

Curso Microservices com Spring Cloud

Curso FJ-33



 **caelum**
ensino e inovação

Apostila gerada especialmente para Daniel Henrique da Silva Pereira Lima - danielhenriquelima@gmail.com



Conheça também:



alura

alura.com.br



Casa do Código
Livros e Tecnologia

casadocodigo.com.br

Blog da Caelum



blog.caelum.com.br

Newsletter



caelum.com.br/newsletter

Facebook



facebook.com.br/caelumbr

Twitter



twitter.com/caelum

Sumário

1 Conhecendo o Caelum Eats	1
1.1 Peça sua comida com o Caelum Eats	1
1.2 Funcionalidades	1
1.3 A Arquitetura do Caelum Eats	13
1.4 Exercício: Executando o back-end	14
1.5 Exercício: Executando o front-end	15
1.6 Um negócio em expansão	15
2 Decompondo o monólito	17
2.1 Da bagunça a camadas	17
2.2 A Arquitetura que Grita	18
2.3 Por arquiteturas centradas no domínio	19
2.4 Explorando o domínio	20
2.5 Caçando entidades importantes	21
2.6 Das estruturas de comunicação para o código	23
2.7 A linguagem do negócio depende do Contexto	24
2.8 Bounded Contexts no Caelum Eats	25
2.9 Nem tudo precisa ser público	26
2.10 Exercício opcional: decomposição em pacotes	27
2.11 Módulos	28
2.12 Módulos Maven	29
2.13 Pragmatismo e (um pouco de) Duplicação	31
2.14 Exercício: o monólito decomposto em módulos Maven	32
2.15 O Monólito Modular	33
2.16 O que é um monólito, afinal?	34
2.17 Para saber mais: Limitações dos Módulos Maven e JARs	37
2.18 Para saber mais: JPMS, um sistema de módulos para o Java	39
2.19 Para saber mais: Módulos plugáveis	40
2.20 Para saber mais: OSGi	42

3 Extraindo serviços	44
3.1 O contexto que levou aos Microservices	44
3.2 Do monólito (modular) aos Microservices	45
3.3 Prós e contras de uma Arquitetura de Microservices	47
3.4 Quão micro deve ser um Microservice?	55
3.5 Microservices e SOA	57
3.6 Microservices e a Cloud	58
3.7 Microservices chassis	60
3.8 Decidindo por uma Arquitetura de Microservices no Caelum Eats	61
3.9 Como falhar numa migração: o Big Bang Rewrite	63
3.10 Estrangulando o monólito	63
3.11 Extraindo serviços do Monólito do Caelum Eats	68
3.12 Quebrando o Domínio	69
3.13 Criando um Microservice de Pagamentos	71
3.14 Exercício: Executando o novo serviço de pagamentos	78
3.15 Criando um Microservice de distância	80
3.16 Exercício: Executando o novo serviço de distância	86
4 Migrando dados	89
4.1 Banco de Dados Compartilhado: uma boa ideia?	89
4.2 Um Banco de Dados por serviço	91
4.3 Bancos de Dados separados no Caelum Eats	92
4.4 Criando uma nova instância do MySQL a partir do Docker	93
4.5 Criando uma instância do MongoDB a partir do Docker	95
4.6 Simplificando o gerenciamento dos containers com Docker Compose	96
4.7 Exercício: Gerenciando containers de infraestrutura com Docker Compose	97
4.8 Separando Schemas	98
4.9 Separando schema do BD de pagamentos do monólito	99
4.10 Exercício: migrando dados de pagamento para schema separado	103
4.11 Migrando dados de um servidor MySQL para outro	104
4.12 Exercício: migrando dados de pagamento para um servidor MySQL específico	104
4.13 Apontando serviço de pagamentos para o BD específico	107
4.14 Exercício: fazendo serviço de pagamentos apontar para o BD específico	107
4.15 Migrando dados do MySQL para MongoDB	108
4.16 Exercício: migrando dados de restaurantes do MySQL para o MongoDB	109
4.17 Configurando MongoDB no serviço de distância	112
4.18 Exercício: Migrando dados de distância para o MongoDB	115
4.19 Para saber mais: Change Data Capture	115

5 Integração síncrona (e RESTful)	118
5.1 Em busca das funcionalidades perdidas	118
5.2 Integrando sistemas com o protocolo da Web	119
5.3 REST, o estilo arquitetural da Web	125
5.4 O Modelo de Maturidade de Richardson	127
5.5 Cliente REST com RestTemplate do Spring	130
5.6 Exercício: Integrando o módulo de restaurantes ao serviço de distância com RestTemplate	135
5.7 Cliente REST declarativo com Feign	136
5.8 Exercício: Integrando o serviço de pagamentos e o módulo de pedidos com Feign	139
5.9 O poder dos Links	139
5.10 Exercício opcional: Spring HATEOAS e HAL	145
5.11 HATEOAS e Métodos HTTP	148
5.12 Exercício opcional: Estendendo o Spring HATEOAS	150
5.13 Para saber mais: HAL-FORMS	152
5.14 Para saber mais: Spring Data REST	153
5.15 Formatos e Protocolos Binários	157
5.16 HTTP/2	159
5.17 gRPC	161
5.18 Exercício opcional: implementando um serviço de Recomendações com gRPC	168
5.19 Para saber mais: Todo o poder emana do cliente - explorando uma API GraphQL	177
5.20 Para saber mais: Field Selectors	185
6 API Gateway	187
6.1 Implementando um API Gateway com Zuul	187
6.2 Fazendo a UI usar o API Gateway	188
6.3 Exercício: API Gateway com Zuul	189
6.4 Desabilitando a remoção de cabeçalhos sensíveis no Zuul	190
6.5 Exercício: cabeçalhos sensíveis no Zuul	191
6.6 Invocando o serviço de distância a partir do API Gateway com RestTemplate	191
6.7 Invocando o monólito a partir do API Gateway com Feign	192
6.8 Compondo chamadas no API Gateway	194
6.9 Chamando a composição do API Gateway a partir da UI	195
6.10 Exercício: API Composition no API Gateway	197
6.11 LocationRewriteFilter no Zuul para além de redirecionamentos	198
6.12 Exercício: Customizando o LocationRewriteFilter do Zuul	199
6.13 Exercício opcional: um ZuulFilter de Rate Limiting	200
7 Client Side Load Balancing com Ribbon	203
7.1 Detalhando o log de requests do serviço de distância	203

Sumário	Caelum
7.2 Exercício: executando uma segunda instância do serviço de distância	204
7.3 Client side load balancing no RestTemplate do monólito com Ribbon	205
7.4 Client side load balancing no RestTemplate do API Gateway com Ribbon	206
7.5 Exercício: Client side load balancing no RestTemplate do monólito com Ribbon	207
7.6 Exercício: executando uma segunda instância do monólito	208
7.7 Client side load balancing no Feign do serviço de pagamentos com Ribbon	209
7.8 Client side load balancing no Feign do API Gateway com Ribbon	210
7.9 Exercício: Client side load balancing no Feign com Ribbon	210
8 Service Discovery	211
8.1 Implementando um Service Registry com o Eureka	211
8.2 Exercício: executando o Service Registry	212
8.3 Self Registration do serviço de distância no Eureka Server	212
8.4 Self Registration do serviço de pagamento no Eureka Server	214
8.5 Self Registration do monólito no Eureka Server	215
8.6 Self registration do API Gateway no Eureka Server	215
8.7 Exercício: Sself registration no Eureka Server	216
8.8 Client side discovery no serviço de pagamentos	217
8.9 Client side discovery no API Gateway	218
8.10 Client side discovery no monólito	218
8.11 Exercício: Client Side Discovery com Eureka Client	219
9 Resiliência	221
9.1 Exercício: simulando demora no serviço de distância	221
9.2 Circuit Breaker com Hystrix	222
9.3 Exercício: Circuit Breaker com Hystrix	223
9.4 Fallback no @HystrixCommand	223
9.5 Exercício: Fallback com Hystrix	225
9.6 Exercício: Removendo simulação de demora do serviço de distância	226
9.7 Exercício: Simulando demora no monólito	226
9.8 Circuit Breaker com Hystrix no Feign	227
9.9 Exercício: Integração entre Hystrix e Feign	227
9.10 Fallback com Feign	228
9.11 Exercício: Fallback com Feign	229
9.12 Exercício: Removendo simulação de demora do monólito	229
9.13 Exercício: Forçando uma exceção no serviço de distância	230
9.14 Tentando novamente com Spring Retry	231
9.15 Exercício: Spring Retry	232
9.16 Exponential Backoff	233

9.17 Exercício: Exponential Backoff com Spring Retry	234
9.18 Exercício: Removendo exceção forçada do serviço de distância	234
10 Mensageria e Eventos	236
10.1 Exercício: um serviço de nota fiscal	236
10.2 Exercício: configurando o RabbitMQ no Docker	237
10.3 Publicando um evento de pagamento confirmado com Spring Cloud Stream	237
10.4 Recebendo eventos de pagamentos confirmados com Spring Cloud Stream	240
10.5 Exercício: Evento de Pagamento Confirmado com Spring Cloud Stream	241
10.6 Consumer Groups do Spring Cloud Stream	243
10.7 Exercício: Competing Consumers e Durable Subscriber com Consumer Groups	243
10.8 Configurações de WebSocket para o API Gateway	245
10.9 Publicando evento de atualização de pedido no monólito	246
10.10 Recebendo o evento de atualização de status do pedido no API Gateway	248
10.11 Exercício: notificando novos pedidos e mudança de status do pedido com WebSocket e Eventos	250
11 Contratos	251
11.1 Fornecendo stubs do contrato a partir do servidor	251
11.2 Usando stubs do contrato no cliente	255
11.3 Exercício: Contract Test para comunicação síncrona	258
11.4 Definindo um contrato no publisher	259
11.5 Verificando o contrato no subscriber	264
11.6 Exercício: Contract Test para comunicação assíncrona	266
12 External Configuration	268
12.1 Implementando um Config Server	268
12.2 Configurando Config Clients nos serviços	269
12.3 Exercício: Externalizando configurações para o Config Server	270
12.4 Git como backend do Config Server	271
12.5 Exercício: repositório Git local no Config Server	272
12.6 Movendo configurações específicas dos serviços para o Config Server	273
12.7 Exercícios: Configurações específicas de cada serviço no Config Server	275
13 Monitoramento e Observabilidade	277
13.1 Expondo endpoints do Spring Boot Actuator	277
13.2 Exercício: Health Check API com Spring Boot Actuator	279
13.3 Configurando o Hystrix Dashboard	280
13.4 Exercício: Visualizando circuit breakers com o Hystrix Dashboard	281
13.5 Agregando dados dos circuit-breakers com Turbine	282

Sumário	Caelum
13.6 Agregando baseado em eventos com Turbine Stream	283
13.7 Exercício: Agregando Circuit Breakers com Turbine Stream	284
13.8 Exercício: configurando o Zipkin no Docker Compose	286
13.9 Enviando informações para o Zipkin com Spring Cloud Sleuth	287
13.10 Exercício: Distributed Tracing com Spring Cloud Sleuth e Zipkin	287
13.11 Spring Boot Admin	288
13.12 Exercício: Visualizando os microservices com Spring Boot Admin	289
14 Segurança	291
14.1 Extrair um serviço Administrativo do monólito	291
14.2 Exercício: um serviço Administrativo	294
14.3 Autenticação e Autorização	295
14.4 Sessões e escalabilidade	296
14.5 REST, stateless sessions e self-contained tokens	297
14.6 JWT e JWS	297
14.7 Stateless Sessions no Caelum Eats	299
14.8 Autenticação com Microservices e Single Sign On	302
14.9 Autenticação no API Gateway e Autorização nos Serviços	303
14.10 Access Token e JWT	303
14.11 Autenticação e Autorização nos Microservices do Caelum Eats	304
14.12 Autenticação no API Gateway	305
14.13 Validando o token JWT e implementando autorização no Monólito	309
14.14 Exercício: Autenticação no API Gateway e Autorização no monólito	315
14.15 Deixando de reinventar a roda com OAuth 2.0	318
14.16 Roles	319
14.17 Grant Types	320
14.18 OAuth no Caelum Eats	321
14.19 Authorization Server com Spring Security OAuth 2	322
14.20 JWT como formato de token no Spring Security OAuth 2	325
14.21 Exercício: um Authorization Server com Spring Security OAuth 2	328
14.22 Resource Server com Spring Security OAuth 2	330
14.23 Protegendo o serviço Administrativo	331
14.24 Exercício: Protegendo o serviço Administrativo com Spring Security OAuth 2	333
14.25 Protegendo serviços de infraestrutura	334
14.26 Confidencialidade, Integridade e Autenticidade com HTTPS	335
14.27 Mutual Authentication	337
14.28 Protegendo dados armazenados	337
14.29 Rotação de credenciais	338

14.30 Segurança em um Service Mesh	340
15 Apêndice: Encolhendo o monólito	341
15.1 Desafio: extrair serviços de pedidos e de administração de restaurantes	341
16 Apêndice: Referências	342

Versão: 23.7.30

CONHECENDO O CAELUM EATS

1.1 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

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

1.2 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.



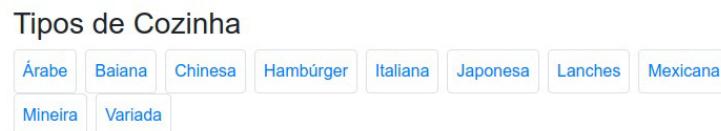
Figura 1.1: Cliente digita o CEP

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.



Restaurantes próximos a 70238-500



Figura 1.2: Cliente vê os restaurantes mais próximos

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

Figura 1.3: Cliente vê cardápio do restaurante escolhido

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

Figura 1.4: Cliente escolhe um item do cardápio

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

Figura 1.5: Cliente vê resumo do 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

Figura 1.6: Cliente digita dados pessoais e de entrega

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

Resumo do pedido	Dados do pagamento
	Forma de pagamento
1x Gyoza Bovino - 6 unidades R\$ 23,50 Sem sal	<input type="text" value="Ticket Restaurante"/>
Taxa de entrega: R\$ 6,00	Nome no cartão
Total: R\$ 29,50	<input type="text" value="JOAQUIM"/>
	Número do cartão
	<input type="text" value="8989 8098 0989 0890"/>
	Data de expiração
	<input type="text" value="February 2022"/>
	Código de Segurança
	<input type="text" value="909"/>
	Criar pagamento

Figura 1.7: Cliente informa dados de 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

Figura 1.8: Cliente confirma ou cancela o 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

Figura 1.9: Cliente acompanha o status do pedido

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

Figura 1.10: Cliente avalia o pedido

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

Figura 1.11: Dono do restaurante efetua login

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



Figura 1.12: Dono do restaurante vê 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

Figura 1.13: Dono do restaurante atualiza o cadastro

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

Figura 1.14: Dono do restaurante vê os pedidos pendentes

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:

Senha:

Confirmação da Senha:

Registrar usuário

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

Figura 1.15: Dono de um novo restaurante se registra

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:

Senha:

Logar

Figura 1.16: Administrador efetua login

Depois do login efetuado, o administrador verá o menu.



Figura 1.17: Administrador vê menu

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

Tipos de Cozinha

Tipos de Cozinha		
Nome		
Árabe	<button>Editar</button>	<button>Remover</button>
Baiana	<button>Editar</button>	<button>Remover</button>
Chinesa	<button>Editar</button>	<button>Remover</button>
Hambúrguer	<button>Editar</button>	<button>Remover</button>
Italiana	<button>Editar</button>	<button>Remover</button>

Figura 1.18: Administrador cadastra tipos de cozinha

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

Figura 1.19: Administrador cadastrava formas de pagamento

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		

Figura 1.20: Administrador aprova novo restaurante

1.3 A ARQUITETURA DO CAELUM EATS

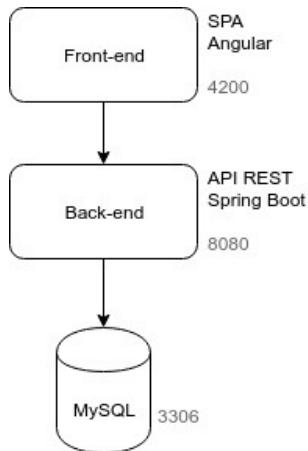


Figura 1.21: Arquitetura do Caelum Eats

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

1.4 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-monolito.git
```

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 URIs
 - os serviços e suas funcionalidades.

Saber inglês é muito importante em TI



cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

1.5 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.git
```

2. Abra um Terminal e digite:

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

3. Instale as dependências do front-end com o comando:

```
npm install
```

4. Execute a aplicação com o comando:

```
ng serve
```

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.

