getValor(), p.getNome(), p.getNumero(),
            new FormaDePagamentoDto(p.getFormaDePagamento()),
            p.getFormaDePagamentoId(),
            new PedidoDto(p.getPedido()), p.getPedidoId());
```

```
}
```

```
}
```

Remova os imports desnecessários:

```
import br.com.caelum.eats.administrative.FormaDeP...
import br.com.caelum.eats.pedido.PedidoDto;
```

Ao confirmar um pagamento, a classe `PagamentoController` atualiza o status do pedido.

Por enquanto, vamos simplificar a confirmação de pagamento, que ficará semelhante a criação e cancelamento: apenas o status do pagamento será atualizado.

Depois voltaremos com a atualização do pedido.

```
##### fj33-eats-pagamento-
service/src/main/java(br/com/caelum/eats/pagamento/PagamentoController.java)
```

```
// anotações ...
class PagamentoController {

    private PagamentoRepository pagamentoRepo;
    private PedidoService pedidos;

    // demais métodos...

    @PutMapping("/{id}")
    public PagamentoDto confirma(@PathVariable Long id) {
        Pagamento pagamento = pagamentoRepo.findById(id).orElseTh
        pagamento.setStatus(Pagamento.Status.CONFIRMADO);
        pagamentoRepo.save(pagamento);
        Long pedidoId = pagamento.getPedido().getId();
```

```
-P-edido-pedido==pedidos.-perI-dC-omI-tens+pedidoI-d  
-pedido.-setS-tatus+P-edido.-S-tatus.-P-A-G-0-> ;  
-pedidos.-atualizaS-tatus+P-edido.-S-tatus.-P-A-G-0,-  
    return new PagamentoDto(pagamento);  
}  
}
```

Ah! Limpe os imports:

```
import br.com.caetum.eats.pedido.P-edido;  
import br.com.caetum.eats.pedido.P-edidos-service;
```

Fazendo a UI chamar novo serviço de pagamentos

Adicione uma propriedade `pagamentoUrl`, que aponta para o endereço do novo serviço de pagamentos, no arquivo `environment.ts`:

```
##### fj33-eats-ui/src/environments/environment.ts
```

```
export const environment = {  
  production: false,  
  baseUrl: '//localhost:8080'  
 , pagamentoUrl: '//localhost:8081' //adicionado  
};
```

Use a nova propriedade `pagamentoUrl` na classe `PagamentoService`:

```
##### fj33-eats-ui/src/app/services/pagamento.service.ts
```

```
export class PagamentoService {  
  
  private API == environment.baseUrl + '/pagamentos'  
  private API = environment.pagamentoUrl + '/pagamentos';
```

```
// restante do código ...  
}
```

No eats-pagamento-service, trocamos referências às entidades Pedido e FormaDePagamento pelos respectivos ids. Essa mudança afeta o código do front-end. Faça o ajuste dos ids na classe PagamentoService:

```
##### fj33-eats-ui/src/app/services/pagamento.service.ts
```

```
export class PagamentoService {  
  
    // código omitido ...  
  
    cria(pagamento): Observable<any> {  
        this.ajustaIds(pagamento); // adicionado  
        return this.http.post(`.${this.API}`, pagamento);  
    }  
  
    confirma(pagamento): Observable<any> {  
        this.ajustaIds(pagamento); // adicionado  
        return this.http.put(`.${this.API}/${pagamento.id}`, null);  
    }  
  
    cancela(pagamento): Observable<any> {  
        this.ajustaIds(pagamento); // adicionado  
        return this.http.delete(`.${this.API}/${pagamento.id}`);  
    }  
  
    // adicionado  
    private ajustaIds(pagamento) {  
        pagamento.formaDePagamentoId = pagamento.formaDePagamentoId || pagamento.formaDePagamentoId;  
        pagamento.pedidoId = pagamento.pedidoId || pagamento.pedidoId;  
    }  
}
```

O código do método privado `ajustaIds` define as propriedades

`formaDePagamentoId` e `pedidoId`, caso ainda não estejam presentes.

No componente `PagamentoPedidoComponent`, precisamos fazer ajustes para usar o atributo `pedidoId` do pagamento:

```
##### fj33-eats-ui/src/app/pedido/pagamento/pagamento-
pedido.component.ts
```

```
export class PagamentoPedidoComponent implements OnInit {

    // código omitido ...

    confirmaPagamento() {
        this.pagamentoService.confirma(this.pagamento)
            .subscribe(pagamento => this.router.navigate([
                'pagamento',
                pagamento.id
            ]))
            .subscribe(pagamento => this.router.navigateByUrl(`pedidos/${pagamento.id}`))
    }

    // restante do código ...
}
```

Com o monólito e o serviço de pagamentos sendo executados, podemos testar o pagamento de um novo pedido.

Deve ocorrer um *Erro no Servidor*. O Console do navegador, acessível com F12, deve ter um erro parecido com:

```
Access to XMLHttpRequest at 'http://localhost:8081/pagamentos' from
origin 'http://localhost:4200' has been blocked by CORS policy:
Response to preflight request doesn't pass access control check: No
'Access-Control-Allow-Origin' header is present on the requested
resource.
```

Isso acontece porque precisamos habilitar o CORS no serviço de pagamentos, que está sendo invocado diretamente pelo navegador.

Habilitando CORS no serviço de pagamentos

Para habilitar o Cross-Origin Resource Sharing (CORS) no serviço de pagamento, é necessário definir uma classe `CorsConfig` no pacote `br.com.caelum.eats.pagamento`, semelhante à do módulo `eats-application` do monólito:

```
@Configuration
class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**").allowedMethods("*").allowCred
    }

}
```

Faça um novo pedido, crie e confirme um pagamento. Deve funcionar!

Note apenas um detalhe: o status do pedido, exibido na tela após a confirmação do pagamento, **está REALIZADO e não PAGO**. Isso ocorre porque removemos a chamada à classe `PedidoService`, que ainda está no módulo `eats-pedido` do monólito. Corrigiremos esse detalhe mais adiante no curso.

Apagando código de pagamentos do monólito

Remova a dependência a `eats-pagamento` do `pom.xml` do módulo `eats-application` do monólito:

```
#####
fj33-eats-monolito-modular/eats/eats-application/pom.xml

<dependency>
    <groupId>br.com/caelum</groupId>
    <artifactId>eats-pagamento</artifactId>
```

```
—<version>${project.version}</version>
-</dependency>
```

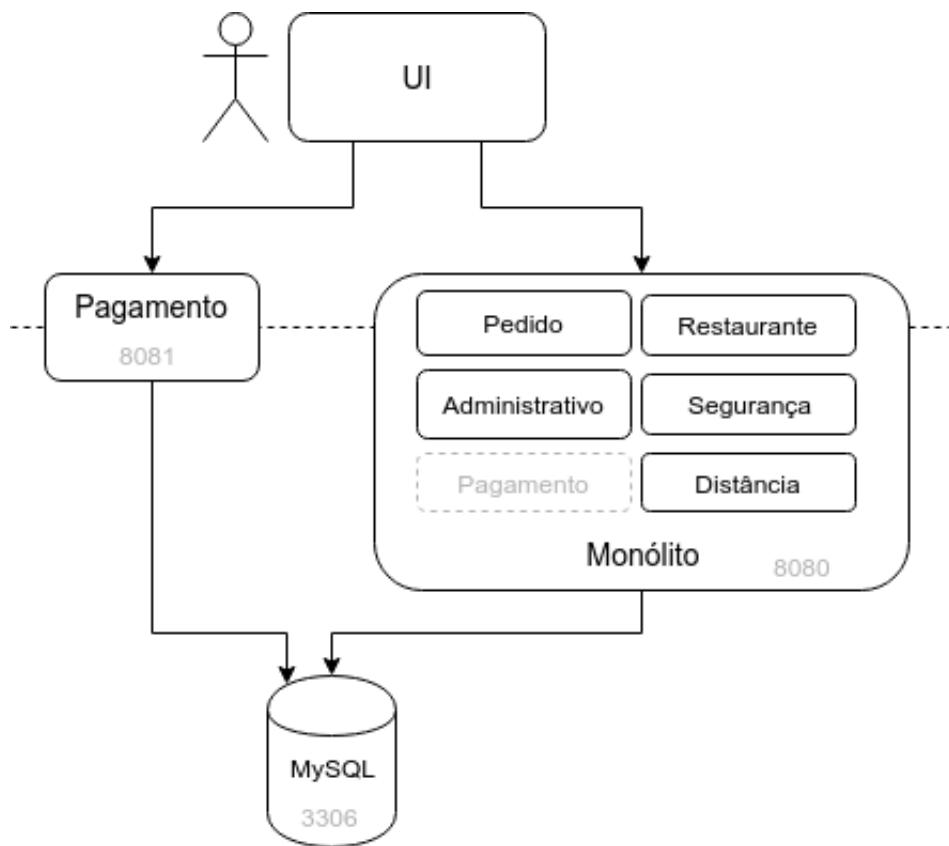
No projeto pai dos módulos, o projeto eats, remova o módulo eats-pagamento do pom.xml:

```
##### fj33-eats-monolito-modular/eats/pom.xml
```

```
<modules>
  <module>eats-administrativo</module>
  <module>eats-pagamento</module>
  <module>eats-restaurante</module>
  <module>eats-pedido</module>
  <module>eats-distancia</module>
  <module>eats-seguranca</module>
  <module>eats-application</module>
</modules>
```

Apague o módulo eats-pagamento do monólito. Pelo Eclipse, tecle *Delete* em cima do módulo, selecione a opção *Delete project contents on disk (cannot be undone)* e clique em *OK*. O diretório com o código do módulo eats-pagamento será removido do disco.

Extraímos nosso primeiro serviço do monólito. A evolução do código de pagamento, incluindo a exploração de novos meios de pagamento, pode ser feita em uma base de código separada do monólito. Porém, ainda mantivemos o mesmo BD, que será migrado em capítulos posteriores.



Exercício: Executando o novo serviço de pagamentos

1. Abra um Terminal e, no Desktop, clone o projeto com o código do serviço de pagamentos:

```
cd ~/Desktop
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-pa
```

Vamos criar um workspace do Eclipse separado para os Microservices, mantendo aberto o workspace com o monólito. Para isso, clique no ícone do Eclipse da área de trabalho. Em *Workspace*, defina `/home/<usuario-do-curso>/workspace-microservices`, onde `<usuario-do-curso>` é o login do curso.

No Eclipse, importe o projeto `fj33-eats-pagamento-service`, usando o menu *File > Import > Existing Maven Projects*.

Então, execute a classe `EatsPagamentoServiceApplication`.

Teste a criação de um pagamento com o cURL:

```
curl -X POST  
-i  
-H 'Content-Type: application/json'  
-d '{ "valor": 51.8, "nome": "JOÃO DA SILVA", "numero": "1111  
http://localhost:8081/pagamentos
```

Para que você não precise digitar muito, o comando acima está disponível em: <https://gitlab.com/snippets/1859389>

No comando acima, usamos as seguintes opções do cURL:

- `-x` define o método HTTP a ser utilizado
- `-i` inclui informações detalhadas da resposta
- `-H` define um cabeçalho HTTP
- `-d` define uma representação do recurso a ser enviado ao serviço

A resposta deve ser algo parecido com:

```
HTTP/1.1 200  
Content-Type: application/json; charset=UTF-8  
Transfer-Encoding: chunked  
Date: Tue, 21 May 2019 20:27:10 GMT
```

```
{ "id":7, "valor":51.8, "nome":"JOÃO DA SILVA",  
"numero":"1111 2222 3333 4444", "expiracao":"2022-07", "codi  
"status":"CRIADO", "formaDePagamentoId":2, "pedidoId":1}
```

Observação: há outros clientes para testar APIs RESTful, como o Postman. Fique à vontade para usá-los. Peça ajuda ao instrutor para instalá-los.

Usando o id retornado no passo anterior, teste a confirmação do

pagamento pelo cURL, com o seguinte comando:

```
curl -X PUT -i http://localhost:8081/pagamentos/7
```

Você deve obter uma resposta semelhante a:

```
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Tue, 21 May 2019 20:31:08 GMT
```

```
{ "id":7, "valor":51.8, "nome":"JOÃO DA SILVA",
  "numero":"1111 2222 3333 4444", "expiracao":"2022-07", "codi
  "status":"CONFIRMADO","formaDePagamentoId":2,"pedidoId":1}
```

Observe que o status foi modificado para **CONFIRMADO**.

2. Pare a execução do monólito, caso esteja no ar.

Vá até o diretório do monólito. Obtenha o código da branch `cap3-extrai-pagamento-service`, que já tem o serviço de pagamentos extraído.

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap3-extrai-pagamento-service
```

Execute novamente a classe `EatsApplication` do módulo `eats-application` do monólito.

3. Pare a execução da UI.

No diretório da UI, mude a branch para `cap3-extrai-pagamento-service`, que contém as alterações necessárias para invocar o novo serviço de pagamentos.

```
cd ~/Desktop/fj33-eats-ui  
git checkout -f cap3-extrai-pagamento-service
```

Execute novamente a UI com o comando `ng serve`.

Acesse `http://localhost:4200` e realize um pedido. Tente criar um pagamento.

Observe que, após a confirmação do pagamento, o status do pedido **está REALIZADO e não PAGO**. Isso ocorre porque removemos a chamada à classe `PedidoService`, cujo código ainda está no monólito. Corrigiremos esse detalhe mais adiante no curso.

Criando um Microservice de distância

Abra `https://start.spring.io/` no navegador. Em *Project*, mantenha *Maven Project*. Em *Language*, mantenha Java. Em *Spring Boot*, mantenha a versão padrão. No trecho de *Project Metadata*, defina:

- `br.com.caelum` em *Group*
- `eats-distancia-service` em *Artifact*

Clique em *More options*. Mantenha o valor em *Name*. Apague a *Description*, deixando-a em branco. Em *Package Name*, mude para `br.com.caelum.eats.distancia`.

Mantenha o *Packaging* como `Jar`. Mantenha a *Java Version* em 8.

Em *Dependencies*, adicione:

- Web
- DevTools
- Lombok
- JPA
- MySQL

Clique em *Generate Project*.

Descompacte o `eats-distancia-service.zip` para seu Desktop.

Edite o arquivo `src/main/resources/application.properties`, modificando a porta para 8082, apontando para o BD do monólito, além de definir configurações do JPA e de serialização de JSON:

```
##### fj33-eats-distancia-
service/src/main/resources/application.properties

server.port = 8082

#DATASOURCE CONFIGS
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true
spring.datasource.username=<SEU USUARIO>
spring.datasource.password=<SUAS SENHAS>

#JPA CONFIGS
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true

spring.jackson.serialization.fail-on-empty-beans=false
```

Troque `<SEU USUARIO>` e `<SUAS SENHAS>` pelos valores do BD.

Extraindo código de distância do monólito

Copie para o pacote `br.com.caelum.eats.distancia` do serviço `eats-distancia-service`, as seguintes classes do módulo `eats-distancia` do monólito:

- `DistanciaService`
- `RestauranteComDistanciaDto`
- `RestaurantesMaisProximosController`
- `ResourceNotFoundException`

Além disso, já antecipando problemas com CORS no front-end, copie do módulo eats-application do monólito, para o pacote `br.com.caelum.eats.distancia` do serviço de distância, a classe:

- `CorsConfig`

Há alguns erros de compilação na classe `DistanciaService`, que corrigiremos nos passos seguintes.

O motivo de um dos erros de compilação é uma referência à classe `Restaurante` do módulo eats-restaurante do monólito.

Copie essa classe para o pacote `br.com.caelum.eats.distancia` do serviço de distância. Ajuste o pacote, caso seja necessário.

Remova, na classe `Restaurante` copiada, a referência à entidade `TipoDeCozinha`, trocando-a pelo id.

Remova por completo a referência à classe `User`.

```
##### fj33-eats-distancia-
service/src/main/java;br/com/caelum/eats/distancia/Restaurante.java
```

```
// anotações
public class Restaurante {

    // código omitido ...

    @ManyToOne(optional=false)
    private TipoDeCozinha tipoDeCozinha;

    @Column(nullable=false)
    private Long tipoDeCozinhaId;

    @OneToOne
    private User user;

}
```

Ajuste os imports:

```
import javax.persistence.ManyToOne;
import javax.persistence.OneToOne;

import br.com.caelum.eats.administrative.TipoDeCozinha;
import br.com.caelum.eats.seguranca.Usuario;

import javax.persistence.Column; // adicionado ...
```

Na classe `DistanciaService` de `eats-distancia-service`, remova os imports que referenciam as classes `Restaurante` e `TipoDeCozinha`:

```
##### fj33-eats-distancia-
service/src/main/java(br/com/caelum/eats/distancia/DistanciaService.java
```

```
import br.com.caelum.eats.administrative.TipoDeCozinha;
import br.com.caelum.eats.restaurante.Restaurante;
```

Como a classe `Restaurante` foi copiada para o mesmo pacote de `DistanciaService`, não há a necessidade de importá-la.

Mas e para `TipoDeCozinha`? Utilizaremos apenas o id. Por isso, modifique o método `restaurantesDoTipoDeCozinhaMaisProximosAoCep` de `DistanciaService`:

```
##### fj33-eats-distancia-
service/src/main/java(br/com/caelum/eats/distancia/DistanciaService.java
```

```
public List<RestauranteComDistanciaDto> restaurantesDoTipoDeCozinha(TipoDeCozinha tipo) {
    tipo.setId(new TipoDeCozinha());
    tipo.setI_d(tipo.getId());
```

```
List<Restaurante> aprovadosDoTipoDeCozinha = restaurantes.findAll();
List<Restaurante> aprovadosDoTipoDeCozinha = restaurantes.findAll();
return calculaDistanciaParaOsRestaurantes(aprovadosDoTipoDeCozinha);
}
```

Ainda resta um erro de compilação na classe `DistanciaService`: o uso da classe `RestauranteService`. Poderíamos fazer uma chamada remota, por meio de um cliente REST, ao monólito para obter os dados necessários. Porém, para esse serviço, acessaremos diretamente o BD.

Por isso, crie uma interface `RestauranteRepository` no pacote `br.com.caelum.eats.distancia` de `eats-distancia-service`, que estende `JpaRepository` do Spring Data Jpa e possui os métodos usados por `DistanciaService`:

```
#####
fj33-eats-distancia-
service/src/main/java;br/com/caelum/eats/distancia/RestauranteReposito-
ry.java
```

```
package br.com.caelum.eats.distancia;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;

interface RestauranteRepository extends JpaRepository<Restaurante, Long> {
    Page<Restaurante> findAllByAprovadoAndTipoDeCozinhaId(boolean aprovado, Long id);
    Page<Restaurante> findAllByAprovado(boolean aprovado, Pageable pageable);
}
```

Em `DistanciaService`, use `RestauranteRepository` ao invés de

RestauranteService:

```
##### fj33-eats-distancia-
service/src/main/java(br/com/caelum/eats/distancia/DistanciaService.java

// anotações ...
class DistanciaService {

    // código omitido ...

    private RestauranteService restaurantes;
    private RestauranteRepository restaurantes;

    // restante do código ...

}
```

Limpe o import:

```
import br.com.caelum.eats.restaurante.Restaurante
```

Simplificando o restaurante do serviço de distância

O eats-distancia-service necessita apenas de um subconjunto das informações do restaurante: o `id`, o `cep`, se o restaurante está aprovado e o `tipoDeCozinhaId`.

Enxugue a classe Restaurante do pacote `br.com.caelum.eats.distancia`, deixando apenas as informações realmente necessárias:

```
##### fj33-eats-distancia-
service/src/main/java(br/com/caelum/eats/distancia/Restaurante.java

// anotações ...
```

```
public class Restaurante {  
  
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private Long id;  
  
    @NotNull @Size(max=18)  
    private String cep;  
  
    @NotNull @Size(max=255)  
    private String nome;  
  
    @Size(max=1000)  
    private String descricao;  
  
    @NotNull @Size(max=9)  
    private String cep;  
  
    @NotNull @Size(max=300)  
    private String endereco;  
  
    @Positive  
    private BigDecimal taxaEntregaMReais;  
  
    @Positive @Min(10) @Max(180)  
    private Integer tempoDeEntregaMinimoEmMinutos;  
  
    @Positive @Min(10) @Max(180)  
    private Integer tempoDeEntregaMaximoEmMinutos;  
  
    private Boolean aprovado;  
  
    @Column(nullable=false)  
    private Long tipoDeCozinhaId;  
}
```

fj33-eats-distancia-
service/src/main/java;br/com/caelum/eats/distancia/Restaurante.java

O conteúdo da classe Restaurante do serviço de distância ficará da seguinte maneira:

```
// anotações ...
public class Restaurante {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String cep;

    private Boolean aprovado;

    private Long tipoDeCozinhaId;

}
```

Alguns dos imports podem ser removidos:

```
import java.math.BigDecimal;
import javax.persistence.Table;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Positive;
import javax.validation.constraints.Size;
```

Fazendo a UI chamar serviço de distância

Abra o projeto `fj33-eats-ui` e defina uma nova propriedade `distanciaUrl` no arquivo `environment.ts`:

```
##### fj33-eats-ui/src/environments/environment.ts
```

```
export const environment = {
  production: false,
  baseUrl: '//localhost:8080'
, pagamentoUrl: '//localhost:8081'
, distanciaUrl: '//localhost:8082'
};
```

Modifique a classe `RestauranteService` para que use `distanciaUrl` nos métodos `maisProximosPorCep`, `maisProximosPorCepETipoDeCozinha` e `distanciaPorCepEId`:

```
##### f33-eats-ui/src/app/services/restaurante.service.ts
```

```
export class RestauranteService {

  private API = environment.baseUrl;
  private DISTANCIA_API = environment.distanciaUrl; // adicionei

  // código omitido ...

  maisProximosPorCep(cep: string): Observable<any> {
    return -this.-http.-get(`-${this.API}/restaurantes/
      return this.http.get(` ${this.DISTANCIA_API}/restaurantes/
    }

  maisProximosPorCepETipoDeCozinha(cep: string, tipoDeCozinha]: Observable<any> {
    return -this.-http.-get(`-${this.API}/restaurantes/
      return this.http.get(` ${this.DISTANCIA_API}/restaurantes/
    }

  distanciaPorCepEId(cep: string, restauranteId: string): Observable<any> {
    return -this.-http.-get(`-${this.API}/restaurantes/
      return this.http.get(` ${this.DISTANCIA_API}/restaurantes/
    }

  // restante do código ...
}
```

}

Removendo código de distância do monólito

Remova a dependência a eats-distancia do pom.xml do módulo eats-application:

```
##### fj33-eats-monolito-modular/eats/eats-application/pom.xml
```

```
<dependency>
  <groupId>br.com.caetum</groupId>
  <artifactId>eats-distancia</artifactId>
  <version>${project.version}</version>
</dependency>
```

No pom.xml do projeto eats, o módulo pai, remova a declaração do módulo eats-distancia:

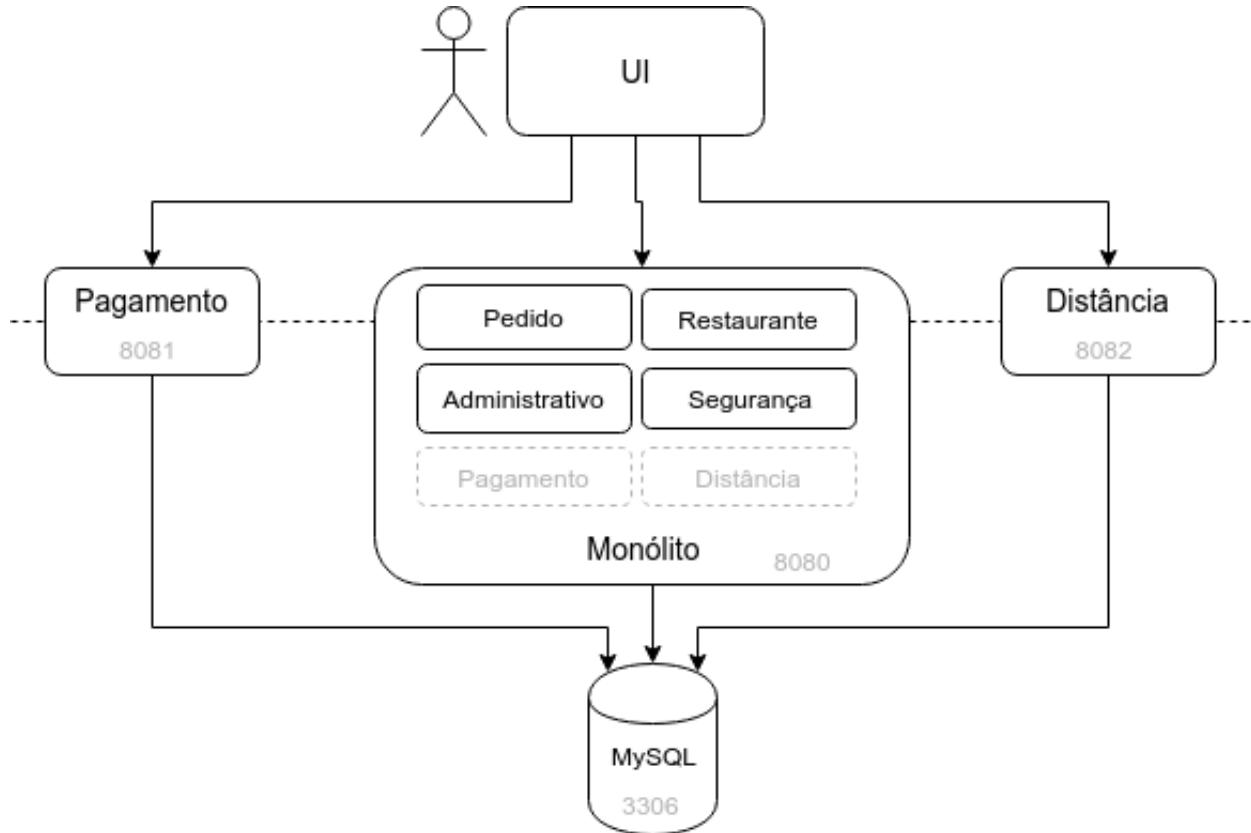
```
##### fj33-eats-monolito-modular/eats/pom.xml
```

```
<modules>
  <module>eats-administrativo</module>
  <module>eats-restaurante</module>
  <module>eats-pedido</module>
  <module>eats-distancia</module>
  <module>eats-seguranca</module>
  <module>eats-application</module>
</modules>
```

Apague o código do módulo eats-distancia do monólito. Pelo Eclipse, tecle *Delete* em cima do módulo, selecione a opção *Delete project contents on disk (cannot be undone)* e clique em *OK*.

Ufa! Mais um serviço extraído do monólito. Em um projeto real, isso seria

feito em paralelo com a extração do serviço de pagamentos, por times independentes. A exploração de novas tecnologias, afim de melhorar o desempenho da busca de restaurantes próximos a um dado CEP, poderia ser feita de maneira separada do monólito. Contudo, o BD continua monolítico.



Exercício: Executando o novo serviço de distância

1. Em um Terminal, clone o projeto do serviço de distância para o seu Desktop:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-di
```

No workspace de Microservices do Eclipse, use o menu *File > Import > Existing Maven Projects* para importar o projeto `fj33-eats-distancia-service`.

Execute a classe `EatsDistanciaServiceApplication`.

Use o cURL para disparar chamadas ao serviço de distância.

Para buscar os restaurantes mais próximos ao CEP 71503-510:

```
curl -i http://localhost:8082/restaurantes/mais-proximos/71503
```

A resposta será algo como:

```
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 22 May 2019 18:44:13 GMT
```

```
[{"restauranteId": 1, "distancia": 8.357388557756333824499961}, {"restauranteId": 2, "distancia": 8.170183211279926638326287}]
[
```

Para buscar os restaurantes mais próximos ao CEP 71503-510 com o tipo de cozinha *Chinesa* (que tem o id 1):

```
curl -i http://localhost:8082/restaurantes/mais-proximos/71503?cozinha=1
```

A resposta será semelhante a:

```
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 22 May 2019 18:44:13 GMT
```

```
[{"restauranteId": 1, "distancia": 18.382449996133800595998764}]
```

Para descobrir a distância de um dado CEP a um restaurante específico:

```
curl -i http://localhost:8082/restaurantes/71503510/restaurant
```

Teremos um resultado parecido com:

```
HTTP/1.1 200
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 22 May 2019 18:44:13 GMT
```

```
{ "restauranteId": 1, "distancia":13.9599876403835738853824495 }
```

2. Interrompa o monólito, caso esteja sendo executado.

No diretório do monólito, vá até a branch `cap3-extrai-distancia-service`, que tem as alterações no monólito logo após da extração do serviço de distância:

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap3-extrai-distancia-service
```

Execute novamente a classe `EatsApplication` do módulo `eats-application` do monólito.

3. Interrompa a UI, se estiver sendo executada.

No diretório da UI, altere a branch para `cap3-extrai-distancia-service`, que contém as mudanças para chamar o novo serviço de distância:

```
cd ~/Desktop/fj33-eats-ui
```

```
git checkout -f cap3-extrai-distancia-service
```

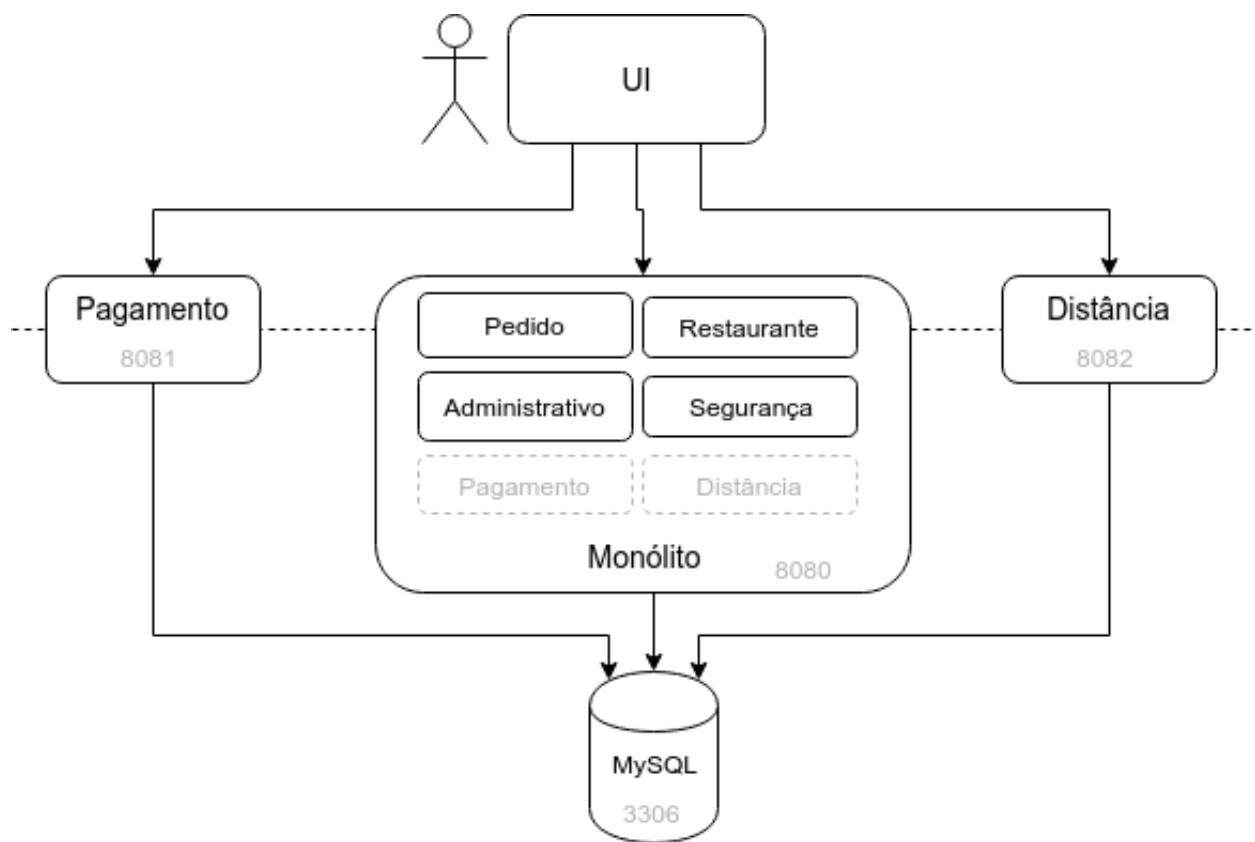
Com o comando `ng serve`, garanta que o front-end esteja rodando.

Acesse `http://localhost:4200`. Busque os restaurantes de um dado CEP, escolha um dos restaurantes retornados e, na tela de detalhes do restaurante, verifique que a distância aparece logo acima da descrição. Deve funcionar!

Migrando dados

Banco de Dados Compartilhado: uma boa ideia?

Mesmo depois de extraímos os serviços de Pagamentos e Distância, mantivemos o mesmo MySQL monolítico.



Há vantagens em manter um BD Compartilhado, como as mencionadas por Chris Richardson na página [Shared Database](#) (RICHARDSON, 2018b):

- os desenvolvedores estão familiarizados com BD relacionais e soluções de ORM
- há o reforço da consistência dos dados com as garantias ACID (Atomicidade, Consistência, Isolamento e Durabilidade) do MySQL e de outros BDs relacionais
- é possível fazer consultas complexas de maneira eficiente, com joins de dados dos múltiplos serviços
- um único BD é mais fácil de operar e monitorar

Era comum em adoções de SOA que fosse mantido um BD Corporativo,

que mantinha todos os dados da organização.

Porém, há diversas desvantagens, como as discutidas por Richardson na mesma página:

- dificuldade em escalar BDs, especialmente, relacionais
- necessidade de diferentes paradigmas de persistência para alguns serviços, como BDs orientados a grafos, como Neo4J, ou BDs bons em armazenar dados pouco estruturados, como MongoDB
- acoplamento no desenvolvimento, fazendo com que haja a necessidade de coordenação para a evolução dos schemas do BD monolítico, o que pode diminuir a velocidade dos times
- acoplamento no *runtime*, fazendo com que um lock em uma tabela ou uma consulta pesada feita por um serviço afete os demais

No livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman foca bastante no acoplamento gerado pelo Shared Database. Newman diz que essa Integração pelo BD é muito comum no mercado. A facilidade de obter e modificar dados de outro serviço diretamente pelo BD explica a popularidade. É como se o schema do BD fosse uma API. O acoplamento é feito por detalhes de implementação e uma mudança no schema do BD quebra os "clientes" da integração. Uma migração para outro paradigma de BD fica impossibilitada. A promessa de autonomia de uma Arquitetura de Microservices seria uma promessa não cumprida. Ficaria difícil evitar mudanças que quebram o contrato, o que inevitavelmente levaria a medo de qualquer mudança.

Sam Newman conta, no livro [Monolith to Microservices](#) (NEWMAN, 2019), sobre uma experiência em um banco de investimento em que o time chegou a conclusão que uma reestruturação do schema do BD iriam aumentar drasticamente a performance do sistema. Então, descobriram que outras aplicações tinham acesso de leitura, e até de escrita, ao BD. Como o mesmo usuário e senha eram utilizados, era impossível saber quais eram essas aplicações e o que estava sendo acessado. Por uma análise de tráfego de rede, estimaram que cerca de 20 outras aplicações estavam usando integração pelo BD. Eventualmente, as credenciais

foram desabilitadas e o time esperou o contato das pessoas que mantinham essas aplicações. Então, descobriram que a maioria das aplicações não tinha uma equipe para mantê-las. Ou seja, o schema antigo teria que ser mantido. O BD passou a ser uma API pública. O time de Newman resolveu o problema criando um schema privado e projetando os dados em Views públicas com informações limitadas, para que os outros sistemas acessassem.

Em [um tweet](#) (PONTE, 2019), Rafael Ponte, deixa claro que usar um Shared Database é integração de sistemas e que, nesse cenário, é preciso manter um contrato bem definido. Dessa maneira, a evolução dos schemas e a manutenção de longo prazo ficam facilitadas. Segundo Ponte, o problema é que o mercado utilizado o que ele chamada de *orgia de dados*, em que os sistemas acessando diretamente os dados, sem um contrato claro. E BDs permitem diversas maneiras de definir contratos:

- Grants a schemas
- API via procedures
- API via views
- API via tabelas de integração
- Eventos e signals

Rafael Ponte explica que tabelas de integração são especialmente úteis, pois são simples de implementar e provêm um contrato bem definido. Nenhum sistema conhece a estrutura de tabelas do outro, os detalhes de implementação do BD original. Há, portanto, *information hiding* e encapsulamento. Um sistema produz dados para a tabela de integração e outro sistema consome esses dados, na cadência em que desejar.

Ponte afirma existem diversas estratégias de integração via BD. Procedures permitem uma maior flexibilidade na implementação, permitindo validação, enrichment, queuing, roteamento, etc. Views funcionam como uma API read-only, onde o sistema consumidor não conhece nada da estrutura interna de tabelas.

Um Banco de Dados por serviço

No artigo [Database per service](#) (RICHARDSON, 2018c), Chris Richardson argumenta em favor de um BD separado para cada serviço. O BD é um detalhe de implementação do serviço e não deve ser acessado diretamente por outros serviços.

Pattern: Database per service

Faça com que os dados de um Microservice sejam apenas acessíveis por sua API.

Podemos fazer um paralelo com o conceito de encapsulamento em Orientação a Objetos. Um objeto deve proteger seus detalhes internos e tornar os atributos privados é uma condição para isso. Qualquer manipulação dos atributos deve ser feita pelos métodos públicos.

No nível de serviços, os dados estarão em algum mecanismo de persistência, que devem ser privados. O acesso aos dados deve ser feito pela API do serviço, não diretamente.

A autonomia e o desacoplamento oferecidos por essa abordagem cumprem a promessa de uma Arquitetura de Microservices. As mudanças na implementação da persistência de um serviço não afetariam os demais. Cada serviço poderia usar o tipo de BD mais adequado às suas necessidades. Seria possível escalar um BD de um serviço independentemente, otimizando recursos computacionais.

Claro, não deixam de existir pontos negativos nessa abordagem. Temos que lidar com uma possível falta de consistência dos dados. Cenários de negócio transacionais passam a ser um desafio. Consultas que juntam dados de vários serviços são dificultadas. Há também a complexidade de operar e monitorar vários BDs distintos. Se forem usados múltiplos paradigmas de persistência, talvez seja difícil ter os especialistas necessários na organização.

Um Servidor de Banco de Dados por serviço

Richardson cita algumas estratégias para tornar privados os dados

persistidos de um serviço:

- Tabelas Privadas por Serviço: cada serviço tem um conjunto de tabelas que só deve ser acessada por esse serviço. Pode ser reforçado por um usuário para cada serviço e o uso de grants.
- Schema por Serviço: cada serviço tem seu próprio Schema no BD.
- Servidor de BD por Serviço: cada serviço tem seu próprio servidor de BD separado. Serviços com muitos acessos ou consultas pesadas trazem a necessidade de um servidor de BD separado.

Ter Tabelas Privadas ou um Schema Separado por serviço pode ser usado como um passo em direção à uma eventual migração para um servidor separado.

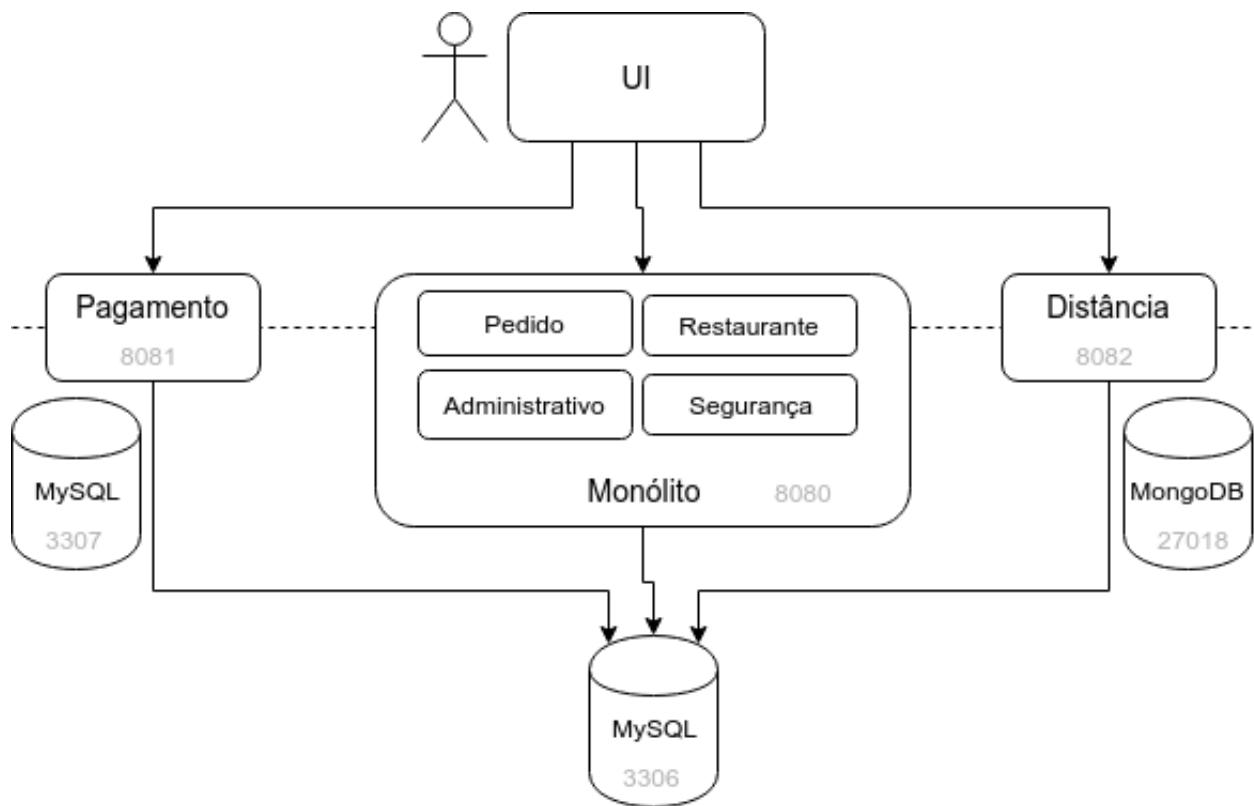
Quando há a necessidade, para um serviço, de mecanismo de persistência com um paradigma diferente dos demais serviços, o servidor do BD deverá ser separado.

Bancos de Dados separados no Caelum Eats

No Caelum Eats, vamos criar servidores de BD separados do MySQL do Monólito para os serviços de Pagamentos e de Distância.

O time de Pagamentos também usará um MySQL.

Já o time de Distância planeja explorar uma nova tecnologia de persistência, mais alinhada com as necessidades de geoprocessamento: o MongoDB. É um BD NoSQL, orientado a documentos, com um paradigma diferente do relacional.



Por enquanto, apenas criaremos os servidores de cada BD. Daria trabalho instalar e configurar os BDs manualmente. Então, para essas necessidades de infraestrutura, usaremos o Docker!

O curso [Infraestrutura ágil com Docker e Docker Swarm](#) (DO-26) aprofunda nos conceitos do Docker e tecnologias relacionadas.

Criando uma nova instância do MySQL a partir do Docker

Abra um Terminal e baixe a imagem do MySQL 5.7 para sua máquina com o seguinte comando:

```
docker image pull mysql:5.7
```

Suba um container do MySQL 5.7 com o seguinte comando:

```
docker container run --rm -d -p 3307:3306 --name eats.mysql -e
```

Usamos as configurações:

- `--rm` para remover o container quando ao sair.
- `-d`, ou `--detach`, para rodar o container no background, imprimindo o id do container e liberando o Terminal para outros comandos.
- `-p`, ou `--publish`, que associa a porta do container ao host. No nosso caso, associamos a porta 3307 do host à porta padrão do MySQL (3306) do container.
- `--name`, define um apelido para o container.
- `-e`, ou `--env`, define variáveis de ambiente para o container. No caso, definimos a senha do usuário `root` por meio da variável `MYSQL_ROOT_PASSWORD`. Também definimos um database a ser criado na inicialização do container e seu usuário e senha, pelas variáveis `MYSQL_DATABASE`, `MYSQL_USER` e `MYSQL_PASSWORD`, respectivamente.

Mais detalhes sobre essas opções podem ser encontrados em:

<https://docs.docker.com/engine/reference/commandline/run/>

Liste os containers que estão sendo executados pelo Docker com o comando:

Deve aparecer algo como:

CONTAINER ID	IMAGE
183bc210a6071b46c4dd790858e07573b28cfa6394a7017cb9fa6d4c9af71563	mysql:5.

É possível formatar as informações, deixando a saída do comando mais enxuta. Para isso, use a opção `--format`:

```
docker container ps --format "{{.Image}}\t{{.Names}}"
```

O resultado será semelhante a:

```
mysql:5.7      eats.mysql
```

Acesse os logs do container `eats.mysql` com o comando:

```
docker container logs eats.mysql
```

Podemos executar um comando dentro de um container por meio do `docker exec`.

Para acessar a interface de linha de comando do MySQL (o comando `mysql`) com o database e usuário criados em passos anteriores, devemos executar:

```
docker container exec -it eats.mysql mysql -u pagamento -p eats
```

A opção `-i` (ou `--interactive`) repassa a entrada padrão do host para o container do Docker.

Já a opção `-t` (ou `--tty`) simula um Terminal dentro do container.

Informe a senha `pagamento123`, registrada em passos anteriores.

Devem ser impressas informações sobre o MySQL, cuja versão deve ser *5.7.26 MySQL Community Server (GPL)*.

Digite o seguinte comando:

Deve ser exibido algo semelhante a:

```
+-----+
| Database          |
+-----+
| information_schema |
```

```
| eats_pagamento      |
+-----+
2 rows in set (0.00 sec)
```

Para sair, digite `exit`.

Pare a execução do container `eats.mysql` com o comando a seguir:

```
docker container stop eats.mysql
```

Criando uma instância do MongoDB a partir do Docker

Baixe a imagem do MongoDB 3.6 com o comando a seguir:

```
docker image pull mongo:3.6
```

Execute o MongoDB 3.6 em um container com o comando:

```
docker container run --rm -d -p 27018:27017 --name eats.mongo
```

Note que mudamos a porta do host para 27018. A porta padrão do MongoDB é 27017.

Liste os containers, obtenha os logs de `eats.mongo` e pare a execução. Use como exemplo os comandos listados no exercício do MySQL.

Simplificando o gerenciamento dos containers com Docker Compose

O Docker Compose permite definir uma série de services (cuidado com o nome!) que permitem descrever a configuração de containers. Com essa

ferramenta, é possível disparar novas instâncias de maneira muito fácil!

Para isso, basta definirmos um arquivo `docker-compose.yml`. Os services devem ter um nome e referências às imagens do Docker Hub e podem ter definições de portas utilizadas, variáveis de ambiente e diversas outras configurações.

```
##### docker-compose.yml
```

```
version: '3'
```

```
services:
```

```
    mysql.pagamento:
```

```
        image: mysql:5.7
```

```
        restart: on-failure
```

```
        ports:
```

```
            - "3307:3306"
```

```
        environment:
```

```
            MYSQL_ROOT_PASSWORD: caelum123
```

```
            MYSQL_DATABASE: eats_pagamento
```

```
            MYSQL_USER: pagamento
```

```
            MYSQL_PASSWORD: pagamento123
```

```
    mongo.distancia:
```

```
        image: mongo:3.6
```

```
        restart: on-failure
```

```
        ports:
```

```
            - "27018:27017"
```

Para subir os serviços definidos no `docker-compose.yml`, execute o comando:

A opção `-d`, ou `--detach`, roda os containers no background, liberando o Terminal.

É possível executar um Terminal diretamente em uma dos containers criados pelo Docker Compose com o comando `docker-compose exec`.

Por exemplo, para acessar o comando `mongo`, a interface de linha de comando do MongoDB, do service `mongo.distancia`, faça:

```
docker-compose exec mongo.distancia mongo
```

Você pode obter os logs de ambos os containers com o seguinte comando:

Caso queira os logs apenas de um container específico, basta passar o nome do service (o termo para uma configuração do Docker Compose). Para o MySQL, seria algo como:

```
docker-compose logs mysql.pagamento
```

Para interromper todos os services sem remover os containers, volumes e imagens associados, use:

Depois de parados com `stop`, para iniciá-los novamente, faça um `docker-compose start`.

É possível parar e remover um service específico, passando seu nome no final do comando.

ATENÇÃO: **evite** usar o comando `docker-compose down` durante o curso. Esse comando apagará todos os dados dos seus BD. Use apenas o comando `docker-compose stop`.

Exercício: Gerenciando containers de infraestrutura com Docker Compose

1. No seu Desktop, defina um arquivo `docker-compose.yml` com o conteúdo anterior, que pode ser encontrado em:
<https://gitlab.com/snippets/1859850>

Observação: mantenha os TABs certinhos. São muito importantes em um arquivo `.yaml`. Em caso de dúvida, peça ajuda ao instrutor.

2. No Desktop, suba ambos os containers, do MySQL e do MongoDB, com o comando:

```
cd ~/Desktop  
docker-compose up -d
```

Observe os containers sendo executados com o comando do Docker:

Deverá ser impresso algo como:

CONTAINER ID	IMAGE	COMMAND	CREATED
49bf0d3241ad	mysql:5.7	"docker-entrypoint..."	26 minutes
4890dc9e898	mongo:3.6	"docker-entrypoint..."	26 minutes

3. Acesse o MongoDB do service `mongo.distancia` com o comando:

```
docker-compose exec mongo.distancia mongo
```

Devem aparecer informações sobre o MongoDB, como a versão, que deve ser algo como *MongoDB server version: 3.6.12*.

Digite o seguinte comando:

Deve ser impresso algo parecido com:

```
admin    0.000GB  
config   0.000GB  
local    0.000GB
```

Para sair, digite `quit()`, com os parênteses.

4. Observe os logs dos services com o comando:

Separando Schemas

Agora temos um container com um MySQL específico para o serviço de Pagamentos. Vamos migrar os dados para esse servidor de BD. Mas o faremos de maneira progressiva e metódica.

No livro [Monolith to Microservices](#) (NEWMAN, 2019), Sam Newman descreve, entre várias abordagens de migração, o uso de Views como um passo em direção a esconder informações entre serviços distintos que usam um Shared Database.

Um passo importante nessa progressão é usar, nos diferentes serviços, Schemas Separados dentro do Shared Database (ou, poderíamos dizer, um *database* separado em um mesmo SGBD). Como comentado em capítulos anteriores, no livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman recomenda o uso de Schemas Separados mesmo mantendo o código no Monólito Modular.

Pattern: Schemas separados

Inicie a decomposição dos dados do Monólito usando Schemas Separados no mesmo servidor de BD, alinhados aos Bounded Contexts.

No livro [Monolith to Microservices](#) (NEWMAN, 2019), Sam Newman argumenta que usar Schemas Separados seria uma *separação lógica*, enquanto usar um servidor de BD separado seria uma *separação física*. A separação lógica permite mudanças independentes e encapsulamento, enquanto que a separação física potencialmente melhor vazão, latência, uso de recursos e isolamento de falhas. Contudo, a separação lógica é uma condição para a separação física.

Mesmo com Schemas Separados, se for utilizado um mesmo servidor de

BD, podemos ter usuários que tem acesso a mais de um Schema e, portanto, que conseguem fazer migração de dados.

Uma vez que decidimos por Schemas Separados, a integridade oferecida por *foreign keys* (FKs) nos BDs relacionais tem que ser deixada de lado. Essa perda traz duas consequências:

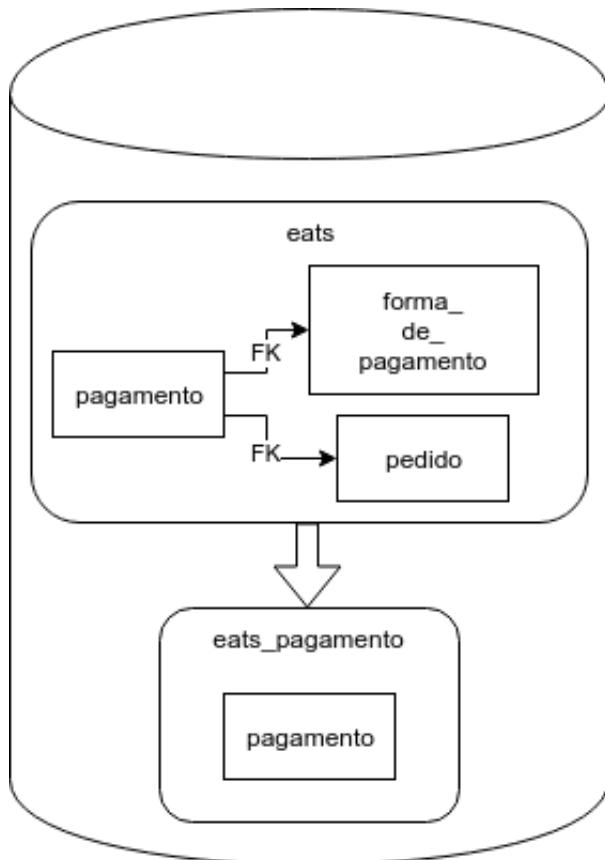
- consultas que fazem join dos dados tem que ser feitas em memória, tornando a operação mais lenta
- perda de consistência dos dados, cujos efeitos discutiremos mais adiante

Tanto Sam Newman como Chris Richardson indicam como referência para a evolução de BDs relacionais o livro [Refactoring Databases](#) (SADALAGE; AMBER, 2006) de Pramod Sadalage e Scott Ambler.

Separando schema do BD de pagamentos do monólito

Em capítulos anteriores, quebramos o Modelo de Domínio de Pagamento para que não dependesse de `FormaDePagamento` nem de `Pedido`, que são dos módulos Administrativo e de Pedido do Monólito, respectivamente. Porém, as FKs foram mantidas.

Nessa momento, criaremos um Schema Separado para o serviço de Pagamentos. Não existiram FKs às tabelas que representam `FormaDePagamento` e `Pedido`. Serão mantidos apenas os ids dessas tabelas.



Mas como efetuar essa alteração?

Criaremos scripts .sql com instruções DDL (Data Definition Language), como `CREATE TABLE` ou `ALTER TABLE`, para criar as estruturas das tabelas e, instruções DML (Data Manipulation Languagem), como `INSERT` ou `UPDATE`, para popular os dados.

Para executar esses scripts, usaremos uma ferramenta de Migration.

Entre as bibliotecas mais usadas para Migration em projetos Java estão Liquibase e Flyway. Ambas estão bem integradas com o Spring Boot. O Liquibase permite que as Migrations sejam definidas em XML, JSON, YAML e SQL. Já no Flyway, podem ser usados SQL e Java. Uma grande vantagem do Liquibase é a possibilidade de ter Migrations de rollback na versão *community*.

Uma ferramenta de migração de dados tem uma maneira de definir a versão dos scripts e de controlar quais scripts já foram executados. Assim, é possível recriar um BD do zero, saber qual é a versão atual de um BD específico e evoluir para novas versões.

O Monólito já usa o Flyway para DDL e DML.

No caso do Flyway, há uma nomenclatura padrão para o nome dos arquivos .sql:

- Um v como prefixo.
- Um número de versão incremental e único, como 0001 ou 0919. Pode haver pontos para versões intermediárias, como 2.5
- Dois underscores (_) como separador
- Uma descrição
- A extensão .sql como sufixo.

Um exemplo de nome de arquivo seria v0001__cria-tabela-pagamento.sql.

Para manter a versão atual do BD e saber quais scripts foram executados, o Flyway mantém uma tabela chamada `flyway_schema_history`. No livro [Refactoring Databases](#) (SADALAGE; AMBER, 2006), os autores já demonstram a necessidade de manter qual a última versão do Schema em uma tabela que chamam de *Database Configuration*.

Um novo Schema não teria essa tabela e, portanto, estaria vazia, significando que todos os scripts devem ser executados. Essa tabela tem colunas como:

- `version`, que contém as versões executadas;
- `script`, que contém o nome do arquivo executado;
- `installed_on`, que contém a data/hora da execução;
- `checksum`, que contém um número calculado a partir do arquivo .sql;
- `success`, que indica se a execução foi bem sucedida.

Observação: o checksum é checado para todos os scripts ao iniciar a aplicação. Não mude ou remova scripts porque a aplicação pode deixar de subir. Cuidado!

Usaremos o Flyway também para o serviço de Pagamentos.

Para isso, deve ser adicionada uma dependência ao Flyway no `pom.xml` do `eats-pagamento-service`:

```
##### fj33-eats-pagamento-service/pom.xml
```

```
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-core</artifactId>
</dependency>
```

O database do serviço de pagamentos precisa ser modificado para um novo. Podemos chamá-lo de `eats_pagamento`.

```
##### fj33-eats-pagamento-
service/src/main/resources/application.properties
```

```
spring.datasource.url=jdbc:mysql://localhost/eats?use
spring.datasource.url=jdbc:mysql://localhost/eats_pagamento?cr
```

O mesmo usuário `root` deve ter acesso a ambos os databases: `eats`, do monólito, e `eats_pagamento`, do serviço de pagamentos. Dessa maneira, é possível executar scripts que migram dados de um database para outro.

Numa nova pasta `db/migration` em `src/main/resources` deve ser criada uma primeira migration, que cria a tabela `pagamento`. O arquivo pode ter o nome `V0001_cria-tabela-pagamento.sql` e o seguinte conteúdo:

```
##### fj33-eats-pagamento-
service/src/main/resources/db/migration/V0001_cria-tabela-
pagamento.sql
```

```
CREATE TABLE pagamento (
```

```
    id bigint(20) NOT NULL AUTO_INCREMENT,  
    valor decimal(19,2) NOT NULL,  
    nome varchar(100) DEFAULT NULL,  
    numero varchar(19) DEFAULT NULL,  
    expiracao varchar(7) NOT NULL,  
    codigo varchar(3) DEFAULT NULL,  
    status varchar(255) NOT NULL,  
    forma_de_pagamento_id bigint(20) NOT NULL,  
    pedido_id bigint(20) NOT NULL,  
    PRIMARY KEY (id)  
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

O conteúdo acima pode ser encontrado na seguinte URL:

<https://gitlab.com/snippets/1859564>

Uma segunda migration, de nome v0002_migra-dados-de-pagamento.sql, obtém os dados do database eats, do monólito, e os insere no database eats_pagamento. Crie o arquivo em db/migration, conforme a seguir:

```
##### fj33-eats-pagamento-  
service/src/main/resources/db/migration/V0002_migra-dados-de-  
pagamento.sql
```

```
insert into eats_pagamento.pagamento  
  (id, valor, nome, numero, expiracao, codigo, status, forma_c  
   select id, valor, nome, numero, expiracao, codigo, status,  
   from eats.pagamento;
```

O trecho de código acima pode ser encontrado em:

<https://gitlab.com/snippets/1859568>

Essa migração só é possível porque o usuário tem acesso aos dois databases.

Após executar EatsPagamentoServiceApplication, nos logs, devem

aparecer informações sobre a execução dos scripts .sql. Algo como:

```
2019-05-22 18:33:56.439  INFO 30484 --- [ restartedMain] o.f.c.internal.li  
2019-05-22 18:33:56.448  INFO 30484 --- [ restartedMain] com.zaxxer.hikari  
2019-05-22 18:33:56.632  INFO 30484 --- [ restartedMain] com.zaxxer.hikari  
2019-05-22 18:33:56.635  INFO 30484 --- [ restartedMain] o.f.c.internal.da  
2019-05-22 18:33:56.708  INFO 30484 --- [ restartedMain] o.f.core.internal  
2019-05-22 18:33:56.840  INFO 30484 --- [ restartedMain] o.f.c.i.s.JdbcTab  
2019-05-22 18:33:57.346  INFO 30484 --- [ restartedMain] o.f.core.internal  
2019-05-22 18:33:57.349  INFO 30484 --- [ restartedMain] o.f.core.internal  
2019-05-22 18:33:57.596  INFO 30484 --- [ restartedMain] o.f.core.internal  
2019-05-22 18:33:57.650  INFO 30484 --- [ restartedMain] o.f.core.internal
```

Para verificar se o conteúdo do database `eats_pagamento` condiz com o esperado, podemos acessar o MySQL em um Terminal:

```
mysql -u <SEU USUÁRIO> -p eats_pagamento
```

<SEU USUÁRIO> deve ser trocado pelo usuário do banco de dados. Deve ser solicitada uma senha.

Dentro do MySQL, deve ser executada a seguinte query:

Os pagamentos devem ter sido migrados.

Novos pagamentos serão armazenados apenas no schema `eats_pagamento`. Os dados do serviço de Pagamentos são suficientemente independentes para serem mantidos em um BD separado.

É importante lembrar que a mudança do status do pedido para *PAGO*, que perdemos ao extrair o serviço de Pagamentos do Monólito, ainda precisa ser resolvida. Faremos isso mais adiante.

Exercício: migrando dados de pagamento para

schema separado

1. Pare a execução de `EatsPagamentoServiceApplication`.

Obtenha as configurações e scripts de migração para outro schema da branch `cap5_migrando_pagamentos_para_schema_separado` do serviço de pagamentos:

```
cd ~/Desktop/fj33-eats-pagamento-service  
git checkout -f cap5_migrando_pagamentos_para_schema_separado
```

Execute `EatsPagamentoServiceApplication`. Observe o resultado da execução das migrations nos logs.

2. Verifique se o conteúdo do database `eats_pagamento` condiz com o esperado, digitando os seguintes comandos em um Terminal:

```
mysql -u <SEU USUÁRIO> -p eats_pagamento
```

Troque `<SEU USUÁRIO>` pelo usuário informado pelo instrutor. Quando solicitada, digite a senha informada pelo instrutor.

Dentro do MySQL, execute a seguinte query:

Os pagamentos devem ter sido migrados. Note as colunas `forma_de_pagamento_id` e `pedido_id`.

Migrando dados de um servidor MySQL para outro

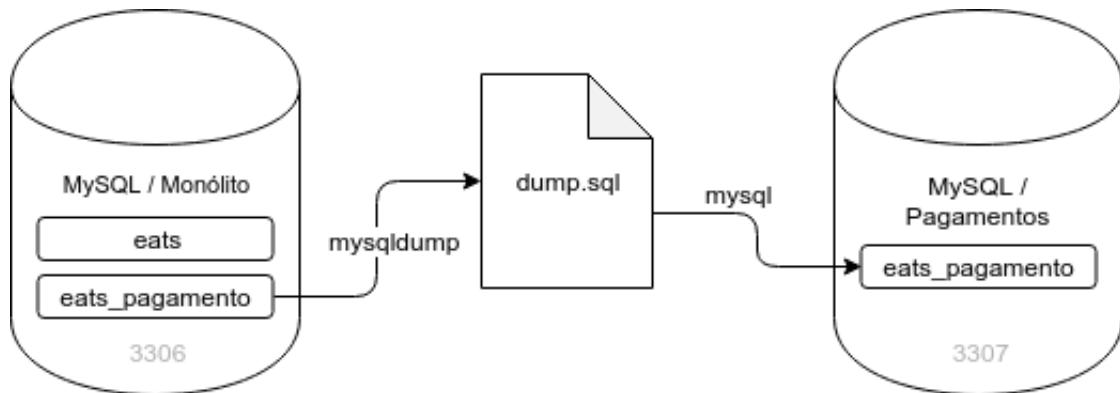
Nesse momento, temos um servidor de BD com Schemas separados para o Monólito e para o serviço de Pagamentos. Também temos um servidor de BD específico para Pagamentos, mas ainda vazio.

No MySQL, o Schema (ou *database*) pode ser criado, se ainda não existir, quando a aplicação conecta com o BD se usarmos a propriedade `createDatabaseIfNotExist`. Em um projeto Spring Boot, isso pode ser definido na URL de conexão do *data source*:

```
spring.datasource.url=jdbc:mysql://localhost:3307/eats_pagame
```

Com o Schema criado no MySQL de Pagamentos, precisamos criar as estruturas das tabelas e migrar os dados. Para isso, podemos gerar um dump com o comando `mysqldump` a partir do Schema `eats_pagamento` do MySQL do Monólito. Será gerado um script `.sql` com todo o DDL e DML do Schema.

O script com o dump pode ser carregado no outro MySQL, específico de Pagamentos, com o comando `mysql`. Mão à obra!



Exercício: migrando dados de pagamento para um servidor MySQL específico

1. Abra um Terminal e faça um dump do dados de pagamento com o comando a seguir:

```
mysqldump -u <SEU USUÁRIO> -p --opt eats_pagamento > eats_pagame
```

Peça ao instrutor o usuário do BD e use em `<SEU USUÁRIO>`. Peça também a senha.

O comando anterior cria um arquivo `eats_pagamento.sql` com todo o schema e dados do database `eats_pagamento`.

A opção `--opt` equivale às opções:

- `--add-drop-table`, que adiciona um `DROP TABLE` antes de cada `CREATE TABLE`.
- `--add-locks`, que faz um `LOCK TABLES` e `UNLOCK TABLES` em volta de cada `CREATE TABLE`.
- `--create-options`, que inclui opções específicas do MySQL nos `CREATE TABLE`.
- `--disable-keys`, que desabilita e habilita PKs e FKs em volta de cada `INSERT`.
- `--extended-insert`, que faz um `INSERT` de múltiplos registros de uma vez.
- `--lock-tables`, que trava as tabelas antes de realizar o dump.
- `--quick`, que lê os registros um a um.
- `--set-charset`, que adiciona `default_character_set` ao dump.

Caso o MySQL monolítico esteja dockerizado, execute o comando `mysqldump` pelo Docker:

```
docker exec -it <NOME-DO-CONTAINER> mysqldump -u root -p --
```

O valor de `<NOME-DO-CONTAINER>` deve ser o nome do container do MySQL do monólito, que pode ser descoberto com o comando `docker ps`.

2. Garanta que o container MySQL do serviço de pagamentos está sendo executado. Para isso, execute em um Terminal:

```
docker-compose up -d mysql.pagamento
```

3. Pela linha de comando, vamos executar, no container de `mysql.pagamento`, o script `eats_pagamento.sql`.

Para isso, vamos usar o comando `mysql` informando *host* e porta do container Docker:

```
mysql -u pagamento -p --host 127.0.0.1 --port 3307 eats_pagame
```

Observação: o comando mysql não aceita localhost, apenas o IP 127.0.0.1.

Quando for solicitada a senha, informe a que definimos no arquivo do Docker Compose: pagamento123.

No caso do comando anterior não funcionar, copie o arquivo eats_pagamento.sql para o container do MySQL de pagamentos usando o Docker:

```
docker cp eats_pagamento.sql <NOME-DO-CONTAINER>:/eats_pagame
```

Então, execute o bash no container do MySQL de pagamentos:

```
docker exec -it <NOME-DO-CONTAINER> bash
```

Finalmente, dentro do container do MySQL de pagamentos, faça o import do dump:

```
mysql -upagamento -p eats_pagamento < eats_pagamento.sql
```

Lembrando que o <NOME-DO-CONTAINER> pode ser descoberto com um docker ps.

4. Para verificar se a importação do dump foi realizada com sucesso, vamos acessar o comando mysql sem passar nenhum arquivo:

```
mysql -u pagamento -p --host 127.0.0.1 --port 3307 eats_pagame
```

Informe a senha `pagamento123`.

Perceba que o MySQL deve estar na versão *5.7.26 MySQL Community Server (GPL)*, a que definimos no arquivo do Docker Compose.

Se o comando `mysql` não funcionar, execute o `bash` no container do MySQL de pagamentos:

```
docker exec -it <NOME-DO-CONTAINER> bash
```

Dentro do container, execute o comando `mysql`:

```
mysql -upagamento -p eats_pagamento
```

Digite o seguinte comando SQL e verifique o resultado:

Devem ser exibidos todos os pagamentos já efetuados!

Para sair, digite `exit`.

Apontando serviço de pagamentos para o BD específico

O serviço de pagamentos deve deixar de usar o MySQL do monólito e passar a usar a sua própria instância do MySQL, que contém seu próprio schema e apenas os dados necessários.

Para isso, basta alterarmos a URL, usuário e senha de BD do serviço de pagamentos, para que apontem para o container Docker do `mysql.pagamento`:

```
##### fj33-eats-pagamento-
service/src/main/resources/application.properties
```

```
spring.datasource.url=jdbc:mysql://localhost/eats_pag  
spring.datasource.url=jdbc:mysql://localhost:3307/eats_pagamer
```

```
spring.datasource.username=<SEU-USUARIO>  
spring.datasource.username=pagamento
```

```
spring.datasource.password=<SEU-SENHA>  
spring.datasource.password=pagamento123
```

Note que a porta 3307 foi incluída na URL, mas mantivemos ainda localhost.

Exercício: fazendo serviço de pagamentos apontar para o BD específico

1. Obtenha as alterações no datasource do serviço de pagamentos da branch cap5_apontando_pagamentos_para_BD_proprio:

```
cd ~/Desktop/fj33-eats-pagamento-service  
git checkout -f cap5_apontando_pagamentos_para_BD_proprio
```

Reinicie o serviço de pagamentos, executando a classe EatsPagamentoServiceApplication.

2. Abra um Terminal e crie um novo pagamento:

```
curl -X POST  
-i  
-H 'Content-Type: application/json'  
-d '{ "valor": 9.99, "nome": "MARIA DE SOUSA", "numero": "77"  
http://localhost:8081/pagamentos
```

Se desejar, baseie-se na seguinte URL, modificando os valores:
<https://gitlab.com/snippets/1859389>

A resposta deve ter sucesso, com status 200 e o um id e status CRIADO no corpo da resposta.

3. Pelo Eclipse, inicie o monólito e o serviço de distância. Suba também o front-end. Faça um novo pedido, até efetuar o pagamento. Deve funcionar!
4. (opcional) Apague a tabela pagamento do database eats, do monólito. Remova também o database eats_pagamento do MySQL do monólito. Atenção: muito cuidado para não remover dados indesejados!

Migrando dados do MySQL para MongoDB

Provisionamos, pelo Docker Compose, um MongoDB específico para o serviço de Distância. Por enquanto, não há dados nesse BD.

O MongoDB não é um BD relacional, mas de um paradigma orientado a documentos.

Não existem tabelas no MongoDB, mas *collections*. As collections armazenam *documents*. Um document é *schemaless*, pois não tem colunas e tipos definidos. Um document tem um *id* como identificador, que deve ser único.

No MongoDB, um *database* agrupa várias collections, de maneira semelhante ao MySQL.

Há um conflito entre os conceitos de um BD relacional como o MySQL e de um BD orientado a documentos, como o MongoDB. Por isso, as estratégias de migração devem ser diferentes.

Devemos exportar um subconjunto dos dados de um Restaurante, que são relevantes para o serviço de Distância: o *id*, o *cep*, o *tipoDeCozinhaId* e o atributo *aprovado*, que indica se o restaurante já foi revisado e aprovado pelo Administrativo do Caelum Eats.

Não é possível fazer um dump para um script .sql. Porém, como a nossa migração é simples, podemos usar um arquivo CSV com os dados de restaurantes que são relevantes para o serviço de Distância. Já que restaurantes não aprovados não são interessantes para o cálculo de distância, podemos fazer uma filtragem, mantendo apenas os restaurantes já aprovados.

Para criar esse CSV a partir do MySQL, podemos usar um `select` com a instrução `into outfile`:

```
select r.id, r.cep, r.tipo_de_cozinha_id from restaurante r w
```

A consulta anterior criará um arquivo `/tmp/restaurantes.csv`, com uma estrutura semelhante à seguinte:

```
##### /tmp/restaurantes.csv
```

```
"1","70238500","1"  
"2","71458-074","6"
```

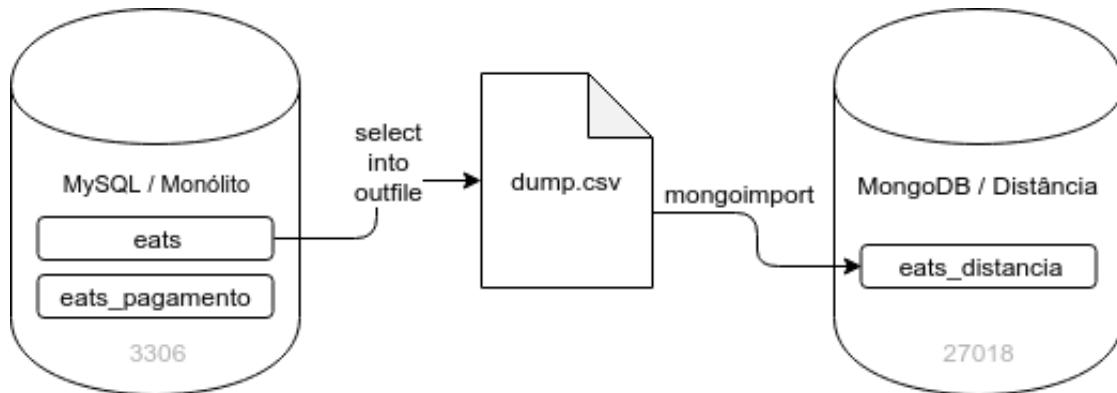
Para importar o CSV para o MongoDB, podemos usar a ferramenta `mongoimport`. Algumas opções do comando:

- `--db`, o database de destino
- `--collection`, a collection de destino
- `--type`, o tipo do arquivo (no caso, um CSV)
- `--file`, o caminho do arquivo a ser importado
- `--fields`, para definir os nomes das propriedades do document

Perceba que não há os nomes das propriedades no arquivo `restaurantes.csv`. Por isso, devemos definí-las usando a opção `--fields`. O campo de identificação do document deve se chamar `_id`.

Para importar o conteúdo do CSV para a collection `restaurantes` do database `eats_distancia`, com os campos `_id`, `cep` e `tipoDeCozinhaId`, devemos executar o seguinte comando:

```
mongoimport --db eats_distancia --collection restaurantes --ty
```



Exercício: migrando dados de restaurantes do MySQL para o MongoDB

1. Em um Terminal, acesse o MySQL do monólito com o usuário `root`, já acessando `eats`, o database monolítico:

Peça a senha de `root` do MySQL para o instrutor. Se não houver senha, omita a opção `-p`.

Na CLI do MySQL, faça uma consulta que obtém os dados relevantes do MySQL para o serviço de distância: o id do restaurante, o cep e o id do tipo de cozinha. O cálculo de distância é feito somente para restaurantes aprovados. Por isso, podemos por um filtro na consulta, mantendo apenas os restaurantes aprovados.

O resultado pode ser exportado para um arquivo CSV, um formato que pode ser facilmente importado em um MongoDB.

```
mysql> select r.id, r.cep, r.tipo_de_cozinha_id from restaurar
```

A query do código anterior pode ser obtida em:

<https://gitlab.com/snippets/1894030>

Caso a instância do MySQL monolítico esteja dockerizada, execute o comando `mysql` pelo Docker:

```
docker exec -it <NOME-DO-CONTAINER> mysql -u root -p eats
```

Digite a senha de root do MySQL do monólito. Execute a query, salvando o arquivo `restaurantes.csv` no diretório `/var/lib/mysql-files/`.

```
mysql> select r.id, r.cep, r.tipo_de_cozinha_id from restaurante
```

A query do código anterior pode ser obtida em:

<https://gitlab.com/snippets/1895021>

Então, obtenha o texto do `restaurantes.csv` e o copie para o diretório `/tmp` com o seguinte comando:

```
docker exec -it <NOME-DO-CONTAINER> cat /var/lib/mysql-files/
```

Lembrando que o `<NOME-DO-CONTAINER>` pode ser descoberto com um `docker ps`.

2. Certifique-se que o container MongoDB do serviço de distância definido no Docker Compose esteja no ar. Para isso, execute em um Terminal:

```
docker-compose up -d mongo.distancia
```

Copie o CSV exportado a partir do MySQL para o seu Desktop:

```
cp /tmp/restaurantes.csv ~/Desktop
```

Descubra o nome do container do MongoDB de distância, com o comando `docker ps`. O container do MongoDB terá como sufixo `mongo.distancia_1`.

Copie o arquivo CSV com os dados exportados para o container do MongoDB:

```
docker cp ~/Desktop/restaurantes.csv <NOME-DO-CONTAINER>:/rest
```

Troque `<NOME-DO-CONTAINER>` pelo nome descoberto no passo anterior.

Execute um bash no container com o comando:

```
docker exec -it <NOME-DO-CONTAINER> bash
```

Importe os dados do CSV para a collection `restaurantes` do MongoDB de distância:

```
mongoimport --db eats_distancia --collection restaurantes --ty
```

O comando acima pode ser encontrado em:

<https://gitlab.com/snippets/1894035>

Ainda no bash do MongoDB, acesse o database de distância com o Mongo Shell:

Dentro do Mongo Shell, verifique a collection de restaurantes foi criada:

Deve ser retornado algo como:

```
restaurantes
```

Veja os documentos da collection `restaurantes` com o comando:

O resultado será semelhante a:

```
{ "_id" : 1, "cep" : 70238500, "tipoDeCozinhaId" : 1 }
{ "_id" : 2, "cep" : "71503-511", "tipoDeCozinhaId" : 7 }
{ "_id" : 3, "cep" : "70238-500", "tipoDeCozinhaId" : 9 }
```

Pronto, os dados foram migrados para o MongoDB!

Apenas os restaurantes já aprovados terão seus dados migrados.

Restaurantes ainda não aprovados ou novos restaurantes não aparecerão para o serviço de distância.

Configurando MongoDB no serviço de distância

O `starter` do Spring Data MongoDB deve ser adicionado ao `pom.xml` do `eats-distancia-service`.

Já as dependências ao Spring Data JPA e ao driver do MySQL devem ser removidas.

```
##### fj33-eats-distancia-service/pom.xml
```

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

<dependency>
    <groupId>mysql</groupId>
```

```
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>

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

Devem ocorrer vários erros de compilação.

A classe `Restaurante` do serviço de distância deve ser modificada, removendo as anotações do JPA.

A anotação `@Document`, do Spring Data MongoDB, deve ser adicionada.

A anotação `@Id` deve ser mantida, porém o import será trocado para `org.springframework.data.annotation.Id`, uma anotação genérica do Spring Data.

Perceba que, apesar do campo ser `_id` no document, o manteremos como `id` no código Java. A anotação `@Id` cuidará de informar qual dos atributos é o identificador do documento e está relacionado ao campo `_id`.

O atributo `aprovado` pode ser removido, já que a migração dos dados foi feita de maneira que o database de distância do MongoDB só contém restaurantes já aprovados.

```
##### fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/Restaurante.java
```

```
@Document(collection = "restaurantes") // adicionado
@Entity
@Data
@NoArgsConstructor
```

```
@AllArgsConstructor
public class Restaurante {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String cep;

    private Boolean aprovado;

    @Column(nullable = false)
    private Long tipoDeCozinhaId;

}
```

Os seguintes imports devem ser removidos:

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
```

E os imports a seguir devem ser adicionados:

Os imports corretos são os seguintes:

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
```

Note que o `@Id` foi importado de `org.springframework.data.annotation` e **não** de `javax.persistence` (do JPA).

A interface `RestauranteRepository` deve ser modificada, para que passe

a herdar de um `MongoRepository`.

Como removemos o atributo `aprovado`, as definições de métodos devem ser ajustadas.

```
##### fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/RestauranteReposito
ry.java
```

```
interface RestauranteRepository extends JpaRepository<Restau
rante, Long>
```

```
    Page<Restaurante> findAllByAprovadoAndTipoDeCozinhaId(
        Long tipoDeCozin
```

```
        boolean aprovado,
        Pageable limit);
```

```
}
```

Os imports devem ser corrigidos:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.mongodb.repository.MongoReposito
```

Como removemos o atributo `aprovado`, é necessário alterar a chamada ao `RestauranteRepository` em alguns métodos do `DistanciaService`:

```
##### fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/DistanciaService.java
```

```
// anotações....
class DistanciaService {

    // atributos...
```

```

public List<RestauranteComDistanciaDto> restaurantesMaisProximos()
{
    List<Restaurante> aprovados = restaurantes.findAll(LIMIT);
    return calculaDistanciaParaOsRestaurantes(aprovados, cep);
}

public List<RestauranteComDistanciaDto> restaurantesDoTipoDeCozinha()
{
    List<Restaurante> aprovadosDoTipoDeCozinha = restaurantes.findAll();
    return calculaDistanciaParaOsRestaurantes(aprovadosDoTipoDeCozinha);
}

// restante do código...
}

```

No arquivo application.properties do eats-distancia-service, devem ser adicionadas as configurações do MongoDB. As configurações de datasource do MySQL e do JPA devem ser removidas.

```

#####
# f33-eats-distancia-
service/src/main/resources/application.properties

spring.data.mongodb.database=eats_distancia
spring.data.mongodb.port=27018

#DATASOURCE-CONFIGS
spring.datasource.url=jdbc:mysql://localhost/eats?useSSL=false
spring.datasource.username=<SEU-USUÁRIO>
spring.datasource.password=<SUA-SENHA>

#JPA-CONFIGS
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true

```

O database padrão do MongoDB é test. A porta padrão é 27017.

Para saber sobre outras propriedades, consulte:

<https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

Exercício: Migrando dados de distância para o MongoDB

1. Interrompa o serviço de distância.

Obtenha o código da branch `cap5_migrando_distancia_para_mongodb` do `fj33-eats-distancia-service`:

```
cd ~/Desktop/fj33-eats-distancia-service
git checkout -f cap5_migrando_distancia_para_mongodb
```

Certifique-se que o MongoDB do serviço de distância esteja no ar com o comando:

```
cd ~/Desktop
docker-compose up -d mongo.distancia
```

Execute novamente a classe `EatsDistanciaServiceApplication`.

Use o cURL para testar algumas URLs do serviço de distância, como as seguir:

```
curl -i http://localhost:8082/restaurantes/mais-proximos/71503510
curl -i http://localhost:8082/restaurantes/mais-proximos/71503510
curl -i http://localhost:8082/restaurantes/71503510/restaurant
```

Observação: como é disparado um GET, é possível testar as URLs

anteriores diretamente pelo navegador.

Para saber mais: Change Data Capture

Mudar dados de um servidor de BD para outro sempre foi um desafio, mesmo para projetos com BDs monolíticos.

Nesse capítulo, fizemos um dump com um script SQL para que fosse importado no MySQL de Pagamentos e um dump com um arquivo CSV para que fosse importado no MongoDB de Distância.

Esse processo tem um uso limitado em uma aplicação real. O sistema que usa o BD original teria que ficar fora do ar durante o dump e import no BD de destino. Se o sistema for mantido no ar, os dados continuariam a ser alterados, inseridos e deletados.

Um dump, porém, é um passo útil para alimentar o novo BD com um snapshot dos dados em um momento inicial. Sam Newman descreve esse passo, no livro [Monolith to Microservices](#) (NEWMAN, 2019), como *Bulk Synchronize Data*.

Logo em seguida, é necessário ter alguma forma de sincronização. Uma maneira é usar **Change Data Capture** (CDC): as modificações no BD original são detectadas e alimentadas no novo BD. A técnica é descrita no livro de Sam Newman e também por Mario Amaral e outros membros da equipe da Elo7, no episódio [Estratégias de migração de dados no Elo7](#) (AMARAL et al., 2019) do podcast Hipsters On The Road.

Pattern: Change Data Capture (CDC)

Capture as mudanças em um BD, para que ações sejam tomadas a partir dos dados modificados.

Uma das maneiras de implementar CDC é usando *triggers*. É algo que os BDs já fazem e não há a necessidade de introduzir nenhuma nova tecnologia. Porém, como Sam Newman diz em seu livro, as ferramentas e o controle de mudanças de triggers deixam a desejar e podem complicar

a aplicação se forem usadas exageradamente. Além disso, na maioria dos BDs só é possível executar SQL. E o destino for um BD não relacional ou um outro sistema?

Sam Newman diz, em seu livro, que uma outra maneira de implementar é utilizar um *Batch Delta Copier*: um programa que roda de tempos em tempos, com um cron ou similares, e consulta o BD original, verificando os dados que foram alterados e copiando os dados para o BD de destino. Porém, a lógica de saber o que foi alterado pode ser complexa e requerer a adição de colunas de *timestamps* nas tabelas. Além disso, as consultas podem ser pesadas, afetando a performance do BD original.