1.6 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

2.1 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.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

2.2 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.

2.3 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

2.4 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 é 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).

Saber inglês é muito importante em TI



cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

2.5 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.

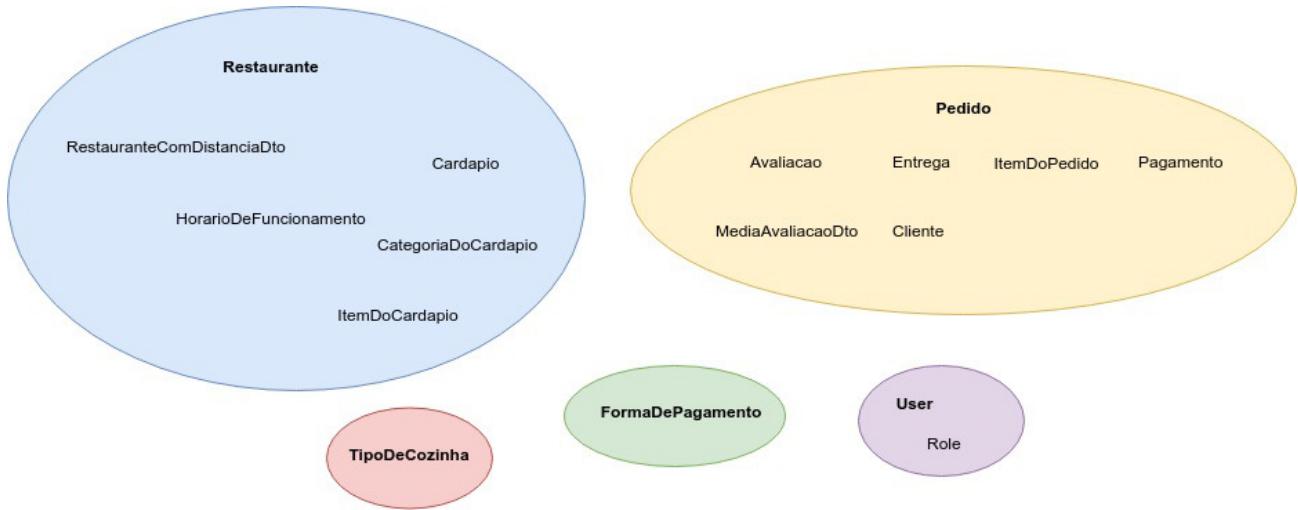


Figura 2.1: Agrupando agregados no Caelum Eats

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

2.6 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.

2.7 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 vão 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*.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

2.8 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
```

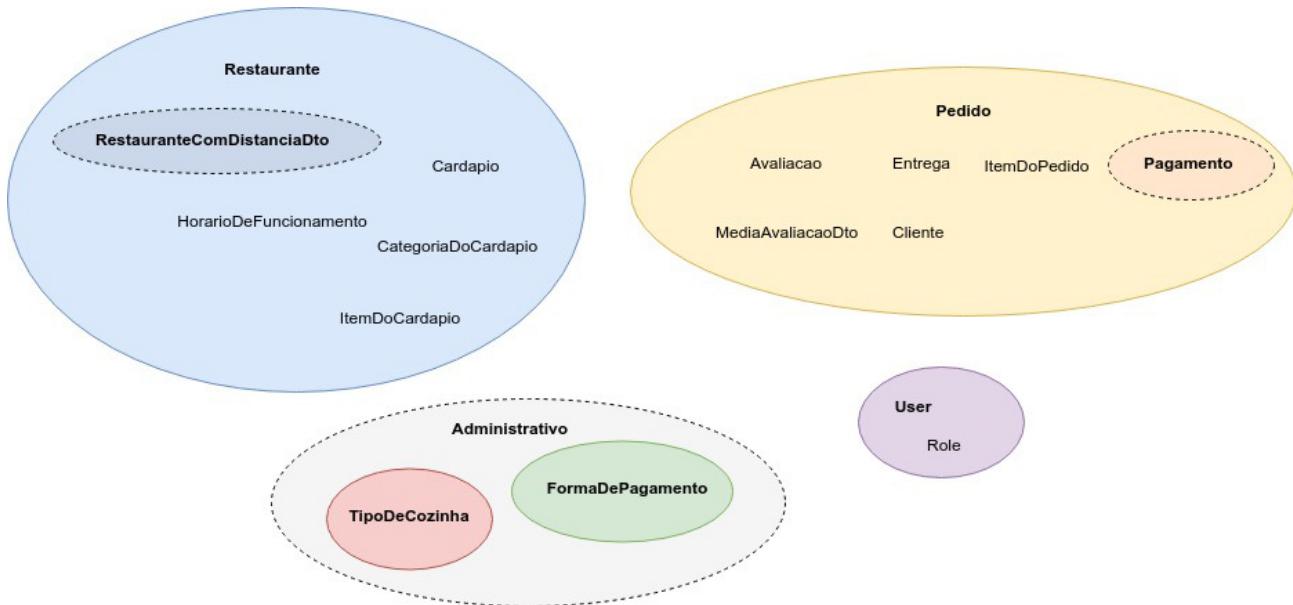


Figura 2.2: Reagrupando agregados inspirado por Bounded Contexts

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
```

2.9 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.

2.10 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-pacotes.git
```

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 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.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

2.11 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.

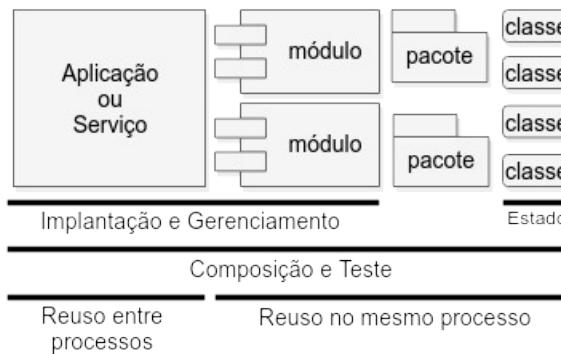


Figura 2.3: Características de módulos

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.

2.12 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:

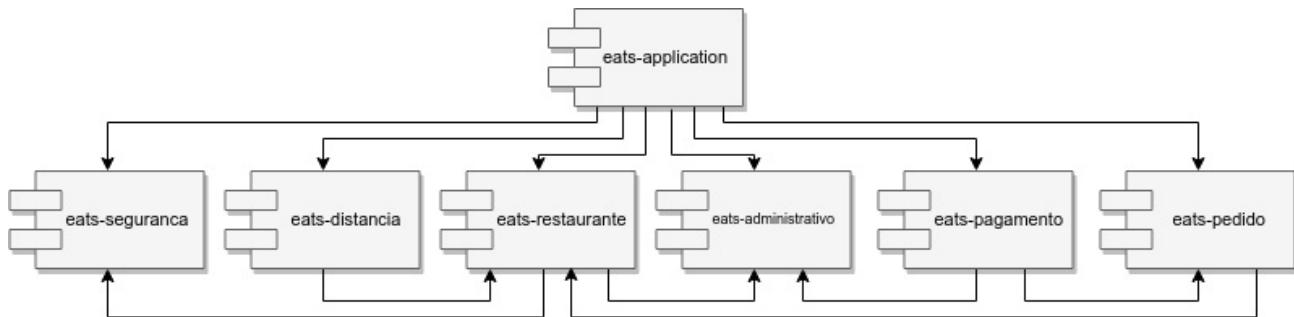


Figura 2.4: Dependências entre módulos do Caelum Eats

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.

2.13 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 evitariam̄os 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.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

2.14 EXERCÍCIO: O MONÓLITO DECOMPOSTO EM MÓDULOS MAVEN

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

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

- 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.
- Importe, pelo menu *File > Import > Existing Maven Projects* do Eclipse, o projeto `fj33-eats-monolito-modular`.
- 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.

2.15 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.

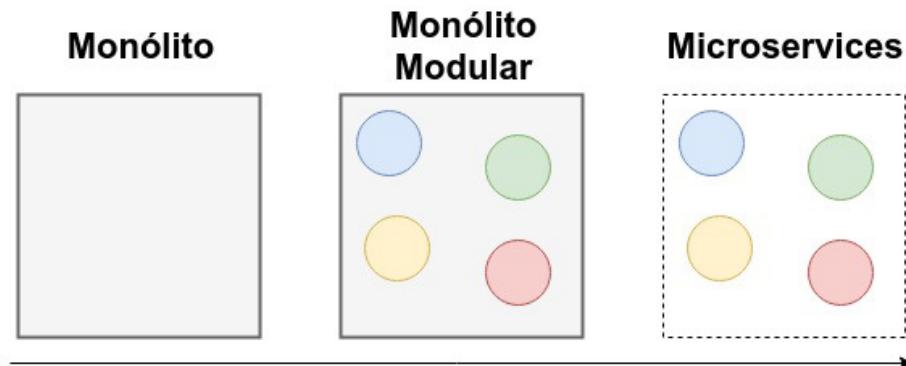


Figura 2.5: O Monólito Modular como um meio-termo

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.

2.16 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.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

2.17 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*

2.18 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.

2.19 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

fornecer 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.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

2.20 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](#) (KNOERNSCCHILD, 2012), Kirk Knoernschild usa repetidamente o termo μServices para se referir aos services OSGi.

EXTRAINDO SERVIÇOS

3.1 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 máquina físicas. O poder computacional poderia ser consumido sob-demanda, como luz ou água, permitindo que a infraestrutura de TI seja ajustada às reais necessidades, minimizando máquinas ociosas.

Em 2009, foi organizada a primeira conferência devopsdays, que unia tópicos de desenvolvimento

de software e operações de TI, cunhando o termo **DevOps**.

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

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.

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

3.2 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.

3.3 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, reaproveitaram 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 gambiaras). 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 diminui 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 FALÁCIAS 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 intraprocessos 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 desconsideram 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 ESSENIAL 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) , cunha 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.

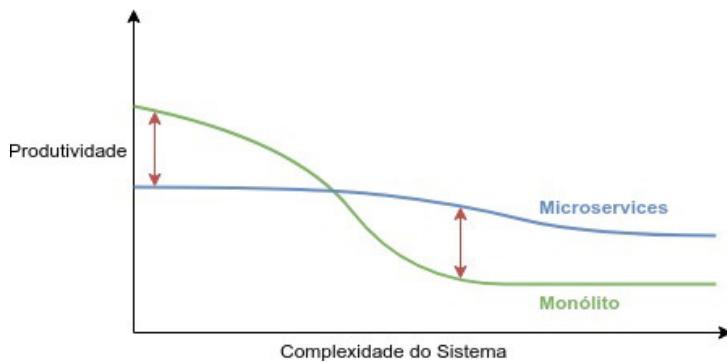


Figura 3.1: O ágio cobrado pelos Microservices em um projeto simples

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 partioná-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 construir os 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.

3.4 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.*

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

3.5 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](#) 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.

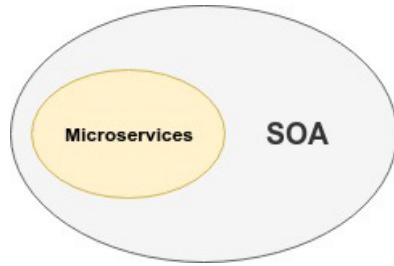


Figura 3.2: Para diversos autores, Microservices são um sabor de SOA

3.6 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 desnortinalizados (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.

3.7 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.

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.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

3.8 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

3.9 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.

3.10 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.



Figura 3.3: Figueira que matou árvore original deixando uma coluna oca, por P. N. Srinivas (SRINIVAS, 2010)

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 daquele quebra 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 extrair 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.

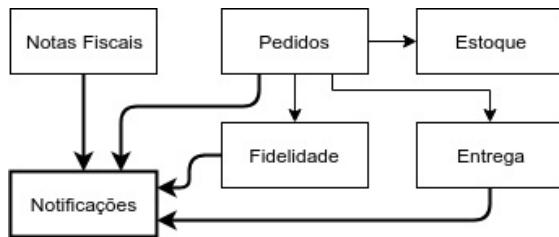


Figura 3.4: Dependências impactam na facilidade de extração

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.

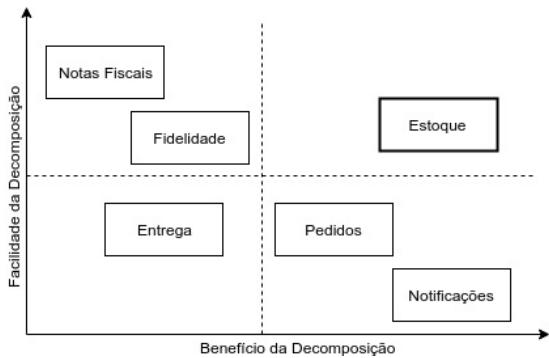


Figura 3.5: Quadrante de priorização de decomposições

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

3.11 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.

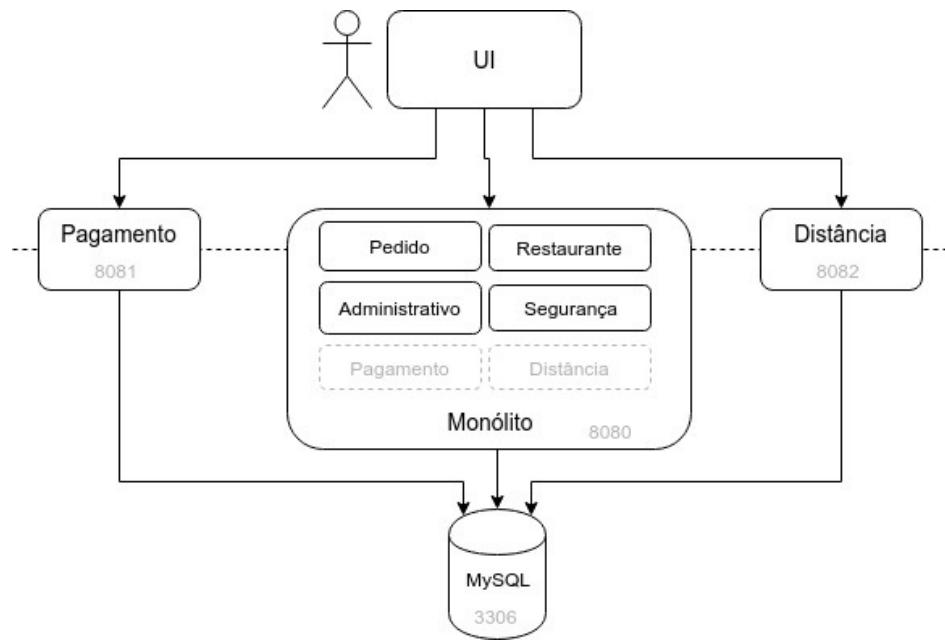


Figura 3.6: Estrangulando o Monólito do Caelum Eats

3.12 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;
}

```

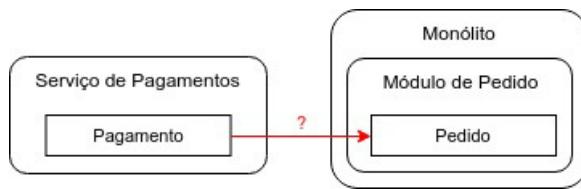


Figura 3.7: Referência a um objeto além da fronteira do serviço

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;  
}
```

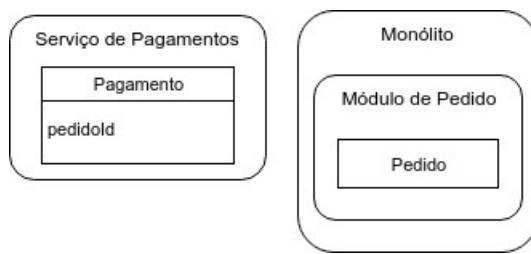


Figura 3.8: Referências a objetos além da fronteira devem ser por sua identidade

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.

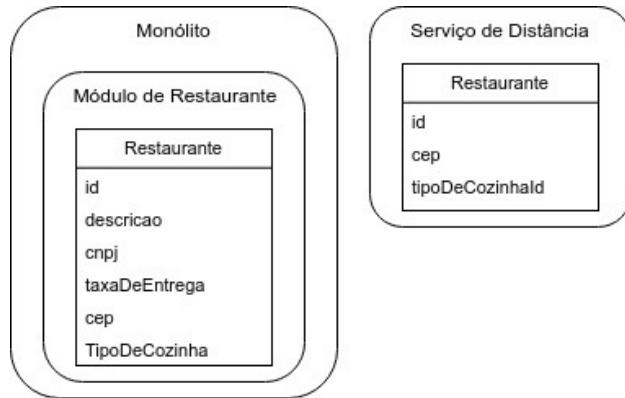


Figura 3.9: Diferentes serviços tem diferentes interesses em relação a uma Entidade

3.13 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.java

// 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(), p.getExpiracao(), p.getCodigo(), p.getStatus(),
            new FormaDePagamentoDto(p.getFormaDePagamento()),
            p.getFormaDePagamentoId(),
            new PedidoDto(p.getPedido())),
        p.getPedidoId());
    }

}
```

Remova os imports desnecessários:

```
import br.com.caelum.eats.administrativo.FormaDePagamentoDto;
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).orElseThrow(() -> new ResourceNotFoundException());
        pagamento.setStatus(Pagamento.Status.CONFIRMADO);
        pagamentoRepo.save(pagamento);
        Long pedidoId = pagamento.getPedido().getId();
        Pedido pedido = pedidos.findByIdComItens(pedidoId);
        pedido.setStatus(Pedido.Status.PAGO);
        pedidos.atualizaStatus(Pedido.Status.PAGO, pedido);
        return new PagamentoDto(pagamento);
    }
}
```

Ah! Limpe os imports:

```
import br.com.caelum.eats.pedido.Pedido;
import br.com.caelum.eats.pedido.PedidoService;
```

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.formaDePagamento.id;
    pagamento.pedidoId = pagamento.pedidoId || pagamento.pedido.id;
  }
}

```

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.navigateByUrl(`pedidos/${pagamento.pedido.id}/status`));
      .subscribe(pagamento => this.router.navigateByUrl(`pedidos/${pagamento.pedidoId}/status`));
  }
}

```

```
// 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("*").allowCredentials(true);  
    }  
}
```

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.