Uma outra maneira de implementar CDC, descrita por Renato Sardinha no post [Introdução ao Change Data Capture \(CDC\)](#) (SARDINHA, 2019) do blog de desenvolvimento da Elo7, é publicar eventos (que estudaremos mais adiante) junto ao código que faz as modificações no BD original. A vantagem é que os eventos poderiam ser consumidos por qualquer outro sistema, não só BDs. Sardinha levanta a complexidade dessa solução: a arquitetura requer um sistema de Mensageria, há a necessidade dos desenvolvedores emitirem esses eventos manualmente e, se alterações forem feitas diretamente no BD por SQL, os eventos não seriam disparados.

A conclusão que os livros, podcasts e posts mencionados chegam é a mesma: podem ser usados os **transaction logs** dos BDs para implementar CDC. A maioria dos BDs relacionais mantém um log de transações (no MySQL, o binlog), que contém um registro de todas as mudanças comitadas e é usado na replicação entre nós de um cluster de BDs.

Existem *transaction log miners* como o [Debezium](#), que lêem o transaction log de BDs como MySQL, PostgreSQL, MongoDB, Oracle e SQL Server e publicam eventos automaticamente em um Message Broker (especificamente o Kafka, no caso do Debezium). A solução é complexa e requer um sistema de Mensageria, mas consegue obter atualizações feitas diretamente no BD e permite que os eventos sejam consumidos

por diferentes ferramentas. Além do Debezium, existem ferramentas semelhantes como o [LinkedIn Databus](#) para o Oracle, o [DynamoDB Streams](#) para o DynamoDB da Amazon e o [Eventuate Tram](#), mantido por Chris Richardson.

Com o CDC funcionando com Debezium ou outra ferramenta parecida, podemos usar uma estratégia progressiva descrita por Sam Newman, em seu livro e, de maneira semelhante, pelo pessoal da Elo 7:

- Inicialmente, é feito um dump (ou Bulk Synchronize)
- Depois, leituras e escritas são mantidas no BD original e os dados são escritos no novo BD com CDC. Assim, é possível observar o comportamento do novo BD com o volume de dados real.
- Em passo seguinte, o BD original fica apenas para leitura e a leitura e escrita é feita no novo BD. No caso de problemas inesperados, o BD original fica como solução paliativa.
- Finalmente, com a estabilização das operações e da migração, o BD original é removido.

É importante salientar que uma Migração de Dados não acontece de uma hora pra outra. Jeff Barr, da Amazon, diz no post [Migration Complete – Amazon's Consumer Business Just Turned off its Final Oracle Database](#) (BARR, 2019), que a migração de BDs Oracle para BDs da AWS de diferentes paradigmas, como DynamoDB, Aurora e Redshift, foi feita ao longo de vários anos.

Integração síncrona (e RESTful)

Em busca das funcionalidades perdidas

Ao extraímos os serviços de Pagamentos e de Distância do Monólito, o Caelum Eats perdeu algumas funcionalidades.

Depois de termos extraído o serviço de Pagamentos, depois de confirmar um pagamento, o status do pedido é mostrado como *REALIZADO* e não como *PAGO*. Isso acontece porque a confirmação é feita no serviço de Pagamentos e removemos o código que atualizava o status do pedido, que é parte do módulo de Pedido do Monólito.

Já no caso do serviço de Distância, a extração em si não fez com que nenhuma funcionalidade fosse perdida. Porém, ao migramos os dados para um BD próprio, copiamos apenas os restaurantes do momento da migração. Mas dados de restaurantes podem ser modificados e novos restaurantes podem ser aprovados. E esses dados de restaurantes não estão sendo replicados para o BD de Distância.

Para que essas funcionalidades perdidas voltem a funcionar, temos que fazer uma integração entre os serviços de Pagamento e Distância e o Monólito.

Integrando sistemas com o protocolo da Web

Vamos implementar essa integração entre sistemas usando o protocolo da Web, o HTTP (Hyper Text Transfer Protocol).

A história do HTTP

Mas da onde vem o HTTP?

No década de 80, (o agora Sir) Tim Berners-Lee trabalhava no CERN, a Organização Europeia para a Pesquisa Nuclear. Em 1989, Berners-Lee criou uma aplicação que provia uma UI para diferentes documentos

como relatórios, notas, documentação, etc. Para isso, baseou-se no conceito de *hypertext*, em que nós de informação são ligados a outros nós formando uma teia em que o usuário pode navegar. E com o nascimento de Internet, a rede mundial de computadores, essa navegação poderia expor informações de diferentes servidores dentro e fora do CERN. Berners-Lee chamou essa teia mundial de documentos ligados uns aos outros de *World Wide Web*.

Então, a equipe de Tim Berners-Lee criou alguns softwares para essa aplicação:

- um servidor Web que provia documentos pela Internet
- um cliente Web, o navegador, que permitia aos usuários visualizar e seguir os links entre os documentos
- o HTML, um formato para os documentos
- o HTTP, o protocolo de comunicação entre o navegador e o servidor Web

O protocolo HTTP foi inicialmente especificado, em sua versão 1.0, pela Internet Engineering Task Force (IETF) na [RFC 1945](#) (BERNES-LEE et al., 1996), em um grupo de trabalho liderado por Tim Berners-Lee, Roy Fielding e Henrik Frystyk. Desde então, diversas atualizações foram feitas no protocolo, em diferentes RFCs.

O HTTP é um protocolo do tipo request/response, em que um cliente como um navegador faz uma chamada ao servidor Web e fica aguardando os dados de resposta. Tanto o cliente como o servidor precisam estar no ar ao mesmo tempo para que a chamada seja feita com sucesso. Portanto, podemos dizer que o HTTP é um protocolo síncrono.

HTTP é o protocolo do maior Sistema Distribuído do mundo, a Web, que usada diariamente por bilhões de pessoas. Podemos dizer que é um protocolo bem sucedido!

O HTTP tem algumas ideias interessantes. Vamos estudá-las a seguir.

Recursos

Um **recurso** é um substantivo, uma coisa que está em um servidor Web e pode ser acessado por diferentes clientes. Pode ser um livro, uma lista de restaurantes, um post em um blog.

Todo recurso tem um endereço, uma **URL** (*Uniform Resource Locator*). Por exemplo, a URL dos tópicos mais recentes de Java no fórum da Alura:

<https://cursos.alura.com.br/forum/subcategoria-java/todos/1>

URL é um conceito da Internet e não só da Web. A especificação inicial foi feita na [RFC 1738](#) (BERNES-LEE et al., 1994) pela IETF. Podemos ter URLs para recursos disponíveis por FTP, SMTP ou AMQP. Por exemplo, uma URL de conexão com o RabbitMQ que usaremos mais adiante no curso:

```
amqp://eats:caelum123@rabbitmq:5672
```

Um **URI** (*Uniform Resource Identifier*) é uma generalização de URLs que identifica um recurso que não necessariamente está exposto em uma rede. Foi especificado inicialmente pela IETF na [RFC 2396](#) (BERNES-LEE et al., 1998). Por exemplo, um Data URI que representa uma imagem PNG de um pequeno ponto vermelho:

```
data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAUAAAAFCAYAAACNbyb1AAAHE1EQVQI12P4//8/w38GIAXDIBKE0DHxgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJgg==
```

Representações

No HTTP, um recurso pode ter diferentes **representações**. Por exemplo, os dados de um livro da [Casa do Código](#), disponível na URL <https://www.casadocodigo.com.br/products/livro-git-github>, pode ser representado em XML:

```
<livro>
  <nome>Controlando versões com Git e GitHub</nome>
  <autores>
    <autor>Alexandre Aquiles</autor>
    <autor>Rodrigo Caneppele</autor>
  </autores>
  <paginas>220</paginas>
  <ISBN>978-85-66250-53-4</ISBN>
</livro>
```

O mesmo recurso, da URL

<https://www.casadocodigo.com.br/products/livro-git-github>, pode ser representado em JSON:

```
{
  "nome": "Controlando versões com Git e GitHub",
  "autores": [
    { "autor": "Alexandre Aquiles" },
    { "autor": "Rodrigo Caneppele" }
  ],
  "paginas": 220,
  "ISBN": "978-85-66250-53-4"
}
```

E é possível ter representações do mesmo recurso, o livro da [Casa do Código](#), em formatos de ebook como PDF, EPUB e MOBI.

As representações de um recurso devem seguir um **Media Type**. Media Types são padronizados pela Internet Assigned Numbers Authority (IANA), a mesma organização que mantém endereços IP, time zones, os top level domains do DNS, etc.

Curiosidade: os Media Types eram originalmente definidos como MIME (Multipurpose Internet Mail Extensions) Types, em uma especificação que definia o conteúdo de emails e seus anexos.

Entre os Media Types comuns, estão:

- `text/html` para HTML
- `text/plain` para texto puro (mas cuidado com a codificação!)
- `image/png` para imagens no formato PNG
- `application/json`, para JSON
- `application/xml` para XML
- `application/pdf` para PDF
- `application/epub+zip` para ebooks EPUB
- `application/vnd.amazon.mobi8-ebook` para ebooks MOBI
- `application/vnd.ms-excel` para arquivos `.xls` do Microsoft Excel

Métodos

Para indicar uma ação a ser efetuada em um determinado recurso, o HTTP define os **métodos**. Se os recursos são os substantivos, os métodos são os verbos, como são comumente chamados.

O HTTP define apenas 9 métodos, cada um com o seu significado e uso diferentes:

- `GET`: usado para obter uma representação de um recurso em uma determinada URL.
- `HEAD`: usado para obter os metadados (cabeçalhos) de um recurso sem a sua representação. É um GET sem o corpo do response.
- `POST`: a representação do recurso passada no request é usada para criar um novo recurso subordinado no servidor, com sua própria URL.
- `PUT`: o request contém uma representação do recurso que será utilizada para atualizar ou criar um recurso na URL informada.
- `PATCH`: o request contém uma representação parcial de um recurso, que será utilizada para atualizá-lo. É uma adição tardia ao protocolo, especificada na [RFC 5789](#) (DUSSEAUT; SNELL, 2010).
- `DELETE`: o recurso da URL informada é removido do servidor.

- **OPTIONS**: retorna os métodos HTTP suportados por uma URL.
- **TRACE**: repete o request, para que o cliente saiba se há alguma alteração feita por servidores intermediários.
- **CONNECT**: transforma o request em um túnel TCP/IP para permitir comunicação encriptada através de um proxy.

Os métodos que não causam efeitos colaterais e cuja intenção é recuperação de dados são classificados como **safe**. São eles: GET, HEAD, OPTIONS e TRACE. Já os métodos que mudam os recursos ou causam efeitos em sistemas externos, como transações financeiras ou transmissão de emails, não são considerados safe.

Já os métodos em que múltiplos requests idênticos tem o mesmo efeito de apenas um request são classificados de **idempotent**, podendo ter ou não efeitos colaterais. Todos os métodos safe são idempotentes e também os métodos PUT e DELETE. Os métodos POST e CONNECT não são idempotentes.

O método PATCH não é considerado nem safe nem idempotente pela [RFC 5789](#) (DUSSEAUlt; SNELL, 2010), já que parte de um estado específico do recurso no servidor e só contém aquilo que deve ser alterado. Dessa maneira, múltiplos PATCHs podem ter efeitos distintos no servidor, pois o estado do recurso pode ser diferente entre os requests.

A ausência de efeitos colaterais e idempotências dos métodos assim classificados fica a cargo do desenvolvedor, não sendo garantidas pelo protocolo nem pelos servidores Web.

Resumindo:

- métodos safe: GET, HEAD, OPTIONS e TRACE
- métodos idempotentes: os métodos safe, PUT e DELETE
- métodos nem safe nem idempotentes: POST, CONNECT e PATCH

É importante notar que apenas esses 9 métodos, ou até um subconjunto

deles, são suficientes para a maioria das aplicações distribuídas. Geralmente são descritos como uma **interface uniforme**.

Cabeçalhos

Tanto um request como um response HTTP podem ter, além de um corpo, metadados nos **Cabeçalhos** HTTP. Os cabeçalhos possíveis são especificados por RFCs na IETF e atualizados pela IANA. Alguns dos cabeçalhos mais utilizados:

- **Accept**: usado no request para indicar qual representação (Media Type) é aceito no response
- **Access-Control-Allow-Origin**: usado no response por chamadas CORS para indicar quais origins podem acessar um recurso
- **Authorization**: usado no request para passar credenciais de autenticação
- **Allow**: usado no response para indicar os métodos HTTP válidos para o recurso
- **Content-type**: a representação (Media Type) usada no request ou no response
- **ETag**: usado no response para indicar a versão de um recurso
- **If-None-Match**: usado no request com um ETag de um recurso, permitindo *caching*
- **Location**: usado no response para indicar uma URL de redirecionamento ou o endereço de um novo recurso

Os cabeçalhos HTTP **Accept** e **Content-type** permitem a **Content negotiation** (negociação de conteúdo), em que um cliente pode negociar com um servidor Web representações aceitáveis.

Por exemplo, um cliente pode indicar no request que aceita JSON e XML como representações, com seguinte cabeçalho:

```
Accept: application/json, application/xml
```

O servidor Web pode escolher entre essas duas representações. Se entre os formatos suportados pelo servidor não estiver JSON mas apenas XML, o response teria o cabeçalho:

```
Content-type: application/xml
```

No corpo do response, estaria um XML representado os dados do recurso.

Um navegador sempre usa o cabeçalho `Accept` em seus requests. Por exemplo, no Mozilla Firefox:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

O cabeçalho anterior indica que o navegador Mozilla Firefox aceita do servidor as representações HTML, XHTML ou XML, nessa ordem. Se nenhuma dessas estiver disponível, pode ser enviada qualquer representação, indicada pelo `*/*`.

O parâmetro `q` utilizado no cabeçalho anterior é um *relative quality factor*, ou fator relativo de qualidade, que indica a preferência por uma representação. O valor varia entre `0`, indicando menor preferência, e `1`, o valor padrão que indica uma maior preferência. No cabeçalho `Accept` anterior, o HTML e XHTML tem valor `1`, XML tem valor `0.9` e qualquer outra representação com valor `0.8`.

Códigos de Status

Um response HTTP pode ter diferentes códigos de status, especificados em RFCs da IETF e mantidos pela IANA.

O primeiro dígito indica a categoria do response:

- 1xx (Informational): o request foi recebido e o processamento continua
- 2xx (Success): o request foi recebido, entendido e aceito com sucesso
- 3xx (Redirection): mais ações são necessárias para completar o request
- 4xx (Client Error): o request é inválido e contém erros causados pelo cliente
- 5xx (Server Error): o servidor falhou em completar um request válido

Alguns dos códigos de status mais comuns:

- 101 Switching Protocols: o cliente solicitou a troca de protocolos e o servidor aceitou, trocando para o protocolo indicado no cabeçalho Upgrade. Usado para iniciar uma conexão a um WebSocket.
- 200 OK: código padrão de sucesso.
- 201 Created: indica que um novo recurso foi criado. Em geral, o response contém a URL do novo recurso no cabeçalho Location.
- 202 Accepted: o request foi aceito para processamento mas ainda não foi completado.
- 204 No Content: o request foi processado com sucesso mas não há corpo no response.
- 301 Moved Permanently: todos os requests futuros devem ser redirecionados para a URL indicada no cabeçalho Location.
- 302 Found: o cliente deve redirecionar para a URL indicada no cabeçalho Location. Navegadores, frameworks e aplicações a implementam como um *redirect*, que seria o intuito do código 303.
- 303 See Other: o request foi completado com sucesso mas o

response deve ser encontrado na URL indicada no cabeçalho `Location` por meio de um `GET`. O intuito era ser utilizado para um *redirect*, de maneira a implementar o pattern *POST/redirect/GET*. Criado a partir do `HTTP 1.1`.

- `304 Not Modified`: indica que a versão (`ETag`) do recurso não foi modificada em relação à informada no cabeçalho `If-None-Match` do request. Portanto, não há a necessidade de transmitir uma representação do recurso. O cliente tem a última versão em seu cache.
- `400 Bad Request`: o cliente enviou um request inválido.
- `401 Unauthorized`: o cliente tentou acessar um recurso protegido em que não tem as permissões necessárias. O request pode ser refeito se passado um cabeçalho `Authorization` que contenha credenciais de um usuário com permissão para acessar o recurso.
- `403 Forbidden`: o cliente tentou uma ação proibida ou o usuário indicado no cabeçalho `Authorization` não tem acesso ao recurso solicitado.
- `404 Not Found`: o recurso não existe no servidor.
- `405 Method Not Allowed`: o cliente usou um método HTTP não suportado pelo recurso solicitado.
- `406 Not Acceptable`: o servidor não consegue gerar uma representação compatível com nenhum valor do cabeçalho `Accept` do request.
- `409 Conflict`: o request não pode ser processado por causa de um conflito no estado atual do recurso, como múltiplas atualizações simultâneas.
- `415 Unsupported Media Type`: o request foi enviado com uma

representação, indicada no `Content-Type`, não suportada pelo servidor.

- `429 Too Many Requests`: o cliente enviou requests excessivos em uma determinada fatia de tempo. Usado ao implementar *rate limiting* com o intuito de prevenir contra ataques Denial of Service (DoS).
- `500 Internal Server Error`: um erro inesperado aconteceu no servidor. O erro do "*Bad, bad server. No donut for your.*" do Orkut e da baleia do Twitter.
- `503 Service Unavailable`: servidor em manutenção ou sobrecarregado temporariamente.
- `504 Gateway Timeout`: o servidor está servindo como um proxy para um request mas não recebeu a tempo um response do servidor de destino.

Links

A Web tem esse nome por ser uma teia de documentos ligados entre si. Links, ou hypertext, são conceitos muito importantes na Web e podem ser usado na integração de sistemas. Veremos como mais adiante.

REST, o estilo arquitetural da Web

Roy Fielding, um dos autores das especificações do protocolo HTTP e cofundador do Apache HTTP Server, estudou diferentes estilos arquiteturais de Sistemas Distribuídos em sua tese de PhD: [Architectural Styles and the Design of Network-based Software Architectures](#) (FIELDING, 2000).

As restrições e princípios que fundamentam o estilo arquitetural da Web são descrita por Fielding da seguinte maneira:

- *Cliente/Servidor*: a UI é separada do armazenamento dos dados, permitindo a portabilidade para diferentes plataformas e

simplificando o Servidor.

- *Stateless*: cada request do cliente deve conter todos os dados necessários, sem tomar vantagem de nenhum contexto armazenado no servidor. Sessões de usuário devem ser mantidas no cliente. Essa característica melhor: a Escalabilidade, já que não há uso de recursos entre requests diferentes; Confiabilidade, já que torna mais fácil a recuperação de falhas parciais; Visibilidade, já que não há a necessidade de monitorar além de um único request. Como desvantagem, há uma piora na Performance da rede, um aumento de dados repetitivos entre requests e uma dependência da implementação correta dos múltiplos clientes.
- *Cache*: os requests podem ser classificados como cacheáveis, fazendo com que o cliente possa reusar o response para requests equivalentes. Assim, a Latência é reduzida de maneira a melhorar a Eficiência, Escalabilidade e a Performance percebida pelo usuário. Porém, a Confiabilidade pode ser afetada caso haja aumento significante de dados desatualizados.
- *Interface Uniforme*: URLs, representações, métodos padronizados e links são restrições da Web que simplificam a Arquitetura e aumentam a Visibilidade das interações, encorajando a evolução independente de cada parte. Por outro lado, são menos eficientes que protocolos específicos.
- *Sistema em Camadas*: cada componente só conhece a camada com a qual interage imediatamente, minimizando a complexidade, aumentando a independência e permitindo intermediários como *load balancers*, *firewalls* e *caches*. Porém, pode ser adicionada latência.
- *Code-On-Demand*: os clientes podem estender as funcionalidades por meio da execução de *applets* e *scripts*, aumentando a Extensibilidade. Por outro lado, a Visibilidade do sistema diminui.

Fielding chama esse estilo arquitetural da Web de Representational State Transfer, ou simplesmente **REST**. Adicionando o sufixo *-ful*, que denota "que possui a característica de" em inglês, podemos chamar serviços que seguem esse estilo arquitetural de *RESTful*.

Um excelente resumo de boas práticas e princípios de uma API RESTful podem ser encontrado no blog da Caelum, no post [REST: Princípios e boas práticas](#) (FERREIRA, 2017), disponível em: <https://blog.caelum.com.br/rest-principios-e-boas-praticas>

Leonard Richardson e Sam Ruby, no livro [RESTful Web Services](#) (RICHARDSON; RUBY, 2007), contrastam serviços no estilo *Remote Procedure Call* (RPC) com serviços *Resource-Oriented*.

Um Web Service no estilo RPC expõe operações de uma aplicação. Não há recursos, representações nem métodos. O SOAP é um exemplo de um protocolo nesse estilo RPC: há apenas um recurso com só uma URL (a do Web Service), há apenas uma representação (XML), há apenas um método (POST, em geral). Cada `operation` do Web Service SOAP é exposta num WSDL.

Richardson e Ruby mostram, no livro, uma API de busca do Google implementada com SOAP. Para buscar sobre "REST" deveríamos efetuar o seguinte request:

```
POST http://api.google.com/search/beta2
Content-Type: application/soap+xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope">
  <soap:Body>
    <gs:doGoogleSearch xmlns:gs="urn:GoogleSearch">
      <q>REST</q>
      ...
    </gs:doGoogleSearch>
  </soap:Body>
</soap:Envelope>
```

A mesma consulta pode ser feita pela API mais moderna do Google, que expõe um recurso `search` que recebe um parâmetro `q` com o termo a ser

consultado:

```
GET http://www.google.com/search?q=REST
Content-Type: text/html
```

O Modelo de Maturidade de Richardson

No post [Richardson Maturity Model](#) (FOWLER, 2010), Martin Fowler explica uma heurística para a maturidade da adoção de REST descrita por Leonard Richardson no "ato 3" de sua palestra [Justice Will Take Us Millions Of Intricate Moves](#) (RICHARDSON, 2008).

O Modelo de Maturidade descrito por Richardson indica uma progressão na adoção das tecnologias da Web.

Nível 0 - O Pântano do POX

No Nível 0 de Maturidade, o HTTP é usado apenas como um mecanismo de transporte para interações remotas no estilo RPC, sem a filosofia da Web.

Fowler usa o termo Plain Old XML (POX) para descrever APIs que provêem só um endpoint, todas as chamadas usam POST, a única representação é XML. As mensagens de erro contém um status code de sucesso (200) com os detalhes do erro no corpo do response. É o caso de tecnologias como SOAP e XML-RPC.

É importante ressaltar que o uso de XML é apenas um exemplo. Poderiam ser utilizados JSON, YAML ou qualquer outra representação. Esse nível de maturidade trata de uma única representação.

Para ilustrar esse tipo de API, Fowler usa um exemplo de consultas médicas. Por exemplo, para agendar uma consulta:

```
POST /agendamentoService HTTP/1.1
```

```
<horariosDisponiveisRequest data="2010-01-04" doutor="huberman">
```

O retorno seria algo como:

```
HTTP/1.1 200 OK
```

```
<listaDeHorariosDisponiveis>
  <horario inicio="14:00" fim="14:50">
    <doutor id="huberman"/>
  </horario>
  <horario inicio="16:00" fim="16:50">
    <doutor id="huberman"/>
  </horario>
</listaDeHorariosDisponiveis>
```

Para marcar uma consulta:

```
POST /agendamentoService HTTP/1.1
```

```
<agendamentoConsultaRequest>
  <horario doutor="huberman" inicio="14:00" fim="14:50"/>
  <paciente id="alexandre"/>
</agendamentoConsultaRequest>
```

A resposta de sucesso, com a confirmação da consulta, seria algo como:

```
HTTP/1.1 200 OK
```

```
<consulta>
  <horario doutor="huberman" inicio="14:00" fim="14:50"/>
```

```
<paciente id="alexandre"/>  
</consulta>
```

Em caso de erro, teríamos uma resposta ainda com o código de status 200, mas com os detalhes do erro no corpo do response:

```
HTTP/1.1 200 OK
```

```
<agendamentoConsultaRequestFailure>  
  <horario doutor="huberman" inicio="14:00" fim="14:50"/>  
  <paciente id="alexandre"/>  
  <motivo>Horário não disponível</motivo>  
</agendamentoConsultaRequestFailure>
```

Perceba que, no exemplo de Fowler, não foi utilizado SOAP mas uma API que usa HTTP, só que sem os conceitos do protocolo.

Nível 1 - Recursos

Um primeiro passo, o Nível 1 de Maturidade, é ter diferentes recursos, cada um com sua URL, ao invés de um único endpoint para toda a API.

No exemplo de consultas médicas de Fowler, poderíamos ter um recurso específico para um doutor:

```
POST /doutores/huberman HTTP/1.1
```

```
<horariosDisponiveisRequest data="2010-01-04"/>
```

O response seria semelhante ao anterior, mas com uma maneira de endereçar individualmente cada horário disponível para um doutor específico:

HTTP/1.1 200 OK

```
<listaDeHorariosDisponiveis>
  <horario id="1234" doutor="huberman" inicio="14:00" fim="14:15">
    <horario id="5678" doutor="huberman" inicio="16:00" fim="16:15">
  </listaDeHorariosDisponiveis>
```

Com um endereço para cada horário, marcar uma consulta seria fazer um **POST** para um recurso específico:

POST /horarios/1234 HTTP/1.1

```
<agendamentoConsulta>
  <paciente id="alexandre"/>
</agendamentoConsulta>
```

O response seria semelhante ao anterior:

HTTP/1.1 200 OK

```
<consulta>
  <horario id="1234" doutor="huberman" inicio="14:00" fim="14:15">
    <paciente id="alexandre"/>
  </consulta>
```

Nível 2 - Verbos HTTP

O segundo passo, o Nível 2 de Maturidade, é utilizar os verbos (ou métodos) HTTP o mais perto o possível de seu intuito original.

Para obter a lista de horários disponíveis, poderíamos usar um **GET**:

```
GET /doutores/huberman/horarios?data=2010-01-04&status=disponivel HTTP/1.1
```

A resposta seria a mesma de antes:

```
HTTP/1.1 200 OK
```

```
<listaDeHorariosDisponiveis>
  <horario id="1234" doutor="huberman" inicio="14:00" fim="14:15">
    <horario id="5678" doutor="huberman" inicio="16:00" fim="16:15">
</listaDeHorariosDisponiveis>
```

No Nível 2 de Maturidade, Fowler diz que o uso de `GET` para consultas é crucial. Como o HTTP define o `GET` como uma operação *safe*, isso significa que não há mudanças significativas no estado de nenhum dos dados. Assim, é seguro invocar um `GET` diversas vezes seguidas e obter os mesmos resultados. Uma consequência importante é que é possível fazer *cache* dos resultados, melhorando a Performance.

Para marcar uma consulta, é necessário um verbo HTTP que permite a mudança de estado. Poderíamos usar um `POST`, da mesma maneira anterior:

```
POST /horarios/1234 HTTP/1.1
```

```
<agendamentoConsulta>
  <paciente id="alexandre"/>
</agendamentoConsulta>
```

Uma API de Nível 2 de Maturidade, deve usar os códigos de status e cabeçalhos a seu favor. Para indicar que uma nova consulta foi criada, com o agendamento do paciente naquele horário podemos usar o status

201 created. Esse status deve incluir, no response, um cabeçalho `Location` com a URL do novo recurso. Essa nova URL pode ser usada pelo cliente para obter, com um `GET`, mais detalhes sobre o recurso que acabou de ser criado. Portanto, a resposta teria alguns detalhes diferentes da anterior:

```
HTTP/1.1 201 Created
Location: horarios/1234/consulta
```

```
<consulta>
  <horario id="1234" doutor="huberman" inicio="14:00" fim="14:
    <paciente id="alexandre"/>
</consulta>
```

No caso de um erro, devem ser usados códigos 4XX ou 5XX. Por exemplo, para indicar que houve uma atualização do recurso por outro cliente, pode ser usado o status 409 `Conflict` com uma nova lista de horários disponíveis no corpo do response:

```
HTTP/1.1 409 Conflict
```

```
<listaDeHorariosDisponiveis>
  <horario id="5678" doutor="huberman" inicio="16:00" fim="16:
</listaDeHorariosDisponiveis>
```

Nível 3 - Controles de Hypermedia

Um dos conceitos importantes do HTTP, o protocolo da Web, é o uso de hypertext.

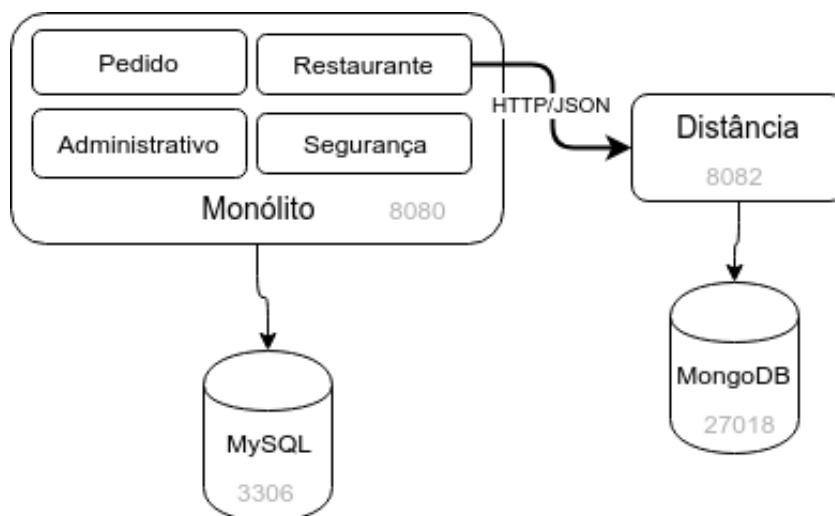
O Nível 3, o nível final, de Maturidade de uma API RESTful é atingido quando são utilizados links. Mas falaremos sobre isso mais adiante.

Cliente REST com RestTemplate do Spring

Precisamos de que o módulo de Restaurante do Monólito avise ao serviço de Distância que novos restaurantes foram aprovados e que houve atualização no cep e/ou tipo de cozinha de restaurantes já cadastrados no BD de Distância.

Para isso, vamos expandir a API RESTful do serviço de Distância para que receba novos restaurantes aprovados e os insira no BD de Distância e atualize dados alterados de restaurantes existentes.

O serviço de Distância será o servidor e o módulo de Restaurante do Monólito terá o código do cliente REST. Para implementarmos esse cliente, usaremos a classe `RestTemplate` do Spring.



No `eats-distancia-service`, crie um Controller chamado `RestaurantesController` no pacote `br.com.caelum.eats.distancia` com um método que insere um novo restaurante e outro que atualiza um restaurante existente. Defina mensagens de log em cada método.

```
##### fj33-eats-distancia-
service/src/main/java;br/com/caelum/eats/distancia/RestaurantesController.java
```

```
@RestController
@AllArgsConstructor
```

```
@Slf4j
class RestaurantesController {

    private RestauranteRepository repo;

    @PostMapping("/restaurantes")
    ResponseEntity<Restaurante> adiciona(@RequestBody Restaurante restaurante) {
        log.info("Insere novo restaurante: " + restaurante);
        Restaurante salvo = repo.insert(restaurante);
        UriComponents uriComponents = uriBuilder.path("/restaurantes/{id}")
            .buildAndExpand(restaurante.getId());
        URI uri = uriComponents.toUri();
        return ResponseEntity.created(uri).contentType(MediaType.APPLICATION_JSON);
    }

    @PutMapping("/restaurantes/{id}")
    Restaurante atualiza(@PathVariable("id") Long id, @RequestBody Restaurante restaurante) {
        if (!repo.existsById(id)) {
            throw new ResourceNotFoundException();
        }
        log.info("Atualiza restaurante: " + restaurante);
        return repo.save(restaurante);
    }

}
```

Certifique-se que os imports estão corretos:

```
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.RestController;

import lombok.AllArgsConstructor;
import lombok.extern.slf4j.Slf4j;
```

No application.properties do módulo eats-application do monólito,

crie uma propriedade `configuracao.distancia.service.url` para indicar a URL do serviço de distância:

```
##### fj33-eats-monolito-modular/eats/eats-
application/src/main/resources/application.properties
```

```
configuracao.distancia.service.url=http://localhost:8082
```

No módulo `eats-application` do monólito, crie uma classe `RestClientConfig` no pacote `br.com.caelum.eats`, que fornece um `RestTemplate` do Spring:

```
##### fj33-eats-monolito-modular/eats/eats-
application/src/main/java;br/com/caelum/eats/RestClientConfig.java
```

```
@Configuration
class RestClientConfig {

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

}
```

Faça os imports adequados:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;
```

No módulo `eats-restaurante` do monólito, crie uma classe `RestauranteParaServicoDeDistancia` no pacote

`br.com.caelum.eats.restaurante` que contém apenas as informações adequadas para o serviço de distância. Crie um construtor que recebe um `Restaurante` e popula os dados necessários:

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java;br/com/caelum/eats/restaurante/RestaurantePa
raServicoDeDistancia.java
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
class RestauranteParaServicoDeDistancia {

    private Long id;
    private String cep;
    private Long tipoDeCozinhaId;

    RestauranteParaServicoDeDistancia(Restaurante restaurante){
        this(restaurante.getId(), restaurante.getCep(), restaurante
    }

}
```

Não esqueça de definir os imports:

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
```

Observação: a anotação `@Data` do Lombok define um Java Bean com getters, setters para campos mutáveis, `equals` e `hashCode` e `toString`.

Crie uma classe `DistanciaRestClient` no pacote `br.com.caelum.eats.restaurante` do módulo `eats-restaurante` do monólito. Defina como dependências um `RestTemplate` e uma `String`

para armazenar a propriedade `configuracao.distancia.service.url`.

Anote a classe com `@Service` do Spring.

Defina métodos que chamam o serviço de distância para:

- inserir um novo restaurante aprovado, enviando um POST para `/restaurantes` com `O RestauranteParaServicoDeDistancia` como corpo da requisição
- atualizar um restaurante já existente, enviando um PUT para `/restaurantes/{id}`, com o `id` adequado e um `RestauranteParaServicoDeDistancia` no corpo da requisição

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/DistanciaRest
Client.java
```

```
@Service
class DistanciaRestClient {

    private String distanciaServiceUrl;
    private RestTemplate restTemplate;

    DistanciaRestClient(RestTemplate restTemplate,
                        @Value("${configuracao.c
        this.distanciaServiceUrl = distanciaServiceUrl;
        this.restTemplate = restTemplate;
    }

    void novoRestauranteAprovado(Restaurante restaurante) {
        RestauranteParaServicoDeDistancia restauranteParaDistancia
        String url = distanciaServiceUrl+"/restaurantes";
        ResponseEntity<RestauranteParaServicoDeDistancia> response
            restTemplate.postForEntity(url, restauranteParaDistanc
        HttpStatus statusCode = responseEntity.getStatusCode();
        if (!HttpStatus.CREATED.equals(statusCode)) {
            throw new RuntimeException("Status diferente do esperad
        }
    }
}
```

```
    }

    void restauranteAtualizado(Restaurante restaurante) {
        RestauranteParaServicoDeDistancia restauranteParaDistancia =
            new RestauranteParaServicoDeDistancia();
        String url = distanciaServiceUrl + "/restaurantes/" + restaurante.getId();
        restTemplate.put(url, restauranteParaDistancia, Restaurante.class);
    }

}
```

Os imports corretos são:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;
```

Altere a classe `RestauranteController` do módulo `eats-restaurante` do monólito para que:

- tenha um `DistanciaRestClient` como dependência
- no caso de aprovação de um restaurante, invoque o método `novoRestauranteAprovado` de `DistanciaRestClient`
- no caso de atualização do CEP ou tipo de cozinha de um restaurante já aprovado, invoque o método `restauranteAtualizado` de `DistanciaRestClient`

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/RestauranteC
ontroller.java
```

```
// anotações ...
class RestauranteController {

    private RestauranteRepository restauranteRepo;
    private CardapioRepository cardapioRepo;
```

```
private DistanciaRestClient distanciaRestClient; // adiciona  
// métodos omitidos ...  
  
@PutMapping("/parceiros/restaurantes/{id}")  
Restaurante atualiza(@RequestBody Restaurante restaurante) {  
    Restaurante doBD = restauranteRepo.getOne(restaurante.getId());  
    restaurante.setUser(doBD.getUser());  
    restaurante.setAprovado(doBD.getAprovado());  
  
    Restaurante salvo = restauranteRepo.save(restaurante);  
  
    if (restaurante.getAprovado() &&  
        (cepDiferente(restaurante, doBD) || tipoDeCozinhaDiferente(restaurante, doBD))) {  
        distanciaRestClient.restauranteAtualizado(restaurante);  
    }  
  
    return salvo;  
}  
  
// método omitido ...  
  
@Transactional  
@PatchMapping("/admin/restaurantes/{id}")  
void aprova(@PathVariable("id") Long id) {  
    restauranteRepo.aprovaPorId(id);  
  
    // adicionado  
    Restaurante restaurante = restauranteRepo.getOne(id);  
    distanciaRestClient.novoRestauranteAprovado(restaurante);  
}  
  
private boolean tipoDeCozinhaDiferente(Restaurante restaurante, Restaurante doBD)  
    return !doBD.getTipoDeCozinha().getId().equals(restaurante.getId());  
}  
  
private boolean cepDiferente(Restaurante restaurante, Restaurante doBD)  
    return !doBD.getCep().equals(restaurante.getCep());
```

```
}
```

```
}
```

Observação: pensando em design de código, será que os métodos auxiliares `tipoDeCozinhaDiferente` e `cepDiferente` deveriam ficar em `RestauranteController` mesmo?

Exercício: Integrando o módulo de restaurantes ao serviço de distância com RestTemplate

1. Interrompa o monólito e o serviço de distância.

Em um terminal, vá até a branch `cap6-integracao-monolito-distancia-com-rest-template` dos projetos `fj33-eats-monolito-modular` e `fj33-eats-distancia-service`:

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap6-integracao-monolito-distancia-com-rest-te
cd ~/Desktop/fj33-eats-distancia-service
git checkout -f cap6-integracao-monolito-distancia-com-rest-te
```

Suba o monólito executando a classe `EatsApplication` e o serviço de distância por meio da classe `EatsDistanciaServiceApplication`.

2. Efetue login como um dono de restaurante.

O restaurante Long Fu, que já vem pré-cadastrado, tem o usuário `longfu` e a senha `123456`.

Faça uma mudança no tipo de cozinha ou CEP do restaurante.

Verifique nos logs que o restaurante foi atualizado no serviço de distância.

Se desejar, cadastre um novo restaurante. Então, faça login como Administrador do Caelum Eats: o usuário é `admin` e a senha é `123456`.

Aprove o novo restaurante. O serviço de distância deve ter sido chamado. Veja nos logs.

No diretório do `docker-compose.yml`, acesse o database de distância no MongoDB com o Mongo Shell:

```
cd ~/Desktop  
docker-compose exec mongo.distancia mongo eats_distancia
```

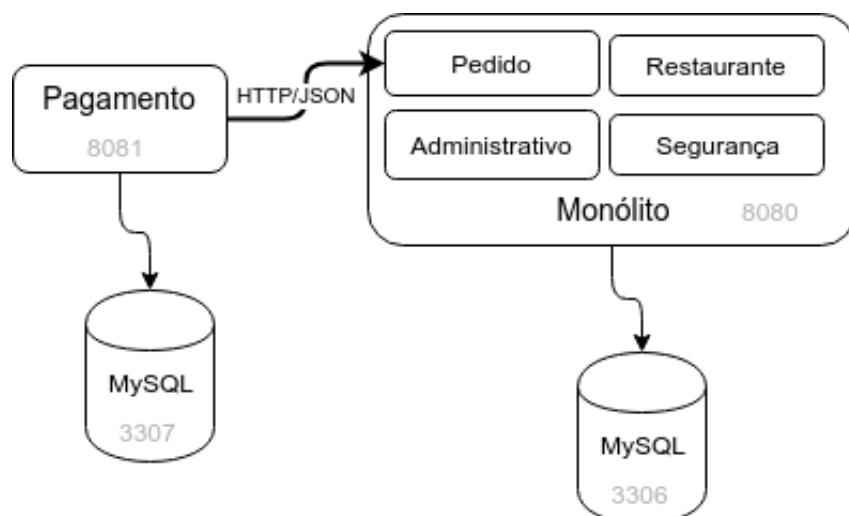
Então, veja o conteúdo da collection `restaurantes` com o comando:

Cliente REST declarativo com Feign

Depois de confirmar um pagamento, o status do pedido ainda permanece como *REALIZADO*. Precisamos implementar uma maneira do serviço de Pagamentos avisar que um determinado pedido foi pago.

Para isso, vamos acrescentar à API RESTful do módulo de Pedido do Monólito um recurso para notificar o pagamento de um pedido.

O módulo de Pedido será o servidor, enquanto o cliente REST será o serviço de Pagamentos. Faremos a implementação de maneira declarativa com o Feign.



Adicione ao PedidoController, do módulo eats-pedido do monólito, um método que muda o status do pedido para PAGO:

```
##### fj33-eats-monolito-modular/eats/eats-
pedido/src/main/java/br/com/caelum/eats/pedido/PedidoController.java
```

```
@PutMapping("/pedidos/{id}/pago")
void pago(@PathVariable("id") Long id) {
    Pedido pedido = repo.porIdComItens(id);
    if (pedido == null) {
        throw new ResourceNotFoundException();
    }
    pedido.setStatus(Pedido.Status.PAGO);
    repo.atualizaStatus(Pedido.Status.PAGO, pedido);
}
```

No arquivo application.properties de eats-pagamento-service, adicione uma propriedade configuracao.pedido.service.url que contém a URL do monólito:

```
##### fj33-eats-pagamento-
service/src/main/resources/application.properties
```

```
configuracao.pedido.service.url=http://localhost:8080
```

No pom.xml de eats-pagamento-service, adicione uma dependência ao Spring Cloud na versão Greenwich.SR2, em dependencyManagement:

```
##### fj33-eats-pagamento-service/pom.xml
```

```
<dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
```

```
<version>Greenwich.SR2</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

Feito isso, adicione o *starter* do OpenFeign como dependência:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Anote a classe `EatsPagamentoServiceApplication` com `@EnableFeignClients` para habilitar o Feign:

```
##### fj33-eats-pagamento-
service/src/main/java/br/com/caelum/eats/pagamento/EatsPagamentoSe
rviceApplication.java
```

```
@EnableFeignClients // adicionado
@SpringBootApplication
public class EatsPagamentoServiceApplication {

    // código omitido ...
}
```

O import correto é o seguinte:

```
import org.springframework.cloud.openfeign.EnableFeignClients;
```

Defina, no pacote br.com.caelum.eats.pagamento de eats-pagamento-service, uma interface PedidoRestClient com um método avisaQueFoiPago, anotados da seguinte maneira:

```
##### fj33-eats-pagamento-
service/src/main/java(br/com/caelum/eats/pagamento/PedidoRestClient.j
ava
```

```
@FeignClient(url="${configuracao.pedido.service.url}", name="r
interface PedidoRestClient {

    @PutMapping("/pedidos/{pedidoId}/pago")
    void avisaQueFoiPago(@PathVariable("pedidoId") Long pedidoIc
}
```

Ajuste os imports:

```
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PutMapping;
```

Em PagamentoController, do serviço de pagamento, defina um PedidoRestClient como atributo e use o método avisaQueFoiPago passando o id do pedido:

```
##### fj33-eats-pagamento-
service/src/main/java(br/com/caelum/eats/pagamento/PagamentoControl
ler.java
```

```
// anotações ...
public class PagamentoController {

    private PagamentoRepository pagamentoRepo;
```

```
private PedidoRestClient pedidoClient; // adicionado

// código omitido ...

@PutMapping("/{id}")
public PagamentoDto confirma(@PathVariable Long id) {
    Pagamento pagamento = pagamentoRepo.findById(id).orElseThrow();
    pagamento.setStatus(Pagamento.Status.CONFIRMADO);
    pagamentoRepo.save(pagamento);

    // adicionado
    Long pedidoId = pagamento.getPedidoId();
    pedidoClient.avisaQueFoiPago(pedidoId);

    return new PagamentoDto(pagamento);
}

// restante do código ...

}
```

Exercício: Integrando o serviço de pagamentos e o módulo de pedidos com Feign

1. Interrompa o monólito e o serviço de pagamentos.

Em um terminal, vá até a branch `cap6-integracao-pagamento-monolito-com-feign` dos projetos `fj33-eats-monolito-modular` e `fj33-eats-pagamento-service`:

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap6-integracao-pagamento-monolito-com-feign

cd ~/Desktop/fj33-eats-pagamento-service
git checkout -f cap6-integracao-pagamento-monolito-com-feign
```

Suba o monólito executando a classe `EatsApplication` e o serviço de pagamentos por meio da classe `EatsPagamentoServiceApplication`.

2. Certifique-se que o serviço de pagamentos foi reiniciado e que os demais serviços e o front-end estão no ar.

Faça um novo pedido, realizando e confirmado um pagamento.

Veja que, depois dessa mudança, o status do pedido fica como **PAGO** e não apenas como *REALIZADO*.

O poder dos Links

Ao descrevermos as boas ideias do procolo HTTP, mencionamos a importância dos links. Quando falamos sobre REST e sobre o Nível 3, o máximo, do Modelo de Maturidade de Leonard Richardson, falamos mais uma vez de links.

Roy Fielding, o criador do termo REST, diz no post [REST APIs must be hypertext-driven](#) (FIELDING, 2008) que toda API, para ser considerada RESTful, deve necessariamente usar links.

Hypertext? Hypermedia?

HyperText é um conceito tão importante para a Web que está nas iniciais de seu protocolo, o HTTP, e de seu formato de documentos original, o HTML.

O termo hypertext, foi cunhado por Ted Nelson e publicado no artigo [Complex Information Processing](#) (NELSON, 1965) para denotar um material escrito ou pictórico interconectado de maneira tão complexa que não pode ser representado convenientemente em papel. A ideia original é que seria algo que iria além do texto e deveria ser representado em uma tela interativa.

Nos comentários do post [REST APIs must be hypertext-driven](#) (FIELDING, 2008), Fielding dá a sua definição:

Quando eu digo hypertext, quero dizer a apresentação simultânea de informações e controles, de forma que as informações se tornem o meio pelo qual o usuário (ou autômato) obtém escolhas e seleciona ações.

A hypermedia é apenas uma expansão sobre o que o texto significa para incluir âncoras temporais em um fluxo de mídia; a maioria dos pesquisadores abandonou a distinção.

Hypertext não precisa ser HTML num navegador. Máquinas podem seguir links quando entendem o formato de dados e os tipos de relacionamento.

Monte a sua história seguindo links

No final da década de 1970 e começo da década de 1980, surgiu a série de livros-jogo "Escolha a sua aventura" (em inglês, *Choose Your Own Adventure*), lançada pela editora americana Bantam Books e editada pela Ediouro no Brasil.

Títulos como "A Gruta do Tempo" (em inglês, *The Cave of Time*), de Edward Packard, permitiam que o leitor fizesse escolhas e determinasse o rumo da história.

Na página 1 do livro, a história era contada linearmente, até que o personagem chegava a um ponto de decisão, em que eram oferecidas algumas opções para o leitor determinar o rumo da narrativa. Por exemplo:

- Se quiser seguir o velho camponês para ver aonde ele vai, vá para a página 3.
- Se preferir voltar para casa, vá para a página 15.
- Se quiser sentar e esperar, vá para a página 71.

O leitor poderia coletar itens no decorrer da história que seriam determinadas em trechos posteriores. Por exemplo:

- Se você tiver o item mágico "Olho do Falcão", vá para a página 210.

- Se você não tiver o item, mas possui a perícia "Rastreamento", vá para a página 19.
- Caso contrário, vá para a página 101.

Essa ideia de montar a sua história seguindo links parece muito com o uso de hypertext para APIs RESTful.

Em sua palestra [Getting Things Done with REST](#) (ROBINSON, 2011), Ian Robinson cita talvez o mais famoso desses livros-jogo, o de título "O Feiticeiro da Montanha de Fogo" (em inglês, *The Warlock of Firetop Mountain*), escrito por Ian Livingstone e Steve Jackson e publicado em 1982 pela Puffin Books.

Links para a transição de estados

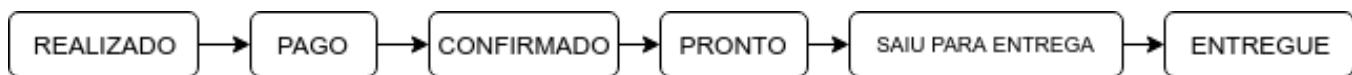
Algumas entidades de negócio mudam de estados no decorrer do seu uso em uma aplicação.

No Caelum, um restaurante é cadastrado e, para entrar nos resultados de buscas feitas pelos usuários, precisa ser aprovado pelo setor Administrativo. Podemos dizer que um restaurante começa com o estado CADASTRADO e então pode passar para o estado de APROVADO.

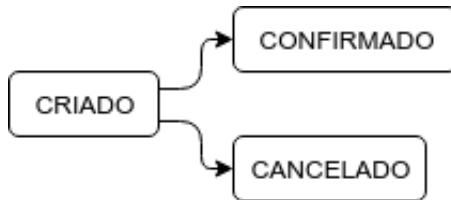


Para o restaurante, não há nenhum que indica os estados possíveis. Há apenas um atributo `aprovado` na classe `Restaurante` do módulo de Restaurante do Monólito.

Já para um pedido, há mais estados possíveis. Os valores desses estados estão representados na enum `status` da classe `Pedido` do módulo de Pedido do Monólito.



Um pagamento tem uma transição mais interessante: depois de `CRIADO`, pode ser `CONFIRMADO` ou `CANCELADO`.



Se observarmos a implementação da API de pagamentos do eats-pagamento-service,

- um `POST` em `/pagamentos` adiciona um novo pagamento com o estado `CRIADO` e retorna um código `201` com a URL do novo recurso no cabeçalho `Location`. Por exemplo, `/pagamentos/15`
- um `PUT` em `/pagamentos/15` confirma o pagamento de `id 15`, fazendo sua transição para o estado `CONFIRMADO`
- um `DELETE` em `/pagamentos/15` cancela o pagamento de `id 15`, deixando-o no estado `CANCELADO`

Ao invocar a URL de um pagamento com diferentes métodos HTTP, fazemos a transição dos estados de um pagamento.

Para um cliente HTTP fazer essa transição de estados, seu programador deve saber previamente quais os estados possíveis e quais URLs devem ser chamadas. No caso de uma mudança de URL, o programador teria que corrigir o código.

Poderíamos tornar o cliente HTTP mais flexível se representássemos as transições de estados possíveis por meio de links. Essa ideia é comumente chamada de **Hypermedia As The Engine Of Application State (HATEOAS)**.

Para a transição de estados de um pagamento, poderíamos ter um link de confirmação e um link de cancelamento.

Ainda teríamos que saber a utilidade de cada link. Para isso, temos o *link relation*, uma descrição definida em um atributo `rel` associado ao link.

Há um [padrão de link relations](#) mantido pela IANA. Alguns deles:

- `self`: um link para o próprio recurso
- `search`: um link para o um recurso de busca.
- `next`: um link para o próximo recurso de uma série. Comumente usado em paginação.
- `previous`: um link para o recurso anterior de uma série. Comumente usado em paginação.
- `first`: um link para o primeiro recurso de uma série. Comumente usado em paginação.
- `last`: um link para o último recurso de uma série. Comumente usado em paginação.

Podemos criar os nossos próprios link relations. Podemos associar o link de confirmação ao link relation `confirma` e o de cancelamento ao `cancela`.

Representando Links

Como representar esse links?

Em um XML, podemos usar o elemento `<link>`, que é usado em um HTML para incluir um CSS em uma página. Por exemplo, para o pagamento:

```
<pagamento>
  <id>1</id>
  <valor>51.8</valor>
  <nome>ANDERSON DA SILVA</nome>
  <numero>1111 2222 3333 4444</numero>
  <expiracao>2022-07</expiracao>
  <codigo>123</codigo>
  <status>CRIADO</status>
  <formaDePagamentoId>2</formaDePagamentoId>
  <pedidoId>1</pedidoId>
  <link rel="self" href="http://localhost:8081/pagamentos/1" />
  <link rel="confirma" href="http://localhost:8081/pagamentos/1">
  <link rel="cancela" href="http://localhost:8081/pagamentos/1">
```

```
</pagamento>
```

E para JSON? Poderíamos criar a nossa própria representação de links, mas já existe o HAL, ou *JSON Hypertext Application Language*, descrita por Mike Kelly na especificação preliminar (Internet-Draft) [draft-kelly-json-hal-00](#) (KELLY, 2012) da IETF. Apesar do status preliminar, a especificação é usada em diferentes tecnologias e frameworks. O media type associado ao HAL é `application/hal+json`. Em teoria, seria possível definir um HAL expressado numa representação XML.

No HAL, é adicionado ao JSON da aplicação uma propriedade `_links` que é um objeto contendo uma propriedade para cada link relation. Os links relations são definidos como objetos cujo link está na propriedade `href`. Por exemplo, para um pagamento:

```
{
  "id":1,
  "valor":51.80,
  "nome":"ANDERSON DA SILVA",
  "numero":"1111 2222 3333 4444",
  "expiracao":"2022-07",
  "codigo":"123",
  "status":"CRIADO",
  "formaDePagamentoId":2,
  "pedidoId":1,
  "_links":{
    "self":{
      "href":"http://localhost:8081/pagamentos/1"
    },
    "confirma":{
      "href":"http://localhost:8081/pagamentos/1"
    },
    "cancela":{
      "href":"http://localhost:8081/pagamentos/1"
    }
  }
}
```

A representação HAL de um recurso pode ter outros recursos embutidos, descritos na propriedade `_embedded`. Por exemplo, um pedido poderia ter uma lista de itens embutida, com representações e links para cada item.

Na [RFC 5988](#) (NOTTINGHAM, 2010) da IETF é definido um cabeçalho `Link` e uma série de link relations padronizados. A paginação da API do GitHub segue esse padrão:

```
Link: <https://api.github.com/repositories/237159/pulls?page=2>; rel="next"
```

A classe [Link](#), disponível a partir da especificação JAX-RS 2.0 do Java EE 7, usa o cabeçalho `Link` como formato de hypermedia.

Revisitando o Modelo de Maturidade de Richardson

Quando falamos sobre o Nível 3, o máximo, do Modelo de Maturidade de Leonard Richardson descrito por Martin Fowler no post [Richardson Maturity Model](#) (FOWLER, 2010), apenas mencionamos que links são importantes.

Agora, sabemos que links podem ser usados para descrever a transição de estados da aplicação, o que chamamos de HATEOAS.

Voltando ao exemplo de consultas médicas, cada horário da lista de horários disponíveis pode conter um link de agendamento:

```
GET /doutores/huberman/horarios?data=2010-01-04&status=disponivel HTTP/1.1
```

A resposta seria a mesma de antes:

```
HTTP/1.1 200 OK
```

```
<listaDeHorariosDisponiveis>
  <horario id="1234" doutor="huberman" inicio="14:00" fim="14:30">
    <link rel="agendamento" href="/horarios/1234">
  </horario>
  <horario id="5678" doutor="huberman" inicio="16:00" fim="16:30">
    <link rel="agendamento" href="/horarios/5678">
  </horario>
</listaDeHorariosDisponiveis>
```

Os links descrevem o que pode ser feito a seguir e as URLs que precisam ser manipuladas.

Seguindo o link de agendamento, o `POST` para marcar uma consulta seria feito da mesma maneira anterior:

```
POST /horarios/1234 HTTP/1.1
```

```
<agendamentoConsulta>
  <paciente id="alexandre"/>
</agendamentoConsulta>
```

O resultado poderia conter links para diferentes possibilidades:

```
HTTP/1.1 201 Created
Location: horarios/1234/consulta
```

```
<consulta>
  <horario id="1234" doutor="huberman" inicio="14:00" fim="14:30">
    <paciente id="alexandre"/>
    <link rel="agendamentoExame" href="/horarios/1234/consulta/exame">
    <link rel="cancelamento" href="/horarios/1234/consulta">
    <link rel="mudancaHorario" href="/doutor/huberman/horarios?c...
  </consulta>
```

Assim, as URLs podem ser modificadas sem que o código dos clientes quebre. Um benefício adicional é que os clientes podem conhecer e explorar os próximos passos.

Idealmente, um cliente deveria depender apenas da URL raiz de uma API e, a partir dela, navegar pelos links, descobrindo o que pode ser feito com a API.

Fowler menciona a ideia de Ian Robinson de que o Modelo de Maturidade de Richardson está relacionado com técnicas comuns de design:

- O Nível 1 trata de como lidar com a complexidade usando "dividir e conquistar", dividindo um grande endpoint para o serviço todo em vários recursos.
- O Nível 2 adiciona os métodos HTTP como um padrão, de maneira que possamos lidar com situações semelhantes da mesma maneira, evitando variações desnecessárias.
- O Nível 3 introduz a capacidade de descoberta (em inglês, *Discoverability*), fornecendo uma maneira de tornar o protocolo auto-documentado.

Exercício opcional: Spring HATEOAS e HAL

1. Adicione o Spring HATEOAS como dependência no `pom.xml` de `eats-pagamento-service`:

```
##### fj33-eats-pagamento-service/pom.xml
```

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-hateoas</artifactId>
</dependency>
```

2. Nos métodos de `PagamentoController`, retorne um `Resource` com uma lista de `Link` do Spring HATEOAS.

- Em todos os métodos, defina um *link relation* `self` que aponta para o próprio recurso, através da URL do método `detalha`
- Nos métodos `detalha` e `cria`, defina *link relations* `confirma` e `cancela`, apontando para as URLs associadas aos respectivos métodos de `PagamentoController`.

Para criar os *links*, utilize os métodos estáticos `methodOn` e `linkTo` de `ControllerLinkBuilder`.

O código de `PagamentoController` ficará semelhante a:

```
##### fj33-eats-pagamento-
service/src/main/java/br/com/caelum/eats/pagamento/PagamentoController.java
```

```
@RestController
@RequestMapping("/pagamentos")
@AllArgsConstructor
class PagamentoController {

    private PagamentoRepository pagamentoRepo;
    private PedidoRestClient pedidoClient;

    @GetMapping("/{id}")
    public Resource<PagamentoDto> detalha(@PathVariable Long id)
        Pagamento pagamento = pagamentoRepo.findById(id).orElseThr

        List<Link> links = new ArrayList<>();

        Link self = linkTo(methodOn(PagamentoController.class).det
        links.add(self);

        if (Pagamento.Status.CRIADO.equals(pagamento.getStatus()))
            Link confirma = linkTo(methodOn(PagamentoController.clas
        links.add(confirma);

        Link cancela = linkTo(methodOn(PagamentoController.class
        links.add(cancela);
```

```
}

PagamentoDto dto = new PagamentoDto(pagamento);
Resource<PagamentoDto> resource = new Resource<PagamentoDt

return resource;
}

@PostMapping
public ResponseEntity<Resource<PagamentoDto>> cria(@RequestE
    UriComponentsBuilder uriBuilder) {
    pagamento.setStatus(Pagamento.Status.CRIADO);
    Pagamento salvo = pagamentoRepo.save(pagamento);
    URI path = uriBuilder.path("/pagamentos/{id}").buildAndExp
    PagamentoDto dto = new PagamentoDto(salvo);

    Long id = salvo.getId();

    List<Link> links = new ArrayList<>();

    Link self = linkTo(methodOn(PagamentoController.class).det
    links.add(self);

    Link confirma = linkTo(methodOn(PagamentoController.class)
    links.add(confirma);

    Link cancela = linkTo(methodOn(PagamentoController.class).
    links.add(cancela);

    Resource<PagamentoDto> resource = new Resource<PagamentoDt
    return ResponseEntity.created(path).body(resource);
}

@PutMapping("/{id}")
public Resource<PagamentoDto> confirma(@PathVariable Long id
    Pagamento pagamento = pagamentoRepo.findById(id).orElseThr
    pagamento.setStatus(Pagamento.Status.CONFIRMADO);
    pagamentoRepo.save(pagamento);

    Long pedidoId = pagamento.getPedidoId();
```

```

pedidoClient.avisaQueFoiPago(pedidoId);

List<Link> links = new ArrayList<>();

Link self = linkTo(methodOn(PagamentoController.class).det
links.add(self);

PagamentoDto dto = new PagamentoDto(pagamento);
Resource<PagamentoDto> resource = new Resource<PagamentoDt

return resource;
}

@Override
public Resource<PagamentoDto> cancela(@PathVariable Long id)
    Pagamento pagamento = pagamentoRepo.findById(id).orElseThr
    pagamento.setStatus(Pagamento.Status.CANCELADO);
    pagamentoRepo.save(pagamento);

    List<Link> links = new ArrayList<>();

    Link self = linkTo(methodOn(PagamentoController.class).det
links.add(self);

    PagamentoDto dto = new PagamentoDto(pagamento);
    Resource<PagamentoDto> resource = new Resource<PagamentoDt

    return resource;
}

}

```

3. Reinicie o serviço de pagamentos e obtenha o pagamento de um `id` já cadastrado:

```
curl -i http://localhost:8081/pagamentos/1
```

A resposta será algo como:

```
HTTP/1.1 200
Content-Type: application/hal+json; charset=UTF-8
Transfer-Encoding: chunked
Date: Tue, 28 May 2019 19:04:43 GMT
```

```
{
  "id":1,
  "valor":51.80,
  "nome":"ANDERSON DA SILVA",
  "numero":"1111 2222 3333 4444",
  "expiracao":"2022-07",
  "codigo":"123",
  "status":"CRIADO",
  "formaDePagamentoId":2,
  "pedidoId":1,
  "_links": {
    "self": {
      "href": "http://localhost:8081/pagamentos/1"
    },
    "confirma": {
      "href": "http://localhost:8081/pagamentos/1"
    },
    "cancela": {
      "href": "http://localhost:8081/pagamentos/1"
    }
  }
}
```

Teste também a criação, confirmação e cancelamento de novos pagamentos.

4. Altere o código do front-end para usar os *link relations* apropriados ao confirmar ou cancelar um pagamento:

```
##### fj33-eats-ui/src/app/services/pagamento.service.ts
```

```

confirma(pagamento): Observable<any> {
  this._ajustaIds(-pagamento)-;
}

const url = pagamento._links.confirma.href; // adicionado

return -this._http.put(`-${this.API}/${pagamento.id}`);
return this.http.put(url, null); // modificado
}

cancela(pagamento): Observable<any> {
  this._ajustaIds(-pagamento)-;

  const url = pagamento._links.cancela.href; // adicionado

  return -this._http.delete(`-${this.API}/${pagamento.id}`);
  return this.http.delete(url); // modificado
}

```

Observação: o método auxiliar `ajustaIds` não é mais necessário ao confirmar e cancelar um pagamento, já que o `id` do pagamento não é mais usado para montar a URL. Porém, o método ainda é usado ao criar um pagamento.

5. Faça um novo pedido e efetue um pagamento. Deve continuar funcionando!

HATEOAS e Métodos HTTP

Observe o JSON com a representação de um pagamento retornado pelo serviço de Pagamentos:

```
{
  "id":1,
  "valor":51.80,
  ...
  "_links":{
    "self":{
```

```
        "href": "http://localhost:8081/pagamentos/1"
    },
    "confirma": {
        "href": "http://localhost:8081/pagamentos/1"
    },
    "cancela": {
        "href": "http://localhost:8081/pagamentos/1"
    }
}
```

Há links, cada um com seu link relation distinto: `self`, `confirma` e `cancela`.

Mas um detalhe importante é que todos os links são iguais! Tanto para confirmar como para cancelar o pagamento do JSON anterior, o link é:
<http://localhost:8081/pagamentos/1>

Além de saber sobre o significado de cada link relation (em outros termos, sua semântica), um cliente dessa API deve saber qual método HTTP utilizar para efetuar a confirmação ou cancelamento do pagamento.

Essa necessidade de um conhecimento prévio do cliente sobre o método HTTP a ser utilizado diminui a Discoverability da API.

Há [bastante discussão](#) sobre o fato de se o método HTTP deve ser associado a um link relation.

Uma visão mais purista diria que não devemos associar link relations a métodos HTTP.

Poderíamos utilizar uma chamada `OPTIONS` na URL do `href` do link relation e descobrir pelo cabeçalho `Allow` quais os métodos permitidos. Isso levaria a mais uma chamada pela rede entre o cliente e o servidor, impactando negativamente a Performance da aplicação.

Uma outra ideia é que poderíamos usar sempre o método `PUT`, passando

o status desejado (CONFIRMADO ou CANCELADO) no corpo da requisição.

Uma visão mais pragmática é usada na [API do PayPal](#), em que um atributo `method` é associado ao link relation:

```
{  
  "id": "8AA831015G517922L",  
  "status": "CREATED",  
  "links": [  
    {  
      "rel": "self",  
      "method": "GET",  
      "href": "https://api.paypal.com/v2/payments/authorizatio  
},  
    {  
      "rel": "capture",  
      "method": "POST",  
      "href": "https://api.paypal.com/v2/payments/authorizatio  
},  
    {  
      "rel": "void",  
      "method": "POST",  
      "href": "https://api.paypal.com/v2/payments/authorizatio  
},  
    {  
      "rel": "reauthorize",  
      "method": "POST",  
      "href": "https://api.paypal.com/v2/payments/authorizatio  
}  
]  
}
```

Note que a API do PayPal não usa HAL: os links ficam no atributo `links`, e não `_links`, que é um array, e não um objeto.

Podemos nos inspirar na API do PayPal e adicionar um atributo `method`

em cada link.

Exercício opcional: Estendendo o Spring HATEOAS

1. Crie uma classe `LinkWithMethod` que estende o `Link` do Spring HATEOAS e define um atributo adicional chamado `method`, que armazenará o método HTTP dos links. Defina um construtor que recebe um `Link` e uma `String` com o método HTTP:

```
##### fj33-eats-pagamento-
service/src/main/java/br/com/caelum/eats/pagamento/LinkWithMethod.java
```

```
@Getter
public class LinkWithMethod extends Link {

    private static final long serialVersionUID = 1L;

    private String method;

    public LinkWithMethod(Link link, String method) {
        super(link.getHref(), link.getRel());
        this.method = method;
    }
}
```

Os imports são os seguintes:

```
import org.springframework.hateoas.Link;
import lombok.Getter;
```

2. Na classe `PagamentoController`, adicione um `LinkWithMethod` na lista para os links de confirmação e cancelamento, passando o método HTTP adequado.

Use o trecho abaixo nos métodos `detalha` e `cria` de

`PagamentoController`:

```
##### fj33-eats-pagamento-
service/src/main/java/br/com/caelum/eats/pagamento/PagamentoController.java
```

```
Link confirma = linkTo(methodOn(PagamentoController.class).cor-
links.add(confirma);
```

```
links.add(new LinkWithMethod(confirma, "PUT")); // modificado
```

```
Link cancela = linkTo(methodOn(PagamentoController.class).canc-
links.add(cancela);
```

```
links.add(new LinkWithMethod(cancela, "DELETE")); // modificaç
```

3. Usando o cURL, obtenha novamente uma representação de um pagamento já cadastrado:

```
curl -i http://localhost:8081/pagamentos/1
```

Deve ser retornado algo parecido com:

```
HTTP/1.1 200
Content-Type: application/hal+json; charset=UTF-8
Transfer-Encoding: chunked
Date: Tue, 28 May 2019 19:04:43 GMT
```

```
{
  "id":1,
  "valor":51.80,
  "nome":"ANDERSON DA SILVA",
  "numero":"1111 2222 3333 4444",
  "expiracao":"2022-07",
  "codigo":"123",
```

```

    "status": "CRIADO",
    "formaDePagamentoId": 2,
    "pedidoId": 1,
    "_links": {
        "self": {
            "href": "http://localhost:8081/pagamentos/1"
        },
        "confirma": {
            "href": "http://localhost:8081/pagamentos/1",
            "method": "PUT"
        },
        "cancela": {
            "href": "http://localhost:8081/pagamentos/1",
            "method": "DELETE"
        }
    }
}

```

Observe os métodos HTTP na propriedade `method` dos *link relations* `confirma` e `cancela`.

4. Ajuste o código do front-end para usar o `method` de cada *link relation*:

```
##### fj33-eats-ui/src/app/services/pagamento.service.ts
```

```

confirma(pagamento): Observable<any> {
    const url = pagamento._links.confirma.href;

    return this.http.put(url, null);
}

const method = pagamento._links.confirma.method;
return this.http.request(method, url);
}

cancela(pagamento): Observable<any> {
    const url = pagamento._links.cancela.href;
}

```

```
return -this.-http.-delete(-url)-;-  
  
const method = pagamento._links.cancela.method;  
return this.http.request(method, url);  
}
```

-
4. (desafio) Modifique o PagamentoController para usar HAL-FORMS, disponível nas últimas versões do Spring HATEOAS.

Para saber mais: HAL-FORMS

[HAL-FORMS](#) (AMUNDSEN, 2016) é uma extensão do HAL especificada por Mike Amundsen que adiciona um atributo `_templates`, permitindo atribuir um método HTTP e outras propriedades a um link. O media type proposto é `application/prs.hal-forms+json`.

Um exemplo de response HAL-FORMS com os dados de uma pessoa:

```
{  
    "id" : 1,  
    "firstName" : "Frodo",  
    "lastName" : "Baggins",  
    "role" : "ring bearer",  
    "_links" : {  
        "self" : {  
            "href" : "http://localhost:8080/employees/1"  
        },  
        "employees" : {  
            "href" : "http://localhost:8080/employees"  
        }  
    },  
    "_templates" : {  
        "default" : {  
            "title" : null,  
            "method" : "put",  
            "contentType" : "",  
            "properties" : [ {
```

```

        "name" : "firstName",
        "required" : true
    }, {
        "name" : "id",
        "required" : true
    }, {
        "name" : "lastName",
        "required" : true
    }, {
        "name" : "role",
        "required" : true
    } ]
},
"deleteEmployee" : {
    "title" : null,
    "method" : "delete",
    "contentType" : "",
    "properties" : [ ]
}
}
}

```

O template `default` do HAL-FORMS presume que o recurso será editado por meio de um `PUT` na URL do link relation `self` e as propriedades associadas: `id`, `firstName`, `lastName` e `role`, todas obrigatórias nesse exemplo. Os dados desse template pode ser usados, por exemplo, para construir um `<form>` no front-end.

Já o template `deleteEmployee` tem associado o método `DELETE`, sem nenhuma propriedade.

O Spring HATEOAS na versão 1.0.0.RELEASE contém suporte a HAL-FORMS, que pode ser habilitado com a anotação `@EnableHypermediaSupport(type = HypermediaType.HAL_FORMS)`.

Além disso, há suporte a [Uniform Basis for Exchanging Representations \(UBER\)](#), [Collection+JSON](#) e [Application-Level Profile Semantics \(ALPS\)](#),

todos trabalhos experimentais de Mike Amundsen focados em hypermedia e que compõe a Affordance API do Spring HATEOAS.

Para saber mais: Spring Data REST

O Spring Data REST parte do Spring Data para expor entidades e repositórios como recursos REST, utilizando hypermedia e HAL como representação. Os mecanismos de persistência suportados são BDs relacionais com JPA, MongoDB, Neo4j e Gemfire.

Para utilizá-lo, basta incluir o starter no `pom.xml` da aplicação:

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

Se utilizar MySQL como BD, também precisamos incluir o Spring Data JPA e o driver do MySQL. Além disso, são necessárias as configurações de data sources no `application.properties`.

Vamos definir, como exemplo, uma entidade `Pessoa`:

```
@Entity
@Data
public class Pessoa {

    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String sobrenome;

}
```

O `@Data` do Lombok provê getters e setters para cada atributo, além de

implementações para equals, hashCode e toString.

Também deve ser definido um repository:

```
public interface PessoaRepository extends JpaRepository<Pessoa> {  
    List<Pessoa> findBySobrenome(@Param("sobrenome") String sobrenome);  
}
```

E pronto! Ao subirmos a aplicação já temos uma API RESTful com hypermedia!

A raiz da API provê links para as entidades expostas:

```
GET http://localhost:8080
```

A resposta é:

```
200 OK
```

```
Content-Type: application/hal+json
```

```
{  
    "_links" : {  
        "pessoas" : {  
            "href" : "http://localhost:8080/pessoas{?page,size,sort}"  
            "templated" : true  
        },  
        "profile" : {  
            "href" : "http://localhost:8080/profile"  
        }  
    }  
}
```

Note que o response é um HAL com o link relation pessoas.

O link relation profile está associada a especificação ALPS, que provê uma maneira de descrever os links, classificando-os em safe, idempotente, entre outros.

O recurso pessoas já é paginado, já que um JpaRepository extende a interface PagingAndSortingRepository do Spring Data Core.

Podemos criar uma nova pessoa fazendo o seguinte request ao recurso pessoas:

```
POST http://localhost:8080/pessoas
Content-Type: application/json
```

Com o payload:

```
{
  "nome": "Alexandre",
  "sobrenome": "Aquiles"
}
```

Como response, teremos:

```
201 Created
Content-Type: application/json
Location: http://localhost:8080/pessoas/3
```

```
{
  "nome": "Alexandre",
  "sobrenome": "Aquiles",
  "_links": {
    "self": {
      "href": "http://localhost:8080/pessoas/3"
```

```
},
"pessoa": {
    "href": "http://localhost:8080/pessoas/3"
}
}
```

Perceba que é retornado o status 201 com a URL do novo recurso no cabeçalho `Location`.

No corpo do response, são retornados os dados do recurso criado junto aos links.

Se dispararmos um `GET` ao `href` do link relation `self`, a URL `http://localhost:8080/pessoas/3`, teremos um payload semelhante ao anterior.

Podemos editar um recurso com um `PUT`, que sobreescrava os dados do recurso com a representação passada no request. Os atributos omitidos ficarão como nulos.

Se quisermos passar apenas um subconjunto dos dados, podemos usar um `PATCH`.

Para remover um recurso, podemos usar um `DELETE`.

Para listarmos todas as pessoas, podemos consultar o recurso do link relation `pessoas` da raiz da API:

```
GET http://localhost:8080/pessoas
```

O response será paginado, com

```
{
  "_embedded" : {
```

```
"pessoas" : [ {
    "nome" : "Anderson",
    "sobrenome" : "da Silva",
    "_links" : {
        "self" : {
            "href" : "http://localhost:8080/pessoas/2"
        },
        "pessoa" : {
            "href" : "http://localhost:8080/pessoas/2"
        }
    }
}, {
    "nome" : "Alexandre",
    "sobrenome" : "Aquiles",
    "_links" : {
        "self" : {
            "href" : "http://localhost:8080/pessoas/3"
        },
        "pessoa" : {
            "href" : "http://localhost:8080/pessoas/3"
        }
    }
}
],
"_links" : {
    "self" : {
        "href" : "http://localhost:8080/pessoas{?page,size,sort}"
        "templated" : true
    },
    "profile" : {
        "href" : "http://localhost:8080/profile/pessoas"
    },
    "search" : {
        "href" : "http://localhost:8080/pessoas/search"
    }
},
"page" : {
    "size" : 20,
    "totalElements" : 2,
    "totalPages" : 1,
```

```
        "number" : 0
    }
}
```

Os itens da lista ficam no atributo `_embedded`. Cada item contém seus dados e links.

Há dados de paginação: o `size` indica o tamanho máximo de elementos de uma página, 20 é o padrão mas pode ser alterado com o parâmetro `size`; `number` indica o número da página atual; `totalPages`, o número de páginas; e `totalElements`, o número total de elementos cadastrados.

Há os links da própria lista. Se houver mais de uma página, teremos os link relations `next` e `last`.

Ao seguirmos o link relation `search`, são exibidas as consultas possíveis:

```
GET http://localhost:8080/pessoas/search
```

```
{
  "_links" : {
    "findBySobrenome" : {
      "href" : "http://localhost:8080/pessoas/search/findBySobrenome",
      "templated" : true
    },
    "self" : {
      "href" : "http://localhost:8080/pessoas/search"
    }
  }
}
```

Podemos usar o link relation `findBySobrenome`, para buscar todas as pessoas com sobrenome Aquiles:

```
GET http://localhost:8080/pessoas/search/findBySobrenome?sobrenome=Aquiles
```

Obteremos no response:

```
{
  "_embedded" : {
    "pessoas" : [ {
      "nome" : "Alexandre",
      "sobrenome" : "Aquiles",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/pessoas/3"
        },
        "pessoa" : {
          "href" : "http://localhost:8080/pessoas/3"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/pessoas/search/findBySobrenome"
    }
  }
}
```

Formatos e Protocolos Binários

Na palestra [PB vs. Thrift vs. Avro](#) (ANISHCHENKO, 2012), Igor Anishchenko demonstra que o protocolo HTTP/1.1 com representações como XML e JSON pode ser ineficiente se comparado com alternativas binárias como:

- Apache Thrift, usado no Facebook e em projetos com o Hadoop
- Protocol Buffers, usado na Google

- RMI, que é parte da plataforma Java

Anishchenko, ao serializar um objeto Curso com 5 objetos Pessoa e com um objeto Telefone associados, mostra o tamanho:

- Protocol Buffers: 250
- Thrift TCompactProtocol: 278
- Thrift TBinaryProtocol: 460
- HTTP/JSON: 559
- HTTP/XML: 836
- RMI: 905

O protocolo Thrift TBinaryProtocol é otimizado em termos de processamento e o Thrift TCompactProtocol, em tamanho.

É interessante notar que a serialização RMI foi a menos eficiente em termos de tamanho. Não é um formato otimizado.

Os formatos binários Thrift TCompactProtocol e Protocol Buffers apresentam um tamanho de cerca de metade do HTTP/JSON para esse caso simples.

Então, Anishchenko compara 10000 chamadas de uma busca por uma listagem de códigos de Curso e, em seguida, os dados do Curso associado a esse código. São avaliados o tempo de resposta, porcentagem de uso de CPU no servidor e no cliente.

Os resultados para o tempo de resposta:

- Thrift TCompactProtocol: 01 min 05 s
- Thrift TBinaryProtocol: 01 min 13 s
- Protocol Buffers: 01 min 19 s
- RMI: 02 min 14 s
- HTTP/JSON: 04 min 44 s
- HTTP/XML: 05 min 27 s

Os resultados para porcentagem de uso de CPU do servidor:

- Thrift TBinaryProtocol: 33 %
- Thrift TCompactProtocol: 30 %
- Protocol Buffers: 30 %
- HTTP/JSON: 20 %
- RMI: 16 %
- HTTP/XML: 12 %

Os resultados para CPU do cliente:

- Thrift TCompactProtocol: 22.5 %
- Thrift TBinaryProtocol: 21 %
- Protocol Buffers: 37.75 %
- RMI: 46.5 %
- HTTP/JSON: 75 %
- HTTP/XML: 80.75 %

Pelos dados de Anishchenko, Thrift toma mais processamento do servidor, suavizando o processamento no cliente.

É preciso tomar cuidado com microbenchmarks desse tipo. Os resultados podem ser enganosos. Meça você mesmo!

Uma diferença importante entre o Thrift e Protocol Buffers é que, apesar de ambos oferecerem maneira de definir interfaces de serviços, apenas o Thrift fornece implementações de clientes e servidores. Ao usar Protocol Buffers, é necessário implementar manualmente o servidor e o cliente. Recentemente, o Google abriu o código de um projeto que já fornece essas implementações, que veremos logo adiante.

Outra alternativa mencionada por Anishchenko é o Apache Avro, usado pelo Apache Kafka, entre outros projetos.

Quando usar protocolos binários?

Grandes empresas como Facebook, Google, Twitter e Linkedin expõe APIs RESTful para uso externo. Dentro de seus datacenters, porém, usam protocolos binários para aumentar a eficiência na comunicação

entre serviços.

Em alguns cenários de aplicação Mobile, um formato de serialização mais compacto e com menos processamento no cliente pode ser interessante já que há limitações de CPU, bateria e banda de rede.

HTTP/2

Nada impede que formatos binários de serialização de dados sejam usados com um transporte HTTP. Mas o protocolo HTTP/1.1 em si é ineficiente por ser baseado em texto, com diversos cabeçalhos e uma conexão TCP a cada request/response.

Ilya Grigorik, no livro [High Performance Browser Networking](#) (GRIGORIK, 2013), descreve o trabalho de desenvolvedores do Google em um protocolo experimental com o objetivo de reduzir em 50% o tempo de carregamento de páginas. O resultado desse experimento foi o protocolo SPDY, que foi progressivamente adotado por diferentes empresas.

Grigorik relata que, em 2012, o HTTP Working Group da IETF começou a trabalhar num draft inspirado no SPDY. Em 2015, foi publicada a [RFC 7540](#) (BELSHE et al., 2015), que especifica o HTTP/2.

O protocolo HTTP/2 aproveita melhor as conexões TCP sobre as quais é construído, codificando e comprimindo os dados em *frames* binários, que contém os cabeçalhos separados dos dados. Há ainda a possibilidade de *streams* múltiplas, iniciadas pelo cliente ou pelo servidor.

Grigorik descreve a terminologia do HTTP/2:

- Stream: um fluxo bidirecional de bytes em uma mesma conexão, que pode transportar uma ou mais mensagens.
- Mensagem: uma sequência completa de frames que equivalem a um request ou response.
- Frame: a menor unidade de comunicação do HTTP/2, transporta um tipo específico de dados, como cabeçalhos HTTP, o payload, etc.

Cada frame tem no mínimo um cabeçalho que identifica a qual stream pertence.

No HTTP/1.x, era possível realizar múltiplos requests paralelos, cada um em uma conexão TCP diferente, mas apenas um response poderia ser entregue por vez. Com o HTTP/2, é possível, na mesma conexão TCP, obter frames independentes intercalados que são remontados do lado de quem recebe os dados, no cliente ou no servidor. Isso é chamado de *multiplexing*.

Um servidor HTTP/2 pode fazer um *server push*, enviando múltiplos frames em resposta a um mesmo request do cliente. Dessa maneira, como resposta a um request de um HTML, o servidor poderia enviar, além do HTML, todos os JS e CSS associados em diferentes frames. Assim, não há a necessidade de concatenação, como havia no HTTP/1.x.

No livro, Grigorik diz que, apesar do HTTP/2 não requerer o uso de conexões seguras, TLS é o mecanismo mais confiável e eficiente para HTTP/2, eliminando a necessidade de latência ou roundtrips extras. Além disso, os navegadores adicionam a restrição de só habilitar HTTP/2 sobre uma conexão TLS.

Os recursos, URLs, métodos, cabeçalhos e códigos de status são mantidos no HTTP/2. Porém, como há uma nova codificação binária, tanto o servidor como o cliente precisam ser compatíveis com o HTTP/2.

gRPC

Anteriormente, discutimos como o uso de formatos binários de serialização dos dados podem aumentar a eficiência na comunicação entre serviços, ou até mesmo entre um navegador e um servidor Web. Uma característica do Protocol Buffers, um formato binário definido pela Google, é há uma maneira de definir uma interface para um serviço mas não são fornecidas implementações de clientes e servidores.

No post [gRPC: a true internet-scale RPC framework](#) (TALWAR, 2016),

Varun Talwar revela que a Google usou por 15 anos um projeto chamado Stubby, uma framework RPC que fornecia essa implementação de clientes e servidores, usando Protocol Buffers como formato de dados e lidando com dezenas de bilhões de requests por segundo.

Como mencionamos anteriormente, RPC (Remote Procedure Call) é uma forma de integrar Sistemas Distribuídos que expõe as operações de uma aplicação. Exemplos de mecanismos RPC são Web Services CORBA, DCOM, RMI, SOAP e Thrift.

Uma API RESTful não seria exatamente RPC, apesar da comunicação síncrona, porque é *Resource-Oriented*.

Em um framework RPC, o cliente invoca o servidor como se fosse uma chamada local. O framework lida com as complexidades de serialização de dados, comunicação pela rede, entre outras preocupações.

Em 2015, a Google lançou o projeto open-source **gRPC**, uma evolução do Stubby que provê um framework para comunicação RPC que usa Protocol Buffers como formato padrão de dados. Como protocolo para transmissão de dados, o gRPC é implementado sobre HTTP/2, aproveitando os frames binários e streams. Podem ser gerados servidores e clientes em linguagens como C++, C#, Java, Go, PHP, Python, Ruby, JS/Node, entre outras.