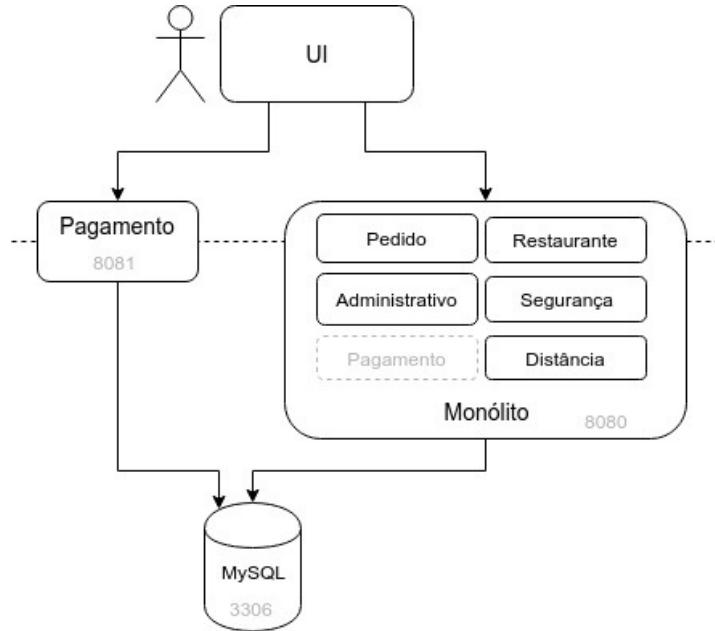


Figura 3.10: Serviço de pagamentos extraído do monólito

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

3.14 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-pagamento-service.git
```

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 2222 3333 4444", "expiracao": "2022-07", "codigo": "123", "formaDePagamentoId": 2, "pedidoId": 1 }'  
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", "codigo":"123",
  "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", "codigo":"123",
  "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.

3.15 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.

Edita 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=<SUA SENHA>

#JPA CONFIGS
```

```
spring.jpa.hibernate.ddl-auto=validate  
spring.jpa.show-sql=true  
  
spring.jackson.serialization.fail-on-empty-beans=false
```

Troque <SEU USUÁRIO> e <SUA SENHA> 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.administrativo.TipoDeCozinha;
import br.com.caelum.eats.seguranca.User;

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.administrativo.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> restaurantesDoTipoDeCozinhaMaisProximosAoCep(Long tipoDeCozin
haId, String cep) {
    TipoDeCozinha tipo = new TipoDeCozinha();
    tipo.setId(tipoDeCozinhaId);

    List<Restaurante> aprovadosDoTipoDeCozinha = restaurantes.findAllByAprovadoAndTipoDeCozinha(true, t
ipo, LIMIT).getContent();
    List<Restaurante> aprovadosDoTipoDeCozinha = restaurantes.findAllByAprovadoAndTipoDeCozinhaId(true,
tipoDeCozinhaId, LIMIT).getContent(); // modificado ...

    return calculaDistanciaParaOsRestaurantes(aprovadosDoTipoDeCozinha, cep);
}
```

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/RestauranteRepository.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 tipoDeCozinhaId, Pageable limit);

    Page<Restaurante> findAllByAprovado(boolean aprovado, Pageable limit);

}

```

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.RestauranteService;
```

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;

    @NotBlank @Size(max=18)
    private String cnpj;

    @NotBlank @Size(max=255)
    private String nome;

    @Size(max=1000)

```

```

private String descricao;
@NotBlank @Size(max=9)
private String cep;
@NotBlank @Size(max=300)
private String endereco;
@Positive
private BigDecimal taxaDeEntregaEmReais;
@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 :

```

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

export class RestauranteService {

  private API = environment.baseUrl;
  private DISTANCIA_API = environment.distanciaUrl; // adicionado

  // código omitido ...

  maisProximosPorCep(cep: string): Observable<any> {
    return this.http.get(`.${this.API}/restaurantes/mais-proximos/${cep}`);
    return this.http.get(`.${this.DISTANCIA_API}/restaurantes/mais-proximos/${cep}`); // modificado
  }

  maisProximosPorCepETipoDeCozinha(cep: string, tipoDeCozinhaId: string): Observable<any> {
    return this.http.get(`.${this.API}/restaurantes/mais-proximos/${cep}/tipos-de-cozinha/${tipoDeCozinhaId}`);
    return this.http.get(`.${this.DISTANCIA_API}/restaurantes/mais-proximos/${cep}/tipos-de-cozinha/${tipoDeCozinhaId}`); // modificado
  }

  distanciaPorCepEId(cep: string, restauranteId: string): Observable<any> {
    return this.http.get(`.${this.API}/restaurantes/${cep}/restaurante/${restauranteId}`);
    return this.http.get(`.${this.DISTANCIA_API}/restaurantes/${cep}/restaurante/${restauranteId}`); // modificado
  }

  // 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.caelum</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.

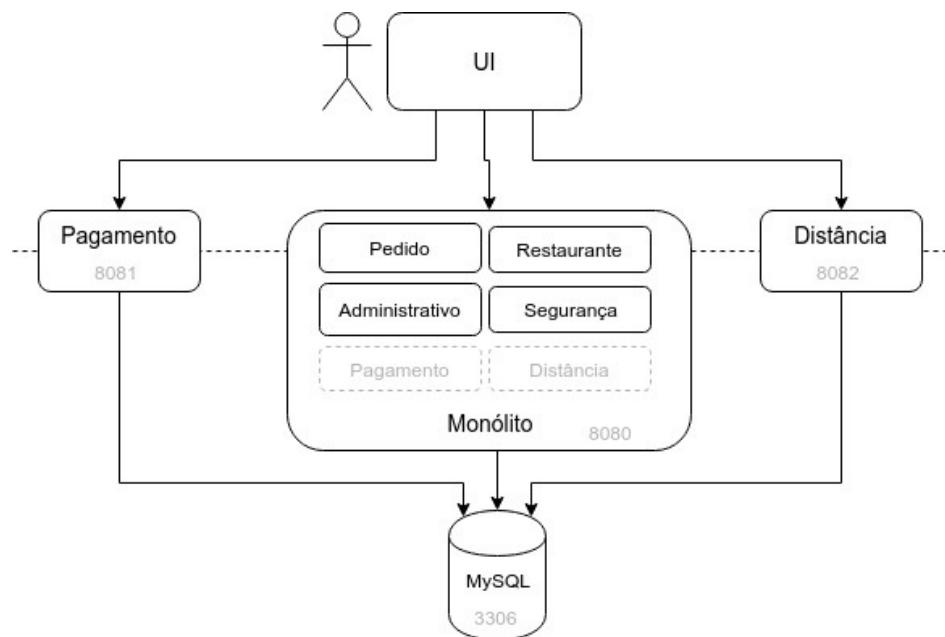


Figura 3.11: Serviço de distância extraído do monólito

3.16 EXERCÍCIO: EXECUTANDO O NOVO SERVIÇO DE DISTÂNCIA

- 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-distancia-service.git

```

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/71503510
```

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":8.357388557756333824499961338005959987640380859375},
  { "restauranteId": 2, "distancia":8.17018321127992663832628750242292881011962890625}
]
```

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/71503510/tipos-de-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":18.38244999613380059599876403835738855775633085935} ]
```

Para descobrir a distância de um dado CEP a um restaurante específico:

```
curl -i http://localhost:8082/restaurantes/71503510/restaurante/1
```

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":13.95998764038357388538244999613380055775633085935}
```

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!

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

MIGRANDO DADOS

4.1 BANCO DE DADOS COMPARTILHADO: UMA BOA IDEIA?

Mesmo depois de extrairmos os serviços de Pagamentos e Distância, mantivemos o mesmo MySQL monolítico.

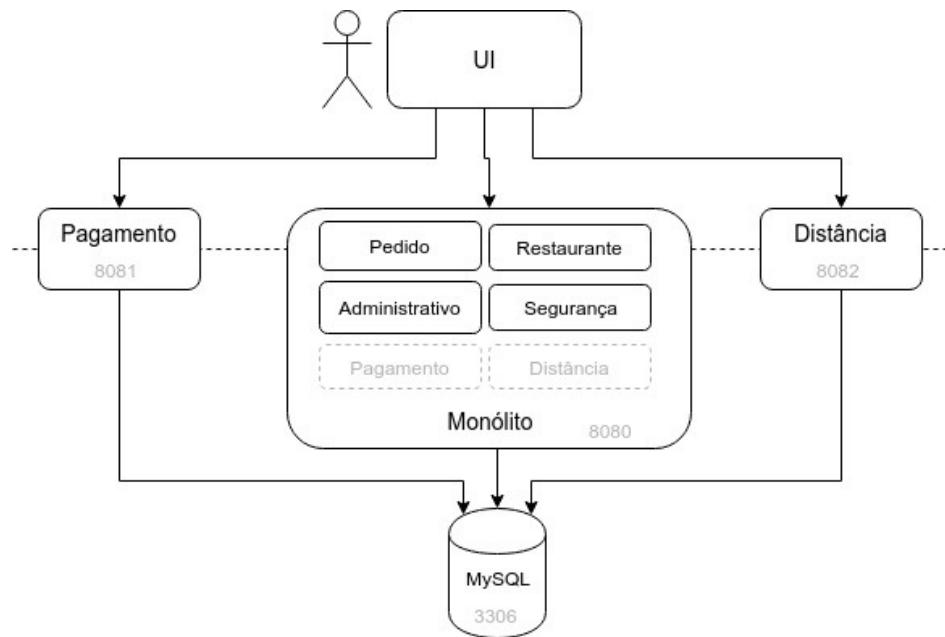


Figura 4.1: BD Compartilhado no Caelum Eats

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.

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

4.2 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.

4.3 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.

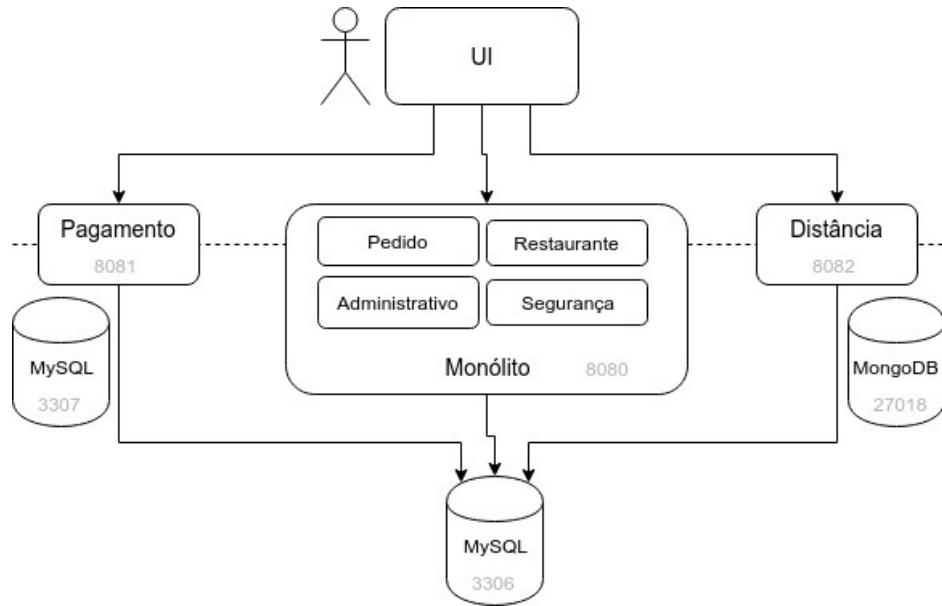


Figura 4.2: Servidores de BD separados para os serviços de Pagamentos e Distância

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.

4.4 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 MYSQL_ROOT_PASSWORD=caelum123 -e MYSQL_DATABASE=eats_pagamento -e MYSQL_USER=pagamento -e MYSQL_PASSWORD=pagamento123 mysql:5.7
```

Usamos as configurações:

- `--rm` para remover o container quando 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:

```
docker container ps
```

Deve aparecer algo como:

CONTAINER ID	CREATED	STATUS	PORTS	IMAGE	COMMAND	NAMES
183bc210a6071b46c4dd790858e07573b28cfa6394a7017cb9fa6d4c9af71563	16 minutes ago	Up 16 minutes	33060/tcp, 0.0.0.0:3307->3306/tcp	mysql:5.7	"docker-entrypoint.sh mysqld"	eats.mysql

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_pagamento
```

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:

```
show databases;
```

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
```

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

4.5 CRIANDO UMA INSTÂNCIA DO MONGODB A PARTIR DO DOCKER

Baixe a imagem do MongoDB 3.6 com o comando a seguir:

```
docker container 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 mongo:3.6
```

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.