IDL com Protocol Buffers

Em uma solução RPC, o servidor precisa expor quais as operações podem ser chamadas e qual o modelo de dados das requisições e das respostas. No RMI, essa definição é feita em uma interface Java. Em um WebService SOAP, as operações são definidas em um WSDL. No antigo CORBA, era utilizado um IDL (Interface Definition Language).

No gRPC, o IDL é definido com Protocol Buffers.

A estrutura de serialização dos dados é definida em um arquivo com a

extensão .proto.

Nesse arquivo, definimos um ou mais `Message`, cada contendo um ou mais campos tipados. Entre os tipos possíveis são: `double`, `float`, `int32` (um `int` no Java), `int64` (um `long` no Java), `bool`, `string` e `bytes`, entre outros. É possível definir um `enum`. Um campo pode ser singular, o padrão, ou `repeated`, indicando uma lista ordenada com zero ou mais elementos. Cada campo deve ter um número único, que é usado para identificar o campo na serialização binária.

Também é possível definir um ou mais `service`, que definem chamadas remotas com as `Message` de `request` e de `response`. É possível definir alguns tipos de métodos no `service`:

- RPC simples, como uma chamada de função normal, em que o cliente envia um `request` para o servidor e espera um `response`
- *Server-side streaming RPC*, em que o cliente envia um `request` e o servidor responde com um fluxo de dados
- *Client-side streaming RPC*, em que o cliente enviar um fluxo de dados e o servidor responde com uma `Message` simples
- *Bidirectional streaming RPC*, em que tanto o cliente e o servidor trabalham com fluxos de dados

Por exemplo, para termos algo semelhante à busca dos restaurantes mais próximos do serviço de Distância, poderíamos criar o seguinte arquivo `distancia.proto`, com um RPC simples:

```
syntax = "proto3";  
  
package distancia; // não deve ser reverse domain name  
  
option java_package = "br.com.caelum.eats.distancia"; // pacot  
  
service Distancia {  
  
    rpc MaisProximos (MaisProximosRequest) returns (MaisProximos
```

```
}

message MaisProximosRequest {
    string cep = 1;
}

message MaisProximosResponse {
    repeated RestauranteComDistancia restaurantes = 1;
}

message RestauranteComDistancia {
    int64 restauranteId = 1;
    double distancia = 2;
}
```

A partir do arquivo `.proto`, podem ser geradas classes (ou o equivalente da linguagem) com o compilador do Protocol Buffers:

```
protoc -I=src/main/proto --java_out=target/generated-sources/r
```

A opção `-I` ou `--proto_path` especifica o diretório que contém arquivos `.proto`.

A opção `--java_out` indica o diretório raiz onde devem ser colocados os `.java` gerados. No caso do arquivo `distancia.proto`, com a opção `target` no `--java_out`, os arquivos seriam gerados no diretório `target/br/com/caelum/eats/distancia`.

Existem análogos em outras linguagens como `--cpp_out` para C++, `--go_out` para Go, `--python_out` para Python, entre outras opções.

O último argumento do comando `protoc` é o caminho a um ou mais arquivos `.proto`.

O comando anterior criaria, no diretório `target/generated-sources/protobuf/java/`, uma classe `DistanciaOuterClass.java` que contém os seguintes outros tipos:

- a interface `MaisProximosRequestOrBuilder`
- a classe `MaisProximosRequest`
- a classe `MaisProximosRequest.Builder`
- a interface `MaisProximosResponseOrBuilder`
- a classe `MaisProximosResponse`
- a classe `MaisProximosResponse.Builder`
- a interface `RestauranteComDistanciaOrBuilder`
- a classe `RestauranteComDistancia`
- a classe `RestauranteComDistancia.Builder`

A partir dessas classes é possível criar instâncias de `MaisProximosRequest` e `MaisProximosRequest` e serializá-las para o formato Protocol Buffers.

Para que a execução do compilador do Protocol Buffers faça parte do build, podemos usar plugins para Maven e Gradle.

No Maven, deve ser adicionada uma dependência ao `protobuf-java` no `pom.xml`:

```
<dependency>
  <groupId>com.google.protobuf</groupId>
  <artifactId>protobuf-java</artifactId>
  <version>3.10.0</version>
</dependency>
```

O arquivo `distancia.proto`, com o código anterior, deve ser definido no diretório `src/main/proto`.

Deve ser definida a seguinte `extension`:

```
<extensions>
  <extension>
    <groupId>kr.motd.maven</groupId>
    <artifactId>os-maven-plugin</artifactId>
    <version>1.6.2</version>
  </extension>
</extensions>
```

Então, deve ser definido o seguinte `plugin`:

```
<plugin>
  <groupId>org.xolstice.maven.plugins</groupId>
  <artifactId>protobuf-maven-plugin</artifactId>
  <version>0.6.1</version>
  <configuration>
    <protocArtifact>com.google.protobuf:protoc:3.10.0:exe:${os.name}
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Quando o comando `mvn clean install` for executado, as classes mencionadas anteriormente serão geradas no diretório `target/generated-sources/protobuf/java/` e compiladas no diretório

target/classes.

Gerando servidores e clientes com gRPC

Para gerar classes base para o servidor e stubs para os cliente com gRPC, devemos adicionar as seguintes dependências ao `pom.xml`:

```
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty-shaded</artifactId>
    <version>1.24.0</version>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-protobuf</artifactId>
    <version>1.24.0</version>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-stub</artifactId>
    <version>1.24.0</version>
</dependency>
```

Podemos remover a dependência à biblioteca `protobuf-java`, já que essa é uma dependência transitiva de `grpc-protobuf`.

Devemos, também, adicionar o plugin do gRPC para o compilador do Protocol Buffers.

Para isso, devem ser adicionadas as seguintes configurações ao

`<configuration> do protobuf-maven-plugin`

```
<pluginId>grpc-java</pluginId>
<pluginArtifact>io.grpc:protoc-gen-grpc-java:1.24.0:exe:${os.c
```

Também deve ser adicionado o seguinte <goal>:

```
<goal>compile-custom</goal>
```

Ao executarmos `mvn clean install`, além da classe `DistanciaOuterClass.java` criada anteriormente, seria criada a classe `DistanciaGrpc`, no diretório `target/generated-sources/protobuf/grpc-java/`, contendo internamente os seguintes outros tipos:

- a interface `DistanciaImplBase`, cujos métodos devem ser estendidos para criar o servidor
- a classe `DistanciaBlockingStub`, para criar um cliente síncrono, cujas chamadas bloqueiam a thread esperando o resultado do servidor
- as classes `DistanciaStub` e `DistanciaFutureStub`, para criar clientes assíncronos, cujas chamadas são não-bloqueantes
- as classes de uso interno `MethodHandlers`,
`DistanciaBaseDescriptorSupplier`,
`DistanciaFileDescriptorSupplier` e
`DistanciaMethodDescriptorSupplier`

A implementação do servidor gRPC poderia ser algo semelhante a:

```
public class DistanciaGrpcService extends DistanciaGrpc.DistanciaGrpcBase {  
  
    @Override  
    public void maisProximos(MaisProximosRequest request, StreamObserver<MaisProximosResponse> responseObserver) {  
        // obtém CEP do request gRPC  
        String cep = request.getCep();  
  
        List<RestauranteComDistanciaDto> maisProximos = // obtém 100 restaurantes mais próximos  
        // monta lista de restaurantes para gRPC  
        List<RestauranteComDistancia> restaurantesParaGrpc = new ArrayList<>();  
        for (RestauranteComDistanciaDto maisProximo : maisProximos) {  
            RestaurantesParaGrpcBuilder builder = RestaurantesParaGrpc.newBuilder();  
            builder.setNome(maisProximo.getNome());  
            builder.setCep(maisProximo.getCep());  
            builder.setLatitude(maisProximo.getLatitude());  
            builder.setLongitude(maisProximo.getLongitude());  
            builder.setDistancia(maisProximo.getDistancia());  
            restaurantesParaGrpc.add(builder.build());  
        }  
        responseObserver.onNext(RestaurantesParaGrpc.build(restaurantesParaGrpc));  
        responseObserver.onCompleted();  
    }  
}
```

```

        RestauranteComDistancia restauranteParaGrpc = Restaurant
                .newBuilder()
                .setDistancia(maisProximo.getDis
                .setRestauranteId(maisProximo.g
                .build();
        restaurantesParaGrpc.add(restauranteParaGrpc);
    }

    // monta response gRPC
    MaisProximosResponse maisProximosResponse = MaisProximosRe
            .newBuilder()
            .addAllRestaurantes(restaurantesPa
            .build();

    responseObserver.onNext(maisProximosResponse);
    responseObserver.onCompleted();
}

}

```

Os imports corretos seriam os seguintes:

```

import br.com.caelum.eats.distancia.DistanciaOuterClass.MaisPr
import br.com.caelum.eats.distancia.DistanciaOuterClass.MaisPr
import br.com.caelum.eats.distancia.DistanciaOuterClass.Restau
import io.grpc.stub.StreamObserver;

```

Para subir um servidor na porta 6565, deveria ser implementada a seguinte classe:

```

public class ServidorGrpc {

    public static void main(String[] args) throws IOException, I
        Server server = ServerBuilder.forPort(6565)
                .addService(new DistanciaGrpcService())

```

```
        .build()
        .start();
    server.awaitTermination();
    Runtime.getRuntime().addShutdownHook(new Thread() {
        @Override
        public void run() {
            server.shutdown();
        }
    });
}

}
```

Os imports seriam:

```
import io.grpc.Server;
import io.grpc.ServerBuilder;
```

A implementação do cliente seria algo semelhante a:

```
public class DistanciaGrpcClient {

    public static void main(String[] args) {
        ManagedChannel channel = ManagedChannelBuilder.forAddress(
            "localhost", 50051)
            .usePlaintext() // desabilita TLS (apenas para testes)
            .build();
        DistanciaBlockingStub distanciaStub = DistanciaGrpc.newBlockingStub(channel);

        MaisProximosRequest request = MaisProximosRequest
            .newBuilder()
            .setCep("71500-100")
            .build();

        MaisProximosResponse response = distanciaStub.maisProximos(request);

        List<RestauranteComDistancia> restaurantesComDistancia = response.getRestaurantes();
    }
}
```

```
// Usa lista de restaurantes com distância...
}

}
```

A lista de imports:

```
import br.com.caelum.eats.distancia.DistanciaGrpc.DistanciaBla
import br.com.caelum.eats.distancia.DistanciaOuterClass.MaisPr
import br.com.caelum.eats.distancia.DistanciaOuterClass.MaisPr
import br.com.caelum.eats.distancia.DistanciaOuterClass.Restau
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
```

Integrando gRPC ao Spring Boot

Ainda não há suporte oficial ao gRPC no Spring Boot, mas a comunidade criou uma série de *starters* para facilitar a integração entre um projeto Spring Boot e o gRPC.

Um desses starters é da empresa LogNet. Para usá-lo basta adicionar, ao `pom.xml`, a seguinte dependência:

```
<dependency>
  <groupId>io.github.lognet</groupId>
  <artifactId>grpc-spring-boot-starter</artifactId>
  <version>3.4.3</version>
</dependency>
```

Podemos remover as dependências às bibliotecas `grpc-protobuf`, `grpc-stub` e `grpc-netty-shaded`, que já são definidas pelo starter da LogNet.

Então, não há a necessidade de subir manualmente um servidor, como fizemos na classe `ServidorGrpc`.

Basta anotar a implementação da classe base com `@GRpcService`:

```
@GRpcService // adicionado  
public class DistanciaGrpcService extends DistanciaGrpc.Distar  
  
// código omitido...  
  
}
```

O import correto é:

```
import org.lognet.springboot.grpc.GRpcService;
```

O restante da implementação é exatamente o mesmo. A diferença é que é possível injetar dependências gerenciadas pelo Spring, como services e repositories.

É iniciado um servidor na porta 6565. Podemos alterar essa porta com a propriedade `grpc.port` no `application.properties`.

Há ainda integração com outras bibliotecas do ecossistema Spring Boot e Spring Cloud, que podem ser estudadas na documentação do starter:
<https://github.com/LogNet/grpc-spring-boot-starter>

Exercício opcional: implementando um serviço de Recomendações com gRPC

Objetivo

Vamos implementar um serviço de Recomendações que recebe uma lista de ids de restaurantes e retorna uma outra lista com os elementos reordenados de acordo com uma suposta recomendação.

Por enquanto, faremos uma implementação fajuta: apenas vamos

embaralhar a lista original.

Implementaremos o serviço de Recomendações usando gRPC.

O serviço de Distância deve chamar o serviço de Recomendações logo depois de recuperar a lista de restaurantes mais próximos.

Passo a passo

1. Primeiramente, vamos criar uma definição Protocol Buffers para o serviço. Para isso, criaremos um novo projeto que conterá somente o IDL e as classes geradas. Esse projeto será compartilhado tanto pelo novo serviço de Recomendações como pelo serviço de Distância.

Crie um novo projeto Maven chamado `fj33-recomendacoes-idl` no Desktop:

Defina, no `pom.xml`, o seguinte conteúdo:

```
##### fj33-recomendacoes-idl/pom.xml
```

```
<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://
<modelVersion>4.0.0</modelVersion>

<groupId>br.com.caelum</groupId>
<artifactId>recomendacoes-idl</artifactId>
<version>0.0.1-SNAPSHOT</version>

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

<dependencies>

<dependency>
```

```
<groupId>io.grpc</groupId>
<artifactId>grpc-netty-shaded</artifactId>
<version>1.24.0</version>
</dependency>
<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-protobuf</artifactId>
  <version>1.24.0</version>
</dependency>
<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-stub</artifactId>
  <version>1.24.0</version>
</dependency>

</dependencies>

<build>
  <extensions>
    <extension>
      <groupId>kr.motd.maven</groupId>
      <artifactId>os-maven-plugin</artifactId>
      <version>1.6.2</version>
    </extension>
  </extensions>
  <plugins>
    <plugin>
      <groupId>org.xolstice.maven.plugins</groupId>
      <artifactId>protobuf-maven-plugin</artifactId>
      <version>0.6.1</version>
      <configuration>
        <protocArtifact>com.google.protobuf:protoc:3.10.0:exe
        <pluginId>grpc-java</pluginId>
        <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.24.0:
      </configuration>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>compile-custom</goal>
```

```
    </goals>
  </execution>
</executions>
</plugin>
</plugins>
</build>

</project>
```



Crie um diretório `src/main/proto` e, dentro dele, um arquivo `recomendacoes.proto` com o seguinte conteúdo:

```
##### fj33-recomendacoes-idl/src/main/proto/recomendacoes.proto

syntax = "proto3";

package recomendacoes;

option java_package = "br.com.caelum.eats.recomendacoes.grpc";

message Restaurantes {
  repeated int64 restauranteId = 1;
}

service RecomendacoesDeRestaurantes {
  rpc Recomendacoes(Restaurantes) returns (Restaurantes) {}
}
```



O arquivo anterior define um serviço `RecomendacoesDeRestaurantes` com um método `Recomendacoes` que recebe uma mensagem que contém uma lista (denotada pelo `repeated`) de `restauranteId`.

Então, deve-se fazer o build do projeto, criando um JAR com as classes geradas que será implantado no repositório local do Maven:

```
cd ~/Desktop/recomendacoes-idl
```

```
mvn clean install
```

O recomendacoes-idl-0.0.1-SNAPSHOT.jar deve conter as classes:

- RecomendacoesDeRestaurantesGrpc, com a classe base do servidor e os stubs do cliente
 - Recomendacoes, com as classes Java que serão serializadas para Protocol Buffers
2. Crie um projeto fj33-recomendacoes-service que utiliza o Spring Boot e contém um pom.xml com o seguinte conteúdo:

```
##### fj33-recomendacoes-service/pom.xml
```

```
<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://
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.8.RELEASE</version>
  <relativePath /> <!-- lookup parent from repository -->
</parent>

<groupId>br.com.caelum</groupId>
<artifactId>recomendacoes-service</artifactId>
<version>0.0.1-SNAPSHOT</version>

<properties>
  <java.version>1.8</java.version>
  <maven-jar-plugin.version>3.1.1</maven-jar-plugin.version>
</properties>

<dependencies>

<dependency>
```

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

</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>
```

Perceba que não é um projeto Web.

Adicione ao `pom.xml` dependências ao starter gRPC da LogNet e ao projeto `recomendacoes-idl`:

fj33-recomendacoes-service/pom.xml

```
<dependency>
  <groupId>io.github.lognet</groupId>
  <artifactId>grpc-spring-boot-starter</artifactId>
  <version>3.4.3</version>
</dependency>

<dependency>
  <groupId>br.com.caelum</groupId>
  <artifactId>recomendacoes-idl</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>
</dependency>
```

Mantenha inalterada a classe `RecomendacoesApplication`, no pacote `br.com.caelum.eats.recomendacoes`:

```
##### fj33-recomendacoes-
service/src/main/java(br/com/caelum/eats/recomendacoes/Recomendac
oesApplication.java
```

```
@SpringBootApplication
public class RecomendacoesApplication {

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

Crie, no pacote `br.com.caelum.eats.recomendacoes`, uma classe `RecomendacoesService`. Essa classe deve estender de `RecomendacoesDeRestaurantesGrpc.RecomendacoesDeRestaurantesImplBase` e ser anotada com `@GRpcService`. No método `recomendacoes`, faça um `shuffle` da lista de ids de restaurantes e retorne o resultado:

```
##### fj33-recomendacoes-
service/src/main/java(br/com/caelum/eats/recomendacoes/Recomendac
oesApplication.java
```

```
@GRpcService
public class RecomendacoesService extends RecomendacoesDeRestaurante
{
    @Override
    public void recomendacoes(Restaurantes request, StreamObserver<List<Long>> response) {
        List<Long> idsDeRestaurantes = request.getRestaurantesIdList();
        Collections.shuffle(idsDeRestaurantes);
        response.onNext(idsDeRestaurantes);
        response.onCompleted();
    }
}
```

```
// simula recomendacao
List<Long> idsDeRestaurantesOrdenadosPorRecomendacoes = ic
if (idsDeRestaurantes.size() > 1) {
    idsDeRestaurantesOrdenadosPorRecomendacoes = new ArrayList<
        Collections.shuffle(idsDeRestaurantesOrdenadosPorRecomen
    }
}

Restaurantes response = Restaurantes
    .newBuilder()
    .addAllRestauranteId(idsDeRestaurantesOrdenadosPorRecomen
    .build();

response0bserver.onNext(response);
response0bserver.onCompleted();
}

}
```

Devem ser definidos os seguintes imports:

```
##### fj33-recomendacoes-
service/src/main/java/br/com/caelum/eats/recomendacoes/Recomendac
oesApplication.java

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

import org.lognet.springboot.grpc.GRpcService;

import br.com.caelum.eats.recomendacoes.grpc.Recomendacoes.Res
import br.com.caelum.eats.recomendacoes.grpc.RecomendacoesDeRe
import io.grpc.stub.StreamObserver;
```

Suba o serviço de Recomendações, executando a classe RecomendacoesApplication. No Console, deverá aparecer algo como:

```
2019-10-30 14:13:23.278 INFO 25955 --- [ restartedMain] b.c.c.e.r.Recomen  
2019-10-30 14:13:23.279 INFO 25955 --- [ restartedMain] o.l.springboot.gr  
2019-10-30 14:13:23.339 INFO 25955 --- [ restartedMain] o.l.springboot.gr  
2019-10-30 14:13:23.538 INFO 25955 --- [ restartedMain] o.l.springboot.gr
```

3. Vamos fazer a implementação do cliente no serviço de Distância.

Adicione ao `pom.xml` do serviço de Distância dependências aos projetos do gRPC e ao `recomendacoes-idl`:

```
##### fj33-eats-distancia-service/pom.xml
```

```
<dependency>  
    <groupId>io.grpc</groupId>  
    <artifactId>grpc-netty-shaded</artifactId>  
    <version>1.24.0</version>  
</dependency>  
<dependency>  
    <groupId>io.grpc</groupId>  
    <artifactId>grpc-protobuf</artifactId>  
    <version>1.24.0</version>  
</dependency>  
<dependency>  
    <groupId>io.grpc</groupId>  
    <artifactId>grpc-stub</artifactId>  
    <version>1.24.0</version>  
</dependency>  
  
<dependency>  
    <groupId>br.com.caelum</groupId>  
    <artifactId>recomendacoes-idl</artifactId>  
    <version>0.0.1-SNAPSHOT</version>  
</dependency>
```

Adicione, ao `application.properties` do serviço de Distância, propriedades para o host e a porta do serviço de Recomendações:

```
##### fj33-eats-distancia-
service/src/main/resources/application.properties
```

```
recomendacoes.service.host=localhost
recomendacoes.service.port=6565
```

No pacote `br.com.caelum.eats.distancia`, crie uma nova classe `RecomendacoesGrpcClient` que será o cliente gRPC do serviço de Recomendações. Anote-a com `@Service`, para ser gerenciada pelo Spring, e `@Slf4j`, do Lombok, para logs.

```
##### fj33-eats-distancia-
service/src/main/java(br/com/caelum/eats/distancia/RecomendacoesGrp
cClient.java
```

```
@Slf4j
@Service
class RecomendacoesGrpcClient {
```

```
}
```

Receba o host e a porta do serviço de Recomendações no construtor, anotando os parâmetros com `@Value`, apontando para as propriedades definidas anteriormente e armazenando os valores em atributos.

```
##### fj33-eats-distancia-
service/src/main/java(br/com/caelum/eats/distancia/RecomendacoesGrp
cClient.java
```

```
// anotações omitidas...
class RecomendacoesGrpcClient {

    private String recomendacoesServiceHost; // adicionado
    private Integer recomendacoesServicePort; // adicionado
```

```
// adicionado
RecomendacoesGrpcClient(@Value("${recomendacoes.service.host")
    @Value("${recomendacoes.service.port}") Integer
    this.recomendacoesServiceHost = recomendacoesServiceHost;
    this.recomendacoesServicePort = recomendacoesServicePort;
}

}
```

Crie um método `conectaAoRecomendacoesGrpcService` que faz a conexão com o serviço gRPC usando um `ManagedChannel` para criar um *blocking stub*, salvando as instâncias em atributos, e anote o método com `@PostConstruct`. Também crie um método `desconectaDoRecomendacoesGrpcService`, que pára a conexão gRPC, e anote-o com `@PreDestroy`.

```
##### fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/RecomendacoesGrp
cClient.java
```

```
// anotações omitidas...
class RecomendacoesGrpcClient {

    // demais atributos omitidos ...
    private ManagedChannel channel; // adicionado
    private RecomendacoesDeRestaurantesBlockingStub recomendacoe

    // construtor omitido ...

    // adicionado
    @PostConstruct
    void conectaAoRecomendacoesGrpcService() {
        channel = ManagedChannelBuilder.forAddress(recomendacoesSe
            .usePlaintext() // desabilita TLS porque precisa
            .build();
        recomendacoes = RecomendacoesDeRestaurantesGrpc.newBlockir
```

```
}

// adicionado
@PreDestroy
void desconectaDoRecomendacoesGrpcService() {
    channel.shutdown();
}

}
```

Defina um método `ordenaPorRecomendacoes` que recebe uma lista de ids de restaurantes, montando o request, invocando o stub gRPC e retornado a lista obtida do serviço de Recomendações.

```
##### fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/RecomendacoesGrp
cClient.java
```

```
// anotações omitidas...
class RecomendacoesGrpcClient {

    // código omitido...

    List<Long> ordenaPorRecomendacoes(List<Long> idsDeRestaurant
        Restaurantes restaurantes = Restaurantes
            .newBuilder()
            .addAllRestauranteId(idsDeRe
            .build();

    Restaurantes restaurantesOrdenadosPorRecomendacao = recom

    List<Long> restaurantesOrdenados = restaurantesOrdenadosPc
        log.info("Restaurantes ordenados: {}", restaurantesOrdenad

    return restaurantesOrdenados;
}
```

```
}
```

Os imports da classe `RecomendacoesGrpcClient` devem ser os seguintes:

```
##### fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/RecomendacoesGrp
cClient.java

import java.util.List;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

import br.com.caelum.eats.recomendacoes.grpc.RecomendacoesRes
import br.com.caelum.eats.recomendacoes.grpc.RecomendacoesDeRe
import br.com.caelum.eats.recomendacoes.grpc.RecomendacoesDeRe
import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
import lombok.extern.slf4j.Slf4j;
```

Use a classe `RecomendacoesGrpcClient` em `DistanciaService`.

Em um novo método auxiliar `ordenaPorRecomendacoes`, que recebe uma lista de restaurantes, extraí os ids e invoca o cliente gRPC do serviço de Recomendações, reordenando a lista de restaurantes de acordo com a lista de ids retornada :

```
##### fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/DistanciaService.java
```

```
//anotações omitidas...
class DistanciaService {
```

```
// demais atributos omitidos...

private RecomendacoesGrpcClient recomendacoesClient; // adicionado

// demais métodos omitidos...

// adicionado
private List<Restaurante> ordenaPorRecomendacoes(List<Restaurante> restaurantes) {
    if (restaurantes.size() > 1) {
        List<Long> idsDeRestaurantes = restaurantes
            .stream()
            .map(Restaurante::getId)
            .collect(Collectors.toCollection(LinkedList::new));
        List<Long> idsDeRestaurantesOrdenadosPorRecomendacao = restaurantes
            .stream()
            .sorted(Comparator.comparing(restaurante ->
                idsDeRestaurantesOrdenadosPorRecomendacao
                    .indexOf(restaurante.getId())));

        return restaurantesOrdenadosPorRecomendacao;
    }
    return restaurantes;
}

}
```

Utilize o método `ordenaPorRecomendacoes` no método `calculaDistanciaParaOsRestaurantes`:

```
#####
# fj33-eats-distancia-
# service/src/main/java/br/com/caelum/eats/distancia/DistanciaService.java
#
//anotações omitidas...
```

```
class DistanciaService {  
    // código omitido...  
  
    private List<RestauranteComDistanciaDto> calculaDistanciaPar  
        List<Restaurante  
        String cep) {  
        return restaurantes  
            return ordenaPorRecomendacoes(restaurantes) // modificado  
                .stream()  
                .map(restaurante -> {  
                    String cepDoRestaurante = restaurante.getCep()  
                    BigDecimal distancia = distanciaDoCep(cepDoRes  
                    Long restauranteId = restaurante.getId();  
                    return new RestauranteComDistanciaDto(restaur  
                })  
                .collect(Collectors.toList());  
    }  
}
```

Execute a classe `EatsDistanciaServiceApplication`.

4. Em um Terminal, use o cURL para invocar o serviço de Distância algumas vezes, da seguinte maneira:

```
curl http://localhost:8082/restaurantes/mais-proximos/78000-77
```

Perceba que a ordem dos restaurantes é modificada aleatoriamente.

Por exemplo, se tivermos cadastrados os restaurantes de ids 1, 2 e 3 no serviço de Distância, obteremos respostas semelhantes às que seguem.

Na primeira chamada, teremos algo como:

```
[{"restauranteId":3,"distancia":9.4187908418432879642523403163067996501922}, {"restauranteId":1,"distancia":6.5896083315768390065159110235981643199920}, {"restauranteId":2,"distancia":3.5266267579559347211670683464035391807556}]
```

Já na segunda chamada:

```
[{"restauranteId":2,"distancia":12.516594449293343416229618014767765998840}, {"restauranteId":1,"distancia":0.0912536301024052809438558142574038356542}, {"restauranteId":3,"distancia":7.1085681255576895765102563018444925546646}]
```

Na terceira:

```
[{"restauranteId":1,"distancia":13.142009392201535078470442385878413915634}, {"restauranteId":3,"distancia":4.5362206318291011797327882959507405757904}, {"restauranteId":2,"distancia":11.979686395696951706213440047577023506164}]
```

Para saber mais: Todo o poder emana do cliente - explorando uma API GraphQL

O texto dessa seção é baseado no post [Todo o poder emana do cliente: explorando uma API GraphQL](#) (AQUILES, 2017) do blog da Caelum, disponível em: <https://blog.caelum.com.br/todo-o-poder-emana-do-cliente-explorando-uma-api-graphql>

Quais as limitações de uma API REST?

Para ilustrar o que pode ser melhorado em uma API REST, vamos utilizar a [versão 3](#) da API do GitHub, considerada muito consistente e aderente aos princípios REST.

Queremos uma maneira de avaliar bibliotecas open-source. Para isso, dado um repositório do GitHub, desejamos descobrir:

- o número de stars
- o número de pull requests abertos

Como exemplo, vamos usar o repositório de uma biblioteca NodeJS muito usada: o framework Web minimalista Express.

Obtendo detalhes de um repositório

Lendo a documentação da API do GitHub, descobrimos que para [obter detalhes sobre um repositório](#), devemos enviar uma requisição GET para `/repos/:owner/:repo`. Então, para o repositório do Express, devemos fazer:

```
GET https://api.github.com/repos/expressjs/express
```

Como resposta, obtemos:

- 2.2 KB *gzipados* transferidos, incluindo cabeçalhos
- 6.1 KB de JSON em 110 linhas, quando descompactado

```
200 OK
```

```
Content-type: application/json; charset=utf-8
```

```
{  
  "id": 237159,  
  "name": "express",  
  "full_name": "expressjs/express",  
  "private": false,  
  "html_url": "https://github.com/expressjs/express",  
  "description": "Fast, unopinionated, minimalist web framev  
  "fork": false,  
  "issues_url": "https://api.github.com/repos/expressjs/expr
```

```
"pulls_url": "https://api.github.com/repos/expressjs/expre  
"stargazers_count": 33508,  
...  
}
```

O JSON retornado tem diversas informações sobre o repositório do Express. Por meio da propriedade `stargazers_count`, descobrimos que há mais de 33 mil stars.

Porém, **não** temos o número de pull requests abertos.

Obtendo os pull requests de um repositório

Na propriedade `pulls_url`, temos apenas uma URL:

<https://api.github.com/repos/expressjs/express/pulls{/number}>.

Um bom palpite é que sem esse `{/number}` teremos a lista de todos os pull requests, o que pode ser confirmado na [seção de pull requests](#) da documentação da API REST do GitHub.

O `{/number}` da URL segue o modelo proposto pela [RFC 6570](#) (URI Template).

Mas como filtrar apenas pelos pull requests abertos?

Na mesma documentação, verificamos que podemos usar a URL `/repos/:owner/:repo/pulls?state=open` ou simplesmente `/repos/:owner/:repo/pulls`, já que o filtro por pull requests abertos é aplicado por padrão. Em outras palavras, precisamos de outra requisição:

```
GET https://api.github.com/repos/expressjs/express/pulls
```

A resposta é:

- 54.1 KB *gzipados* transferidos, incluindo cabeçalhos

- 514 KB de JSON em 9150 linhas, quando descompactado

```
200 OK
```

```
Content-type: application/json; charset=utf-8
Link: <https://api.github.com/repositories/237159/pulls?page=2>; rel="next"
<https://api.github.com/repositories/237159/pulls?page=2>; rel="last"
```

```
[  
 {  
     //um pull request...  
     "url": "https://api.github.com/repos/expressjs/express/  
     "id": 134639441,  
     "html_url": "https://github.com/expressjs/express/pull/  
     "diff_url": "https://github.com/expressjs/express/pull/  
     "patch_url": "https://github.com/expressjs/express/pull/  
     "issue_url": "https://api.github.com/repos/expressjs/express/  
     "number": 3391,  
     "state": "open",  
     "locked": false,  
     "title": "Update guide to ES6",  
     "user": {  
         "login": "jevtovich",  
         "id": 13847095,  
         "avatar_url": "https://avatars3.githubusercontent.com/u/13847095?v=4&s=400",  
         ...  
     },  
     "body": "",  
     "created_at": "2017-08-08T11:40:32Z",  
     "updated_at": "2017-08-08T17:28:01Z",  
     ...  
 },  
 {  
     //outro pull request...  
     "url": "https://api.github.com/repos/expressjs/express/  
     "id": 134634529,  
     ...  
 },  
 ...  
 ]
```

É retornado um array de 30 objetos que representam os pull requests. Cada objeto ocupa uma média de 300 linhas, com informações sobre status, descrição, autores, commits e diversas URLs relacionadas.

Disso tudo, só queremos saber a contagem: 30 pull requests. Não precisamos de **nenhuma** outra informação.

Mas há outra questão: o resultado é paginado com 30 resultados por página, por padrão, conforme descrito na [seção de paginação](#) da documentação da API REST do GitHub.

As URLs das próximas páginas devem ser obtidas a partir do cabeçalho de resposta `Link`, extraindo o `rel` (*link relation*) `next`.

Os links para as próximas páginas seguem o conceito de hipermídia do REST e foram implementados usando o cabeçalho `Link` e o formato descrito na [RFC 5988](#) (Web Linking). Essa RFC sugere um punhado de link relations padronizados.

Então, a partir do `next`, seguimos para a próxima página:

```
GET https://api.github.com/repositories/237159/pulls?page=2
```

Temos como resposta:

- 26.9 KB *gzipados* transferidos, incluindo cabeçalhos
- 248 KB de JSON em 4394 linhas, quando descompactado

```
200 OK
```

```
Content-type: application/json; charset=utf-8
```

```
Link: <https://api.github.com/repositories/237159/pulls?page=1>; rel="first"
      <https://api.github.com/repositories/237159/pulls?page=1>; rel="prev"
```

```
[  
 {  
   //um pull request...  
   "url": "https://api.github.com/repos/expressjs/express/pull/41965836",  
   ...  
 },  
 {  
   //outro pull request...  
   "url": "https://api.github.com/repos/expressjs/express/pull/39735937",  
   ...  
 },  
 ...  
 ]
```

O array retornado contabiliza mais 14 objetos representando os pull requests. Dessa vez, não há o link relation next, indicando que é a última página.

Então, sabemos que há 44 (30 + 14) pull requests abertos no repositório do Express.

Resumindo a consulta REST

No momento da escrita desse artigo, o número de stars do Express no GitHub é 33508 e o de pull requests abertos é 44. Para descobrir isso, tivemos que:

- disparar 3 requisições ao servidor
- baixar 83.2 KB de informações gzipadas e cabeçalhos
- fazer parse de 768.1 KB de JSON ou 13654 linhas O que daria pra melhorar? Ir menos vezes ao servidor, baixando menos dados!

Não é um problema com o REST em si, mas uma discrepância entre a modelagem atual da API e as nossas necessidades.

Poderíamos pedir para o GitHub implementar um recurso específico que retornasse somente as informações, tudo em apenas um request.

Mas será que o pessoal do GitHub vai nos atender?

Mais flexibilidade e eficiência com GraphQL

Numa API GraphQL, o cliente diz exatamente os dados que quer da API, tornando a requisição muito **flexível**.

A API, por sua vez, retorna apenas os dados que o cliente pediu, fazendo com que a transferência da resposta seja bastante **eficiente**.

Mas afinal de contas, o que é GraphQL?

GraphQL *não* é um banco de dados, *não* é um substituto do SQL, *não* é uma ferramenta do lado do servidor e *não* é específico para React (apesar de muito usado por essa comunidade).

Um servidor que aceita requisições GraphQL poderia ser implementado em *qualquer* linguagem usando qualquer banco de dados. Há várias [bibliotecas](#) de diferentes plataformas que ajudam a implementar esse servidor.

Clientes que enviam requisições GraphQL também poderiam ser implementados em qualquer tecnologia: web, mobile, desktop, etc. Diversas [bibliotecas](#) auxiliam nessa tarefa.

GraphQL é uma **query language para APIs** que foi [especificada](#) pelo Facebook em 2012 para uso interno e aberta ao público em 2015.

A *query language* do GraphQL é **fortemente tipada** e descreve, através de um *schema*, o modelo de dados oferecido pelo serviço. Esse schema pode ser usado para verificar se uma dada requisição é válida e, caso seja, executar as tarefas no back-end e estruturar os dados da resposta.

Um cliente pode enviar 3 tipos de requisições GraphQL, os *root types*:

- *query*, para consultas;
- *mutation*, para enviar dados;
- *subscription*, para comunicação baseada em eventos.

Montando uma consulta GraphQL

A [versão 4](#) da API do GitHub, a mais recente, dá suporte a requisições GraphQL.

Para fazer nossa consulta às stars e aos pull requests abertos do repositório do Express usando a API GraphQL do GitHub, devemos começar com a query:

Vamos usar o campo `repository` da query, que recebe os argumentos `owner` e `name`, ambos obrigatórios e do tipo String. Para buscar pelo Express, devemos fazer:

```
query {  
  repository (owner: "expressjs", name: "express") {  
  }  
}
```

A partir do objeto `repository`, podemos descobrir o número de stars por meio do campo `stargazers`, que é uma connection do tipo `StargazerConnection`. Como queremos apenas a quantidade de itens, só precisamos obter propriedade `totalCount` dessa connection.

```
query {  
  repository (owner: "expressjs", name: "express") {  
    stargazers {  
      totalCount  
    }  
  }  
}
```

Para encontrarmos o número de pull requests abertos, basta usarmos o campo `pullRequests` do `repository`, uma connection do tipo `PullRequestConnection`. O filtro por pull requests abertos não é aplicado por padrão. Por isso, usaremos o argumento `states`. Da connection, obteremos apenas o `totalCount`.

```
query {
  repository(owner: "expressjs", name: "express") {
    stargazers {
      totalCount
    }
    pullRequests(states: OPEN) {
      totalCount
    }
  }
}
```

Basicamente, é essa a nossa consulta! Bacana, não?

Uma maneira de “rascunhar” consultas GraphQL é usar a ferramenta [GraphiQL](#), que permite explorar APIs pelo navegador. Há até code completion! Boa parte das APIs GraphQL dá suporte, incluindo [a do GitHub](#).

Tá, mas como enviar a consulta para a API?

A maneira mais comum de publicar APIs GraphQL é usar a boa e velha Web, com seu protocolo HTTP.

Apesar do HTTP ser o mais usado para publicar APIs GraphQL, teoricamente não há limitações em usar outros protocolos.

Uma API GraphQL possui apenas um *endpoint* e, consequentemente, só uma URL.

É possível enviar requisições GraphQL usando o método `GET` do HTTP,

com a consulta como um parâmetro na URL. Porém, como as consultas são relativamente grandes e requisições GET tem um limite de tamanho, o método mais utilizado pelas APIs GraphQL é o POST, com a consulta no corpo da requisição.

No caso do GitHub a URL do endpoint GraphQL é:

<https://api.github.com/graphql>

O GitHub só dá suporte ao método POST e o corpo da requisição deve ser um JSON cuja propriedade query conterá uma String com a nossa consulta.

Mesmo para consultas a repositórios públicos, a API GraphQL do GitHub precisa de um token de autorização.

```
POST https://api.github.com/graphql
Content-type: application/json
Authorization: bearer f023615deb415e...
```

```
{
  "query": "query {
    repository(owner: \"expressjs\", name: \"express\") {
      stargazers {
        totalCount
      }
      pullRequests(states: OPEN) {
        totalCount
      }
    }
  }"
}
```

O retorno será um JSON em que os dados estarão na propriedade data:

```
{
```

```
"data": {  
    "repository": {  
        "stargazers": {  
            "totalCount": 33508  
        },  
        "pullRequests": {  
            "totalCount": 44  
        }  
    }  
}
```

Na verdade, os JSONs de requisição e resposta ficam em apenas 1 linha. Formatamos o código anterior em várias linhas para melhor legibilidade.

Repare que os campos da consulta, dentro da `query`, tem exatamente a mesma estrutura do retorno da API. É como se a resposta fosse a própria consulta, mas com os valores preenchidos. Por isso, montar consultas com GraphQL é razoavelmente intuitivo.

Resumindo a consulta GraphQL

Obtivemos os mesmos resultados: 33508 stars e 44 pull requests. Para isso, tivemos que:

- disparar apenas 1 requisição ao servidor
- baixar somente 996 bytes de informações *gzipadas*, incluindo cabeçalhos
- fazer parse só de 93 bytes de JSON

São 66,67% requisições a menos, 98,82% menos dados e cabeçalhos trafegados e 99,99% menos JSON a ser “parseado”. Ou seja, **MUITO mais rápido**.

Considerações finais

O GraphQL dá bastante poder ao cliente. Isso é especialmente útil quando a equipe que implementa o cliente é totalmente separada da que implementa o servidor. Mas há casos mais simples, em que as equipes do cliente e servidor trabalham juntas. Então, não haveria tanta dificuldade em manter uma API RESTful customizada para o cliente.

Poderíamos buscar outros dados da API do GitHub: o número de issues abertas, a data da última release, informações sobre o último commit, etc.

Uma coisa é certa: com uma consulta GraphQL, eu faria menos requisições e receberia menos dados desnecessários. Mais flexibilidade e mais eficiência.

Considerando o Modelo de Maturidade de Richardson, podemos considerar que o GraphQL está no Nível 0:

- não diferentes recursos, apenas uma URI como ponto de entrada para toda a API GraphQL
- não há a ideia de diferentes verbos HTTP, só é usado POST
- não há diferentes representações, apenas um JSON que contém uma estrutura GraphQL

Existem várias outras questões que surgem ao estudar o GraphQL:

- como fazer um servidor que atenda a toda essa flexibilidade?
- é possível gerar uma documentação a partir do código para a minha API?
- vale a pena migrar minha API pra GraphQL?
- posso fazer uma “casca” GraphQL para uma API REST já existente?
- como implementar um cliente sem muito trabalho?
- quais os pontos ruins dessa tecnologia e desafios na implementação?

Para saber mais: Field Selectors

Um maneira de otimizar uma API REST já existente é implementar um

mecanismo de obter um subconjunto das representações de um recurso.

A API do Linkedin, por exemplo, implementa [field projections](#), que permitem selecionar os campos retornados. Por exemplo:

```
GET https://api.linkedin.com/v2/people/id=-f_Ut43FoQ?projection=(id,localiz
```

```
{  
  "id": "-f_Ut43FoQ",  
  "localizedFirstName": "Dwight",  
  "localizedLastName": "Schrute"  
}
```

A Graph API do Facebook é uma API REST (não GraphQL!) que permite que programadores interajam com a plataforma do Facebook para ler ou enviar dados de usuário, páginas, fotos, entre outros. Essa API implementa field expansions uma maneira de retornar [apenas os campos](#):

```
https://graph.facebook.com/{your-user-id}?fields=birthday,email,hometown&ac
```

```
{  
  "hometown": "Your, Hometown",  
  "birthday": "01/01/1985",  
  "email": "your-email@email.addresss.com",  
  "id": "{your-user-id}"  
}
```

Diversas APIs do Google permitem [partial responses](#), em que apenas os campos necessários são retornados:

```
GET https://www.googleapis.com/demo/v1?fields=kind,items(title,characterist
```

```
{  
  "kind": "demo",  
  "items": [{  
    "title": "First title",  
    "characteristics": {  
      "length": "short"  
    }  
, {  
    "title": "Second title",  
    "characteristics": {  
      "length": "long"  
    }  
,  
  ]  
}
```

API Gateway

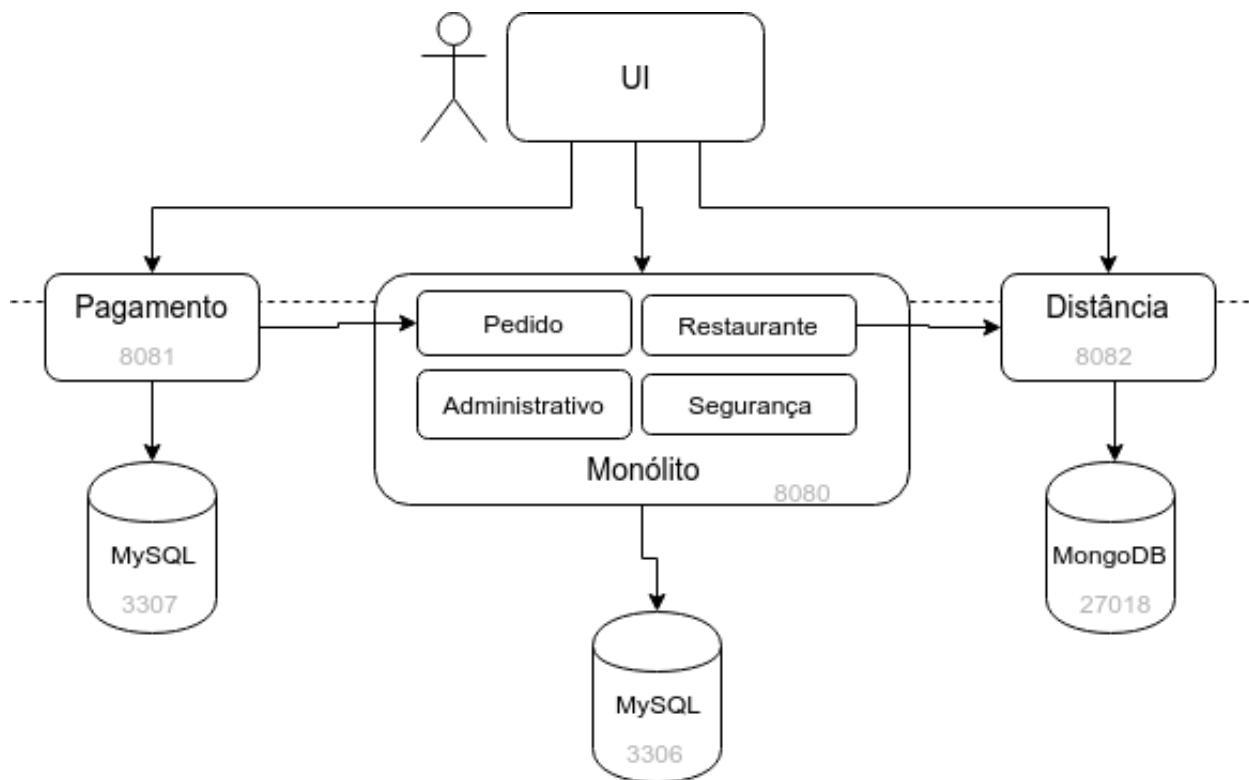
No momento em que quebramos o Monólito, extraímos os serviços de Pagamentos e Distância, tivemos que alterar a UI do Caelum Eats.

UI passou a chamar diretamente os serviços de Pagamentos, de Distância e o Monólito e, para isso, precisou conhecer as URLs dos novos serviços.

O arquivo `environment.ts` do código da UI reflete esse fato:

```
##### fj33-eats-ui/src/environments/environment.ts
```

```
export const environment = {  
  production: false,  
  baseUrl: '//localhost:8080',  
  pagamentoUrl: '//localhost:8081',  
  distanciaUrl: '//localhost:8082'  
};
```



Esse cenário, em que a UI invoca diretamente as APIs de cada serviço,

traz alguns problemas. Os principais deles são:

- *Falta de encapsulamento*: expondo todos os serviços, a UI conhece detalhes sobre a implementação do sistema como um todo. E isso é problemático porque a implementação muda constantemente: novos serviços serão criados, outros serviços serão extraídos do Monólito, alguns serão divididos em dois e outros mesclados em apenas um serviço. A cada alteração, a UI deverá ser modificada em conjunto com os serviços modificados. O mesmo ocorrerá com o deploy.
- *Segurança*: sob uma perspectiva de Segurança da Informação, há uma grande *superfície exposta* na Arquitetura atual do Caelum Eats. Cada serviço exposto pode ser alvo de ataques.

Poderíamos criar um *edge service*, que fica exposto na fronteira da rede e funciona como um *proxy reverso* para os demais serviços, também chamados de *downstream services*.

Um *proxy*, ou *forward proxy*, age como intermediário entre seus clientes e a Internet. É comumente usado em redes internas de organizações para limitar acesso a redes sociais e monitorar o tráfego de rede. Um *reverse proxy* age como intermediário entre a Internet e servidores de uma rede interna, afim de proteger o acesso e limitar o conhecimento dos clientes externos sobre a estrutura interna da rede.

Assim, a UI chamaría apenas esse *edge service*, sem conhecer as URLs dos *downstream services*. Aumentaríamos o encapsulamento e diminuiríamos a superfície exposta no perímetro da rede.

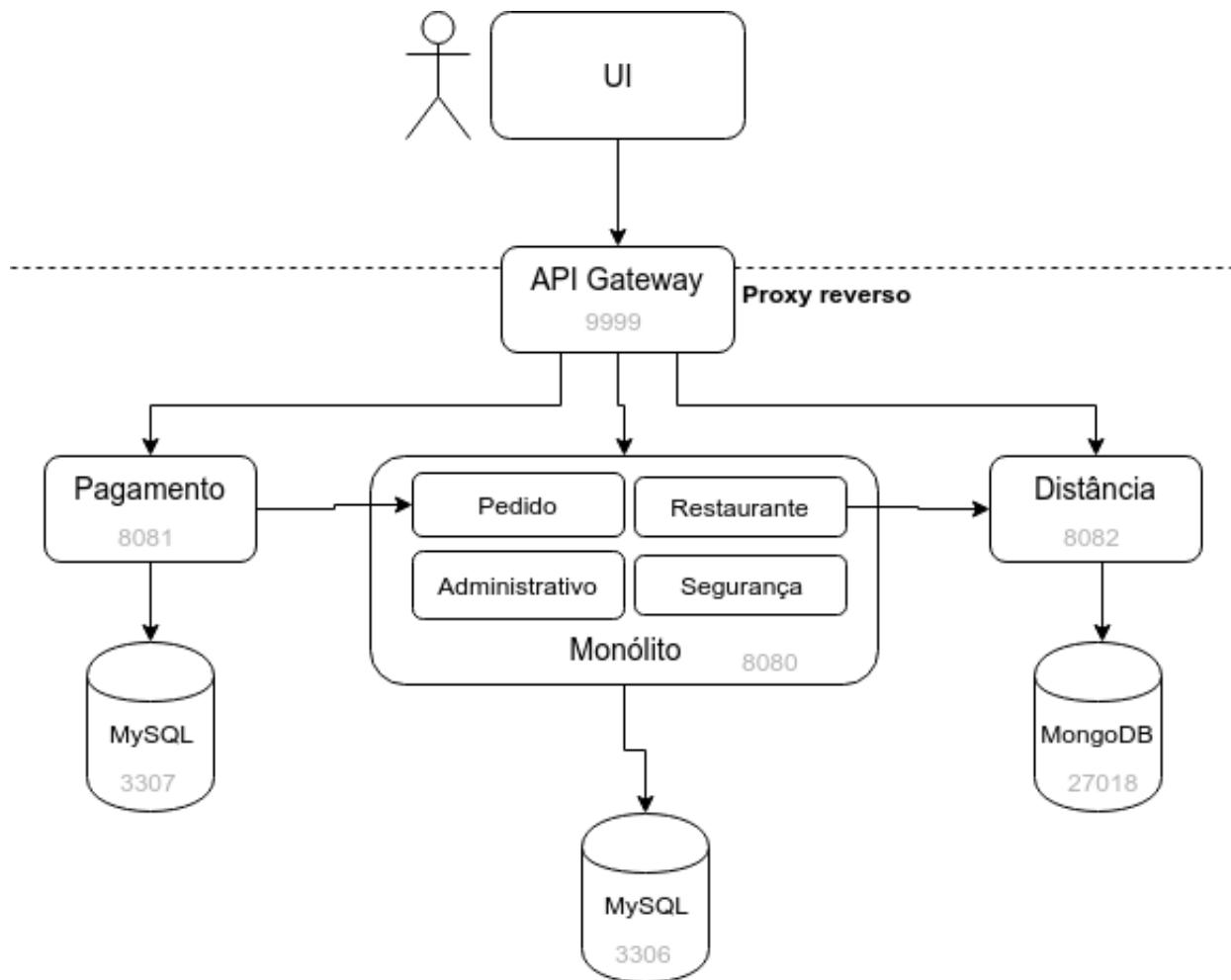
A API desse *edge service* serviria como a *API pública* da organização.

Chamadas entre serviços, feitas na rede interna, não precisariam passar por esse *edge service*.

Em uma Arquitetura de Microservices, esse tipo de *edge service* é chamado de **API Gateway**.

Pattern: API Gateway

Um edge service que é o ponto de entrada de uma API para seus clientes externos.



Implementações de API Gateway

Existem diferentes implementações de API Gateway disponíveis:

- [AWS API Gateway](#), disponibilizado pela Amazon na plataforma AWS
- [Kong](#), open-source, um conjunto de extensões do NGINX [escritos em Lua](#) que provê diversas capacidades e permite a criação de plugins
- [Traefik](#), open-source, implementado em Go
- [Spring Cloud Gateway](#), parte da plataforma Spring, implementado com o framework Web reativo Spring WebFlux
- [Zuul](#), open-source, um projeto da Netflix feito em Java

No orquestrador de containers [Kubernetes](#), há o conceito de [Ingress](#), uma abstração que provê rotas HTTP e HTTPS de fora do cluster para

[Services](#) internos. Um Ingress deve ter uma implementação, chamada de [Ingress Controller](#). Um Ingress Controller mantido pelo time do Kubernetes é o NGINX. Podem ser usados como Ingress Controller o Traefik e Kong, mencionados anteriormente. Há também diversas outras implementações como [Ambassador](#), [Contour](#) da VMWare, [Gloo](#).

Zuul

No artigo de lançamento do Zuul no blog de Tecnologia da Netflix, [Announcing Zuul: Edge Service in the Cloud](#) (COHEN; HAWTHORNE, 2013), é afirmado que o Zuul é usado na Netflix API de diversas formas. Entre elas:

- Autenticação e Segurança em geral
- Insights
- Teste de Stress
- Testes (ou releases) Canário, um release parcial para um subconjunto das máquinas de produção, para minimizar o impacto e o risco de uma nova versão
- Roteamento Dinâmico, que veremos em capítulos posteriores

Mikey Cohen, na palestra [Zuul @ Netflix](#) (COHEN, 2016), diz que, na arquitetura da Netflix, há mais de 20 clusters Zuul em produção, que lidam com dezenas de bilhões de requests por dia em 3 regiões AWS diferentes.

Há duas versões do Zuul, cujas diferenças são explicadas com detalhes na palestra [Zuul's Journey to Non-Blocking](#) (GONIGBERG, 2017):

- Zuul 1, que usa a API de Servlet sobre um Tomcat
- Zuul 2, que usa I/O Assíncrono com Netty

Curiosidade: na palestra mencionada anteriormente, Arthur Gonigberg lembra que Zuul é o nome do mostro do filme Ghostbusters conhecido como The Gatekeeper (em português, algo como porteiro).

Usaremos o projeto [Spring Cloud Netflix Zuul](#), que integra o Zuul 1 ao Spring Boot.

Implementando um API Gateway com Zuul

Pelo navegador, abra `https://start.spring.io/`.

Em *Project*, mantenha *Maven Project*. Em *Language*, mantenha *Java*. Em *Spring Boot*, mantenha a versão padrão. No trecho de *Project Metadata*, defina:

- `br.com.caelum` em *Group*
- `api-gateway` em *Artifact*

Mantenha os valores em *More options*.

Mantenha o *Packaging* como `Jar`. Mantenha a *Java Version* em 8.

Em *Dependencies*, adicione:

- Zuul
- DevTools

Clique em *Generate Project*.

Extraia o `api-gateway.zip` e copie a pasta para seu Desktop.

Adicione a anotação `@EnableZuulProxy` à classe `ApiGatewayApplication`:

```
##### fj33-api-
gateway/src/main/java;br\com\caelum\apigateway\ApiGatewayApplication
.java
```

```
@EnableZuulProxy
@SpringBootApplication
public class ApiGatewayApplication {

    public static void main(String[] args) {
```

```
        SpringApplication.run(ApiGatewayApplication.class, args);  
    }  
}
```

Não deixe de adicionar o import:

```
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;
```

No arquivo `src/main/resources/application.properties`:

- modifique a porta para 9999
- desabilite o Eureka, por enquanto (o abordaremos mais adiante)
- para as URLs do serviço de pagamento, parecidas com
`http://localhost:9999/pagamentos/algum-recurso`, redirecione para `http://localhost:8081`. Para manter o prefixo `/pagamentos`, desabilite a propriedade `stripPrefix`.
- para as URLs do serviço de distância, algo como
`http://localhost:9999/distancia/algum-recurso`, redirecione para `http://localhost:8082`. O prefixo `/distancia` será removido, já que esse é o comportamento padrão.
- para as demais URLs, redirecione para `http://localhost:8080`, o monólito.

O arquivo ficará semelhante a:

```
##### fj33-api-gateway/src/main/resources/application.properties
```

```
server.port = 9999
```

```
ribbon.eureka.enabled=false
```

```
zuul.routes.pagamentos.url=http://localhost:8081  
zuul.routes.pagamentos.stripPrefix=false
```

```
zuul.routes.distancia.url=http://localhost:8082
```

```
zuul.routes.monolito.path=/*
```

```
zuul.routes.monolito.url=http://localhost:8080
```

Com as configurações anteriores, a URL

`http://localhost:9999/pagamentos/1` será direcionada para

`http://localhost:8081/pagamentos/1`, mantendo o prefixo pagamentos.

Já a URL `http://localhost:9999/distancia/restaurantes/mais-proximos/71503510` será direcionada para

`http://localhost:8082/restaurantes/mais-proximos/71503510`,

removendo o prefixo distancia.

Outras URLs, que não iniciam com /pagamentos ou /distancia, serão direcionadas para o Monólito. Por exemplo, a URL

`http://localhost:9999/restaurantes/1`, será direcionada para

`http://localhost:8080/restaurantes/1`.

Fazendo a UI usar o API Gateway

Remova as URLs específicas dos serviços de distância e pagamento, mantendo apenas a baseUrl, que deve apontar para o API Gateway:

```
##### fj33-eats-ui/src/environments/environment.ts
```

```
export const environment = {
```

```
  production: false,
```

```
  baseUrl: 'http://localhost:8080' -
```

```
  baseUrl: 'http://localhost:9999' // modificado
```

```
  , - pagamentoUrl: 'http://localhost:8081' -
```

```
  , - distanciaUrl: 'http://localhost:8082' -
```

```
};
```

Em PagamentoService, troque pagamentoUrl por baseUrl:

```
##### fj33-eats-ui/src/app/services/pagamento.service.ts
```

```
export class PagamentoService {  
  
    private API == environment.pagamentoUrl +- +/pagamentos;  
    private API = environment.baseUrl + '/pagamentos'; // modifi  
  
    // restante do código ...  
  
}
```

Use apenas baseUrl em RestauranteService, alterando o atributo DISTANCIA_API:

```
##### fj33-eats-ui/src/app/services/restaurante.service.ts
```

```
export class RestauranteService {  
  
    private API = environment.baseUrl;  
  
    private DISTANCIA_API == environment.distanciaUrl;  
    private DISTANCIA_API = environment.baseUrl + '/distancia';  
  
    // código omitido ...  
  
}
```

Exercício: API Gateway com Zuul

1. Em um Terminal, clone o repositório `fj33-api-gateway` para o seu Desktop:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-api-gat
```

No workspace de microservices do Eclipse, importe o projeto `fj33-api-gateway`, usando o menu *File > Import > Existing Maven Projects* e apontando para o diretório `fj33-api-gateway` do Desktop.

2. Execute a classe `ApiGatewayApplication`, certificando-se que os serviços de pagamento e distância estão no ar, assim como o monólito.

Alguns exemplos de URLs:

- `http://localhost:9999/pagamentos/1`
- `http://localhost:9999/distancia/restaurantes/mais-proximos/71503510`
- `http://localhost:9999/restaurantes/1`

Note que as URLs anteriores, apesar de serem invocados no API Gateway, invocam o serviço de pagamento, o de distância e o monólito, respectivamente.

3. Vá até a branch `cap7-ui-chama-api-gateway` do projeto `fj33-eats-ui`:

```
cd ~/Desktop/fj33-eats-ui  
git checkout -f cap7-ui-chama-api-gateway
```

4. Com o monólito, os serviços de pagamentos e distância e o API Gateway no ar, suba o front-end por meio do comando `ng serve`.

Faça um novo pedido e efetue o pagamento. Deve funcionar!

Tente fazer o login como administrador (`admin/123456`) e acessar a página

de restaurantes em aprovação. Deve ocorrer um erro *401 Unauthorized*, que não acontecia antes da UI passar pelo API Gateway. Por que será que acontece esse erro?

Desabilitando a remoção de cabeçalhos sensíveis no Zuul

Na Netflix, o Zuul é usado para Autenticação. Os tokens provenientes da UI são validados e, se o usuário for válido, é repassado para os *downstream services*.

Por padrão, o Zuul remove os cabeçalhos HTTP `Cookie`, `Set-Cookie`, `Authorization`.

Por enquanto, no Caelum Eats, não será feita nenhuma Autenticação no API Gateway.

Por isso, vamos desabilitar essa remoção no `application.properties`:

```
##### fj33-api-gateway/src/main/resources/application.properties
```

Exercício: cabeçalhos sensíveis no Zuul

1. Pare o API Gateway.

Obtenha o código da branch `cap7-cabecalhos-sensiveis-no-zuul` do projeto `fj33-api-gateway`:

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap7-cabecalhos-sensiveis-no-zuul
```

Execute a classe `ApiGatewayApplication`. Zuul no ar!

2. Faça novamente login como administrador (`admin/123456`) e acesse a página de restaurantes em aprovação. Deve funcionar!

API Composition no API Gateway

Após escolher um restaurante, um cliente do Caelum Eats pode ver detalhes do restaurante como o nome, descrição, tipo de cozinha, tempos de entrega, distância ao CEP informado, cardápio e avaliações.

Long Fu

★ 4,1

Chinesa • 40 min - 25 min • 11,7 km

O melhor da China aqui do seu lado.

Cardápio

Avaliações

ENTRADAS

Gyoza Bovino - 6 unidades

Massa fina cozida a vapor recheada com carne temperada com gengibre

R\$ 23,50

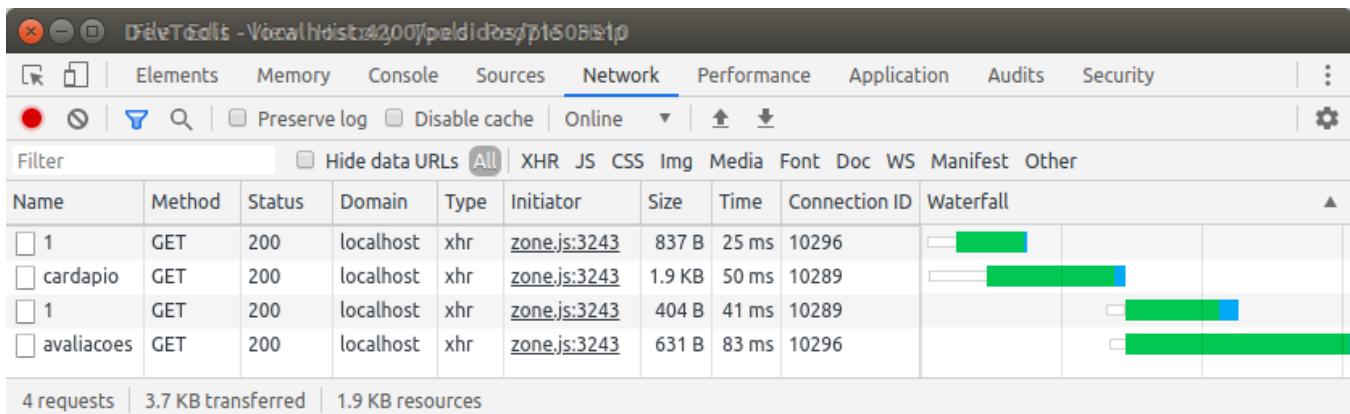
Escolhe

Para obter todos os dados necessários para a tela de detalhes do restaurante, o front-end Angular do Caelum Eats precisa disparar as seguintes chamadas GET via AJAX:

- `http://localhost:9999/restaurantes/1`, para buscar os dados básicos do restaurante
- `http://localhost:9999/restaurantes/1/cardapio`, para buscar os dados do cardápio
- `http://localhost:9999/restaurantes/1/avaliacoes`, para buscar as

avaliações

- `http://localhost:9999/distancia/restaurantes/71503510/restaurantes/1`, para buscar a distância do CEP informado



As três primeiras chamadas mostradas anteriormente são para o Monólito. Poderia ser feita uma otimização no módulo de Restaurante para, dado um id de um restaurante, obter os dados básicos juntamente ao cardápio. A chamada que busca avaliações faz parte do módulo Pedido do Monólito. Se desejássemos manter a separação entre os módulos, facilitando uma posterior extração como serviço, é interessante deixar a busca de avaliações separada da busca dos dados do restaurante.

A quarta chamada é feita ao serviço de Distância, afim de obter a distância do restaurante ao CEP determinado pelo cliente. São buscadas informações, portanto, de outro serviço.

São 4 chamadas ao backend, passando pelo API Gateway, para obter os dados de uma tela.

Uma Arquitetura de Microservices vai levar a uma granularidade mais fina dos serviços. E, se não tomarmos cuidado, essa granularidade terá o efeito de aumentar o número de requisições feitas pelo front-end.

Muitas chamadas ao backend acabam impactando a performance. Cada chamada terá, além da demora do processamento nos serviços, uma latência de rede adicional. Como lembra Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), seria um uso muito ineficiente do plano

de dados de um Mobile.

No post [Optimizing the Netflix API](#) (CHRISTENSEN, 2013), Ben Christensen mostra que essa "tagarelice" (em inglês, *chattiness*) era um problema da Netflix API. Devido à natureza genérica e granular das APIs, cada chamada só retornava uma pequena porção dos dados necessários. As aplicações do Netflix para cada dispositivo (smartphones, TVs, videogames) tinham que realizar diversas chamadas para unir os dados necessários em uma determinada experiência de usuário.

A solução encontrada na Netflix, semelhante a citada por Sam Newman, é unir os vários requests necessários para cada dispositivo em um único request otimizado. O preço da latência da rede WAN (Internet) seria pago apenas uma vez e trocado por chamadas entre os servidores, numa rede LAN de menor latência. Um benefício adicional, além da eficiência, é que as chamadas na rede local são mais estáveis e confiáveis.

Essa ideia de compor APIs de serviços, com granularidade mais fina, em uma API de granularidade maior é chamada de **API Composition**.

API Composition e consultas

Chris Richardson, no livro [Microservice Patterns](#) (RICHARDSON, 2018a), tem uma outra abordagem ao descrever API Composition: o foco é em como fazer consultas em uma Arquitetura de Microservices.

Se, com um BD Monolítico, uma consulta poderia ser feita com um SQL e alguns joins, com Microservices, os dados tem que ser recuperados de diversos BDs de serviços diferentes. Os clientes dos serviços recuperariam os dados pelas APIs do serviços e combinariam os resultados em memória. No fim das contas, trata-se de um *in-memory join*.

Pattern: API Composition

Implemente uma consulta que recupera dados de diversos serviços

por sua API e combina os resultados.

Para Richardson, uma API Composition envolve dois participantes:

- um *API Composer*, que invoca os serviços e combina os resultados
- dois ou mais *Provider services*, que são a fonte dos dados consultados

Richardson discute onde implementar a API Composition:

- no front-end. É o caso que discutimos no Caelum Eats e na Netflix API. Há uma ineficiência no uso da rede, devido à alta latência da Internet.
- no API Gateway. Faz sentido se as consultas fazem parte da API pública da organização. A performance de rede é aumentada, pois as consultas aos *downstream services* são feitas em uma LAN, de menor latência. Deve-se evitar lógica de negócio no API Gateway.
- em um serviço específico para a composição. Útil para agregações invocadas internamente pelos serviços, que não fazem parte da API pública e para casos em que há uma lógica de negócio complexa envolvida na composição. Problemático quando não há um time responsável pelo serviço de composição.

Podemos implementar a API Composition no API Gateway. Temos que mantê-la estritamente técnica, tomando cuidado para que lógica de negócio não acabe vazando para o API Gateway.

Avaliando pontos negativos de API Composition

Chris Richardson ainda discute sobre algumas desvantagens de API Composition como um todo, em qualquer uma das abordagens descritas anteriormente:

- há uma certa ineficiência em invocar n APIs e realizar um *in-memory join*. Mesmo em uma rede interna, a latência vai afetar negativamente a performance da consulta. Pode haver um grande impacto especialmente em datasets maiores.

- há um impacto na disponibilidade, já que as n APIs consultadas precisam estar no ar ao mesmo tempo para que a consulta seja realizada com sucesso.
- há a possibilidade de inconsistência nos dados, já que evitamos transações distribuídas.

Para mitigar algumas dessas desvantagens, Richardson sugere:

- caches, que aumentam a disponibilidade e a performance, mas impactam ainda mais a consistência, com o risco de termos dados desatualizados.
- uma solução que envolve Mensageria que discutiremos mais adiante

API Composition no API Gateway do Caelum Eats

Conforme mencionamos anteriormente, na tela de detalhes do restaurante, a UI faz 4 chamadas AJAX:

- 3 chamadas distintas ao Monólito para buscar os dados básicos do restaurante, o cardápio e as avaliações.
- 1 chamada ao serviço de distância, para buscar a distância do restaurante ao CEP informado.

Implementaremos um API Composition em que o API Gateway buscará em 1 chamada os dados básicos do restaurante junta à distância do restaurante ao CEP.

Poderíamos fazer um API Composition em que 1 chamada obteria todos os dados necessários para a tela de detalhe de restaurantes. Mas isso fica como um desafio!

O API Gateway com o Spring Cloud Netflix Zuul é código feito com Spring Boot.

Então, para implementar essa API Composition criaremos clientes REST que invocam o serviço de Distância e o Monólito. Utilizaremos o RestTemplate do Spring para invocar o serviço de Distância e o Feign

para invocar o Monólito. Seria possível utilizar só o Feign, mas estudaremos a diferença entre o RestTemplate e Feign em integrações com algumas ferramentas do Spring Cloud.

Para expor a API Composition para a UI, utilizaremos um bom e velho `@RestController` do Spring MVC. Para não precisamos criar um *domain model* do serviço de Distância e do Monólito no código do API Gateway, mesclaremos os dados de cada *downstream service* com um simples Map.

Invocando o serviço de distância a partir do API Gateway com RestTemplate

Adicione o Lombok como dependência no `pom.xml` do projeto `api-gateway`:

```
##### fj33-api-gateway/pom.xml
```

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
```

Crie uma classe `RestClientConfig` no pacote `br.com.caelum.apigateway`, que fornece um `RestTemplate` do Spring:

```
##### fj33-api-gateway/src/main/java;br/com/caelum/apigateway/RestClientConfig.java
```

```
@Configuration
class RestClientConfig {

  @Bean
  RestTemplate restTemplate() {
```

```
    return new RestTemplate();
}

}
```

Faça os imports adequados:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;
```

Observação: estamos usando o RestTemplate ao invés do Feign porque estudaremos a diferença entre os dois mais adiante.

Ainda no pacote br.com.caelum.apigateway, crie um @Service chamado DistanciaRestClient que recebe um RestTemplate e o valor de zuul.routes.distancia.url, que contém a URL do serviço de distância.

No método comDistanciaPorCepEId, dispare um GET à URL do serviço de distância que retorna a quilometragem de um restaurante a um dado CEP.

Como queremos apenas mesclar as respostas na API Composition, não precisamos de um *domain model*. Por isso, podemos usar um Map como tipo de retorno.

```
##### fj33-api-
gateway/src/main/java(br/com/caelum/apigateway/DistanciaRestClient.jav
a
```

```
@Service
class DistanciaRestClient {

    private RestTemplate restTemplate;
    private String distanciaServiceUrl;
```

```
DistanciaRestClient(RestTemplate restTemplate,
    @Value("${zuul.routes.distancia.url}") String distanciaUrl
) {
    this.restTemplate = restTemplate;
    this.distanciaServiceUrl = distanciaServiceUrl;
}

Map<String, Object> porCepEId(String cep, Long restauranteId) {
    String url = distanciaServiceUrl + "/restaurantes/" + cep;
    return restTemplate.getForObject(url, Map.class);
}
```

Ajuste os imports:

```
import java.util.Map;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;
```

Observação: é possível resolver o warning de *unchecked conversion* usando um `ParameterizedTypeReference` com o método `exchange` do `RestTemplate`.

Invocando o monólito a partir do API Gateway com Feign

Adicione o Feign como dependência no `pom.xml` do projeto `api-gateway`:

```
##### fj33-api-gateway/pom.xml
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
```

```
</dependency>
```

Na classe `ApiGatewayApplication`, adicione a anotação `@EnableFeignClients`:

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/ApiGatewayApplication
.java
```

```
@EnableFeignClients // adicionado
@EnableZuulProxy
@SpringBootApplication
public class ApiGatewayApplication {

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

}
```

O import a ser adicionado está a seguir:

```
import org.springframework.cloud.openfeign.EnableFeignClients;
```

Crie uma interface `RestauranteRestClient`, que define um método `porId` que recebe um `id` e retorna um `Map`. Anote esse método com as anotações do Spring Web, para que dispare um GET à URL do monólito que detalha um restaurante.

A interface deve ser anotada com `@FeignClient`, apontando para a configuração do monólito no Zuul.

```
@FeignClient("monolito")
```

```
interface RestauranteRestClient {  
  
    @GetMapping("/restaurantes/{id}")  
    Map<String, Object> porId(@PathVariable("id") Long id);  
  
}
```

Ajuste os imports:

```
import java.util.Map;  
  
import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;
```

A configuração do monólito no Zuul precisa ser ligeiramente alterada para que o Feign funcione:

```
##### fj33-api-gateway/src/main/resources/application.properties
```

```
zuul.routes.monolito.uri=http://localhost:8080  
monolito.ribbon.listOfServers=http://localhost:8080
```

Mais adiante estudaremos cuidadosamente o Ribbon.

Compondo chamadas no API Gateway

No api-gateway, crie um `RestauranteComDistanciaController`, que invoca dado um CEP e um id de restaurante obtém:

- os detalhes do restaurante usando `RestauranteRestClient`
- a quilometragem entre o restaurante e o CEP usando `DistanciaRestClient`

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/RestauranteComDista-
nciaController.java

@RestController
@AllArgsConstructor
class RestauranteComDistanciaController {

    private RestauranteRestClient restauranteRestClient;
    private DistanciaRestClient distanciaRestClient;

    @GetMapping("/restaurantes-com-distancia/{cep}/restaurante")
    public Map<String, Object> porCepEIdComDistancia(@PathVariable
                                                       @PathVariable
                                                       String cep,
                                                       String id) {
        Map<String, Object> dadosRestaurante = restauranteRestClient.get(id);
        Map<String, Object> dadosDistancia = distanciaRestClient.get(cep);
        dadosRestaurante.putAll(dadosDistancia);
        return dadosRestaurante;
    }
}
```

Não esqueça dos imports:

```
import java.util.Map;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

import lombok.AllArgsConstructor;
```

Se tentarmos acessar, pelo navegador ou pelo cURL, a URL
`http://localhost:9999/restaurantes-com-`
distancia/71503510/restaurante/1 termos como status da resposta um
401 Unauthorized.

Isso ocorre porque, como o prefixo não é pagamentos nem distancia, a requisição é repassada ao monólito pelo Zuul.

Devemos configurar uma rota no Zuul, usando o forward para o endereço local:

```
##### fj33-api-gateway/src/main/resources/application.properties
```

```
zuul.routes.local.path=/restaurantes-com-distancia/**  
zuul.routes.local.url=forward:/restaurantes-com-distancia
```

A rota acima deve ficar logo antes da rota do monólito, porque esta última é `/**`, um "coringa" que corresponde a qualquer URL solicitada.

Um novo acesso a URL `http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1` terá como resposta um JSON com os dados do restaurante e de distância mesclados.

Chamando a composição do API Gateway a partir da UI

No projeto `eats-ui`, adicione um método que chama a nova URL do API Gateway em `RestauranteService`:

```
##### fj33-eats-ui/src/app/services/restaurante.service.ts
```

```
export class RestauranteService {
```

```
// código omitido ...
```

```
porCepEIdComDistancia(cep: string, restauranteId: string): () =>  
  return this.http.get(`${this.API}/restaurantes-com-distancia/${cep}/${restauranteId}`)
```

```
}
```

Altere ao RestauranteComponent para que chame o novo método porCepEIdComDistancia.

Não será mais necessário invocar o método distanciaPorCepEId, porque o restaurante já terá a distância.

fj33-eats-

ui/src/app/pedido/restaurante/restaurante.component.ts

```
export class RestauranteComponent implements OnInit {  
  
    // código omitido ...  
  
    ngOnInit() {  
  
        this._restaurantesService._porI-d(f restauranteI-d)  
        this.restaurantesService.porCepEIdComDistancia(this.cep, r  
            .subscribe(restaurante => {  
  
                this.restaurante = restaurante;  
                this.pedido.restaurante = restaurante;  
  
                this._restaurantesService._distanciaP-orC-epE-I-  
                    .subscribef restauranteC-omD-istancia-->—{  
                    this._restaurante._distancia=—restauranteC-  
                })  
  
        // código omitido ...  
  
    }  
  
    // restante do código ...  
  
}
```

Ao buscar os restaurantes a partir de um CEP e escolhermos um deles

ou também ao acessar diretamente uma URL como `http://localhost:4200/pedidos/71503510/restaurante/1`, deve ocorrer um *Erro no servidor*.

No Console do navegador, podemos perceber que o erro é relacionado a CORS:

Access to XMLHttpRequest at '<http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1>' from origin '<http://localhost:4200>' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.

Para resolver o erro de CORS, devemos adicionar ao API Gateway uma classe `CorsConfig` semelhante a que temos nos serviços de pagamentos e distância e também no monólito:

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/CorsConfig.java
```

```
@Configuration
class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**").allowedMethods("*").allowCred
    }

}
```

Depois de reiniciar o API Gateway, os detalhes do restaurante devem ser exibidos, assim como sua distância a um CEP informado.

CORS é uma tecnologia do front-end, já que é uma maneira de relaxar a *same origin policy* de chamadas AJAX de um navegador.

Como apenas o API Gateway será chamado diretamente pelo navegador

e não há restrições de chamadas entre servidores Web, podemos apagar as classes `CorsConfig` dos serviços de pagamento e distância, assim como a do módulo `eats-application` do monólito.

Exercício: API Composition no API Gateway

1. Pare o API Gateway.

Faça o checkout da branch `cap7-api-composition-no-api-gateway` do projeto `fj33-api-gateway`:

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap7-api-composition-no-api-gateway
```

Certifique-se que o monólito e o serviço de distância estejam no ar.

Rode novamente a classe `ApiGatewayApplication`.

Tente acessar a URL `http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1`

Deve ser retornado algo parecido com:

```
{  
  "restauranteId":1,  
  "distancia":11.393642891403121808480136678554117679595947265,  
  "cep":"70238500",  
  "descricao":"O melhor da China aqui do seu lado.",  
  "endereco":"ShC/SUL COMERCIO LOCAL QD 404-BL D LJ 17-ASA SUL",  
  "nome":"Long Fu",  
  "taxaDeEntregaEmReais":6.00,  
  "tempoDeEntregaMaximoEmMinutos":25,  
  "tempoDeEntregaMinimoEmMinutos":40,  
  "tipoDeCozinha":{  
    "id":1,  
    "nome":"Chinesa"  
  },
```

```
"id":1  
}
```

-
- Como a UI chama apenas o API Gateway e CORS é uma tecnologia de front-end, devemos remover a classe `CorsConfig` do monólito modular e dos serviços de pagamento e distância. Essa classe já está incluída no código do API Gateway.

```
cd ~/Desktop/fj33-eats-monolito-modular  
git checkout -f cap7-api-composition-no-api-gateway
```

```
cd ~/Desktop/fj33-eats-pagamento-service  
git checkout -f cap7-api-composition-no-api-gateway
```

```
cd ~/Desktop/fj33-eats-distancia-service  
git checkout -f cap7-api-composition-no-api-gateway
```

Reinicie o monólito e os serviços de pagamento e distância.

- Obtenha o código da branch `cap7-api-composition-no-api-gateway` do projeto `fj33-eats-ui`:

```
cd ~/Desktop/fj33-eats-ui  
git checkout -f cap7-api-composition-no-api-gateway
```

Com os serviços de distância e o monólito rodando, inicie o front-end com o comando `ng serve`.

Digite um CEP, busque os restaurantes próximos e escolha algum. Na página de detalhes de um restaurante, chamamos a API Composition. Veja se os dados do restaurante e a distância são exibidos corretamente.

Para saber mais: BFF

Microservices devem ser independentes. O time de desenvolvimento de um microservice deve dominar todo o seu ciclo de vida, da concepção a operação. Como disse o CTO da Amazon, Werner Vogels, em [entrevista a Association for Computing Machinery \(ACM\)](#) (VOEGELS, 2006): "You build it, you run it."

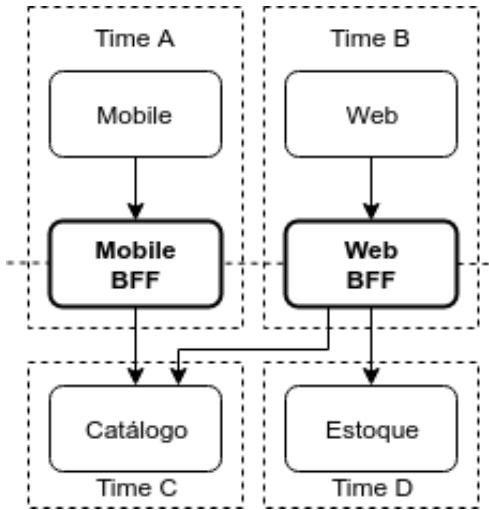
À medida que mais responsabilidades são colocadas no API Gateway e mais API Compositions são implementadas, múltiplos times passam a contribuir com o mesmo código. A independência dos microservices e seus times é afetada.

Para piorar o problema, raramente temos apenas uma UI. Além da Web, podemos ter UIs Mobile e/ou Desktop. Cada UI tem necessidades diferentes. Por exemplo, uma UI Mobile vai ter menos espaço na tela e, portanto, vai exibir menos dados. Também no Mobile, há limitações de bateria e no plano de dados do usuário. E uma UI Mobile tem capacidades extra, como uma câmeras e GPS. Diferentes experiências de usuário requerem diferentes dados dos serviços e, em consequência, da API.

Como resolver isso?

Poderíamos ter um time para a API. Só que esse time teria uma coordenação com cada *downstream service* e também com cada UI.

Uma solução melhor, usada pela SoundCloud e descrita por Phil Calçado no post [The Back-end for Front-end Pattern \(BFF\)](#) (CALÇADO, 2015) é ter um API Gateway com API Compositions específicas para cada front-end. Calçado relata que Nick Fisher, tech lead do time Web da SoundCloud, cunhou o termo *Back-end for Front-end*, ou simplesmente **BFF**.



BFF não é uma tecnologia nem algo que você compra, baixa ou configura. BFF é uma maneira de utilizar um API Gateway mais alinhada com os times de front-end.

Os times de front-end passam a ter a necessidade de um (ou mais) desenvolvedor(es) com habilidades de back-end, que mantém o código que compõe os *downstream services* da maneira adequada para o front-end. Phil Calçado deixa claro no post mencionado anteriormente que o BFF é parte da aplicação e pode ser usado para implementar um *presentation model*, que amplia o *domain model* para incluir dados relevantes apenas para UI como flags e títulos de janelas.

Sam Newman, no post [Backends For Frontends](#) (NEWMAN, 2015b), discute que um detalhe negativo do uso de BFFs é que pode levar a duplicação de código. Mas Newman diz que prefere aceitar um pouco de duplicação a criar abstrações que levem a uma necessidade de coordenação entre serviços. Porém, se a duplicação puder ser modelada em termos de domínio, pode ser extraída para um novo serviço.

Newman ainda sugere que, se o tipo de usuário final for diferente, é interessante termos BFFs separados. No Caelum Eats, por exemplo, se tivermos uma aplicação Web para os clientes e outra diferente para os donos de restaurante, teríamos dois BFFs distintos para cada tipo de usuário.

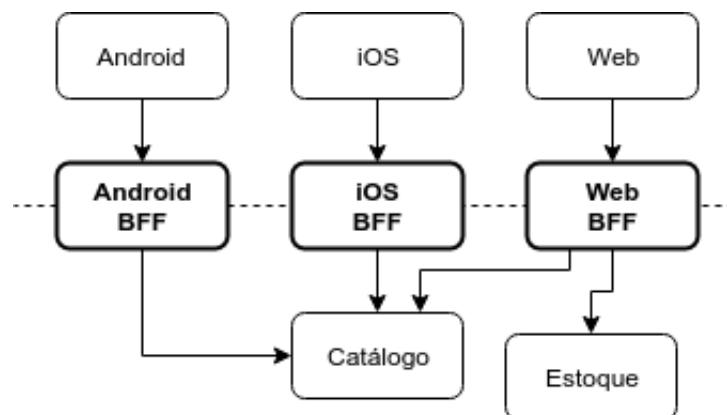
No mesmo post, Newman diz que um BFF pode ser usado como cache para resultados de API Composition e para server-side rendering.

Um BFF por plataforma?

Aplicações de diferentes plataformas Mobile, como Android e iOS, devem utilizar o mesmo BFF ou não?

Isso é discutido por Sam Newman no post [Backends For Frontends](#) (NEWMAN, 2015b). Para Newman, há algumas motivações para termos BFFs distintos para cada plataforma:

- se as experiências de usuário do Android e iOS são suficientemente distintas.
- se há um time distinto para as aplicações Android e iOS.



Na palestra [Evoluindo uma Arquitetura inteiramente sobre APIs](#) (CALÇADO, 2013), menciona uma outra motivação para BFFs separados por plataforma: uma app iOS é publicada na Apple Store em questão de semanas ou meses, enquanto uma app Android é publicada na Play Store em questão de horas. Se tivermos apenas um BFF, teremos que manter APIs compatíveis com ambas as versões das apps. Já com um BFF, há maior controle, ainda que haja alguma duplicação.

Zuul Filters

A descrição da arquitetura do Zuul, feita no artigo [Announcing Zuul: Edge Service in the Cloud](#) (COHEN; HAWTHORNE, 2013), revela que o Zuul é um simples proxy reverso que pode ser usado com filtros feitos em alguma linguagem da JVM. Os Zuul Filters podem realizar diferentes ações durante o roteamento de requests e responses HTTP como:

roteamento dinâmico, *load balancing*, *rate limiting*, *encoding* para devices específicos, *failover*, autenticação, etc.

Os Zuul Filters possuem as seguintes características:

- **Type**: em geral, a fase do roteamento em que o filtro será aplicado. Pode conter tipos customizados.
- **Execution Order**: define a prioridade do filtro. Quanto menor, mais prioritário.
- **Criteria**: define os critérios necessários para a execução do filtro.
- **Action**: as ações a serem efetuadas pelo filtro.

Os Zuul já contém alguns Type padronizados para os Zuul Filters:

- `pre`: executados antes do roteamento. Pode ser usado para autenticação e load balancing, por exemplo.
- `routing`: executados durante o roteamento.
- `post`: executados depois que o roteamento foi feito. Pode ser usado para manipular cabeçalhos HTTP e registrar estatísticas, por exemplo.
- `error`: executados no caso de um erro em outras fases.

Um exemplo de um filtro implementado em Java, que atrasa em 20 segundos o roteamento de requests vindos de um Master System:

```
class DeviceDelayFilter extends ZuulFilter {  
  
    @Override  
    public String filterType() {  
        return "pre";  
    }  
  
    @Override  
    public int filterOrder() {  
        return 5;  
    }  
}
```

```
@Override  
public boolean shouldFilter() {  
    String deviceType = RequestContext.getRequest().getPara("deviceType");  
    return deviceType != null && deviceType.equals("Master")  
}  
  
@Override  
public Object run() {  
    Thread.sleep(20000);  
}  
}
```

Filtros dinâmicos podem ser implementados em Groovy e são lidos do disco, compilados e executados dinamicamente no próximo request roteado. Um diretório do servidor é consultado periodicamente para obter mudanças ou filtros adicionais.

De acordo com a [documentação](#) do Spring Cloud Netflix Zuul, já há uma série de Zuul Filters implementados em Java.

Há alguns `pre filters`, como:

- `ServletDetectionFilter`: detecta se a request passou pela Servlet Dispatcher do Spring.
- `FormBodyWrapperFilter`: faz o parse de dados de forms.
- `DebugFilter`: se `debug` estiver setado, seta outras propriedades de debug do routing e do request.
- `PreDecorationFilter`: usa o `RouteLocator` para determinar onde e como fazer o roteamento e seta cabeçalhos relacionados a proxy como `x-Forwarded-Host` e `x-Forwarded-Port`.

Há alguns `route filters`:

- `SendForwardFilter`: repassa o resultado do response roteado para o response original.
- `SimpleHostRoutingFilter`: envia os requests para a URL configurada

na propriedade `routeHost` do `RequestContext`.

- `RibbonRoutingFilter`: usa ferramentas que veremos posteriormente para fazer roteamento dinâmico e balanceamento de carga.

Há um `error filter`:

- `SendForwardFilter`: faz o forward de qualquer exceção para `/error`.

LocationRewriteFilter no Zuul para além de redirecionamentos

Ao usar um API Gateway como Proxy, precisamos ficar atentos a URLs retornadas nos payloads e cabeçalhos HTTP.

O cabeçalho `Location` é comumente utilizado por redirects (status 301 `Moved Permanently`, 302 `Found`, entre outros). Esse cabeçalho contém um novo endereço que o cliente HTTP, em geral um navegador, tem que acessar logo em seguida.

Esse cabeçalho `Location` também é utilizado, por exemplo, quando um novo recurso é criado no servidor (status 201 `Created`).

O Spring Cloud Netflix Zuul tem um Filter padrão, o `LocationRewriteFilter`, que reescreve as URLs, colocando no `Location` o endereço do próprio Zuul, ao invés de manter o endereço do serviço.

Porém, esse Filter só funciona para redirecionamentos (3xx) e não para outros status como 2xx.

Por exemplo, vamos criar um novo pagamento usando o Zuul como Proxy:

```
curl -X POST -i -H 'Content-Type: application/json'  
-d '{"va51.8, "nome": "JOÃO DA SILVA", "numero": "1111 2222 3  
http://localhost:9999/pagamentos
```

O response, incluindo cabeçalhos, será semelhante a:

```
HTTP/1.1 201
Location: http://localhost:8081/pagamentos/2
Date: Mon, 06 Jan 2020 17:47:56 GMT
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
```

```
{"id":2, "valor":51.8, "nome":"JOÃO DA SILVA", "numero":"1111
"expiracao":"2022-07", "codigo":"123", "status":"CRIADO",
"formaDePagamentoId":2, "pedidoId":1}
```

Perceba que, apesar de invocarmos o serviço de pagamentos pelo Zuul, o cabeçalho `Location` contém a porta 8081, do serviço original, na URL:

```
Location: http://localhost:8081/pagamentos/2
```

Vamos customizá-lo, para que funcione com respostas bem sucedidas, de status 2xx.

O código do `LocationRewriteFilter` do Spring Cloud Netflix Zuul pode ser encontrado em: <http://bit.ly/spring-cloud-location-rewrite-filter>

Para isso, crie uma classe `LocationRewriteConfig` no pacote `br.com.caelum.apigateway`, definindo uma subclasse anônima de `LocationRewriteFilter`, modificando alguns detalhes.

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/LocationRewriteConfig
.java
```

```
@Configuration
```

```
class LocationRewriteConfig {  
  
    @Bean  
    LocationRewriteFilter locationRewriteFilter() {  
        return new LocationRewriteFilter() {  
            @Override  
            public boolean shouldFilter() {  
                int statusCode = RequestContext.getCurrentContext().get...  
                return HttpStatus.valueOf(statusCode).is3xxRedirection...  
            }  
        };  
    }  
  
}  
  
-----
```

Tome bastante cuidado com os imports:

```
import org.springframework.cloud.netflix.zuul.filters.post.LocationRewriteFilter;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.http.HttpStatus;  
  
import com.netflix.zuul.context.RequestContext;
```

Agora sim! Ao receber um status 201 `Created`, depois de criar algum recurso em um serviço, o API Gateway terá o `Location` dele próprio, e não do serviço original.

Exercício: Customizando o `LocationRewriteFilter` do Zuul

1. Através de um cliente REST, tente adicionar um pagamento passando pelo API Gateway. Para isso, utilize a porta 9999.

Com o cURL é algo como:

```
curl -X POST  
-i  
-H 'Content-Type: application/json'  
-d '{ "valor": 51.8, "nome": "JOÃO DA SILVA", "numero": "1111-1111-1111-1111", "cartao": "4325123456789012", "validade": "12/2025", "cvv": "123" }'  
http://localhost:9999/pagamentos
```

Lembrando que um comando semelhante ao anterior, mas com a porta 8081, está disponível em: <https://gitlab.com/snippets/1859389>

Note no cabeçalho `Location` do response que, mesmo utilizando a porta 9999 na requisição, a porta da resposta é a 8081.

```
Location: http://localhost:8081/pagamentos/40
```

2. Pare o API Gateway.

No projeto `fj33-api-gateway`, faça o checkout da branch `cap7-customizando-location-filter-do-zuul`:

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap7-customizando-location-filter-do-zuul
```

Execute a classe `ApiGatewayApplication`.

3. Teste novamente a criação de um pagamento com um cliente REST. Perceba que o cabeçalho `Location` agora tem a porta 9999, do API Gateway.
4. (desafio - opcional) Se você fez os exercícios opcionais de Spring HATEOAS, note que as URLs dos links ainda contém a porta 8081. Implemente um Filter do Zuul que modifique as URLs do corpo de um `response` para que apontem para a porta 9999, do API Gateway.

Exercício opcional: um ZuulFilter de Rate Limiting

1. Adicione, no pom.xml de api-gateway, uma dependência a biblioteca Google Guava:

```
##### fj33-api-gateway/pom.xml
```

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>15.0</version>
</dependency>
```

A biblioteca Google Guava possui uma implementação de *rate limiter*, que restringe o acesso a recurso em uma determinada taxa configurável.

2. Crie um ZuulFilter que retorna uma falha com status 429 TOO MANY REQUESTS se a taxa de acesso ultrapassar 1 requisição a cada 30 segundos:

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/RateLimitingZuulFilter.j
ava
```

```
@Component
public class RateLimitingZuulFilter extends ZuulFilter {

    private final RateLimiter rateLimiter = RateLimiter.create(1

    @Override
    public String filterType() {
        return FilterConstants.PRE_TYPE;
    }

    @Override
    public int filterOrder() {
```

```
    return Ordered.HIGHEST_PRECEDENCE + 100;
}

@Override
public boolean shouldFilter() {
    return true;
}

@Override
public Object run() {
    try {
        RequestContext currentContext = RequestContext.getCurrentContext();
        HttpServletResponse response = currentContext.getResponse();

        if (!this.rateLimiter.tryAcquire()) {
            response.setStatus(HttpStatus.TOO_MANY_REQUESTS.value());
            response.getWriter().append(HttpStatus.TOO_MANY_REQUESTS.getReasonPhrase());
            currentContext.setSendZuulResponse(false);
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    return null;
}
}
```

Os imports são os seguintes:

```
import java.io.IOException;

import javax.servlet.http.HttpServletResponse;

import org.springframework.cloud.netflix.zuul.filters.support.*;
import org.springframework.core.Ordered;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;

import com.google.common.util.concurrent.RateLimiter;
```

```
import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
```

3. Garanta que `ApiGatewayApplication` foi reiniciado e acesse alguma várias vezes seguidas pelo navegador, uma URL como `http://localhost:9999/restaurantes/1`.

Deve ocorrer um erro 429 Too Many Requests.

4. Apague (ou desabilite comentando a anotação `@Component`) a classe `RateLimitingZuulFilter` para que não cause erros na aplicação no restante do curso.

Para saber mais: Micro front-ends

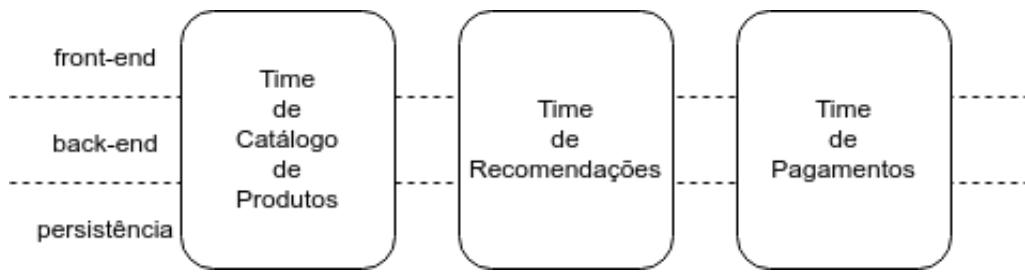
Quebramos o código do back-end em alguns serviços independentes e alinhados com verticais de negócio. Mas e o front-end? Continua monolítico!

Toda mudança relevante no back-end, em qualquer serviço (ou no monólito), traria a necessidade de uma mudança no front-end. Todo deploy relevante do back-end, iria requerer um deploy no front-end.

Idealmente, uma Arquitetura de Microservices, teria times independentes, multidisciplinares e orientados pelo domínio, cuidando da concepção à operação do software. E isso deveria ser realmente ponta a ponta (em inglês, *end-to-end*), incluindo a UI.

Michael Geers criou, em 2017, o site micro-frontends.org (GEERS, 2017), em que descreve uma abordagem que aplica os princípios de Microservices em websites e webapps.

A ideia de Micro Front-ends é que os times cuidem de um aspecto de negócio específico de ponta a ponta, incluindo concepção, front-end, back-end e operações.



Com Micro Front-ends, uma página é composta por componentes de diferentes times.

No caso de sites estáticos, a composição da página pode ser feita no lado do servidor.

Já para single-page apps, a composição deve ser feita no próprio navegador, com código JavaScript.

No site microservices.io, mantido por Chris Richardson, autor do livro [Microservice Patterns](#) (RICHARDSON, 2018a), a composição no servidor é chamada de *Server-side page fragment composition* e no navegador é chamada de *Client-side UI composition*.

Navegadores modernos são compatíveis com [Web Components](#), um conjunto de especificações que permitem a criação de componentes customizados como, por exemplo, <lista-de-recomendacoes>. As principais especificações são: [Custom Elements](#), [Shadow DOM](#), [ES Modules](#) e [HTML Template](#). É uma capacidade oferecida por frameworks como Angular, React e Vue, mas com APIs do próprio navegador. Dessa maneira, esses diferentes frameworks podem ter Web Componentes como destino do build.

Time de Catálogo de Produtos

Kubernetes

Tudo sobre orquestração de containers

Lucas Santos

Ebook
.pdf, .epub e .mobi

R\$ 29,90

Time de Pagamentos

Comprar

Time de Recomendações

Livros Relacionados

Michael Geers declara alguns princípios de Micro Front-ends em seu site:

- **Independência de Tecnologia:** cada time deve escolher suas tecnologias com o mínimo de coordenação com outros times. Uma tecnologia como os Web Components, mencionados anteriormente, ajudam a implementar esse princípio.
- **Código Isolado:** evite variáveis globais e estado compartilhado, mesmo se os times utilizarem o mesmo framework.
- **Uso de Prefixos:** onde o isolamento não é possível, como CSS, Local Storage, Cookies e eventos, estabeleça prefixos e convenções de nomes para evitar colisões e deixar claro qual time cuida de qual componente.
- **Uso de Recursos Nativos:** prefira recursos nativos do navegador a APIs customizadas. Por exemplo, para comunicação entre componentes, use [DOM Events](#) ao invés de um mecanismo específico de um framework.
- **Resiliência:** construa a aplicação de maneira que ainda seja útil no caso de uma demora ou uma falha no JavaScript. Conceitos como Universal Rendering e Progressive Enhancement ajudam nesse

cenário.

Discussão: ESB x API Gateway

No livro [SOA in Practice](#) (JOSUTTIS, 2007), Nicholai Josuttis afirma que, entre as responsabilidades de um Enterprise Service Bus (ESB), estão: interoperabilidade entre diferentes plataformas e linguagens de programação, incluindo a tradução de protocolos; transformação de dados; roteamento inteligente; segurança; confiabilidade; gerenciamento de serviços; monitoramento e logging; e orquestração.

Como mencionado anteriormente, o time de Tecnologia da Netflix, no post [Announcing Zuul: Edge Service in the Cloud](#) (COHEN; HAWTHORNE, 2013), afirma que o API Gateway Zuul é usado para: roteamento dinâmico; segurança; insights (ou seja, monitoramento); entre outras responsabilidades. Além disso, vimos durante o capítulo que podemos usar um API Gateway como um API Composer.

Há certa semelhança entre as responsabilidades de um ESB e de um API Gateway. Seria um API Gateway uma reencarnação do ESB para uma Arquitetura de Microservices?

Uma diferença importante é a **topologia de rede**.

Em geral, um ESB é implantado como um ponto central de comunicação, em uma topologia conhecida como *hub and spoke*, de maneira a evitar uma comunicação ponto a ponto. Assim, os serviços não precisam conhecer os protocolos e endereços uns dos outros, mas apenas os do ESB. Numa implantação *hub and spoke*, a ideia é que qualquer comunicação entre serviços seja feita pelo ESB, levando a um baixo acoplamento.

Já um API Gateway é um proxy reverso, implantado no perímetro da rede. Chamadas de aplicações externas passam pelo API Gateway, porém as chamadas entre os serviços são feitas diretamente, em uma comunicação ponto a ponto. Isso poderia levar a um alto acoplamento

entre os serviços, mas nos próximos capítulos veremos maneiras de mitigar esse risco.

Uma outra diferença é relacionada a **presença ou não de regras de negócio**.

No SOA clássico, há a ideia de que serviços podem ser compostos em outros serviços formando novos processos de negócio. O ESB passa a coordenar a execução de diversos serviços de acordo com uma lógica de negócio. O ESB funciona como um maestro que coordena os diferentes músicos (os serviços) de uma orquestra, num processo conhecido como Orquestração. Tecnologias como BPEL ajudam nessa tarefa. Isso pode levar a necessidade de coordenação no desenvolvimento e deploy de diversos serviços.

Autonomia é um dos conceitos mais importantes de uma Arquitetura de Microservices. A centralização de uma Orquestração dá lugar à descentralização de uma Coreografia, em que os diferentes serviços reagem em conjunto a eventos de negócio. Assim, não há um coordenador ou maestro e, em consequência, não há a necessidade de colocar regras de negócio fora dos serviços. Canais de comunicação devem ser "ignorantes". Martin Fowler e James Lewis, em seu seminal artigo [Microservices](#) (FOWLER; LEWIS, 2014), cunharam um lema nesse sentido: *smart endpoints, dumb pipes*.

Para seguir o lema *smart endpoints, dumb pipes* e favorecer a autonomia dos serviços, um API Gateway não deveria conter regras de negócio. Um API Gateway deveria ajudar na implementação de requisitos transversais (também chamados de não-funcionais): segurança, resiliência, escalabilidade, entre diversos outros.

Em uma [thread no Twitter de Maio de 2019](#) (NEWMAN, 2019b), Sam Newman critica o uso do API Gateway como orquestrador de regras de negócio:

As pessoas [no SOA clássico] fariam a orquestração de processos de

negócios nas camadas do message broker. Agora é comum ver pessoas fazendo o mesmo nos API Gateways, que estão rapidamente se tornando o Enterprise Service Bus dos Microservices.

A inserção da lógica de negócios no middleware é problemática do ponto de vista da implantação, pois é necessário coordenar o rollout, mas também em termos de controle, pois as pessoas que construíram o aplicativo geralmente eram diferentes das pessoas que gerenciavam alterações no middleware.

E as nossas API Compositions não seriam, no fim das contas, colocar regras de negócio no API Gateway? Não, já que evitamos processar os dados, usando a API Composition apenas como agregação de chamadas remotas, visando a otimização das chamadas da UI externa.

Load Balancing

Escalabilidade

Em um dia como a Black Friday, é comum um aumento drástico nos acessos a lojas online como a Casa do Código. No caso de um desastre natural, há um incremento vigoroso no uso de redes sociais como o Twitter. Como esses softwares conseguem lidar com essa demanda anormal?

Além das funcionalidades acessadas pelos usuários, um software possui uma série de características operacionais transversais, que não estão relacionadas a uma capacidade de negócio específica. São características comuns a todo bom software, geralmente chamadas de requisitos não-funcionais ou atributos de qualidade. São requisitos relevantes para a Arquitetura de Software como, por exemplo: segurança, resiliência, performance, usabilidade, manutenibilidade, disponibilidade, escalabilidade, entre vários outros. Também são chamados de *-ilidades*, já que esse é o sufixo de boa parte desses requisitos.

Uma dessas *ilidades* é a característica de um sistema de conseguir lidar com esse aumento vigoroso no uso: a **Escalabilidade**.

Um sistema pode ter como característica a **Escalabilidade Vertical**, em que o aumento na capacidade computacional se deve a uma máquina mais poderosa: mais memória, mais CPUs e mais disco.

Já um que apresenta **Escalabilidade Horizontal** aumenta seu poder computacional adicionando mais máquinas *commodity* a um Sistema Distribuído. Em uma Arquitetura de Microservices, significa adicionar **redundância**, replicando o código de um mesmo serviço em mais de uma máquina.

Redundância não é suficiente. Vários outras características são necessárias para que um serviço possa ser escalado horizontalmente.

Por exemplo, um serviço *Stateless* favorece bastante a Escalabilidade Horizontal. Lembrando a definição de Roy Fielding sua tese [Architectural Styles and the Design of Network-based Software Architectures](#) (FIELDING, 2000): uma sistema é *Stateless* quando cada request do cliente deve conter todos os dados necessários, sem tomar vantagem de nenhum contexto armazenado no servidor

Em sistemas executados em alguma plataforma de Cloud Computing, manter recursos alocados sem utilização custa caro. Por isso, é importante que o sistema possa aumentar e *diminuir* a sua capacidade de maneira automática. Essa característica é conhecida como *Elasticidade*.

Será que um Monólito ser escalável horizontalmente? Pode sim, se for construído de maneira que o código do projeto possa ser replicado em múltiplas máquinas.

Disponibilidade

O pesquisador da Google Jeffrey Dean, na palestra [Designs, Lessons and Advice from Building Large Distributed Systems](#) (DEAN, 2009), lista alguns problemas de hardware que ocorrem no primeiro ano de um novo cluster:

- 3 falhas em roteadores, tirando o cluster do ar por 1 hora
- 8 manutenções na rede, causando 30 minutos de perdas de conexões
- 12 reloads em roteadores, derrubando o DNS por alguns minutos
- 20 falhas em racks, matando dezenas de máquinas e levando
- milhares de falhas em discos rígidos

A principal coisa que notei, no entanto, foi que os discos rígidos foram conectados por velcro. Perguntei a um dos Googlers o motivo disso. 'Ah', disse ele, 'os discos rígidos falham tanto que não queremos que eles sejam parafusados. Nós apenas os arrancamos, jogamos na lixeira e conectamos um novo com velcro.'

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015)

Há diversas outras fontes de erro, como mudança de configurações de Firewall, alterações no DNS ainda não propagadas, atualizações em serviços externos para versões não compatíveis, falhas em um Sistema Operacional e, claro, bugs involuntariamente adicionados pelos desenvolvedores de um serviço.

Fica claro que, no mundo real, um Sistema Distribuído tem diversos motivos para ficar fora do ar.

Estar operando normalmente é algo relacionado a outra *ilidade*: a **Disponibilidade**. Trata-se da proporção de tempo em que um sistema se mantém em operação.

A Disponibilidade está relacionada a um período de tempo: diário, semanal, mensal ou anual. O tempo em que um sistema passa operando normalmente nesse período de tempo é chamado de *uptime*. Já o tempo em que um sistema falha em completar requisições nesse período de tempo é o *downtime*. Portanto:

$$\text{Disponibilidade} = \text{uptime} / (\text{uptime} + \text{downtime})$$

Algumas medidas de Disponibilidade comuns:

- um sistema que fica fora do ar durante 3,65 dias por ano tem Disponibilidade anual de 99 %, conhecida como *two nines*
- um sistema que fica fora do ar durante 8,77 horas por ano tem Disponibilidade anual de 99.9 %, conhecida como *three nines*
- um sistema que fica fora do ar durante 5,26 minutos por ano tem Disponibilidade anual de 99,999 %, conhecida como *five nines*

Mais adiante na apostila vamos revisitar o conceito de Disponibilidade.

Em um Sistema Distribuído, manter a Disponibilidade passa ser mais

difícil. Para estar disponível, um Sistema Distribuído com 3 serviços distintos tem que ter todos no ar ao mesmo tempo. Se a Disponibilidade de cada serviço for 99 %, a Disponibilidade do todo é de 99³ %, ou **97 %**. Ou seja, a disponibilidade geral é menor do que a disponibilidade de cada serviço!

O que podemos fazer para ajudar a aumentar a Disponibilidade de um Sistema Distribuído?

Devemos eliminar pontos únicos de falha (em inglês, *Single Points of Failure*) adicionando **redundância**, replicando o código de um mesmo serviço em mais de uma máquina. Assim, se uma das máquinas falharem, teremos outras que tratarão as requisições ao serviço.

Redundância por si só não é suficiente para garantir Alta Disponibilidade de um Sistema Distribuído. De maneira semelhante à Escalabilidade, há outras características necessárias. Ser *Stateless*, por exemplo, facilita a redundância, favorecendo a Disponibilidade.

Load Balancing

Perceba que, tanto para atingir Escalabilidade Horizontal como para conseguirmos Alta Disponibilidade, devemos usar redundância, replicando os mesmos serviços em diferentes instâncias.

Surge um problema: qual das instâncias deve ser chamada por clientes externos, como uma UI Web? E, em uma chamada entre serviços, qual instância deve ser invocada?

Precisamos fazer um **Balanceamento de Carga** (em inglês, *Load Balancing*), distribuindo as requisições entre as instâncias disponíveis de um serviço.

Podemos implementar o Load Balancing de duas maneiras diferentes: do lado de quem provê as instâncias (servidor) e do lado de quem chama as instâncias (cliente).

Um detalhe importante: se um *load balancer* estiver fora do ar, todas as instâncias de um serviço ficarão inatingíveis. Para que não seja um *Single Point of Failure*, é necessário que haja redundância do próprio load balancer.

Load Balancers que usam dados de protocolos no nível de rede ou transporte como IP, TCP, FTP e UDP para fazer o balanceamento são chamados de *L4 load balancers*, já que vão até o nível 4 (Transporte) do Modelo OSI de redes computacionais. Já quando um load balancer lida dados de protocolos do nível de aplicação, como cabeçalhos do HTTP, são chamados de *L7 load balancers*, correspondendo ao nível 7 (Aplicação) do Modelo OSI.

Sistemas escaláveis horizontalmente atingem disponibilidade e escalabilidade por meio da multiplicidade. A adição de mais máquinas para aumentar a capacidade melhora a resiliência a impulsos (choques agudos e curtos no sistema). Os servidores menores usados em arquiteturas escaláveis horizontalmente também costumam muito menos e permitem aumentar a capacidade em pequenos incrementos.

A criação de sistemas escaláveis horizontalmente implica em alguma forma de balanceamento de carga. O balanceamento de carga tem tudo a ver com a distribuição de solicitações em um pool ou farm de servidores para atender a todas as requisições corretamente no menor tempo possível.

Michael Nygard, em seu livro [Release It! Second Edition](#) (NYGARD, 2018)

Server Side Load Balancing

A maneira mais comum de implementar Load Balancing é do lado do servidor: o **Server Side Load Balancing**.

Esses server side load balancers podem ser hardware, como as *appliances* da

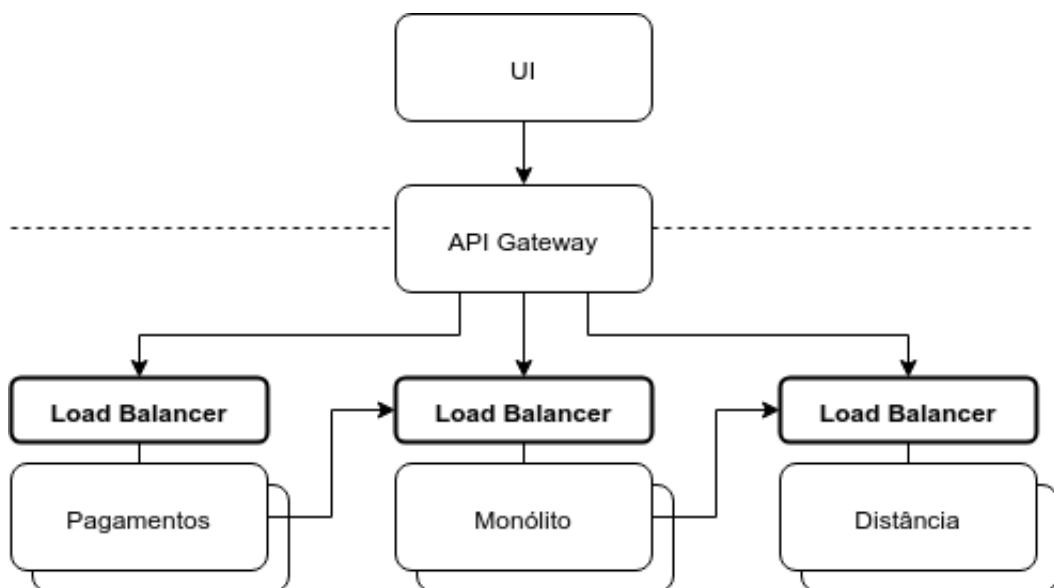
Cisco ou F5. Em geral, são L4 load balancers mas também podem ser L7. São equipamentos caríssimos.

É bem comum termos softwares que funcionam como server side load balancers. Em geral, são L7 load balancers. Proxys reversos como Squid, Apache httpd, NGInx e HAProxy são comumente usados para Load Balancing.

Plataformas de Cloud Computing tem load balancers já integrados com as outras soluções oferecidas. A Amazon AWS, por exemplo, tem o AWS Elastic Load Balancer (ELB), que está integrado com AWS EC2 e AWS Lambda, por exemplo. É possível escolher entre L4 e L7 load balancers.

Em termos de topologia de rede, um load balancer deve ser colocado na frente de cada grupo de instâncias de um determinado serviço.

No caso do Caelum Eats, isso significaria que precisaríamos de um load balancer em frente do Monólito, outro em frente do serviço de distância e outro em frente do serviço de pagamentos. Toda chamada do API Gateway a esses serviços, assim como toda chamada entre esses serviços deveria passar pelos respectivos load balancers, para que a carga seja distribuída entre as instâncias dos serviços.



No orquestrador de containers [Kubernetes](#), as instâncias de uma aplicação são chamadas de [Pods](#). O Kubernetes já provê uma abstração

para o agrupamento dos diferentes Pods de uma aplicação, chamada de [Service](#). Um Service funciona como um server side load balancer, expondo os Pods de uma aplicação sob um mesmo DNS name.

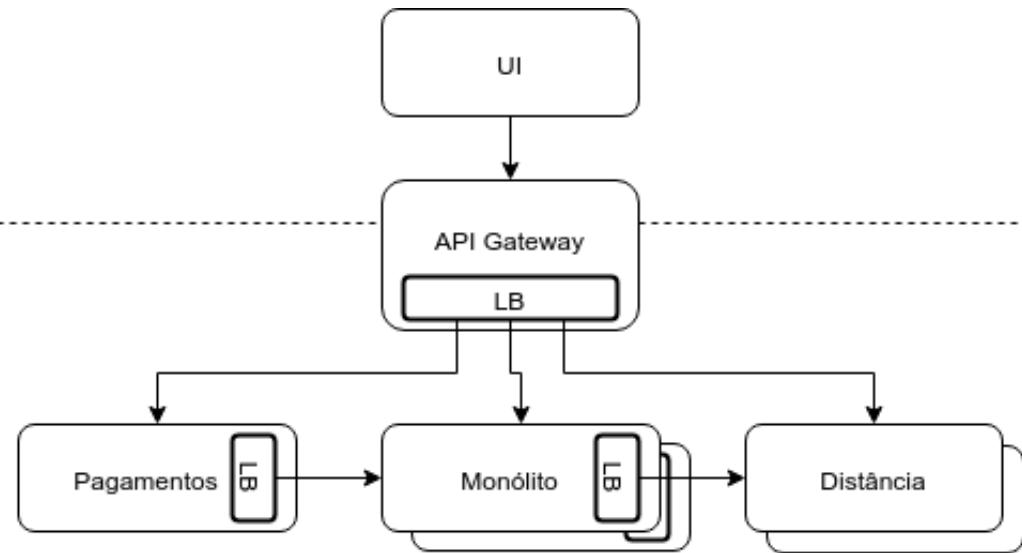
No livro [Release It! Second Edition](#) (NYGARD, 2018), Michael Nygard menciona que o próprio DNS pode ser usado como um L7 load balancer: é possível associar múltiplos IPs a um mesmo domínio. Ao contrário de outros load balancers, o balanceamento não é feito durante uma requisição, mas na resolução de nomes. É feito um balanceamento do tipo *round-robin*, em que os IPs oferecidos ao cliente são rotacionados sequencialmente. Uma grande desvantagem do DNS é que os IPs devem ser acessíveis diretamente. Além disso, não há uma maneira de modificar o algoritmo de balanceamento e, uma vez que o cliente esteja conectado a um IP, não há como redirecionar para outras instâncias.

Client Side Load Balancing

Uma alternativa é que um componente dentro da própria aplicação sirva como load balancer: é o que chamamos de **Client Side Load Balancing**.

Esse componente da aplicação precisa, de alguma forma, saber quais instâncias estão disponíveis para quais serviços. Isso pode ser feito através de configurações.

A topologia de rede é drasticamente simplificada, já que o load balancer não é externo à aplicação. Além disso, cada instância terá o mesmo componente com a mesma configuração, ou seja, já há redundância do load balancer!



Ribbon

A Netflix lançou como parte de sua iniciativa open-source o [Ribbon](#), um Client Side Load Balancer integrado com outras ferramentas do Netflix OSS.

Uma das necessidades do Ribbon é saber o IP ou DNS name das instâncias de um determinado serviço. Isso pode ser feito por uma configuração estática (`ConfigurationBasedServerList`), ou de maneira dinâmica, de modo a remover, de tempos em tempos, instâncias indisponíveis e adicionar novas instâncias (`DiscoveryEnabledNIWSServerList`). Nesse capítulo, focaremos na configuração estática. Em um capítulo posterior, estudaremos a configuração dinâmica da lista de instâncias.

O Ribbon possui diferentes *rules*, que são lógicas de escolha de uma instância da lista. Entre elas:

- `RoundRobinRule`: usa o algoritmo *round robin*, que rotaciona a lista de instâncias, alternando sequencialmente entre as instâncias disponíveis. É o algoritmo padrão. Se tivermos 3 instâncias, as chamadas seriam à primeira, à segunda e à terceira, e então à primeira novamente, e assim por diante.
- `AvailabilityFilteringRule`: um algoritmo que pula as instâncias indisponíveis. Por padrão, uma instância é considerada indisponível

se há falha em 3 conexões consecutivas. Depois de 30 segundos, há nova tentativa de conexão. Se houver falha nas novas tentativas, há um aumento exponencial do tempo de espera. Todos esses valores são configuráveis.

- `WeightedResponseTimeRule`: é coletado o tempo de resposta de cada instância. Quanto maior o tempo de resposta, menor a probabilidade da instância ser obtida da lista.

Além disso, o Ribbon mantém estatísticas da latência e frequência de falha de cada instância.

Também é possível agrupar clientes e servidores em zonas (equivalente a *data centers*), mantendo chamadas numa mesma zona de maneira a minimizar a latência.

É possível obter detalhes sobre essas configurações no Wiki de documentação do Ribbon no GitHub:

<https://github.com/Netflix/ribbon/wiki/Working-with-load-balancers>

Em Abril de 2016, o time da Netflix responsável pelo Ribbon informou que o projeto está em [modo de manutenção](#) e novas funcionalidades não serão adicionadas. Afirmaram também que o projeto é bastante estável e que as partes principais são utilizadas em produção. Também disseram que os planos são migrar para uma solução baseada em gRPC, com interceptadores customizados para load balancing e service discovery.

Spring Cloud Netflix Ribbon

O projeto Spring Cloud Netflix Ribbon integra o Ribbon com o ecossistema do Spring.

Para utilizá-lo em um projeto Spring Boot, basta adicionar como dependência o artefato `spring-cloud-starter-netflix-ribbon`. Com o Maven, basta declarar:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>
```

Observação: para que o Maven obtenha as versões corretas do Spring Cloud e das dependências transitivas, o artefato `spring-cloud-dependencies` deve ser adicionado ao Dependency Management com tipo `pom`. Faremos isso nos próximos exercícios para os projetos que ainda não possuem essa declaração.

Digamos que temos duas instâncias de um serviço de Estoque sendo executadas nas portas 8089 e 9099.

Uma chamada à instância de porta 8089 para obter o item de estoque de `id` igual a 1 seria feita com a URL:

`http://localhost:8089/item/1`

Já para a outra instância, teríamos:

`http://localhost:9099/item/1`

Com o Spring Cloud Netflix Ribbon, poderíamos ter a seguinte configuração no `application.properties` do projeto:

`estoque.ribbon.listOfServers=http://localhost:8089,http://loc`

Com a configuração anterior, o nome `estoque` está associado pelo Ribbon à lista com as duas instâncias do serviço. É como se fosse criado um Virtual Host chamado `estoque`. Como o algoritmo padrão é o *round robin*,

chamadas a `http://estoque` que passam pelo Ribbon seriam alternadas entre as duas instâncias.

Uma outra configuração que devemos fazer por enquanto é desabilitar o Eureka, que está integrado ao Ribbon mas estudaremos em capítulos posteriores:

```
ribbon.eureka.enabled=false
```

Integrando com RestTemplate

Então, poderíamos usar a seguinte URL em uma chamada com `RestTemplate` do Spring:

```
String url = "http://estoque/item/1";
Map dados = restTemplate.getForObject(url, Map.class);
```

Porém, ao executarmos esse código, teríamos uma exceção semelhante a:

```
java.net.UnknownHostException: estoque
```

O `RestTemplate` não entende o que significa `estoque`. Na verdade, a chamada não foi interceptada pelo Ribbon.

Para associar o Ribbon ao `RestTemplate`, devemos utilizar a anotação `@LoadBalanced` no momento da criação da instância do `RestTemplate`:

```
@Configuration
public class RestClientConfig {

    @LoadBalanced // adicionado
```

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

A anotação `@LoadBalanced` é do pacote `org.springframework.cloud.client.loadbalancer`. Perceba que não é de um pacote específico do Ribbon, mas de um pacote mais genérico. Essa anotação está definida no projeto Spring Cloud Commons, que contém diversas abstrações que permitem deixar o código desacoplado de implementações específicas. É o poder das abstrações!

Integrando com Feign

O Feign já é totalmente integrado com o Ribbon. A presença do JAR do Ribbon faz com que as chamadas feitas pelo Feign sejam interceptadas e o load balancing seja efetuado.

Basta fornecer o nome do serviço na anotação `@FeignClient`:

```
@FeignClient("estoque")  
public interface EstoqueRestClient {  
  
    @GetMapping("/item/{id}")  
    Map obtemDadosDoItem(@PathVariable("id") Long id);  
  
}
```

Com o código anterior, seria feita uma requisição à URL `http://estoque/item/1`, alternando entre as duas instâncias configuradas pelo Ribbon.

Integrando com Zuul

O Zuul já é totalmente integrado com o Ribbon. Não há nem a necessidade de declarar o Spring Cloud Netflix Ribbon como dependência do projeto.

Nas configurações do Zuul, precisaríamos criar uma rota com o nome do serviço, usando a propriedade `path` para determinar qual o prefixo de URL associado:

```
zuul.routes.estoque.path=/estoque/**
```

Com a configuração anterior, qualquer requisição ao Zuul cuja URL for iniciada com `/estoque` seria redirecionada para a configuração do serviço `estoque` do Ribbon que, por sua vez, cuidaria da distribuição das chamadas às diferentes instâncias.

Detalhando o log de requests do serviço de distância

Para que todas as requisições do serviço de distância sejam logadas (e com mais informações), vamos configurar um `CommonsRequestLoggingFilter`.

Para isso, crie a classe `RequestLogConfig` no pacote `br.com.caelum.eats.distancia`:

```
##### fj33-eats-distancia-
service/src/main/java(br/com/caelum/eats/distancia/RequestLogConfig.java)
```

```
@Configuration
class RequestLogConfig {

    @Bean
    CommonsRequestLoggingFilter requestLoggingFilter() {
        CommonsRequestLoggingFilter loggingFilter = new CommonsRec
```

```
        loggingFilter.setIncludeClientInfo(true);
        loggingFilter.setIncludePayload(true);
        loggingFilter.setIncludeHeaders(true);
        loggingFilter.setIncludeQueryString(true);
        return loggingFilter;
    }

}
```

Os imports são os seguintes:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.filter.CommonsRequestLoggingFil
```

O nível de log do CommonsRequestLoggingFilter deve ser modificado para DEBUG no application.properties:

```
##### fj33-eats-distancia-
service/src/main/resources/application.properties
```

```
logging.level.org.springframework.web.filter.CommonsRequestLoc
```

Exercício: executando uma segunda instância do serviço de distância

1. Interrompa o serviço de distância.

No projeto fj33-eats-distancia-service, vá até a branch cap8-detalhando-o-log-de-resquests-do-servico-de-distancia:

```
cd ~/Desktop/fj33-eats-distancia-service
git checkout -f cap8-detalhando-o-log-de-resquests-do-servico-
```

Execute a classe `EatsDistanciaServiceApplication`.

2. Configure a segunda instância do serviço de distância para que seja executada na porta 9092.

No Eclipse, acesse o menu *Run > Run Configurations....*

Clique com o botão direito na configuração `EatsDistanciaServiceApplication` e, então, na opção *Duplicate*.

Deve ser criada a configuração `EatsDistanciaServiceApplication (1)`.

Na aba *Arguments*, defina 9092 como a porta dessa segunda instância, em *VM Arguments*:

```
-Dserver.port=9092
```

Clique em *Run*. Nova instância do serviço de distância no ar!

3. Acesse uma URL do serviço de distância que está sendo executado na porta 8082 como, por exemplo, a URL
`http://localhost:8082/restaurantes/mais-proximos/71503510`. Verifique os logs no Console do Eclipse, na configuração `EatsDistanciaServiceApplication`.

Use a porta para 9092, por meio de uma URL como

`http://localhost:9092/restaurantes/mais-proximos/71503510`. Note que os logs do Console do Eclipse agora são da configuração `EatsDistanciaServiceApplication (1)`.

Client side load balancing no RestTemplate do monólito com Ribbon

No `pom.xml` do módulo `eats`, o módulo pai do monólito, adicione uma

dependência ao *Spring Cloud* na versão Greenwich.SR2, em `dependencyManagement`:

```
##### fj33-eats-monolito-modular/eats/pom.xml
```

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Greenwich.SR2</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Adicione o *starter* do Ribbon como dependência do módulo `eats-application` do monólito:

```
##### fj33-eats-monolito-modular/eats/eats-application/pom.xml
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>
```

Para que a instância do `RestTemplate` configurada no módulo `eats-application` do monólito use o Ribbon, anote o método `restTemplate` de `RestClientConfig` com `@LoadBalanced`:

```
##### fj33-eats-monolito-modular/eats/eats-application/src/main/java/br/com/caelum/eats/RestClientConfig.java
```

```
@Configuration
```

```
public class RestClientConfig {  
  
    @LoadBalanced // adicionado  
    @Bean  
    public RestTemplate restTemplate() {  
        return new RestTemplate();  
    }  
  
}
```

O import correto é:

```
import org.springframework.cloud.client.loadbalancer.LoadBala
```

Mude o arquivo `application.properties`, do módulo `eats-application` do monólito, para que seja configurado o *virtual host* `distoria`, com uma lista de servidores cujas chamadas serão alternadas.

Faça com que a propriedade `configuracao.distancia.service.url` aponte para esse *virtual host*.

Por enquanto, desabilite o Eureka, que será abordado mais adiante.

```
##### fj33-eats-monolito-modular/eats/eats-  
application/src/main/resources/application.properties
```

```
configuracao.distancia.service.url=http://localhost:8  
configuracao.distancia.service.url=http://distancia
```

```
distancia.ribbon.listOfServers=http://localhost:8082,http://lc  
ribbon.eureka.enabled=false
```

Client side load balancing no RestTemplate do API Gateway com Ribbon

O Zuul já é integrado com o Ribbon e, por isso, não precisamos colocá-lo como dependência.

Modifique o `application.properties` do `api-gateway`, para que use o Ribbon como *load balancer* nas chamadas ao serviço de distância.

Troque a configuração do Zuul do serviço de distância para fazer um *matching* pelo `path`. Em seguida, configure a lista de servidores do Ribbon com as instâncias do serviço de distância.

Adicione a propriedade `configuracao.distancia.service.url`, usando a URL `http://distancia` do Ribbon. Essa propriedade será usada no `DistanciaRestClient`.

```
##### fj33-api-gateway/src/main/resources/application.properties

zuul.routes.distancia.path=http://localhost:8082
zuul.routes.distancia.path=/distancia/**
distancia.ribbon.listOfServers=http://localhost:8082,http://lo
configuracao.distancia.service.url=http://distancia
```

Modifique a anotação `@Value` do construtor de `DistanciaRestClient` para que use a propriedade `configuracao.distancia.service.url`:

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/DistanciaRestClient.jav
a
```

```
class DistanciaRestClient {

    // código omitido ...

    DistanciaRestClient(RestTemplate restTemplate,
        @Value("-${zuul.routes.distancia.path}")->str:
```

```
    @Value("${configuracao.distancia.service.url}") String  
  
    this.restTemplate = restTemplate;  
    this.distanciaServiceUrl = distanciaServiceUrl;  
  
}  
  
// restante do código ...  
  
}
```

Na classe `RestClientConfig` do `api-gateway`, faça com que o `RestTemplate` seja `@LoadBalanced`:

```
##### fj33-api-  
gateway/src/main/java/br/com/caelum/apigateway/RestClientConfig.java
```

```
@Configuration  
class RestClientConfig {  
  
    @LoadBalanced // adicionado  
    @Bean  
    RestTemplate restTemplate() {  
        return new RestTemplate();  
    }  
  
}
```

Lembrando que o import correto é:

```
import org.springframework.cloud.client.loadbalancer.LoadBala
```

Exercício: Client side load balancing no RestTemplate com Ribbon

1. Interrompa o monólito e o API Gateway.

Faça o checkout da branch `cap8-client-side-load-balancing-no-rest-template-com-ribbon` dos projetos `fj33-eats-monolito-modular` e `fj33-api-gateway`:

```
cd ~/Desktop/fj33-eats-monolito-modular  
git checkout -f cap8-client-side-load-balancing-no-rest-template-com-ribbon
```

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap8-client-side-load-balancing-no-rest-template-com-ribbon
```

Execute novamente o monólito e o API Gateway.

2. Certifique-se que o monólito, o serviço de distância, o API Gateway e a UI estejam no ar.

Teste a alteração do CEP e/ou tipo de cozinha de um restaurante. Para isso, efetue o login como um dono de restaurante. Se desejar, use as credenciais pré-cadastradas (`longfu/123456`) do restaurante Long Fu.

Observe qual instância do serviço de distância foi invocada.

Tente alterar novamente o CEP e/ou tipo de cozinha do restaurante. Note que foi invocada a outra instância do serviço de distância.

A cada alteração, as instâncias são invocadas alternadamente.

3. Teste também a API Composition do API Gateway, que invoca o serviço de distância usando um `RestTemplate` do Spring, agora com `@LoadBalanced`, na classe `DistanciaRestClient`.

Observe, pelos logs, que a URL `http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1` também alterna entre as instâncias.

O Zuul já está integrado com o Ribbon. Então, ao utilizarmos o Zuul como proxy, a alternância entre as instâncias já é efetuada. Teste isso acessando a URL

`http://localhost:9999/distancia/restaurantes/mais-proximos/71503510.`

Exercício: executando uma segunda instância do monólito

1. Faça com que uma segunda instância do monólito rode com a porta 9090.

No workspace do monólito, acesse o menu *Run > Run Configurations...* do Eclipse e clique com o botão direito na configuração `EatsApplication` e depois clique em *Duplicate*.

Na configuração `EatsApplication (1)` que foi criada, acesse a aba *Arguments* e defina `9090` como a porta da segunda instância, em *VM Arguments*:

```
-Dserver.port=9090
```

Clique em *Run*. Nova instância do monólito no ar!

Client side load balancing no Feign do serviço de pagamentos com Ribbon

Adicione como dependência o *starter* do Ribbon no `pom.xml` do `eats-pagamento-service`:

```
##### fj33-eats-pagamento-service/pom.xml
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
```

```
</dependency>
```

Configure a URL do monólito para use uma lista de servidores do Ribbon e, por enquanto, desabilite o Eureka:

```
##### fj33-eats-pagamento-
service/src/main/resources/application.properties
```

```
configuracao-pedido-service.url=http://localhost:8080
monolito.ribbon.listOfServers=http://localhost:8080,http://loc
ribbon.eureka.enabled=false
```

Troque a anotação do Feign em `PedidoRestClient` para que aponte para a configuração `monolito` do Ribbon:

```
##### fj33-eats-pagamento-
service/src/main/java/br/com/caelum/eats/pagamento/PedidoRestClient.j
ava
```

```
@FeignClient(url = "${configuracao-pedido-service.url}")
@FeignClient("monolito") // modificado
public interface PedidoRestClient {

    // código omitido ...

}
```

Client side load balancing no Feign do API Gateway com Ribbon

No `application.properties` do `api-gateway`, adicione da URL da segunda instância do monólito:

```
##### fj33-api-gateway/src/main/resources/application.properties
```

```
monolito.ribbon.listOfServers=http://localhost:8080  
monolito.ribbon.listOfServers=http://localhost:8080,http://loc
```

Exercício: Client side load balancing no Feign com Ribbon

1. Pare o serviço de pagamentos e o API Gateway.

Vá até a branch `cap8-client-side-load-balancing-no-feign-com-ribbon` nos projetos `fj33-eats-pagamento-service` e `fj33-api-gateway`:

```
cd ~/Desktop/fj33-eats-pagamento-service  
git checkout -f cap8-client-side-load-balancing-no-feign-com-ribbon
```

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap8-client-side-load-balancing-no-feign-com-ribbon
```

Execute novamente o serviço de pagamentos e o API Gateway.

2. Garanta que o serviço de pagamentos foi reiniciado e que as duas instâncias do monólito estão no ar.

Use um cliente REST como o cURL para confirmar um pagamento:

```
curl -X PUT -i http://localhost:8081/pagamentos/1
```

Teste várias vezes seguidas e note que os logs são alternados entre `EatsApplication` e `EatsApplication (1)`, as instâncias do monólito.

Observação: confirmar um pagamento já confirmado tem o mesmo efeito, incluindo o aviso de pagamento ao monólito.

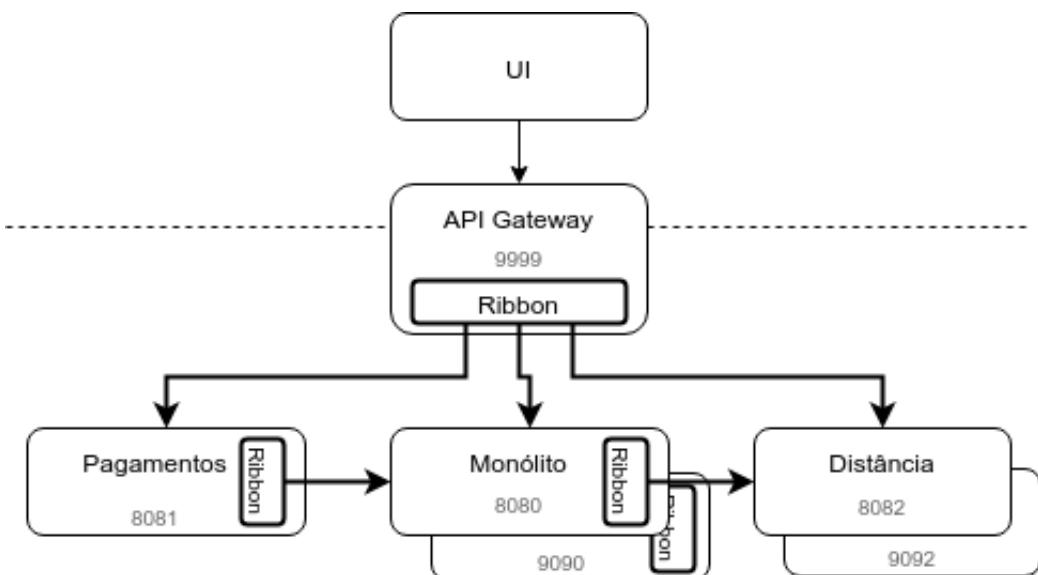
3. Acesse pelo API Gateway, por duas vezes seguidas, uma URL do monólito como `http://localhost:9999/restaurantes/1`.

Veja que os logs são alternados entre os Consoles de `EatsApplication` e `EatsApplication (1)`.

Service Discovery

No capítulo anterior, temos as seguintes instâncias:

- 2 instâncias do Monólito, uma na porta 8080 e outra na porta 9090
- 1 instância do serviço de Pagamentos, na porta 8081
- 2 instâncias do serviço de Distância, uma na porta 8082 e outra na porta 9092



Para que o Ribbon faça o Client Side Load Balancing dessas instâncias, os `application.properties` das aplicações cliente devem conter os endereços de todas instâncias dos serviços que são chamados.

No API Gateway, o Monólito e os serviços de Pagamentos e Distância são usados para proxy e/ou API Composition. Para efetuar o Load Balancing corretamente, há referências aos endereços das duas instâncias do serviço de Distância e também das do Monólito:

```
##### fj33-api-gateway/src/main/resources/application.properties
```

```
distancia.ribbon.listOfServers=http://localhost:8082,http://lc  
...  
monolito.ribbon.listOfServers=http://localhost:8080,http://loc
```

Como o serviço de Pagamento chama o módulo de Pedidos do Monólito para avisar que foi confirmado um pagamento, há a necessidade do endereço das instâncias do Monólito em suas configurações:

```
##### fj33-eats-pagamento-
service/src/main/resources/application.properties
```

```
monolito.ribbon.listOfServers=http://localhost:8080,http://loc
```

Já o Monólito chama o serviço de Distância quando há uma alteração em um CEP e/ou um tipo de cozinha de um restaurante. Por isso, o endereço das instâncias do serviço de Distância está entre suas configurações.

```
##### fj33-eats-monolito-modular/eats/eats-
application/src/main/resources/application.properties
```

```
distanzia.ribbon.listOfServers=http://localhost:8082,http://lc
```

Qual seria o impacto de uma nova instância do serviço de Distância?

Teríamos que alterar as configurações dos `application.properties` de todos os clientes desse serviço, tanto do Monólito como do API Gateway, fazer novo build dos projetos afetados e publicar o JAR.

Será que há uma maneira mais fácil. Sim! No Spring Boot, é possível sobrescrever a maioria das configurações com variáveis de ambiente. A propriedade `distanzia.ribbon.listofServers` seria equivalente a `DISTANCIA_RIBBON_LISTOFSERVERS`. Para adicionar uma instância do serviço de Distância na porta 9876 no Linux, teríamos algo como:

```
export DISTANCIA_RIBBON_LISTOFSERVERS=http://localhost:8082,ht
```

Bem melhor que ter que recompilar e republicar o JAR.

Mais ainda assim, teríamos que configurar variáveis de ambiente nas máquinas de todos os clientes e reiniciar suas aplicações.

Registrando serviços

Como afirma Chris Richardson, em seu livro [Microservices Patterns](#) (RICHARDSON, 2018a), numa aplicação tradicional que é executada em hardware físico, os endereços de rede das instâncias dos serviços geralmente são estáticos. Seu código poderia ler URLs de um arquivo de configuração que é atualizado apenas ocasionalmente. Mas uma aplicação moderna, baseada em cloud, não é tão simples: há a necessidade de maior dinamicidade.

Instâncias de serviços tem URLs determinadas dinamicamente, Além disso, o conjunto de instâncias de um serviço muda dinamicamente por causa de falhas, atualizações ou *autoscaling*. É necessário que o código do cliente descubra dinamicamente quais os endereços das instâncias disponíveis. É o que chamamos de **Service Discovery**.

Para implementar Service Discovery, precisamos de um catálogo que mantenha as URLs das instâncias de um serviço. Chamamos esse catálogo de **Service Registry**.

Quando instâncias de um serviço são iniciadas ou paradas, o Service Registry deve ser atualizado. Ao invocar um serviço, um cliente consulta o Service Registry para obter as instâncias disponíveis e direciona a requisição a uma delas.

Sam Newman afirma, em seu livro [Building Microservices](#) (NEWMAN, 2015), que as instâncias registradas no Service Registry podem ser usadas para saber o que deve ser monitorado e quais as APIs disponíveis para os desenvolvedores.

Newman indica que DNS poderia ser utilizado como um Service Registry. Porém, poucos servidores DNS são configurados para lidar com

instâncias altamente "descartáveis", fazendo com que a atualização do DNS seja trabalhosa. Além disso, os clientes mantêm os endereços obtidos do DNS por pelo menos o TTL (*time to live*). Isso pode levar a endereços desatualizados.

Para lidar com a dinamicidade da criação e destruição de instâncias de serviços em um ambiente Cloud, diferentes softwares são usados como Service Registry. Entre eles:

- [ZooKeeper](#), da Apache Software Foundation, é um sistema de coordenação distribuída originalmente desenvolvido como parte do projeto Hadoop. É bastante genérico em suas ofertas, lidando com gerenciamento de configurações, sincronização de dados, *leader election*, fila de mensagens e Service Discovery. É possível que clientes sejam alertados sobre mudanças nas configurações.
- [Consul](#), da HashiCorp, lida com gerenciamento de configurações e Service Discovery. Expõe uma RESTful API, é compatível com DNS, provê um *key-value store* distribuído, entre outras funcionalidades. Por meio de requisições HTTP, uma instância de serviço pode ser registrada, um endereço pode ser consultado, etc.
- [Eureka](#), da iniciativa open-source da Netflix, foca em Service Discovery e provê uma RESTful API e clientes Java. É integrado com o Ribbon, a ferramenta de Client Side Load Balancing da Netflix OSS.

Um problema que surge: como atualizar o registro das instâncias disponíveis no Service Registry?

Uma maneira bem ágil é cada instância avisar ao Service Registry que está no ar ou vai ser derrubada. No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson chama essa solução de *Self Registration*.

Pattern: Self Registration

Uma instância de um serviço se registra no Service Registry.

Do lado do cliente, há a necessidade de uma consulta ao Service Registry para obter quais instâncias de um serviço estão registradas. Então, o cliente usa um algoritmo de Load Balancing para selecionar uma instância da lista e efetua a requisição. Para melhorar a performance, o cliente pode fazer um cache das instâncias. No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson chama essa solução de *Client-Side Discovery*.

Pattern: Client-side Discovery

Um cliente recupera a lista de instâncias disponíveis de um serviço do Service Registry e faz Load Balancing entre os endereços obtidos.

Eureka

Nesse curso, usaremos o Eureka, que é integrado com o ecossistema Spring através do projeto Spring Cloud Netflix Eureka. Para implementar o Service Registry, será utilizado o projeto Spring Cloud Netflix Eureka Server, disponível pelo artefato `spring-cloud-starter-netflix-eureka-server`. Por padrão, o Eureka Serve usa a porta 8761.

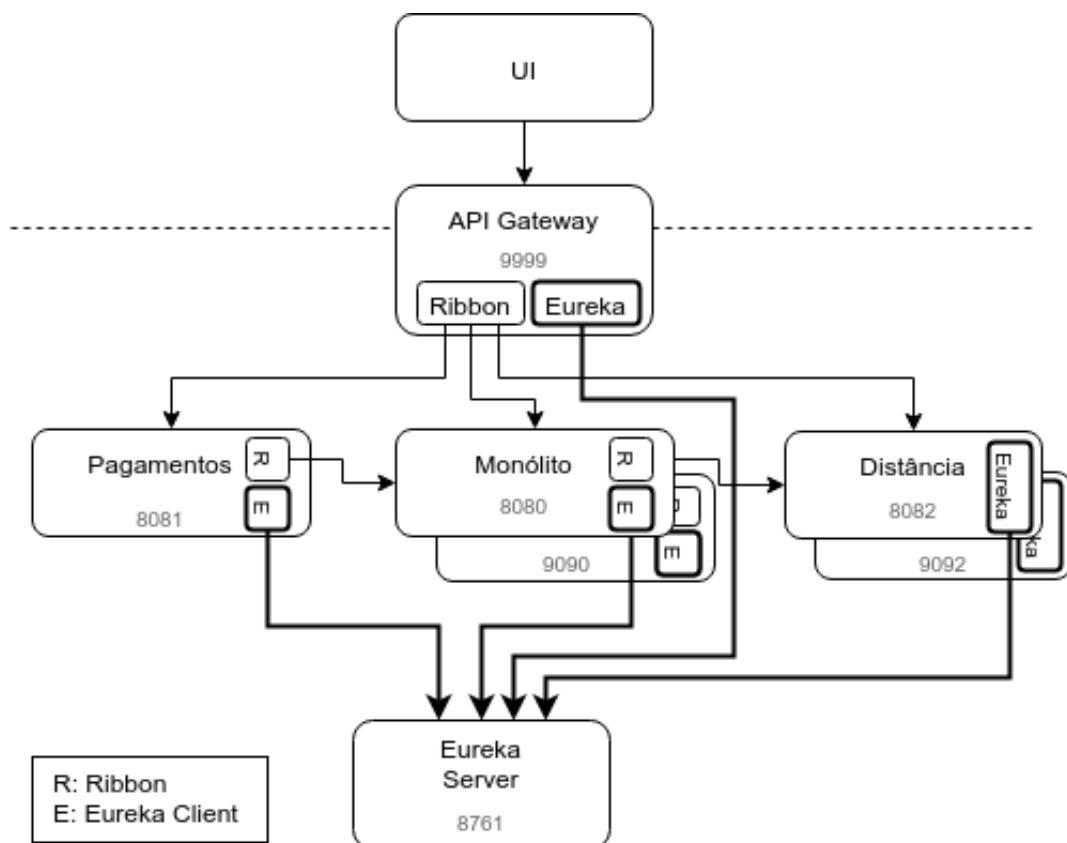
Para implementar o pattern *Self Registration*, as instâncias dos serviços precisam invocar a API do Service Registry. No caso ecossistema Spring, há o Spring Cloud Netflix Eureka Client, disponível no artefato `spring-cloud-starter-netflix-eureka-client`. Essa biblioteca que provê uma maneira baseada em anotações de uma instância registrar-se no Service Registry.

Para clientes feitos em linguagens que não rodam na JVM, o Eureka Server disponibiliza uma API RESTful. A documentação dessa API pode ser encontrada em: <https://github.com/Netflix/eureka/wiki/Eureka-REST-operations>

A biblioteca Eureka Client implementa também o pattern *Client-Side Discovery*.

No curso usaremos, no lado do cliente, o Spring Cloud Netflix Eureka Client por meio do artefato `spring-cloud-starter-netflix-eureka-client`. O Eureka Client fica responsável por obter a lista de instâncias registradas e disponíveis no Eureka Server. O Load Balancing fica por conta do Ribbon, que vimos em capítulo anterior.

O API Gateway, o Monólito e os serviços de Pagamentos e Distância terão o Spring Cloud Netflix Eureka Client e serão registrados no Eureka Server. O Ribbon obterá da Eureka Client a lista de instâncias que será usada no Load Balancing. As configurações de lista de servidores poderão ser removidas do `application.properties`!



Implementando um Service Registry com o Eureka

Pelo navegador, abra <https://start.spring.io/>. Em *Project*, mantenha *Maven Project*. Em *Language*, mantenha *Java*. Em *Spring Boot*, mantenha a versão padrão. No trecho de *Project Metadata*, defina:

- `br.com.caelum` em *Group*

- service-registry em *Artifact*

Mantenha os valores em *More options*.

Mantenha o *Packaging* como *Jar*. Mantenha a *Java Version* em 8.

Em *Dependencies*, adicione:

- Eureka Server

Clique em *Generate Project*.

Extraia o *service-registry.zip* e copie a pasta para seu Desktop.

Adicione a anotação `@EnableEurekaServer` à classe

ServiceRegistryApplication:

```
##### fj33-service-
registry/src/main/java/br/com/caelum/serviceregistry/ServiceRegistryAppli-
cation.java
```

```
@EnableEurekaServer
@SpringBootApplication
public class ServiceRegistryApplication {

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

}
```

Adicione o import:

```
import org.springframework.cloud.netflix.eureka.server.EnableE
```

No arquivo *application.properties*, modifique a porta para 8761, a porta

padrão do Eureka Server, e adicione algumas configurações para que o próprio service *registry* não se registre nele mesmo.

```
##### fj33-service-registry/src/main/resources/application.properties  
  
server.port=8761  
  
eureka.client.register-with-eureka=false  
eureka.client.fetch-registry=false  
logging.level.com.netflix.eureka=OFF  
logging.level.com.netflix.discovery=OFF
```

Exercício: executando o Service Registry

1. Em um Terminal, clone o repositório `fj33-service-registry` para seu Desktop:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-service
```

2. No Eclipse, no workspace de microservices, importe o projeto `fj33-service-registry`, usando o menu *File > Import > Existing Maven Projects*.

Execute a classe `ServiceRegistryApplication`.

Acesse, por um navegador, a URL `http://localhost:8761`. Esse é o Eureka!

Por enquanto, a seção *Instances currently registered with Eureka*, que mostra quais serviços estão registrados, está vazia.

Self Registration do serviço de distância no Eureka Server

No pom.xml do eats-distancia-service, adicione uma dependência ao Spring Cloud na versão Greenwich.SR2, em dependencyManagement:

```
##### fj33-eats-distancia-service/pom.xml
```

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Greenwich.SR2</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Adicione o starter do Eureka Client como dependência:

```
##### fj33-eats-distancia-service/pom.xml
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Adicione a anotação @EnableDiscoveryClient à classe EatsDistanciaApplication:

```
@EnableDiscoveryClient // adicionado
@SpringBootApplication
public class EatsDistanciaApplication {

  // código omitido ...
}
```

```
}
```

Adicione o import:

```
import org.springframework.cloud.client.discovery.EnableDiscovery
```

É preciso identificar o serviço de distância para o Eureka Server. Para isso, adicione a propriedade `spring.application.name` ao `application.properties`:

```
##### fj33-eats-distancia-
service/src/main/resources/application.properties
```

```
spring.application.name=distancia
```

A URL padrão usada pelo Eureka Client é `http://localhost:8761/`.

Porém, um problema é que não há uma configuração para a URL do Eureka Server que seja customizada nos clientes para ambientes como de testes, homologação e produção.

É preciso definir essa configuração customizável no `application.properties`:

```
##### fj33-eats-distancia-
service/src/main/resources/application.properties
```

```
eureka.client.serviceUrl.defaultZone=${EUREKA_URI:http://local
```

Dessa maneira, caso seja necessário modificar a URL padrão do Eureka Server, basta definir a variável de ambiente `EUREKA_URI`.

Self Registration do serviço de pagamentos no Eureka Server

No `pom.xml` do `eats-pagamento-service`, adicione como dependência o `starter` do Eureka Client:

```
##### fj33-eats-pagamento-service/pom.xml
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Anote a classe `EatsPagamentoServiceApplication` com `@EnableDiscoveryClient`:

```
##### fj33-eats-pagamento-
service/src/main/java/br/com/caelum/eats/pagamento/EatsPagamentoSe
rviceApplication.java
```

```
@EnableDiscoveryClient // adicionado
@EnableFeignClients
@SpringBootApplication
public class EatsPagamentoServiceApplication {
```

```
}
```

Lembrando que o import é:

```
import org.springframework.cloud.client.discovery.EnableDiscov
```

Defina, no `application.properties`, um nome para aplicação, que será usado no Eureka Server. Além disso, adicione a configuração

customizável para a URL do Eureka Server:

```
##### fj33-eats-pagamento-
service/src/main/resources/application.properties
```

```
spring.application.name=pagamentos
```

```
eureka.client.serviceUrl.defaultZone=${EUREKA_URI:http://local
```

Self Registration do monólito no Eureka Server

No `pom.xml` do módulo `eats-application` do monólito, adicione como dependência o *starter* do Eureka Client:

```
##### fj33-eats-monolito-modular/eats/eats-application/pom.xml
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Anote a classe `EatsApplication` com `@EnableDiscoveryClient`:

```
##### fj33-eats-monolito-modular/eats/eats-
application/src/main/java/br/com/caelum/eats/EatsApplication.java
```

```
@EnableDiscoveryClient // adicionado
@SpringBootApplication
public class EatsApplication {

    // código omitido ...
}
```

Novamente, lembrando que o import correto:

```
import org.springframework.cloud.client.discovery.EnableDiscoveryClient
```

Defina, no `application.properties`, um nome para aplicação e a URL do Eureka Server:

```
##### fj33-eats-monolito-modular/eats/eats-application/src/main/resources/application.properties
```

```
spring.application.name=monolito
```

```
eureka.client.serviceUrl.defaultZone=${EUREKA_URI:http://localhost:8761/eureka}
```

Self registration do API Gateway no Eureka Server

Adicione como dependência o *starter* do Eureka Client, No `pom.xml` do `api-gateway`:

```
##### fj33-api-gateway/pom.xml
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Anote a classe `ApiGatewayApplication` com `@EnableDiscoveryClient`:

```
##### fj33-api-gateway/src/main/java/com/caelum/apigateway/ApiGatewayApplication.java
```

```
@EnableDiscoveryClient // adicionado  
@EnableFeignClients  
@EnableZuulProxy  
@SpringBootApplication  
public class ApiGatewayApplication {  
  
    // código omitido ...  
  
}
```

Lembre do novo import:

```
import org.springframework.cloud.client.discovery.EnableDiscov
```

No application.properties, defina apigateway como nome da aplicação.
Defina também a URL do Eureka Server:

```
##### fj33-api-gateway/src/main/resources/application.properties  
  
spring.application.name=apigateway  
  
eureka.client.serviceUrl.defaultZone=${EUREKA_URI:http://local
```

Exercício: Self registration no Eureka Server

1. Interrompa a execução do monólito, dos serviços de pagamentos e distância e do API Gateway.

Faça o checkout da branch cap9-self-registration-no-eureka-server nos projetos do monólito, do API Gateway e dos serviço de pagamentos e distância:

```
cd ~/Desktop/fj33-eats-monolito-modular
```

```
git checkout -f cap9-self-registration-no-eureka-server  
  
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap9-self-registration-no-eureka-server  
  
cd ~/Desktop/fj33-eats-distancia-service  
git checkout -f cap9-self-registration-no-eureka-server  
  
cd ~/Desktop/fj33-eats-pagamento-service  
git checkout -f cap9-self-registration-no-eureka-server
```

2. Pare as instâncias do serviço de distância.

Execute a *run configuration* `EatsDistanciaApplication`.

Acesse o Eureka Server pelo navegador, na URL

`http://localhost:8761/`. Observe que a aplicação *DISTANCIA* aparece entre as instâncias registradas com Eureka.

Então, execute a segunda instância do serviço de distância, usando a *run configuration* `EatsDistanciaApplication (1)`.

Recarregue a página do Eureka Server e note que são indicadas duas instâncias, com suas respectivas portas. Em *Status*, deve aparecer algo como `UP (2) - 192.168.0.90:distancia:9092 , 192.168.0.90:distancia:8082.`

3. Pare o serviço de pagamento.

Em seguida, execute novamente a classe `EatsPagamentoServiceApplication`.

Com o serviço em execução, vá até a página do Eureka Server e veja que *PAGAMENTOS* está entre as instâncias registradas.

4. Pare as duas instâncias do monólito.

A seguir, execute novamente a *run configuration* EatsApplication.

Observe *MONOLITO* como instância registrada no Eureka Server.

Execute a segunda instância do monólito com a *run configuration* EatsApplication (1).

Note o registro da segunda instância no Eureka Server, também em *MONOLITO*.

5. Pare o API Gateway.

Logo após, execute novamente ApiGatewayApplication.

Note, no Eureka Server, o registro da instância APIGATEWAY.

Client side discovery no serviço de pagamentos

No application.properties de eats-pagamento-service, apague a lista de servidores de distância do Ribbon, para que seja obtida do Eureka Server e, também, a configuração que desabilita o Eureka Client no Ribbon, que é habilitado por padrão:

```
##### fj33-eats-pagamento-
service/src/main/resources/application.properties
```

```
monolithic.ribbon.listOfServers=http://localhost:8080, h
-ribbon.eureka.enabled=false
```

Client side discovery no API Gateway

Modifique o application.properties do API Gateway, para que o Eureka Client seja habilitado e que não haja mais listas de servidores do Ribbon.

Limpe as configurações, já que boa parte delas serão obtidas pelas próprias URLs requisitadas e os nomes no Eureka Server.

Mantenha as que fazem sentido e modifique ligeiramente algumas delas.

```
##### fj33-api-gateway/src/main/resources/application.properties
```

```
ribbon.eureka.enabled=false
```

```
zuul.routes.pagamentos.url=http://localhost:8081  
zuul.routes.pagamentos.stripPrefix=false
```

```
zuul.routes.distancia.path=/distancia/**
```

```
distancia.ribbon.listOfServers=http://localhost:8082,  
configuracao.distancia.service.url=http://distancia
```

```
zuul.routes.local.path=/restaurantes-com-distancia/**
```

```
zuul.routes.local.url=forward:/restaurantes-com-distancia
```

```
zuul.routes.monolito.path=/**
```

```
zuul.routes.monolito=/**
```

```
monolito.ribbon.listOfServers=http://localhost:8080, h
```

Client side discovery no monólito

Remova, do `application.properties` do módulo `eats-application` do monólito, a lista de servidores de distância do Ribbon e a configuração que desabilita o Eureka Client:

```
##### fj33-eats-monolito-modular/eats/eats-  
application/src/main/resources/application.properties
```

```
distancia.ribbon.listOfServers=http://localhost:8082,  
ribbon.eureka.enabled=false
```

Exercício: Client Side Discovery com Eureka Client

1. Pare o monólito, o serviço de pagamentos e o API Gateway.

Obtenha o código da branch `cap9-client-side-discovery` dos repositórios do monólito, do API Gateway e do serviço de pagamentos:

```
cd ~/Desktop/fj33-eats-monolito-modular  
git checkout -f cap9-client-side-discovery
```

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap9-client-side-discovery
```

```
cd ~/Desktop/fj33-eats-pagamento-service  
git checkout -f cap9-client-side-discovery
```

Execute novamente o monólito, o serviço de pagamentos e o API Gateway.

2. Com as duas instâncias do monólito no ar, use um cliente REST como o cURL para confirmar um pagamento:

```
curl -X PUT -i http://localhost:8081/pagamentos/1
```

Note que os logs são alternados entre `EatsApplication` e `EatsApplication (1)`, quando testamos o comando acima várias vezes.

3. Teste, pelo navegador ou por um cliente REST, as seguintes URLs:

- `http://localhost:9999/restaurantes/1`, observando se os logs são alternados entre as instâncias do monólito
- `http://localhost:9999/distancia/restaurantes/mais-proximos/71503510`, e note a alternância entre logs das instâncias do serviço de distância
- `http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1`, que alterna tanto entre

instâncias do monólito como do serviço de distância

4. Com a UI, os serviços e o monólito no ar, faça login em um restaurante (`longfu/123456` está pré-cadastrado) e modifique o tipo de cozinha ou o CEP. Realize essa operação mais de uma vez.

Perceba que as instâncias do serviço de distância são chamadas alternadamente.

Para saber mais: usando o Consul como Service Registry

Como mencionado anteriormente, o [Consul](#) da HashiCorp é, entre outras utilidades, um Service Registry que provê uma RESTful API e é compatível com DNS. É implementado na linguagem Go e o uso de memória tende a ser consideravelmente menor que o Eureka Server.

O Consul foi implementado como um Sistema Distribuído altamente disponível e, por isso, implementa algoritmos como o consensus protocol Raft e o gossip protocol SWIM para membros e *broadcast*. Idealmente, deve ser executado em um cluster de 3 servidores em um mesmo *datacenter*.

Para definir um servidor Consul podemos usar a imagem `consul` do DockerHub. A porta 8500 é usada pela API HTTP e por uma Web UI semelhante à do Eureka Server. Já a porta 8600 é usada para resolver consultas DNS.

Mesmo com apenas um servidor, é necessário configurarmos o endereço do cluster Consul, apontando para uma interface de rede válida, através da variável de ambiente `CONSUL_BIND_INTERFACE`.

```
consul:  
  image: consul:1.5  
  restart: on-failure  
  ports:
```

```
- "8500:8500"
- "8600:8600"
environment:
CONSUL_BIND_INTERFACE: eth0
```

Usar o Consul em um projeto Spring Cloud não dá tanto trabalho. O projeto [Spring Cloud Consul](#) tem bibliotecas de compatibilidade com várias das funcionalidades do Consul, com integração com outros componentes do Spring Cloud, incluindo o Zuul e o Ribbon.