4.6 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:

```
docker-compose up -d
```

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:

```
docker-compose logs
```

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:

```
docker-compose stop
```

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.*

4.7 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 `.yml`. 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:

```
docker container ps
```

Deverá ser impresso algo como:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS		NAMES		
49bf0d3241ad	mysql:5.7	"docker-entrypoint..."	26 minutes ago	Up 3 minutes
33060/tcp, 0.0.0.0:3307->3306/tcp		eats-microservices_mysql.pagamento_1		
4890dcb9e898	mongo:3.6	"docker-entrypoint..."	26 minutes ago	Up 3 minutes
0.0.0.0:27018->27017/tcp		eats-microservices_mongo.distancia_1		

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:

```
show dbs
```

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:

```
docker-compose logs
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

4.8 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.

4.9 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.

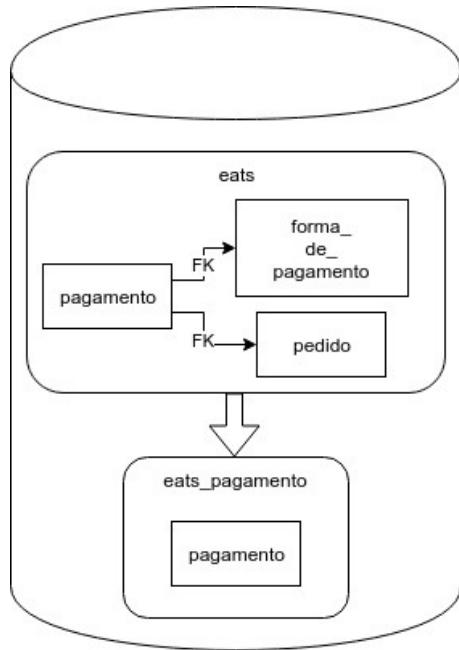


Figura 4.3: Schema separado para o BD de pagamentos

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?createDatabaseIfNotExist=true
spring.datasource.url=jdbc:mysql://localhost/eats_pagamento?createDatabaseIfNotExist=true
```

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 de 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_de_pagamento_id, pedido_id)
select id, valor, nome, numero, expiracao, codigo, status, forma_de_pagamento_id, pedido_id
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.license.VersionPrinter      :
Flyway Community Edition 5.2.4 by Boxfuse
2019-05-22 18:33:56.448  INFO 30484 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource      :
HikariPool-1 - Starting...
2019-05-22 18:33:56.632  INFO 30484 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource      :
HikariPool-1 - Start completed.
2019-05-22 18:33:56.635  INFO 30484 --- [ restartedMain] o.f.c.internal.database.DatabaseFactory      :
Database: jdbc:mysql://localhost/eats_pagamento (MySQL 5.6)
2019-05-22 18:33:56.708  INFO 30484 --- [ restartedMain] o.f.core.internal.command.DbValidate       :
Successfully validated 2 migrations (execution time 00:00.016s)
2019-05-22 18:33:56.840  INFO 30484 --- [ restartedMain] o.f.c.i.s.JdbcTableSchemaHistory          :
Creating Schema History table: `eats_pagamento`.`flyway_schema_history`
2019-05-22 18:33:57.346  INFO 30484 --- [ restartedMain] o.f.core.internal.command.DbMigrate        :
Current version of schema `eats_pagamento`: << Empty Schema >>
2019-05-22 18:33:57.349  INFO 30484 --- [ restartedMain] o.f.core.internal.command.DbMigrate        :
Migrating schema `eats_pagamento` to version 0001 - cria-tabela-pagamento
2019-05-22 18:33:57.596  INFO 30484 --- [ restartedMain] o.f.core.internal.command.DbMigrate        :
Migrating schema `eats_pagamento` to version 0002 - migra-dados-de-pagamento
2019-05-22 18:33:57.650  INFO 30484 --- [ restartedMain] o.f.core.internal.command.DbMigrate        :
```

```
Successfully applied 2 migrations to schema `eats_pagamento` (execution time 00:00.810s)
```

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:

```
select * from pagamento;
```

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.

4.10 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:

```
select * from pagamento;
```

Os pagamentos devem ter sido migrados. Note as colunas `forma_de_pagamento_id` e `pedido_id`.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

4.11 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_pagamento?createDatabaseIfNotExist=true
```

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!

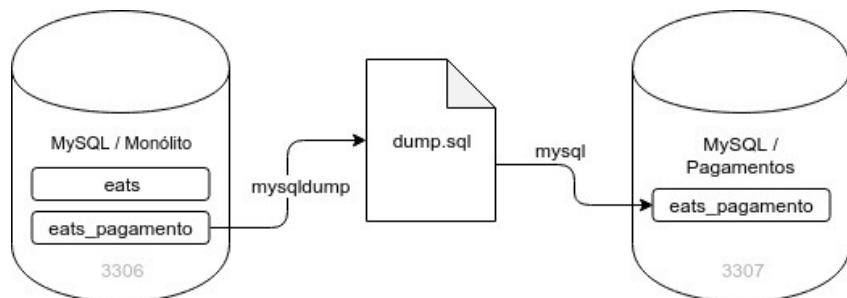


Figura 4.4: Dump do schema de Pagamentos importado para servidor de BD específico

4.12 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_pagamento.sql
```

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 dump
- `--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 --opt eats_pagamento > eats_pagamento.sql
```

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_pagamento < eats_pagamento.sql
```

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_pagamento.sql
```

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`.

1. 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_pagamento
```

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:

```
select * from pagamento;
```

Devem ser exibidos todos os pagamentos já efetuados!

Para sair, digite `exit`.

4.13 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_pagamento?createDatabaseIfNotExist=true
spring.datasource.url=jdbc:mysql://localhost:3307/eats_pagamento?createDatabaseIfNotExist=true

spring.datasource.username=<SEU_USUÁRIO>
spring.datasource.username=pagamento

spring.datasource.password=<SUAS_SENHA>
spring.datasource.password=pagamento123
```

Note que a porta 3307 foi incluída na URL, mas mantivemos ainda `localhost`.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

4.14 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
```

Reinic peace 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": "777 2222 8888 4444", "expiracao": "2025  
-04", "codigo": "777", "formaDePagamentoId": 1, "pedidoId": 2 }'  
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!

4.15 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 where r.aprovado = true into outfile '/tmp/restaurantes.csv' fields terminated by ',' enclosed by '\"' lines terminated by '\n';
```

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 --type csv --fields=_id,cep,tipoDeCozinhaId --file restaurantes.csv
```

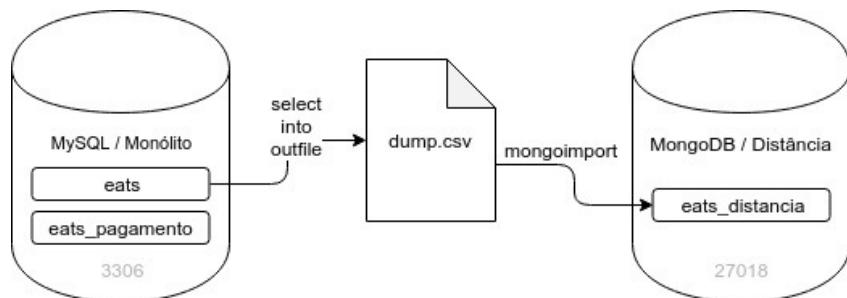


Figura 4.5: Dump para CSV para MongoDB de Distância

4.16 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:

```
mysql -u root -p eats
```

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 restaurante r where r.aprovado = true into outfile '/tmp/restaurantes.csv' fields terminated by ',' enclosed by '\"' lines terminated by '\n';
```

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 r where r.aprovado = true into outfile '/var/lib/mysql-files/restaurantes.csv' fields terminated by ',' enclosed by '\"' lines terminated by '\n';
```

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/restaurantes.csv > /tmp/restaurantes.csv
```

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>:/restaurantes.csv
```

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 --type csv --fields=_id,cep,tipoDeCozinhaId --file restaurantes.csv
```

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:

```
mongo eats_distancia
```

Dentro do Mongo Shell, verifique a collection de restaurantes foi criada:

```
show collections;
```

Deve ser retornado algo como:

```
restaurantes
```

Veja os documentos da collection `restaurantes` com o comando:

```
db.restaurantes.find();
```

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.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

4.17 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/com/caelum/eats/distancia/Restaurante.java

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String cep;

private Boolean aprovado;

@Column(nullable = false)
private Long tipoDeCozinhaId;

}
```

Os seguinte 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/com/caelum/eats/distancia/RestauranteRepository.java

interface RestauranteRepository extends JpaRepository<Restaurante, Long> {
interface RestauranteRepository extends MongoRepository<Restaurante, Long> {

    Page<Restaurante> findAllByAprovadoAndTipoDeCozinhaId(boolean aprovado, Long tipoDeCozinhaId, Pageable limit);
    Page<Restaurante> findAllByTipoDeCozinhaId(Long tipoDeCozinhaId, Pageable limit);

    Page<Restaurante> findAllByAprovado(boolean aprovado, Pageable limit);
    Page<Restaurante> findAll(Pageable limit);

}
```

Os imports devem ser corrigidos:

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.mongodb.repository.MongoRepository;
```

Como removemos o atributo aprovado, é necessário alterar a chamada ao RestauranteRepository em alguns métodos do DistanciaService :

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

```
// anotações...
class DistanciaService {

    // atributos...

    public List<RestauranteComDistanciaDto> restaurantesMaisProximosAoCep(String cep) {
        List<Restaurante> aprovados = restaurantes.findAllByAprovado(true, LIMIT).getContent();
        List<Restaurante> aprovados = restaurantes.findAll(LIMIT).getContent(); // modificado
        return calculaDistanciaParaOsRestaurantes(aprovados, cep);
    }

    public List<RestauranteComDistanciaDto> restaurantesDoTipoDeCozinhaMaisProximosAoCep(Long tipoDeCozinhaId, String cep) {
        List<Restaurante> aprovadosDoTipoDeCozinha = restaurantes.findAllByAprovadoAndTipoDeCozinhaId(true, tipoDeCozinhaId, LIMIT).getContent();
        List<Restaurante> aprovadosDoTipoDeCozinha = restaurantes.findAllByTipoDeCozinhaId(tipoDeCozinhaId, LIMIT).getContent();
        return calculaDistanciaParaOsRestaurantes(aprovadosDoTipoDeCozinha, cep);
    }

    // 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.

```
# fj33-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?createDatabaseIfNotExist=true
```

```
spring.datasource.username=<SEU USUÁRIO>
spring.datasource.password=<SUAS SENHAS>

#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>

4.18 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/tipos-de-cozinha/1
curl -i http://localhost:8082/restaurantes/71503510/restaurante/1
```

Observação: como é disparado um GET, é possível testar as URLs anteriores diretamente pelo navegador.

4.19 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.

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

INTEGRAÇÃO SÍNCRONA (E RESTFUL)

5.1 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.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

5.2 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](#) (BERNERS-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,iVBORw0KGgoAAAANSUhEUgAAAAUAAAFCAYAAACNbyb1AAAHE1EQVQI12P4//8/w38GIAxDIBKE0DHxgljNBAA09TXL0Y40HwAAAABJRU5ErkJgg==

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](#) (DUSSEAUlt; 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](#) (DUSSEAUT; 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 representando 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.

5.3 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

```

5.4 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 provêm 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="14:50"/>
  <horario id="5678" doutor="huberman" inicio="16:00" fim="16:50"/>
</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="14:50"/>
  <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="14:50"/>
  <horario id="5678" doutor="huberman" inicio="16:00" fim="16:50"/>
</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="14:50"/>
  <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="16:50"/>
</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.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

5.5 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.

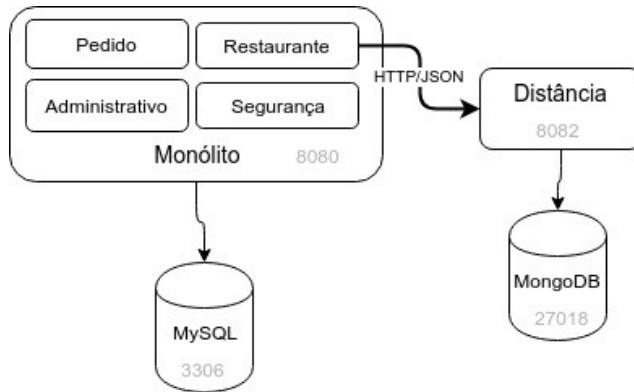


Figura 5.1: Módulo de Restaurante faz chamada HTTP ao serviço de Distância

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.

```
# f33-eats-distancia-service/src/main/java/br/com/caelum/eats/distancia/RestaurantesController.java

@RestController
@AllArgsConstructor
@Slf4j
class RestaurantesController {

    private RestaurantesRepository repo;

    @PostMapping("/restaurantes")
    ResponseEntity<Restaurante> adiciona(@RequestBody Restaurante restaurante, UriComponentsBuilder uri
Builder) {
        log.info("Insere novo restaurante: " + restaurante);
        Restaurante salvo = repo.insert(restaurante);
        UriComponents uriComponents = uriBuilder.path("/restaurantes/{id}").buildAndExpand(salvo.getId());
    ;
        URI uri = uriComponents.toUri();
        return ResponseEntity.created(uri).contentType(MediaType.APPLICATION_JSON).body(salvo);
    }

    @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:

```
# f33-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:

```
# f33-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 RestauranteParaServiçoDeDistancia 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:

```
# f33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/RestauranteParaServiçoDeDistancia.java
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
class RestauranteParaServiçoDeDistancia {

    private Long id;
```

```

private String cep;
private Long tipoDeCozinhaId;

RestauranteParaServicoDeDistancia(Restaurante restaurante){
    this(restaurante.getId(), restaurante.getCep(), restaurante.getTipoDeCozinha().getId());
}

}

```

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/DistanciaRestClient.java
```

```

@Service
class DistanciaRestClient {

    private String distanciaServiceUrl;
    private RestTemplate restTemplate;

    DistanciaRestClient(RestTemplate restTemplate,
                        @Value("${configuracao.distancia.service.url}") String distanciaServiceUrl) {
        this.distanciaServiceUrl = distanciaServiceUrl;
        this.restTemplate = restTemplate;
    }

    void novoRestauranteAprovado(Restaurante restaurante) {
        RestauranteParaServicoDeDistancia restauranteParaDistancia = new RestauranteParaServicoDeDistancia(restaurante);
        String url = distanciaServiceUrl + "/restaurantes";
        ResponseEntity<RestauranteParaServicoDeDistancia> responseEntity =
            restTemplate.postForEntity(url, restauranteParaDistancia, RestauranteParaServicoDeDistancia.class);
        HttpStatus statusCode = responseEntity.getStatusCode();
        if (!HttpStatus.CREATED.equals(statusCode)) {

```

```

        throw new RuntimeException("Status diferente do esperado: " + statusCode);
    }
}

void restauranteAtualizado(Restaurante restaurante) {
    RestauranteParaServicoDeDistancia restauranteParaDistancia = new RestauranteParaServicoDeDistancia(
        restaurante);
    String url = distanciaServiceUrl + "/restaurantes/" + restaurante.getId();
    restTemplate.put(url, restauranteParaDistancia, RestauranteParaServicoDeDistancia.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/RestauranteController.java

```

// anotações ...
class RestauranteController {

    private RestauranteRepository restauranteRepo;
    private CardapioRepository cardapioRepo;
    private DistanciaRestClient distanciaRestClient; // adicionado

    // 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.getTipoDeCozinha().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?

5.6 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-template

cd ~/Desktop/fj33-eats-distancia-service
git checkout -f cap6-integracao-monolito-distancia-com-rest-template

```

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:

```
db.restaurantes.find();
```

5.7 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.

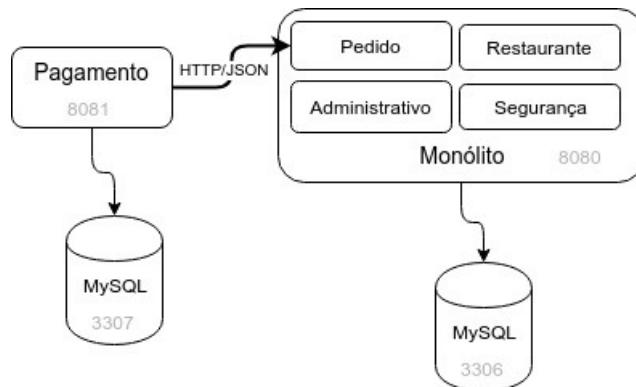


Figura 5.2: Serviço de Pagamentos chama módulo de Pedido do Monólito

Adicione ao `PedidoController`, do módulo `eats-pedido` do monólito, um método que muda o status do pedido para *PAGO*:

```
# f33-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/EatsPagamentoServiceApplication.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.java  
  
{@FeignClient(url="${configuracao.pedido.service.url}", name="pedido")  
interface PedidoRestClient {  
  
  @PutMapping("/pedidos/{pedidoId}/pago")
```

```
void avisaQueFoiPago(@PathVariable("pedidoId") Long pedidoId);  
}
```

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/PagamentoController.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(() -> new ResourceNotFoundException());  
        pagamento.setStatus(Pagamento.Status.CONFIRMADO);  
        pagamentoRepo.save(pagamento);  
  
        // adicionado  
        Long pedidoId = pagamento.getPedidoId();  
        pedidoClient.avisaQueFoiPago(pedidoId);  
  
        return new PagamentoDto(pagamento);  
    }  
  
    // restante do código ...  
}
```

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

5.8 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 pagamento foi reiniciado e que os demais serviços e o front-end estão no ar.

Faça um novo pedido, realizando e confirmando um pagamento.

Veja que, depois dessa mudança, o status do pedido fica como **PAGO** e não apenas como **REALIZADO**.

5.9 O PODER DOS LINKS

Ao descrevermos as boas ideias do protocolo 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**.

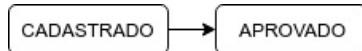


Figura 5.3: A transição de estados de um restaurante

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.



Figura 5.4: A transição de estados de um pedido

Um pagamento tem uma transição mais interessante: depois de **CREADO**, pode ser **CONFIRMADO** ou **CANCELADO**.

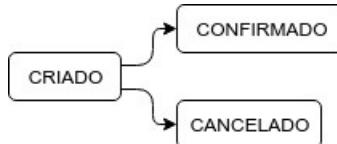


Figura 5.5: A transição de estados de um pagamento

Se observarmos a implementação da API de pagamentos do `eats-pagamento-service`,

- um `POST` em `/pagamentos` adiciona um novo pagamento com o estado `CREADO` 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", <https://api.github.com/repositories/237159/pulls?page=2>; rel="last"
```

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="14:50">
    <link rel="agendamento" href="/horarios/1234">
  </horario>
  <horario id="5678" doutor="huberman" inicio="16:00" fim="16:50">
    <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="14:50"/>
  <paciente id="alexandre"/>
  <link rel="agendamentoExame" href="/horarios/1234/consulta/exames">
  <link rel="cancelamento" href="/horarios/1234/consulta">
  <link rel="mudancaHorario" href="/doutor/huberman/horarios?data=2010-01-04&status=disponivel">
</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.

5.10 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).orElseThrow(() -> new ResourceNotFoundException());
    }
}
```

```

List<Link> links = new ArrayList<>();

Link self = linkTo(methodOn(PagamentoController.class).detalha(id)).withSelfRel();
links.add(self);

if (Pagamento.Status.CRIADO.equals(pagamento.getStatus())) {
    Link confirma = linkTo(methodOn(PagamentoController.class).confirma(id)).withRel("confirma");
    links.add(confirma);

    Link cancela = linkTo(methodOn(PagamentoController.class).cancela(id)).withRel("cancela");
    links.add(cancela);
}

PagamentoDto dto = new PagamentoDto(pagamento);
Resource<PagamentoDto> resource = new Resource<PagamentoDto>(dto, links);

return resource;
}

@PostMapping
public ResponseEntity<Resource<PagamentoDto>> cria(@RequestBody Pagamento pagamento,
    UriComponentsBuilder uriBuilder) {
    pagamento.setStatus(Pagamento.Status.CRIADO);
    Pagamento salvo = pagamentoRepo.save(pagamento);
    URI path = uriBuilder.path("/pagamentos/{id}").buildAndExpand(salvo.getId()).toUri();
    PagamentoDto dto = new PagamentoDto(salvo);

    Long id = salvo.getId();

    List<Link> links = new ArrayList<>();

    Link self = linkTo(methodOn(PagamentoController.class).detalha(id)).withSelfRel();
    links.add(self);

    Link confirma = linkTo(methodOn(PagamentoController.class).confirma(id)).withRel("confirma");
    links.add(confirma);

    Link cancela = linkTo(methodOn(PagamentoController.class).cancela(id)).withRel("cancela");
    links.add(cancela);

    Resource<PagamentoDto> resource = new Resource<PagamentoDto>(dto, links);
    return ResponseEntity.created(path).body(resource);
}

@PutMapping("/{id}")
public Resource<PagamentoDto> confirma(@PathVariable Long id) {
    Pagamento pagamento = pagamentoRepo.findById(id).orElseThrow(() -> new ResourceNotFoundException());
    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).detalha(id)).withSelfRel();
    links.add(self);

    PagamentoDto dto = new PagamentoDto(pagamento);
    Resource<PagamentoDto> resource = new Resource<PagamentoDto>(dto, links);

    return resource;
}

```

```

}

@RequestMapping("/{id}")
public Resource<PagamentoDto> cancela(@PathVariable Long id) {
    Pagamento pagamento = pagamentoRepo.findById(id).orElseThrow(() -> new ResourceNotFoundException());
    pagamento.setStatus(Pagamento.Status.CANCELADO);
    pagamentoRepo.save(pagamento);

    List<Link> links = new ArrayList<>();

    Link self = linkTo(methodOn(PagamentoController.class).detalha(id)).withSelfRel();
    links.add(self);

    PagamentoDto dto = new PagamentoDto(pagamento);
    Resource<PagamentoDto> resource = new Resource<PagamentoDto>(dto, links);

    return resource;
}
}

```

- Reinic peace 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.

- 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}`, null);
  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!

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

5.11 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/authorizations/8AA831015G517922L"
    },
    {
      "rel": "capture",
      "method": "POST",
      "href": "https://api.paypal.com/v2/payments/authorizations/8AA831015G517922L/capture"
    },
    {
      "rel": "void",
      "method": "POST",
      "href": "https://api.paypal.com/v2/payments/authorizations/8AA831015G517922L/void"
    }
  ]
}

```

```

        "href": "https://api.paypal.com/v2/payments/authorizations/8AA831015G517922L/void"
    },
    {
        "rel": "reauthorize",
        "method": "POST",
        "href": "https://api.paypal.com/v2/payments/authorizations/8AA831015G517922L/reauthorize"
    }
]
}

```

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.

5.12 EXERCÍCIO OPCIONAL: ESTENDENDO O SPRING HATEOAS

- 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;
```

- 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).confirma(id)).withRel("confirma");
links.add(confirma);
links.add(new LinkWithMethod(confirma, "PUT")); // modificado
```

```

Link cancela = linkTo(methodOn(PagamentoController.class).cancela(id)).withRel("cancela");
links.add(cancela);
links.add(new LinkWithMethod(cancela, "DELETE")); // modificado

```

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);
}

```

5. (desafio) Modifique o `PagamentoController` para usar HAL-FORMS, disponível nas últimas versões do Spring HATEOAS.

5.13 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.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

5.14 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, Long> {

    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"  
                }  
            }  
        }  
    }  
}
```

```

        "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{?sobrenome}",
            "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?sobrenome=Aquiles"
    }
  }
}
```

5.15 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.

5.16 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.

Saber inglês é muito importante em TI



cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

5.17 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 envia 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 `distanzia.proto`, com um RPC simples:

```
syntax = "proto3";

package distancia; // não deve ser reverse domain name

option java_package = "br.com.caelum.eats.distancia"; // pacote na convenção Java

service Distancia {

    rpc MaisProximos (MaisProximosRequest) returns (MaisProximosResponse) {}

}
```

```

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/protobuf/java/ src/main/proto/distancia.proto
```

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 MaisProximosResponse 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.detected.classifier}</protocArtifact>
  </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.detected.classifier}</pluginArtifact>

```

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.DistanciaImplBase {
    @Override
    public void maisProximos(MaisProximosRequest request, StreamObserver<MaisProximosResponse> response
    Observer) {
        // obtém CEP do request gRPC
        String cep = request.getCep();
    }
}

```

```

List<RestauranteComDistanciaDto> maisProximos = // obtém lista de restaurantes mais próximos

// monta lista de restaurantes para gRPC
List<RestauranteComDistancia> restaurantesParaGrpc = new ArrayList<>();
for (RestauranteComDistanciaDto maisProximo : maisProximos) {
    RestauranteComDistancia restauranteParaGrpc = RestauranteComDistancia
        .newBuilder()
        .setDistancia(maisProximo.getDistancia().doubleValue())
        .setRestauranteId(maisProximo.getRestauranteId())
        .build();
    restaurantesParaGrpc.add(restauranteParaGrpc);
}

// monta response gRPC
MaisProximosResponse maisProximosResponse = MaisProximosResponse
    .newBuilder()
    .addAllRestaurantes(restaurantesParaGrpc)
    .build();

responseObserver.onNext(maisProximosResponse);
responseObserver.onCompleted();

}
}

```

Os imports corretos seriam os seguintes:

```

import br.com.caelum.eats.distancia.DistanciaOuterClass.MaisProximosRequest;
import br.com.caelum.eats.distancia.DistanciaOuterClass.MaisProximosResponse;
import br.com.caelum.eats.distancia.DistanciaOuterClass.RestauranteComDistancia;
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, InterruptedException {
        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", 6565)
            .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.getRestaurantesList();

        // Usa lista de restaurantes com distância...
    }
}

```

A lista de imports:

```

import br.com.caelum.eats.distancia.DistanciaGrpc.DistanciaBlockingStub;
import br.com.caelum.eats.distancia.DistanciaOuterClass.MaisProximosRequest;
import br.com.caelum.eats.distancia.DistanciaOuterClass.MaisProximosResponse;
import br.com.caelum.eats.distancia.DistanciaOuterClass.RestauranteComDistancia;
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.DistanciaImplBase {

```

```
// 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>

5.18 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://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```

<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>
    <dependency>
        <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:${os.detected.classifier}</protocArtifact>
            <pluginId>grpc-java</pluginId>
            <pluginArtifact>io.grpc:protoc-gen-grpc-java:1.24.0:exe:${os.detected.classifier}</pluginArtifact>
        </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://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.8.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/RecomendacoesApplication.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/RecomendacoesApplication.java

@GRpcService
public class RecomendacoesService extends RecomendacoesDeRestaurantesGrpc.RecomendacoesDeRestaurantesImplBase {

    @Override
    public void recomendacoes(Restaurantes request, StreamObserver<Restaurantes> responseObserver) {
        List<Long> idsDeRestaurantes = request.getRestauranteIdList();

        // simula recomendacao
        List<Long> idsDeRestaurantesOrdenadosPorRecomendacoes = idsDeRestaurantes;
        if (idsDeRestaurantes.size() > 1) {
            idsDeRestaurantesOrdenadosPorRecomendacoes = new ArrayList<Long>(idsDeRestaurantes);
            Collections.shuffle(idsDeRestaurantesOrdenadosPorRecomendacoes);
        }

        Restaurantes response = Restaurantes
            .newBuilder()
            .addAllRestauranteId(idsDeRestaurantesOrdenadosPorRecomendacoes)
            .build();

        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }
}

```

Devem ser definidos os seguintes imports:

```

#                                         fj33-recomendacoes-
service/src/main/java(br/com/caelum/eats/recomendacoes/RecomendacoesApplication.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.Restaurantes;
import br.com.caelum.eats.recomendacoes.grpc.RecomendacoesDeRestaurantesGrpc;
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.RecomendacoesApplication
: Started RecomendacoesApplication in 1.497 seconds (JVM running for 1.91)
2019-10-30 14:13:23.279 INFO 25955 --- [ restartedMain] o.l.springboot.grpc.GRpcServerRunner
: Starting gRPC Server ...
2019-10-30 14:13:23.339 INFO 25955 --- [ restartedMain] o.l.springboot.grpc.GRpcServerRunner
: 'br.com.caelum.eats.recomendacoes.RecomendacoesService' service has been registered.
2019-10-30 14:13:23.538 INFO 25955 --- [ restartedMain] o.l.springboot.grpc.GRpcServerRunner
: gRPC Server started, listening on port 6565.
```

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/RecomendacoesGrpcClient.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/RecomendacoesGrpcClient.java  
  
// anotações omitidas...  
class RecomendacoesGrpcClient {  
  
    private String recomendacoesServiceHost; // adicionado  
    private Integer recomendacoesServicePort; // adicionado  
  
    // adicionado  
    RecomendacoesGrpcClient(@Value("${recomendacoes.service.host}") String recomendacoesServiceHost,  
                           @Value("${recomendacoes.service.port}") Integer recomendacoesServicePort) {  
        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/RecomendacoesGrpcClient.java  
  
// anotações omitidas...  
class RecomendacoesGrpcClient {  
  
    // demais atributos omitidos ...  
    private ManagedChannel channel; // adicionado  
    private RecomendacoesDeRestaurantesBlockingStub recomendacoes; // adicionado  
  
    // construtor omitido ...  
  
    // adicionado  
    @PostConstruct  
    void conectaAoRecomendacoesGrpcService() {  
        channel = ManagedChannelBuilder.forAddress(recomendacoesServiceHost, recomendacoesServicePort)  
            .usePlaintext() // desabilita TLS porque precisa de certificado (apenas para testes)  
            .build();  
        recomendacoes = RecomendacoesDeRestaurantesGrpc.newBlockingStub(channel);  
    }  
  
    // 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 retornando a lista obtida do serviço de Recomendações.

```

#                                         fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/RecomendacoesGrpcClient.java

// anotações omitidas...
class RecomendacoesGrpcClient {

    // código omitido...

    List<Long> ordenaPorRecomendacoes(List<Long> idsDeRestaurantes) {
        Restaurantes restaurantes = Restaurantes
            .newBuilder()
            .addAllRestauranteId(idsDeRestaurantes)
            .build();

        Restaurantes restaurantesOrdenadosPorRecomendacao = recomendacoes.recomendacoes(restaurantes);

        List<Long> restaurantesOrdenados = restaurantesOrdenadosPorRecomendacao.getRestauranteIdList();

        log.info("Restaurantes ordenados: {}", restaurantesOrdenados);

        return restaurantesOrdenados;
    }

}

```

Os imports da classe `RecomendacoesGrpcClient` devem ser os seguintes:

```

#                                         fj33-eats-distancia-
service/src/main/java/br/com/caelum/eats/distancia/RecomendacoesGrpcClient.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.Recomendacoes.Restaurantes;
import br.com.caelum.eats.recomendacoes.grpc.RecomendacoesDeRestaurantesGrpc;
import br.com.caelum.eats.recomendacoes.grpc.RecomendacoesDeRestaurantesGrpc.RecomendacoesDeRestaurantesBlockingStub;
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, extrais 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.toList());

            List<Long> idsDeRestaurantesOrdenadosPorRecomendacao = recomendacoesClient.ordenaPorRecomenda-
coes(idsDeRestaurantes);

            List<Restaurante> restaurantesOrdenadosPorRecomendacao = new ArrayList<>(restaurantes);
            restaurantesOrdenadosPorRecomendacao
                .sort(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> calculaDistanciaParaOsRestaurantes(
        List<Restaurante> restaurantes,
        String cep) {
        return restaurantes;
        return ordenaPorRecomendacoes(restaurantes) // modificado
            .stream()
            .map(restaurante -> {
                String cepDoRestaurante = restaurante.getCep();
                RestauranteComDistanciaDto restauranteComDistancia = new RestauranteComDistanciaDto();
                restauranteComDistancia.setNome(restaurante.getNome());
                restauranteComDistancia.setCep(cepDoRestaurante);
                restauranteComDistancia.setLatitude(restaurante.getLatitude());
                restauranteComDistancia.setLongitude(restaurante.getLongitude());
                restauranteComDistancia.setDistancia(distanciaService.calculaDistancia(
                    restaurante.getLatLong(), Point.of(cep)));
                return restauranteComDistancia;
            });
    }
}
```

```

        BigDecimal distancia = distanciaDoCep(cepDoRestaurante, cep);
        Long restauranteId = restaurante.getId();
        return new RestauranteComDistanciaDto(restauranteId, distancia);
    }
    .collect(Collectors.toList());
}
}

```

Execute a classe `EatsDistanciaServiceApplication`.

- Em um Terminal, use o cURL para invocar o serviço de Distância algumas vezes, da seguinte maneira:

```
curl http://localhost:8082/restaurantes/mais-proximos/78000-777
```

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":9.4187908418432879642523403163067996501922607421875},
  {"restauranteId":1,"distancia":6.5896083315768390065159110235981643199920654296875},
  {"restauranteId":2,"distancia":3.526626757955934721167068346403539180755615234375}
]
```

Já na segunda chamada:

```
[
  {"restauranteId":2,"distancia":12.51659444929334341622961801476776599884033203125},
  {"restauranteId":1,"distancia":0.09125363010240528094385581425740383565425872802734375},
  {"restauranteId":3,"distancia":7.10856812555768957651025630184449255466461181640625}
]
```

Na terceira:

```
[
  {"restauranteId":1,"distancia":13.1420093922015350784704423858784139156341552734375},
  {"restauranteId":3,"distancia":4.5362206318291011797327882959507405757904052734375},
  {"restauranteId":2,"distancia":11.97968639569695170621344004757702350616455078125}
]
```

5.19 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](https://blog.caelum.com.br/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 framework for node.",
  "fork": false,
  "issues_url": "https://api.github.com/repos/expressjs/express/issues{/number}",
  "pulls_url": "https://api.github.com/repos/expressjs/express/pulls{/number}",
  "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/pulls/3391",
    "id": 134639441,
    "html_url": "https://github.com/expressjs/express/pull/3391",
    "diff_url": "https://github.com/expressjs/express/pull/3391.diff",
    "patch_url": "https://github.com/expressjs/express/pull/3391.patch",
    "issue_url": "https://api.github.com/repos/expressjs/express/issues/3391",
    "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",
      ...
    },
    "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/pulls/3390",
    "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/pulls/2730",
    "id": 41965836,
    ...
  },
  {
    //outro pull request...
    "url": "https://api.github.com/repos/expressjs/express/pulls/2703",
    "id": 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:

```
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?