Para usar o Spring Cloud Consul, basta declarar como dependência de cada projeto o artefato `spring-cloud-starter-consul-all`. Com Maven:

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-consul-all</artifactId>
</dependency>
```

Observação: o Dependency Management `spring-cloud-dependencies` deve estar declarado no `pom.xml`.

No `application.properties` de cada serviço e dos respectivos clientes, devemos configurar o endereço correto do servidor Consul:

```
spring.cloud.consul.host=localhost
spring.cloud.consul.port=8500
spring.cloud.consul.discovery.register-health-check=false
```

Como ainda não vimos o conceito de *health checking* no curso, desabilitaremos essa funcionalidade.

Finalmente, basta adicionar a anotação `@EnableDiscoveryClient` na classe principal (ou em alguma classe de configuração):

```
@EnableDiscoveryClient // adicionado  
@SpringBootApplication
```

Note que a anotação `@EnableDiscoveryClient` é abstrata, sendo declarada no pacote `org.springframework.cloud.client.discovery`, do Spring Cloud Commons. Ao migrar um projeto do Eureka para o Consul, não há a necessidade de alterar as classes principais. É o poder das abstrações!

Para saber mais: Qual a diferença entre o Eureka e o AWS ELB?

Na [documentação do Eureka](#), há a seguinte comparação:

O AWS *Elastic Load Balancer (ELB)* é uma solução de *Load Balancing* para edge services expostos ao tráfego da web do usuário final. O Eureka preenche a necessidade de *Load Balancing* nas **chamadas entre serviços**. Embora você possa, teoricamente, colocar seus serviços internos atrás do AWS ELB, no EC2, você os expõe ao mundo exterior e perdendo toda a utilidade dos security groups da AWS.

O AWS ELB também é uma solução tradicional de *Load Balancing* baseada em proxy, enquanto no Eureka é diferente, pois o *Load Balancing* ocorre **no nível da instância**. As instâncias do cliente conhecem com quais servidores precisam conversar [...]

Outro aspecto importante que diferencia o *Load Balancing* baseado em proxy [do AWS ELB] do *Load Balancing* do Eureka é que seu aplicativo pode ser **resiliente às interrupções dos load balancers**, pois as informações sobre os servidores disponíveis são armazenadas em cache no cliente [...]

Para saber mais: Third party registration e Server-side Discovery

No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson afirma que há plataformas que possuem Service Discovery implícita, como se tivessem um Service Registry já embutido. A própria plataforma fica responsável pelo registro das instâncias, pela descoberta das instâncias disponíveis e pelo balanceamento de carga. Dessa forma, não há a necessidade de código ou bibliotecas de Service Discovery nos serviços e em seus clientes. Consequentemente, esse tipo de implementação é nativamente multi-plataforma.

Richardson lista dois *patterns* relacionados:

- **Third party registration:** ao invés de um serviço registrar-se no Service Registry, um outro componente chamado *Registrar*, que normalmente faz parte da plataforma de implantação, lida com o registro.
- **Server-side Discovery:** Em vez de um cliente consultar o Service Registry, é feita uma solicitação para um DNS name, que é resolvido para um componente chamado *Request Router*, que consulta o registro do serviço e faz o Load Balancing.

Como dissemos em capítulo anterior, no orquestrador de containers [Kubernetes](#), há o conceito de Service, que expõe um conjunto de Pods sob um mesmo DNS name. Richardson afirma que um Service do Kubernetes é uma forma de Service Discovery provido pela infra-estrutura. Os dados de um cluster Kubernetes são armazenados no [etcd](#), um BD distribuído do tipo chave-valor.

Outras plataformas como [Marathon](#), um orquestrador de containers para [Datacenter Operating System \(DC/OS\)](#) e [Apache Mesos](#), implementam os mesmos patterns.

Resiliência

Revisitando o conceito de Disponibilidade

Em capítulos anteriores, definimos Disponibilidade como a proporção de tempo, em um determinado período, em que o sistema opera normalmente.

Mas o que acontece se um sistema estiver extremamente lento?

Conforme mencionamos nos primeiros capítulos, ao falar sobre Consistência em um Sistema Distribuído, no paper [Consistency Tradeoffs in Modern Distributed Database System Design](#) (ABADI, 2012), Daniel Abadi amplia o Teorema CAP, cunhando o Teorema PACELC: no caso de uma Partição na rede, um Sistema Distribuído precisa escolher entre Disponibilidade (em inglês, *Availability*) e Consistência; se não houver Partição na rede, a escolha é entre Latência e Consistência. Simplificando, podemos incluir alta latência de rede como uma forma de indisponibilidade.

Observe que Disponibilidade e Latência são sem dúvida a mesma coisa: um sistema indisponível fornece essencialmente uma Latência extremamente alta. Para os fins desta discussão, considero como indisponíveis sistemas com latências maiores que um timeout típico para uma requisição, como alguns segundos; e como "Alta Latência", sistemas com latências menores que um timeout típico, mas ainda se aproximando de centenas de milissegundos.

Daniel Abadi, no paper [Consistency Tradeoffs in Modern Distributed Database System Design](#) (ABADI, 2012)

Falhas em Cascata

Em um Sistema Distribuído implementado de maneira ingênua, uma lentidão em um serviço pode levar o sistema todo a ficar indisponível. Isso é comumente conhecido como falhas em cascata (em inglês,

Cascading Failures).

No livro [Microservices in Action](#) (BRUCE; PEREIRA, 2018), os autores ligam esse tipo de comportamento em cascata com um comportamento emergente de Sistemas Complexos: *um evento perturba um Sistema, levando a algum efeito que, por sua vez, aumenta a magnitude do distúrbio inicial. [...] Considere uma debandada em um rebanho de animais: o pânico faz um animal correr que, por sua vez, espalha o pânico para outros animais, o que faz com que eles corram, e assim por diante. Em Microservices, uma sobrecarga pode causar um efeito dominó: uma falha em um serviço aumenta falhas nos serviços que o chamam e, por sua vez, nos serviços que chamam esses. No pior caso, o resultado é uma indisponibilidade generalizada.*

No livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman descreve o caso em um sistema de agregador de anúncios, que unia resultados de diferentes sistemas legados. Um dos legados mais antigos e pouco usados, que correspondia a menos de 5% do faturamento, passou a responder de maneira muito lenta. A demora fez com que o único pool de conexões fosse preenchido e, como as threads ficavam esperando indefinidamente, o sistema todo veio abaixo.

Responder muito lentamente é um dos piores modos de falha que você pode experimentar. Se um sistema não está no ar, você descobre rapidamente. Quando apenas está lento, você acaba esperando um pouco antes de desistir. [...] descobrimos da maneira mais difícil que os sistemas lentos são muito mais difíceis de lidar do que os sistemas que falham rapidamente. Em um Sistema Distribuído, a Latência mata.

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015)

Patterns de Estabilidade

Desejamos que nosso Sistema Distribuído apresente como característica a **Resiliência**. Segundo o Dicionário Michaelis, um significado figurativo

da palavra [Resiliência](#) é a capacidade de rápida adaptação ou recuperação. A qual *ilidade* a Resiliência estaria ligada? À **Estabilidade** que, também segundo o Dicionário Michaelis, é característica daquilo que é estável; solidez.

Michael Nygard, em seu livro [Release It! Second Edition](#) (NYGARD, 2018), define Estabilidade como a característica observável em *um sistema resiliente que se mantém processando transações [de negócio], mesmo quando há impulsos transitórios, estresses persistentes ou falhas de componentes que interrompem o processamento normal.*

Nygard lista uma série de patterns de Estabilidade. Alguns deles serão descritos a seguir.

Timeouts

No exemplo do agregador de anúncios mencionado por Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), a lentidão em um dos legados ocasionou uma interrupção no sistema todo. O motivo mencionado por Newman é que o pool de conexões único utilizado esperava "para sempre" e, com uma alta demanda, o pool foi exaurido e o serviço de anúncios não poderia chamar nenhum outro legado. A biblioteca de pool de conexões dava suporte a um **Timeout**, mas estava desabilitada por padrão!

Newman, recomenda que todas as chamadas remotas tenham Timeouts configurados. E qual valor definir? Se for longo demais, ainda causará lentidão no sistema. Se for rápido demais, uma chamada bem sucedida por ser considerado como falha. Uma boa solução é usar valores default de bibliotecas, ajustando valores para cenários específicos.

Em seu livro [Release It! Second Edition](#) (NYGARD, 2018), Michael Nygard argumenta que Timeouts provêem *isolamento de falhas*, já que um problema em outro serviço, dispositivo ou sistema não deve ser um problema de quem o invoca. Nygard relata que, infelizmente, muitas APIs e bibliotecas de clientes de sistemas como Bancos de Dados não

provêem maneiras de setar Timeouts. Para Nygard, qualquer pool de recursos que bloqueia threads deve ter Timeouts.

Fail Fast

Michael Nygard, ainda no livro [Release It! Second Edition](#) (NYGARD, 2018), diz: *se respostas lentas são piores que nenhuma resposta, o pior dos mundos certamente são falhas lentas.* Se um sistema puder detectar que vai falhar, é melhor que retorne o mais rápido possível um response de erro para seus clientes. Falhe rápido (em inglês, **Fail Fast**). Como predizer uma falha? Nygard afirma que um load balancer, por exemplo, deve recusar novas requisições se não houver nenhum servidor para balanceamento no ar, evitando enfileirar requisições.

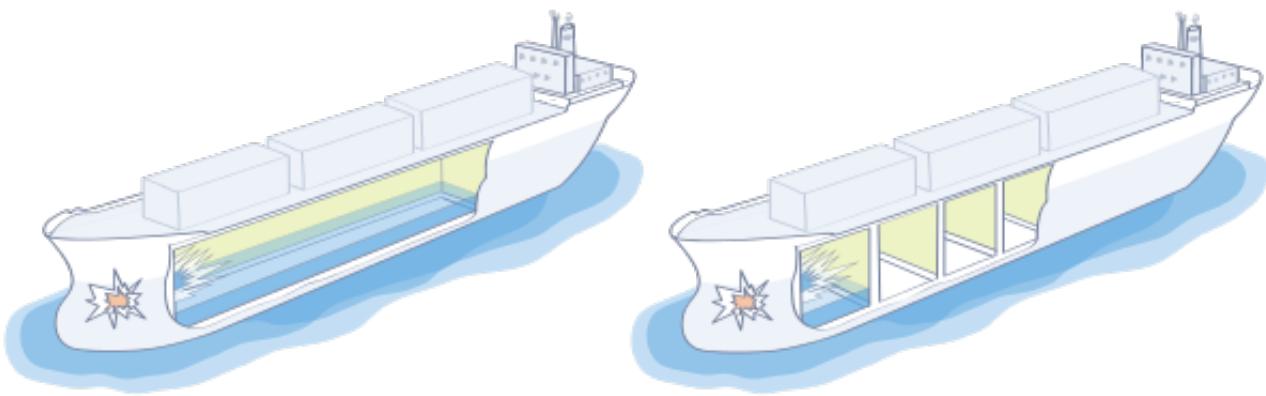
Para Nygard, no código de uma aplicação, parâmetros devem ser validados e recursos obtidos, como conexões a BDs ou a outros sistemas, assim que possível. Assim, se houver uma falha de validação ou na obtenção de algum recurso, é possível falhar rapidamente. É importante notar que isso serve como um princípio, porém não é aplicável em todas as situações.

Para Nygard, tanto o Timeout como Fail Fast são patterns que tratam de problemas de latência, sendo dois lados da mesma moeda. O Timeout é útil para proteger o seu sistema de falhas dos outros. O Fail Fast é útil quando você precisa avisar o porquê não será possível processar alguma requisição. Timeouts são para saídas e Fail Fast para entradas.

Bulkheads

Ainda no livro [Release It! Second Edition](#) (NYGARD, 2018), Michael Nygard empresta um conceito da Engenharia Naval: as anteparas (em inglês, **Bulkheads**): *em um navio, Bulkheads são divisórias metálicas que podem ser seladas para dividir o navio em compartimentos separados. Quando as escotilhas são fechadas, uma Bulkhead impede que a água se move de uma seção para outra. Dessa maneira, um único dano no casco não afunda irreversivelmente o navio.* A Bulkhead aplica um princípio de

contenção de dados.



Observação: a fonte das imagens anteriores é a [documentação do OpenLiberty](#), um Microservice Chassis baseado no IBM WebSphere.

Nygard afirma que, em TI, redundância física é a maneira mais comum de aplicar a ideia de Bulkheads: uma falha no hardware de um servidor não afetaria os outros. O mesmo princípio pode ser atingido executando múltiplas instâncias de um serviço em um mesmo servidor.

Para Nygard, há maneiras mais granulares de aplicar Bulkheads. Por exemplo, é possível apartar em pools diferentes threads de um processo que tem responsabilidades distintas, separando threads que tratam requests de threads administrativas. Assim, se as threads da aplicação travarem, é possível usar as threads administrativas para obter um dump ou fazer um *shut down*. Nygard menciona ainda que um Sistema Operacional pode alocar um processo a um core específico (ou a um grupo de cores), o que é conhecido como *CPU Binding*. Dessa maneira, a sobrecarga em um processo não degrada a performance da máquina toda, já outros cores estarão liberados para outros processos.

Há também maneiras menos granulares. No caso de *cloud computing*, podem ser exploradas diferentes topologias como zonas e regiões da AWS.

Você pode partitionar pools de threads em uma aplicação, CPUs em um servidor ou servidores em um cluster.

Michael Nygard, no livro [Release It! Second Edition](#) (NYGARD, 2018)

Em seu livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman afirma que as barreiras arquiteturais entre microservices são, no fim das contas, Bulkheads: uma falha em um serviço pode degradar certas funcionalidades, mas não pára tudo.

No exemplo do agregador de anúncios de Newman, todas as conexões aos sistemas legados compartilhavam um mesmo pool de conexões. A falha em um legado derrubou o único pool de conexões e, consequentemente, impediu que chamadas fossem feitas aos outros legados, tornando o sistema inutilizável. Usando a ideia de Bulkheads, cada sistema legado deveria ter seu próprio pool de conexões, de maneira a isolar possíveis falhas.

Circuit Breaker

Michael Nygard, no livro [Release It! Second Edition](#) (NYGARD, 2018), conta que quando a fiação elétrica começou a ser construída nas casas, à medida que as pessoas plugassem mais aparelhos, os fios iam esquentando mais e mais, até que a casa fosse incendiada, eventualmente. A indústria então passou a usar fusíveis residenciais que queimavam antes da fiação (*fail fast*), protegendo as casas. Só que os fusíveis são descartáveis e as pessoas começaram a usa moedas de cobre no lugar. Resultado: casas queimadas. Então, inventaram o disjuntor (em inglês, **Circuit Breaker**), atualmente presente em qualquer prédio residencial ou comercial. Um Circuit Breaker detecta um uso excessivo de corrente e abre, desarmando como um fusível, desligando todos os aparelhos. Mas, diferentemente de um fusível, um Circuit Breaker pode ser fechado novamente, de maneira manual, assim que não houver perigo.

O princípio por trás de um Circuit Breaker é permitir que um subsistema falhe sem destruir o sistema todo. Assim, um circuito elétrico pode apresentar corrente excessiva devido a um curto-circuito, por exemplo, mas impede que a fiação queime a casa toda.

Nygard diz que, em software, podemos envolver operações perigosas

em um componente que oferece uma alternativa quando o sistema não está saudável: um Circuit Breaker. Essas operações são, em geral, chamadas a outros sistemas mas podem ser operações internas a um serviço.

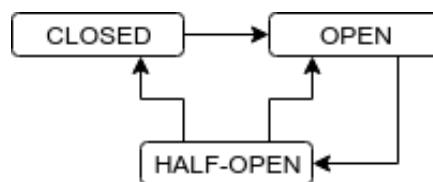
No estado fechado (em inglês, *Closed*), o Circuit Breaker executa as operações normalmente. Se houver uma falha, o Circuit Breaker a contabiliza. Falhas podem ser:

- demoras que ocasionam Timeouts
- falhas em conexões, quando o sistema a ser chamado estiver fora do ar
- retornos com erro, como o status 500 em uma chamada HTTP

Se o número (ou frequência) de falhas passa um certo limite, o Circuit Breaker fica no estado aberto (em inglês, *Open*). Uma chamada a um Circuit Breaker no estado Open falha imediatamente (*fail fast*), sem nem tentar executar a operação solicitada.

Como contar o número de falhas? Nygard argumenta que a densidade de falhas em um período de tempo é o mais importante: 5 falhas em 5 horas é muito diferente de 5 falhas nos últimos 30 segundos. Para auxiliar na implementação, Nygard menciona o pattern Leaky Bucket, em que um contador é incrementado a cada falha mas é zerado periodicamente.

Um disjuntor residencial precisa ser fechado manualmente. Em software, ao contrário, podemos automatizar o fechamento. Para isso, depois de algum tempo, o Circuit Breaker passa para o estado meio aberto (em inglês, *Half-Open*). Nesse estado, a próxima chamada é executada normalmente. Se a chamada for bem sucedida, o Circuit Breaker passa ao estado de *Closed*. Se falhar, o Circuit Breaker volta ao estado *Open*.



Vamos voltar ao caso de uma falha em cascata: um serviço

sobrecarregado é continuamente requisitado, deixando-o mais sobrecarregado ainda e exaurindo recursos do serviço que o chama. Como suavizar um serviço sobrecarregado? Falhando rapidamente! Como detectar que um serviço está sobrecarregado? Com um Timeout! Como evitar a exaustão de recursos nos serviços que o chamam? Com Circuit Breakers! O intuito final de um Circuit Breaker é suavizar um sistema sobrecarregado, evitando falhas em cascata. E com fechamento é automático.

Um Circuit Breaker evita chamadas quando uma Integração apresenta problemas. Um Timeout indica que há um problema em uma Integração.

Michael Nygard, no livro [Release It! Second Edition](#) (NYGARD, 2018)

Há diferentes frameworks que implementam Circuit Breakers:

- [Polly](#), implementado em .NET. Em: <https://github.com/App-vNext/Polly>
- [Resilience4j](#), implementa vários patterns de resiliência em Java. Em: <https://github.com/resilience4j/resilience4j>
- [Hystrix](#), implementado em Java e parte da iniciativa open-source da Netflix. Em: <https://github.com/Netflix/Hystrix>

Pattern: Circuit Breaker

Um proxy para chamadas remotas que rejeita imediatamente invocações por um período depois que falhas consecutivas ultrapassam um limite especificado.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a)

Hystrix

Hystrix é uma biblioteca open-source mantida pela Netflix que é usada para impedir falhas em cascata, auxiliar na recuperação rápida e isolar

threads. O pattern Circuit Breaker é implementado, além de funcionalidades de monitoramento que estudaremos em capítulos posteriores.

Curiosidade: Hystrix é o nome do gênero dos roedores conhecidos comumente como porcos-espinho. O animal espinhento é o slogan da biblioteca da Netflix.

É preciso estender a classe abstrata `HystrixCommand` para uma única resposta ou `HystrixObservableCommand` para um estilo reativo. Internamente, tudo é convertido para um `HystrixObservableCommand`. Há maneiras bloqueantes e não-bloqueantes de executar esse comando. Há a possibilidade de habilitar caches de respostas.

O projeto [hystrix-javanica](#), mantido pela comunidade, usa aspectos (AOP) para permitir que um método com a anotação `@HystrixCommand` seja executado em um Circuit Breaker.

O Hystrix permite a execução dos Commands em Thread Pools, o padrão, ou Semáforos. Com Semáforos é possível usar Thread Local e, em frameworks, escopos de sessão/request. A *isolation strategy* pode ser modificada através da propriedade `execution.isolation.strategy`.

Para o Hystrix, uma falha em uma requisição acontece se houve Timeout, exceção ou erro HTTP 500. Um Thread Pool ou Semáforo cheios causam uma rejeição da requisição.

O Hystrix trabalha com uma janela de tempo de 10s (configurável com `metrics.rollingStats.timeInMilliseconds`) e, se houverem mais de 20 (`circuitBreaker.requestVolumeThreshold`) falhas consecutivas em requisições, o Hystrix passa a coletar estatísticas. As métricas são coletadas em 10 (`metrics.rollingStats.numBuckets`) buckets de 1s cada, em uma técnica chamada *rolling window sampling*, que suaviza flutuações e [otimiza cálculos](#) para altos volumes de requisições. Se mais de 50% (`circuitBreaker.errorThresholdPercentage`) das requisições falharem, o Circuit Breaker fica Open (ou *tripped*). Caso o Circuit Breaker

se mantenha Closed, as estatísticas são limpas na expiração da janela de tempo.

Enquanto o Circuit Breaker estiver aberto, novas requisições já falharão, sem chegar o serviço de destino. Há uma outra janela de 5s (`circuitBreaker.sleepWindowInMilliseconds`) que passa o Circuit Breaker para Half-Open momentaneamente, deixando passar uma requisição. Se a chamada for bem sucedida, o Circuit Breaker passa para Closed. Senão, continuará Open e tentará nova requisição em mais 5s.

Com as anotações do `hystrix-javanica`, as propriedades mencionadas anteriormente podem ser modificadas da seguinte maneira:

```
@HystrixCommand(  
    commandPoolProperties = {  
        @HystrixProperty(name="circuitBreaker.requestVolumeThreshold", value="10"),  
        @HystrixProperty(name="circuitBreaker.errorThresholdPercentage", value="50"),  
        @HystrixProperty(name="circuitBreaker.sleepWindowInMilliseconds", value="10000"),  
        @HystrixProperty(name="metrics.rollingStats.timeInMilliseconds", value="10000"),  
        @HystrixProperty(name="metrics.rollingStats.numBuckets", value="30"),  
        @HystrixProperty(name="execution.isolation.strategy", value="thread")  
    }  
)
```

A documentação do Hystrix descreve o funcionamento do Circuit Breaker em detalhes: <https://github.com/Netflix/Hystrix/wiki/How-it-Works>

O projeto Hystrix não está mais sendoativamente desenvolvido e passou para modo de manutenção (não revisarão issues e não aceitarão pull requests), de acordo com a documentação da biblioteca. Na Netflix, o Hystrix é usado apenas em aplicações já existentes. Para novos projetos, a Netflix diz usar bibliotecas como Resilience4j. A documentação diz que, ao invés de configurações estáticas, o foco está em implementações adaptativas que reagem a performance em produção.

Spring Cloud Netflix Hystrix

O Spring Cloud provê o projeto `spring-cloud-starter-netflix-hystrix`, que integra o Circuit Breaker da Netflix com o ecossistema Spring.

A anotação `@EnableCircuitBreaker` deve ser adicionada à classe principal ou em alguma classe de configuração.

Para executar um método em um Circuit Breaker, devemos anotá-lo com `@HystrixCommand`, do projeto `hystrix-javanica`. Serão criados proxies dinâmicos para cada um desses métodos.

O Spring Cloud Netflix Hystrix fornece um endpoint do Spring Actuator com métricas do Hystrix, que estudaremos em capítulos posteriores.

Exercício: simulando demora no serviço de distância

1. Altere o método `calculaDistancia` da classe `DistanciaService` do serviço de distância, para que invoque o método que simula uma demora de 10 a 20 segundos:

```
##### fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/DistanciaService.java
```

```
class DistanciaService {

    // código omitido ...

    private BigDecimal calculaDistancia() {
        simulaDemora(); // modificado
        return new BigDecimal(Math.random() * 15);
    }

}
```

2. Em um Terminal, use o ApacheBench para simular a consulta da distância entre um CEP e um restaurante específico, cujos dados

são compostos no API Gateway, com 100 requisições ao todo e 10 requisições concorrentes.

O comando será parecido com o seguinte:

```
ab -n 100 -c 10 http://localhost:9999/restaurantes-com-distanc
```

A opção `-n` define o número total de requisições. A opção `-c`, o número de requisições concorrentes.

Entre os resultados aparecerá algo como:

```
Connection Times (ms)
                      min     mean[+/-sd]   median     max
Connect:        0      0       0.1      0       1
Processing:  10097  15133  2817.3  14917  19635
Waiting:      10096  15131  2817.4  14917  19632
Total:        10097  15133  2817.3  14917  19636
```

A requisição mais demorada, no exemplo anterior, foi de 19,6 segundos. Inviável!

Circuit Breaker com Hystrix

No `pom.xml` do API Gateway, adicione o *starter* do Spring Cloud Netflix Hystrix:

```
##### fj33-api-gateway/pom.xml
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

Adicione a anotação `@EnableCircuitBreaker` à classe

`ApiGatewayApplication`:

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/ApiGatewayApplication
.java
```

```
@EnableCircuitBreaker // adicionado
// demais anotações...
public class ApiGatewayApplication {
    // código omitido...
}
```

Não deixe de adicionar o import correto:

```
import org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;
```

Na classe `DistanciaRestClient` do API Gateway, habilite o *circuit breaker* no método `porCepEId`, com a anotação `@HystrixCommand`:

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/DistanciaRestClient.jav
a
```

```
@Service
class DistanciaRestClient {
```

```
// código omitido...
```

```
@HystrixCommand // adicionado
Map<String, Object> porCepEId(String cep, Long restaurante)
    String url = distanciaServiceUrl + "/restaurantes/" + cep
    return restTemplate.getForObject(url, Map.class);
}
```

```
}
```

O import é o seguinte:

```
import com.netflix.hystrix.contrib.javanica.annotation.Hystrix
```

Exercício: Circuit Breaker com Hystrix

1. Mude para a branch `cap10-circuit-breaker-com-hystrix` do projeto `fj33-api-gateway`:

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap10-circuit-breaker-com-hystrix
```

2. Reinicie o API Gateway e execute novamente a simulação com o ApacheBench, com o comando:

```
ab -n 100 -c 10 http://localhost:9999/restaurantes-com-distanc
```

Observe nos resultados uma diminuição no tempo máximo de request de 19,6 para 1,5 segundos:

	min	mean[+/-sd]	median	max
Connect:	0	0	0.7	0
Processing:	75	381	360.8	275
Waiting:	67	375	359.0	270
Total:	75	382	360.9	1558

Degradando funcionalidades com Fallbacks

Por enquanto, em uma chamada a um Circuit Breaker aberto, é lançada a exceção `HystrixRuntimeException`.

Michael Nygard, no livro [Release It! Second Edition](#) (NYGARD, 2018), discute que é preferível que a exceção seja diferente de um Timeout comum, para quem chama poder tratá-la adequadamente.

Nygard menciona que é interessante que um Circuit Breaker tenha uma resposta alternativa, um **Fallback**, como: uma resposta genérica, a última resposta bem sucedida, um valor em cache, uma resposta de um serviço secundário.

Em seu livro, Nygard afirma que qualquer estratégia de Fallback pode ter impacto nos negócios, já que há uma *degradação automática das funcionalidades*. Por isso, é essencial involver os *stakeholders* ao decidir a estratégia a ser tomada. O autor exemplifica: um sistema de varejo deve aceitar um pedido se não pode confirmar a disponibilidade dos itens em estoque? E se o cartão de crédito não puder ser verificado? Um Fallback pode ser uma maneira interessante de abordar esse tipo de requisito.

No livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman foca na ideia de que a maneira correta de degradar as funcionalidades de um sistema em caso de falha em um serviço não é uma discussão técnica. É necessário saber o que é tecnicamente possível mas a ação a ser tomada deve ser orientada pelo contexto de negócio. Em um e-commerce, por exemplo, no caso de um serviço de checkout estar fora do ar, é possível manter a listagem do catálogo e colocar um telefone para que os clientes possam fechar a compra.

Fallback no @HystrixCommand

Se acessarmos repetidas vezes, em um navegador, a URL a seguir:

<http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1>

Deve ocorrer, em algumas das vezes, uma exceção semelhante a:

```
There was an unexpected error (type=Internal Server Error, status=500).
route:SendForwardFilter
com.netflix.hystrix.exception.HystrixRuntimeException: porCepEId timed-out
    at com.netflix.hystrix.AbstractCommand$22.call(AbstractCommand.java:832)
    at com.netflix.hystrix.AbstractCommand$22.call(AbstractCommand.java:807)
    at rx.internal.operators.OperatorOnErrorResumeNextViaFunction$4.onError(0
...

```

A mensagem da exceção (*porCepEId timed-out and fallback failed*), indica que houve um erro de timeout.

Em outras tentativas, teremos uma exceção semelhante, mas cuja mensagem indica que o Circuit Breaker está aberto e a resposta foi *short-circuited*, não chegando a invocar o serviço de destino da requisição:

```
There was an unexpected error (type=Internal Server Error, status=500).
route:SendForwardFilter
com.netflix.hystrix.exception.HystrixRuntimeException: porCepEId short-circuited
    at com.netflix.hystrix.AbstractCommand$22.call(AbstractCommand.java:832)
    at com.netflix.hystrix.AbstractCommand$22.call(AbstractCommand.java:807)
    at rx.internal.operators.OperatorOnErrorResumeNextViaFunction$4.onError(0
...

```

É possível fornecer um *fallback*, passando o nome de um método na propriedade `fallbackMethod` da anotação `@HystrixCommand`.

Defina o método `restauranteSemDistanciaNemDetalhes`, que retorna apenas o restaurante com o id. Se a outra parte da API Composition, a interface `RestauranteRestClient` não der erro e retornar os dados do restaurante, teríamos todos os detalhes do restaurante menos a distância.

fj33-api-

gateway/src/main/java/br/com/caelum/apigateway/DistanciaRestClient.java

```
@Service
class DistanciaRestClient {

    // código omitido...

    @HystrixCommand(fallbackMethod="restauranteSemDistanciaNemDetalhes")
    Map<String, Object> porCepEId(String cep, Long restauranteId) {
        String url = distanciaServiceUrl+"/restaurantes/"+cep+"/restaurante";
        return restTemplate.getForObject(url, Map.class);
    }

    // método adicionado
    Map<String, Object> restauranteSemDistanciaNemDetalhes(String cep) {
        Map<String, Object> resultado = new HashMap<>();
        resultado.put("restauranteId", restauranteId);
        resultado.put("cep", cep);
        return resultado;
    }
}
```

O seguinte import deve ser adicionado:

```
import java.util.HashMap;
```

Observação: uma solução interessante seria manter um cache das distâncias entre CEPs e restaurantes e usá-lo como fallback, se possível. Porém, a *hit ratio*, a taxa de sucesso das consultas ao cache, deve ser baixa, já que os CEPs dos clientes mudam bastante.

Exercício: Fallback com Hystrix

1. Acesse repetidas vezes, em um navegador, a URL a seguir:

<http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1>

Deve ocorrer, em algumas das vezes, uma exceção

HystrixRuntimeException com as mensagens:

- *porCepEId timed-out and fallback failed.*
- *porCepEId short-circuited and fallback failed.*

2. No projeto `fj33-api-gateway`, obtenha o código da branch `cap10-fallback-no-hystrix-command`:

```
cd ~/Desktop/fj33-api-gateway
git checkout -f cap10-fallback-no-hystrix-command
```

Reinicie o API Gateway.

3. Tente acessar várias vezes a URL testada anteriormente:

<http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1>

Observe que não ocorre mais uma exceção, mas não há a informação de distância. Apenas os detalhes do restaurante são retornados.

Algo semelhante a:

```
{
  "id": 1,
  "cnpj": "98444252000104",
  "nome": "Long Fu",
  "descricao": "O melhor da China aqui do seu lado.",
  "cep": "71503510",
  "endereco": "ShC/SUL COMERCIO LOCAL QD 404-BL D LJ 17-ASA SL",
  "taxaDeEntregaEmReais": 6,
```

```
"tempoDeEntregaMinimoEmMinutos": 40,  
"tempoDeEntregaMaximoEmMinutos": 25,  
"aprovado": true,  
"tipoDeCozinha": {  
    "id": 1,  
    "nome": "Chinesa"  
},  
"restauranteId": 1  
}
```

Exercício: Removendo simulação de demora do serviço de distância

1. Comente a chamada ao método que simula a demora em `DistanciaService` do `eats-distancia-service`. Veja se, quando não há demora, a distância volta a ser incluída na resposta.

```
##### fj33-eats-distancia-  
service/src/main/java/br/com/caelum/eats/distancia/DistanciaService.java
```

```
class DistanciaService {  
  
    // código omitido ...  
  
    private BigDecimal calculaDistancia() {  
        //simulaDemora(); // modificado  
        return new BigDecimal(Math.random() * 15);  
    }  
  
}
```

Exercício: Simulando demora no monólito

1. Altere o método `detalha` da classe `RestauranteController` do monólito para que tenha uma espera de 20 segundos:

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/RestauranteC
ontroller.java
```

```
// anotações ...
class RestauranteController {

    // código omitido ...

    @GetMapping("/restaurantes/{id}")
    RestauranteDto detalha(@PathVariable("id") Long id) {

        // trecho de código adicionado ...
        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        Restaurante restaurante = restauranteRepo.findById(id).orE
        return new RestauranteDto(restaurante);
    }

    // restante do código ...
}
```

Circuit Breaker com Hystrix no Feign

No `application.properties` do API Gateway, é preciso adicionar a seguinte linha:

```
##### fj33-api-gateway/src/main/resources/application.properties

feign.hystrix.enabled=true
```

A integração entre o Feign e o Hystrix vem desabilitada por padrão, nas versões mais recentes. Por isso, é necessário habilitá-la.

Exercício: Integração entre Hystrix e Feign

1. Faça o checkout da branch cap10-circuit-breaker-com-hystrix-no-feign do projeto fj33-api-gateway:

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap10-circuit-breaker-com-hystrix-no-feign
```

Reinic peace o API Gateway.

2. Tente acessar novamente a URL:

<http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1>

Observação: a URL anterior, além de obter a distância do serviço apropriado, obtém os detalhes do restaurante do monólito utilizando o Feign na implementação do cliente REST.

Deve ocorrer a seguinte exceção:

Quando o Circuit Breaker estiver aberto, a mensagem da exceção `HystrixRuntimeException` será um pouco diferente:

RestauranteRestClient#porId(Long) short-circuited and no fallback available.

Fallback com Feign

No Feign, definimos de maneira declarativa o cliente REST, por meio de uma interface.

A estratégia de Fallback na integração entre Hystrix e Feign é fornecer uma implementação para essa interface. Engenhoso!

No api-gateway, crie uma classe `RestauranteRestClientFallback`, que implementa a interface `RestauranteRestClient`. No método `porId`, deve ser fornecida uma lógica de fallback para o detalhamento de um restaurante. Anote essa nova classe com `@Component`, para que seja gerenciada pelo Spring.

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/RestauranteRestClient
Fallback.java
```

```
@Component
class RestauranteRestClientFallback implements RestauranteRest

    @Override
    public Map<String, Object> porId(Long id) {
        Map<String, Object> resultado = new HashMap<>();
        resultado.put("id", id);
        return resultado;
    }

}
```

A seguir, estão os imports corretos:

```
import java.util.HashMap;
import java.util.Map;

import org.springframework.stereotype.Component;
```

Observação: Uma solução mais interessante seria manter um cache dos dados dos restaurantes, com o `id` como chave, que seria usado em caso de fallback. Nesse caso, a *hit ratio*, a taxa de sucesso das consultas ao cache, seria bem alta: há um número limitado de restaurantes, que são escolhidos repetidas vezes, e os dados são raramente alterados.

Altere a anotação `@FeignClient` de `RestauranteRestClient`, passando na propriedade `fallback` a classe criada no passo anterior.

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/RestauranteRestClient.
java
```

```
@FeignClient("-monolito")
@FeignClient(name = "monolito", fallback=RestauranteRestClient
interface RestauranteRestClient {

    @GetMapping("/restaurantes/{id}")
    Map<String, Object> porId(@PathVariable("id") Long id);

}
```

Exercício: Fallback com Feign

1. Vá até a branch `cap10-fallback-com-feign` do projeto `fj33-api-gateway`:

```
cd ~/Desktop/fj33-api-gateway
git checkout -f cap10-fallback-com-feign
```

Certifique-se que o API Gateway foi reiniciado.

2. Por mais algumas vezes, tente acessar a URL:

<http://localhost:9999/restaurantes-com->

[distancia/71503510/restaurante/1](http://localhost:9999/distancia/71503510/restaurante/1)

Veja que são mostrados apenas o id e a distância do restaurante. Os demais campos não são exibidos.

Exercício: Removendo simulação de demora do monólito

1. Remova da classe `RestauranteController` do monólito, a simulação de demora.

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/RestauranteC
ontroller.java
```

```
// anotações ...
class RestauranteController {

    // código omitido ...

    @GetMapping("/restaurantes/{id}")
    public RestauranteDto detalha(@PathVariable("id") Long id) {

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        Restaurante restaurante = restauranteRepo.findById(id).orElse(
            return new RestauranteDto(restaurante);
    }
}
```

Teste novamente a URL: <http://localhost:9999/distancia/71503510/restaurante/1>

Os detalhes do restaurante devem voltar a ser exibidos!

Retry e Backoff

Uma outra maneira de lidar com falhas é tentar novamente (em inglês, **Retry**) de maneira automática.

Um Retry ajuda no caso de um erro intermitente, como uma falha de conexão com um BD ou um pacote descartado na rede. Para erros persistentes ou sistêmicos, um Retry não é uma boa opção.

No livro [Microservices in Action](#) (BRUCE; PEREIRA, 2018), os autores indicam que um Retry pode minimizar o impacto para os usuários e evitar a necessidade de intervenção pelo time de Operações. Os autores também mencionam que operações idempotentes, que ao serem repetidas têm o mesmo efeito, são mais compatíveis com um Retry.

Michael Nygard, no livro [Release It! Second Edition](#) (NYGARD, 2018), afirma que Circuit Breakers são diferentes de Retries, já que os Circuit Breakers impedem que operações sejam executadas ao invés de refazê-las.

Um Circuit Breaker visa suavizar um sistema sobrecarregado. Um Retry o tornaria mais sobrecarregado ainda.

Como mencionam os autores do livro [Microservices in Action](#) (BRUCE; PEREIRA, 2018), no caso de um erro persistente, para evitar que os retries contribuam para uma falha em cascata, é interessante espalhar a carga feita pelas novas tentativas. Para isso, inserimos um tempo de espera entre as tentativas. Isso é conhecido como **Backoff**. O BackOff pode ser exponencial, esperando entre os Retries 2s, 4s, 8s, 16s e assim sucessivamente.

Ainda assim, pode ser que os Exponential Backoffs sejam amplificados, criando *multidões trovejantes de Retries sincronizados*, como escrevem os autores do [Microservices in Action](#) (BRUCE; PEREIRA, 2018). Para evitar esses Retries sincronizados, os autores afirmam que é interessante que um Backoff inclua um elemento randômico, comumente chamado de

Jitter.

No [artigo de lançamento do Zuul 2](#), a Netflix menciona um fato interessante: os Retry Storms, em que os usuários ocasionam um aumento drástico de requisições logo depois de um problema. Quem nunca clicou sem parar em um botão após um erro?

Spring Retry

O Spring Retry era um componente do projeto Spring Batch que foi extraído e tornou-se um projeto próprio.

Para utilizá-lo, basta adicionar como dependência o artefato `spring-retry` do grupo `org.springframework.retry`. No Maven:

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
```

Em um projeto Spring Boot, devemos adicionar `@EnableRetry` na classe principal ou em alguma classe de configuração.

Para que os métodos que, em caso de exceção, tenham novas tentativas automáticas, basta anotá-los com `@Retryable`. Na propriedade `backoff` dessa anotação, é possível passar um `@Backoff`.

As bibliotecas do projeto Spring Cloud Netflix tem [integração com o Spring Retry](#). A simples presença no Classpath da biblioteca de Retry já faz com que várias bibliotecas façam novas tentativas.

O Ribbon possibilita Retries automáticos e é configurável pelas propriedades:

- `servico.ribbon.MaxAutoRetries`, indica o número máximo de novas tentativas em um mesmo servidor, excluindo a primeira tentativa

- `servico.ribbon.MaxAutoRetriesNextServer`, indica o número máximo de servidores para usar num Retry, excluindo o primeiro servidor
- `servico.ribbon.OkToRetryOnAllOperation`, para que todas as operações, incluindo requisições POST, tenham novas tentativas

É possível ainda fazer com que o Ribbon faça Retries para status codes específicos com a propriedade `servico.ribbon.retryableStatusCodes`.

Não há Backoff nos Retries do Ribbon. Porém, é possível configurar políticas de Backoff fornecendo um @Bean do tipo `LoadBalancedRetryFactory`, do Spring Cloud Commons, que deve instanciar uma implementação da interface `BackoffPolicy`, do Spring Retry.

O Zuul usa as estratégias de Retry configuradas no Ribbon. Para desligá-las, basta desabilitar a propriedade `zuul.retryable`. É possível desabilitar o Retry para um rota específica com `zuul.routes.servico.retryable`.

Exercício: Forçando uma exceção no serviço de distância

1. No serviço de distância, force o lançamento de uma exceção no método `atualiza` da classe `RestaurantesController`.

Comente o código que está depois da exceção.

```
##### fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/RestaurantesControll
er.java
```

```
// anotações ...
class RestaurantesController {

    // código omitido ...
```

```
@PutMapping("/restaurantes/{id}")
Restaurante atualiza(@PathVariable("id") Long id, @RequestBody Restaurante restaurante) {
    if (restaurante == null) {
        throw new RuntimeException();
    }
    // código comentado ...
    //if (!repo.existsById(id)) {
    //    throw new ResourceNotFoundException();
    //}
    //log.info("Atualiza restaurante: " + restaurante);
    //return repo.save(restaurante);
}
}
```

Tentando novamente com Spring Retry

No módulo eats-restaurante do monólito, adicione o Spring Retry como dependência:

```
##### fj33-eats-monolito-modular/eats/eats-restaurante/pom.xml
```

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
```

Adicione a anotação `@EnableRetry` na classe `EatsApplication` do módulo eats-application do monólito:

```
##### fj33-eats-monolito-modular/eats/eats-
application/src/main/java/br/com/caelum/eats/EatsApplication.java
```

```
@EnableRetry // adicionado
// outra anotações
```

```
public class EatsApplication {  
    // código omitido...  
}
```

Faça o import adequado:

```
import org.springframework.retry.annotation.EnableRetry;
```

Adicione a anotação `@Slf4j` à classe `DistanciaRestClient`, do módulo `eats-restaurante` do monólito, para configurar um logger que usaremos a seguir:

```
##### fj33-eats-monolito-modular/eats/eats-  
restaurante/src/main/java/br/com/caelum/eats/restaurante/DistanciaRest  
Client.java
```

```
@Slf4j // adicionado  
@Service  
public class DistanciaRestClient {  
    // código omitido...  
}
```

O import é o seguinte:

```
import lombok.extern.slf4j.Slf4j;
```

Em seguida, anote o método `restauranteAtualizado` com `@Retryable` para que faça 5 tentativas, logando as tentativas de acesso:

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/DistanciaRest
Client.java
```

```
// anotações ...
public class DistanciaRestClient {

    // código omitido ...

    @Retryable(maxAttempts=5) // adicionado
    public void restauranteAtualizado(Restaurante restaurante) {
        log.info("monólito tentando chamar distancia-service");

        // código omitido ...
    }

}
```

Certifique-se que o import correto foi realizado:

```
import org.springframework.retry.annotation.Retryable;
```

Exercício: Spring Retry

1. Faça o checkout da branch `cap10-retry` do monólito:

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap10-retry
```

Reinic peace o monólito.

2. Garanta que o monólito, o serviço de distância e que a UI estejam no ar.

Faça login como dono de um restaurante (por exemplo, longfu/123456) e mude o CEP ou tipo de cozinha.

Perceba que nos logs que foram feitas 5 tentativas de chamada ao serviço de distância. Algo como o que segue:

```
2019-06-18 17:30:42.943  INFO 12547 --- [nio-8080-exec-9] b.c.c.e.restaurant
2019-06-18 17:30:43.967  INFO 12547 --- [nio-8080-exec-9] b.c.c.e.restaurant
2019-06-18 17:30:44.990 INFO 12547 --- [nio-8080-exec-9] b.c.c.e.restaurant
2019-06-18 17:30:46.034 INFO 12547 --- [nio-8080-exec-9] b.c.c.e.restaurant
2019-06-18 17:30:46.085 ERROR 12547 --- [nio-8080-exec-9] o.a.c.c.C.[.][/]
org.springframework.web.client.HttpServerErrorException$InternalServerError
at org.springframework.web.client.HttpServerErrorException.create(HttpSer
...

```

Exponential Backoff

Vamos configurar um backoff para ter um tempo progressivo entre as tentativas de 2, 4, 8 e 16 segundos:

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/DistanciaRest
Client.java
```

```
// anotações ...
public class DistanciaRestClient {

    // código omitido ...

    @Retryable(maxAttempts=5)
    @Retryable(maxAttempts=5, backoff=@Backoff(delay=2000, multipl
    public void restauranteAtualizado(Restaurante restaurante) {
        // código omitido ...
    }

}
```

O import a seguir deve ser adicionado:

```
import org.springframework.retry.annotation.Backoff;
```

Para adicionar *jitter*, evitando Retries sincronizados, podemos habilitar a propriedade `random`.

```
@Backoff(delay=2000,multiplier=2, random=true) )
```

Exercício: Exponential Backoff com Spring Retry

1. Vá até a branch `cap10-backoff` do projeto `fj33-eats-monolito-modular`:

```
cd ~/Desktop/fj33-eats-monolito-modular  
git checkout -f cap10-backoff
```

2. Pela UI, faça novamente o login como dono de um restaurante (por exemplo, com `longfu/123456`) e modifique o CEP ou tipo de cozinha.

Note o tempo progressivo nos logs. Será alguma coisa semelhante a:

```
2019-06-18 18:00:18.367  INFO 15044 --- [nio-8080-exec-8] b.c.c.e.restaurant
...
2019-06-18 18:00:20.973  INFO 15044 --- [nio-8080-exec-8] b.c.c.e.restaurant
2019-06-18 18:00:24.994  INFO 15044 --- [nio-8080-exec-8] b.c.c.e.restaurant
2019-06-18 18:00:33.047  INFO 15044 --- [nio-8080-exec-8] b.c.c.e.restaurant
2019-06-18 18:00:49.079  INFO 15044 --- [nio-8080-exec-8] b.c.c.e.restaurant
2019-06-18 18:00:49.127 ERROR 15044 --- [nio-8080-exec-8] o.a.c.c.C.[.[]]
...
```

Exercício: Removendo exceção forçada do serviço de distância

1. Agora que testamos o retry e o backoff, vamos remover a exceção que forçamos anteriormente na classe RestaurantesController do serviço de distância:

```
##### fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/RestaurantesControll
er.java
```

```
// anotações ...
class RestaurantesController {

    // código omitido ...

    @PutMapping("/restaurantes/{id}")
    Restaurante atualiza(@PathVariable("id") Long id, @RequestBc

    throw new RuntimeE-xception();

    // descomente o código abaixo ...

    if (!repo.existsById(id)) {
        throw new ResourceNotFoundException();
    }
    log.info("Atualiza restaurante: " + restaurante);
    return repo.save(restaurante);
}

}
```

Patterns de Resiliência: um resumo

A [documentação do Resilience4j](#), traz um resumo dos patterns de Resiliência de maneira divertida:

Retry

- *Como funciona?* Repete execuções que falharam.
- *Descrição:* Muitas falhas são transientes e podem ser autocorrigidas depois de um pequeno período.
- *Slogans:* "Insira uma moeda para tentar de novo" ou "Talvez seja só um tilt"

Circuit Breaker

- *Como funciona?* Bloqueia temporariamente possíveis falhas.
- *Descrição:* Quando um sistema está sofrendo seriamente, é melhor fracassar rapidamente (*Fail Fast*) do que fazer com que os clientes esperem.
- *Slogans:* "Dá uma folga pra aquele sistema" ou "Baby, don't hurt me, no more"

Rate Limiter

- *Como funciona?* Limita execuções por período.
- *Descrição:* Prepare-se para uma escala e estabeleça a Confiabilidade e a Alta Disponibilidade de seu serviço.
- *Slogans:* "Tá bom para esse minuto!" ou "Bom, vai funcionar da próxima vez"

Timeout

- *Como funciona?* Limita a duração de uma execução.
- *Descrição:* Depois de um certo tempo, um resultado bem sucedido é improvável.
- *Slogans:* "Não espere para sempre"

Bulkhead

- *Como funciona?* Limita execuções concorrentes.
- *Descrição:* Recurso são isolados em pools de maneira que, se algum

falhar, os outros continuarão.

- *Slogans*: "Uma falha não deveria afundar o navio todo" ou "Por favor, não todos de uma vez"

Cache

- *Como funciona?* Memoriza um resultado bem sucedido.
- *Descrição*: Alguma proporção das requisições pode ser semelhante.
- *Slogans*: "Você já pediu esse"

Fallback

- *Como funciona?* Provê um resultado alternativo para falhas.
- *Descrição*: As coisas ainda falharão - planeje o que você fará quando isso acontecer.
- *Slogans*: "Degrade graciosamente" ou "Um pássaro na mão é melhor que dois voando"

Para saber mais: Sidecar

Usamos bibliotecas da Netflix como Ribbon, Eureka e Hystrix em projetos Java. Poderíamos usá-las programas escritos em Groovy, Scala, Clojure ou Kotlin, já todas essas são linguagens que tem a JVM como plataforma. Mas e se tivermos microservices implementados em NodeJS, Python ou Go? É o caso da própria Netflix, que tem serviços em todas essas tecnologias.

Para integrar as bibliotecas mencionadas com linguagens que não rodam na JVM, a estratégia da Netflix foi criar um serviço feito em Java que oferece, por meio de uma API HTTP, funcionalidades como Client Side Load Balancing, Self Registration, Client Side Discovery, Timeouts, Circuit Breakers e Fallbacks. Esse serviço foi chamado de [Prana](#).

Em termos de implantação, toda aplicação de linguagens não-JVM, tem o Prana instalado "ao lado", na mesma máquina (ou container). Por isso, o Prana é chamado de **Sidecar**, uma referência aos carros auxiliares

anexados a algumas motocicletas.

Pattern: Sidecar

Implementa preocupações transversais em um processo ou container que é executado ao lado da instância de um serviço

Chris Richardson, no livro [Microservice Patterns](#) (RICHARDSON, 2018a)

No fim das contas, requisitos não-funcionais como Escalabilidade, Disponibilidade e Resiliência passam da aplicação para o Sidecar. Poderíamos remover responsabilidades inclusive de aplicações escritas em Java!

Chris Richardson cita, no livro [Microservice Patterns](#) (RICHARDSON, 2018a), que um proxy como o [Envoy](#), implementado pela Lyft, é comumente implantado como um Sidecar.

O [Spring Cloud Netflix Sidecar](#) não usa o Prana, mas implementa o mesmo conceito, integrando com diversas outras ferramentas do Spring Cloud.

Para saber mais: Service Mesh

No livro [Microservice Patterns](#) (RICHARDSON, 2018a), Chris Richardson recomenda usar um Microservice Chassis, um framework ou coleção de frameworks, para implementar questões transversais às funcionalidades da aplicação.

Mas, para Richardson, um obstáculo é que frameworks são restritos a uma plataforma específica. É possível usar Spring Boot e Spring Cloud em aplicações escritas em Java ou Kotlin (e talvez em outras linguagens da JVM). Mas se alguns serviços forem escritos em Go, Elixir ou NodeJS, precisaremos de Microservices Chassis específicos.

Uma alternativa emergente identificada por Chris Richardson é

implementar essas preocupações transversais na própria infraestrutura de redes, mediando tanto a comunicação entre os serviços como as requisições de clientes externos: é um **Service Mesh**.

Algumas implementações de Service Meshes:

- Istio (<https://istio.io>), iniciado pela Google, IBM/Red Hat e Lyft
- Linkerd (<https://linkerd.io>), da Buoyant, incubado na Cloud Native Computing Foundation (CNCF)
- Consul (<https://www.consul.io/>), da HashiCorp
- Maesh (<https://containo.us/maesh/>), da Containous, criadora do edge router Trafik

Pattern: Service Mesh

Roteie todo o tráfego de rede dos serviços através de uma camada de rede que implementa questões como Circuit Breakers, Distributed Tracing, Service Discovery, Load Balancing, Criptografia, entre outros.

Chris Richardson, No livro [Microservice Patterns](#) (RICHARDSON, 2018a)

Para Richardson, com um Service Mesh, as implementações necessárias em um Microservice Chassis são minimizadas a Configurações Externalizadas, Health Checks e propagação de informações de Distributed Tracing.

Diz Richardson: um Service Mesh como o Istio integra-se muito bem com um Orquestrador de Containers como o Kubernetes. O Istio usa o proxy Envoy como um Sidecar de um serviço, em um container no mesmo Pod.

No artigo [Why Kubernetes is The New Application Server](#) (BENEVIDES, 2018), compara o Kubernetes junto ao Istio e ao Open Shift a um antigo Servidor de Aplicação do Java EE: o intuito era extrair da aplicação código de requisitos não-funcionais. Benevides faz uma analogia: o Servidor de Aplicação é um CD Player e a aplicação é um CD; uma imagem de um container com a aplicação é um novo formato de CD e o

Kubernetes é um CD Player novo que provê as capacidades necessárias. E há uma grande vantagem: é multi-plataforma, já que o Kubernetes abstrai a infraestrutura.

Benevides argumenta que essa nova plataforma, com Kubernetes, Istio e Open Shift, provê 9 capacidades importantes para implantar em produção aplicações robustas:

- Service Discovery
- Integração
- Elasticidade
- Logging
- Monitoramento
- CI/CD Pipelines
- Resiliência
- Autenticação
- Tracing

Uma mentalidade antifrágil

No livro [Antifragile](#) (TALEB, 2012), Nassim Taleb questiona: qual seria o antônimo de frágil? Robusto, resiliente, sólido? Não! O antônimo de positivo é negativo, não neutro. O exato oposto de frágil seria algo que se beneficia ao receber pancadas, como um pacote com os dizeres "Por favor, lide sem nenhum cuidado". É o que Taleb chama de **antifrágil**.

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), cita os *game days*, simulações em que sistemas são desligados e os times reagem. Newman diz que a Google vai além e tem os exercícios DiRT (*Disaster Recovery Test*), em que há simulações de desastres em larga escala como terremotos.

Em seu livro, Newman cita um [artigo em que Ariel Tseitlin](#) argumenta que a Netflix é a organização *antifrágil*. Tseitlin revela que a Netflix escreve programas que causam erros e os executa *em produção* diariamente. Falhas são incitadas para assegurar que os sistemas da Netflix as

toleram.

A Netflix tem programas como o [Chaos Monkey](#), que desliga máquinas aleatoriamente. O que seria um evento raro, uma catástrofe, passa a ser comum. Isso acontece em produção! Assim, os sistemas precisam ser projetados, implementados e configurados para lidar com essas calamidades corriqueiras. O Chaos Gorilla era um projeto que tirava do ar um data center inteiro (ou o equivalente na AWS). Já o Latency Monkey simulava conexões de rede extremamente lentas. O conjunto dos projetos era conhecido como [Simian Army](#), mas foi descontinuado. Apenas o Chaos Monkey continua sendo ativamente mantido.

Newman relata que, para dar suporte a essa incitação de falhas, a Netflix foca na importância do aprendizado, uma cultura não focada nos culpados e no empoderamento dos desenvolvedores.

Os desenvolvedores da Netflix iniciaram um movimento que chamam de [Chaos Engineering](#): *Chaos Engineering é a disciplina de realizar experimentos sobre sistemas distribuídos com o intuito de construir confiança com relação a capacidade de um sistema distribuído suportar condições adversas em produção.*

Service Meshes, que controlam os serviços na própria infraestrutura, como o [Istio](#) e [Linkerd](#), dão suporte ao que chamam de *Fault Injection*.

Mensageria e Eventos

Um serviço de geração de notas fiscais

Um outro time da Diretoria Financeira do Caelum Eats, alinhado com os especialistas contábeis, preparou um Microservice para a geração de notas fiscais.

Ao receber os ids dos Aggregates `Pagamento` e `Pedido`, os detalhes do pedido são solicitados ao Monólito e usados para gerar um XML com a nota fiscal.

A busca dos detalhes do pedido do Monólito é implementada com o OpenFeign.

O endereço das instâncias disponíveis é obtida .

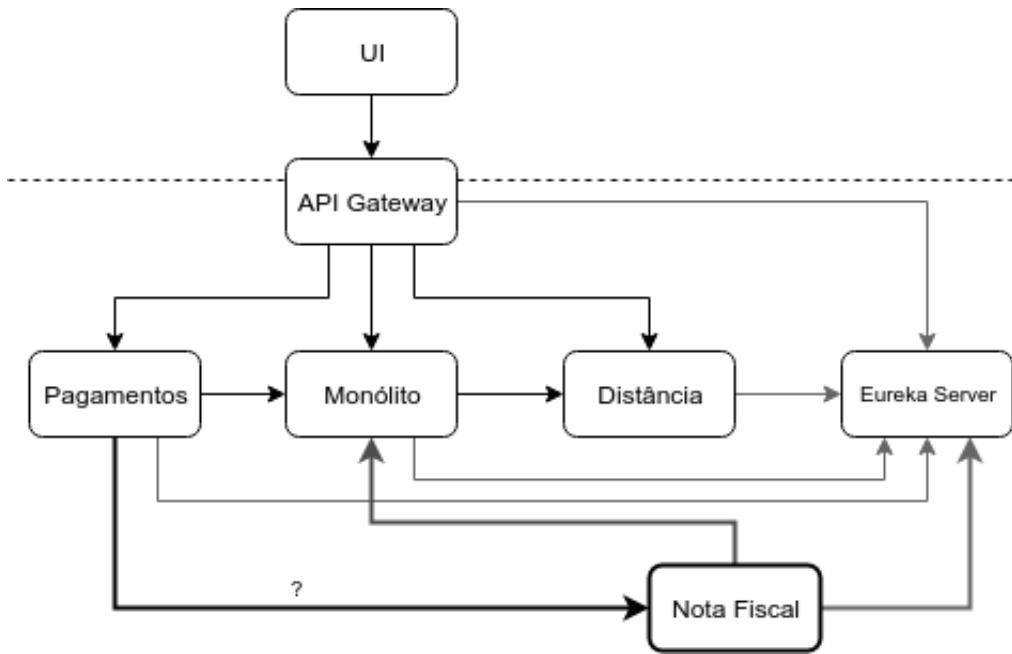
A geração do XML é feita com a biblioteca FreeMarker.

A classe que gerencia a emissão das notas fiscais é a `ProcessadorDePagamentos` que, dados os ids de um pagamento e de um pedido, obtém os detalhes do pedido do monólito usando o Feign.

A nota fiscal deve ser gerada assim que um Pagamento for confirmado.

Será que teremos que colocar mais um cliente Feign ou RestTemplate no serviço de Pagamento, para invocar o serviço de Nota Fiscal? E teremos que usar bibliotecas como Ribbon, Eureka Client e Hystrix para Load Balancing, Service Discovery e Resiliência.

Há uma outra maneira de fazer essa implementação?



Exercício: um serviço de nota fiscal

1. Baixe o projeto do serviço de nota fiscal para seu Desktop usando o Git, com os seguintes comandos:

```
cd ~/Desktop
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-nc
```

2. Abra o Eclipse, usando o workspace dos microservices.
3. No Eclipse, acesse *File > Import > Existing Maven Projects* e clique em *Next*. Em *Root Directory*, aponte para o diretório clonado anteriormente.

Uma outra forma de comunicação

Já tentou alugar um imóvel?

Vamos dizer que você não teve muito sucesso nos aplicativos. O que fazer? Primeiro, conseguir uma lista de imobiliárias e corretores na região desejada. Com essa lista em mãos, seria necessário ligar para cada um dos telefones obtidos. A experiência é frustante: ou ninguém atende os telefones; ou a espera é grande para ser atendido; demoramos pra saber que não há imóveis na região ou que os imóveis ultrapassam nosso

orçamento. E por aí vai... E o pior de tudo é que, ao passar todo esse tempo no telefone, não é possível fazer outras coisas no trabalho ou em casa.

Será que há outro jeito?

Dos telefones obtidos, a maioria é celular. E a maioria tem WhatsApp. E se mandássemos mensagens para as imobiliárias e corretores?

Poderíamos enviar várias mensagens e esquecer do assunto, voltando a realizar as tarefas profissionais ou domésticas. Assim que estivermos livres, podemos ver as respostas, sejam textos ou (os famigerados) áudios.

Comunicação síncrona x assíncrona

Um telefonema é um estilo de comunicação síncrona. Só é possível haver comunicação se a outra parte estiver disponível.

Uma conversa no WhatsApp é um estilo de comunicação **assíncrona**. Alguém envia uma mensagem sem que, necessariamente, o outro lado esteja disponível. Mesmo numa ligação telefônica, há uma maneira assíncrona de comunicação: o correio de voz (em inglês, *voice mail*), em que deixamos recados que podem ser lidos depois.

É possível que uma mesma mensagem seja enviada para diferentes destinatários. Por exemplo, é o que acontece num grupo de família do WhatsApp.

Mas como é possível mandar uma mensagem sem que o destinatário esteja lendo no mesmo momento? Por meio de um **intermediário**. No caso das mensagens de celular, os servidores do WhatsApp fazem essa intermediação. No caso do correio de voz, os servidores das operadoras.

Mensageria

Ciência da Computação é a disciplina que acredita que todos os problemas podem ser resolvidos com mais uma camada de indireção.

Dennis DeBruler, citado por Kent Beck no livro Refactoring (FOWLER et al., 1999)

No livro [Enterprise Integration Patterns](#) (HOHPE; WOOLF, 2003), Gregor Hohpe e Bobby Woolf exploram com detalhes a **Mensageria**: *uma tecnologia que permite comunicação de alta velocidade e assíncrona, programa a programa, com entrega confiável.*

Pattern: Mensageria

Um cliente invoca um serviço usando Mensageria Assíncrona.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a)

Hohpe e Woolf explicam que, em um sistema de Mensageria, o intermediário que provê capacidades de Mensageria é chamado de **Message Broker** ou **Message-Oriented Middleware** (MOM). Um Message Broker pode usar redundância para prover Alta Disponibilidade, Performance e Qualidade de Serviço (em inglês, Quality of Service ou QoS).

Existem diversos Message Brokers no mercado, entre eles:

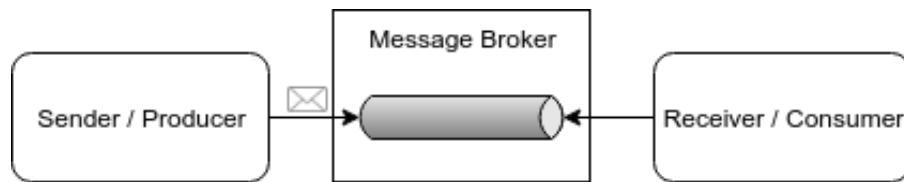
- [IBM MQ](#), o antigo MQSeries
- [TIBCO Messaging](#)
- [Microsoft BizTalk Server](#)
- [AWS Simple Queue Service](#), uma solução mais limitada
- [RabbitMQ](#), implementado em Erlang
- [Apache ActiveMQ](#), que tem a versão "clássica" e a Artemis, baseada no antigo HornetQ da JBoss/Red Hat
- [Apache Kafka](#), que, além de um Message Broker, pode ser usado de outras maneiras

Um broker possui Canais (em inglês, **Channels**), caminhos lógicos que conectam os programas e transmitem mensagens.

Um remetente (em inglês, **Sender**) ou produtor (em inglês, **Producer**) é um programa que envia uma mensagem a um Channel.

Um receptor (em inglês, **Receiver**) ou consumidor (em inglês, **Consumer**) é um programa que recebe uma mensagem lendo, e excluindo, de um Channel.

Os Consumers que filtram as mensagens de um Channel, recebendo apenas as que atendem a um determinado critério são chamados de **Selective Consumers**.



Prós e Contras da Mensageria

Para Gregor Hohpe e Bobby Woolf, ainda no livro [Enterprise Integration Patterns](#) (HOHPE; WOOLF, 2003), entre os benefícios da Mensageria estão:

- *Comunicação assíncrona*: o modelo Send-and-Forget permite que o Producer precise somente esperar que a mensagem seja armazenada no Channel, sem estar atrelado ao Consumer.
- *Taxa de Transferência Máxima*: em uma comunicação síncrona, o Producer aguarda o resultado do Consumer e, por isso, as chamadas são tão rápidas quanto o Consumer pode processá-la. Já no modelo assíncrono da Mensageria, o Producer e o Consumer podem trabalhar em ritmos diferentes. Não há tempo perdido esperando um pelo outro, levando a uma taxa de transferência máxima (em inglês, *maximum throughput*).
- *Throttling*: um problema com as chamadas síncronas é que muitas delas ao mesmo tempo a um único Consumer podem sobrecarregá-lo, causando degradação no desempenho ou falhas. Como o Message Broker enfileira requests até que o Consumer esteja pronto para processá-las, o Consumer pode controlar a taxa na qual

consume, para não ficar sobrecarregado. Os Producers não são afetados por essa limitação porque a comunicação é assíncrona, portanto, não ficam bloqueados.

- *Comunicação Confiável*: os dados são empacotados como mensagens atômicas e independentes transmitidas com a estratégia Store-and-Forward. Se as mensagens forem armazenadas em disco ao invés da memória, temos Entrega Garantida (em inglês, *Guaranteed Delivery*). Problemas com a rede ou com o computador do Consumer são superados com uma (ou mais) nova tentativa automática (em inglês, *automatic retry*).
- *Operação Desconectada*: algumas aplicações são executadas desconectadas de uma rede, sincronizando com servidores quando uma conexão estiver disponível. Por exemplo, um geólogo faz medições no solo em lugares remotos e, quando volta ao escritório, tem os novos dados sincronizados. Uma das maneiras de implementar é usando Mensageria.
- *Mediação*: uma aplicação pode depender do Message Broker, ao invés de depender diretamente de várias outras aplicações. Assim, no caso de algum problema, basta reconectar ao Broker.
- *Gerenciamento de Threads*: não há a necessidade de bloquear uma *thread* e esperar por outra aplicação. Assim, evita-se o esgotamento de threads disponíveis.

Os autores listam também alguns desafios:

- *Programação Complexa*: o modelo assíncrono requer um modelo *event-driven* de programação, levando a diversos *event handlers* para responder as mensagens que chegam. Desenvolver e debugar pode ser mais complexo, já que não há uma sequência de métodos invocados.
- *Problemas com sequências*: as mensagens podem ser entregues fora de ordem. Se a ordem for importante, é preciso restabelecer a sequência programaticamente.
- *Cenários síncronos*: nem todas as aplicações podem operar em um modelo Send-and-Forget.

- *Performance*: pode ser adicionado algum *overhead* na comunicação, principalmente para grandes volumes de dados.
- *Suporte limitado*: algumas tecnologias não tem suporte a alguns Message Brokers.
- *Vendor lock-in*: muitos Message Brokers implementam protocolos proprietários e usualmente não se conectam uns aos outros. Inclusive, a terminologia é diferente entre muitas das soluções de Mensageria disponíveis.

Tipos de Channel

Os autores do livro [Enterprise Integration Patterns](#) (HOHPE; WOOLF, 2003), classificam os Message Channels em dois tipos:

- Point-to-Point Channel, análogo a uma conversa um-a-um no WhatsApp
- Publisher-Subscriber Channel, análogo a um grupo do WhatsApp

Point-to-Point Channel

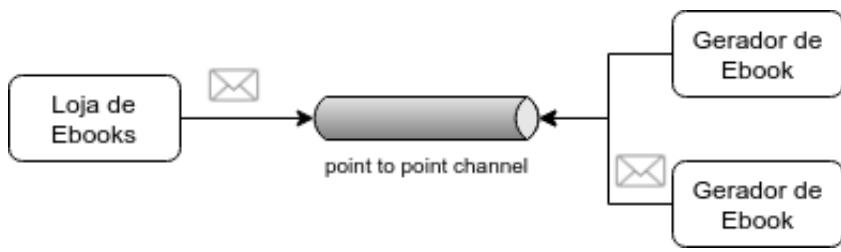
Em um sistema de trading de ações, uma negociação deve ser feita apenas uma vez. Em uma loja de ebooks, um livro comprado por um usuário deve ser gerado apenas uma vez. Como o Producer pode garantir que apenas um Consumer recebe uma determinada mensagem?

Os Consumers poderiam coordenar entre si para saber quem recebeu qual mensagem, garantindo a entrega única. Porém, essa solução seria complexa, aumentaria o tráfego de rede e aumentaria o acoplamento entre Consumers antes independentes.

Uma implementação melhor seria o próprio Message Broker determinar qual Consumer deve obter qual mensagem, garantindo que somente um receba cada mensagem. Hohpe e Woolf chamam esse tipo de comunicação de **Point-to-Point Channel**.

Quando um Point-to-Point Channel tem apenas um Consumer, não é

nada surpreendente que uma mensagem é consumida apenas uma vez. Mas quando há muitos Consumers para um mesmo Channel, os autores os chamam de **Competing Consumers**.



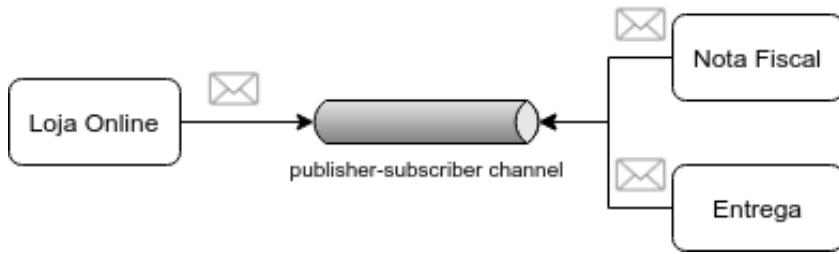
Competing Consumers, em que apenas um dos Receivers disponíveis recebe cada mensagem, permite a Escalabilidade Horizontal. Poderíamos lidar com um aumento na carga do sistema alocando mais Receivers para um mesmo Point-to-Point Channel. O Message Broker ficaria responsável pelo Load Balancing entre as instâncias redundantes.

Publisher-Subscriber Channel

Quando uma compra é finalizada em uma Loja Online, várias coisas precisam ser feitas: a nota fiscal precisa ser gerada, a entrega precisa ser despachada, o estoque precisa ser atualizado. Se a compra finalizada for uma mensagem, como um Producer pode transmitir uma mensagem para todos os Consumers interessados?

O Producer publica uma mensagem em um Message Channel e o Message Broker fica responsável por notificar todos os Consumers inscritos no Channel. Trata-se de um **Publisher-Subscriber Channel**, em que há um publicador (em inglês, *Publisher*) e vários inscritos (em inglês, *Subscribers*).

Um Subscriber deve ser notificado de uma mensagem apenas uma vez. Uma mensagem deve ser considerada consumida somente quando todos os Subscribers foram notificados. Subscribers não devem competir entre si, mas receber todas as mensagens de um Publisher-Subscriber Channel.



Um Publish-Subscribe Channel é, em geral, implementado da seguinte maneira: há um Channel de entrada que é dividido em múltiplos Channels de saída, um para cada Subscriber. Cada mensagem tem uma cópia entregue a cada um dos Channels de saída.

É possível usar um Publisher-Subscriber Channel para monitoramento, bastando plugar um sistema de Monitoramento como um Subscriber.

Digamos que temos um Publish-Subscribe Channel com um sistema de Monitoramento e um sistema de Notas Fiscais como Subscribers. O que acontece quando um Subscriber está fora do ar? Certamente, o outro Subscribers continua a receber as mensagens. Mas e quando o Subscriber volta a funcionar normalmente? Caso seja o sistema de Monitoramento, as mensagens não recebidas podem ser descartadas. Já no caso do sistema de Notas Fiscais, seria interessante que o Message Broker tenha armazenado todas as mensagens não entregues enquanto estava fora do ar. O sistema de Notas Fiscais é o que Hohpe e Woolf chamam de **Durable Subscriber**: um Subscriber que tem as mensagens publicadas salvas enquanto estiver desconectado.

Tipos de Mensagens

No livro [Enterprise Integration Patterns](#) (HOHPE; WOOLF, 2003), Gregor Hohpe e Bobby Woolf descrevem uma Mensagem como dados que são transmitidos em um Message Channel e que consistem de um *header*, que contém metadados usados pelo Message Broker, e um *body*, que contém os dados em si.

Para um Message Broker, todas as mensagens são semelhantes: alguns dados no *body* a serem transmitidos de acordo com o configurado no *header*. Uma Mensagem pode ser binária, um texto CSV, XML ou JSON

ou um objeto Java serializado, por exemplo.

Já para uma aplicação, os autores identificam alguns tipos de mensagens:

- **Document Message**: é usada para transmitir dados entre aplicações. O Consumer decide o que fazer com os dados recebidos.
- **Command Message**: serve como a invocação de um método em outra aplicação. É apenas a requisição, sem uma resposta. Em geral, é usada em um Point-to-Point Channel, e é consumida apenas uma vez por apenas um Consumer.
- **Event Message**: é uma notificação aos Consumers de que algo aconteceu. Em geral, é usada em um Publisher-Subscriber Channel.

Domain Events

No livro [Domain-Driven Design Distilled](#) (VERNON, 2016), Vaughn Vernon diz que um **Domain Event** é uma ocorrência significativa em termos de negócio em um determinado Bounded Context.

Vernon ressalta que o nome de um evento é importante, ligando com o conceito de Ubiquitous Language, em que a linguagem de negócio deve estar representada no código. Um bom nome de um Domain Event deve ser uma referência a algo de negócio que já aconteceu. Por exemplo, em um contexto de gerenciamento ágil teríamos os Domain Events `ProdutoCriado` e `ReleaseAgendada`. Perceba que um Domain Event tem um Aggregate associado, como `Produto` e `Release`.

Mas qual o estímulo que a aplicação recebe para ocorrer um Domain Event como `ProdutoCriado`? Vernon descreve algo como `CriarProduto`, uma ação `criar` feita por um usuário ou outro Bounded Context no Aggregate `Produto`, teria como resultado um `ProdutoCriado`. Esse tipo de ação é chamado de Command por Vernon e pela comunidade DDD.

Observação: o Command do DDD é um conceito de modelagem de

domínio, não necessariamente relacionado a integração síncrona ou assíncrona. Portanto, um Command do DDD não necessariamente será um Command Message.

Nos termos do DDD, um Domain Event precisa ser publicado a todos os Bounded Contexts interessados.

Uma Arquitetura de Microservices que usa Domain Events para integrar diferentes serviços, sendo publicados e consumidos em/de Message Brokers, é chamada de Event-Driven Microservices.

Domain Event

Um Aggregate publica um Domain Event quando é criado ou sofre outra alteração significativa.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a)

Event Storming

No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson cita um workshop usado pela comunidade DDD para explorar e entender um domínio complexo: o **Event Storming**. Para realizar esse workshop, especialistas de domínio devem ser reunidos em uma sala com post-its e um quadro branco.

Richardson cita três passos principais:

1. Peça para os especialistas de domínio fazerem um Brainstorm dos eventos que acontecem no domínio.
2. Identifique, junto aos especialistas de domínio, se a origem dos eventos são ações do usuário, um sistema externo, outro Domain Event ou alguma data ou horário.
3. Colabore com os especialista de domínio para identificar Aggregates, que consomem cada Command e emitem o Domain Event correspondente.

Cada item identificado tem uma cor específica de post-it.

O resultado é um Domain Model centrado em Aggregates e Domain Events.

Domain Events no Caelum Eats

Relembrando: uma nota fiscal deve ser gerada assim que um Pagamento for confirmado.

Isso pode ser modelado como o Domain Event `PagamentoConfirmado`.

O Producer desse evento seria o serviço de Pagamentos, que recebe uma ação do usuário para confirmar um pagamento.

O Consumer seria o serviço de Nota Fiscal, que receberia o evento e geraria a nota fiscal.

O que deveria estar contido no corpo dessa Event Message? Os dados do Pagamento que acabou de ser confirmado e os dados do Pedido relacionado. Como tratam-se de Aggregates do serviço de Pagamentos e do módulo de Pedido do Monólito, respectivamente, poderíamos passar apenas os ids do Pagamento e Pedido na mensagem para o serviço de Nota Fiscal.



Revisando as integrações existentes

Quais as integrações que já implementadas poderiam ser modeladas como Domain Events?

O serviço de Pagamentos invoca o módulo de Pedido do Monólito, por meio de HTTP, para avisar que um pagamento foi confirmado.

O módulo de Restaurante do Monólito, também por meio de HTTP, avisa

que um novo restaurante foi aprovado e que os dados de um restaurante foram atualizados para o serviço de Distância.

Todas essas integrações poderiam ser modeladas como Domain Events:

- o mesmo Domain Event `PagamentoConfirmado` publicado pelo serviço de Pagamentos poderia ser consumido pelo módulo Pedido do Monólito para atualizar o status do Pedido.
- os Domain Events `NovoRestauranteAprovado` e `RestauranteAtualizado` poderiam ser publicados pelo módulo de Restaurante do Monólito e consumidos pelo serviço de Distância.

Protocolos de Mensageria e AMQP

Existem diferentes protocolos de Mensageria. Entre eles:

- *STOMP* (Simple Text-Oriented Messaging Protocol): um protocolo baseado em texto, com alguma similaridade com o HTTP. Possui clientes em diversas linguagens, incluindo JavaScript com WebSocket, que é executado diretamente em navegadores. Diversos Message Brokers implementam esse protocolo, incluindo ActiveMQ (nas versões clássica e Artemis) e RabbitMQ.
- *MQTT* (Message Queue Telemetry Transport): focado em IoT, é um protocolo binário muito eficiente e compacto que foi projetado para redes de alta latência e pouca banda como links de satélite e conexões discadas. Só é compatível com Publisher-Subscriber Channels. ActiveMQ (nas duas versões) e RabbitMQ também dão suporte a esse protocolo. É uma especificação mantida pela ISO e OASIS.
- *AMQP* (Advanced Message Queuing Protocol): um protocolo bastante abrangente. Há suporte a Point-to-Point e Publisher-Subscriber Channels, além de outros modelos de Mensageria. Na versão 1.0, significativamente diferente da 0-9-1, foi padronizado pela ISO e OASIS. Focado em Confiabilidade e Segurança, é implementado por diversos Message Brokers, como o RabbitMQ

(AMQP 0-9-1) e ambas as versões do ActiveMQ (AMQP 1.0).

O JMS é um protocolo ou uma API?

O JMS (Java Message Service), especificado no Java EE, não é um protocolo, mas uma API. Dessa forma, define uma série de abstrações, interfaces e classes utilitárias a serem implementadas em Java pelos fornecedores de Message Brokers. Cada fornecedor pode usar um protocolo diferentes e, por isso, disponibilizam JARs que implementam os detalhes de comunicação.

A terminologia do JMS é própria:

- o conceito de Channel, mais amplo, é chamado de *Destination*.
- um Point-to-Point Channel é chamado de *Queue*. O Producer é chamado de *Sender* e o Consumer, de *Receiver*.
- um Publisher-Subscriber Channel é chamado de *Topic*. O Producer é chamado de *Publisher* e o Consumer, de *Subscriber*.

RabbitMQ e AMQP

RabbitMQ implementa, além dos protocolos STOMP e MQTT, a versão 0-9-1 do protocolo AMQP. A implementação foi feita em Erlang pela empresa Rabbit Technologies, posteriormente adquirida pela SpringSource/Pivotal.

A versão 0-9-1 do AMQP traz algumas terminologias e conceitos diferentes dos patterns estudados anteriormente.

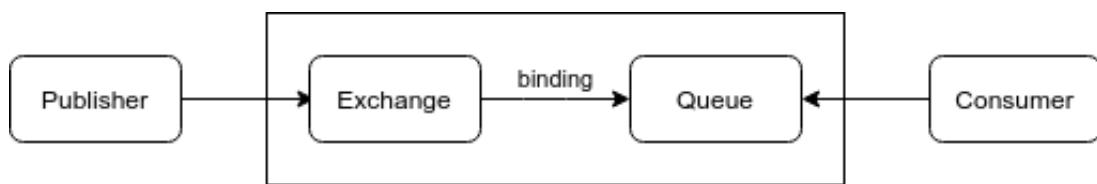
Por exemplo, no AMQP 0-9-1, channels são várias conexões leves que compartilham uma mesma conexão TCP.

Quem produz as mensagens é chamado de **Publisher** (e não Producer), independente do destino ser um Point-to-Point Channel ou um Publisher-Subscriber Channel.

As mensagens não são publicadas diretamente em um Point-to-Point ou

em um Publisher-Subscriber Channel, mas em uma abstração chamada **Exchange**. Dessa forma, um Publisher no AMQP não sabe o tipo de Channel que será utilizado.

A configuração do tipo de Message Channel passa a ser responsabilidade do Message Broker, o que é chamado de *roteamento*. O roteamento é feito, a partir de um Exchange, para um ou mais Point-to-Point Channels chamados de **Queues**, de acordo com regras chamadas **Bindings**. Os **Consumers** recebem mensagens dessas Queues. Um Binding pode ter, opcionalmente, uma *routing key* que irá influenciar qual Queue recebe uma dada mensagem.



O AMQP 0-9-1 é um protocolo "programável" em que a própria aplicação pode criar Exchanges, Bindings e Queues, não apenas o administrador do Message Broker.

Tanto um Exchange como uma Queue podem ter diversos atributos. Entre os mais importantes:

- Nome
- Durabilidade (em inglês, *Durability*): se for durável, o Exchange não é apagado após uma reinicialização do Message Broker; se for transiente, é preciso redeclará-lo toda vez que o Broker é reiniciado.
- *Auto-delete*: o Exchange é removido quando a última Queue de um consumidor é desconectado

Um Exchange também pode ter diferentes tipos:

- *Direct Exchange*: faz a entrega de mensagens com base em uma routing key da mensagem, roteando para uma Queue com a mesma routing key. É uma implementação do pattern Selective Consumer. Caso haja múltiplas instâncias da mesma aplicação como Consumers da Queue, é feito um Load Balancing no estilo Round-

Robin, implementando o pattern Competing Consumers.

- *Default Exchange*: um Exchange sem nome, é um caso especial de Direct Exchange, em que é feito o Binding automaticamente com todas as Queues com uma routing key com o mesmo nome da Queue. Uma mensagem que contém como routing key o nome de uma Queue é roteada diretamente para a Queue. Parece um Point-to-Point Channel em que uma mensagem é enviada diretamente a uma Queue, mas tecnicamente não é o que acontece.
- *Fanout Exchange*: as routing keys são ignoradas. Quando uma mensagem é enviada a um determinado Exchange desse tipo, todas as Queues com Bindings receberão uma cópia da mensagem. Implementa um Publisher-Subscriber Channel.
- *Topic Exchange*: as mensagens são roteadas a todas as Queues que atendem a algum critério de filtragem. É uma implementação de um Publisher-Subscriber Channel com Selective Consumers.
- *Headers Exchange*: ignoram routing keys, usando um ou mais atributos do cabeçalho das mensagens para o roteamento.

O RabbitMQ já vem com alguns Exchanges de cada tipo como `amq.direct`, (`AMQP default`), `amq.fanout`, `amq.topic` e `amq.headers`.

Mais informações na documentação do RabbitMQ:

<https://www.rabbitmq.com/tutorials/amqp-concepts.html>

Exercício: configurando o RabbitMQ no Docker

1. Adicione ao `docker-compose.yml` a configuração de um RabbitMQ na versão 3. Mantenha as portas padrão 5672 para o MOM propriamente dito e 15672 para a UI Web de gerenciamento. Defina o usuário `eats` com a senha `caelum123`:

```
rabbitmq:
```

```
  image: "rabbitmq:3-management"
  restart: on-failure
  ports:
    - "5672:5672"
```

– "15672:15672"

environment:

RABBITMQ_DEFAULT_USER: eats

RABBITMQ_DEFAULT_PASS: caelum123

O docker-compose.yml completo, com a configuração do RabbitMQ, pode ser encontrado em: <https://gitlab.com/snippets/1888246>

2. Execute novamente o seguinte comando:

Deve aparecer algo como:

```
eats-microservices_mysql.pagamento_1 is up-to-date
eats-microservices_mongo.distancia_1 is up-to-date
Creating eats-microservices_rabbitmq_1 ... done
```

3. Para verificar se está tudo OK, acesse a pelo navegador a UI de gerenciamento do RabbitMQ:

<http://localhost:15672/>

O username deve ser eats e a senha caelum123.

Spring Cloud Stream

Parte do Spring Cloud, o Spring Cloud Stream é um framework que facilita a construção de Event-Driven Microservices.

O Spring Cloud Stream provê uma série de abstrações e há uma série de *binders* para diferentes sistemas de Mensageria. Entre eles:

- RabbitMQ
- Apache Kafka
- Kafka Streams
- Amazon Kinesis
- Google PubSub

A terminologia de Stream Processing

O Spring Cloud Stream parte do Spring Messaging e Spring AMQP, mas provê abstrações diferentes, mais relacionadas com o processamento de fluxo de dados (em inglês, *Stream Processing*).

A terminologia de Stream Processing, usada por projetos como [Kafka Streams](#), [Apache Flink](#) e [Akka Streams](#), é baseada nos seguintes termos:

- *Stream*: um fluxo de dados contínuo e sem um fim claro como, por exemplo, os dados de localização de um celular ou os logs de um sistema.
- *Source*: uma fonte de dados, que produz um fluxo de dados. É equivalente a um Producer.
- *Sink*: um escoadouro, que consome um fluxo de dados. É equivalente a um Consumer.
- *Processor*: um transformador do fluxo de dados.

Publicando um evento de pagamento confirmado com Spring Cloud Stream

Adicione, no `pom.xml` do serviço de pagamento, o starter do projeto Spring Cloud Stream Rabbit:

```
##### fj33-eats-pagamento-service/pom.xml
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

Adicione o usuário e senha do RabbitMQ no `application.properties` do serviço de pagamento:

```
##### fj33-eats-pagamento-
```

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

```
spring.rabbitmq.username=eats  
spring.rabbitmq.password=caelum123
```

Crie uma classe `AmqpPagamentoConfig` no pacote `br.com.caelum.eats.pagamento` do serviço de pagamento, anotando-a com `@Configuration`.

Dentro dessa classe, crie uma interface `PagamentoSource`, que define um método `pagamentosConfirmados`, que tem o nome do `exchange` no RabbitMQ. Esse método deve retornar um `MessageChannel` e tem a anotação `@Output`, indicando que o utilizaremos para enviar mensagens ao MOM.

A classe `AmqpPagamentoConfig` também deve ser anotada com `@EnableBinding`, passando como parâmetro a interface `PagamentoSource`:

```
##### fj33-eats-pagamento-  
service/src/main/java(br/com/caelum/eats/pagamento/AmqpPagamentoC  
onfig.java
```

```
@EnableBinding(PagamentoSource.class)  
@Configuration  
class AmqpPagamentoConfig {  
  
    static interface PagamentoSource {  
  
        @Output  
        MessageChannel pagamentosConfirmados();  
    }  
  
}
```

Os imports são os seguintes:

```
import org.springframework.cloud.stream.annotation.EnableBindi
import org.springframework.cloud.stream.annotation.Output;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.MessageChannel;

import br.com.caelum.eats.pagamento.AmqpPagamentoConfig.Pagame
```

Crie uma classe `PagamentoConfirmado`, que representará o payload da mensagem, no pacote `br.com.caelum.eats.pagamento` do serviço de pagamento. Essa classe deverá conter o id do pagamento e o id do pedido:

```
##### fj33-eats-pagamento-
service/src/main/java(br/com/caelum/eats/pagamento/PagamentoConfir
mado.java
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
class PagamentoConfirmado {

    private Long pagamentoId;
    private Long pedidoId;

}
```

Os imports são do Lombok:

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
```

No mesmo pacote de eats-pagamento-service, crie uma classe NotificadorPagamentoConfirmado, anotando-a com @Service.

Injetar PagamentoSource na classe e adicione um método notificaPagamentoConfirmado, que recebe um Pagamento. Nesse método, crie um PagamentoConfirmado e use o MessageChannel de PagamentoSource para enviá-lo para o MOM:

```
##### fj33-eats-pagamento-
service/src/main/java/br/com/caelum/eats/pagamento/NotificadorPagam
entoConfirmado.java
```

```
@Service
@AllArgsConstructor
class NotificadorPagamentoConfirmado {

    private PagamentoSource source;

    void notificaPagamentoConfirmado(Pagamento pagamento) {
        Long pagamentoId = pagamento.getId();
        Long pedidoId = pagamento.getPedidoId();
        PagamentoConfirmado confirmado = new PagamentoConfirmado(pagamento);
        source.pagamentosConfirmados().send(MessageBuilder.withPay
    }

}
```

Faça os imports a seguir:

```
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.stereotype.Service;

import br.com.caelum.eats.pagamento.AmqpPagamentoConfig.Pagame
import lombok.AllArgsConstructor;
```

Em `PagamentoController`, adicione um atributo `NotificadorPagamentoConfirmado` e, no método `confirma`, invoque o método `notificaPagamentoConfirmado`, passando o pagamento que acabou de ser confirmado:

```
##### fj33-eats-pagamento-
service/src/main/java/br/com/caelum/eats/pagamento/PagamentoController.java
```

```
// anotações ...
class PagamentoController {

    // outros atributos ...
    private NotificadorPagamentoConfirmado pagamentoConfirmado;

    // código omitido ...

    @PutMapping("/{id}")
    Resource<PagamentoDto> confirma(@PathVariable Long id) {

        Pagamento pagamento = pagamentoRepo.findById(id).orElseThrow();
        pagamento.setStatus(Pagamento.Status.CONFIRMADO);
        pagamentoRepo.save(pagamento);

        pagamentoConfirmado.notificaPagamentoConfirmado(pagamento)

        Long pedidoId = pagamento.getPedidoId();
        pedidoClient.avisaQueFoiPago(pedidoId);

        return new PagamentoDto(pagamento);
    }

    // código omitido ...
}
```

Recebendo eventos de pagamentos confirmados

com Spring Cloud Stream

Adicione ao pom.xml do eats-nota-fiscal-service uma dependência ao starter do projeto Spring Cloud Stream Rabbit:

```
##### fj33-eats-nota-fiscal-service/pom.xml
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

No application.properties do serviço de nota fiscal, defina o usuário e senha do RabbitMQ :

```
##### fj33-eats-nota-fiscal-
service/src/main/resources/application.properties
```

```
spring.rabbitmq.username=eats
spring.rabbitmq.password=caelum123
```

No pacote br.com.caelum.eats.notafiscal do serviço de nota fiscal, crie uma classe AmqpNotaFiscalConfig , anotando-a com @Configuration.

Defina a interface Pagamentosink, que será para configuração do consumo de mensagens do MOM. Dentro dessa interface, defina o método pagamentosConfirmados, com a anotação @Input e com SubscribableChannel como tipo de retorno.

O nome do exchange no , que é o mesmo do source do serviço de pagamentos, deve ser definido na constante PAGAMENTOS_CONFIRMADOS.

Não deixe de anotar a classe AmqpNotaFiscalConfig com @EnableBinding, tendo como parâmetro a interface PagamentoSink:

```
##### fj33-eats-nota-fiscal-
service/src/main/java/br/com/caelum/eats/notafiscal/AmqpNotaFiscalCon
fig.java
```

```
@EnableBinding(PagamentoSink.class)
@Configuration
class AmqpNotaFiscalConfig {

    static interface PagamentoSink {
        String PAGAMENTOS_CONFIRMADOS = "pagamentosConfirmados";

        @Input
        SubscribableChannel pagamentosConfirmados();
    }

}
```

Adicione os imports corretos:

```
import org.springframework.cloud.stream.annotation.EnableBindi
import org.springframework.cloud.stream.annotation.Input;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.SubscribableChannel;

import br.com.caelum.notafiscal.AmqpNotaFiscalConfig.Pagamento
```

Use a anotação `@StreamListener` no método `processaPagamento` da classe `ProcessadorDePagamentos`, passando a constante `PAGAMENTOS_CONFIRMADOS` de `PagamentoSink`:

```
##### fj33-eats-nota-fiscal-
service/src/main/java/br/com/caelum/notafiscal/ProcessadorDePagament
os.java
```

```
// anotações ...
```

```
class ProcessadorDePagamentos {  
    // código omitido ...  
  
    @StreamListener(PagamentoSink.PAGAMENTOS_CONFIRMADOS) // adj  
    void processaPagamento(PagamentoConfirmado pagamento) {  
        // código omitido ...  
    }  
}
```

Faça os imports adequados:

```
import org.springframework.cloud.stream.annotation.StreamListe  
import br.com.caelum.notafiscal.AmqpNotaFiscalConfig.Pagamento
```

Exercício: Evento de Pagamento Confirmado com Spring Cloud Stream

1. Faça checkout da branch `cap11-evento-de-pagamento-confirmado-com-spring-cloud-stream` nos projetos do serviços de pagamentos e de nota fiscal:

```
cd ~/Desktop/fj33-eats-pagamento-service  
git checkout -f cap11-evento-de-pagamento-confirmado-com-sprir
```

```
cd ~/Desktop/fj33-eats-nota-fiscal-service  
git checkout -f cap11-evento-de-pagamento-confirmado-com-sprir
```

Reinicie o serviço de pagamento.

Inicie o serviço de nota fiscal executando a classe `EatsNotaFiscalServiceApplication`.

2. Certifique-se que o service registry, o serviço de pagamento, o serviço de nota fiscal e o monólito estejam sendo executados.

Confirme um pagamento já existente com o cURL:

```
curl -X PUT -i http://localhost:8081/pagamentos/1
```

Observação: para facilitar testes durante o curso, a API de pagamentos permite reconfirmação de pagamentos. Talvez não seja o ideal...

Acesse a UI de gerenciamento do RabbitMQ, pela URL
<http://localhost:15672>.

Veja nos gráficos que algumas mensagens foram publicadas. Veja pagamentosConfirmados listado em *Exchange*.

Observe, nos logs do serviço de nota fiscal, o XML da nota emitida. Algo parecido com:

```
<xml>
<loja>314276853</loja>
<nat_operacao>Almoços, Jantares, Refeições e Pizzas</nat_operacao>
<pedido>
  <items>
    <item>
      <descricao>Yakimeshi</descricao>
      <un>un</un>
      <codigo>004</codigo>
      <qtde>1</qtde>
      <vlr_unit>21.90</vlr_unit>
      <tipo>P</tipo>
      <class_fiscal>21069090</class_fiscal>
    </item>
    <item>
      <descricao>Coca-Cola Zero Lata 310 ML</descricao>
      <un>un</un>
      <codigo>004</codigo>
```

```
<qtde>2</qtde>
<vlr_unit>5.90</vlr_unit>
<tipo>P</tipo>
<class_fiscal>21069090</class_fiscal>
</item>
</items>
</pedido>
<cliente>
<nome>Isabela</nome>
<tipoPessoa>F</tipoPessoa>
<contribuinte>9</contribuinte>
<cpf_cnpj>169.127.587-54</cpf_cnpj>
<email>isa@gmail.com</email>
<endereco>Rua dos Bobos, n 0</endereco>
<complemento>--</numero>
<cep>10001-202</cep>
</cliente>
</xml>
```

Consumer Groups do Spring Cloud Stream

Adicione um nome de grupo para as instâncias do serviço de nota fiscal, definindo a propriedade

```
spring.cloud.stream.bindings.pagamentosConfirmados.group=NO  
application.properties:
```

```
##### fj33-eats-nota-fiscal-  
service/src/main/resources/application.properties
```

```
spring.cloud.stream.bindings.pagamentosConfirmados.group=notaF
```

Exercício: Competing Consumers e Durable Subscriber com Consumer Groups

1. Pare o serviço de nota fiscal e confirme alguns pagamentos pelo

cURL.

Note que, mesmo com o serviço consumidor parado, a mensagem é publicada no MOM.

Suba novamente o serviço de nota fiscal e perceba que as mensagens publicadas enquanto o serviço estava fora do ar **não** foram recebidas. Essa é a característica de um *non-durable subscriber*.

2. Execute uma segunda instância do serviço de nota fiscal na porta 9093.

No workspace dos microservices, acesse o menu *Run > Run Configurations...* do Eclipse e clique com o botão direito na configuração `EatsNotaFiscalServiceApplication` e depois clique em *Duplicate*.

Na configuração `EatsNotaFiscalServiceApplication (1)` que foi criada, acesse a aba *Arguments* e defina 9093 como a porta da segunda instância, em *VM Arguments*:

```
-Dserver.port=9093
```

Clique em *Run*. Nova instância do serviço de nota fiscal no ar!

3. Use o cURL para confirmar um pagamento. Algo como:

```
curl -X PUT -i http://localhost:9999/pagamentos/1
```

Note que o XML foi impresso nos logs das duas instâncias, `EatsNotaFiscalServiceApplication` e `EatsNotaFiscalServiceApplication (1)`. Ou seja, todas as instâncias recebem todas as mensagens publicadas no exchange `pagamentosConfirmados` do RabbitMQ.

4. Em um Terminal, vá até a branch `cap11-consumer-groups` do serviço de nota fiscal:

```
cd ~/Desktop/fj33-eats-nota-fiscal-service  
git checkout -f cap11-consumer-groups
```

Reinic peace ambas as instâncias do serviço de nota fiscal.

5. Novamente, confirme alguns pagamentos por meio do cURL.

Note que o XML é impresso alternadamente nos logs das instâncias `EatsNotaFiscalServiceApplication` e `EatsNotaFiscalServiceApplication (1)`.

Apenas uma instância do grupo recebe a mensagem, um pattern conhecido como *Competing Consumers*.

6. Pare ambas as instâncias do serviço de nota fiscal. Confirme novos pagamentos usando o cURL.

Perceba que não ocorre nenhum erro.

Acesse a UI de gerenciamento do RabbitMQ, na página que lista as *queues* (filas):

<http://localhost:15672/#/queues>

Perceba que há uma queue para o consumer group chamada `pagamentosConfirmados.notafiscal`, com uma mensagem em *Ready* para cada confirmação efetuada. Isso indica mensagem de pagamento confirmado foi armazenada na queue.

Suba uma (ou ambas) as instâncias do `eats-nota-fiscal-service`. Perceba que os XMLs das notas fiscais foram impressos no log.

Armazenar mensagens publicadas enquanto um subscriber está fora do

ar, entregando-as quando sobem novamente, é um pattern conhecido como *Durable Subscriber*.

Como vimos, os *Consumer Groups* do Spring Cloud Stream / RabbitMQ implementam os patterns *Competing Consumers* e *Durable Subscriber*.

Notificando o Front-End com um WebSocket

Atualmente no Caelum Eats, quando há uma mudança no status de um Pedido, o usuário tem que recarregar a página para ver a atualização. Por exemplo, quando um Pedido está PAGO e o dono do restaurante confirma o Pedido, o usuário só verá CONFIRMADO no status quando recarregar a página.

Na tela de Pedidos Pendentes, um dono de restaurante precisa fazer algo semelhante: de tempos em tempos, precisa lembrar de recarregar a página para verificar os novos pedidos que vão chegando.

Os navegadores já tem há algum tempo a API de **WebSocket**, que abre uma conexão entre o navegador e um servidor Web, permitindo uma comunicação full-duplex. Assim, tanto o navegador como o servidor podem iniciar uma nova mensagem. É feita uma transição de protocolos: a conexão a um WebSocket começa com uma requisição HTTP contendo alguns cabeçalhos específicos como `Upgrade` e `Sec-WebSocket-Key` e uma resposta com o status `101 Switching Protocols` com cabeçalhos como `Sec-WebSocket-Accept`. A partir dessas informações, é criada uma conexão não-HTTP entre o navegador e o servidor.

Diversas aplicações usam WebSockets para prover uma boa experiência aos usuários como: chats, páginas de notícias, placares de futebol, home brokers da Bolsa de Valores.

Poderíamos utilizar um WebSocket na página de status de Pedido e na de Pedidos Pendentes.

Mas há um problema: o Zuul é um proxy HTTP e não funciona com

outros protocolos. O que fazer?

Poderíamos colocar a implementação de WebSockets no próprio API Gateway, recebendo notificações de alteração no status do Pedido do Monólito por meio de alguma Queue do RabbitMQ e repassando para o Front-End com um WebSocket.



Configurações de WebSocket para o API Gateway

Adicione a dependência ao starter de WebSocket do Spring Boot no pom.xml do API Gateway:

```
##### fj33-api-gateway/pom.xml
```

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```

Defina a classe `WebSocketConfig` no pacote `br.com.caelum.apigateway` do API Gateway:

```
##### fj33-api-
gateway/src/main/java;br/com/caelum/apigateway/WebSocketConfig.java
```

```
@EnableWebSocketMessageBroker
@Configuration
class WebSocketConfig implements WebSocketMessageBrokerConfig{}
```

```
@Override  
public void configureMessageBroker(MessageBrokerRegistry registry)  
    registry.enableSimpleBroker("/pedidos", "/parceiros/resta  
}  
  
@Override  
public void registerStompEndpoints(StompEndpointRegistry registry)  
    registry.addEndpoint("/socket").setAllowedOrigins("*").wit  
}  
  
}
```

Não esqueça dos imports:

```
import org.springframework.context.annotation.Configuration;  
import org.springframework.messaging.simp.config.MessageBroker  
import org.springframework.web.socket.config.annotation.Enable  
import org.springframework.web.socket.config.annotation.StompE  
import org.springframework.web.socket.config.annotation.WebSoc
```

No `application.properties` do API Gateway, defina uma rota local do Zuul, usando forwarding, para as URLs que contém o prefixo `/socket`:

```
##### fj33-api-gateway/src/main/resources/application.properties
```

```
zuul.routes.websocket.path=/socket/**  
zuul.routes.websocket.url=forward:/socket
```

ATENÇÃO: essa rota deve vir antes da rota `zuul.routes.monolito`, que está definida como `/**`, um padrão que corresponde a qualquer URL.

Ainda não utilizaremos o WebSocket no API Gateway. Mas está tudo preparado!

Publicando evento de atualização de pedido no monólito

Adicione ao `pom.xml` do módulo `eats-pedido` do monólito, a dependência ao starter do Spring Cloud Stream Rabbit:

```
##### fj33-eats-monolito-modular/eats/eats-pedido/pom.xml
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

Configure usuário e senha do RabbitMQ no `application.properties` do módulo `eats-application` do monólito:

```
##### fj33-eats-monolito-modular/eats/eats-
application/src/main/resources/application.properties
```

```
spring.rabbitmq.username=eats
spring.rabbitmq.password=caelum123
```

Crie a classe `AmqpPedidoConfig` no pacote `br.com.caelum.eats` do módulo de pedidos do monólito, anotada com `@Configuration`.

ATENÇÃO: o pacote deve ser o mencionado anteriormente, para que não sejam necessárias configurações extras no Spring Boot.

Dentro dessa classe, defina uma interface `AtualizacaoPedidoSource` que define o método `pedidoComStatusAtualizado`, com o nome da exchange no RabbitMQ e que tem o tipo de retorno `MessageChannel` e é anotado com `@Output`.

Anote a classe `AmqpPedidoConfig` com `@EnableBinding`, passando a

interface criada.

```
##### fj33-eats-monolito-modular/eats/eats-
pedido/src/main/java;br/com/caelum/eats/AmqpPedidoConfig.java
```

```
@EnableBinding(AtualizacaoPedidoSource.class)
@Configuration
public class AmqpPedidoConfig {

    public static interface AtualizacaoPedidoSource {

        @Output
        MessageChannel pedidoComStatusAtualizado();
    }

}
```

Seguem os imports:

```
import org.springframework.cloud.stream.annotation.EnableBindi
import org.springframework.cloud.stream.annotation.Output;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.MessageChannel;

import br.com.caelum.eats.AmqpPedidoConfig.AtualizacaoPedid

```

Na classe `PedidoController` do módulo de pedido do monólito, adicione um atributo do tipo `AtualizacaoPedidoSource` e o utilize logo depois de atualizar o status do pedido no BD, nos método `atualizaStatus` e `pago`:

```
##### fj33-eats-monolito-modular/eats/eats-
pedido/src/main/java;br/com/caelum/eats/pedido/PedidoController.java
```

```
// anotações ...
class PedidoController {
```

```
private PedidoRepository repo;
private AtualizacaoPedidoSource atualizacaoPedido; // adicionado

// código omitido ...

@PutMapping("/pedidos/{id}/status")
public PedidoDto atualizaStatus(@RequestBody Pedido pedido)
    repo.atualizaStatus(pedido.getStatus(), pedido);

    return new PedidoDto(pedido);

// adicionado
PedidoDto dto = new PedidoDto(pedido);
atualizacaoPedido.pedidoComStatusAtualizado().send(MessageBuilder
    return dto;

}

// código omitido ...

@PutMapping("/pedidos/{id}/pago")
public void pago(@PathVariable("id") Long id) {
    // código omitido ...
    repo.atualizaStatus(Pedido.Status.PAGO, pedido);

    // adicionado
    PedidoDto dto = new PedidoDto(pedido);
    atualizacaoPedido.pedidoComStatusAtualizado().send(MessageBuilder
        return;

}

import org.springframework.messaging.support.MessageBuilder;
import br.com.caelum.eats.AmqpPedidoConfig.AtualizacaoPedidoSc
```

Recebendo o evento de atualização de status do

pedido no API Gateway

Adicione o starter do Spring Cloud Stream Rabbit como dependência no `pom.xml` do API Gateway:

```
##### fj33-api-gateway/pom.xml
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

No pacote `br.com.caelum.apigateway` do API Gateway, defina uma classe que `AmqpApiGatewayConfig`, anotada com `@Configuration` e `@EnableBinding`.

Dentro dessa classe, defina a interface `AtualizacaoPedidoSink` que deve conter o método `pedidoComStatusAtualizado`, anotado com `@Input` e retornando um `SubscribableChannel`. Essa interface deve conter também a constante `PEDIDO_COM_STATUS_ATUALIZADO`:

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/AmqpApiGatewayConf
ig.java
```

```
@EnableBinding(AtualizacaoPedidoSink.class)
@Configuration
class AmqpApiGatewayConfig {

    static interface AtualizacaoPedidoSink {
        String PEDIDO_COM_STATUS_ATUALIZADO = "pedidoComStatusAtua
        @Input
        SubscribableChannel pedidoComStatusAtualizado();
    }
}
```

```
}
```

No `application.properties` do API Gateway, configure o usuário e senha do RabbitMQ. Defina também um Consumer Group para o exchange `pedidoComStatusAtualizado`:

```
##### fj33-api-gateway/src/main/resources/application.properties

spring.rabbitmq.username=eats
spring.rabbitmq.password=caelum123

spring.cloud.stream.bindings.pedidoComStatusAtualizado.group=a
```

Dessa maneira, teremos um Durable Subscriber com uma queue para armazenar as mensagens, no caso do API Gateway estar fora do ar, e Competing Consumers, no caso de mais de uma instância.

Crie uma classe para receber as mensagens de atualização de status do pedido chamada `StatusDoPedidoService`, no pacote `br.com.caelum.apigateway.pedido` do API Gateway.

Anote-a com `@Service` e `@AllArgsConstructor`. Defina um atributo do tipo `simpMessagingTemplate`, cuja instância será injetada pelo Spring.

Crie um método `pedidoAtualizado`, que recebe um `Map<String, Object>` como parâmetro. Nesse método, use o `SimpMessagingTemplate` para enviar o novo status do pedido para o front-end. Se o pedido for pago, envie para uma *destination* específica para os pedidos pendentes do restaurante.

Anote o método `pedidoAtualizado` com `@StreamListener`, passando como parâmetro a constante `PEDIDO_COM_STATUS_ATUALIZADO` de `AtualizacaoPedidoSink`.

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/pedido/StatusDoPedidoService.java

@Service
@AllArgsConstructor
class StatusDoPedidoService {

    private SimpMessagingTemplate websocket;

    @StreamListener(AtualizacaoPedidoSink.PEDIDO_COM_STATUS_ATUALIZADO)
    void pedidoAtualizado(Map<String, Object> pedido) {

        websocket.convertAndSend("/pedidos/" + pedido.get("id") + "/status", pedido);

        if ("PAGO".equals(pedido.get("status"))) {
            Map<String, Object> restaurante = (Map<String, Object>) pedido.get("restaurante");
            websocket.convertAndSend("/parceiros/restaurantes/" + restaurante.get("id"), restaurante);
        }
    }
}
```

Certifique-se que fez os imports adequados:

```
import java.util.Map;

import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.messaging.simp.SimpMessagingTemplate;
import org.springframework.stereotype.Service;

import br.com.caelum.apigateway.AmqpApiGatewayConfig.AtualizacaoPedidoService;
import lombok.AllArgsConstructor;
```

Exercício: notificando novos pedidos e mudança

de status do pedido com WebSocket e Eventos

1. Em um Terminal, faça um checkout da branch `cap11-websocket-e-eventos` do monólito, do API Gateway e da UI:

```
cd ~/Desktop/fj33-eats-monolito-modular  
git checkout -f cap11-websocket-e-eventos
```

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap11-websocket-e-eventos
```

```
cd ~/Desktop/fj33-eats-ui  
git checkout -f cap11-websocket-e-eventos
```

2. Rode o comando abaixo para baixar as bibliotecas SockJS e Stomp, que são usadas pela UI:

```
cd ~/Desktop/fj33-eats-ui  
npm install
```

3. Suba todos os serviços, o monólito e o front-end.

Abra duas janelas de um navegador, de maneira que possa vê-las simultaneamente.

Em uma das janelas, efetue login como dono de um restaurante (por exemplo, `longfu/ 123456`) e vá até a página de pedidos pendentes.

Na outra janela do navegador, efetue um pedido no mesmo restaurante, até confirmar o pagamento.

Perceba que o novo pedido aparece na tela de pedidos pendentes.

Mude o status do pedido para *Confirmado* ou *Pronto* e veja a alteração na tela de acompanhamento do pedido.

Para saber mais: Brokerless Messaging

No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson cita o [ZeroMQ](#), uma especificação/implementação em que os serviços trocam mensagens diretamente, sem um intermediário.

Entre as vantagens, citadas por Richardson estão: a menor latência e tráfego de rede, já que há menos conexões intermediárias; eliminação do Message Broker como gargalo de performance e ponto único de falha; menor complexidade operacional.

Entre as desvantagens: necessidade dos serviços saberem os endereços uns dos outros e, consequentemente, de mecanismos de Service Discovery; Disponibilidade reduzida, porque tanto o Producer como o Consumer precisam estar no ar ao mesmo tempo; entrega garantida e outras características de Message Brokers são difíceis de implementar.

Para saber mais: CQRS

Para boa parte das aplicações, CRUD é o suficiente. Apenas um BD pode ser mantido tanto para atualização como para leitura dos dados.

Mas há consultas que são complexas, como as realizadas para relatórios, estatísticas, séries temporais, gráficos, dashboards, busca textual, entre outros cenários.

No caso um Domínio que requer consultas complexas e múltiplas representações dos dados, a [comunidade DDD](#) tem o costume de ter Domain Models separados: uma para escrita e outro para leitura de informações. Essa técnica é chamada de **Command Query Responsibility Segregation (CQRS)**. O modelo de escrita é chamado de **Command Model** e o de leitura é chamado de **Query Model**.

Como descrito por Martin Fowler em seu artigo [CQRS](#) (FOWLER, 2011), os termos Command e Query do CQRS fazem referência ao trabalho de Bertrand Meyer, criador da linguagem Eiffel e pioneiro da

Orientação a Objetos. Meyer recomendava que os métodos de um objeto fossem separados em:

- Commands: métodos que mudam o estado do objeto mas não retornam valores.
- Queries: métodos que retornam valores mas não tem efeitos colaterais, não mudando o estado do objeto.

Meyer chamava esse princípio de *Command Query Separation*. Há uma forte influência dessa separação em diversas linguagens, incluindo os getters e setters do Java.

A separação em um Command Model e um Query Model do CQRS pode ser usada tanto em um Monólito como em uma Arquitetura de Microservices.

CQRS e Microservices

Ao decompor a aplicação em diferentes serviços, cada um com o seu próprio BD, surge a questão: como fazer consultas complexas que agregam dados de vários serviços? Ou ainda, como criar relatórios em uma Arquitetura de Microservices?

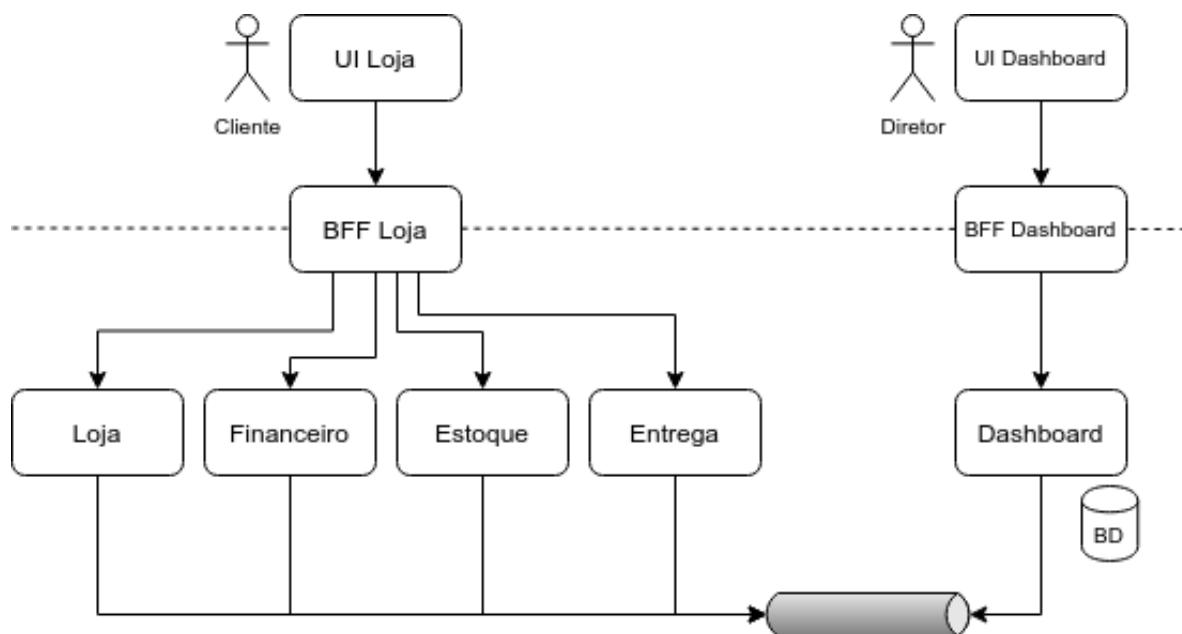
Umas das respostas que demos foi implementar uma API Composition, em que os vários serviços são chamados e os dados são agregados por um API Composer, algo similar a um *in-memory join*. Mas para datasets grandes, essa solução pode ser ineficiente.

No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson indica CQRS como uma solução para consultas complexas e relatórios, contrastando com API Composition.

Digamos que uma aplicação de e-commerce foi decomposta em diferentes serviços, como Loja, Estoque, Entrega e Financeiro. O diretor comercial quer um dashboard que mostre rapidamente uma visão geral do que está acontecendo na empresa no momento, mostrando estatísticas dos pedidos em cada uma das etapas. Como implementar

isso com CQRS?

Poderíamos modelar Domain Events para cada etapa de um pedido, publicando em um Message Channel que tem como Consumer um serviço específico para o Dashboard. Esse serviço de Dashboard receberia cada evento e agregaria os dados, persistindo em BD próprio. Quando o diretor comercial acessar a tela, os dados estarão lá, prontinhos!



Nesse caso, os Commands seriam as atualizações na Loja e nos serviços de Estoque, Entrega e Financeiro, que gerariam Domain Events. A Query seria o serviço de Dashboard, um Consumer dos Domain Events que transformaria os dados para uma representação própria.

No livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman discute a implementação de relatórios em uma Arquitetura de Microservices. Uma das soluções é um *Data Pump*, em que um software mantido pelos times dos serviços lê os dados de um serviço específico e os insere em um BD de relatórios compartilhado. Uma abordagem alternativa descrita por Newman é um *Event Data Pump*, que é bastante semelhante ao CQRS.

Com uma API Composition, a implementação da consulta busca dados dos diversos serviços, agregando os resultados. Já no caso do CQRS, os serviços enviam dados (Domain Events) para um intermediário (o

Message Broker) e a implementação da consulta obtém os dados desse intermediário. Trata-se de uma Inversão de Controle, só que no nível Arquitetural!

Pattern: Command Query Responsibility Segregation (CQRS)

Para implementar uma consulta que envolva dados de vários serviços, use eventos e mantenha uma View que replique os dados dos vários serviços.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a)

Prós e contras do CQRS

No artigo [CQRS](#) (FOWLER, 2011), Martin Fowler diz que CQRS é adequado em uma minoria de casos em que há um domínio complexo com múltiplas representações de dados. Há ainda um ganho de Performance e Escalabilidade nos casos em que há uma grande disparidade entre escritas e leituras. Para Fowler, a complexidade tecnológica pode diminuir a produtividade e aumentar o risco.

No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson identifica os seguintes benefícios quando CQRS é aplicado em uma Arquitetura de Microservices:

- Eficiência: para consultas em datasets grandes, ter os dados já agregados de maneira adequada com CQRS é muito mais eficiente que os in-memory joins feitos em uma API Composition. Além disso, usar CQRS evita as limitações de BDs específicos, permitindo aliar diversidade de consultas com eficiência.
- Separação de responsabilidades: separando os Domain Models de Command e Query, tanto o código da aplicação quanto o do BD provavelmente serão mais simples e fáceis de manter.

Entre as desvantagens de CQRS em Microservices citadas por Richardson, estão:

- Complexidade da Arquitetura: será necessário gerenciar e operar mais BDs, muitas vezes de diferentes paradigmas, e Message Brokers. Em termos de desenvolvimento, termos código para cada um dos BDs e para a publicação dos Domain Events.
- Eventual Consistency: chamada por Richardson de *replication lag*, algo como atraso de replicação, indica que o Query Model pode ainda não ter processado um evento já publicado no Command Model. Por isso, os dados consultados por uma UI, ou outro cliente, podem estar momentaneamente desatualizados.

Para saber mais: Event Sourcing

Muitas aplicações, por motivos regulatórios, precisam manter os detalhes de cada transformação nas entidades de negócio ocasionada por ações dos usuários. É o caso de aplicações bancárias e contábeis, que precisam de um log de auditoria bem abrangente.

Mesmo em outros tipos de aplicações, pode ser interessante manter o log das alterações nas entidades de negócio. Por exemplo, os profissionais de suporte poderiam usar essa informação para auxiliar os usuários. Já os desenvolvedores poderiam debugar e testar de maneira mais semelhante ao uso real de um sistema.

Como implementar esse log de auditoria?

Uma abordagem seria persistir os próprios Domain Events que acontecem em um Aggregate na sequência em que aconteceram.

Por exemplo, ao invés de persistir um `Pedido`, persistiríamos `PedidoRealizado`, `PedidoPago`, `PedidoConfirmado`, `PedidoPronto` e assim por diante, ordenados pela ordem em que ocorreram.

O estado atual de um `Pedido` seria reconstruído a partir dos Domain Events que aconteceram nesse Aggregate.

Isso permitiria reconstruir exatamente o estado de um Aggregate no

passado, consultar o histórico e ajustar eventos passados caso haja mudanças retroativas.

Esse tipo de solução é descrita por Martin Fowler no artigo [Event Sourcing](#) (FOWLER, 2005). Fowler menciona que a maioria dos programadores usa rotineiramente algo semelhante: um Sistema de Controle de Versões. Além disso, Fowler faz uma analogia com um livro contábil de um contador, em que todas as transações são mantidas mas não o estado final de uma conta.

Pattern: Event Sourcing

Persista um Aggregate como uma sequência de Domain Events que representam as mudanças de estado.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a)

Em seu livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson descreve o conceito de um *Event Store*: um híbrido entre um BD, que persiste os dados de um Aggregate, e um Message Broker, que permite que Consumers se inscrevam em Domain Events.

Uma maneira de implementar um *Event Store* é usar um BD Relacional com uma ou mais tabelas de eventos, com os Subscribers fazendo SELECTs diretamente nessas tabelas. Outra maneira é usar ferramentas com versões open-source como:

- [Event Store](#): implementado em .NET por Greg Young, pioneiro do DDD e Event Sourcing.
- [Axon Server](#): implementado em Java, disponibiliza também um framework integrado com o ecossistema Spring, facilitando a implementação.
- [Apache Kafka](#): se configurado para manter eventos por um longo prazo, pode ser usado como um Event Store.
- [Eventuate](#): implementado em Java por Chris Richardson, usa BD Relacionais ou Apache Kafka como Event Stores.

Um Event Store pode ser mais eficiente que um BD comum, já que os novos Domain Events são inseridos no fim de uma sequência, sem a necessidade de alterações em um registro já existente.

Event Sourcing pode influenciar positivamente no desacoplamento: um novo serviço pode ser adicionado e montar seu próprio modelo de dados a partir dos eventos já persistidos.

Dificuldades ao implementar Event Sourcing

Implementar Event Sourcing traz uma série de dificuldades.

Há a curva de aprendizado, já que é uma maneira de programar incomum para a maioria dos desenvolvedores.

Outra dificuldade é de infraestrutura, já que precisaremos de gerenciar e operar um Event Store e/ou um Message Broker.

Implementar consultas, mesmo das mais simples, pode ser ineficiente. Uma solução comum é usar CQRS, tendo um serviço que se inscreve no Event Store e, a partir da sequência de eventos, mantém o Query Model atualizado.

Como representar o estado atual de um Aggregate, como um Pedido? Em seu artigo [Event Sourcing](#) (FOWLER, 2005), Fowler cita três abordagens:

- sempre refazer o estado a partir dos eventos, o que seria bastante lento quando há muitos eventos;
- persistir, em paralelo, o estado final e os eventos, o que poderia levar a inconsistências;
- ou manter snapshots em um dado momento, minimizando o número de eventos necessários para refazer o estado final, algo semelhante ao que é feito pelos contadores nos balanços trimestrais.

A integração com sistemas externos pode impedir o replay dos eventos. Uma das estratégias descritas por Fowler é implementar um intermediário que não reenvia aos sistemas externos chamadas

originadas por eventos repetidos.

E quando os Domain Events são modificados? No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson classifica as mudanças nos eventos entre compatíveis e não compatíveis. Apenas adicionar novos Domain Events e novos campos em eventos já existentes são mudanças compatíveis. A maioria das mudanças são incompatíveis, como mudar nomes ou remover Aggregates, Domain Events ou campos.

Para saber mais: Saga

Consistência dos dados é algo importante em aplicações corporativas.

Em uma aplicação monolítica que tem apenas um BD, essa consistência não é tão complexa. Podemos usar transações, muitas vezes auxiliados por frameworks. No Spring, temos transações declarativas com a anotação `@Transactional`.

Mas em uma Arquitetura de Microservices (ou até em um Monólito) com múltiplos BDs e Message Brokers, a complexidade de manter a consistência dos dados aumenta drasticamente.

Como mencionamos anteriormente, Eric Brewer cunhou na publicação [Towards Robust Distributed Systems](#) (BREWER, 2000) o Teorema CAP, que indica que não é possível termos simultaneamente mais que duas das seguintes características: Consistência dos dados, Disponibilidade (*Availability*, em inglês) e tolerância a Partições de rede. Esse teorema foi ampliado por Daniel Abadi, no paper [Consistency Tradeoffs in Modern Distributed Database System Design](#) (ABADI, 2012), de maneira a incluir alta latência de rede como uma forma de indisponibilidade.

Two-Phase Commit e XA

Uma das maneiras de termos transações distribuídas é similar ao que Jim Gray e Andreas Reuter chamam, no livro [Transaction Processing: Concepts and Techniques](#) (GRAY; REUTER, 1992) de "protocolo de

"casamento". Pense nos personagens de um casamento de novela: os noivos, a plateia e o padre. O padre serve como coordenador da transação. Inicialmente, o padre pergunta aos noivos e a plateia se todos concordam que o casamento aconteça. Se todos concordarem, o casamento é confirmado (commit). Se alguma das partes discordarem, o casamento não é realizado (rollback). Há duas fases: uma fase de votação e uma fase de execução do commit, ambas gerenciadas por um coordenador. Esse tipo de protocolo é chamado de *Two-Phase Commit* (2PC).

Há uma especificação para 2PC mantida pelo The Open Group chamada *eXtended Architecture (XA)*. É mantida uma transação global e cada participante tem que ser compatível com XA. É o caso de boa parte dos BDs relacionais, de alguns Message Brokers mais antigos e do Java/Jakarta EE, com a especificação JTA. Por exemplo, a classe `MysqlXADatasource` implementa um `DataSource` compatível com XA, assim permitindo transações distribuídas.

Um problema é que muitas tecnologias mais modernas não dão suporte a XA, desde BDs como o MongoDB e Cassandra a Message Brokers como RabbitMQ e Kafka.

Essa incompatibilidade está relacionada com a escolha de Consistência em detrimento da Disponibilidade. Transações distribuídas com 2PC são essencialmente síncronas: todos os envolvidos, assim como o coordenador, precisam estar disponíveis ao mesmo tempo.

Sagas

No livro [Implementing Domain-Driven Design](#) (VERNON, 2013), Vaughn Vernon afirma que há domínios em que vários Domain Events devem acontecer para que um processo de longa duração seja finalizado. Esse tipo de processo composto por diferentes etapas é chamado de *Long-Running Process* ou **Saga**.

Vernon também indica que Sagas podem ser usadas para

processamento paralelo distribuído, sem a necessidade de transações distribuídas ou 2PC.

No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson foca em Sagas como uma maneira de implementar transações em uma Arquitetura de Microservices.

Pattern: Saga

Mantenha a consistência de dados entre serviços usando uma sequência de transações locais coordenados por Mensageria.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a)

Richardson descreve duas abordagens de estruturação de Sagas.

- Orquestração
- Coreografia

Sagas com Orquestração

Numa Orquestra Sinfônica, há um maestro responsável por coordenar as atividades dos músicos.

Uma Saga pode ter um coordenador central, o orquestrador, que envia Command Messages para cada serviço. Então, se necessário, os serviços enviam eventos de resposta.

Uma Saga pode ser modelada como um objeto, que mantém o estado geral da transação. Frameworks como o [Axon](#), escrito em Java, e [NServiceBus](#), escrito em C#, ajudam na implementação.

É o estilo recomendado por Chris Richardson, já que:

- evita dependências cíclicas entre serviços;
- minimiza os eventos para os quais os serviços precisam ser inscritos;

- separa a lógica de coordenação da implementação de cada etapa da Saga nos serviços.

Um grande risco é centralizar as regras de negócio, perdendo a autonomia de cada serviço. Richardson recomenda evitar regras de negócio nos orquestradores, mantendo apenas a responsabilidade de sequenciamento das etapas.

Coreografia

Num espetáculo de dança, os dançarinos seguem uma sequência de movimentos pré-estabelecida, sem a necessidade de um coordenador.

Uma Saga pode ser coreografada. Cada serviço recebe um evento com informações do progresso e de domínio, implementa uma etapa do processo agregando mais informações e publica um novo evento. O estado geral do processo é mantido apenas nos próprios eventos, sem a necessidade de um coordenador.

Chris Richardson argumenta que a principal vantagem de uma Saga coreografada é o baixo acoplamento, já que os serviços dependem somente de eventos, sem a necessidade de conhecerem uns aos outros.

Porém, Richardson lista algumas desvantagens:

- pode ser difícil de entender, já que não há um lugar único que define as etapas de uma saga. A lógica está distribuída no código de diversos serviços.
- podem acontecer dependências cíclicas: um serviço pode consumir um evento, publicando outro evento que é consumido por outro serviço que, por sua vez, publica um evento que será consumido pelo primeiro serviço!
- se um dado serviço consumir muitos eventos, pode levar a uma maior sincronização dos deploys entre serviços que deveriam ser autônomo.

No livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman diz

preferir uma abordagem coreografada. Segundo Newman, sistemas que seguem essa abordagem ter menor acoplamento, são mais flexíveis e com menor custo de mudanças. Porém, Newman reconhece que monitorar e rastrear os processos requer mais esforço.

Erros em uma Saga

O que acontece se houver uma falha em um dos participantes de uma Saga?

Não há uma transação global como no 2PC, então não é possível fazer um rollback global.

Tudo o que foi feito até a etapa do processo em que ocorreu a falha deve ser desfeito. São as transações de compensação (em inglês, *compensating transactions*). É algo relativamente comum no sistema bancário em que um pagamento, por exemplo, não pode ser cancelado mas pode ser feito um estorno.

Chris Richardson indica que as compensating transactions devem ser realizadas na ordem inversa das sequências de etapas que já foram aplicadas no processo.

Richardson classifica as etapas de um processo em três tipos:

- *compensatable transactions*: são seguidas por etapas que podem falhar e, por isso, precisam fornecer formas de desfazer o que já foi feito, as compensating transactions.
- *retryable transactions*: podem ser feitas novas tentativas até que a etapa seja bem sucedida. Por isso, nunca falham.
- *pivot transactions*: são seguidas por etapas que nunca falham. Por isso, se essa etapa não falhar, a Saga será finalizada com sucesso.

Discussão: Mensageria e Arquitetura

Ao falarmos de Load Balancing, mencionamos características operacionais transversais (ou *ilities*) como: Escalabilidade Horizontal,

Disponibilidade. Já quando estudamos Circuit Breakers e outros patterns associados, mencionamos Resiliência e Estabilidade.

Como Mensageria se relaciona a esses Atributos de Qualidade de uma Arquitetura?

Com um Point-to-Point Channel com Competing Consumers, em que apenas um Consumer de um grupo recebe cada mensagem, temos a característica da *Escalabilidade Horizontal*. Para aguentar uma demanda maior, basta ter mais Consumers competindo pelas mensagens. Por exemplo, uma loja de ebooks tem 2 instâncias para gerar PDFs, que suportam o tráfego usual. Porém, se o tráfego triplica na Black Friday, podemos colocar 6 ou mais instâncias para competir na geração dos PDFs.

E quanto a *Disponibilidade*? Pelo estilo assíncrono de comunicação, mesmo que os Consumers estejam fora do ar, o Producer pode continuar enviando mensagens.

Um Message Broker oferece uma boa alternativa em termos de *Resiliência* e *Estabilidade*, já que os Consumers podem ficar fora do ar momentaneamente, sem que o Producer seja afetado. Porém, ainda assim é possível sobrecarregar um Consumer. Ajustes nas configurações, números de instâncias e técnicas como *back-pressure* precisam ser levadas em conta.

A Comunicação Assíncrona afeta, de certa forma, a *Usabilidade*, já que algumas respostas a ações do usuário só estariam disponíveis posteriormente. Uma boa metáfora ao conversar com os usuários é um sistema de chamados, em que as solicitações só são respondidas depois de algum tempo.

A "Debugabilidade" e a *Observabilidade* podem ser dificultadas, mas parte da dificuldade é inerente aos Sistemas Distribuídos, usando RPC ou Mensageria. Há ferramentas que ajudam nessas tarefas, que estudaremos mais adiante.

Quanto a *Manutenibilidade*, dependerá da familiaridade dos desenvolvedores com conceitos de Mensageria. Em geral, o estilo assíncrono de programação requer conhecimentos mais avançados do time e os patterns de Mensageria não são tão conhecidos no mercado.

Testes Automatizados e Contratos

O perigo de testar tarde demais

Em tempos remotos, havia uma maneira tradicional de desenvolver, o [Waterfall](#), em que testar o software era uma atividade realizada depois de todo o código pronto e por um grupo diferente de pessoas: a equipe de Testes (em inglês, *Quality Assurance* ou QA). Testes seriam trabalho do time de QA. Programadores ajudarem nos testes seria um desperdício do precioso tempo dos programadores.

Essa separação da programação dos testes trazia uma série de desvantagens. Entre elas:

- uma cultura de conflito entre programadores e QA.
- ineficiência, já que boa parte dos cenários de teste eram roteiros seguidos à risca pelo time de QA, repetidamente.
- acúmulo de trabalho não finalizado, levando a uma sobrecarga do time de QA em fases tardias do projeto
- como QA está no final do processo, tende a ser descartado à medida que a pressão de entrega aumenta

Com o surgimento de outras maneiras de desenvolver software, que pregavam entrega constante de software de valor por times autônomos, as Metodologias Ágeis, houve uma mudança no processo. Para que um software pudesse ser potencialmente colocado em produção continuamente, a verificação proporcionada pelo time de QA deveria ser feita a todo momento.

À medida que mais funcionalidades e código vão se acumulando no decorrer de um projeto, seguir roteiros de teste passa a ser um esforço repetitivo muito árduo para o time de QA.

Testes Automatizados

Para que seja possível testar continuamente o software de maneira

eficiente, as tarefas repetitivas precisam ser minimizadas. Precisamos automatizar os testes!

Para isso, devemos criar um código que invoca o sistema, ou partes dele, com os parâmetros necessários e verifica se o resultado é o esperado.

Há diversos frameworks que ajudam nessa tarefa. Entre os principais da plataforma Java:

- [JUnit](#)
- [TestNG](#)
- [Spock](#), em que os testes são feitos na linguagem Groovy

Nunca no desenvolvimento de software tanto foi devido por tantos a tão poucas linhas de código.

Martin Fowler, citado na página principal de [versões antigas do JUnit](#)

Tipos de Testes Automatizados

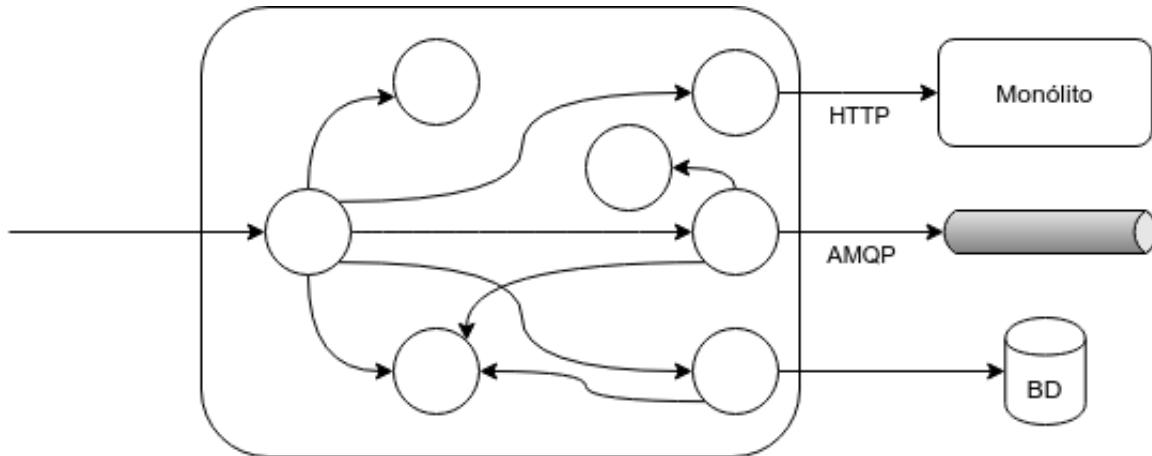
Digamos que somos do time do serviço de Pagamentos e desejamos implementar testes automatizados.

Primeiramente, precisamos ter uma ideia de como está organizado o código.

No serviço de Pagamentos, temos:

- `PagamentoController`, que recebe requests HTTP do API Gateway que, por sua vez, é chamado pela UI
- `PagamentoRepository`, que cuida da persistência de um pagamento no MySQL desse serviço
- `PedidoRestClient`, que invoca o módulo de Pedido do Monólito para avisar que um pedido foi pago
- `NotificadorPagamentoConfirmado`, que manda uma mensagem do Domain Event `PagamentoConfirmado` para um Message Channel que será consumido pelo serviço de Nota Fiscal

- diversas outras classes internas, que não tem nenhuma responsabilidade nas bordas desse sistema



Classes com diferentes responsabilidades podem ser testadas de diferentes maneiras.

Como diz Ham Vocke no artigo [The Practical Test Pyramid](#) (VOCKE, 2018), a terminologia usada na classificação de tipos de testes automatizados é muito confusa. Não há termos bem definidos: o que é teste de componentes para uns, é chamado de testes de integração ou de serviços para outros. E as nuances na escrita e arquitetura fazem com que os termos sejam mais um espectro que definições precisas. A dica de Vocke é ter um consenso nos termos usados pelo time.

Vamos usar a mesma terminologia de Chris Richardson no livro [Microservices Patterns](#) (RICHARDSON, 2018a):

- Unidade
- Integração
- Componentes
- End-to-End

Testes de Unidade

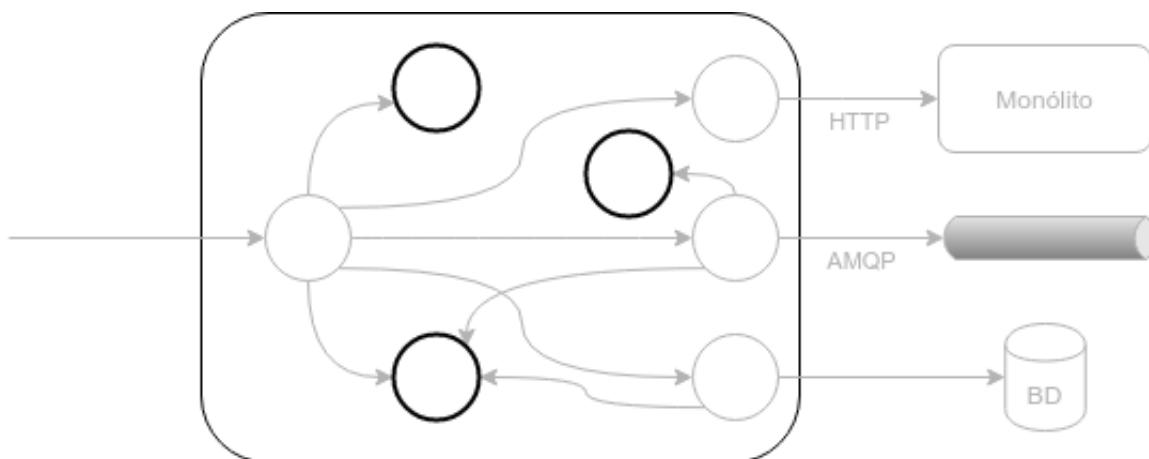
De acordo com Martin Fowler, em seu artigo [Unit Test](#) (FOWLER, 2014c), há diversas variações na definição do que é um Teste de Unidade, mas há alguns elementos em comum:

- são de baixo nível, exercitando uma pequena parte do sistema (ou unidade).
- são escritos pelos próprios programadores usando ferramentas corriqueiras e frameworks como o JUnit.
- são rápidos

Mas o que seria essa unidade a ser testada e cujos resultados são verificados?

Alguns, como Sam Newman no livro [Building Microservices](#) (NEWMAN, 2015), consideram que uma unidade é um único método ou função.

Tanto para Martin Fowler, ainda no artigo [Unit Test](#) (FOWLER, 2014c), como para Chris Richardson no livro [Microservices Patterns](#) (RICHARDSON, 2018a), em linguagens OO, uma unidade é uma classe.



É comum que uma Unidade seja testada de maneira isolada. Isso é trivial para uma regra de negócio que recebe alguns parâmetros, faz alguns cálculos e retorna algum valor.

Mas boa parte das Unidades interagem com outras classes. Nesse caso, há duas abordagens.

Em uma das abordagens, as dependências da Unidade que está sendo testada são trocadas por *Test Doubles* (algo como dublês de teste), que são objetos projetados para serem usados especificamente nos testes. Martin Fowler chama essa abordagem de Testes Solitários (em inglês, *Solitary Tests*), citando o termo criado por Jay Fields.

A outra abordagem traz um conceito diferente de Unidade. Martin Fowler diz que frequentemente considera um conjunto de classes relacionadas como uma Unidade. Os testes para esse tipo de Unidade seriam Testes Sociáveis (em inglês, *Sociable Tests*), em que os colaboradores de uma classe são invocados.

Toby Clemson, na palestra [Testing Strategies in a Microservice Architecture](#) (CLEMSON, 2014), argumenta que, para lógica de domínio complexa, faz mais sentido usar Sociable Tests, que invocam os cálculos e transições de estados dos próprios objetos de domínio.

Testes de Integração

Como testar Unidades na "borda" de uma aplicação, que invocam recursos externos como serviços remotos, BDs e Message Brokers?

Para exercitar essas partes do código, os recursos externos precisam ter resultados pré-estabelecidos. Para isso, são usados Test Doubles.

É o que Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a), chama de Testes de Integração (em inglês, **Integration Tests**).

Um Integration Test tende a ser bem mais lento que um Unit Test.

No artigo [Integration Test](#) (FOWLER, 2018), Martin Fowler indica que há duas noções de Integration Tests no mercado:

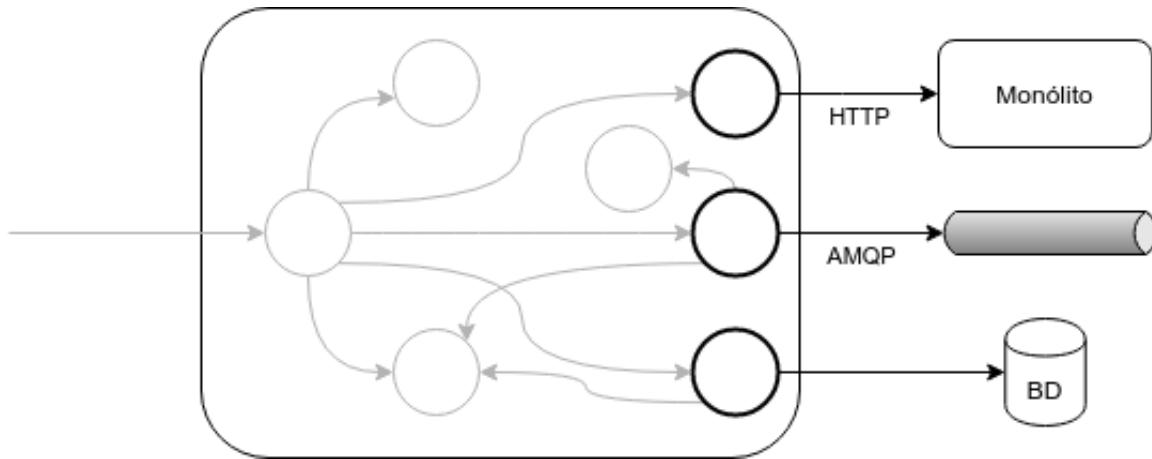
- Testes de Integração restritos (em inglês, *Narrow Integration Tests*), mais focados nos pontos de integração de uma aplicação com recursos externos. É a abordagem usada por aqui.
- Testes de Integração amplos (em inglês, *Broad Integration Tests*), que é o que chamamos aqui de Testes de Componentes.

No Caelum Eats, faríamos Integration Tests (restritos) para classes como:

- `PagamentoRepository`, que integra com o MySQL
- `PedidoRestClient`, que invoca o Monólito usando HTTP como

protocolo

- NotificadorPagamentoConfirmado, que publica o Domain Event PagamentoConfirmado no um Message Channel pagamentosConfirmados



Para a testar um ponto de integração com um BD, como PagamentoRepository, podemos usar um BD em memória como [H2](#), [HSQLDB](#) ou [Derby](#). Outra abordagem é usar um BD real, que pode apresentar erros que não seriam apresentados em um BD em memória. Tecnologias como [Testcontainers](#) ajudam no gerenciamento desses BD reais. Ferramentas de migration como o [Flyway](#) ou [Liquibase](#) auxiliam a criar a estrutura de um BD relacional necessária para os testes.

E como testar integrações com outros serviços e com Message Channels? Veremos isso mais adiante.

Testes de Componentes

Uma outra abordagem seria testar a integração entre os vários componentes de uma aplicação. No caso de uma Arquitetura de Microservices, estaríamos exercitando um serviço individualmente por sua API e verificando se os resultados retornados correspondem aos esperados.

Os recursos usados pelo serviço a ser testado, como BDs, caches, Message Brokers e outros serviços, seriam trocados por Test Doubles.

Chris Richardson chama esse tipo de teste, no livro [Microservices](#)

[Patterns](#) (RICHARDSON, 2018a), de Testes de Componentes (em inglês, **Component Tests**). Essa nomenclatura também é [usada por Martin Fowler](#).

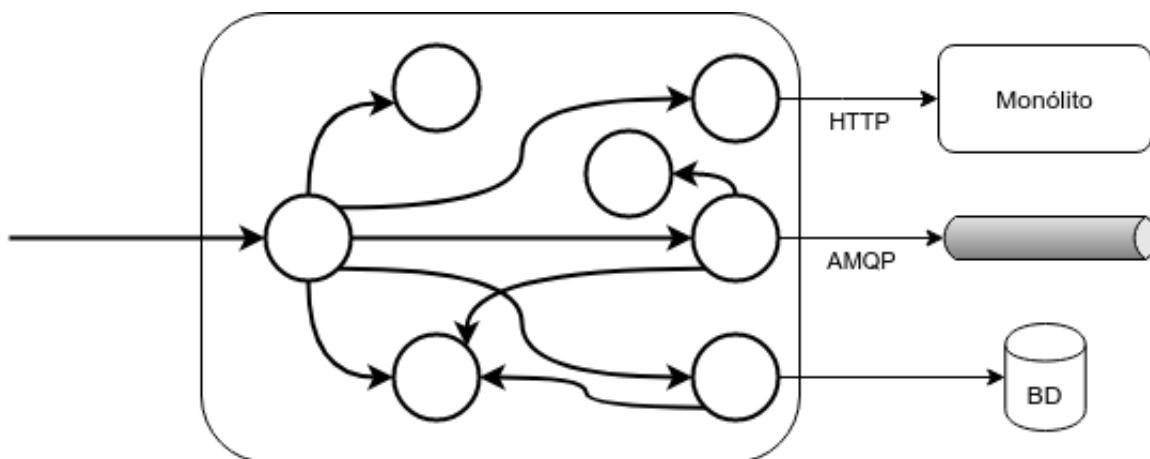
Há bastante discussão sobre a nomenclatura desse tipo de teste.

Alguns usam o termo Integration Tests, equivalendo aos Broad Integration Tests do artigo [Integration Test](#) (FOWLER, 2018) de Martin Fowler.

Como os testes não são pela UI, mas diretamente pela API de um serviço isolado, como se fossem por baixo da "pele" do sistema, Martin Fowler também usa o termo [Subcutaneous Test](#).

Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), usa o termo *Service Tests*, citando o agilista Mike Cohn.

No Caelum Eats, um Component Tests para o serviço de Pagamentos exercitaria o `PagamentoController` que, por sua vez, chamaria o restante das classes.



Testes End-to-End

Testes End-to-End (em inglês, **End-to-End Tests**) exercitam uma aplicação como um todo. Em uma Arquitetura de Microservices, os diversos serviços e toda a infraestrutura usada por esses serviços deve ser colocada no ar.

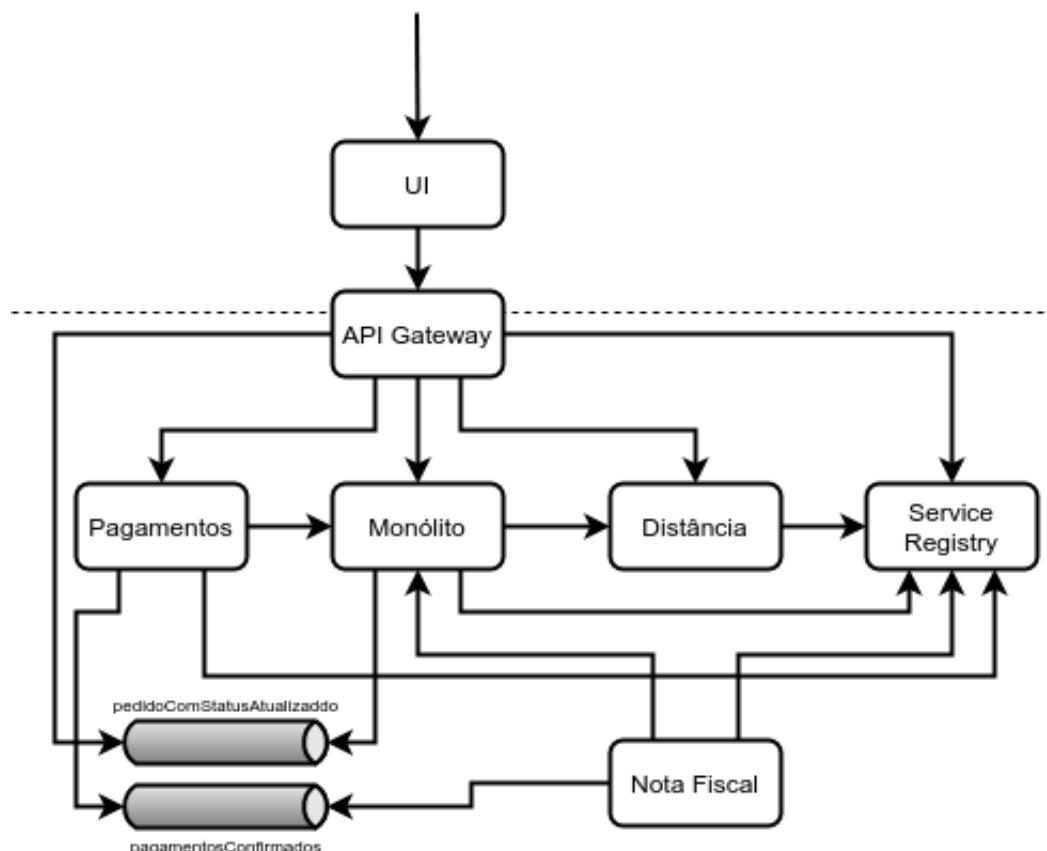
Sam Newman, no livro [Building Microservices](#) (NEWMAN, 2015), cita o

trabalho do agilista Mike Cohn para explicar que o intuito dos End-to-End Tests é exercitar as funcionalidades por meio da UI, passando por tudo o que estiver abaixo, de maneira oferecer uma verificação de grande quantidade do sistema.

Assim como em outros tipos de testes, a terminologia é variada.

Martin Fowler, por exemplo, também usa os nomes [Broad-stack Test](#) e [Full-stack Test](#).

No Caelum Eats, um End-to-End Test deveria ter no ar: a UI, o API Gateway, o Monólito, o serviço de Pagamentos, o serviço de Distância, o serviço de Nota Fiscal, além do Service Registry, do Message Broker e dos BDs de cada serviço. É muita coisa!



Martin Fowler argumenta, no artigo [Broad-stack Test](#) (FOWLER, 2013), que End-to-End Tests não necessariamente precisam usar a UI. O sistema pode ser exercitado por uma API, ainda mantendo no ar vários serviços e a infraestrutura necessária. Ou seja, End-to-End Tests podem ser o que Fowler chama de [Subcutaneous Tests](#).

Uma ideia comum nos End-to-End Tests é simular um fluxo do usuário, baseado nos critérios de aceitação de Histórias do Usuário (em inglês, User Stories). Uma vantagem é prover rastreabilidade entre os requisitos e a execução dos testes. Esse tipo de teste é chamado por Martin Fowler de [StoryTest](#).

No livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman lista diversos pontos negativos de End-to-End Tests:

- Fragilidade: ao testar uma funcionalidade que passa por quatro ou cinco serviços, há muita coisa que pode dar errado. Por exemplo, pode haver uma falha momentânea na rede ou uma demora que leva a um timeout, quebrando o teste mesmo que a funcionalidade sendo testada esteja sem defeitos. Executar novamente o mesmo teste pode levá-lo a passar. A confiança no resultado diminui, o que pode levar os testes a serem ignorados. Newman recomenda uma ideia de Martin Fowler de colocar testes frágeis em quarentena, removendo-os da suíte de testes e encontrando maneiras de deixá-los mais estáveis.
- Código sem dono: quem mantém os End-to-End Tests? Se forem todos os times de todos os serviços, quando houver uma falha há o risco de cada time ignorar, achando que é um problema dos outros times. Tem um time dedicado a End-to-End Tests é ruim, porque tira a autonomia dos times dos serviços.
- Lentidão: há suítes de End-to-End Tests que duram horas, dias e até semanas para serem executadas. O feedback proporcionado pelos testes fica prejudicado. Além disso, colocar esses testes lentos em um Build Pipeline que entrega software continuamente, faria com que mudanças fossem empilhadas. Isso aliado à fragilidade é desastroso!

Newman relata que é comum, em empresas que usam uma Arquitetura de Microservices, remover os End-to-End Tests, favorecendo monitoramento detalhado.

Para minimizar o número de End-to-End Tests, diversos autores como

Chris Richardson, Sam Newman e Martin Fowler, recomendam simular não uma User Story, mas uma série de interações do usuário que visam atingir algum objetivo de negócio. Por exemplo, não seria criado um End-to-End Test somente da primeira etapa de um pedido. O teste deveria incluir da criação e pagamento até a finalização do pedido, sem considerar casos de exceção. É o que Martin Fowler chama de Testes de Jornadas de Usuário (em inglês, [User Journey Tests](#)).

Ainda outros tipos de testes

Como testar de maneira automatizada a Usabilidade do Sistema?

Testes repetitivos, que verificam funcionalidades e detectam regressões devem ser automatizados. Mas ainda são necessários testes manuais, que permitam que o software seja criticado com um olhar humano.

Além de Usabilidade, podem ser feitos Testes Exploratórios por especialistas em testes, buscando maneiras de quebrar o sistema.

Há ainda testes de *ilities*, requisitos técnicos e transversais como Performance, Escalabilidade, Segurança. Esse tipo de teste pode ser feito por especialistas, auxiliados pelas ferramentas adequadas.

Martin Fowler argumenta no artigo [Threshold Test](#) (FOWLER, 2013b), que até é possível fazer testes de Performance de maneira automatizada. Poderíamos chamar um conjunto de operações do sistema, anotando o tempo de resposta. Mas ao invés de verificarmos valores exatos, compararíamos com um valor limite. Num ambiente de Integração Contínua, esse tipo de teste ajuda a identificar uma degradação de performance assim que o código for comitado.

Pirâmide de Testes

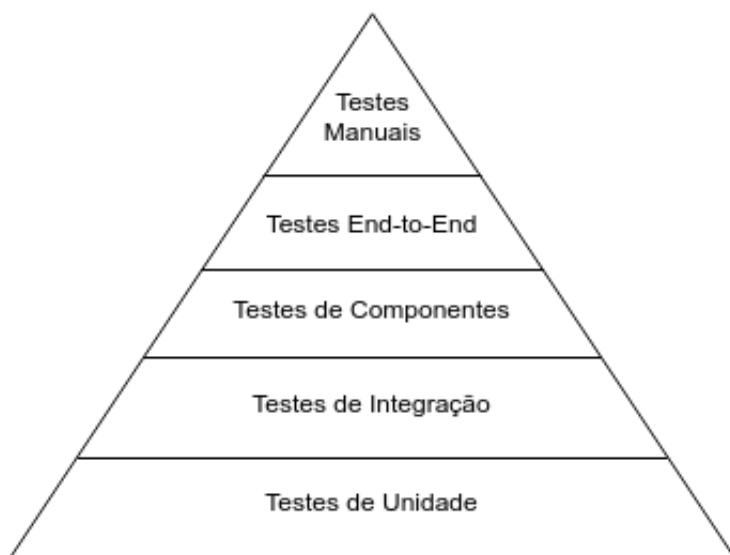
Testes de Unidade são rápidos: uma suíte extensa em geral é executada em milissegundos ou segundos. Além disso, são bem baratos de se escrever porque estão isolados e no mesmo nível de abstração do código. Aliados a práticas como TDD e refatoração, influenciam

positivamente no design do nosso código.

Testes de Integração e de Componentes são um pouco mais lentos, tendo uma suíte executada em alguns minutos, em geral. É comum acessarem outras peças da infra-estrutura como Bancos de Dados ou WebServices. Por isso, seu setup é mais complicado e tais testes acabam sendo mais caros de desenvolver.

Testes End-to-End são os mais lentos, chegando a horas de execução, além de caros, pela complexidade de escrevê-los. Há diversos motivos para pode quebrá-los e, às vezes, apresentam falhas intermitentes.

Martin Fowler cita a ideia do agilista Mike Cohn de uma Pirâmide de Testes, que indica que Testes de Unidade devem ser favorecidos, em detrimento de formas de testes mais lentas, caras e frágeis.



Há relatos de sistemas no mercado que têm classes responsáveis por diversas funcionalidades e regras de negócio, chegando a 15 mil linhas de código.

Além de incompreensível, um código tão desorganizado é difícil de testar porque torna-se complicado exercitar isoladamente uma regra de negócio ou um ponto de integração. Em geral, apenas testes end-to-end são possíveis para esse tipo de código. Como diz Michael Feathers em um [post em seu blog](#) (FEATHERS, 2007), há uma forte sinergia entre um bom design de código e testabilidade.

Martin Fowler [relata em seu blog](#) (FOWLER, 2005b) que uma pessoa no grupo de eXtreme Programming de Sidney fez a seguinte brincadeira:

Detestável (adjetivo): software que não é testável.

Test Doubles

Conforme mencionamos, Testes de Unidade do estilo Solitário, Testes de Integração e Testes de Componentes, trocam dependências a serviços externos com o BDs ou outros serviços por *Test Doubles*, objetos projetados para serem usados especificamente nos testes.

De maneira semelhante às outras terminologias de Testes Automatizados, há uma grande confusão na classificação de Test Doubles. Muitas vezes, o único termo usado é Mock.

No artigo [Test Double](#) (FOWLER, 2006), Martin Fowler sugere o uso dos termos de Gerard Meszaros:

- *Dummy*: um objeto passado apenas para preencher argumentos de um método, mas que não é realmente invocado.
- *Fake*: um objeto com uma implementação funcional, mas que não deve ser usada em produção. Por exemplo, um BD em memória.
- *Stub*: provê respostas pré-estabelecidas às chamadas feitas durante um teste. Ignoram chamadas não esperadas.
- *Spy*: stubs que gravam informações sobre como foram chamados. Por exemplo, um serviço de email que conta quantas mensagens foram enviadas.
- *Mock*: especifica as interações exatas com objetos que o usam. Há uma verificação de que todas as chamadas esperadas devem ser realizadas, retornando os resultados pré-estabelecidos. Nenhuma chamada a mais pode ser feita.

No paper [Mock Roles, not Objects](#) (FREEMAN et al., 2004), os autores, pioneiros de Test Doubles, descrevem o princípio *Only Mock Types You*

Own: não devem ser feitos Mocks de tipos (classes, interfaces, etc) cujo código não podemos modificar, como tipos de bibliotecas externas.

Stub Services e WireMock

Quando implementamos Integration e Component Tests, precisamos exercitar as integrações com serviços externos.

Mas invocar instâncias reais desses serviços externos é algo bastante trabalhoso. Seria necessário executar toda a infraestrutura e os serviços externos usados por esses serviços externos. A complexidade aumentaria absurdamente.

Uma abordagem mais interessante é usar Test Doubles para cada um dos serviços externos usados por uma aplicação. Usando a terminologia estudada anteriormente, criariamos Stubs para esses serviços.

No livro [Building Microservices](#) (NEWMAN, 2015), Sam Newman relata que já criou esses **Stub Services** na mão, executando código baseado em servidores HTTP como Apache, Nginx, Jetty ou implementados em Python.

Mas existem Stub Services já prontos, como:

- [mountebank](#), escrito em NodeJS
- [WireMock](#), escrito em Java

Ambos provêem uma API RESTful que permite que aplicações escritas em múltiplas plataformas registrem quais os requests esperados e os responses associados a esses requests. Então, testes usariam a API como um Stub do serviço real.

Também há como verificar as chamadas realmente realizadas durante o teste, servindo como um Mock.

WireMock

Focando no WireMock, para executá-lo, podemos disparar o JAR executável da seguinte forma:

```
java -jar wiremock-standalone-2.26.0.jar --port 7070
```

Observação: a porta padrão é a 8080. Com a opção --port, modificamos a porta do WireMock para 7070.

Para registrar um novo request e o respectivo response, de maneira a simular uma chamada que obtém os detalhes de um pedido do Monólito, faríamos:

```
POST http://localhost:7070/_admin/mappings/new
```

```
{
  "request": {
    "url": "/pedidos/1",
    "method": "GET"
  },
  "response": {
    "status": 200,
    "body": "{\"id\": 1, \"dataHora\": \"2019-07-18T08:22:01\""
  }
}
```

Então, qualquer request à URL <http://localhost:7070/pedidos/1>, teria como response o status code 200 OK e um JSON com os dados do pedido registrados no passo anterior.

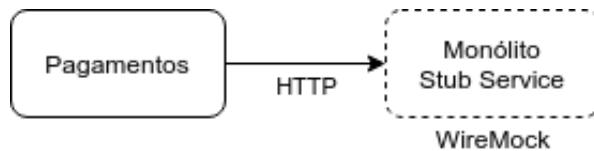
Requisições a outras URLs ocasionariam um status code 404 Not Found.

Quem deve manter o código do Stub Service?

Digamos que estamos criando um Integration Test da classe

`PedidoRestClient`: o ponto de integração do serviço de Pagamentos com o módulo de Pedido do Monólito.

O Integration Test do serviço de Pagamentos poderia utilizar o WireMock para subir um Stub Service do Monólito e carregá-lo com uma série de requests e responses pré-determinados.



Mas quem deve manter esse conjunto de requests e responses?

Seguindo o princípio *Only Mock Types You Own*, o serviço de Pagamentos não deveria manter o Stub Service do Monólito. Deveria ser o próprio time que desenvolve o Monólito.

Para auxiliar nessa tarefa, poderiam ser usados contratos.

Contratos entre serviços

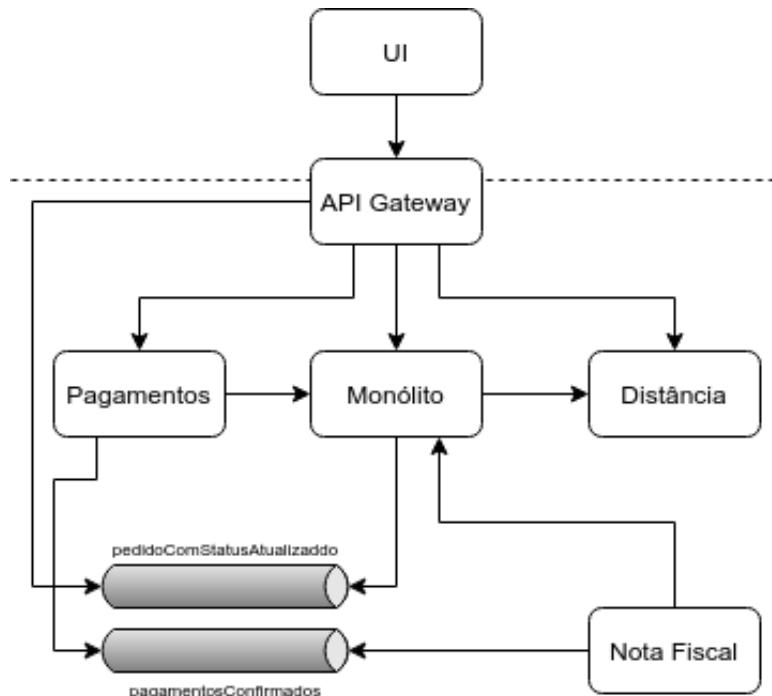
Nesse momento, há diversas integrações entre os serviços.

Temos integrações síncronas e RESTful, que usam HTTP:

- o API Gateway invoca os módulos do Monólito e os serviços de Pagamentos e Distância, tanto como Proxy quanto como API Composer
- o serviço de Pagamentos invoca o módulo de Pedido do Monólito, avisando que um pedido foi pago
- o módulo de Restaurante do Monólito invoca o serviço de Distância, informando que novos restaurantes foram aprovados ou que um restaurante foi atualizado
- o serviço de Nota Fiscal busca detalhes de um pedido do módulo de Pedido do Monólito

Temos integrações assíncronas, que usam Mensageria com AMQP:

- o serviço de Pagamentos produz um Domain Event que indica que um pagamento foi confirmado, que é consumido pelo serviço de Nota Fiscal
- o módulo de Pedido do Monólito produz um Domain Event para cada atualização de status de um pedido, cujo consumidor é o API Gateway, que usa um WebSocket para notificar o front-end



Em cada uma dessas integrações há um contrato.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a), diz que no caso das endpoints REST, o um contrato é composto por:

- o método HTTP usado no request
- a URL invocada no request
- os cabeçalhos HTTP no request e no response, como Content-Type e Accept
- o corpo do request e do response, que contém os dados
- o status code do response

Já para integrações que usam Mensageria, Richardson indica que o contrato é composto por:

- a estrutura do corpo da mensagem
- o Message Channel utilizado

Testes de Contratos

Os contratos entre serviços mencionados anteriormente poderiam ser usados para influenciar na criação de Stub Services que são usados em Integration e Component Tests.

Esse contratos seriam materializados em um ou mais arquivos que provêem exemplos dos requests, responses ou mensagens trafegadas na integração entre os serviços.

Algumas ferramentas ajudam a definir e executar esses contratos. Entre elas:

- [Pact](#), uma ferramenta que checa se chamadas ao serviço real tem os mesmos resultados das definidas no contrato. É implementado em diversas linguagens.
- [Spring Cloud Contract](#), uma ferramenta do ecossistema Spring que ajuda a definir contratos tanto para APIs REST como para Mensageria, que são usados para gerar testes automatizados e Stub Services.

Spring Cloud Contract

Voltando à integração entre o serviço de Pagamentos e o Monólito, como usar o Spring Cloud Contract para testá-la?

O Monólito, nos temos do Spring Cloud Contract, seria o Producer. Já o serviço de Pagamentos, seria o Consumer.

Do lado do Producer, usamos o Spring Cloud Contract Verifier. Com esse projeto, podemos definir exemplos de request/response HTTP ou mensagens usando uma DSL Groovy ou YAML. Esse arquivo é a materialização do contrato entre os serviços.

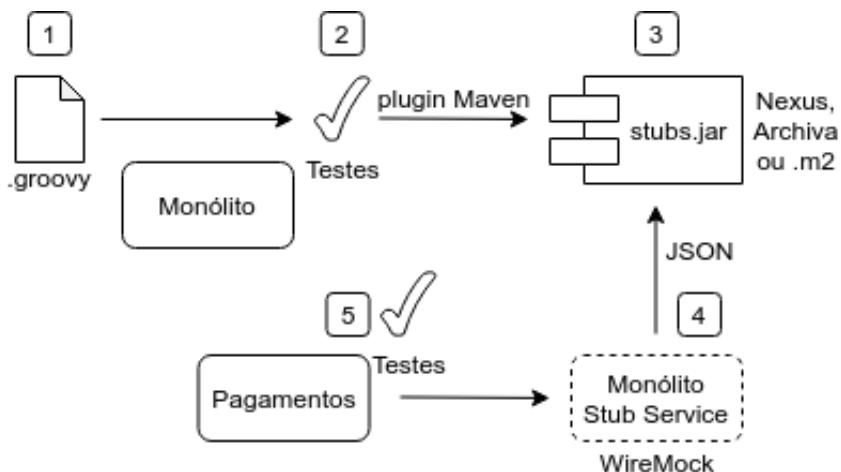
Ainda do lado do Producer, através de um plugin do Maven, esse contrato é usado para gerar automaticamente Component Tests, verificando se o próprio Producer segue os contratos. É necessário

fornecer uma classe base que é estendida por todos os Component Tests gerados. Por exemplo, para um contrato de API REST, a classe base ficaria responsável por configurar o `RestAssuredMockMvc`.

No caso do build do Maven ser bem sucedido, é gerado um JAR contendo o contrato. No caso de contratos de APIs REST, o JAR também contém um JSON no formato do WireMock. É o caso da integração entre o serviço de Pagamentos e o Monólito. Esse JAR é publicado em algum repositório de artefatos como o Nexus, Archiva ou localmente no diretório `.m2`.

Já do lado do Consumer, o JAR é obtido e executado usando o Spring Cloud Contract Stub Runner. No caso de uma API REST, o WireMock é iniciado e o JSON com os exemplos de request/response são carregados.

Então, os pontos de integração são exercitados pelos testes automatizados usando o Stub fornecido pelo Producer.



Fornecendo stubs do contrato a partir do servidor

Adicione ao `pom.xml` do serviço de distância, uma dependência ao starter do Spring Cloud Contract Verifier:

```
##### fj33-eats-distancia-service/pom.xml
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-contract-verifier</artifactId>
  <scope>test</scope>
</dependency>
```



Adicione também o plugin Maven do Spring Cloud Contract:

```
##### fj33-eats-distancia-service/pom.xml
```

```
<plugin>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-contract-maven-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <packageWithBaseClasses>br.com.caelum.eats.distancia.base</packageWithBaseClasses>
  </configuration>
</plugin>
```



Note que na configuração `packageWithBaseClasses` definimos um pacote para as classes base, que serão usadas na execução de testes.

No Eclipse, com o botão direito no projeto `eats-distancia-service`, acesse o menu *New > Folder...*. Defina em *Folder name*, o caminho `src/test/resources/contracts/restaurante`.

Dica: para que o diretório `src/test/resources` seja reconhecido como um source folder faça um refresh no projeto e, com o botão direito no projeto, clique em Maven > Update Project... e, então, em OK.

Dentro desse diretório, crie o arquivo `deveAdicionarNovoRestaurante.groovy`. Esse arquivo conterá o contrato que estamos definindo, utilizando uma DSL Groovy:

```
##### fj33-eats-distancia-
```

service/src/test/resources/contracts/restaurantes/deveAdicionarNovoRestaurante.groovy

```
import org.springframework.cloud.contract.spec.Contract
Contract.make {
    description "deve adicionar novo restaurante"
    request{
        method POST()
        url("/restaurantes")
        body([
            id: 2,
            cep: '71500-000',
            tipoDeCozinhaId: 1
        ])
        headers {
            contentType('application/json')
        }
    }
    response {
        status 201
        body([
            id: 2,
            cep: '71500-000',
            tipoDeCozinhaId: 1
        ])
        headers {
            contentType('application/json')
        }
    }
}
```

No serviço de distância, clique com o botão direto no projeto e então acesse o menu *New > Folder...* e defina, em *Folder name*, o caminho `src/test/java`. Será criado um source folder de testes.

No pacote `br.com.caelum.eats.distancia.base` do *source folder* `src/test/java`, definido anteriormente no plugin do Maven, crie a classe

a classe `RestaurantesBase`, que será a base para a execução de testes baseados no contrato do controller de restaurantes.

Dica: para que o diretório `src/test/java` seja reconhecido como um source folder faça um refresh no projeto e, com o botão direito no projeto, clique em `Maven > Update Project...` e, então, em `OK`.

Nessa classe injete o `RestaurantesController`, passando a instância para o `RestAssuredMockMvc`, uma integração da biblioteca REST Assured com o `MockMvc` do Spring.

Além disso, injetaremos um `RestauranteRepository` anotado com `@MockBean`, fazendo com que a instância seja gerenciada pelo Mockito. Usaremos essa instância como um *stub*, registrando uma chamada ao método `insert` que retorna o próprio objeto passado como parâmetro.

Para evitar que o Spring tente conectar com o MongoDB durante os testes, anote a classe com `@ImportAutoConfiguration`, passando na propriedade `exclude` a classe `MongoAutoConfiguration`.

Observação: o nome da classe `RestaurantesBase` usa como prefixo o diretório de nosso contrato (`restaurantes`). O sufixo `base` é um requisito do Spring Cloud Contract.

```
##### fj33-eats-distancia-
service/src/test/java/br/com/caelum/eats/distancia/base/RestaurantesBa
se.java
```

```
@ImportAutoConfiguration(exclude=MongoAutoConfiguration.class)
@SpringBootTest
@RunWith(SpringRunner.class)
class RestaurantesBase {

    @Autowired
    private RestaurantesController restaurantesController;
```

```
@MockBean  
private RestauranteRepository restauranteRepository;  
  
@Before  
public void before() {  
    RestAssuredMockMvc.standaloneSetup(restauranteController)  
  
    Mockito.when(restauranteRepository.insert(Mockito.any(Rest  
        .thenAnswer((InvocationOnMock invocation) -> {  
            Restaurante restaurante = invocation.getArgument(0);  
            return restaurante;  
        }));  
  
    }  
}
```

Os imports são os seguintes:

```
import org.junit.Before;  
import org.junit.runner.RunWith;  
import org.mockito.Mockito;  
import org.mockito.invocation.InvocationOnMock;  
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;  
  
import br.com.caelum.eats.distancia.Restaurante;  
import br.com.caelum.eats.distancia.RestauranteRepository;  
import br.com.caelum.eats.distancia.RestauranteController;  
import io.restassured.module.mockmvc.RestAssuredMockMvc;  
  
import io.restassured.module.mockmvc.RestAssuredMockMvc;
```

Altere a classe `RestauranteController`, tornando-a pública:

```
// anotações omitidas ...
```

```
public class RestaurantesController { // modificado  
    // código omitido ...  
}
```

Também torne pública a interface RestauranteRepository:

```
public interface RestauranteRepository extends MongoRepository  
    // código omitido ...  
}
```

Abra um Terminal e, no diretório do serviço de distância, execute:

Depois do sucesso no build, podemos observar que uma classe RestaurantesTest foi gerada pelo Spring Cloud Contract:

```
##### fj33-eats-distancia-service/target/generated-test-  
sources/contracts(br/com/caelum/eats/distancia/base/RestaurantesTest.j  
ava
```

```
public class RestaurantesTest extends RestaurantesBase {  
  
    @Test  
    public void validate_deveAdicionarNovoRestaurante() throws E  
        // given:  
        MockMvcRequestSpecification request = given()  
            .header("Content-Type", "application/json")  
            .body("{\"id\":2,\"cep\":\"71500-000\",\"tipoDeCozir  
  
        // when:  
        ResponseOptions response = given().spec(request)  
            .post("/restaurantes");
```

```

// then:
assertThat(response.statusCode()).isEqualTo(201);
assertThat(response.header("Content-Type")).matches("app
// and:
DocumentContext parsedJson = JsonPath.parse(response.get
assertThatJson(parsedJson).field("[ 'tipoDeCozinhaId' ]").
assertThatJson(parsedJson).field("[ 'cep' ]").isEqualTo("7
assertThatJson(parsedJson).field("[ 'id' ]").isEqualTo(2);
}

}

```

A classe `RestaurantesTest` é responsável por verificar que o próprio servidor segue o contrato.

Além do *fat JAR* gerado pelo Spring Boot com a aplicação, o Spring Cloud Contract gera um outro JAR com stubs do contrato: `eats-distancia-service/target/eats-distancia-service-0.0.1-SNAPSHOT-stubs.jar`.

Dentro do diretório `/META-INF/br.com.caelum/eats-distancia-service/0.0.1-SNAPSHOT/` desse JAR, no subdiretório `contracts/restaurantes/`, há a DSL Groovy que descreve o contrato, no arquivo `deveAdicionarNovoRestaurante.groovy`.

Já no subdiretório `mappings/restaurantes/`, há o arquivo `deveAdicionarNovoRestaurante.json`:

```
{
  "id" : "80bcbe99-0504-4ff9-8f32-e9eb0645b646",
  "request" : {
    "url" : "/restaurantes",
    "method" : "POST",
    "headers" : {
      "Content-Type" : {
        "matches" : "application/json.*"
      }
    }
  }
}
```

```
        }
    },
    "bodyPatterns" : [ {
        "matchesJsonPath" : "$[?(@.['tipoDeCozinhaId'] == 1)]"
    }, {
        "matchesJsonPath" : "$[?(@.['cep'] == '71500-000')]"
    }, {
        "matchesJsonPath" : "$[?(@.['id'] == 2)]"
    } ]
},
"response" : {
    "status" : 201,
    "body" : "{\"tipoDeCozinhaId\":1,\"id\":2,\"cep\":\"71500-000\"}",
    "headers" : {
        "Content-Type" : "application/json"
    },
    "transformers" : [ "response-template" ]
},
"uuid" : "80bcbe99-0504-4ff9-8f32-e9eb0645b646"
}
```

Esse JSON é compatível com a ferramenta WireMock, que permite a execução de um *mock server* para testes de API.

Usando stubs do contrato no cliente

No `pom.xml` do módulo `eats-application` do monólito, adicione o starter do Spring Cloud Contract Stub Runner:

```
##### fj33-eats-monolito-modular/eats/eats-application/pom.xml
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>
    <scope>test</scope>
</dependency>
```

No source folder `src/test/java` do módulo `eats-application`, dentro do pacote `br.com.caelum.eats`, crie a classe `DistanciaRestClientWiremockTest`.

Anote-a com `@AutoConfigureStubRunner`, passando no parâmetro `ids`, o `groupId` e `artifactId` do JAR gerado no exercício anterior. Use um + para sempre obter a última versão. Passe também a porta que deve ser usada pelo servidor do WireMock. No parâmetro `stubsMode`, informe que o JAR do contrato será obtido do repositório `LOCAL` (o diretório `.m2`).

Em um método anotado com `@Before`, crie uma instância do `DistanciaRestClient`, o ponto de integração do monólito com o serviço de distância. Passe um `RestTemplate` sem平衡amento de carga e fixe a URL para a porta definida na anotação `@AutoConfigureStubRunner`.

Invoque o método `novoRestauranteAprovado` de `DistanciaRestClient`, passando um objeto `Restaurante` com valores condizentes com o contrato. Como o método é `void`, em caso de exceção force a falha do teste.