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

5.20 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,localizedFirstName,localizedLastName)

{
  "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&access_token={your-user-access-token}

{
  "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,characteristics/length)

{
  "kind": "demo",
  "items": [
    {
      "title": "First title",
      "characteristics": {
        "length": "short"
      }
    },
    {
      "title": "Second title",
      "characteristics": {
        "length": "long"
      }
    }
  ]
}
```

API GATEWAY

6.1 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
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

6.2 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: '//localhost:8080',
  baseUrl: '//localhost:9999' // modificado
  , pagamentoUrl: '//localhost:8081'
  , distanciaUrl: '//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'; // modificado

  // 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'; // modificado

  // código omitido ...

}
```

6.3 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-gateway.git
```

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?

6.4 DESABILITANDO A REMOÇÃO DE CABEÇALHOS SENSÍVEIS NO ZUUL

Por padrão, o Zuul remove os cabeçalhos HTTP `Cookie` , `Set-Cookie` , `Authorization` . Vamos desabilitar essa remoção no `application.properties` :

```
# fj33-api-gateway/src/main/resources/application.properties  
  
zuul.sensitiveHeaders=
```

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

6.5 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!

6.6 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.java

@Service
class DistanciaRestClient {

    private RestTemplate restTemplate;
    private String distanciaServiceUrl;

    DistanciaRestClient(RestTemplate restTemplate,
        @Value("${zuul.routes.distancia.url}") String distanciaServiceUrl) {
        this.restTemplate = restTemplate;
        this.distanciaServiceUrl = distanciaServiceUrl;
    }

    Map<String, Object> porCepEId(String cep, Long restauranteId) {
        String url = distanciaServiceUrl + "/restaurantes/" + cep + "/restaurante/" + restauranteId;
        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`.

6.7 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.url=http://localhost:8080
monolito.ribbon.listOfServers=http://localhost:8080
```

Mais adiante estudaremos cuidadosamente o Ribbon.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

6.8 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`

```
# f33-api-gateway/src/main/java/br/com/caelum/apigateway/RestauranteComDistanciaController.java
```

```
@RestController
@AllArgsConstructor
class RestauranteComDistanciaController {

    private RestauranteRestClient restauranteRestClient;
    private DistanciaRestClient distanciaRestClient;

    @GetMapping("/restaurantes-com-distancia/{cep}/restaurante/{restauranteId}")
    public Map<String, Object> porCepEIdComDistancia(@PathVariable("cep") String cep,
                                                       @PathVariable("restauranteId") Long restauranteId) {
        Map<String, Object> dadosRestaurante = restauranteRestClient.porId(restauranteId);
        Map<String, Object> dadosDistancia = distanciaRestClient.porCepEId(cep, restauranteId);
        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.

6.9 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): Observable<any> {
        return this.http.get(`${this.API}/restaurantes-com-distancia/${cep}/restaurante/${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.porId(restauranteId)
    this.restaurantesService.porCepEIdComDistancia(this.cep, restauranteId) // modificado
    .subscribe(restaurante => {

        this.restaurante = restaurante;
        this pedido.restaurante = restaurante;

        this.restaurantesService.distanciaPorCepEId(this.cep, restauranteId)
            .subscribe(restauranteComDistancia => {
                this.restaurante.distancia = restauranteComDistancia.distancia;
            });
    });

    // 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("*").allowCredentials(true);
    }
}

```

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.

6.10 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":11.393642891403121808480136678554117679595947265625,  
  "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  
}
```

2. 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
```

Reinic peace o monólito e os serviços de pagamento e distância.

3. 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.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

6.11 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 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`.

Vamos customizá-lo, para que funcione com respostas bem sucedidas, de status 2XX .

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 = RequestContextHolder.getCurrentContext().getResponseStatus();
                return HttpStatus.valueOf(statusCode).is3xxRedirection() || HttpStatus.valueOf(statusCode).is2xxSuccessful();
            }
        };
    }

}
```

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.

6.12 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 2222 3333 4444", "expiracao": "2022-07", "codigo": "123", "formaDePagamentoId": 2, "pedidoId": 1 }'
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.

6.13 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.java
```

```
@Component
public class RateLimitingZuulFilter extends ZuulFilter {

    private final RateLimiter rateLimiter = RateLimiter.create(1.0 / 30.0); // permits per second

    @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.FilterConstants;
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.

Saber inglês é muito importante em TI



cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

CLIENT SIDE LOAD BALANCING COM RIBBON

7.1 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 CommonsRequestLoggingFilter();
        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.CommonsRequestLoggingFilter;
```

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.CommonsRequestLoggingFilter=DEBUG
```

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

7.2 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-de-distancia
```

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)`.

7.3 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.LoadBalanced;
```

Mude o arquivo `application.properties`, do módulo `eats-application` do monólito, para que seja configurado o `virtual host` `distorcia`, 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:8082
configuracao.distancia.service.url=http://distorcia

distorcia.ribbon.listOfServers=http://localhost:8082,http://localhost:9092
ribbon.eureka.enabled=false
```

7.4 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://distorcia` do Ribbon. Essa propriedade será usada no `DistanciaRestClient`.

```
# fj33-api-gateway/src/main/resources/application.properties

zuul.routes.distorcia.url=http://localhost:8082

zuul.routes.distorcia.path=/distorcia/**
distorcia.ribbon.listOfServers=http://localhost:8082,http://localhost:9092

configuracao.distancia.service.url=http://distorcia
```

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.java

class DistanciaRestClient {

    // código omitido ...
```

```

DistancaRestClient(RestTemplate restTemplate,
    @Value("${zuul.routes.distancia.url}") String distanciaServiceUrl) {
    @Value("${configuracao.distancia.service.url}") String distanciaServiceUrl) {

    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/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.LoadBalanced;
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

7.5 EXERCÍCIO: CLIENT SIDE LOAD BALANCING NO RESTTEMPLATE DO MONÓLITO 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`.

7.6 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!

7.7 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://localhost:9090
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.java

@FeignClient(url="${configuracao_pedido.service.url}", name="pedido")
@FeignClient("monolito") // modificado
public interface PedidoRestClient {

    // código omitido ...
}
```

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

7.8 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://localhost:9090
```

7.9 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 pagamento 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

8.1 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`:

```
# f33-service-registry/src/main/java(br/com/caelum/serviceregistry/ServiceRegistryApplication.java

@EnableEurekaServer
@SpringBootApplication
public class ServiceRegistryApplication {

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

Adicione o import:

```
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
```

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
```

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

8.2 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-registry.git
```

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.

8.3 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.EnableDiscoveryClient;
```

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://localhost:8761/eureka/}
```

Dessa maneira, caso seja necessário modificar a URL padrão do Eureka Server, basta definir a variável de ambiente `EUREKA_URI`.

8.4 SELF REGISTRATION DO SERVIÇO DE PAGAMENTO 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/EatsPagamentoServiceApplication.java
```

```
@EnableDiscoveryClient // adicionado
@EnableFeignClients
@SpringBootApplication
public class EatsPagamentoServiceApplication {

}
```

Lembrando que o import é:

```
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
```

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://localhost:8761/eureka/}
```

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

8.5 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/}
```

8.6 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/br/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.EnableDiscoveryClient;
```

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://localhost:8761/eureka/}
```

8.7 EXERCÍCIO: SSELF 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*.

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

8.8 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
```

```
monolito.ribbon.listOfServers=http://localhost:8080,http://localhost:9090
ribbon.eureka.enabled=false
```

8.9 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,http://localhost:9092
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,http://localhost:9090
```

8.10 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,http://localhost:9092
ribbon.eureka.enabled=false
```

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

8.11 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.

RESILIÊNCIA

9.1 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-distancia/71503510/restaurante/1
```

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!

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

9.2 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.java
```

```
@Service
class DistanciaRestClient {

    // código omitido...

    @HystrixCommand // adicionado
    Map<String, Object> porCepEId(String cep, Long restauranteId) {
```

```

        String url = distanciaServiceUrl + "/restaurantes/" + cep + "/restaurante/" + restauranteId;
        return restTemplate.getForObject(url, Map.class);
    }

}

```

O import é o seguinte:

```
import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
```

9.3 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-distancia/71503510/restaurante/1
```

Observe nos resultados uma diminuição no tempo máximo de request de 19,6 para 1,5 segundos:

	Connection Times (ms)			
	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	275

9.4 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 and fallback failed.
  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(OperatorOnErrorResumeNextViaFunction.java:140)
  ...

```

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 and fallback failed.
    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(OperatorOnErrorResumeNextViaFunction.java:140)
```

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
    @HystrixCommand(fallbackMethod="restauranteSemDistanciaNemDetalhes") // modificado
    Map<String, Object> porCepEId(String cep, Long restauranteId) {
        String url = distanciaServiceUrl+"/restaurantes/"+cep+"/restaurante/"+restauranteId;
        return restTemplate.getForObject(url, Map.class);
    }

    // método adicionado
    Map<String, Object> restauranteSemDistanciaNemDetalhes(String cep, Long restauranteId) {
        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.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

9.5 EXERCÍCIO: FALBACK 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
```

Reinic peace 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 SUL",
```

```

    "taxaDeEntregaEmReais": 6,
    "tempoDeEntregaMinimoEmMinutos": 40,
    "tempoDeEntregaMaximoEmMinutos": 25,
    "aprovado": true,
    "tipoDeCozinha": {
        "id": 1,
        "nome": "Chinesa"
    },
    "restauranteId": 1
}

```

9.6 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);
    }

}
```

9.7 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/RestauranteController.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).orElseThrow(() -> new ResourceNotFoundException());
    }
}
```

```
        return new RestauranteDto(restaurante);
    }

    // restante do código ...
}
```

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

9.8 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.

9.9 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:

```
There was an unexpected error (type=Internal Server Error, status=500).
route:SendForwardFilter
com.netflix.hystrix.exception.HystrixRuntimeException: RestauranteRestClient#porId(Long) timed-out and no fallback available.
...
...
```

Quando o Circuit Breaker estiver aberto, a mensagem da exceção `HystrixRuntimeException` será um pouco diferente: *RestauranteRestClient#porId(Long) short-circuited and no fallback available.*

9.10 FALBACK 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/RestauranteRestClientFallback.java
```

```
@Component
class RestauranteRestClientFallback implements RestauranteRestClient {

    @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=RestauranteRestClientFallback.class) // modificado
interface RestauranteRestClient {

    @GetMapping("/restaurantes/{id}")
    Map<String, Object> porId(@PathVariable("id") Long id);

}
```

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

9.11 EXERCÍCIO: FALBACK 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>

Veja que são mostrados apenas o id e a distância do restaurante. Os demais campos não são exibidos.

9.12 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/RestauranteController.java

```
// anotações ...
class RestauranteController {

    // código omitido ...

    @GetMapping("/restaurantes/{id}")
    public RestauranteDto detalha(@PathVariable("id") Long id) {

        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        Restaurante restaurante = restauranteRepo.findById(id).orElseThrow(() -> new ResourceNotFoundException());
        return new RestauranteDto(restaurante);
    }
}
```

Teste novamente a URL: <http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1>

Os detalhes do restaurante devem voltar a ser exibidos!

9.13 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/RestaurantesController.java](#)

```
// anotações ...
class RestaurantesController {

    // código omitido ...

    @PutMapping("/restaurantes/{id}")
    Restaurante atualiza(@PathVariable("id") Long id, @RequestBody Restaurante restaurante) {

        throw new RuntimeException();

        // código comentado ...

        //if (!repo.existsById(id)) {
        //    throw new ResourceNotFoundException();
        //}
        //log.info("Atualiza restaurante: " + restaurante);
        //return repo.save(restaurante);
    }
}
```

Saber inglês é muito importante em TI



cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

9.14 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
// outras 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/DistanciaRestClient.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/DistanciaRestClient.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;
```

9.15 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.restaurante.DistanciaRestClient
: monólito tentando chamar distancia-service
2019-06-18 17:30:43.967 INFO 12547 --- [nio-8080-exec-9] b.c.c.e.restaurante.DistanciaRestClient
: monólito tentando chamar distancia-service
```

```

2019-06-18 17:30:44.990 INFO 12547 --- [nio-8080-exec-9] b.c.c.e.restaurante.DistanciaRestClient
: monólito tentando chamar distancia-service
2019-06-18 17:30:46.034 INFO 12547 --- [nio-8080-exec-9] b.c.c.e.restaurante.DistanciaRestClient
: monólito tentando chamar distancia-service
2019-06-18 17:30:46.085 ERROR 12547 --- [nio-8080-exec-9] o.a.c.c.C.[.][/].[dispatcherServlet]
: Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Requ
est processing failed; nested exception is org.springframework.web.client.HttpServerErrorException
$InternalServerException: 500 null] with root cause
org.springframework.web.client.HttpServerErrorException$InternalServerException: 500 null
at org.springframework.web.client.HttpServerErrorException.create(HttpServerErrorException.java:7
9) ~[spring-web-5.1.4.RELEASE.jar:5.1.4.RELEASE]
...

```

9.16 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/DistanciaRestClient.java

// anotações ...
public class DistanciaRestClient {

    // código omitido ...

    @Retryable(maxAttempts=5)
    @Retryable(maxAttempts=5, backoff=@Backoff(delay=2000,multiplier=2))
    public void restauranteAtualizado(Restaurante restaurante) {
        // código omitido ...
    }

}
```

O import a seguir deve ser adicionado:

```
import org.springframework.retry.annotation.Backoff;
```

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

9.17 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.restaurante.DistanciaRestClient  
: monólito tentando chamar distancia-service  
...  
2019-06-18 18:00:20.973 INFO 15044 --- [nio-8080-exec-8] b.c.c.e.restaurante.DistanciaRestClient  
: monólito tentando chamar distancia-service  
2019-06-18 18:00:24.994 INFO 15044 --- [nio-8080-exec-8] b.c.c.e.restaurante.DistanciaRestClient  
: monólito tentando chamar distancia-service  
2019-06-18 18:00:33.047 INFO 15044 --- [nio-8080-exec-8] b.c.c.e.restaurante.DistanciaRestClient  
: monólito tentando chamar distancia-service  
2019-06-18 18:00:49.079 INFO 15044 --- [nio-8080-exec-8] b.c.c.e.restaurante.DistanciaRestClient  
: monólito tentando chamar distancia-service  
2019-06-18 18:00:49.127 ERROR 15044 --- [nio-8080-exec-8] o.a.c.c.C.[.[/.][dispatcherServlet]  
: Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Requ  
est processing failed; nested exception is org.springframework.web.client.HttpServerErrorException  
$InternalServerException: 500 null] with root cause  
...  
...
```

9.18 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/RestaurantesController.java

// anotações ...
class RestaurantesController {

    // código omitido ...

    @PutMapping("/restaurantes/{id}")
    Restaurante atualiza(@PathVariable("id") Long id, @RequestBody Restaurante restaurante) {

        throw new RuntimeException();

        // descomente o código abaixo ...

        if (!repo.existsById(id)) {
            throw new ResourceNotFoundException();
        }
        log.info("Atualiza restaurante: " + restaurante);
        return repo.save(restaurante);
    }

}
```

MENSAGERIA E EVENTOS

10.1 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-nota-fiscal-service.git
```

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.
4. Observe o projeto. Já há configurações para:
 - Clientes REST declarativos com Feign
 - Self registration com Eureka Client

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.

Então, é gerado um XML da nota fiscal usando a biblioteca FreeMarker.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

10.2 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:

```
docker-compose up -d
```

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`.

10.3 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/AmqpPagamentoConfig.java

@EnableBinding(PagamentoSource.class)
@Configuration
class AmqpPagamentoConfig {

    static interface PagamentoSource {

        @Output
        MessageChannel pagamentosConfirmados();
    }
}
```

Os imports são os seguintes:

```
import org.springframework.cloud.stream.annotation.EnableBinding;
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.PagamentoSource;
```

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/PagamentoConfirmado.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 .