```
##### fj33-eats-monolito-modular/eats/eats-
application/src/test/java(br/com/caelum/eats/DistanciaRestClientWiremockTest.java)
```

```
@SpringBootTest
@RunWith(SpringRunner.class)
@AutoConfigureStubRunner(ids = "br.com.caelum:eats-distancia-s
public class DistanciaRestClientWiremockTest {

    private DistanciaRestClient distanciaClient;

    @Before
    public void before() {
        RestTemplate restTemplate = new RestTemplate();
        distanciaClient = new DistanciaRestClient(restTemplate, "t
    }
```

```
@Test
public void deveAdicionarUmNovoRestaurante() {
    TipoDeCozinha tipoDeCozinha = new TipoDeCozinha(1L, "Chi

    Restaurante restaurante = new Restaurante();
    restaurante.setId(2L);
    restaurante.setCep("71500-000");
    restaurante.setTipoDeCozinha(tipoDeCozinha);

    distanciaClient.novoRestauranteAprovado(restaurante);
}

}
```

Observação: o teste anterior falhará quando for lançada uma exceção.

Seguem os imports:

```
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.cloud.contract.stubrunner.spring.AutoConfigureStubRunner;
import org.springframework.cloud.contract.stubrunner.spring.StubRunner;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.web.client.RestTemplate;

import br.com.caelum.eats.administrativo.TipoDeCozinha;
import br.com.caelum.eats.restaurante.DistanciaRestClient;
import br.com.caelum.eats.restaurante.Restaurante;
```

No módulo de restaurante do monólito, torne públicos a classe DistanciaRestClient, seu construtor e o método novoRestauranteAprovado:

```
##### fj33-eats-monolito-modular/eats-
restaurante/src/main/java;br/com/caelum/eats/restaurante/DistanciaRest
```

Client.java

```
@Slf4j
@Service
public class DistanciaRestClient { // modificado

    // código omitido ...

    public DistanciaRestClient(RestTemplate restTemplate, // moc
                               @Value("${configuracao.c
    this.distanciaServiceUrl = distanciaServiceUrl;
    this.restTemplate = restTemplate;
}

public void novoRestauranteAprovado(Restaurante restaurante)
    // código omitido ...
}

// restante do código ...

}
```

Execute a classe `DistanciaRestClientWiremockTest` com o JUnit 4.

Observe, nos logs, a definição no WireMock do contrato descrito no arquivo `deveAdicionarNovoRestaurante.json` do JAR de stubs.

```
2019-07-03 17:41:27.681  INFO [monolito,,,] 32404 --- [tp1306763722-35] Wir
127.0.0.1 - POST /mappings

Connection: [keep-alive]
User-Agent: [Apache-HttpClient/4.5.5 (Java/1.8.0_201)]
Host: [localhost:9992]
Content-Length: [718]
Content-Type: [text/plain; charset=UTF-8]
{
    "id" : "64ce3139-e460-405d-8ebb-fe7f527018c3",
    "nome" : "Pizzaria Vai Que Cola"
}
```

```

"request" : {
    "url" : "/restaurantes",
    "method" : "POST",
    "headers" : {
        "Content-Type" : {
            "matches" : "application/json.*"
        }
    },
    "bodyPatterns" : [ {
        "matchesJsonPath" : "$[?(@.[ 'tipoDeCozinhaId' ] == 1)]"
    }, {
        "matchesJsonPath" : "$[?(@.[ 'cep' ] == '71500-000')]"
    }, {
        "matchesJsonPath" : "$[?(@.[ 'id' ] == 2)]"
    } ]
},
"response" : {
    "status" : 201,
    "body" : "{\"tipoDeCozinhaId\":1,\"id\":2,\"cep\":\"71500-000\"}",
    "headers" : {
        "Content-Type" : "application/json"
    },
    "transformers" : [ "response-template" ]
},
"uuid" : "64ce3139-e460-405d-8ebb-fe7f527018c3"
}

```

Mais adiante, observe que o WireMock recebeu uma requisição POST na URL /restaurantes e enviou a resposta descrita no contrato:

```

2019-07-03 17:41:37.689  INFO [monolito,,,] 32404 --- [tp1306763722-36] Wir
Request received:
127.0.0.1 - POST /restaurantes

User-Agent: [Java/1.8.0_201]
Connection: [keep-alive]
Host: [localhost:9992]
Accept: [application/json, application/*+json]

```

```
Content-Length: [46]
Content-Type: [application/json; charset=UTF-8]
{"id":2,"cep":"71500-000","tipoDeCozinhaId":1}

Matched response definition:
{
    "status" : 201,
    "body" : "{\"tipoDeCozinhaId\":1,\"id\":2,\"cep\":\"71500-000\"}",
    "headers" : {
        "Content-Type" : "application/json"
    },
    "transformers" : [ "response-template" ]
}

Response:
HTTP/1.1 201
Content-Type: [application/json]
Matched-Stub-Id: [64ce3139-e460-405d-8ebb-fe7f527018c3]
```

Exercício: Contract Test para comunicação síncrona

1. Abra um Terminal e vá até a branch `cap12-contrato-cliente-servidor` do projeto do serviço de distância:

```
cd ~/Desktop/fj33-eats-distancia-service
git checkout -f cap12-contrato-cliente-servidor
```

Então, faça o build do serviço de distância, rode o Contract Test no próprio serviço, gere o JAR com os stubs do contrato e instale no repositório local do Maven. Basta executar o comando:

Aguarde a execução do build. As mensagens finais devem conter:

```
[INFO] Results:
[INFO]
```

```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
...
[INFO] Installing /home/<USUARIO-DO-CURSO>/Desktop/fj33-eats-distancia-serv
[INFO] Installing /home/<USUARIO-DO-CURSO>/Desktop/fj33-eats-distancia-serv
[INFO] Installing /home/<USUARIO-DO-CURSO>/Desktop/fj33-eats-distancia-serv
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 02:45 min
[INFO] Finished at: 2019-07-03T16:45:17-03:00
[INFO] -----
```

Observe, pelas mensagens anteriores, que o JAR com os stubs foi instalado no diretório `.m2`, o repositório local Maven, do usuário do curso.

2. No projeto do monólito modular, faça checkout da branch `cap12-contrato-cliente-servidor`:

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap12-contrato-cliente-servidor
```

Faça o refresh do projeto no Eclipse.

Clique com o botão direito na classe `DistanciaRestClientWiremockTest`, do módulo `eats-application` do monólito, e, então, em *Run As... > Run Configurations...*. Clique com o botão direito em *JUnit* e, a seguir, em *New Configuration*. Em *Test runner*, escolha o *JUnit 4*. Então, clique em *Run*.

Aguarde a execução dos testes. Sucesso!

Definindo um contrato no publisher

Adicione, ao `pom.xml` do serviço de pagamentos, as dependências ao

starter do Spring Cloud Contract Verifier e à biblioteca de suporte a testes do Spring Cloud Stream:

```
##### fj33-eats-pagamento-service/pom.xml
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-verifier</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-test-support</artifactId>
    <scope>test</scope>
</dependency>
```

Adicione também o plugin Maven do Spring Cloud Contract, configurando `br.com.caelum.eats.pagamento.base` como pacote das classes base a serem usadas nos testes gerados a partir dos contratos.

```
##### fj33-eats-pagamento-service/pom.xml
```

```
<plugin>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-contract-maven-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <packageWithBaseClasses>br.com.caelum.eats.pagamento.base</packageWithBaseClasses>
    </configuration>
</plugin>
```

Dentro do Eclipse, clique com o botão direito no projeto `eats-pagamento-service`, acessando *New > Folder...* e definindo o caminho `src/test/resources/contracts/pagamentos/confirmados` em *Folder name*.

Dica: para que o diretório `src/test/resources` seja reconhecido como um source folder faça um refresh no projeto e, com o botão direito no projeto, clique em Maven > Update Project... e, então, em OK.

Crie o arquivo `deveAdicionarNovoRestaurante.groovy` nesse diretório, definindo o contrato por meio da DSL Groovy:

```
##### fj33-eats-pagamento-
service/src/test/resources/contracts/pagamentos/confirmados/deveNotif
icarPagamentosConfirmados.groovy
```

```
import org.springframework.cloud.contract.spec.Contract
Contract.make {
    description "deve notificar pagamentos confirmados"
    label 'pagamento_confirmado'
    input {
        triggeredBy( 'novoPagamentoConfirmado()')
    }
    outputMessage {
        sentTo 'pagamentosConfirmados'
        body([
            pagamentoId: 2,
            pedidoId: 3
        ])
        headers {
            messagingContentType(applicationJson())
        }
    }
}
```

Definimos `pagamento_confirmado` como `label`, que será usado nos testes do subscriber. Em `input`, invocamos o método `novoPagamentoConfirmado` da classe base. Já em `outputMessage`, definimos `pagamentosConfirmados` como *destination* esperado, o corpo da mensagem e o *Content Type* nos cabeçalhos.

Defina um source folder de testes no projeto `fj33-eats-pagamento-service`. Para isso, no Eclipse, clique com o botão direto no projeto e então acesse o menu *New > Folder...* e defina, em *Folder name*, o caminho `src/test/java`

No pacote `br.com.caelum.eats.pagamento.base`, do source folder `src/test/java`, crie a classe `PagamentosConfirmadosBase`. Anote essa classe com `@AutoConfigureMessageVerifier`, além das anotações de testes do Spring Boot. Na anotação `@SpringBootTest`, configure o `webEnvironment` para `NONE`.

Para que o teste não tente conectar com o MySQL, use a anotação `@ImportAutoConfiguration` com a class `DataSourceAutoConfiguration` no atributo `exclude`. Ocorrerá um problema na criação de `PagamentoRepository` pelo Spring Data JPA, já que não teremos mais um data source configurado. Por isso, faça um mock de `PagamentoRepository` com `@MockBeans`.

Peça ao Spring para injetar uma instância da classe `NotificadorPagamentoConfirmado`.

Defina um método `novoPagamentoConfirmado`, que usa a instância injetada para chamar o método `notificaPagamentoConfirmado` passando como parâmetro um `Pagamento` com dados compatíveis com o contrato definido anteriormente.

Observação: o nome da classe `PagamentosConfirmadosBase` usa como prefixo o diretório de nosso contrato (`pagamentos/confirmados`) com o sufixo `Base`.

```
##### fj33-eats-pagamento-
service/src/test/java(br/com/caelum/eats/pagamento/base/PagamentosC
onfirmadosBase.java
```

```
@ImportAutoConfiguration(exclude=DataSourceAutoConfiguration.c
```

```
@MockBeans(@MockBean(PagamentoRepository.class))
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment
@AutoConfigureMessageVerifier
public class PagamentosConfirmadosBase {

    @Autowired
    private NotificadorPagamentoConfirmado notificador;

    public void novoPagamentoConfirmado() {
        Pagamento pagamento = new Pagamento();
        pagamento.setId(2L);
        pagamento.setPedidoId(3L);
        notificador.notificaPagamentoConfirmado(pagamento);
    }
}
```

Confira os imports:

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.ImportAutoConfig
import org.springframework.boot.autoconfigure.jdbc.DataSourceA
import org.springframework.boot.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.boot.test.mock.mockito.MockBeans;
import org.springframework.cloud.contract.verifier.messaging.k
import org.springframework.test.context.junit4.SpringRunner;
```

Deve acontecer um erro de compilação no uso de Pagamento, NotificadorPagamentoConfirmado e PagamentoRepository na classe PagamentosConfirmadosBase.

Corrija esse erro, fazendo com que a classe Pagamento seja pública:

```
##### fj33-eats-pagamento-
```

```
service/src/main/java/br/com/caelum/eats/pagamento/Pagamento.java
```

```
public class Pagamento { // modificado  
    // código omitido ...  
}
```

Faça com a classe NotificadorPagamentoConfirmado e o método notificaPagamentoConfirmado sejam públicos:

```
##### fj33-eats-pagamento-  
service/src/main/java/br/com/caelum/eats/pagamento/NotificadorPagam  
entoConfirmado.java
```

```
// anotações omitidas ...  
public class NotificadorPagamentoConfirmado { // modificado  
    // código omitido ...  
  
    public void notificaPagamentoConfirmado(Pagamento pagamento)  
        // código omitido ...  
    }  
}
```

Torne a interface PagamentoRepository pública:

```
##### fj33-eats-pagamento-  
service/src/main/java/br/com/caelum/eats/pagamento/PagamentoReposit  
ory.java
```

```
public interface PagamentoRepository extends JpaRepository<Paç  
}
```

Ajuste os imports na classe PagamentosConfirmadosBase:

```
##### fj33-eats-pagamento-
service/src/test/java/br/com/caelum/eats/pagamento/base/PagamentosC
onfirmadosBase.java
```

```
import br.com.caelum.eats.pagamento.NotificadorPagamentoConfi
import br.com.caelum.eats.pagamento.Pagamento;
import br.com.caelum.eats.pagamento.PagamentoRepository;
```

Faça o build do Maven:

Depois da execução do build, o Spring Cloud Contract deve ter gerado a classe ConfirmadosTest:

```
##### fj33-eats-pagamento-service/target/generated-test-
sources/contracts/br/com/caelum/eats/pagamento/base/pagamentos/Co
nfirmadosTest.java
```

```
public class ConfirmadosTest extends PagamentosConfirmadosBase

    @Inject ContractVerifierMessaging contractVerifierMessaging;
    @Inject ContractVerifierObjectMapper contractVerifierObjectM

    @Test
    public void validate_deveAdicionarNovoRestaurante() throws E
        // when:
        novoPagamentoConfirmado();

        // then:
        ContractVerifierMessage response = contractVerifierMessag
        assertThat(response).isNotNull();
        assertThat(response.getHeader("contentType")).isNotNull()
        assertThat(response.getHeader("contentType").toString())
        // and:
        DocumentContext parsedJson = JsonPath.parse(contractVeri
        assertThatJson(parsedJson).field("[pedidoId]").isEqual
```

```
        assertThatJson(parsedJson).field("[ 'pagamentoId' ]").isEc  
    }  
  
}
```

O intuito dessa classe é verificar que o contrato é seguido pelo próprio publisher.

Com o sucesso dos testes, é gerado o arquivo `eats-pagamento-service-0.0.1-SNAPSHOT-stubs.jar` em `target`, contendo o contrato `deveAdicionarNovoRestaurante.groovy`. Esse JAR será usado na verificação do contrato do lado do subscriber.

Verificando o contrato no subscriber

Adicione, no `pom.xml` do serviço de nota fiscal, as dependências ao starter do Spring Cloud Contract Stub Runner e à biblioteca de suporte a testes do Spring Cloud Stream:

```
##### fj33-eats-nota-fiscal-service/pom.xml
```

```
<dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-starter-contract-stub-runner</artifactId>  
    <scope>test</scope>  
</dependency>  
<dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-stream-test-support</artifactId>  
    <scope>test</scope>  
</dependency>
```

Crie um source folder de testes no serviço de nota fiscal, clicando com o botão direto no projeto. Então, acesse o menu *New > Folder...* e defina, em *Folder name*, o caminho `src/test/java`.

Crie a classe `ProcessadorDePagamentosTest`, dentro do pacote `br.com.caelum.notafiscal` do source folder `src/test/java`.

Anote-a com as anotações de teste do Spring Boot, definindo em `@SpringBootTest` o valor `NONE` no atributo `webEnvironment`.

Adicione também a anotação `@AutoConfigureStubRunner`. No atributo `ids`, aponte para o artefato que conterá os stubs, definindo `br.com.caelum` como `groupId` e `eats-pagamento-service` como `artifactId`. No atributo `stubsMode`, use o modo `LOCAL`.

Faça com que o Spring injete uma instância de `StubTrigger`.

Injete também mocks para `GeradorDeNotaFiscal` e `PedidoRestClient` e um spy para `ProcessadorDePagamentos`, a classe que recebe as mensagens.

Defina um método `deveProcessarPagamentoConfirmado`, anotando-o com `@Test`.

No método de teste, use as instâncias de `GeradorDeNotaFiscal` e `PedidoRestClient` como stubs, registrando respostas as chamadas dos métodos `detalhaPorId` e `geraNotaPara`, respectivamente. O valor dos parâmetros deve considerar os valores definidos no contrato.

Dispare a mensagem usando o label `pagamento_confirmado` no método `trigger` do `StubTrigger`.

Verifique a chamada ao `ProcessadorDePagamentos`, usando um `ArgumentCaptor` do Mockito. Os valores dos parâmetros devem corresponder aos definidos no contrato.

```
##### fj33-eats-nota-fiscal-
service/src/test/java/br/com/caelum/notafiscal/ProcessadorDePagament
osTest.java
```

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = "br.com.caelum:eats-pagamento-s")
public class ProcessadorDePagamentosTest {

    @Autowired
    private StubTrigger stubTrigger;

    @MockBean
    private GeradorDeNotaFiscal notaFiscal;

    @MockBean
    private PedidoRestClient pedidos;

    @SpyBean
    private ProcessadorDePagamentos processadorPagamentos;

    @Test
    public void deveProcessarPagamentoConfirmado() {

        PedidoDto pedidoDto = new PedidoDto();
        Mockito.when(pedidos.detalhaPorId(3L)).thenReturn(pedidoDto);
        Mockito.when(notaFiscal.geraNotaPara(pedidoDto)).thenReturn("nota_3");

        stubTrigger.trigger("pagamento_confirmado");

        ArgumentCaptor<PagamentoConfirmado> pagamentoArg = ArgumentCaptor.forClass(PagamentoConfirmado.class);

        Mockito.verify(processadorPagamentos).processaPagamento(pagamentoArg);

        PagamentoConfirmado pagamentoConfirmado = pagamentoArg.getValue();
        Assert.assertEquals(2L, pagamentoConfirmado.getPagamentoId());
        Assert.assertEquals(3L, pagamentoConfirmado.getPedidoId());
    }

}
```

Os imports são os seguintes:

```
import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.ArgumentCaptor;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.boot.test.mock.mockito.SpyBean;
import org.springframework.cloud.contract.stubrunner.StubTrigger;
import org.springframework.cloud.contract.stubrunner.spring.AutoConfigurationStubRunner;
import org.springframework.cloud.contract.stubrunner.spring.StubRunner;
import org.springframework.test.context.junit4.SpringRunner;

import br.com.caelum.notafiscal.pedido.PedidoDto;
import br.com.caelum.notafiscal.pedido.PedidoRestClient;
```

Rode o teste. Sucesso!

Exercício: Contract Test para comunicação assíncrona

1. Abra um Terminal e vá até a branch `cap12-contrato-publisher-subscriber` do projeto do serviço de pagamentos:

```
cd ~/Desktop/fj33-eats-pagamento-service
git checkout -f cap12-contrato-publisher-subscriber
```

Então, faça o build, rode o Contract Test no próprio serviço, gere o JAR com os stubs do contrato e instale no repositório local do Maven. Para isso, basta executar o comando:

Aguarde a execução do build. As mensagens finais devem conter:

```
[INFO] Results:  
[INFO]  
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0  
[INFO]  
[INFO]  
...  
[INFO] Installing /home/<USUARIO-DO-CURSO>/Desktop/fj33-eats-pagamento-serv  
[INFO] Installing /home/<USUARIO-DO-CURSO>/Desktop/fj33-eats-pagamento-serv  
[INFO] Installing /home/<USUARIO-DO-CURSO>/Desktop/fj33-eats-pagamento-serv  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 02:45 min  
[INFO] Finished at: 2019-07-03T16:45:17-03:00  
[INFO] -----
```

Observe, pelas mensagens anteriores, que o JAR com os stubs foi instalado no diretório `.m2`, o repositório local Maven, do usuário do curso.

2. No projeto do serviço de nota fiscal, faça checkout da branch `cap12-contrato-publisher-subscriber`:

```
cd ~/Desktop/fj33-eats-nota-fiscal-service  
git checkout -f cap12-contrato-publisher-subscriber
```

Faça o refresh do projeto no Eclipse.

Clique com o botão direito na classe `ProcessadorDePagamentosTest` e, então, em *Run As... > Run Configurations....* Clique com o botão direito em *JUnit* e, a seguir, em *New Configuration*. Em *Test runner*, escolha o *JUnit 4*. Então, clique em *Run*.

Aguarde a execução dos testes. Sucesso!

Para saber mais: Consumer-driven Contract

Tests

Em uma abordagem aderente ao TDD, iniciamos uma implementação com um teste que falha, provando que a aplicação ainda não tem o comportamento sob teste.

Como aplicar essa ideia a Contract Tests?

Uma nova funcionalidade que requer a integração entre serviços, poderia ser iniciada pelo lado do Consumer, definindo um novo contrato e, em seguida, compartilhando com o time que mantém o Producer. Poderiam ser utilizados Pull Requests para essa tarefa.

O novo contrato ainda não atendido pelo lado do Producer iria falhar quando exercitado por Component Tests semelhantes aos gerados pelo Spring Cloud Contract Verifier.

Então, o time do Producer faria esses Component Tests passarem. Se tudo der certo com o build, um novo JAR com o contrato seria publicado no repositório de artefatos.

Esse novo JAR seria obtido pelo time do Consumer e poderia ser usado em Integration Tests nos pontos de integração com o Producer.

Pattern: Consumer-driven contract test

Verifique que o serviço atende às expectativas de seus clientes.

Chris Richardson no livro [Microservices Patterns](#) (RICHARDSON, 2018a)

Configuração

Externalizando as configurações

Boa parte das aplicações precisa manter uma série de configurações como credenciais de BD e Message Brokers, URLs e API keys de serviços externos, entre diversas outras.

Há relatos de aplicações implementadas em Java cujas configurações de diferentes ambientes são hard-coded no próprio código. Por exemplo, as credenciais de BD de desenvolvimento seriam usadas por padrão, enquanto credenciais de outros ambientes, como homologação e produção, ficariam comentadas. No caso de um deploy em produção, os desenvolvedores precisariam comentar as configurações de desenvolvimento, descomentando as de produção. O código então teria que ser novamente compilado e só então o entregável (JAR, WAR ou EAR) seria implantado em produção. Além disso, seria uma forte falha de segurança!

No livro [Microservices Patterns](#) (RICHARDSON, 2018a), Chris Richardson recomenda que um único build possa ser implantado em diferentes ambientes, cada um com suas configurações. Para isso, um mecanismo deve prover configurações em *runtime*.

Pattern: Externalized configuration

Forneça valores de configuração a um serviço, como credenciais do banco de dados e URLs, em tempo de execução.

Chris Richardson, no livro [Microservices Patterns](#) (RICHARDSON, 2018a)

Richardson identifica duas abordagens para a externalização de configurações:

- Push

- Pull

Push

No modelo de Push, a própria infraestrutura contém as configurações, por exemplo, por meio de variáveis de ambiente do Sistema Operacional ou arquivos de configuração.

Esse é o modelo de tecnologias de containers como Docker e orquestradores de containers como o Kubernetes.

Plataformas como o Heroku, com seus [12 fatores](#), pregam a definição de configurações em variáveis de ambiente, já que são uma maneira multiplataforma e externa à base de código.

O Spring Boot tem um mecanismo bastante flexível de [configurações externalizadas](#) que obtém configurações de diversas fontes, com regras de precedência bem definidas. Segue uma lista das fontes de configuração mais úteis, da menos importante à mais importante:

- o arquivo de configuração `application.properties` (ou `application.yml`)

Para definir uma taxa de desconto, poderíamos fazer:

- Variáveis de ambiente

O nome seria o mesmo da propriedade anterior, mas em uppercase e com underscore como separador.

No Linux, em um Terminal, a definiríamos como `export TAXA_DESCONTO=0.4`

- JVM System properties

Para defini-las, passaríamos no comando `java` com a opção `-D`:

```
java -Dtaxa.desconto=0.3 -jar app.jar
```

- Argumentos da linha de comando

```
java -jar app.jar --taxa.desconto=0.3
```

Uma limitação do push model é que reconfigurar um serviço já executado requer a reinicialização. Outro problema é o risco das configurações ficarem espalhadas na definição de centenas de serviços.

Pull

No modelo de Pull de configurações, uma instância de um serviço lê os valores de um **Configuration Server**.

Ao ser inicializado, a instância consulta o servidor para obter suas configurações. A URL do Configuration Server tem que ser mantida em um modelo de Push, com variáveis de ambiente ou arquivos de configuração.

Segundo Richardson, usar um Configuration Server traz diversos benefícios:

- Configuração centralizada: toda as configurações são armazenadas em um só lugar, fazendo com que sejam mais fáceis de gerenciar e eliminando duplicação. É possível definir valores padrão que podem ser sobreescritos em serviços específicos.
- Reconfiguração dinâmica: um serviço pode detectar os valores de configuração que foram atualizados e reconfigurar a si mesmo.
- Criptografia: algumas implementações podem manter chaves criptográficas, decriptando valores que os serviços demandarem.

Para Richardson, a principal desvantagem de manter um Configuration Server é que trata-se de mais uma peça na infraestrutura que precisa ser

mantida.

Entre os Configuration Servers disponíveis no mercado, temos:

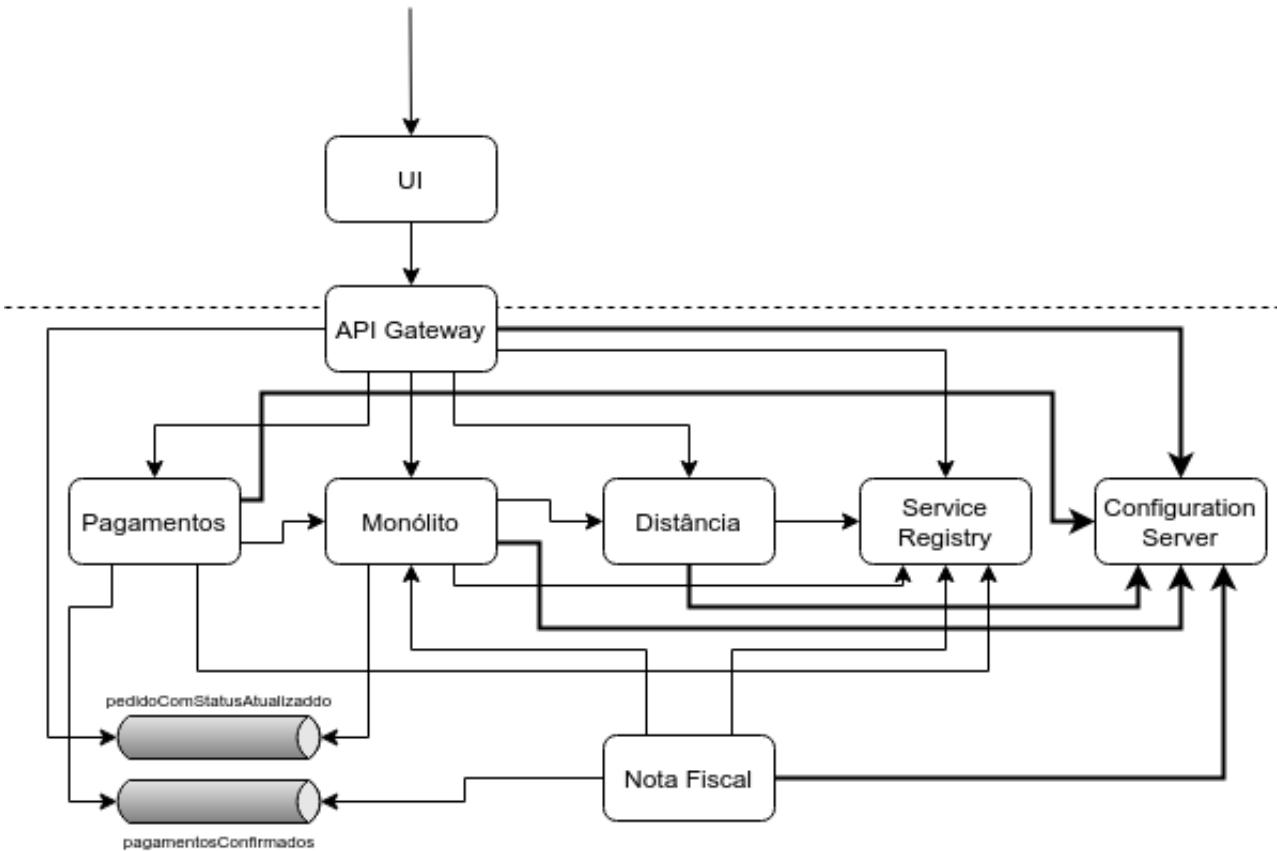
- AWS Parameter Store
- Vault, um gerenciador de credenciais da HashiCorp, que permite armazenamento de credenciais, API Keys e outros dados sensíveis
- Spring Cloud Config Server

Spring Cloud Config

O Spring Cloud Config Server pode armazenar as configurações em:

- arquivos `.properties` OU `.yml`
- repositório Git
- BD acessado por JDBC
- Redis
- AWS S3
- CredHub, um gerenciador de credenciais da Cloud Foundry
- Vault, um gerenciador de credenciais da HashiCorp

Cada serviço deve ter o projeto Spring Cloud Config Client, que obtém os valores das configurações do servidor na inicialização do serviço e as injeta no `ApplicationContext` do Spring.



Surge uma questão: a URL do Config Server realmente ser configurada em cada serviço ou podemos obtê-la do Service Registry?

A [documentação do Spring Cloud Config](#) descreve duas abordagens:

- **Config First Bootstrap:** a URL do Config Server fica em um arquivo `bootstrap.properties` de cada serviço.
- **Discovery First Bootstrap:** o Config Server é registrado no Service Registry (no nosso caso, o Eureka), de onde cada serviço obtém a URL do servidor de configuração. Para isso, a URL do Service Registry deve ser definida no `bootstrap.properties` dos serviços.

Implementando um Config Server

Pelo navegador, abra <https://start.spring.io/>. Em *Project*, mantenha *Maven Project*. Em *Language*, mantenha *Java*. Em *Spring Boot*, mantenha a versão padrão. No trecho de *Project Metadata*, defina:

- `br.com.caelum` em *Group*
- `config-server` em *Artifact*

Mantenha os valores em *More options*.

Mantenha o *Packaging* como *Jar*. Mantenha a *Java Version* em 8.

Em *Dependencies*, adicione:

- Config Server

Clique em *Generate Project*. Extraia o `config-server.zip` e copie a pasta para seu Desktop.

Adicione a anotação `@EnableConfigServer` à classe `ConfigServerApplication`:

```
##### fj33-config-
server/src/main/java/br/com/caelum/configserver/ConfigServerApplication.java
```

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

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

}
```

Adicione o import:

```
import org.springframework.cloud.config.server.EnableConfigSer
```

No arquivo `application.properties`, modifique a porta para 8888, defina `configserver` como *application name* e configure o *profile* para `native`, que obtém os arquivos de configuração de um sistema de arquivos ou do

próprio classpath.

Nossos arquivos de configuração ficarão no diretório `src/main/resources/configs`, sendo copiados para a raiz do JAR e, em *runtime*, disponível pelo classpath. Portanto, configure a propriedade `spring.cloud.config.server.native.searchLocations` para apontar para esse diretório.

```
##### fj33-config-server/src/main/resources/application.properties  
  
server.port=8888  
spring.application.name=configserver  
  
spring.profiles.active=native  
spring.cloud.config.server.native.searchLocations=classpath:/c
```

Crie o *Folder* `configs` dentro de `src/main/resources/configs`. Dentro desse diretório, defina um `application.properties` contendo propriedades comuns à maioria dos serviços, como a URL do Eureka e as credencias do RabbitMQ:

```
spring.rabbitmq.username=eats  
spring.rabbitmq.password=caelum123  
  
eureka.client.serviceUrl.defaultZone=${EUREKA_URI:http://local
```

Configurando Config Clients nos serviços

Vamos usar como exemplo a configuração do Config Client no serviço de pagamento. Os passos para os demais serviços serão semelhantes.

No `pom.xml` de `eats-pagamento-service`, adicione a dependência ao `starter` do Spring Cloud Config Client:

```
##### fj33-eats-pagamento-service/pom.xml
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

Retire do `application.properties` do serviço de pagamentos as configurações comuns que foram definidas no Config Server. Remova também o nome da aplicação:

```
##### fj33-eats-pagamento-
service/src/main/resources/application.properties
```

```
spring.application.name=pagamentos
```

```
eureka.client.serviceUrl.defaultZone=${EUREKA_URI:http://
spring.rabbitmq.username=eats
spring.rabbitmq.password=caelum123
```

Crie o arquivo `bootstrap.properties` no diretório `src/main/resources` do serviço de pagamentos. Nesse arquivo, defina o nome da aplicação e a URL do Config Server:

```
##### fj33-eats-pagamento-
service/src/main/resources/bootstrap.properties
```

```
spring.application.name=pagamentos
```

```
spring.cloud.config.uri=http://localhost:8888
```

Faça o mesmo para:

- o API Gateway
- o monólito
- o serviço de nota fiscal
- o serviço de distância

Observação: no monólito, as configurações devem ser feitas no módulo eats-application.

Git como backend do Config Server

É possível manter as configurações do Config Server em um repositório Git. Assim, podemos manter um histórico da alteração das configurações.

O Git é o backend padrão do Config Server. Por isso, não precisamos ativar nenhum profile.

Temos que configurar o endereço do repositório com a propriedade `spring.cloud.config.server.git.uri`.

Para testes, podemos apontar para um repositório local, na própria máquina do Config Server:

```
##### fj33-config-server/src/main/resources/application.properties

spring.profiles.active=native
spring.cloud.config.server.native.searchLocations=etc
spring.cloud.config.server.git.uri=file://${user.home}/Desktop
```

Podemos também usar o endereço HTTPS de um repositório Git remoto, definindo usuário e senha:

```
##### fj33-config-server/src/main/resources/application.properties

spring.cloud.config.server.git.uri=https://github.com/organiza
spring.cloud.config.server.git.username=meu-usuario
```

```
spring.cloud.config.server.git.password=minha-senha-secreta
```

Também podemos usar SSH: basta usarmos o endereço SSH do repositório e mantermos as chaves no diretório padrão (~/.ssh).

```
##### fj33-config-server/src/main/resources/application.properties
```

```
spring.cloud.config.server.git.uri=git@github.com:organizacao/
```

É possível manter as chaves SSH no próprio `application.properties` do Config Server.

Exercício: repositório Git local no Config Server

1. Faça o clone do Config Server para o seu Desktop com o seguinte comando:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-config-
```

No Eclipse, no workspace de microservices, importe o projeto `config-server`, usando o menu *File > Import > Existing Maven Projects*.

Execute a classe `ConfigServerApplication`.

2. Faça checkout da branch `cap13-repositorio-git-no-config-server` do projeto do Config Server:

```
cd ~/Desktop/fj33-config-server  
git checkout -f cap13-repositorio-git-no-config-server
```

Reinic peace o Config Server, parando e rodando novamente a classe

ConfigServerApplication.

3. No exercício, vamos usar um repositório local do Git para manter nossas configurações.

Crie um repositório Git no diretório config-repo do seu Desktop com os comandos:

```
cd ~/Desktop  
mkdir config-repo  
cd config-repo  
git init
```

Defina um arquivo application.properties no repositório config-repo, com o conteúdo:

```
spring.rabbitmq.username=eats  
spring.rabbitmq.password=caelum123  
  
eureka.client.serviceUrl.defaultZone=${EUREKA_URI:http://local
```

Obtenha o arquivo anterior na seguinte URL:

<https://gitlab.com/snippets/1896483>

```
cd ~/Desktop/config-repo  
git add .  
git commit -m "versão inicial do application.properties"
```

4. Com o Config Server no ar, acesse a seguinte URL:

<http://localhost:8888/application/default>

Você deve obter como resposta, um JSON semelhante a:

```
{  
    "name": "application",  
    "profiles": [  
        "default"  
    ],  
    "label": null,  
    "version": "04d35e5b5ae06c70abd8e08be19dba67f6b45e30",  
    "state": null,  
    "propertySources": [  
        {  
            "name": "file:///home/<USUÁRIO-D0-CURSO>/Desktop/c  
            "source": {  
                "spring.rabbitmq.username": "eats",  
                "spring.rabbitmq.password": "caelum123",  
                "eureka.client.serviceUrl.defaultZone": "${EUF  
            }  
        }  
    ]  
}
```

Faça alguma mudança no `application.properties` do config-repo e acesse novamente a URL anterior. Perceba que o Config Server precisa de um repositório Git, mas obtém o conteúdo do próprio arquivo (*working directory* nos termos do Git), mesmo sem as alterações terem sido comitadas. Isso acontece apenas quando usamos um repositório Git local, o que deve ser usado apenas para testes.

5. Reinicie todos os serviços. Teste a aplicação. Deve continuar funcionando!

Observação: as configurações só são obtidas no start up da aplicação. Se alguma configuração for modificada no Config Server, só será obtida pelos serviços quando forem reiniciados.

Movendo configurações específicas dos serviços para o Config Server

É possível criar, no repositório de configurações do Config Server, configurações específicas para cada serviço e não apenas para aquelas que são comuns a todos os serviços.

Para um backend Git, defina um arquivo `.properties` ou `.yml` cujo nome tem o mesmo valor definido em `spring.application.name`.

Para o monólito, crie um arquivo `monolito.properties` no diretório `config-repo`, que é nosso repositório Git. Passe para esse novo arquivo as configurações de BD e chaves criptográficas, removendo-as do monólito:

```
##### config-repo/monolito.properties
```

```
# DATASOURCE CONFIGS
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=

#JWT CONFIGS
jwt.secret = um-secreto-bem-secreto
jwt.expiration = 604800000
```

Remova essas configurações do `application.properties` do módulo `eats-application` do monólito:

```
##### fj33-eats-monolito-modular/eats/eats-
application/src/main/resources/application.properties
```

```
#DATASOURCE-CONFIGS
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=

# código omitido ...
```

```
#--JWT-CONFIGS  
jwt.secret==um-secreto-bem-secreto  
jwt.expiration==604800000
```

Observação: o novo arquivo deve ser comitado no config-repo, conforme a necessidade. Para repositório locais, que devem ser usados só para testes, o commit não é necessário.

Faça o mesmo para o serviço de pagamentos. Crie o arquivo pagamentos.properties no repositório de configurações, com as configurações de BD:

```
##### config-repo/pagamentos.properties
```

```
#DATASOURCE CONFIGS  
spring.datasource.url=jdbc:mysql://localhost:3307/eats_pagamento  
spring.datasource.username=pagamento  
spring.datasource.password=pagamento123
```

Remova as configurações BD do application.properties do serviço de pagamentos:

```
##### fj33-eats-pagamento-  
service/src/main/resources/application.properties
```

```
#--DATASOURCE-CONFIGS  
spring.datasource.url=jdbc:mysql://localhost:3307/eats_pagamento  
spring.datasource.username=pagamento  
spring.datasource.password=pagamento123
```

Transfira as configurações de BD do serviço de distância para um novo arquivo distancia.properties do config-repo:

```
##### config-repo/distancia.properties
```

```
spring.data.mongodb.database=eats_distancia  
spring.data.mongodb.port=27018
```

Remova as configurações do application.properties de distância:

```
##### eats-distancia-  
service/src/main/resources/application.properties
```

```
spring.data.mongodb.database=eats_distancia  
spring.data.mongodb.port=27018
```

Exercícios: Configurações específicas de cada serviço no Config Server

1. Faça o checkout da branch cap13-movendo-configuracoes-especificas-para-o-config-server no monólito e nos serviços de pagamentos e de distância:

```
cd ~/Desktop/fj33-eats-monolito-modular  
git checkout -f cap13-movendo-configuracoes-especificas-para-c
```

```
cd ~/Desktop/fj33-eats-pagamento-service  
git checkout -f cap13-movendo-configuracoes-especificas-para-c
```

```
cd ~/Desktop/fj33-eats-distancia-service  
git checkout -f cap13-movendo-configuracoes-especificas-para-c
```

Por enquanto, pare o monólito, o serviço de pagamentos e o serviço de distância.

2. Crie o arquivo monolito.properties NO config-repo com o seguinte conteúdo:

```
##### config-repo/monolito.properties

# DATASOURCE CONFIGS
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=

#JWT CONFIGS
jwt.secret = um-secreto-bem-secreto
jwt.expiration = 604800000
```

O conteúdo anterior pode ser encontrado em:

<https://gitlab.com/snippets/1896524>

Observação: não precisamos comitar os novos arquivos no repositório Git porque estamos usando um repositório local.

3. Ainda no config-repo, crie um arquivo pagamentos.properties:

```
##### config-repo/pagamentos.properties
```

```
#DATASOURCE CONFIGS
spring.datasource.url=jdbc:mysql://localhost:3307/eats_pagamento
spring.datasource.username=pagamento
spring.datasource.password=pagamento123
```

É possível obter as configurações anteriores na URL:

<https://gitlab.com/snippets/1896525>

4. Defina também, no config-repo, um arquivo distancia.properties:

```
##### config-repo/distancia.properties
```

```
spring.data.mongodb.database=eats_distancia
spring.data.mongodb.port=27018
```

O código anterior está na URL: <https://gitlab.com/snippets/1896527>

5. Faça com que os serviços sejam reiniciados, para obterem as novas configurações do Config Server. Acesse a UI e teste as funcionalidades.

Monitoramento e Observabilidade

Expondo endpoints do Spring Boot Actuator

Adicione uma dependência ao *starter* do Spring Boot Actuator:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

O módulo `eats-application` do monólito já tem essa dependência.

Essa dependência deve ser adicionada ao `pom.xml` dos projetos:

- `fj33-eats-pagamento-service`
- `fj33-eats-distancia-service`
- `fj33-eats-nota-fiscal-service`
- `fj33-api-gateway`
- `fj33-service-registry`
- `fj33-config-server`

Adicione, ao `application.properties` do `config-repo`, que será aplicado aos clientes do Config Server, uma configuração para expôr todos os *endpoints* disponíveis no Actuator:

```
##### config-repo/application.properties
```

```
management.endpoints.web.exposure.include=*
```

Observação: como estamos usando um repositório Git local, não há a necessidade de comitar as mudanças no arquivo anterior.

A configuração anterior será aplicada aos clientes do Config Server, que

são os seguintes:

- monólito
- serviço de pagamentos
- serviço de distância
- serviço de nota fiscal
- API Gateway

Para impedir que as requisições a endereços do Actuator no API Gateway acabem enviadas para o monólito, faça a configuração a seguir:

```
##### fj33-api-gateway/src/main/resources/application.properties
```

```
zuul.routes.actuator.path=/actuator/**  
zuul.routes.actuator.url=forward:/actuator
```

A configuração anterior deve ser feita antes da rota "coringa", que redirecionar tudo para o monólito.

Exponha também todos os endpoints do Actuator no `config-server` e no `service-registry`:

```
##### fj33-config-server/src/main/resources/application.properties
```

```
management.endpoints.web.exposure.include=*
```

e

```
##### fj33-service-registry/src/main/resources/application.properties
```

```
management.endpoints.web.exposure.include=*
```

Reinic peace os serviços e explore os endpoints do Actuator.

A seguinte URL contém links para os demais endpoints:

<http://localhost:{porta}/actuator>

É possível ver, de maneira detalhada, os valores das configurações:

<http://localhost:{porta}/actuator/configprops>

e

<http://localhost:{porta}/actuator/env>

Podemos verificar (e até modificar) os níveis de log:

<http://localhost:{porta}/actuator/loggers>

Com a URL a seguir, podemos ver uma lista de métricas disponíveis:

<http://localhost:{porta}/actuator/metrics>

Por exemplo, podemos obter o *uptime* da JVM com a URL:

<http://localhost:{porta}/actuator/metrics/process.uptime>

Há uma lista dos `@RequestMapping` da aplicação:

<http://localhost:{porta}/actuator/mappings>

Podemos obter informações sobre os bindings, exchanges e channels do Spring Cloud Stream com as URLs:

<http://localhost:{porta}/actuator/bindings>

e

<http://localhost:{porta}/actuator/channels>

Observação: troque `{porta}` pela porta de algum serviço.

Há ainda endpoints específicos para o serviço que estamos acessando.

Por exemplo, para o API Gateway temos com as rotas e *filters*:

<http://localhost:9999/actuator/routes>

e

<http://localhost:9999/actuator/filters>

Exercício: Health Check API com Spring Boot Actuator

1. Faça checkout da branch `cap14-health-check-api-com-spring-boot-actuator` dos seguintes projetos:

```
cd ~/Desktop/fj33-eats-pagamento-service  
git checkout -f cap14-health-check-api-com-spring-boot-actuator
```

```
cd ~/Desktop/fj33-eats-distancia-service  
git checkout -f cap14-health-check-api-com-spring-boot-actuator
```

```
cd ~/Desktop/fj33-eats-nota-fiscal-service  
git checkout -f cap14-health-check-api-com-spring-boot-actuator
```

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap14-health-check-api-com-spring-boot-actuator
```

```
cd ~/Desktop/fj33-service-registry  
git checkout -f cap14-health-check-api-com-spring-boot-actuator
```

```
cd ~/Desktop/fj33-config-server  
git checkout -f cap14-health-check-api-com-spring-boot-actuator
```

Dê refresh nos projetos no Eclipse e os reinicie.

2. Explore os endpoints do Spring Boot Actuator, baseando-se nos exemplos da seção anterior.

Por exemplo, teste a seguinte URL para visualizar um *stream* (fluxo de

dados) com as informações dos circuit breakers do API Gateway:

<http://localhost:9999/actuator/hystrix.stream>

Também é possível explorar os links retornados pela seguinte URL, trocando {porta} pelas portas dos serviços:

<http://localhost:{porta}/actuator>

Configurando o Hystrix Dashboard

Pelo navegador, abra <https://start.spring.io/>. Em *Project*, mantenha *Maven Project*. Em *Language*, mantenha Java. Em *Spring Boot*, mantenha a versão padrão. No trecho de *Project Metadata*, defina:

- br.com.caelum em *Group*
- hystrix-dashboard em *Artifact*

Mantenha os valores em *More options*.

Mantenha o *Packaging* como *Jar*. Mantenha a *Java Version* em 8.

Em *Dependencies*, adicione:

- Hystrix Dashboard

Clique em *Generate Project*.

Extraia o *hystrix-dashboard.zip* e copie a pasta para seu Desktop.

Adicione a anotação `@EnableHystrixDashboard` à classe *HystrixDashboardApplication*:

```
##### fj33-hystrix-
dashboard/src/main/java/br/com/caelum/hystrixdashboard/HystrixDashb
oardApplication.java
```

```
@EnableHystrixDashboard
```

```
@SpringBootApplication
public class HystrixDashboardApplication {

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

}
```

Adicione o import:

```
import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystri
```

No arquivo application.properties, modifique a porta para 7777:

```
##### fj33-hystrix-
dashboard/src/main/resources/application.properties
```

Exercício: Visualizando circuit breakers com o Hystrix Dashboard

1. Abra um Terminal e clone o projeto `fj33-hystrix-dashboard` para o seu Desktop:

```
cd ~/Desktop
git clone https://gitlab.com/aovs/projetos-cursos/fj33-hystrix-
```

No Eclipse, no workspace de microservices, importe o projeto `hystrix-dashboard`, usando o menu *File > Import > Existing Maven Projects*.

Execute a classe `HystrixDashboardApplication`.

2. Acesse o Hystrix Dashboard, pelo navegador, com a seguinte URL:

<http://localhost:7777/hystrix>

Coloque, na URL, o endpoint de Hystrix Stream Actuator do API Gateway:

<http://localhost:9999/actuator/hystrix.stream>

Clique em *Monitor Stream*.

Em outra aba, acesse URLs do API Gateway como as que seguem:

- <http://localhost:9999/restaurantes/1>, que exibirá o circuit breaker do monolito
- <http://localhost:9999/pagamentos/1>, que exibirá o circuit breaker do serviço de pagamentos
- <http://localhost:9999/distancia/restaurantes/mais-proximos/71503510>, que exibirá o circuit breaker do serviço de distância
- <http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1>, que exibirá os circuit breakers relacionados a composição de chamadas feita no API Gateway

Veja as informações dos circuit breakers do API Gateway no Hystrix Dashboard.

Agregando dados dos circuit-breakers com Turbine

Pelo navegador, abra <https://start.spring.io/>. Em *Project*, mantenha *Maven Project*. Em *Language*, mantenha Java. Em *Spring Boot*, mantenha a versão padrão. No trecho de *Project Metadata*, defina:

- br.com.caelum em *Group*
- turbine em *Artifact*

Mantenha os valores em *More options*.

Mantenha o *Packaging* como Jar. Mantenha a *Java Version* em 8.

Em *Dependencies*, adicione:

- Turbine
- Eureka Client
- Config Client

Clique em *Generate Project*.

Extraia o `turbine.zip` e copie a pasta para seu Desktop.

Adicione as anotações `@EnableDiscoveryClient` e `@EnableTurbine` à classe `TurbineApplication`:

```
##### fj33-
turbine/src/main/java/br/com/caelum/turbine/TurbineApplication.java
```

```
@EnableTurbine
@EnableDiscoveryClient
@SpringBootApplication
public class TurbineApplication {

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

}
```

Não esqueça de ajustar os imports:

```
import org.springframework.cloud.client.discovery.EnableDiscov
import org.springframework.cloud.netflix.turbine.EnableTurbine
```

No arquivo `application.properties`, modifique a porta para 7776.

Adicione configurações que aponta para os nomes das aplicações e para

o cluster default:

```
##### fj33-turbine/src/main/resources/application.properties
```

```
server.port=7776
```

```
turbine.appConfig=apigateway
turbine.clusterNameExpression='default'
```

Defina um arquivo `bootstrap.properties` no diretório de `resources`, configurando o endereço do Config Server:

```
spring.application.name=turbine
spring.cloud.config.uri=http://localhost:8888
```

Agregando baseado em eventos com Turbine Stream

No `pom.xml` do projeto `turbine`, troque a dependência ao starter do Turbine pela do Turbine Stream. Remova a dependência ao starter do Eureka Client. Além disso, adicione o binder do Spring Cloud Stream ao RabbitMQ:

```
##### fj33-turbine/pom.xml
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-turbine</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
```

```
<artifactId>spring-cloud-starter-netflix-turbine-stream</art
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

O status de cada circuit breaker será obtido por meio de eventos no Exchange `springCloudHystrixStream` do RabbitMQ.

Por isso, remova as configurações de aplicações do `application.properties`:

```
##### fj33-turbine/src/main/resources/application.properties
```

```
turbine-appConfig=apigateway
turbine-clusterNameExpression='default'-
```

Remove a anotação `@EnableDiscoveryClient` e troque a anotação `@EnableTurbine` por `@EnableTurbineStream` na classe `TurbineApplication`:

```
##### fj33-
turbine/src/main/java/br/com/caelum/turbine/TurbineApplication.java
```

```
@EnableTurbineStream
@EnableTurbine
@EnableDiscoveryClient
@SpringBootApplication
public class TurbineApplication {

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

```
}
```

Ajuste os imports da seguinte maneira:

```
import org.springframework.cloud.client.discovery.E  
import org.springframework.cloud.netflix.turbine.E  
import org.springframework.cloud.netflix.turbine.stream.Enable
```

Adicione a dependência ao Hystrix Stream no `pom.xml` do API Gateway:

```
##### fj33-api-gateway/pom.xml
```

```
<dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-netflix-hystrix-stream</artifactId>  
</dependency>
```

Ajuste o *destination* do *channel* `hystrixStreamOutput`, no `application.properties` do API Gateway, por meio da propriedade:

```
##### fj33-api-gateway/src/main/resources/application.properties
```

```
spring.cloud.stream.bindings.hystrixStreamOutput.destination=s
```

Exercício: Agregando Circuit Breakers com TURBINE STREAM

1. Faça o clone do projeto `fj33-turbine` para o seu Desktop:

```
cd ~/Desktop/fj33-turbine  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-turbine
```

No Eclipse, no workspace de microservices, importe o projeto `turbine`, usando o menu *File > Import > Existing Maven Projects*.

Execute a classe `TurbineApplication`.

2. Faça o checkout da branch `cap14-turbine-stream` do projeto `fj33-api-gateway`:

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap14-turbine-stream
```

Dê o refresh do API Gateway no Eclipse e o reinicie.

3. Acesse o Turbine pela URL a seguir:

<http://localhost:7776/turbine.stream>

Em outra janela do navegador, faça algumas chamadas ao API Gateway, como as do exercício anterior.

Observe, na página do Turbine, um fluxo de dados parecido com:

```
: ping  
data: {"reportingHostsLast10Seconds":0,"name":"meta","type":"meta","timestamp":1537500000000,"version":1}  
  
: ping  
data: {"reportingHostsLast10Seconds":0,"name":"meta","type":"meta","timestamp":1537500000000,"version":1}  
  
: ping  
data: {"rollingCountFallbackSuccess":0,"rollingCountFallbackFailure":0,"processes":0,"timestamp":1537500000000,"version":1}
```

Garanta que o Hystrix Dashboard esteja rodando e vá a URL:

<http://localhost:7777/hystrix>

Na URL, use o endereço da stream do Turbine:

<http://localhost:7776/turbine.stream>

Faça algumas chamadas ao API Gateway. Veja os status dos circuit breakers.

Pare os serviços e o monólito e faça mais chamadas ao API Gateway. Veja o resultado no Hystrix Dashboard.

Exercício: configurando o Zipkin no Docker Compose

1. Para provisionar uma instância do Zipkin, adicione as seguintes configurações ao `docker-compose.yml` do seu Desktop:

```
##### docker-compose.yml
```

```
zipkin:  
  image: openzipkin/zipkin  
  ports:  
    - "9410:9410"  
    - "9411:9411"  
  depends_on:  
    - rabbitmq  
  environment:  
    RABBIT_URI: "amqp://eats:caelum123@rabbitmq:5672"
```

O `docker-compose.yml` completo, com a configuração do Zipkin, pode ser encontrado em: <https://gitlab.com/snippets/1888247>

2. Execute o servidor do Zipkin pelo Docker Compose com o comando:
3. Acesse a UI Web do Zipkin pelo navegador através da URL:

<http://localhost:9411/zipkin/>

Enviando informações para o Zipkin com Spring Cloud Sleuth

Adicione uma dependência ao starter do Spring Cloud Zipkin no `pom.xml` do API Gateway:

```
##### fj33-api-gateway/pom.xml
```

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

Faça o mesmo no `pom.xml` do:

- módulo `eats-application` do monólito
- serviço de pagamentos
- serviço de distância
- serviço de nota fiscal

Exercício: Distributed Tracing com Spring Cloud Sleuth e Zipkin

1. Vá até a branch `cap14-spring-cloud-sleuth` dos projetos do API Gateway, do Monólito Modular e dos serviços de distância, pagamentos e notas fiscais:

```
cd ~/Desktop/fj33-api-gateway
git checkout -f cap14-spring-cloud-sleuth
```

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap14-spring-cloud-sleuth
```

```
cd ~/Desktop/fj33-eats-pagamento-service
git checkout -f cap14-spring-cloud-sleuth
```

```
cd ~/Desktop/fj33-eats-distancia-service
git checkout -f cap14-spring-cloud-sleuth
```

```
cd ~/Desktop/fj33-eats-nota-fiscal-service  
git checkout -f cap14-spring-cloud-sleuth
```

Faça refresh no Eclipse e reinicie os projetos.

2. Por padrão, o Spring Cloud Sleuth faz rastreamento por amostragem de 10% das chamadas. É um bom valor, mas inviável pelo pouco volume de nossos requests.

Por isso, altere a porcentagem de amostragem para 100%, modificando o arquivo `application.properties` do `config-repo`:

```
##### config-repo/application.properties
```

```
spring.sleuth.sampler.probability=1.0
```

3. Reinicie os serviços que foram modificados no passo anterior.
Garanta que a UI esteja no ar.

Faça um novo pedido, até a confirmação do pagamento. Faça o login como dono do restaurante e aprove o pedido. Edite o tipo de cozinha e/ou CEP de um restaurante.

Vá até a interface Web do Zipkin acessando: <http://localhost:9411/zipkin/>

Selecione um serviço em *Service Name*. Então, clique em *Find traces* e veja os rastreamentos. Clique para ver os detalhes.

Na aba *Dependencies*, veja um gráfico com as dependências entre os serviços baseadas no uso real (e não apenas em diagramas arquiteturais).

Spring Boot Admin

Pelo navegador, abra <https://start.spring.io/>. Em *Project*, mantenha

Maven Project. Em *Language*, mantenha Java. Em *Spring Boot*, mantenha a versão padrão. No trecho de *Project Metadata*, defina:

- br.com.caelum em *Group*
- admin-server em *Artifact*

Mantenha os valores em *More options*.

Mantenha o *Packaging* como *Jar*. Mantenha a *Java Version* em 8.

Em *Dependencies*, adicione:

- Spring Boot Admin (Server)
- Config Client
- Eureka Discovery Client

Clique em *Generate Project*.

Extraia o `admin-server.zip` e copie a pasta para seu Desktop.

Adicione a anotação `@EnableAdminServer` à classe

`AdminServerApplication`:

```
##### fj33-admin-
server/src/main/java;br/com/caelum/adminserver/AdminServerApplication
.java
```

```
@EnableAdminServer
@SpringBootApplication
public class AdminServerApplication {

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

}
```

Adicione o import:

```
import de.codecentric.boot.admin.server.config.EnableAdminServer
```

No arquivo application.properties, modifique a porta para 6666:

```
##### fj33-admin-server/src/main/resources/application.properties
```

Crie um arquivo bootstrap.properties no diretório src/main/resources do Admin Server, definindo o nome da aplicação e o endereço do Config Server:

```
spring.application.name=adminserver  
spring.cloud.config.uri=http://localhost:8888
```

Exercício: Visualizando os microservices com Spring Boot Admin

1. Faça clone do projeto fj33-admin-server:

```
git clone https://gitlab.com/aovs/projetos-cursos/fj33-admin-s
```

No Eclipse, no workspace de microservices, importe o projeto fj33-admin-server, usando o menu *File > Import > Existing Maven Projects*.

Execute a classe AdminServerApplication.

2. Pelo navegador, acesse a URL:

<http://localhost:6666>

Veja informações sobre as aplicações e instâncias.

Em *Wallboard*, há uma visualização interessante do status dos serviços.

Segurança

Extraindo um serviço Administrativo do monólito

Primeiramente, vamos extrair o módulo eats-administrativo do monólito para um serviço eats-administrativo-service.

Para isso, criamos um novo projeto Spring Boot com as seguintes dependências:

- Spring Boot DevTools
- Spring Boot Actuator
- Spring Data JPA
- Spring Web Starter
- Config Client
- Eureka Discovery Client
- Zipkin Client

Então, movemos as seguintes classes do módulo Administrativo do monólito para o novo serviço:

- FormaDePagamento
- FormaDePagamentoController
- FormaDePagamentoRepository
- TipoDeCozinha
- TipoDeCozinhaController
- TipoDeCozinhaRepository

O serviço administrativo deve apontar para o Config Server, definindo um `bootstrap.properties` com `administrativo` como *application name*. No arquivo `administrativo.properties` do config-repo, definiremos as configurações de data source.

Inicialmente, o serviço administrativo pode apontar para o mesmo BD do monólito. Aos poucos, deve ser feita a migração das tabelas `forma_de_pagamento` e `tipo_de_cozinha` para um BD próprio.

No `application.properties`, deve ser definida 8084 na porta a ser utilizada.

Então, o módulo `eats-administrativo` do monólito pode ser removido, assim como suas autorizações no módulo `eats-segurança`.

Remova a dependência a `eats-administrativo` do `pom.xml` do módulo `eats-application` do monólito:

```
##### fj33-eats-monolito-modular/eats/eats-application/pom.xml
```

```
<dependency>
  <groupId>br.com.caetum</groupId>
  <artifactId>eats-administrativo</artifactId>
  <version>${project.version}</version>
</dependency>
```

Faço o mesmo nos arquivos `pom.xml` dos módulos `eats-restaurante` e `eats-pedido` do monólito.

No projeto pai dos módulos, o projeto `eats`, remova o módulo `eats-administrativo` do `pom.xml`:

```
##### fj33-eats-monolito-modular/eats/pom.xml
```

```
<modules>
  <module>eats-administrativo</module>
  <module>eats-restaurante</module>
  <module>eats-pedido</module>
  <module>eats-segurança</module>
  <module>eats-application</module>
</modules>
```

Apague o módulo `eats-administrativo` do monólito. Pelo Eclipse, tecle *Delete* em cima do módulo, selecione a opção *Delete project contents on*

disk (cannot be undone) e clique em *OK*.

Remova, da classe `SecurityConfig` do módulo `eats-segurança` do monólito, as configurações de autorização dos endpoints que foram movidos:

```
##### fj33-eats-monolito-modular/eats/eats-
segurança/src/main/java/br/com/caelum/eats/SecurityConfig.java

class SecurityConfig extends WebSecurityConfigurerAdapter {

    // código omitido ...

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/restaurantes/**", "/pedidos/**").permitAll()
            .antMatchers("/actuator/**").permitAll()
            .antMatchers("/admin/**").hasRole("ROLE_ADMIN")
        // código omitido ...
    }

}
```

Também é necessário alterar as referências às classes `TipoDeCozinha` e `FormaDePagamento` no `DistanciaRestClientWiremockTest` do módulo `eats-application`.

Já no módulo de restaurantes do monólito, é preciso alterar referências às classes migradas para o serviço Administrativo para apenas utilizarem os ids dos agregados `TipoDeCozinha` e `FormaDePagamento`. Isso afeta diversas classes do módulo `eats-restaurante`:

- Restaurante
- RestauranteController
- RestauranteDto

- RestauranteFormaDePagamento
- RestauranteFormaDePagamentoController
- RestauranteFormaDePagamentoRepository
- RestauranteParaServicoDeDistancia
- RestauranteRepository
- RestauranteService

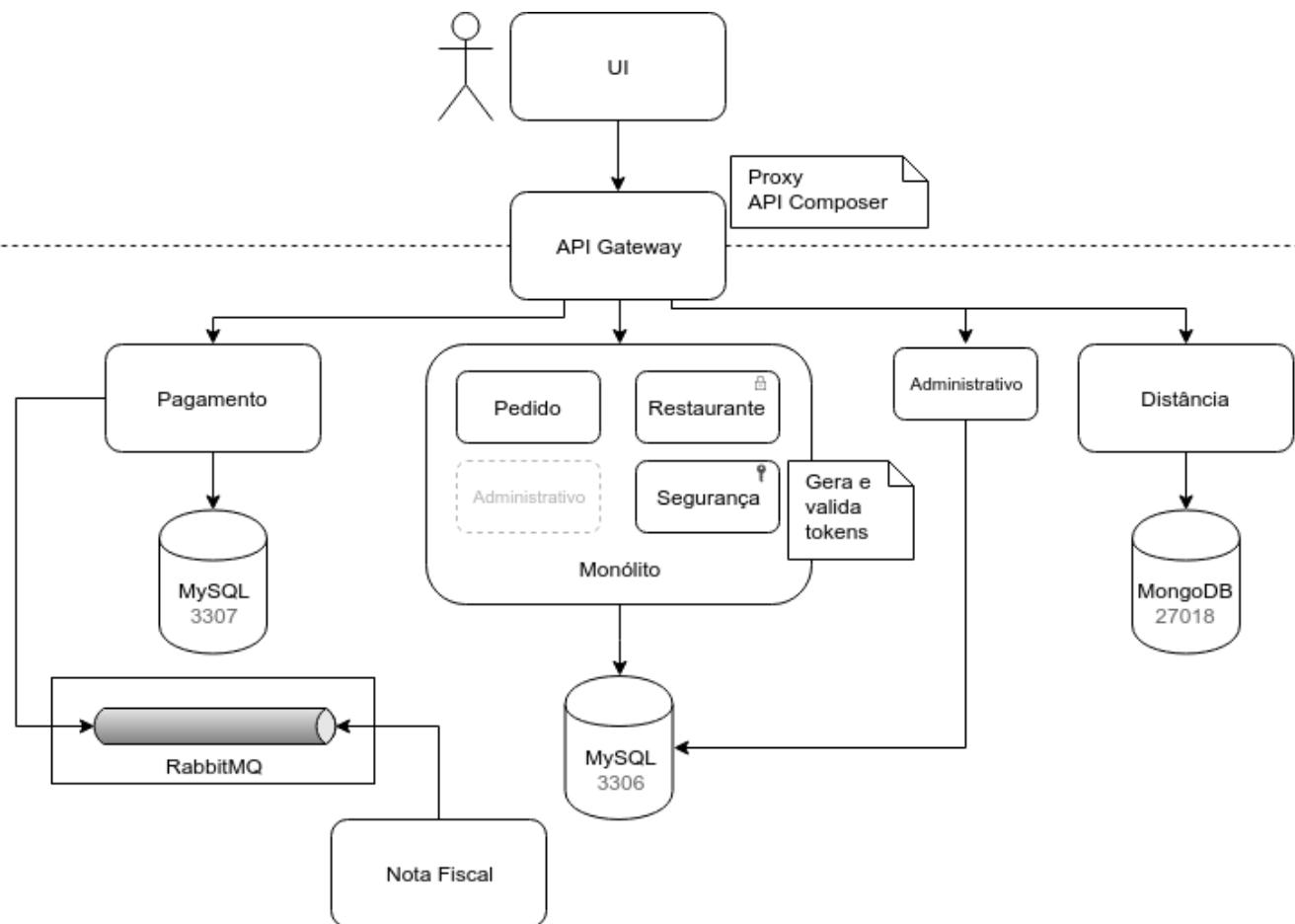
A UI também será afetada. Uma das mudanças é que chamadas relativas a tipos de cozinha e formas de pagamento devem ser direcionadas para o serviço Administrativo. Esse serviço registra-se no Eureka Server com o nome `administrativo`, o seu application name. O API Gateway faz o roteamento dinâmico baseado nas instâncias disponíveis no Service Registry. Por isso, podemos trocar chamadas como a seguinte para utilizarem o prefixo `administrativo`:

```
##### fj33-eats-ui/src/app/services/tipo-de-cozinha.service.ts
```

```
export class TipoDeCozinhaService {  
  
  private API === environment.baseUrl;  
  private API = environment.baseUrl + '/administrativo';  
  
  // código omitido ...  
  
}
```

O mesmo deve ser feito para a classe `FormaDePagamentoService`.

As diversas mudanças no módulo de restaurantes do monólito também afetam a UI.



Exercício: um serviço Administrativo

- Crie um arquivo `administrativo.properties` no `config-repo`, definindo um data source que aponta para o mesmo BD do monólito:

```
##### config-repo/administrativo.properties
```

```
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=
```

- Clone o projeto `fj33-eats-administrativo-service` para o seu Desktop:

```
cd ~/Desktop
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-ac...
```

No Eclipse, no workspace de microservices, importe o projeto `fj33-eats-administrativo-service`, usando o menu *File > Import > Existing Maven Projects*.

Com o Service Registry e o Config Server no ar, suba o serviço Administrativo executando a classe `EatsAdministrativoServiceApplication`.

3. Faça checkout da branch `cap15-extrai-administrativo-service` do monólito modular e da UI:

```
cd ~/Desktop/fj33-eats-monolito-modular  
git checkout -f cap15-extrai-administrativo-service
```

```
cd ~/Desktop/fj33-eats-ui  
git checkout -f cap15-extrai-administrativo-service
```

Faça refresh do monólito modular no Eclipse.

Suba os serviços e o front-end. Teste o Caelum Eats. Deve funcionar!

Autenticação e Autorização

Grande parte das aplicações tem diferentes perfis de usuário, que têm permissão de acesso a diferentes funcionalidades. Isso é o que chamamos de **Autorização**.

No caso do Caelum Eats, qualquer usuário pode acessar fazer pedidos e acompanhá-los. Porém, a administração do sistema, que permite cadastrar tipos de cozinha, formas de pagamento e aprovar restaurantes, só é acessível pelo perfil de administrador. Já os dados de um restaurante e o gerenciamento dos pedidos pendentes só são acessíveis pelo dono de cada restaurante.

Um usuário precisa identificar-se, ou seja, dizer quem está acessando a aplicação. Isso é o que chamamos de **Autenticação**.

Uma vez que o usuário está autenticado e sua identidade é conhecida, é possível que a aplicação reforce as permissões de acesso.

Existem algumas maneiras mais comuns de um sistema confirmar a identidade de um usuário:

- algo que o usuário sabe, um segredo, como uma senha
- algo que o usuário tem, como um token físico ou por uma app mobile
- algo que o usuário é, como biometria das digitais, íris ou reconhecimento facial

Two-factor authentication (2FA), ou autenticação de dois fatores, é a associação de duas formas de autenticação para minimizar as chances de alguém mal intencionado identificar-se como outro usuário, no caso de apoderar-se de um dos fatores de autenticação.

Sessões e escalabilidade

Após a autenticação, uma aplicação Web tradicional guarda a identidade do usuário em uma **sessão**, que comumente é armazenada em memória, mas pode ser armazenada em disco ou em um BD.

O cliente da aplicação, em geral um navegador, deve armazenar um id da sessão. Em toda requisição, o cliente passa esse id para identificar o usuário.

O que acontece quando há um aumento drástico no número de usuários em momento de pico de uso, como na Black Friday?

Se a aplicação suportar esse aumento na carga, podemos dizer que possui a característica arquitetural da **Escalabilidade**. Quando a escalabilidade é atingida aumentando o número de máquinas, dizemos que é a escalabilidade horizontal.

Mas, se escalarmos horizontalmente a aplicação, onde fica armazenada a sessão se temos mais de uma máquina como servidor Web? Uma

estratégia são as *sticky sessions*, em que cada usuário tem sua sessão em uma máquina específica.

Mas quando alguma máquina falhar, o usuário seria deslogado e não teria mais acesso às funcionalidades. Para que a experiência do usuário seja transparente, de maneira que ele não perceba a falha em uma máquina, há a técnica da **replicação de sessão**, em que cada servidor compartilha, pela rede, suas sessões com outros servidores. Isso traz uma sobrecarga de processamento, armazenamento e tráfego na rede.

REST, stateless sessions e self-contained tokens

Em sua tese de doutorado *Architectural Styles and the Design of Network-based Software Architectures*, Roy Fielding descreve o estilo arquitetural da Web e o chama de **Representational State Transfer (REST)**. Uma das características do REST é que a comunicação deve ser **Stateless**: toda informação deve estar contida na requisição do cliente ao servidor, sem a necessidade de nenhum contexto armazenado no servidor.

Manter sessões nos servidores é manter estado. Portanto, podemos dizer que utilizar sessões não é RESTful porque não segue a característica do REST de ser stateless.

Mas então como fazer um mecanismo de autenticação que seja stateless e, por consequência, mais próximo do REST?

Usando tokens! Há tokens opacos, que são apenas um texto randômico e que não carregam nenhuma informação. Porém, há os **self-contained tokens**, que contém informações sobre o usuário e/ou sobre o sistema cliente. Cada requisição teria um self-contained token em seu cabeçalho, com todas as informações necessárias para a aplicação. Assim, tiramos a necessidade de armazenamento da sessão no lado do servidor.

A grande questão é como ter um token que contém informações e, ao mesmo tempo, garantir sua integridade, confirmando que os dados do

token não foram manipulados?

JWT e JWS

JWT (JSON Web Token) é um formato de token compacto e self-contained que serve propagar informações de identidade, permissões de um usuário em uma aplicação de maneira segura. Foi definido na RFC 7519 da Internet Engineering Task Force (IETF), em Maio de 2015.

O Working Group da IETF chamado Javascript Object Signing and Encryption (JOSE), definiu duas outras RFCs relacionadas:

- JSON Web Signature (JWS), definido na RFC 7515, que representa em JSON conteúdo assinado digitalmente
- JSON Web Encryption (JWE), definido na RFC 7516, que representa em JSON conteúdo criptografado

Para garantir a integridade dos dados de um token, é suficiente usarmos o JWS.

Um JWS consiste de três partes, separadas por .:

```
BASE64URL(UTF8(Cabeçalho)) || '.' ||  
BASE64URL(Payload) || '.' ||  
BASE64URL(Assinatura JWS)
```

Todas as partes do JWS são codificadas em *Base64 URL encoded*. Base64 é uma representação em texto de dados binários. URL encoded significa que caracteres especiais são codificados com %, da mesma maneira como são passados via parâmetros de URLs.

Um exemplo de um JWS usado no Caelum Eats seria o seguinte:

```
eyJhbGciOiJIUzI1NiJ9.  
eyJpc3MiOiJDYWVsdW0gRWF0cyIsInN1YiI6IjIiLCJyb2xlcyI6WyJQQVJDRU1STyJdLCJ1c2V  
GOWiEeJMP9t0tv2lQpNiDU211WKL6h5Z60kNcA-f4EY
```

Os trechos anteriores podem ser descodificados de Base64 para texto normal usando um site como: <http://www.base64url.com/>

O primeiro trecho, eyJhbGciOiJIUzI1NiJ9, é o cabeçalho do JWS.

Quando descodificado, é:

O valor de `alg` indica que foi utilizado o algoritmo HMAC (hash-based message authentication code) com SHA-256 como função de hash. Nesse algoritmo, há uma chave secreta (um texto) simétrica, que deve ser conhecida tanto pela parte que cria o token como pela parte que o validará. Se essa chave secreta for descoberta por um agente mal intencionado, pode ser usada para gerar tokens válidos deliberadamente.

O segundo trecho,

eyJpc3MiOiJDYWVsdW0gRWF0cyIsInN1YiI6IjIiLCJyb2xlcycI6WyJQQVJDRU1STyJdLCJ1c2VybmFtZSI6ImxvbmdmdSIsImhdCI6MTU2NjQ5ODA5MSwiZXhwIjoxNTY3MTAyODkxfQ, contém os dados (payload) do JWS:

```
{  
  "iss": "Caelum Eats",  
  "sub": "2",  
  "roles": [  
    "PARCEIRO"  
,  
  "username": "longfu",  
  "iat": 1566498091,  
  "exp": 1567102891  
}
```

O valor de `iss` é o issuer, a aplicação que gerou o token. O valor de `sub` é o subject, que contém informações do usuário. Os valores de `iat` e `exp`, são as datas de geração e expiração do token, respectivamente. Os demais valores são *claims* customizadas, que declaram informações adicionais do usuário.

O terceiro trecho, `G0wiEEeJMP9t0tv21QpNiDU211WKL6h5Z60kNcA-f4EY`, é a assinatura do JWS e não pode ser decodificada para um texto. O que importam são os bytes.

No site <https://jwt.io/> conseguimos obter o algoritmo utilizado, os dados do payload e até validar um JWT.

Se soubermos a chave secreta, podemos verificar se a assinatura bate com o payload do JWS. Se bater, o token é válido. Dessa maneira, conseguimos garantir que não houve manipulação dos dados e, portanto, sua integridade.

Um detalhe importante é que um JWS não garante a confidencialidade dos dados. Se houver algum software bisbilhotando os dados trafegados na rede, o payload do JWS pode ser lido, já que é apenas codificado em Base64 URL encoded. A confidencialidade pode ser reforçada por meio de TLS no canal de comunicação ou por meio de JWE.

Uma grande desvantagem de um JWT é que o token é irrevogável antes de sua expiração. Isso implica que, enquanto o token não estiver expirado será válido. Por isso, implementar um mecanismo de logout pelo usuário passa a ser complicado. Poderíamos trabalhar com intervalos pequenos de expiração, mas isso afetaria a experiência do usuário, já que frequentemente a expiração levaria o usuário a efetuar novo login. Uma maneira comum de implementar logout é ter um cache com JWT invalidados. Porém, isso nos leva novamente a uma solução *stateful*.

Stateless Sessions no Caelum Eats

Até o momento, um login de um dono de restaurante ou do administrador do sistema dispara a execução do `AuthenticationController` do módulo `eats-seguranca` do monólito. No método `authenticate`, é gerado e retornado um token JWS.

O token JWS é armazenado em um `localStorage` no front-end. Há um

interceptor do Angular que, antes de cada requisição AJAX, adiciona o cabeçalho `Authorization: Bearer` com o valor do token armazenado.

No back-end, a classe `JwtAuthenticationFilter` é executada a cada requisição e o token JWS é extraído dos cabeçalhos HTTP e validado. Caso seja válido, é recuperado o `sub` (Subject) e obtido o usuário do BD com seus ROLES (`ADMIN` ou `PARCEIRO`), setando um `Authentication` no contexto de segurança:

```
##### fj33-eats-monolito-modular/eats/eats-
seguranca/src/main/java/br/com/caelum/eats/seguranca/JwtAuthenticati
onFilter.java
```

```
@Component
@AllArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFil

    private JwtTokenManager tokenManager;
    private UserService usersService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
        throws ServletException, IOException {
        String jwt = getTokenFromRequest(request);
        if (tokenManager.isValid(jwt)) {
            Long userId = tokenManager.getUserIdFromToken(jwt);
            UserDetails userDetails = usersService.loadUserById(user
                UsernamePasswordAuthenticationToken authentication = new
                    null, userDetails.getAuthorities());
            SecurityContextHolder.getContext().setAuthentication(auth
        }

        chain.doFilter(request, response);
    }

    private String getTokenFromRequest(HttpServletRequest reques
        // código omitido ...
    }
```

```
}
```

A geração, validação e recuperação dos dados do token é feita por meio da classe `JwtTokenManager`, que utiliza a biblioteca `jjwt`:

```
##### fj33-eats-monolito-modular/eats/eats-
seguranca/src/main/java/br/com/caelum/eats/seguranca/JwtTokenManag
er.java
```

```
@Component
class JwtTokenManager {

    private String secret;
    private long expirationInMillis;

    public JwtTokenManager(          @Value("${jwt.secret}") String
                                    @Value("${jwt.expiration}") long expirationInMil
        this.secret = secret;
        this.expirationInMillis = expirationInMillis;
    }

    public String generateToken(User user) {
        final Date now = new Date();
        final Date expiration = new Date(now.getTime() + this.exp
        return Jwts.builder()
            .setIssuer("Caelum Eats")
            .setSubject(Long.toString(user.getId()))
            .claim("username", user.getName())
            .claim("roles", user.getRoles())
            .setIssuedAt(now)
            .setExpiration(expiration)
            .signWith(SignatureAlgorithm.HS256, this.secret)
            .compact();
    }

    public boolean isValid(String jwt) {
        try {
```

```
Jwts
    .parser()
    .setSigningKey(this.secret)
    .parseClaimsJws(jwt);
    return true;
} catch (JwtException | IllegalArgumentException e) {
    return false;
}
}

public Long getUserIdFromToken(String jwt) {
    Claims claims = Jwts.parser().setSigningKey(this.secret).r
    return Long.parseLong(claims.getSubject());
}

}
```

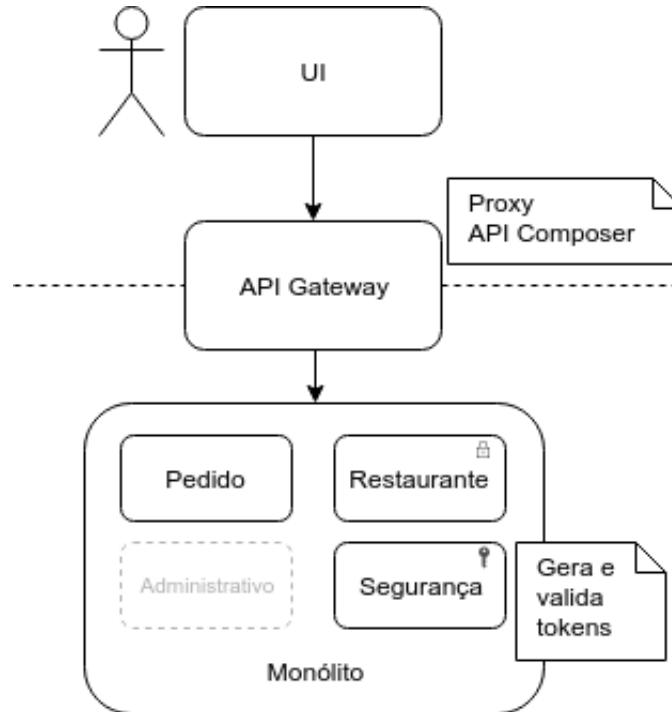
As configurações de autorização estão definidas na classe `SecurityConfig` do módulo `eats-segurança` do monólito.

Antes da extração do serviço administrativo, para as URLs que começavam com `/admin`, o ROLE do usuário deveria ser `ADMIN` e teria acesso a tudo relativo à administração da aplicação. Esse tipo de autorização, em que um determinado ROLE tem acesso a qualquer endpoint relacionado é o que chamamos de *role-based authorization*.

Porém, ao extraímos o serviço administrativo, perdemos a autorização feita no módulo de segurança do monólito. Ainda não implementamos autorização no novo serviço.

No caso da URL começar com `/parceiros/restaurantes/do-usuario/{username}` OU `/parceiros/restaurantes/{restauranteId}`, é necessária uma autorização mais elaborada, que verifica se o usuário tem permissão a um restaurante específico, por meio da classe `RestauranteAuthorizationService`. Esse tipo de autorização, em que um usuário ter permissão em apenas alguns objetos de negócio é o que

chamamos de *ACL-based authorization*. A sigla ACL significa Access Control List.



Autenticação com Microservices e Single Sign On

Poderíamos implementar a autenticação numa Arquitetura de Microservices de duas maneiras:

- o usuário precisa autenticar novamente ao acessar cada serviço
- a autenticação é feita apenas uma vez e as informações de identidade do usuário são repassadas para os serviços

Autenticar várias vezes, a cada serviço, é algo que deixaria a experiência do usuário terrível. Além disso, todos os serviços teriam que ser *edge services*, expostos à rede externa.

Autenticar apenas uma vez e repassar as dados do usuário autenticado permite que os serviços não fiquem expostos, diminuindo a superfície de ataque. Além disso, a experiência para o usuário é transparente, como se todas as funcionalidades fossem parte da mesma aplicação. É esse tipo de solução que chamamos de **Single Sign On (SSO)**.

Autenticação no API Gateway e Autorização nos Serviços

No livro Microservice Patterns, Chris Richardson descreve uma maneira comum de lidar com autenticação em uma arquitetura de Microservices: implementá-la API Gateway, o único edge service que fica exposto para o mundo externo. Dessa maneira, as chamadas a URLs protegidas já seriam barradas antes de passar para a rede interna, no caso do usuário não estar autenticado.

E a autorização? Poderíamos fazê-la também no API Gateway. É algo razoável para role-based authorization, em que é preciso saber apenas o ROLE do usuário. Porém, implementar ACL-based authorization no API Gateway levaria a um alto acoplamento com os serviços, já que precisamos saber se um dado usuário tem permissão para um objeto de negócio específico. Então, provavelmente uma atualização em um serviço iria querer uma atualização sincronizada no API Gateway, diminuindo a independência de cada serviço. Portanto, uma ideia melhor é fazer a autorização, role-based ou ACL-based, em cada serviço.

Access Token e JWT

Com a autenticação sendo feito no API Gateway e a autorização em cada *downstream service*, surge um problema: como passar a identidade de um usuário do API Gateway para cada serviço?

Há duas alternativas:

- um token opaco: simplesmente uma string ou UUID que precisaria ser validada por cada serviço no emissor do token através de uma chamada remota.
- um self-contained token: um token que contém as informações do usuário e que tem sua integridade protegida através de uma assinatura. Assim, o próprio recipiente do token pode validar as informações checando a assinatura. Tanto o emissor como o recipiente devem compartilhar chaves para que a emissão e a

checagem do token possam ser realizadas.

Pattern: Acess Token

O API Gateway passa um token contendo informações sobre o usuário, como sua identidade e seus roles, para os demais serviços.

Implementamos stateless sessions no monólito com um JWS, um tipo de JWT que é um token self-contained e assinado. Podemos usar o mesmo mecanismo, fazendo com que o API Gateway repasse o JWT para cada serviço. Cada serviço checaria a assinatura e extrairia, do payload do JWT, o subject, que contém o id do usuário, e os respectivos roles, usando essas informações para checar a permissão do usuário ao recurso solicitado.

Autenticação e Autorização nos Microservices do Caelum Eats

A solução de stateless sessions com JWT do Caelum Eats, foi pensada e implementada visando uma aplicação monolítica.

E o resto dos serviços?

Temos serviços de infraestrutura como:

- API Gateway
- Service Registry
- Config Server
- Hystrix Dashboard
- Turbine
- Admin Server

Como estamos tratando de autorização relacionada a um determinado usuário, deixaremos para um outro momento a discussão da autenticação e autorização desses serviços de infraestrutura.

Temos serviços alinhados a contextos delimitados (bounded contexts) da

Caelum Eats, como:

- Distância
- Pagamento
- Nota Fiscal
- Administrativo, um novo serviço que acabamos de extrair

Há ainda módulos do monólito relacionados a contextos delimitados:

- Pedido
- Restaurante

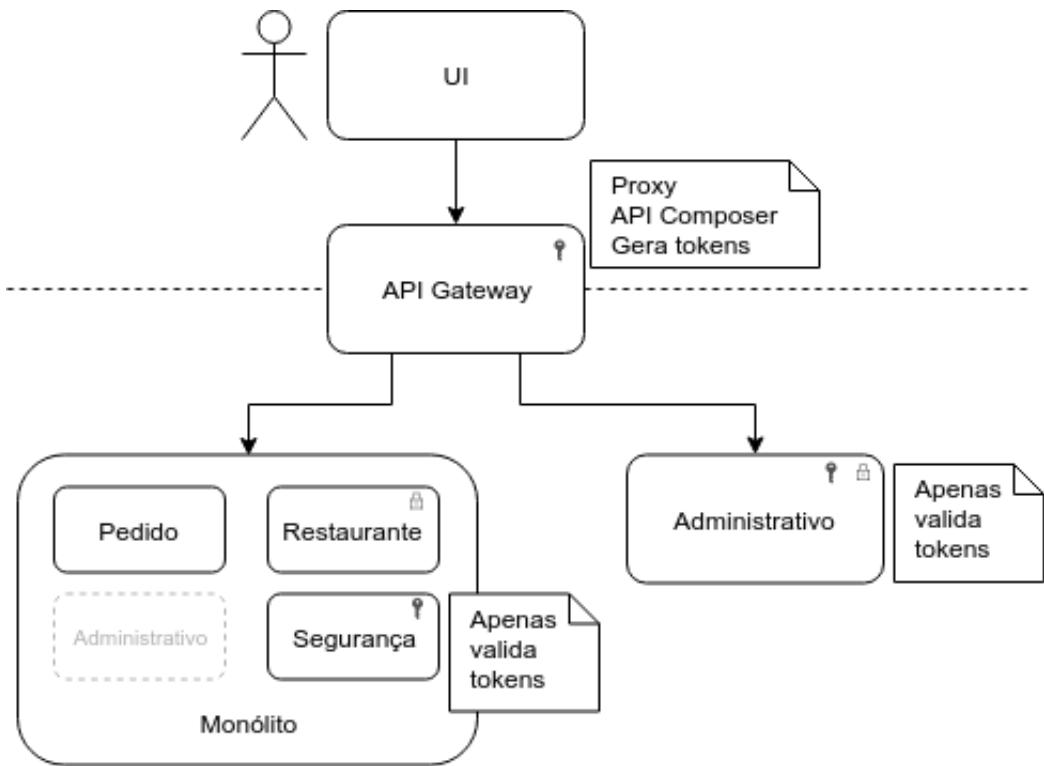
O único módulo cujos endpoints tem seu acesso protegido é o módulo de Restaurante do monólito. O módulo Administrativo foi extraído para um serviço próprio e não implementamos a autorização.

O monólito possui também um módulo de Segurança, que trata desse requisito transversal e contém o código de configuração do Spring Security.

O módulo Administrativo do monólito era protegido por meio de role-based authorization, bastando o usuário estar no role ADMIN para acessar os endpoints de administração de tipos de cozinha e formas de pagamento. Esse tipo de autorização não está sendo feito no eats-administrativo-service.

Já o módulo de Restaurante efetua ACL-based authorization, limitando o acesso do usuário com role PARCEIRO a um restaurante específico.

Vamos modificar esse cenário, passando a responsabilidade de geração de tokens JWT/JWS para o API Gateway, que também será responsável pelo cadastro de novos usuários. A validação do token e autorização dos recursos ficará a cargo do módulo Restaurante do monólito e do serviço Administrativo.



Autenticação no API Gateway

Poderíamos ter um BD específico para conter dados de usuários nas tabelas `user`, `role` e `userAuthorities`. Porém, para simplificar, vamos manter os dados de usuários no BD do próprio monólito.

No `config-repo`, adicione um arquivo `apigateway.properties` com os dados de conexão do BD do monólito, além das configurações da chave e expiração do JWT, que são usadas na geração do token:

```
##### config-repo/apigateway.properties
```

```
#DATASOURCE CONFIGS
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=123456

#JWT CONFIGS
jwt.secret = um-secreto-bem-secreto
jwt.expiration = 604800000
```

Adicione, ao API Gateway, dependências ao starter do Spring Data JPA e

ao driver do MySQL. Adicione também o JWT, biblioteca que gera e valida tokens JWT, e ao starter do Spring Security.

fj33-api-gateway/pom.xml

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>

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

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>

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

Copie as classes a seguir do módulo eats-seguranca do monólito para o pacote `br.com.caelum.apigateway.seguranca` do API Gateway:

- AuthenticationController
- AuthenticationDto
- Role
- User
- UserInfoDto
- UserRepository
- UserService

Copie a seguinte classe do módulo de segurança do monólito para o pacote `br.com.caelum.apigateway` do API Gateway:

- `PasswordEncoderConfig`

Não esqueça de ajustar o pacote das classes copiadas.

Essas classes fazem a autenticação de usuários, assim como o cadastro de novos donos de restaurante.

Defina uma classe `SecurityConfig` no pacote `br.com.caelum.apigateway` para que permita toda e qualquer requisição, desabilitando a autorização, que será feita pelos serviços. A autenticação será *stateless*.

```
##### fj33-api-
gateway/src/main/java;br/com/caelum/apigateway/SecurityConfig.java

@Configuration
@EnableWebSecurity
@AllArgsConstructor
class SecurityConfig extends WebSecurityConfigurerAdapter {

    private UserService userService;
    private PasswordEncoder passwordEncoder;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().permitAll()
            .and().cors()
            .and().csrf().disable()
            .formLogin().disable()
            .httpBasic().disable()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) {
        auth.userDetailsService(userService).passwordEncoder(passwordEncoder);
    }
}
```

```
}

@Override
@Bean(BeanIds.AUTHENTICATION_MANAGER)
public AuthenticationManager authenticationManagerBean() thr
    return super.authenticationManagerBean();
}

}
```

Não deixe de fazer os imports corretos:

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/SecurityConfig.java
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.BeanIds;
import org.springframework.security.config.annotation.authentication.configuration;
import org.springframework.security.config.annotation.web.builders.HttpBuilder;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.password.PasswordEncoder;

import br.com.caelum.apigateway.seguranca.UserService;
import lombok.AllArgsConstructor;
```

Defina, no pacote `br.com.caelum.apigateway.seguranca` do API Gateway, uma classe `JwtTokenManager`, responsável pela geração dos tokens. A validação e extração de informações de um token serão responsabilidade de cada serviço.

É importante adicionar o `username` e os `roles` do usuário aos *claims* do JWT.

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/seguranca/JwtTokenM
anager.java

@Component
class JwtTokenManager {

    private String secret;
    private long expirationInMillis;

    public JwtTokenManager(@Value("${jwt.secret}") String secret
                           @Value("${jwt.expiration}") long expirationInMillis)
    {
        this.secret = secret;
        this.expirationInMillis = expirationInMillis;
    }

    public String generateToken(User user) {
        final Date now = new Date();
        final Date expiration = new Date(now.getTime() + this.expirationInMillis);
        return Jwts.builder()
            .setIssuer("Caelum Eats")
            .setSubject(Long.toString(user.getId()))
            .claim("roles", user.getRoles())
            .claim("username", user.getUsername())
            .setIssuedAt(now)
            .setExpiration(expiration)
            .signWith(SignatureAlgorithm.HS256, this.secret)
            .compact();
    }
}
```

Cheque os imports:

```
##### fj33-api-
gateway/src/main/java/br/com/caelum/apigateway/seguranca/JwtTokenM
anager.java
```

```
import java.util.Date;  
  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
import io.jsonwebtoken.Jwts;  
import io.jsonwebtoken.SignatureAlgorithm;
```

Ainda no API Gateway, adicione um *forward* para a URL do `AuthenticationController`, de maneira que o Zuul não tente fazer o proxy dessa chamada:

```
##### fj33-api-gateway/src/main/resources/application.properties
```

```
zuul.routes.auth.path=/auth/**  
zuul.routes.auth.url=forward:/auth
```

Observação: essa configuração deve ficar antes da rota "coringa", que direciona todas as requisições para o monólito.

Podemos fazer uma chamada como a seguinte, que autentica o dono do restaurante Long Fu:

```
curl -i -X POST -H 'Content-type: application/json' -d '{"user
```

O retorno obtido será algo como:

```
{"userId":2,"username":"longfu","roles":["PARCEIRO"],"token":"eyJhbGciOiJIU
```

São retornados, no corpo da resposta, informações sobre o usuário, seus roles e um token. O token, no formato JWS, contém as mesmas informações do corpo da resposta, mas em base 64, e uma assinatura.

Validando o token JWT e implementando autorização no Monólito

Remova as seguintes classes do módulo eats-segurança do monólito, cujas responsabilidades foram passadas para o API Gateway. Elas estão no pacote `br.com.caelum.eats.segurança`:

- AuthenticationController
- AuthenticationDto
- UserInfoDto
- UserRepository
- UserService

Remova também a classe a seguir, do pacote `br.com.caelum.eats`:

- PasswordEncoderConfig

A classe `SecurityConfig` deve apresentar um erro de compilação.

Altere a classe `SecurityConfig` do módulo de segurança do monólito, removendo o código associado a autenticação e cadastro de novos usuários:

```
##### fj33-eats-monolito-modular/eats/eats-
segurança/src/main/java;br/com/caelum/eats/SecurityConfig.java
```

```
@Configuration
@EnableWebSecurity
@AllArgsConstructor
class SecurityConfig extends WebSecurityConfigurerAdapter {

    private UserService userService;
    private JwtAuthenticationFilter jwtAuthenticationFilter;
    private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;
    private PasswordEncoder passwordEncoder;

    // código omitido...
```

```
@Override  
protected void configure(final AuthenticationManager auth,  
    UserDetailsService userDetailsService, PasswordEncoder encoder)  
{  
  
    @Override  
    @Bean(BEAN_ID_AUTHENTICATION_MANAGER_BEAN)  
    public AuthenticationManager authenticationManagerBean()  
    {  
        return super.authenticationManagerBean();  
    }  
  
}  
  
-----
```

Remove os seguintes imports:

```
##### fj33-eats-monolito-modular/eats/eats-  
segurança/src/main/java/br/com/caelum/eats/SecurityConfig.java
```

```
import org.springframework.context.annotation.Bean;  
import org.springframework.security.authentication.  
import org.springframework.security.config.BeanIds;  
import org.springframework.security.config.annotation.  
import org.springframework.security.crypto.password.  
import br.com.caelum.eats.segurança.UserService;
```

```
-----  
Modifique o JwtTokenManager do módulo de segurança do monólito,  
removendo o código de geração de token e o atributo  
expirationInMillis, deixando apenas a validação e extração de dados  
do token.
```

```
##### fj33-eats-monolito-modular/eats/eats-  
segurança/src/main/java/br/com/caelum/eats/segurança/JwtTokenManager.java
```

```
@Component
class JwtTokenManager {

    private String secret;
    private long expirationInMillis;

    public JwtTokenManager(@Value("${jwt.secret}") String secret
                           @Value("${jwt.expiration}") long expiration)
    {
        this.secret = secret;
        this.expirationInMillis = expiration;
    }

    public String generateToken(User user) {
        ...
    }

    public boolean isValid(String jwt) {
        // não modificado ...
    }

    @SuppressWarnings("unchecked")
    public User getUserFromToken(String jwt) {
        // não modificado ...
    }
}
```

Remova os imports desnecessários:

```
#####
fj33-eats-monolito-modular/eats/eats-
seguranca/src/main/java/br/com/caelum/eats/seguranca/JwtTokenManag
er.java
```

```
import java.util.Date;
import io.jsonwebtoken.SignatureAlgorithm;
```

Remova as anotações do JPA e Beans Validator das classes User e Role do módulo de segurança do monólito. O cadastro de usuários será feito pelo API Gateway.

```
##### fj33-eats-monolito-modular/eats/eats-
segurança/src/main/java/br/com/caelum/eats/segurança/User.java
```

```
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
public class User implements UserDetails {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    @NotNull @JsonIgnore
    private String name;

    @NotNull @JsonIgnore
    private String password;

    @ManyToMany(fetch=FetchType.EAGER) @JsonIgnore
    private List<Role> authorities = new ArrayList<>();

    // restante do código ...
}
```

Limpe os imports da classe User:

```
##### fj33-eats-monolito-modular/eats/eats-
segurança/src/main/java/br/com/caelum/eats/segurança/User.java
```

```
import javax.persistence.Entity;
import javax.persistence.FetchType;
```

```
-import javax.persistence.GeneratedValue;
-import javax.persistence.GenerationType;
-import javax.persistence.Id;
-import javax.persistence.ManyToOne;
-import javax.validation.constraints.NotNull;
```

import com.fasterxml.jackson.annotation.JsonIgnore

```
##### fj33-eats-monolito-modular/eats/eats-
segurança/src/main/java/br/com/caelum/eats/segurança/Role.java
```

```
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Role implements GrantedAuthority {

    // código omitido...

    @Id
    private String authority;

    // restante do código...
```

Limpe também os imports da classe Role:

```
##### fj33-eats-monolito-modular/eats/eats-
segurança/src/main/java/br/com/caelum/eats/segurança/Role.java
```

```
import javax.persistence.Entity;
import javax.persistence.Id;
```

Como a classe `User` não é mais uma entidade, devemos modificar seu relacionamento na classe `Restaurante` do módulo `eats-restaurante` do

monólito:

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/Restaurante.ja
va
```

```
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Restaurante {

    // código omitido...

    @OneToOne
    private User user;
    private Long userId; // modificado

}
```

Os imports devem ser ajustados:

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/Restaurante.ja
va
```

```
import javax.persistence.OneToOne;
import br.com.caelum.eats.segurança.Usuario;
```

Modifique também o uso do atributo `user` do `Restaurante` na classe `RestauranteController`:

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/RestauranteC
ontroller.java
```

```
@RestController
@AllArgsConstructor
class RestauranteController {

    // código omitido...

    @PutMapping("/parceiros/restaurantes/{id}")
    public Restaurante atualiza(@RequestBody Restaurante restaurante) {
        Restaurante doBD = restauranteRepo.getOne(restaurante.getId());
        restaurante.setUser(doBD.getUser()); // modificado
        restaurante.setAprovado(doBD.getAprovado());
        // código omitido...
        return restauranteRepo.save(restaurante);
    }

    // código omitido...
}

}
```

Ajuste a interface RestauranteRepository:

```
#####
fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/RestauranteR
epository.java
```

```
interface RestauranteRepository extends JpaRepository<Restaurante, Long> {
    // código omitido...
    Restaurante findByUser(User user);
    Restaurante findByUserId(Long userId); // modificado
}
```

```
// código omitido...
}
```

Limpe o import:

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java(br/com/caelum/eats/restaurante/RestauranteR
epository.java
```

```
import br.com.caelum.eats.segurança.User;
```

Faça com que a classe `RestauranteAuthorizationService` use o novo método do repository:

```
##### fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java(br/com/caelum/eats/restaurante/RestauranteA
uthorizationService.java
```

```
@Service
@AllArgsConstructor
class RestauranteAuthorizationService {

    private RestauranteRepository restauranteRepo;

    public boolean checaId(Authentication authentication, long id) {
        User user = (User) authentication.getPrincipal();
        if (user.isInRole(Role.ROLES.PARCEIRO)) {
            Restaurante restaurante = restauranteRepo.findById(id);
            if (restaurante != null) {
                return id == restaurante.getId();
            }
        }
        return false;
}
```

```
// código omitido...
```

```
}
```

Mude o `monolito.properties` do `config-repo`, removendo a configuração de expiração do token JWT. Essa configuração será usada apenas pelo gerador de tokens, o API Gateway. A chave privada, presente na propriedade `jwt.secret` ainda deve ser mantida, pois é usada na validação do token HS256.

```
##### config-repo/monolito.properties
```

```
jwt.expiration=604800000
```

As URLs que não tem acesso protegido continuam funcionando sem a necessidade de um token. Por exemplo:

<http://localhost:9999/restaurantes/1>

ou

<http://localhost:8080/restaurantes/1>

Porém, URLs protegidas precisarão de um *access token* válido e que foi emitido para um usuário que tenha permissão para fazer operações no recurso solicitado.

Se tentarmos acessar uma URL como a seguir, teremos o acesso negado:

<http://localhost:8080/parceiros/restaurantes/1>

A resposta será um erro HTTP 401 (Unauthorized).

Devemos usar um token obtido na autenticação como o API Gateway,

colocando-o no cabeçalho HTTP Authorization, com Bearer como prefixo. O comando cURL em um Terminal, parece com:

```
curl -i -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJpc3N
```

Deverá ser obtida uma resposta bem sucedida!

```
HTTP/1.1 200
Date: Fri, 23 Aug 2019 00:56:00 GMT
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked

{"id":1,"cnpj":"98444252000104","nome":"Long Fu","descricao":"O melhor da C
```

Isso indica que o módulo de segurança do monólito reconheceu o token como válido e extraiu a informação dos roles do usuário, reconhecendo-o no role PARCEIRO.

Exercício: Autenticação no API Gateway e Autorização no monólito

1. Faça checkout da branch cap15-autenticacao-no-api-gateway-e-autorizacao-nos-servicos nos projeto do monólito modular, API Gateway e UI:

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap15-autenticacao-no-api-gateway-e-autorizacao-nos-servicos
```

```
cd ~/Desktop/fj33-api-gateway  
git checkout -f cap15-autenticacao-no-api-gateway-e-autorizacao  
  
cd ~/Desktop/fj33-eats-ui  
git checkout -f cap15-autenticacao-no-api-gateway-e-autorizacao
```

Faça refresh no Eclipse nos projetos do monólito modular e do API Gateway.

2. Poderíamos ter um BD específico para conter dados de usuários nas tabelas `user`, `role` e `user_authorities`. Porém, para simplificar, vamos manter os dados de usuários no BD do próprio monólito.

No `config-repo`, adicione um arquivo `apigateway.properties` com o dados de conexão do BD do monólito, além das configurações da chave e expiração do JWT, que são usadas na geração do token:

```
##### config-repo/apigateway.properties
```

```
#DATASOURCE CONFIGS  
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true  
spring.datasource.username=root  
spring.datasource.password=  
  
#JWT CONFIGS  
jwt.secret = um-secreto-bem-secreto  
jwt.expiration = 604800000
```

3. Execute `ApiGatewayApplication`, certificando-se que o Service Registry e o Config Server estão no ar.

Então, abra o terminal e simule a autenticação do dono do restaurante Long Fu:

```
curl -i -X POST -H 'Content-type: application/json' -d '{"user
```

Use o seguinte snippet, para evitar digitação:

<https://gitlab.com/snippets/1888245>

Você deve obter um retorno parecido com:

```
HTTP/1.1 200
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Fri, 23 Aug 2019 00:05:22 GMT

{"userId":2,"username":"longfu","roles":["PARCEIRO"],"token":"eyJhbGciOiJIU
```

São retornados, no corpo da resposta, informações sobre o usuário, seus roles e um token. Guarde esse token: o usaremos em breve!

4. Execute o `EatsApplication` do módulo `eats-application` do monólito. Certifique-se que o Service Registry, Config Server e API Gateway estejam sendo executados.

As URLs que não tem acesso protegido continuam funcionando. Por exemplo, acesse, pelo navegador, a URL a seguir para obter todas as formas de pagamento:

<http://localhost:9999/restaurantes/1>

ou

<http://localhost:8080/restaurantes/1>

Deve funcionar e retornar algo como:

```
{"id":1,"cnpj":"98444252000104","nome":"Long Fu","descricao":"
```

Porém, URLs protegidas precisarão de um *access token* válido e que foi emitido para um usuário que tenha permissão para fazer operações no recurso solicitado.

Tente acessar uma variação da URL anterior que só é acessível para usuários com o role PARCEIRO:

<http://localhost:8080/parceiros/restaurantes/1>

A resposta será um erro HTTP 401 (Unauthorized), com uma mensagem de acesso negado.

Use o token obtido no exercício anterior, de autenticação no API Gateway, colocando-o no cabeçalho HTTP `Authorization`, depois do valor `Bearer`. Faça o seguinte comando cURL em um Terminal:

```
curl -i -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJpc3M
```

Você pode encontrar o comando anterior em:

<https://gitlab.com/snippets/1888252>

Deverá ser obtida uma resposta bem sucedida, com os dados da forma de pagamento alterados!

```
HTTP/1.1 200
Date: Fri, 23 Aug 2019 00:56:00 GMT
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json; charset=UTF-8
```

Transfer-Encoding: chunked

```
{"id":1,"cnpj":"98444252000104","nome":"Long Fu","descricao":"O melhor da C
```

Isso indica que o módulo de segurança do monólito reconheceu o token como válido e extraiu a informação dos roles do usuário, reconhecendo-o no role PARCEIRO.

5. Altere o payload do JWT, definindo um valor diferente para o `sub`, o `Subject`, que indica o id do usuário assim como para o `ROLE`.

Para isso, vá até um site como o <http://www.base64url.com/> e defina no campo *Base 64 URL Encoding* o payload do token JWT recebido do API Gateway.

Altere o `sub` para `1`, simulando um usuário malicioso tentando forjar um token para roubar a identidade de outro usuário, de id diferente. Mude também o role para `ADMIN` e o nome do usuário para `admin`.

O texto codificado em Base 64 URL Encoding será algo como:

```
eyJpc3MiOiJDYVVsdW0gRWF0cyIsInN1YiI6IjEiLCJyb2xlcyI6WyJBRE1JTlJdLCJ1c2Vyb...F
```

Através de um Terminal, use o cURL para tentar alterar uma forma de pagamento utilizando o payload modificado do JWT:

```
curl -i -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJpc3M...
```

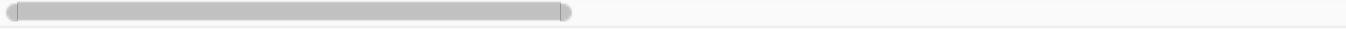
Obtenha o comando anterior na seguinte URL:

<https://gitlab.com/snippets/1888416>

Como a assinatura do JWT não bate com o payload, o acesso deverá ser negado:

```
HTTP/1.1 401
Date: Fri, 23 Aug 2019 12:47:07 GMT
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked

{"timestamp": "2019-08-23T12:47:07.785+0000", "status": 401, "error": "Unauthori
```



Teste também com o cURL o acesso direto ao monólito, usando a porta 8080. O acesso deve ser negado, da mesma maneira.

6. (desafio) Faça com que o novo serviço Administrativo também tenha autorização, fazendo com que apenas usuários no role ADMIN tenham acesso a URLs que começam com /admin.

Deixando de reinventar a roda com OAuth 2.0

Da maneira como implementamos a autenticação anteriormente, acabamos definindo mais uma responsabilidade para o API Gateway: além de proxy e API Composer, passou a servir como autenticador e gerador de tokens. E, para isso, o API Gateway precisou conhecer tabelas dos usuários e seus respectivos roles. E mais: implementamos a geração e verificação de tokens manualmente.

Autenticação, autorização, tokens, usuário e roles são necessidades comuns e poderiam ser implementadas de maneira genérica. Melhor ainda se houvesse um padrão aberto, que permitisse implementação por diferentes fornecedores. Assim, os desenvolvedores poderiam focar mais em código de negócio e menos em código de segurança.

Há um framework de autorização baseado em tokens que permite que

não nos preocupemos com detalhes de implementação de autenticação e autorização: o padrão **OAuth 2.0**. Foi definido na RFC 6749 da Internet Engineering Task Force (IETF), em Outubro de 2012.

Há extensões do OAuth 2.0 como o OpenID Connect (OIDC), que fornece uma camada de autenticação baseada em tokens JWT em cima do OAuth 2.0.

O foco original do OAuth 2.0, na verdade, é permitir que aplicações de terceiros usem informações de usuários em serviços como Google, Facebook e GitHub. Quando efetuamos login em uma aplicação com uma conta do Facebook ou quando permitimos que um serviço de Integração Contínua como o Travis CI acesse nosso repositório no GitHub, estamos usando OAuth 2.0.

Um padrão como o OAuth 2.0 nos permite instalar softwares como KeyCloak, WSO2 Identity Server, OpenAM ou Gluu e até usar soluções prontas de *identity as a service* (IDaaS) como Auth0 ou Okta.

E, claro, podemos usar as soluções do Spring: **Spring Security OAuth**, que estende o Spring Security fornecendo implementações para OAuth 1 e OAuth 2.0. Há ainda o **Spring Cloud Security**, que traz soluções compatíveis com outros projetos do Spring Cloud.

Roles

O OAuth 2.0 define quatro componentes, chamados de roles na especificação:

- **Resource Owner**: em geral, o usuário que tem algum recurso protegido como sua conta no Facebook, suas fotos no Flickr, seus repositórios no GitHub ou seu restaurante no Caelum Eats.
- **Resource Server**: provê o recurso protegido e permite o acesso mediante o uso de access tokens válidos.
- **Client**: a aplicação, Web, Single Page Application (SPA), Desktop ou Mobile, que deseja acessar os recursos do *Resource Owner*. Um

Client precisa estar registrado no *Authorization Server*, sendo identificado por um *client id* e um *client secret*.

- **Authorization Server:** provê uma API para autenticar usuário e gerar access tokens. Pode estar na mesma aplicação do *Resource Server*.

O padrão OAuth 2.0 não especifica um formato para o access token. Se for usado um **access token opaco**, como uma String randômica ou UUID, a validação feita pelo Resource Server deve invocar o Authorization Server. Já no caso de um **self-contained access token** como um JWT/JWS, o próprio token contém informações para sua validação.

Grant Types

O padrão OAuth 2.0 é bastante flexível e especifica diferentes maneiras de um *Client* obter um access token, chamadas de *grant types*:

- **Password:** usada quando há uma forte relação de confiança entre o Client e o Authorization Server, como quando ambos são da mesma organização. O usuário informa suas credenciais (username e senha) diretamente para o Client, que repassa essas credenciais do usuário para o Authorization Server, junto com seu client id e client secret.
- **Client credentials:** usada quando não há um usuário envolvido, apenas um sistema chamando um recurso protegido de outro sistema. Apenas as credenciais do Client são informadas para o Authorization Server.
- **Authorization Code:** usada quando aplicações de terceiros desejam acessar informações de um recurso protegido sem que o Client conheça explicitamente as credenciais do usuário. Por exemplo, quando um usuário (Resource Owner) permite que o Travis CI (Client) acesse os seus repositórios do GitHub (Authorization Server e Resource Server). No momento em que o usuário cobra seu GitHub no Travis CI, é redirecionado para uma tela de login do

GitHub. Depois de efetuar o login no GitHub e escolher as permissões (ou *scopes* nos termos do OAuth), é redirecionado para um servidor do Travis CI com um *authorization code* como parâmetro da URL. Então, o Travis CI invoca o GitHub passando esse authorization code para obter um access token. As aplicações de terceiro que utilizam um authorization code são, em geral, aplicações Web clássicas com renderização das páginas no *server-side*.

- **Implicit:** o usuário é direcionado a uma página de login do Authorization Server, mas o redirect é feito diretamente para o user-agent (o navegador, no caso da Web) já enviando o access token. Dessa forma, o Client SPA ou Mobile conhece diretamente o access token. Isso traz uma maior eficiência porém traz vulnerabilidades.

A RFC 8252 (OAuth 2.0 for Native Apps), de Outubro de 2017, traz indicações de como fazer autenticação e autorização com OAuth 2.0 para aplicações mobile nativas.

No OAuth 2.0, um access token deve ter um tempo de expiração. Um token expirado levaria à necessidade de nova autenticação pelo usuário. Um Authorization Server pode emitir um *refresh token*, de expiração mais longa, que seria utilizado para obter um novo access token, sem a necessidade de nova autenticação. De acordo com a especificação, o grant type Implicit não deve permitir um refresh token, já que o token é conhecido e armazenado no próprio user-agent.

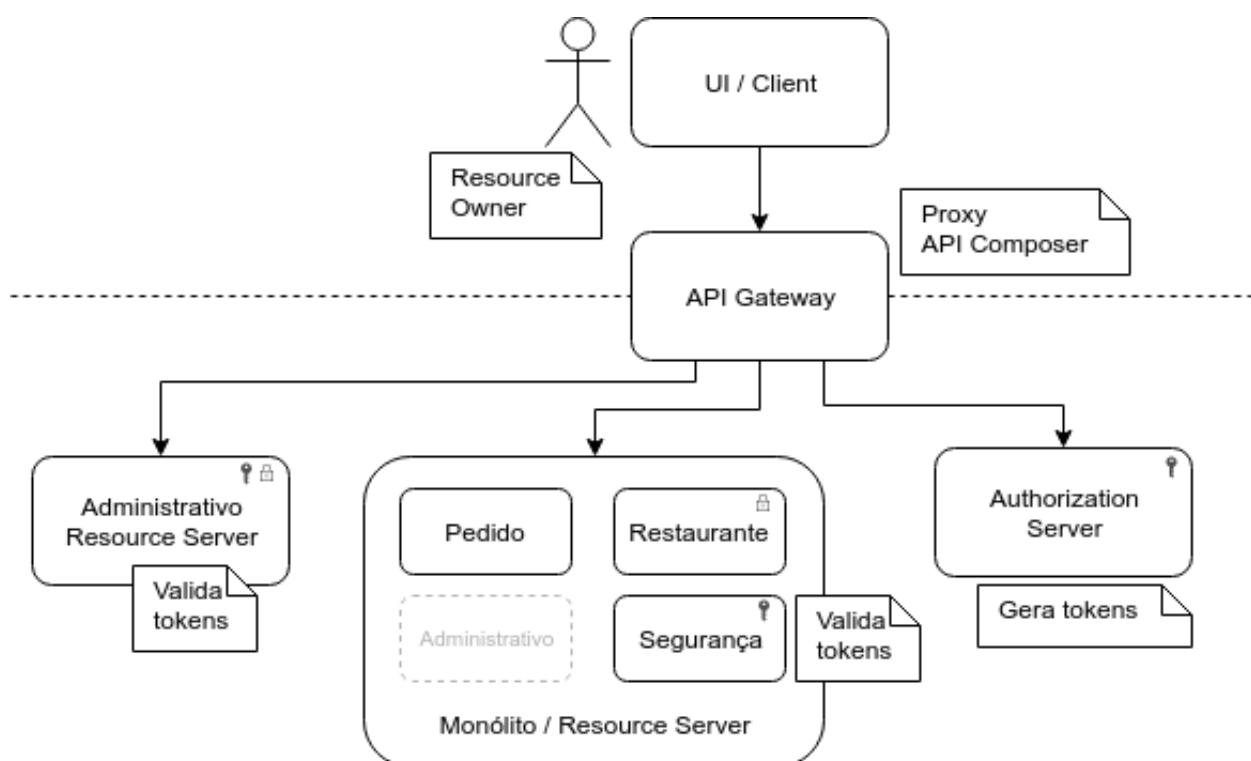
OAuth no Caelum Eats

Podemos dizer que o API Gateway, que conhece os dados de usuário e seus roles, gera tokens e faz autenticação, é análogo a um Authorization Server do OAuth. O monólito, com a implementação de autorização para os módulos de Restaurante e Admin, serve como um Resource Server do OAuth. O front-end em Angular seria o Client do OAuth.

A autenticação no API Gateway é feita usando o nome do usuário e a respectiva senha que são informadas na própria aplicação do Angular.

Ou seja, o Client conhece as credenciais do usuário e as repassa para o Authorization Server para autenticá-lo. Isso é análogo a um **Password grant type** do OAuth.

Poderíamos reimplementar a autenticação e autorização com OAuth usando código já pronto das bibliotecas Spring Security OAuth 2 e Spring Cloud Security, diminuindo o código que precisamos manter e cujas vulnerabilidades temos que sanar. Para isso, podemos definir um Authorization Server separado do API Gateway, responsável apenas pela autenticação e gerenciamento de tokens.



Authorization Server com Spring Security OAuth 2

Para implementarmos um Authorization Server compatível com OAuth 2.0, devemos criar um novo projeto Spring Boot e adicionar como dependência o starter do Spring Cloud OAuth2:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

Com a dependência ao `spring-cloud-starter-oauth2` definida, devemos anotar a Application com `@EnableAuthorizationServer`.

No `application.properties`, devemos definir um client id e seu respectivo client secret:

```
security.oauth2.client.client-id=eats  
security.oauth2.client.client-secret=eats123
```

A configuração anterior define apenas um Client. Se tivermos registro de diferentes clients, podemos fornecer uma implementação da interface `ClientDetailsService`, que define o método `loadClientByClientId`. Nesse método, recebemos uma String com o client id e devemos retornar um objeto que implementa a interface `ClientDetails`.

Com essas configurações mínimas, teremos um Authorization Server que dá suporte a todos os grant types do OAuth 2.0 mencionados acima.

Se quisermos usar o Password grant type, devemos fornecer uma implementação da interface `UserDetailsService`, usada pelo Spring Security para obter os detalhes dos usuários. Essa implementação é exatamente igual ao que implementamos no API Gateway, nas classes `UserService`, `User` e `Role`, `UserRepository` e `SecurityConfig`. Para obter o registro dos usuários, o Authorization Server deve ter um data source que aponte para as tabelas de usuários e seus roles.

Ao executar o Authorization Server, podemos gerar um token enviando uma requisição POST ao endpoint `/oauth/token`. As credenciais do Client devem ser autenticadas com HTTP Basic. Devem ser definidos como parâmetros o grant type e o scope. Como não definimos nenhum scope, devemos usar `any`. No caso do Password grant type, devemos informar também as credenciais do usuário.

```
curl -i -X POST  
--basic -u eats:eats123  
-H 'Content-Type: application/x-www-form-urlencoded'  
-d 'grant_type=password&username=admin&password=123456&scope=  
http://localhost:8085/oauth/token
```

Como resposta, obteremos um access token e um refresh token, ambos opacos.

```
HTTP/1.1 200  
Pragma: no-cache  
Cache-Control: no-store  
X-Content-Type-Options: nosniff  
X-XSS-Protection: 1; mode=block  
X-Frame-Options: DENY  
Content-Type: application/json; charset=UTF-8  
Transfer-Encoding: chunked  
Date: Wed, 28 Aug 2019 13:54:22 GMT  
  
{ "access_token": "bdb22855-5705-4533-b925-f1091d576db7", "token_type": "bearer"
```

Podemos checar um token opaco por meio de uma requisição GET ao endpoint /oauth/check_token, passando o access token obtido no parâmetro token:

```
curl -i localhost:8080/oauth/check_token/?token=bdb22855-5705-
```

O corpo da resposta deve conter o username e os roles do usuário, entre outras informações:

```
HTTP/1.1 200  
X-Content-Type-Options: nosniff  
X-XSS-Protection: 1; mode=block
```

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 28 Aug 2019 14:56:32 GMT

{"active":true,"exp":1567046599,"user_name":"admin","authorities":["ROLE_AD
```

Erros comuns

Se as credenciais do Client estiverem incorretas

```
curl -i -X POST --basic -u eats:SENHA_ERRADA -H 'Content-Type:
```

receberemos um status 401 (Unauthorized):

```
HTTP/1.1 401
...
>{"timestamp":"2019-08-28T14:39:58.413+0000","status":401,"error":"Unauthori
```

Se as credenciais do usuário estiverem incorretas, no caso de um
Password grant type

```
curl -i -X POST --basic -u eats:eats123 -H 'Content-Type: appl
```

receberemos um status 400 (Bad Request), com *Bad credentials* como
mensagem de erro

```
HTTP/1.1 400
...

```

```
{"error": "invalid_grant", "error_description": "Bad credentials"}
```

Se omitirmos o scope

```
curl -i -X POST --basic -u eats:eats123 -H 'Content-Type: appl
```

receberemos um status 400 (Bad Request), com *Empty scope* como mensagem de erro

```
HTTP/1.1 400
```

...

```
{"error": "invalid_scope", "error_description": "Empty scope (either the client
```

Se omitirmos o grant type

```
curl -i -X POST --basic -u eats:eats123 -H 'Content-Type: appl
```

receberemos um status 400 (Bad Request), com *Missing grant type* como mensagem de erro

```
HTTP/1.1 400
```

...

```
{"error": "invalid_request", "error_description": "Missing grant type"}
```

Se informarmos um grant type incorreto

```
curl -i -X POST --basic -u eats:eats123 -H 'Content-Type: appl
```

receberemos um status 400 (Bad Request), com *Unsupported grant type* como mensagem de erro

```
HTTP/1.1 400
...
{"error": "unsupported_grant_type", "error_description": "Unsupported grant ty
```

Se, ao checarmos um token, passarmos um token expirado ou inválido

```
curl -i localhost:8085/oauth/check_token/?token=TOKEN_INVALIDO
```

receberemos um status 400 (Bad Request), com *Token was not recognised* como mensagem de erro

```
HTTP/1.1 400
...
{"error": "invalid_token", "error_description": "Token was not recognised"}
```

JWT como formato de token no Spring Security OAuth 2

A dependência `spring-cloud-starter-oauth2` já tem como dependência transitiva a biblioteca `spring-security-jwt`, que provê suporte a JWT no Spring Security.

Precisamos fazer algumas configurações para que o token gerado seja um JWT. Para isso, devemos definir uma implementação para a interface `AuthorizationServerConfigurer`. Podemos usar a classe `AuthorizationServerConfigurerAdapter` como auxílio.

As configurações são as seguintes:

- um objeto da classe `JwtTokenStore`, que implementa a interface `TokenStore`
- um objeto da classe `JwtAccessTokenConverter`, que implementa a

`interface AccessTokenConverter`. A classe `JwtAccessTokenConverter` gera, por padrão, um chave privada de assinatura (`signingKey`) randômica. É interessante definir uma propriedade `jwt.secret`, como havíamos feito anteriormente.

- uma implementação de `clientDetailsService` para que as propriedades `security.oauth2.client.client-id` e `security.oauth2.client.client-secret` funcionem e definam o id e a senha do Client com sucesso. Podemos usar a classe `ClientDetailsServiceConfigurer`. Os valores das propriedades de Client id e secret podem ser obtidas usando `OAuth2ClientProperties`.
- devemos definir o `AuthenticationManager` configurado na classe `SecurityConfig` por meio da classe `AuthorizationServerEndpointsConfigurer`

Fazemos todas essas configurações na classe `OAuthServerConfig` a seguir:

```
@Configuration
public class OAuthServerConfig extends AuthorizationServerConf

    private final AuthenticationManager authenticationManager;
    private final OAuth2ClientProperties clientProperties;
    private final String jwtSecret;

    public OAuthServerConfiguration(AuthenticationManager auther
                                    OAuth2ClientProperties clientProperties,
                                    @Value("${jwt.secret}") String jwtSecret) {
        this.authenticationManager = authenticationManager;
        this.clientProperties = clientProperties;
        this.jwtSecret = jwtSecret;
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients
        clients.inMemory()
```

```
        .withClient(clientProperties.getClientId())
        .secret(clientProperties.getClientSecret());
    }

@Override
public void configure(AuthorizationServerEndpointsConfigurer
    endpoints.tokenStore(tokenStore())
        .accessTokenConverter(accessTokenConverter())
        .authenticationManager(authenticationManager);
}

@Bean
public TokenStore tokenStore() {
    return new JwtTokenStore(accessTokenConverter());
}

@Bean
public JwtAccessTokenConverter accessTokenConverter() {
    JwtAccessTokenConverter converter = new JwtAccessTokenConverter();
    converter.setSigningKey(this.jwtSecret);
    return converter;
}

}
```

A configuração padrão habilitada pela anotação

@EnableAuthorizationServer usa um NoOpsPasswordEncoder, que faz com que as senhas sejam lidas em texto puro. Porém, como definimos o BCryptPasswordEncoder no nosso SecurityConfig, precisaremos modificar a propriedade security.oauth2.client.client-secret no arquivo application.properties:

```
security.oauth2.client.client-secret=$2a$10$1YJxJHAbtsSCeyqgNj
```

Ao executar novamente o Authorization Server, os tokens serão gerados no formato JWT/JWS.

Podemos testar novamente com o cURL:

```
curl -i -X POST --basic -u eats:eats123 -H 'Content-Type: appl
```

Teremos uma resposta bem sucedida, com um access token no formato JWT:

```
HTTP/1.1 200
Pragma: no-cache
Cache-Control: no-store
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
X-Frame-Options: DENY
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 28 Aug 2019 18:11:25 GMT

{"access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1NjcwNTkwo
```

O access token anterior contém, como todo JWS, 3 partes.

O cabeçalho:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Que pode ser decodificado, usando um Base 64 URL Decoder, para:

```
{"alg": "HS256", "typ": "JWT"}
```

Já a segunda parte é o payload, que contém os claims do JWT:

```
eyJleHAiOjE1NjcwNTkwODUsInVzZXJfbmFtZSI6ImFkbWluIiwiYXV0aG9yaXRpZXMiolsiUk9
```

Após a decodificação Base64, teremos:

```
{  
  "exp":1567059085,  
  "user_name":"admin",  
  "authorities":["ROLE_ADMIN"],  
  "jti":"689d0a4e-1f4f-498c-8c31-9b5eb32ef1b8",  
  "client_id":"eats",  
  "scope": ["any"]}
```

Perceba que temos o `user_name` e os respectivos roles em `authorities`.

Há também uma propriedade `jti` (JWT ID), uma String randômica (UUID) que serve como um *nonce*: um valor é diferente a cada request e previne o sistema contra *replay attacks*.

A terceira parte é a assinatura:

```
ztypX3GJPYU8UNhHRtmEtQ7SLiizdzOrdCRJt64ovF4
```

Como usamos o algoritmo `HS256`, um algoritmo de chaves simétricas, a chave privada setada em `signingKey` precisa ser conhecida para validar a assinatura.

Exercício: um Authorization Server com Spring Security OAuth 2

1. Abra um Terminal e baixe o projeto `fj33-authorization-server` para o seu Desktop usando o Git:

```
cd ~/Desktop  
git clone https://gitlab.com/aovs/projetos-cursos/fj33-authori
```

2. No workspace de microservices do Eclipse, acesse *File > Import > Existing Maven Projects* e clique em *Next*. Em *Root Directory*, aponte para o diretório clonado anteriormente.

Veja o código das classes `AuthorizationServerApplication` e `OAuthServerConfig`, além dos arquivos `bootstrap.properties` e `application.properties`.

Note que o `spring.application.name` é `authorizationserver`. A porta definida para o Authorization Server é 8085.

3. Crie o arquivo `authorizationserver.properties` no config-repo, com o seguinte conteúdo:

```
#DATASOURCE CONFIGS
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=

jwt.secret = um-secreto-bem-secreto

security.oauth2.client.client-id=eats
security.oauth2.client.client-secret=$2a$10$1YJxJHAbtsSCeyqgNj
```

O código anterior pode ser encontrado em:

<https://gitlab.com/snippets/1890756>

Note que copiamos o `jwt.secret` e os dados do BD do monólito. Isso indica que o BD será mantido de maneira monolítica. Eventualmente, seria possível fazer a migração de dados de usuário para um BD específico.

Além disso, definimos as propriedades de Client id e secret do Spring Security OAuth 2.

Não deixe de comitar o novo arquivo no repositório Git.

4. Execute a classe AuthorizationServerApplication.

Então, abra um terminal e execute o seguinte comando:

```
curl -i -X POST --basic -u eats:eats123 -H 'Content-Type: appl
```

Se não quiser digitar, é possível encontrar o comando anterior no seguinte link: <https://gitlab.com/snippets/1890014>

Como resposta, deverá ser exibido algo como:

```
HTTP/1.1 200
```

```
...
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJleHAiOjE1NjcwNTkwODUsInVzZXJfbmFtZSI6ImFkbWluIiwiYXV0aG9yaXRpZXMiolsiu
    ZtYpx3GJPYU8UNhHRtmEtQ7SLiiZdZOrdCRJt64ovF4",
  "token_type": "bearer",
  "expires_in": 43199,
  "scope": "any",
  "jti": "689d0a4e-1f4f-498c-8c31-9b5eb32ef1b8"
}
```

Pegue o conteúdo da propriedade `access_token` e analise o cabeçalho e o payload em: <https://jwt.io>

O payload deverá conter algo semelhante a:

```
{
  "exp": 1567059085,
  "user_name": "admin",
  "authorities": [
    "ROLE_ADMIN"
  ],
```

```
"jti": "689d0a4e-1f4f-498c-8c31-9b5eb32ef1b8",
"client_id": "eats",
"scope": [
    "any"
]
}
```

5. Remova o código de autenticação do API Gateway.

Para isso, delete as seguintes classes do API Gateway:

- `AuthenticationController`
- `AuthenticationDto`
- `JwtTokenManager`
- `PasswordEncoderConfig`
- `Rote`
- `SecurityConfig`
- `User`
- `UserInfoDto`
- `UserRepository`
- `UserService`

Remova as seguintes dependências do `pom.xml` do API Gateway:

- `jjwt`
- `mysql-connector-java`
- `spring-boot-starter-data-jpa`
- `spring-boot-starter-security`

Apague a seguinte rota do `application.properties` do API Gateway:

```
zuul.routes.auth.path=/auth/**  
zuul.routes.auth.url=forward://auth
```

Delete o arquivo `apigateway.properties` do config-repo.

6. (desafio - trabalhoso) Aplique uma estratégia de migração de dados de usuário do monólito para um BD específico para o Authorization Server.

Resource Server com Spring Security OAuth 2

Para definir um Resource Server com o Spring Security OAuth 2, que consiga validar e decodificar os tokens (opacos ou JWT) emitidos pelo Authorization Server, basta anotar a aplicação ou uma configuração com `@EnableResourceServer`.

Podemos definir, na configuração `security.oauth2.resource.token-info-uri`, a URI de validação de tokens opacos.

No caso de token self-contained JWT, devemos definir a propriedade `security.oauth2.resource.jwt.key-value`. Pode ser a chave simétrica, no caso de algoritmos como o HS256, ou a chave pública, como no RS256. A chave pública em um algoritmo assimétrico pode ser baixada do servidor quando definida a propriedade `security.oauth2.resource.jwt.key-uri`.

Por padrão, todos os endereços requerem autenticação. Porém, é possível customizar esse e outros detalhes fornecendo uma implementação da interface `ResourceServerConfigurer`. É possível herdar da classe `ResourceServerConfigurerAdapter` para facilitar as configurações.

Protegendo o serviço Administrativo

Adicione os starters do Spring Security OAuth 2 e Spring Cloud Security ao `pom.xml` do `eats-administrativo-service`:

```
##### fj33-eats-administrativo-service/pom.xml
```

```
<dependency>
<groupId>org.springframework.cloud</groupId>
```

```
<artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-security</artifactId>
</dependency>
```

Anote a classe EatsAdministrativoServiceApplication com
@EnableResourceServer:

```
##### fj33-eats-administrativo-
service/src/main/java/br/com/caelum/eats/administrativo/EatsAdministrati
voServiceApplication.java
```

```
@EnableResourceServer // adicionado
@EnableDiscoveryClient
@SpringBootApplication
public class EatsAdministrativoServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(EatsAdministrativoServiceApplication.
    }

}
```

O import correto é o seguinte:

```
##### fj33-eats-administrativo-
service/src/main/java/br/com/caelum/eats/administrativo/EatsAdministrati
voServiceApplication.java
```

```
import org.springframework.security.oauth2.config.annotation.v
```

Adicione ao administrativo.properties do config-repo, a mesma chave

usada no Authorization Server, porém na propriedade
security.oauth2.resource.jwt.key-value:

```
##### config-repo/administrativo.properties
```

```
security.oauth2.resource.jwt.key-value = um-secreto-bem-secret
```

Crie uma classe `OAuthResourceServerConfig`. Herde da classe `ResourceServerConfigurerAdapter` e permita que todos acessem a listagem de tipos de cozinha e formas de pagamento, assim como os endpoints do Spring Boot Actuator. As URLs que começam com /admin devem ser restritas a usuário que tem o role ADMIN.

```
##### f33-eats-administrativo-
service/src/main/java/br/com/caelum/eats/administrativo/OAuthResource
ServerConfig.java
```

```
@Configuration
class OAuthResourceServerConfig extends ResourceServerConfigur

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/tipos-de-cozinha/**", "/formas-de-pagamento/**")
            .antMatchers("/actuator/**").permitAll()
            .antMatchers("/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and().cors()
            .and().csrf().disable()
            .formLogin().disable()
            .httpBasic().disable();
    }

}
```

Certifique-se que os imports estão corretos:

```
##### fj33-eats-administrativo-
service/src/main/java/br/com/caelum/eats/administrativo/OAuthResource
ServerConfig.java
```

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.build
import org.springframework.security.oauth2.config.annotation.v
```

URLs abertas, como a que lista todas as formas de pagamento, terão sucesso sem nenhum token ser passado:

<http://localhost:8084/formas-de-pagamento>

Já URLs como a que permite a alteração dos dados de uma forma de pagamento estarão protegidas:

```
curl -i -X PUT -H 'Content-type: application/json' -d '{"id":
```

Como resposta, teríamos um erro 401 (Unauthorized):

```
HTTP/1.1 401
{"error": "unauthorized", "error_description": "Full authentication is require
```

Será necessário passar um token obtido do Authorization Server que contém o role `ADMIN` para que a chamada anterior seja bem sucedida.

Exercício: Protegendo o serviço Administrativo com Spring Security OAuth 2

1. Faça checkout da branch `cap15-resource-server-com-spring-security-oauth-2` do serviço Administrativo:

```
cd ~/Desktop/fj33-eats-administrativo-service  
git checkout -f cap15-resource-server-com-spring-security-oaut
```

Faça refresh do projeto no Eclipse e o reinicie.

2. Abra um terminal e tente listas todas as formas de pagamento sem passar nenhum token:

```
curl http://localhost:8084/formas-de-pagamento
```

A resposta deve ser bem sucedida, contendo algo como:

```
[{"id":4,"tipo":"VALE_REFEICAO","nome":"Alelo"}, {"id":3,"tipo":"CARTAO_CRED
```

Vamos tentar editar uma forma de pagamento, chamando um endpoint que começa com /admin, sem um token:

```
curl -i -X PUT -H 'Content-type: application/json' -d '{"id":
```

O comando anterior pode ser encontrado em:

<https://gitlab.com/snippets/1888251>

Deve ser retornado um erro 401 (Unauthorized), com a descrição *Full authentication is required to access this resource*, indicando que o acesso ao recurso depende de autenticação:

```
HTTP/1.1 401  
Pragma: no-cache  
WWW-Authenticate: Bearer realm="oauth2-resource", error="unauthorized", err  
Cache-Control: no-store  
X-Content-Type-Options: nosniff
```

```
X-XSS-Protection: 1; mode=block  
X-Frame-Options: DENY  
Content-Type: application/json; charset=UTF-8  
Transfer-Encoding: chunked  
Date: Thu, 29 Aug 2019 20:12:57 GMT
```

```
{"error": "unauthorized", "error_description": "Full authentication is require
```

Devemos incluir, no cabeçalho Authorization, o token JWT obtido anteriormente:

```
curl -i -X PUT -H 'Content-type: application/json' -H 'Authori
```

O comando acima pode ser encontrado em:

<https://gitlab.com/snippets/1890417>

Observação: troque TOKEN-JWT-AQUI pelo token obtido do Authorization Server em exercícios anteriores.

A resposta será um sucesso!

```
HTTP/1.1 200  
X-Content-Type-Options: nosniff  
X-XSS-Protection: 1; mode=block  
Cache-Control: no-cache, no-store, max-age=0, must-revalidate  
Pragma: no-cache  
Expires: 0  
X-Frame-Options: DENY  
Content-Type: application/json; charset=UTF-8  
Transfer-Encoding: chunked  
Date: Thu, 29 Aug 2019 20:13:02 GMT
```

```
{"id": 3, "tipo": "CARTAO_CREDITO", "nome": "Amex Express"}
```

Protegendo serviços de infraestrutura

Temos serviços de infraestrutura como:

- API Gateway
- Service Registry
- Config Server
- Hystrix Dashboard
- Turbine
- Admin Server

O API Gateway é *edge service* do Caelum Eats, que fica na fronteira da infra-estrutura. Serve como um proxy que repassa requisições e as respectivas respostas. Podemos fazer um *rate limiting*, cortando requisições de um mesmo cliente a partir de uma certa taxa de requisições por segundo, afim de evitar um ataque de DDoS (Distributed Denial of Service), que visa deixar um sistema fora do ar. O API Gateway também serve como um API Composer, que dispara requisições a vários serviços, agregando as respostas. Nesse caso, é preciso avaliar se a composição requer algum tipo de autorização. No caso implementado, a composição que agrupa dados de um restaurante com sua distância a um determinado CEP, é feita por meio de dados públicas. Portanto, não há a necessidade de autorização. Nesse cenário de composição, a avaliação da necessidade de autorização, deve ser feita caso a caso. Uma ideia simples é repassar erros de autorização dos serviços invocados.

Uma vulnerabilidade da nossa aplicação é que uma vez que o endereço do Service Registry é conhecido, é possível descobrir nomes, hosts e portas de todos os serviços. A partir dos nomes dos serviços, podemos consultar o Config Server e observar detalhes de configuração de cada serviço.

Podemos, de maneira bem fácil, proteger o Config Server, o Service Registry e demais serviços de infraestrutura que criamos.

Basta adicionarmos, às dependências do Maven, o Spring Security:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Para que o Spring Security não use uma senha randômica, devemos definir usuário e senha como propriedades no `application.properties`. Por exemplo, para o Config Server:

```
security.user.name=configUser
security.user.password=configPassword
```

Nos demais serviços, devemos adicionar ao `bootstrap.properties`:

```
spring.cloud.config.uri=http://localhost:8888
spring.cloud.config.username=configUser
spring.cloud.config.password=configPassword
```

No caso do Service Registry, faríamos o mesmo processo.

Definiríamos o endereço nos clientes do Eureka da seguinte maneira:

```
eureka.client.serviceUrl.defaultZone=http://eurekaUser:eurekaP
```

Confidencialidade, Integridade e Autenticidade com HTTPS

O protocolo HTTP é baseado em texto e, sem uma estratégia de confidencialidade, as informações serão trafegadas como texto puro da UI para o seu sistema e nas chamadas entre os serviços. Dados como usuário, senha, cartões de crédito estariam totalmente expostos.

HTTPS é uma extensão ao HTTP que usa TLS (Transport Layer Security) para prover confidencialidade aos dados por meio de criptografia. O protocolo SSL (Security Sockets Layer) é o predecessor do TLS e está *deprecated*.

Além disso, o HTTPS provê a integridade dos dados, evitando que sejam manipulados no meio do caminho, bem como a autenticidade do servidor, garantindo que o servidor é exatamente o que o cliente espera.

A confidencialidade, integridade e autenticidade do servidor no HTTPS é atingida por meio de criptografia assimétrica (public-key cryptography). O servidor tem um par de chaves (key pair): uma pública e uma privada. Algo criptografado com a chave pública só pode ser descriptografado com a chave privada, garantindo confidencialidade. Algo criptografado com a chave privada pode ser verificado com a chave pública, validando a autenticidade.

A chave pública faz parte de um certificado digital, que é emitido por uma Autoridade Certificadora (Certificate Authority) como Comodo, Symantec, Verizon ou Let's Encrypt. Toda a infraestrutura dos certificados digitais é baseada na confiança de ambas as partes, cliente e servidor, nessas Autoridades Certificadoras.

Mas o HTTPS não é um mar de rosas: os certificados têm validade e precisam ser gerenciados. A automação do gerenciamento de certificados ainda deixa a desejar, mas tem melhorado progressivamente. Let's Encrypt sendo uma referência nessa automação.

Certificados gerados sem uma autoridade certificadora (self-signed certificates) não são confiáveis e apresentam erros em navegadores e outros sistemas.

Com o comando `keytool`, que vem com a JDK, podemos gerar um self-signed certificate:

```
keytool -genkey -alias eats -storetype JKS -keyalg RSA -keysize
```

Será solicitada uma senha e uma série de outras informações e gerado o arquivo `eats-keystore.jks`.

Podemos configurar o `application.properties` de uma aplicação Spring Boot da seguinte maneira:

```
server.port=8443  
server.ssl.key-store=eats-keystore.jks  
server.ssl.key-store-password=a-senha-escolhida  
server.ssl.keyAlias=eats
```

Mutual Authentication

Um outro detalhe do HTTPS é que não há garantias da autenticidade do cliente, apenas do servidor.

Para garantir a autenticidade do cliente e do servidor, podemos fazer com que ambos tenham certificados digitais. Quando o cliente é um navegador, isso não é possível porque é inviável exigir a cada um dos usuários a instalação de um certificado. Por isso, o uso mútuo de certificados é comumente usado na comunicação servidor-servidor.

Cada serviço deve ter dois *stores* com chaves criptográficas, que possuem a extensão `.jks` na plataforma Java:

- uma *key store*, que contém a chave privada de um determinado serviço, além de um certificado com a respectiva chave pública
- uma *trust store*, que contém os certificados com chaves públicas dos clientes e servidores ou de Autoridades Certificadoras considerados confiáveis

O `application.properties` deve ter configurações tanto do key store como do trust store, além da propriedade `server.ssl.client-auth` que indica o uso de autenticação mútua e pode ter os valores `none`, `want` (não

obrigatório) e `need` (obrigatório).

```
server.ssl.key-store=eats-keystore.jks  
server.ssl.key-store-password=a-senha-escolhida  
server.ssl.keyAlias=eats
```

```
server.ssl.trust-store=eats-truststore.jks  
server.ssl.trust-store-password=senha-do-trust-store  
  
server.ssl.client-auth=need
```

Protegendo dados armazenados

Mesmo investindo esforço em proteger a rede, a comunicação entre os serviços (*data at transit*) e os serviços em si, é preciso preparar nosso ambiente para uma possível invasão.

Uma vulnerabilidade está nos dados armazenados (*data at rest*) em BDs, arquivos de configuração e backups. Em especial, devemos proteger dados sensíveis como cartões de crédito, senhas e chaves criptográficas. Muitos ataques importantes exploraram a falta de criptografia de dados armazenados ou falhas nos algoritmos criptográficos utilizados.

Em seu livro *Building Microservices*, Sam Newman indica algumas medidas que devem ser tomadas para proteger os dados armazenados:

- use implementações padrão de algoritmos criptográficos conhecidos, ficando atento a possíveis vulnerabilidades e aplicando *patches* regularmente. Não tente criar o seu algoritmo. Para senhas, use Strings randômicas (salts) que minimizam ataques baseados em tabelas de hashes.
- limite a encriptação a tabelas dos BDs e a arquivos que realmente são sensíveis para evitar impactos negativos na performance da aplicação

- criptografe os dados sensíveis logo que entram no sistema, descriptografe sob demanda e assegure que os dados não são armazenados em outros lugares
- assegure que os backups estejam criptografados
- armazene as chaves criptográficas em um software ou appliance (hardware) específico para gerenciamento de chaves.

Rotação de credenciais

Em Junho de 2014, a Code Spaces, uma concorrente do GitHub que fornecia Git e SVN na nuvem, sofreu um ataque em que o invasor, após chantagem, apagou quase todos os dados, configurações de máquinas e backups da empresa. O ataque levou a empresa à falência! Isso aconteceu porque o invasor teve acesso ao painel de controle do AWS e conseguiu apagar quase todos os artefatos, incluindo os backups.

Não se sabe ao certo como o invasor conseguiu o acesso indevido ao painel de controle do AWS, mas há a hipótese de que obteve as credenciais de acesso de um antigo funcionário da empresa.

É imprescindível que as credenciais tenham acesso limitado, minimizando o potencial de destruição de um possível invasor.

Outra coisa importante é que as senhas dos usuários, chaves criptográficas, API keys e outras credenciais sejam modificadas de tempos em tempos. Assim, ataques feitos com a ajuda funcionários desonestos terão efeito limitado. Se possível, essa **rotação de credenciais** deve ser feita de maneira automatizada.

Há alguns softwares que automatizam o gerenciamento de credenciais:

- Vault, da HashiCorp
- AWS Secrets Manager
- KeyWiz, da Square
- CredHyb, da Cloud Foundry

Um outro aspecto do caso da Code Spaces é que os backups eram

feitos no próprio AWS. É importante que tenhamos offsite backups, em caso de comprometimento de um provedor de cloud computing.

Vault

Vault é uma solução de gerenciamento de credenciais da HashiCorp, a mesma empresa que mantém o Vagrant, Consul, Terraform, entre outros.

O Vault armazena de maneira segura e controla o acesso de tokens, senhas, API Keys, chaves criptográficas, e certificados digitais. Provê uma CLI, uma API HTTP e uma UI Web para gerenciamento. É possível criar, revogar e rotacionar credenciais de maneira automatizada.

Para que a senha, por exemplo, de um BD seja alterada pelo Vault, é necessário que seja configurado um usuário do BD que possa criar e remover outros usuários.

Segue um exemplo dos comandos da CLI do Vault para criação de credenciais com duração de 1 hora no MySQL:

```
vault secrets enable mysql  
vault write mysql/config/connection connection_url="root:root@  
vault write mysql/config/lease lease=1h lease_max=24h  
vault write mysql/roles/readonly sql="CREATE USER '{{name}}'@'  
[REDACTED]
```

As credenciais dos backends precisam ser conhecidas pelo Vault. No caso do MySQL, o usuário `root` e a respectiva senha precisam ser conhecidos. Essas configurações são armazenadas de maneira criptografada na representação interna do Vault. O Vault pode usar para armazenamento Consul, Etcd, o sistema de arquivos, entre diversos outros mecanismos.

Os dados do Vault são criptografados com uma chave simétrica. Essa chave simétrica é criptografada com uma *master key*. E a *master key* é criptografada usando o algoritmo *Shamir's secret sharing*, em que mais

de uma chave é necessária para descriptografar os dados. Por padrão, o Vault usa 5 chaves ao todo, sendo 3 delas necessárias para a descriptografia.

O Spring Cloud Config Server permite o uso do Vault como repositório de configurações: <https://cloud.spring.io/spring-cloud-config/reference/html/#vault-backend>

Há ainda o Spring Cloud Vault, que provê um cliente Vault para aplicações Spring Boot: <https://cloud.spring.io/spring-cloud-vault/reference/html/>

Segurança em um Service Mesh

Conforme discutimos em capítulos anteriores, um Service Mesh como Istio ou Linkerd cuidam de várias necessidades de infraestrutura em uma Arquitetura de Microservices como resiliência, monitoramento, load balancing e service discovery.

Além dessas, um Service Mesh pode cuidar de necessidades de segurança como Confidencialidade, Autenticidade, Autenticação, Autorização e Auditoria. Assim, removemos a responsabilidade da segurança dos serviços e passaríamos para a infraestrutura que os conecta.

O Istio, por exemplo, provê de maneira descomplicada:

- Mutual Authentication com TLS
- gerenciamento de chaves e rotação de credenciais com o componente Citadel
- whitelists e blacklists para restringir o acesso de certos serviços
- configuração de rate limiting, afim de evitar ataques DDoS (Distributed Denial of Service)

Apêndice: Encolhendo o monólito

Desafio: extrair serviços de pedidos e de administração de restaurantes

Objetivo

Extraia os módulos `eats-pedido` e `eats-restaurante` do monólito para serviços próprios.

Escolha um mecanismo de persistência adequado, fazendo a migração, se necessária.

Minimize as dependências entre os serviços.

Não deixe de pensar em client side load balancing e self registration no Service Registry.

Em caso de necessidade, use circuit breakers e retries.

Considere o uso de eventos e mensageria.

Faça com que os novos serviços usem o Config Server e enviem informações de monitoramento.

Apêndice: Referências

ABADI, Daniel J. *Consistency Tradeoffs in Modern Distributed Database System Design*. IEEE Computer Society, Feb. 2012. Em:
<http://www.cs.umd.edu/~abadi/papers/abadi-pacelc.pdf>

AMARAL, Mario et al. *Estratégias de migração de dados no Elo7 – Hipsters On The Road #07*. 2019. Em: <https://hipsters.tech/estrategias-de-migracao-de-dados-no-elo7-hipsters-on-the-road-07/>

AMUNDSEN, Mike. *The HAL-FORMS Media Type*. 2016. Em:
<https://rwcbook.github.io/hal-forms/>

ANISHCHENKO, Igor. *PB vs. Thrift vs. Avro*. 2012. Em:
<https://pt.slideshare.net/IgorAnishchenko/pb-vs-thrift-vs-avro>

AQUILES, Alexandre. *Todo o poder emana do cliente: explorando uma API GraphQL*. 2017. Em: <https://blog.caelum.com.br/todo-o-poder-emana-do-cliente-explorando-uma-api-graphql/>

ASERG-UFMG, Applied Software Engineering Research Group. *Does Conway's Law apply to Linux?* 2017. Em:
<https://medium.com/@aserg.ufmg/does-conways-law-apply-to-linux-6acf23c1ef15>

BARR, Jeff. *Migration Complete – Amazon's Consumer Business Just Turned off its Final Oracle Database*. 2019. Em:
<https://aws.amazon.com/pt/blogs/aws/migration-complete-amazons-consumer-business-just-turned-off-its-final-oracle-database>

BELSHE, Mike et al. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. Internet Engineering Task Force (IETF), 2015. Em:
<https://tools.ietf.org/html/rfc7540>

BENEVIDES, Rafael. *Why Kubernetes is The New Application Server*. 2018. Em: <https://developers.redhat.com/blog/2018/06/28/why-kubernetes-is-the-new-application-server>

[kubernetes-is-the-new-application-server/](#)

BERNERS-LEE, Tim et al. *Uniform Resource Locators (URL)*. Internet Engineering Task Force (IETF), 1994. Em:
<https://tools.ietf.org/html/rfc1738>

BERNERS-LEE, Tim et al. *Hypertext Transfer Protocol -- HTTP/1.0*. Internet Engineering Task Force (IETF), 1996. Em:
<https://tools.ietf.org/html/rfc1945>

BERNERS-LEE, Tim et al. *Uniform Resource Identifiers (URI): Generic Syntax*. Internet Engineering Task Force (IETF), 1998. Em:
<https://tools.ietf.org/html/rfc2396>

BREWER, Eric. *Towards Robust Distributed Systems*. Principles Of Distributed Computing. 2000. Em:
http://pld.cs.luc.edu/courses/353/spr11/notes/brewer_keynote.pdf

BROOKS, Fred. *No Silver Bullet: Essence and Accident in Software Engineering*. Elsevier Science B. V. 1986. Em:
<http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/no-silver-bullet.pdf>

BROWN, Simon. *Modular monoliths*. 2015. Em:
<http://www.codingthearchitecture.com/presentations/sa2015-modular-monoliths>

BRUCE, Morgan; PEREIRA, Paulo A. *Microservices in Action*. Manning Publications, 2018. 392 p. Em:
<https://www.manning.com/books/microservices-in-action>

CALÇADO, Phil. *Evoluindo uma Arquitetura inteiramente sobre APIs: o caso da SoundCloud*. 2013. Em:
<https://www.infoq.com/br/presentations/evoluindo-uma-arquitetura-soundcloud/>

Calçado, Phil. *The Back-end for Front-end Pattern (BFF)*. 2015. Em:

https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html

CALÇADO, Phil. 2018. Em:

<https://twitter.com/pcalcado/status/963183090339385345>

CHRISTENSEN, Ben. *Optimizing the Netflix API*. 2013. Em:

<https://medium.com/netflix-techblog/optimizing-the-netflix-api-5c9ac715cf19>

CLEMSON, Toby. *Testing Strategies in a Microservice Architecture*. 2014.

Em: <https://martinfowler.com/articles/microservice-testing/>

COCKCROFT, Adrian. *Patterns for Continuous Delivery, High Availability, DevOps & Cloud Native Open Source with NetflixOSS*. 2013. Em:

<https://www.slideshare.net/adrianco/yowworkshop-131203193626phpapp01-1>

COHEN, Mickey. Zuul @ Netflix. SpringOne Platform, 2016. Em:

<https://www.slideshare.net/MickeyCohen1/zuul-netflix-springone-platform>

CONWAY, Melvin. *How Do Committees Invent?* Datamation, Abr. 1968.

Em: http://www.melconway.com/Home/Committees_Paper.html

COSTA, Luiz et al. *Monolitos modulares e vacinas – Hipsters On The Road #17*. 2019. Em: <https://hipsters.tech/monolitos-autonomos-e-vacinas-hipsters-on-the-road-17/>

CNCF TOC (Techinical Oversight Committee). *CNCF Cloud Native Definition v1.0*. 2018. Em:

<https://github.com/cncf/toc/blob/master/DEFINITION.md>

DEAN, Jeffrey. *Designs, Lessons and Advice from Building Large Distributed Systems*. 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware. 2009. Em:

<http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf>

DUSSEAUT, Lisa; SNELL James M. *PATCH Method for HTTP*. Internet Engineering Task Force (IETF), 2010. Em:

<https://tools.ietf.org/html/rfc5789>

EVANS, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional. 2003. 560 p. Em:

<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215>

FEATHERS, Michael. *The Deep Synergy Between Testability and Good Design* 2007. Em:

https://michaelfeathers.typepad.com/michael_feathers_blog/2007/09/the-deep-synerg.html

FEATHERS, Michael. *Microservices Until Macro Complexity*. 2014. Em:

<https://michaelfeathers.silvrback.com/microservices-until-macro-complexity>

FERREIRA, Rodrigo. *REST: Princípios e boas práticas*. 2017. Em:

<https://blog.caelum.com.br/rest-principios-e-boas-praticas>

FIELDING, Roy. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. Em:

<https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

FIELDING, Roy. *REST APIs must be hypertext-driven*. 2008. Em:

<https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>

FOOTE, Brian; YODER, Joseph. *Big Ball Of Mud*. 1999. Em:

<http://www.laputan.org/mud/mud.html>

FOWLER, Martin et al. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999. 464 p. Em:

<https://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672/>

FOWLER, Martin. *Patterns of Enterprise Application Architecture*.

Addison-Wesley Professional, 2002. 560 p. Em:

<https://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420>

FOWLER, Martin. *StranglerFigApplication*. 2004. Em:

<https://martinfowler.com/bliki/StranglerFigApplication.html>

FOWLER, Martin. *EventSourcing*. 2005. Em:

<https://martinfowler.com/eaaDev/EventSourcing.html>

FOWLER, Martin. *Detestable*. 2005b. Em:

<https://martinfowler.com/bliki/Detestable.html>

FOWLER, Martin. *TestDouble*. 2006. Em:

<https://www.martinfowler.com/bliki/TestDouble.html>

FOWLER, Martin. *Richardson Maturity Model*. 2010. Em:

<https://martinfowler.com/articles/richardsonMaturityModel.html>

FOWLER, Martin. CQRS. 2011. Em:

<https://martinfowler.com/bliki/CQRS.html>

FOWLER, Martin. *BroadStackTest*. 2013. Em:

<https://www.martinfowler.com/bliki/BroadStackTest.html>

FOWLER, Martin. *ThresholdTest*. 2013b. Em:

<https://www.martinfowler.com/bliki/ThresholdTest.html>

FOWLER, Martin; LEWIS, James. *Microservices: a definition of this new technical term*. 2014. Em:

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

FOWLER, Martin. *Microservices and the First Law of Distributed Objects*. 2014a. Em: <https://martinfowler.com/articles/distributed-objects-microservices.html>

FOWLER, Martin. *MicroservicePrerequisites*. 2014b. Em:

<https://martinfowler.com/bliki/MicroservicePrerequisites.html>

FOWLER, Martin. *UnitTest*. 2014c. Em:
<https://martinfowler.com/bliki/UnitTest.html>

FOWLER, Martin. *Microservice Trade-Offs*. 2015a. Em:
<https://martinfowler.com/articles/microservice-trade-offs.html>

FOWLER, Martin. *MicroservicePremium*. 2015b. Em:
<https://martinfowler.com/bliki/MicroservicePremium.html>

FOWLER, Martin. *MonolithFirst*. 2015c. Em:
<https://martinfowler.com/bliki/MonolithFirst.html>

FOWLER, Martin. *Integration Test*. 2018. Em:
<https://martinfowler.com/bliki/IntegrationTest.html>

FREEMAN, Steve et al. *Mock Roles, not Objects*. 2004. Em:
<http://jmock.org/oopsla2004.pdf>

GEERS, Michael. *Micro Frontends*. 2017. Em: <https://micro-frontends.org/>

GONIGBERG, Arthur. *Zuul's Journey to Non-Blocking*. Strange Loop, 2017. Em: https://www.youtube.com/watch?v=2oXqbLhMS_A

GRAY, Jim; REUTER, Andreas. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992. 1070 p. Em:
<https://books.google.com.br/books?id=VFKbCgAAQBAJ&pg=PA7&dq=marriage+two+phase+commit>

GRIGORIK, Ilya. *High Performance Browser Networking*. O'Reilly, 2013. 400 p. Em: <https://hpbn.co/>

HOHPE, Gregor; WOOLF, Bobby. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003. 741 p. Em:
<https://www.amazon.com.br/Enterprise-Integration-Patterns-Designing-Deploying/dp/0321200683>

JACOBSON, Ivar. *Object Oriented Software Engineering: A Use-Case*

Driven Approach. Addison-Wesley Professional, 1992. 552 p. Em:
<https://www.amazon.com/Object-Oriented-Software-Engineering-Approach/dp/0201544350>

JOSUTTIS, Nicholai. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, 2007. 344 p. Em:
<https://www.amazon.com.br/SOA-Practice-Distributed-System-Design/dp/0596529554>

KELLY, Mike. *JSON Hypertext Application Language*. Internet Engineering Task Force (IETF) Internet-Draft, 2012. Em:
<https://tools.ietf.org/html/draft-kelly-json-hal-00>

KNOERNSCHILD, Kirk. *Java Application Architecture: Modularity Patterns with Examples Using OSGi*. Prentice Hall, 2012. 384 p. Em:
<https://www.amazon.com.br/Java-Application-Architecture-Modularity-Patterns/dp/0321247132>.

KRIENS, Peter. μ Services. 2010. Em:
<https://blog.osgi.org/2010/03/services.html>

LAMPORT, Leslie. *distribution*. 1987. Em:
<https://lamport.azurewebsites.net/pubs/distributed-system.txt>

LEWIS, James. *Episode 213: James Lewis on Microservices*. 2014. Em:
<https://www.se-radio.net/2014/10/episode-213-james-lewis-on-microservices/>

MACCORMACK, Alan et al. *Exploring the Duality between Product and Organizational Architectures: A Test of the "Mirroring" Hypothesis*. 2008. Em: https://www.hbs.edu/faculty/Publication%20Files/08-039_1861e507-1dc1-4602-85b8-90d71559d85b.pdf

MARTIN, Robert Cecil. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009. 464 p. Em:
<https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882>

MARTIN, Robert Cecil. *Screaming Architecture*. 2011. Em:
<https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html>

MARTIN, Robert Cecil. *Microservices and Jars*. 2014. Em:
<https://blog.cleancoder.com/uncle-bob/2014/09/19/MicroServicesAndJars.html>.

MARTIN, Robert Cecil. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017. 432 p. Em:
<https://www.amazon.com/Clean-Architecture-Craftsmans-Software-Structure/dp/0134494164>.

MASON, Ross. *OSGi? No thanks*. 2010. Em:
<https://blogs.mulesoft.com/dev/news-dev/osgi-no-thanks/>

NELSON, Teodor Holm. *Complex Information Processing: A File Structure for the Complex, the Changing, and the Indeterminate*. Association for Computing Machinery (ACM) National Conference. 1965. Em: <https://elmcip.net/node/7367>

NEWMAN, Sam. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015. 280 p. Em:
<https://www.oreilly.com/library/view/building-microservices/9781491950340/>

NEWMAN, Sam. *Pattern: Backends For Frontends*. 2015b. Em:
<https://samnewman.io/patterns/architectural/bff/>

NEWMAN, Sam. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly, 2019. Early Release. 284 p. Em:
<https://learning.oreilly.com/library/view/monolith-to-microservices/9781492047834/>

NEWMAN, Sam. 2019b. Em:
<https://twitter.com/samnewman/status/1131117455907205120>

NYGARD, Michael. *Release It! Second Edition: Design and Deploy Production-Ready Software*. Pragmatic Programmers, 2018. 376 p. Em: <https://pragprog.com/book/mnne2/release-it-second-edition>

NOTTINGHAM, Mark. *Web Linking*. Internet Engineering Task Force (IETF), 2010. <https://tools.ietf.org/html/rfc5988>

PARLOG, Nicolai. *The Java Module System*. Manning Publications, 2018. Manning Early Access Program, version 10. 503 p. Em: <https://www.manning.com/books/the-java-module-system>

PONTE, Rafael. 2019. Em:

<https://twitter.com/rponte/status/1186337012724441089>

RICHARDS, Mark. *Microservices AntiPatterns and Pitfalls*. O'Reilly, 2016. 66 p. Em: <https://learning.oreilly.com/library/view/microservices-antipatterns-and/9781492042716/>

RICHARDSON, Chris. *Microservice Patterns*. Manning Publications, 2018a. 520 p. Em: <https://www.manning.com/books/microservices-patterns>

RICHARDSON, Chris. *Pattern: Shared Database*. 2018b. Em: <https://microservices.io/patterns/data/shared-database.html>

RICHARDSON, Chris. *Pattern: Database per Service*. 2018c. Em: <https://microservices.io/patterns/data/database-per-service.html>

RICHARDSON, Leonard; RUBY, Sam. *RESTful Web Services: Web Services for the Real World*. O'Reilly, 2007. 448 p. Em: <https://learning.oreilly.com/library/view/restful-web-services/9780596529260/>

RICHARDSON, Leonard. *Justice Will Take Us Millions Of Intricate Moves - Act Three: The Maturity Heuristic*. 2008. Em: <https://www.crummy.com/writing/speaking/2008-QCon/act3.html>

ROBINSON, Ian. *Getting Things Done with REST*. QCon London 2011. Em:

<https://www.infoq.com/presentations/Getting-Things-Done-with-REST>

RUBEY, Raymond J. *Pasta Theory of Software*. 1992. Em:

<https://www.gnu.org/fun/jokes/pasta.code.html>

SADALAGE, Pramod; AMBER, Scott. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006. 384 p. Em:
<https://learning.oreilly.com/library/view/refactoring-databases-evolutionary/0321293533/>

SARDINHA, Renato. *Introdução ao Change Data Capture (CDC)*. 2019. Em: <https://elo7.dev/cdc-parte-1/>

SIMONS, Matthew; LEROY, Jonny. *Contending with Creaky Platforms* C/O. Cutter IT Journal, Dec. 2010. Em:
<http://jonnyleroy.com/2011/02/03/dealing-with-creaky-legacy-platforms/>

SPOLSKY, Joel. *Things You Should Never Do, Part I*. 2000. Em:
<https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/>

SRINIVAS, Prashanth Nuggehalli. *Strangler fig inside* (licença GFDL) 2010. Em: https://en.wikipedia.org/wiki/File:Strangler_fig_inside.jpg

TALWAR, Varun. *gRPC: a true internet-scale RPC framework is now 1.0 and ready for production deployments*. 2016. Em:
<https://cloud.google.com/blog/products/gcp/grpc-a-true-internet-scale-rpc-framework-is-now-1-and-ready-for-production-deployments>

TALEB, Nassim. *Antifragile: Things That Gain from Disorder*. Random House, 2012. 519 p. Em: <https://www.amazon.com.br/Antifragile-Things-That-Gain-Disorder/dp/1400067820/>

TILKOV, Stefan. *Don't start with a monolith: ... when your goal is a microservices architecture*. 2015. Em:

<https://martinfowler.com/articles/dont-start-monolith.html>

VERNON, Vaughn. *Implementing Domain-Driven Design*. Addison-

Wesley Professional, 2013. 656 p. Em:

<https://www.amazon.com.br/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577>

VERNON, Vaughn. *Domain-Driven Design Distilled*. Addison-Wesley Professional, 2016. 176 p. Em: <https://www.amazon.com.br/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420>

VOCKE, Ham. *The Practical Test Pyramid*. 2018. Em:

<https://martinfowler.com/articles/practical-test-pyramid.html>

VOEGELS, Werner. *A Conversation with Werner Voegels*. ACM Queue, Mai. 2006. Em: <https://dl.acm.org/doi/pdf/10.1145/1142055.1142065?download=true>

SIMONS, Matthew; LEROY, Jonny. *Contending with Creaky Platforms* C/O. Cutter IT Journal, Dec. 2010. Em:

<http://jonnyleroy.com/2011/02/03/dealing-with-creaky-legacy-platforms/>

WALLS, Craig. *Modular Java: Creating Flexible Applications with OSGi and Spring*. The Pragmatic Bookshelf, 2009. 260 p. Em:

<https://pragprog.com/book/cwosg/modular-java>

WESTEINDE, Kirsten. *Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity*. 2019. Em:

<https://engineering.shopify.com/blogs/engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity>

WIGGINS, Adam. *The Twelve-Factor App*. 2011. Em:

<https://www.12factor.net>

WINTON, David. *Code Rush*. 2000. Em: <https://www.youtube.com/watch?v=4Q7FTjhvZ7Y>

Join GitHub today

[Dismiss](#)

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)

Branch: master ▾

[apostila-fj33](#) / README.md

[Find file](#)

[Copy path](#)



alexandreaquiles Adicionando README e licença

34bc7d2 on 25 Oct 2019

1 contributor

9 lines (5 sloc) 419 Bytes

[Raw](#)

[Blame](#)

[History](#)



Apostila do curso Microservices com Spring Cloud da Caelum

Para mais informações sobre o curso, acesse:

<https://www.caelum.com.br/curso-microservices-spring-cloud>

Licença

Esta obra está licenciada com uma Licença [Creative Commons Atribuição-NãoComercial-SemDerivações 4.0 Internacional](#).