Injete 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/NotificadorPagamentoConfirmado.java
```

```
@Service  
@AllArgsConstructor  
class NotificadorPagamentoConfirmado {  
  
    private PagamentoSource source;  
  
    void notificaPagamentoConfirmado(Pagamento pagamento) {  
        Long pagamentoId = pagamento.getId();  
        Long pedidoId = pagamento.getPedidoId();  
        PagamentoConfirmado confirmado = new PagamentoConfirmado(pagamentoId, pedidoId);  
        source.pagamentosConfirmados().send(MessageBuilder.withPayload(confirmado).build());  
    }  
}
```

Faça os imports a seguir:

```
import org.springframework.messaging.support.MessageBuilder;  
import org.springframework.stereotype.Service;  
  
import br.com.caelum.eats.pagamento.AmqpPagamentoConfig.PagamentoSource;  
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; // adicionado  
  
    // código omitido ...  
  
    @PutMapping("/{id}")  
    Resource<PagamentoDto> confirma(@PathVariable Long id) {  
  
        Pagamento pagamento = pagamentoRepo.findById(id).orElseThrow(() -> new ResourceNotFoundException());  
        pagamento.setStatus(Pagamento.Status.CONFIRMADO);  
        pagamentoRepo.save(pagamento);  
  
        pagamentoConfirmado.notificaPagamentoConfirmado(pagamento); // adicionado  
  
        Long pedidoId = pagamento.getPedidoId();
```

```

pedidoClient.avisaQueFoiPago(pedidoId);

return new PagamentoDto(pagamento);
}

// código omitido ...
}

```

10.4 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/AmqpNotaFiscalConfig.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.EnableBinding;
import org.springframework.cloud.stream.annotation.Input;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.SubscribableChannel;

import br.com.caelum.notafiscal.AmqpNotaFiscalConfig.PagamentoSink;
```

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/ProcessadorDePagamentos.java
```

```
// anotações ...
class ProcessadorDePagamentos {

    // código omitido ...

    @StreamListener(PagamentoSink.PAGAMENTOS_CONFIRMADOS) // adicionado
    void processaPagamento(PagamentoConfirmado pagamento) {
        // código omitido ...
    }
}
```

Faça os imports adequados:

```
import org.springframework.cloud.stream.annotation.StreamListener;
import br.com.caelum.notafiscal.AmqpNotaFiscalConfig.PagamentoSink;
```

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

10.5 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-spring-cloud-stream
```

```
cd ~/Desktop/fj33-eats-nota-fiscal-service  
git checkout -f cap11-evento-de-pagamento-confirmado-com-spring-cloud-stream
```

Reinic peace 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>

```

10.6 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=notafiscal
```

10.7 EXERCÍCIO: COMPETING CONSUMERS E DURABLE SUBSCRIBER COM CONSUMER GROUPS

- 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*.

- 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!

- 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*.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

10.8 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

@Configuration
@EnableWebSocketMessageBroker
class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/pedidos", "/parceiros/restaurantes");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/socket").setAllowedOrigins(" * ").withSockJS();
    }
}
```

Não esqueça dos imports:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;
import org.springframework.web.socket.config.annotation.WebSocketMessageBrokerConfigurer;
```

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!

10.9 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.EnableBinding;
import org.springframework.cloud.stream.annotation.Output;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.MessageChannel;

import br.com.caelum.eats.AmqpPedidoConfig.AtualizacaoPedidoSource;

```

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 métodos `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.withPayload(dto).build());
        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.withPayload(dto).build());
    }
}

```

```
import org.springframework.messaging.support.MessageBuilder;
import br.com.caelum.eats.AmqpPedidoConfig.AtualizacaoPedidoSource;
```

10.10 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/AmqpApiGatewayConfig.java
```

```
@EnableBinding(AtualizacaoPedidoSink.class)
@Configuration
class AmqpApiGatewayConfig {

    static interface AtualizacaoPedidoSink {

        String PEDIDO_COM_STATUS_ATUALIZADO = "pedidoComStatusAtualizado";

        @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=apigateway
```

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") + "/pedidos/pendentes",  
pedido);  
        }  
    }  
}
```

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.AtualizacaoPedidoSink;  
import lombok.AllArgsConstructor;
```

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

10.11 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.

CONTRATOS

11.1 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/restaurantes`.

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/RestaurantesBase.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(restaurantesController);

        Mockito.when(restauranteRepository.insert(Mockito.any(Restaurante.class)))
            .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.RestaurantesController;
import io.restassured.module.mockmvc.RestAssuredMockMvc;

import io.restassured.module.mockmvc.RestAssuredMockMvc;
```

Altere a classe `RestaurantesController`, tornando-a pública:

```
// anotações omitidas ...
public class RestaurantesController { // modificado

// código omitido ...

}
```

Também torne pública a interface `RestauranteRepository`:

```
public interface RestauranteRepository extends MongoRepository<Restaurante, Long> {

// código omitido ...

}
```

Abra um Terminal e, no diretório do serviço de distância, execute:

```
mvn clean install
```

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.java

public class RestaurantesTest extends RestaurantesBase {

    @Test
    public void validate_deveAdicionarNovoRestaurante() throws Exception {
        // given:
        MockMvcRequestSpecification request = given()
            .header("Content-Type", "application/json")
            .body("{\"id\":2,\"cep\":\"71500-000\",\"tipoDeCozinhaId\":1}");

        // when:
        ResponseOptions response = given().spec(request)
            .post("/restaurantes");

        // then:
        assertThat(response.statusCode()).isEqualTo(201);
        assertThat(response.header("Content-Type")).matches("application/json.*");
        // and:
        DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
        assertThatJson(parsedJson).field("[\"tipoDeCozinhaId\"]").isEqualTo(1);
        assertThatJson(parsedJson).field("[\"cep\"]").isEqualTo("71500-000");
        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.

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

11.2 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 balanceamento 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-service:+:stubs:9992", stubsMode = Stubs
Mode.LOCAL)
public class DistanciaRestClientWiremockTest {

    private DistanciaRestClient distanciaClient;

    @Before
    public void before() {
        RestTemplate restTemplate = new RestTemplate();
        distanciaClient = new DistanciaRestClient(restTemplate, "http://localhost:9992");
    }

    @Test
    public void deveAdicionarUmNovoRestaurante() {
        TipoDeCozinha tipoDeCozinha = new TipoDeCozinha(1L, "Chinesa");

        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.StubRunnerProperties.StubsMode;
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`:

```

# f33-eats-monolito-modular/eats-restaurante/src/main/java(br/com/caelum/eats/restaurante/DistanciaRestClient.java

@Slf4j
@Service
public class DistanciaRestClient { // modificado

    // código omitido ...

    public DistanciaRestClient(RestTemplate restTemplate, // modificado
                               @Value("${configuracao.distancia.service.url}") String distanciaServiceUrl) {
        this.distanciaServiceUrl = distanciaServiceUrl;
        this.restTemplate = restTemplate;
    }

    public void novoRestauranteAprovado(Restaurante restaurante) { // modificado
        // 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] WireMock
      : Admin request received:
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",
  "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] WireMock:
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]

```

11.3 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:

```
mvn clean install
```

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-service/target/eats-distancia-service-0.0.1-SNAPSHOT.jar to /home/<USUARIO-DO-CURSO>/.m2/repository/br/com/caelum/eats-distancia-service/0.0.1-SNAPSHOT/eats-distancia-service-0.0.1-SNAPSHOT.jar  
[INFO] Installing /home/<USUARIO-DO-CURSO>/Desktop/fj33-eats-distancia-service/pom.xml to /home/<USUARIO-DO-CURSO>/.m2/repository/br/com/caelum/eats-distancia-service/0.0.1-SNAPSHOT/eats-distancia-service-0.0.1-SNAPSHOT.pom  
[INFO] Installing /home/<USUARIO-DO-CURSO>/Desktop/fj33-eats-distancia-service/target/eats-distancia-service-0.0.1-SNAPSHOT-stubs.jar to /home/<USUARIO-DO-CURSO>/.m2/repository/br/com/caelum/eats-distancia-service/0.0.1-SNAPSHOT/eats-distancia-service-0.0.1-SNAPSHOT-stubs.jar  
[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!

11.4 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/deveNotificarPagamentosConfirmados.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/PagamentosConfirmadosBase.java)
```

```
@ImportAutoConfiguration(exclude=DataSourceAutoConfiguration.class)
@MockBeans(@MockBean(PagamentoRepository.class))
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.NONE)
@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.ImportAutoConfiguration;
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;
import org.springframework.boot.test.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.boot.AutoConfigureMessageVerifier;
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/NotificadorPagamentoConfirmado.java
```

```
// anotações omitidas ...
public class NotificadorPagamentoConfirmado { // modificado
    // código omitido ...
    public void notificaPagamentoConfirmado(Pagamento pagamento) { // modificado
        // código omitido ...
    }
}
```

Torne a interface PagamentoRepository pública:

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

```
public interface PagamentoRepository extends JpaRepository<Pagamento, Long> { // modificado
}
```

Ajuste os imports na classe PagamentosConfirmadosBase :

```
# fj33-eats-pagamento-service/src/test/java/br/com/caelum/eats/pagamento/base/PagamentosConfirmadosBase.java
```

```
import br.com.caelum.eats.pagamento.NotificadorPagamentoConfirmado;
import br.com.caelum.eats.pagamento.Pagamento;
```

```
import br.com.caelum.eats.pagamento.PagamentoRepository;
```

Faça o build do Maven:

```
mvn clean install
```

Depois da execução do build, o Spring Cloud Contract deve ter gerado a classe `ConfirmadosTest`:

```
# f33-eats-pagamento-service/target/generated-test-sources/contracts(br/com/caelum/eats/pagamento/base/pagamentos/ConfirmadosTest.java)
```

```
public class ConfirmadosTest extends PagamentosConfirmadosBase {  
  
    @Inject ContractVerifierMessaging contractVerifierMessaging;  
    @Inject ContractVerifierObjectMapper contractVerifierObjectMapper;  
  
    @Test  
    public void validate_deveAdicionarNovoRestaurante() throws Exception {  
        // when:  
        novoPagamentoConfirmado();  
  
        // then:  
        ContractVerifierMessage response = contractVerifierMessaging.receive("pagamentosConfirmados");  
        assertThat(response).isNotNull();  
        assertThat(response.getHeader("contentType")).isNotNull();  
        assertThat(response.getHeader("contentType").toString()).isEqualTo("application/json");  
        // and:  
        DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload()));  
        assertThatJson(parsedJson).field("[pedidoId]").isEqualTo(3);  
        assertThatJson(parsedJson).field("[pagamentoId]").isEqualTo(2);  
    }  
}
```

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.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

11.5 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/ProcessadorDePagamentosTest.java

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
@AutoConfigureStubRunner(ids = "br.com.caelum:eats-pagamento-service", stubsMode = StubRunnerProperties.StubsMode.LOCAL)
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("<xml>...</xml>");

        stubTrigger.trigger("pagamento_confirmado");

        ArgumentCaptor<PagamentoConfirmado> pagamentoArg = ArgumentCaptor.forClass(PagamentoConfirmado.class);

        Mockito.verify(processadorPagamentos).processaPagamento(pagamentoArg.capture());

        PagamentoConfirmado pagamentoConfirmado = pagamentoArg.getValue();
        Assert.assertEquals(2L, pagamentoConfirmado.getPagamentoId().longValue());
        Assert.assertEquals(3L, pagamentoConfirmado.getPedidoId().longValue());
    }
}
```

}

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.AutoConfigureStubRunner;
import org.springframework.cloud.contract.stubrunner.spring.StubRunnerProperties;
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!

11.6 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:

```
mvn clean install
```

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-service/target/eats-pagamento-service-0.0.1-SNAPSHOT.jar to /home/<USUARIO-DO-CURSO>/.m2/repository/br/com/caelum/eats-pagamento-service/0.0.1-SNAPSHOT/eats-pagamento-service-0.0.1-SNAPSHOT.jar
[INFO] Installing /home/<USUARIO-DO-CURSO>/Desktop/fj33-eats-pagamento-service/pom.xml to /home/<USUARIO-DO-CURSO>/.m2/repository/br/com/caelum/eats-pagamento-service/0.0.1-SNAPSHOT/eats-pagamento-service-0.0.1-SNAPSHOT.pom
[INFO] Installing /home/<USUARIO-DO-CURSO>/Desktop/fj33-eats-pagamento-service/target/eats-pagamento-service-0.0.1-SNAPSHOT-stubs.jar to /home/<USUARIO-DO-CURSO>/.m2/repository/br/com/caelum/eats-pagamento-service/0.0.1-SNAPSHOT/eats-pagamento-service-0.0.1-SNAPSHOT-stubs.jar
[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!

EXTERNAL CONFIGURATION

12.1 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.EnableConfigServer;
```

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:/configs
```

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://localhost:8761/eureka/}
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

12.2 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_URL:http://localhost:8761/eureka/}

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`.

12.3 EXERCÍCIO: EXTERNALIZANDO CONFIGURAÇÕES PARA O 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-server.git
```

No Eclipse, no workspace de microservices, importe o projeto `config-server`, usando o menu *File > Import > Existing Maven Projects*.

Execute a classe `ConfigServerApplication`.

2. Obtenha a branch `cap13-configuracao-externalizada-para-o-config-server` dos projetos dos serviços de pagamentos, de distância e de nota fiscal, do monólito e do API Gateway:

```
cd ~/Desktop/fj33-eats-pagamento-service
git checkout -f cap13-configuracao-externalizada-para-o-config-server
```

```

cd ~/Desktop/fj33-eats-distancia-service
git checkout -f cap13-configuracao-externalizada-para-o-config-server

cd ~/Desktop/fj33-eats-nota-fiscal-service
git checkout -f cap13-configuracao-externalizada-para-o-config-server

cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap13-configuracao-externalizada-para-o-config-server

cd ~/Desktop/fj33-api-gateway
git checkout -f cap13-configuracao-externalizada-para-o-config-server

```

3. Reinicie todos os serviços. Garanta que a UI esteja no ar. Teste a aplicação, por exemplo, fazendo um pedido até o final e confirmando-o no restaurante. Deve funcionar!

12.4 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=classpath:/configs

spring.cloud.config.server.git.uri=file://${user.home}/Desktop/config-repo

```

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/organizacao/repositorio-de-configuracoes
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/repositorio-de-configuracoes

```

É possível manter as chaves SSH no próprio `application.properties` do Config Server.

O Config Server ainda tem como backend para as configurações:

- BD acessado por JDBC

- Redis
- AWS S3
- CredHub, um gerenciador de credenciais da Cloud Foundry
- Vault, um gerenciador de credenciais da HashiCorp

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

12.5 EXERCÍCIO: REPOSITÓRIO GIT LOCAL NO CONFIG SERVER

1. 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`.

2. 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://localhost:8761/eureka/}
```

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"
```

3. 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/<USUARIO-DO-CURSO>/Desktop/config-repo/application.properties",  
            "source": {  
                "spring.rabbitmq.username": "eats",  
                "spring.rabbitmq.password": "caelum123",  
                "eureka.client.serviceUrl.defaultZone": "${EUREKA_URI:http://localhost:8761/eureka/}"  
            }  
        }  
    ]  
}
```

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.

4. 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.

12.6 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-secredo-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-secredo-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?createDatabaseIfNotExist=true
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?createDatabaseIfNotExist=true
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
```

12.7 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-o-config-server
```

```
cd ~/Desktop/fj33-eats-pagamento-service
git checkout -f cap13-movendo-configuracoes-especificas-para-o-config-server
```

```
cd ~/Desktop/fj33-eats-distancia-service
git checkout -f cap13-movendo-configuracoes-especificas-para-o-config-server
```

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?createDatabaseIfNotExist=true
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.



Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

MONITORAMENTO E OBSERVABILIDADE

13.1 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>

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

13.2 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>

13.3 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/HystrixDashboardApplication.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.EnableHystrixDashboard;
```

No arquivo `application.properties`, modifique a porta para 7777 :

```
# fj33-hystrix-dashboard/src/main/resources/application.properties

server.port=7777
```

13.4 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-dashboard.git
```

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/restaurante-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.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

13.5 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.EnableDiscoveryClient;
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

```

13.6 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</artifactId>
</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.EnableDiscoveryClient;
import org.springframework.cloud.netflix.turbine.EnableTurbine;
import org.springframework.cloud.netflix.turbine.stream.EnableTurbineStream;
```

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=springCloudHystrixStream
```

13.7 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.git
```

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":1562070789955}  
  
: ping  
data: {"reportingHostsLast10Seconds":0,"name":"meta","type":"meta","timestamp":1562070792956}  
  
: ping  
data: {"rollingCountFallbackSuccess":0,"rollingCountFallbackFailure":0,"propertyValue_circuitBreakerRequestVolumeThreshold":20,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"latencyTotal_mean":0,"rollingMaxConcurrentExecutionCount":0,"type":"HystrixCommand","rollingCountResponsesFromCache":0,"rollingCountBadRequests":0,"rollingCountTimeout":0,"propertyValue_executionIsolationStrategy":"THREAD","rollingCountFailure":0,"rollingCountExceptionsThrown":0,"rollingCountFallbackMissing":0,"threadPool":"monolito","latencyExecute_mean":0,"isCircuitBreakerOpen":false,"errorCount":0,"rollingCountSemaphoreRejected":0,"group":"monolito","latencyTotal":{"0":0,"99":0,"100":0,"25":0,"90":0,"50":0,"95":0,"99.5":0,"75":0}, "requestCount":0,"rollingCountCollapsedRequests":0,"rollingCountShortCircuited":0,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"latencyExecute":{"0":0,"99":0,"100":0,"25":0,"90":0,"50":0,"95":0,"99.5":0,"75":0}, "rollingCountEmit":0,"currentConcurrentExecutionCount":1,"propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"errorPercentage":0,"rollingCountThreadPoolRejected":0,"propertyValue_circuitBreakerEnabled":true,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyValue_requestCacheEnabled":true,"rollingCountFallbackRejection":0,"propertyValue_requestLogEnabled":true,"rollingCountFallbackEmit":0,"rollingCountSuccess":0,"propertyValue_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_circuitBreakerErrorThresholdPercentage":50,"propertyValue_circuitBreakerForceClosed":false,"name":"RestauranteRestClient#orId(Long)","reportingHosts":1,"propertyValue_executionIsolationThreadPoolKeyOverride":"null","propertyValue_executionIsolationThreadTimeoutInMilliseconds":1000,"propertyValue_executionTimeoutInMilliseconds":1000}
```

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.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

13.8 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:

```
docker-compose up
```

3. Acesse a UI Web do Zipkin pelo navegador através da URL:

<http://localhost:9411/zipkin/>

13.9 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

13.10 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).

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

13.11 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

server.port=6666
```

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
```

13.12 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-server.git
```

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

14.1 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.caelum</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-seguranca</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-seguranca do monólito, as configurações de autorização dos endpoints que foram movidos:

```
# fj33-eats-monolito-modular/eats/eats-seguranca/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/**", "/tipos-de-cozinha/**", "/formas-de-pagamento/**").permitAll()
            .antMatchers("/actuator/**").permitAll()
            .antMatchers("/admin/**").hasRole(Role.ROLES.ADMIN.name())
            // 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
- RestauranteParaServiçoDeDistancia
- 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.

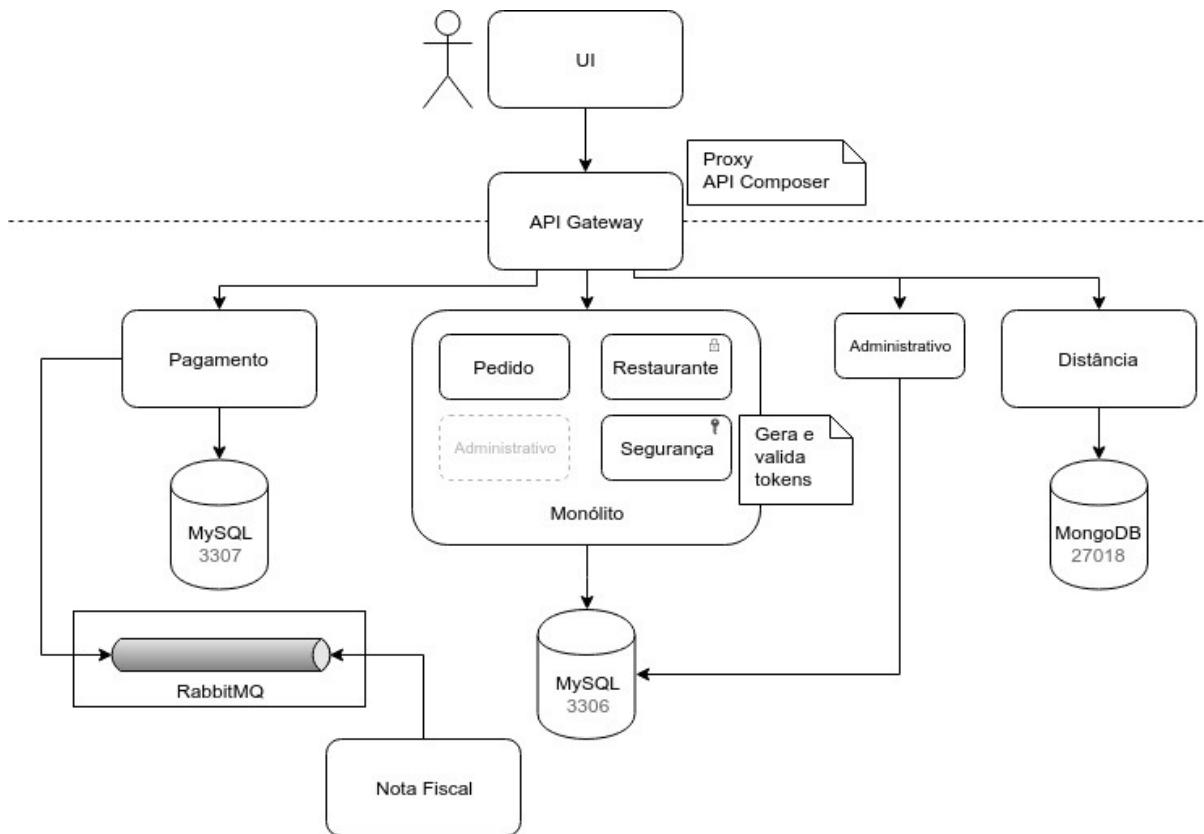


Figura 14.1: Serviço administrativo extraído do monólito

Saber inglês é muito importante em TI

alura língua

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

14.2 EXERCÍCIO: UM SERVIÇO ADMINISTRATIVO

1. 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=
```

2. Clone o projeto `fj33-eats-administrativo-service` para o seu Desktop:

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

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!

14.3 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.

14.4 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.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

14.5 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 contêm 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?

14.6 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.
eyJpc3MiOiJDYWVsdW0gRWF0cyIsInN1YiI6IjIiLCJyb2xlcI6WyJQQVJDRU1STyJdLCJ1c2VybmFtZSI6ImxvbmdmdSIsImhdCI6MTU2NjQ50DA5MSwiZXhwIjoxNTY3MTAyODkxfQ.
G0wiEeJMP9t0tV2lQpNiDU211WKL6h5Z60kNcA-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:

```
{"alg": "HS256"}
```

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,
`eyJpc3MiOiJDYWVsdW0gRWF0cyIsInN1YiI6IjIiLCJyb2xlcI6WyJQQVJDRU1STyJdLCJ1c2VybmFtZSI6ImxvbmdmdSIsImhdCI6MTU2NjQ50DA5MSwiZXhwIjoxNTY3MTAyODkxfQ` , 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, `G0wiEeJMP9t0tV21QpNiDU211WKL6h5Z60kNcA-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 espirado 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*.

14.7 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-segurança` 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-segurança/src/main/java/br/com/caelum/eats/segurança/JwtAuthenticationFilter.java
```

```
@Component
@AllArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private JwtTokenManager tokenManager;
    private UserService usersService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
        throws ServletException, IOException {
        String jwt = getTokenFromRequest(request);
        if (tokenManager.isValid(jwt)) {
            Long userId = tokenManager.getUserIdFromToken(jwt);
            UserDetails userDetails = usersService.loadUserById(userId);
            UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(userDetails,
                null, userDetails.getAuthorities());
            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
        chain.doFilter(request, response);
    }

    private String getTokenFromRequest(HttpServletRequest request) {
        // 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-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 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("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).parseClaimsJws(jwt).getBody();
    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 extrairmos 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*.

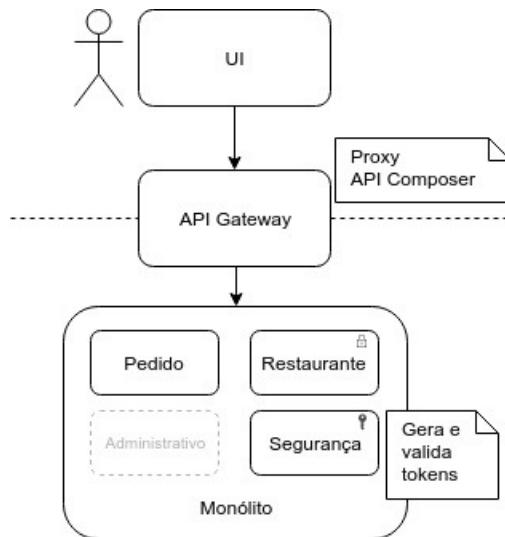


Figura 14.2: Geração e validação de tokens no módulo de segurança do monólito

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

14.8 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)**.

14.9 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.

14.10 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.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

14.11 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.

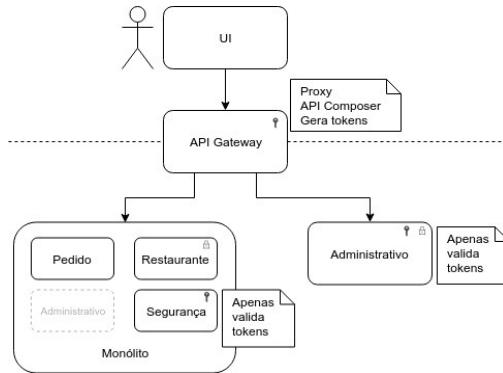


Figura 14.3: Geração de tokens no API Gateway e validação no módulo de segurança do monólito e no serviço Administrativo

14.12 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=

#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-segurança do monólito para o pacote br.com.caelum.apigateway.segurança 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) throws Exception {
        auth.userDetailsService(userService).passwordEncoder(passwordEncoder);
    }

    @Override
    @Bean(BeanIds.AUTHENTICATION_MANAGER)
    public AuthenticationManager authenticationManagerBean() throws Exception {
        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.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
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/JwtTokenManager.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/JwtTokenManager.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 '{"username":"longfu", "password":"123456"}' http://localhost:9999/auth
```

O retorno obtido será algo como:

```
{"userId":2,"username":"longfu","roles":["PARCEIRO"],"token":"eyJhbGciOiJIUzI1NiJ9eyJpc3MiOiJDYwVsdW0gRWF0cyIsInN1YiI6IjIiLCJyb2xlcI6WyJQQVJDRU1STyJdLCJ1c2VybmFtZSI6ImxvbmdmdSISImhdCI6MTU20TU1MDYyOCwiZXhwIjoxNTcwMTU1NDI4fQ.S1V7aBNN206NpxPEEAiibtluJD9Bd-gHPK-MaUHcgxs"}
```

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.

14.13 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 AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userService).passwordEncoder(this.passwordEncoder);
}

@Override
@Bean(BeanIds.AUTHENTICATION_MANAGER)
public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
}

}

}

```

Remove os seguintes imports:

```
# fj33-eats-monolito-modular/eats/eats-seguranca/src/main/java/br/com/caelum/eats/SecurityConfig.java
```

```

import org.springframework.context.annotation.Bean;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.BeanIds;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.crypto.password.PasswordEncoder;
import br.com.caelum.eats.seguranca.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-seguranca/src/main/java/br/com/caelum/eats/seguranca/JwtTokenManager.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) {
    }

    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-segurança/src/main/java/br/com/caelum/eats/segurança/JwtTokenManager.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;

    @NotBlank @JsonIgnore
    private String name;

    @NotBlank @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.NotBlank;

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.java
```

```
@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.java  
  
import javax.persistence.OneToMany;  
import br.com.caelum.eats.segurança.User;
```

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/RestauranteController.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());  
        restaurante.setUserId(doBD.getUserId()); // 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/RestauranteRepository.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/RestauranteRepository.java

import br.com.caelum.eats.seguranca.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/RestauranteAuthorizationService.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.findByUser(user);
            Restaurante restaurante = restauranteRepo.findById(user.getId());
            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 eyJhbGciOiJIUzI1NiJ9eyJpc3Mi0iJDYWVsdW0gRWF0cyIsInN1YiI6IjIiLCJyb2xlcI6WyJQQVJDRUlSTyJdLCJ1c2VybmdmdSISImIhdCI6MTU2OTU1MDYyOCwiZXhwIjoxNTcwMTU1NDI4fQ.S1V7aBN206NpxPEEalibtluJD9Bd-gHPK-MaUHcgxs' http://localhost:8080/parceiros/restaurantes/1
```

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 China aqui do seu lado.", "cep":"70238500", "endereco":"ShC/SUL COMERCIO LOCAL QD 404-BL D LJ 17-ASA SUL", "taxaDeEntregaEmReais":6.00, "tempoDeEntregaMinimoEmMinutos":40, "tempoDeEntregaMaximoEmMinutos":25, "aprovado":true, "tipoDeCozinhaId":1}
```

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.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

14.14 EXERCÍCIO: AUTENTICAÇÃO NO API GATEWAY E AUTORIZAÇÃO NO MONÓLITO

1. Faça checkout da branch `cap15-autenticacao-no-api-gateway-e-autorizacao-nos-servicos` nos projetos 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-nos-servicos  
  
cd ~/Desktop/fj33-eats-ui  
git checkout -f cap15-autenticacao-no-api-gateway-e-autorizacao-nos-servicos
```

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 `userAuthorities` . 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 '{"username":"longfu", "password":"123456"}'  
http://localhost:9999/auth
```

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":"eyJhbGciOiJIUzI1NiJ9.eyJpc3Mi0iJDYWVW  
sdW0gRWF0cyIsInN1YiI6IjIiLCJyb2xlcyI6WyJQQVJDRU1STyJdLCJ1c2VybmFtZSI6ImxvbmdmdSIsImhdCI6MTU20TU1M  
DY0CwiZXhwIjoxNTcwMTU1NDI4fQ.S1V7aBNN206NpxPEaIibtluJD9Bd-gHPK-MaUHcgxs"}
```

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":"O melhor da China aqui do seu lado.",  
"cep":"70238500","endereco":"Shc/SUL COMERCIO LOCAL QD 404-BL D LJ 17-ASA SUL","taxaDeEntregaEmReais":6.00,"tempoDeEntregaMinimoEmMinutos":40,"tempoDeEntregaMaximoEmMinutos":25,"aprovado":true,"tipoDeCozinhaId":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.

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 eyJhbGciOiJIUzI1NiJ9eyJpc3Mi0iJDYwVsdW0gRWF0cyIsInN1YiI6IjIiLCJyb2xlcyI6WyJQQVJDRU1STyJdLCJ1c2VybmFtZSI6ImxvbmmdmSiSImlhhdCI6MTU2OTU1MDYyOCwiZXhwIjoxNTcwMTU1NDI4fQ.S1V7aBNN206NpxPEEaIibtluJD9Bd-gHPK-MaUHcgxs' http://localhost:8080/parceiros/restaurantes/1
```

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 China aqui do seu lado.","cep":"70238500","endereco":"ShC/SUL COMERCIO LOCAL QD 404-BL D LJ 17-ASA SUL","taxaDeEntregaEmReais":6.00,"tempoDeEntregaMinimoEmMinutos":40,"tempoDeEntregaMaximoEmMinutos":25,"aprovado":true,"tipoDeCozinhaId":1}
```

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:

```
eyJpc3Mi0iJDYwVsdW0gRWF0cyIsInN1YiI6IjEiLCJyb2xlcyI6WyJBRE1JTiJdLCJ1c2VybmFtZSI6ImFkbWluIiwiawF0IjoxNTY2NTE4NzIyLCJleHAi0jE1NjcxMjM1MjJ9
```

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 eyJhbGciOiJIUzI1NiJ9eyJpc3MiOiJDYwVsdW0gRWF0cyIsInN1YiI6IjEiLCJyb2xlcyI6WyJBRE1JTidLCJ1c2VybmtZSI6ImFkbWluIiwiawF0IjoxNTY2NTE4NzIyLCJleHAiOjE1NjcxMjM1MjJ9.S1V7aBNN206NpxPEEaIibtlujD9Bd-gHPK-MaUHcgxs' http://localhost:8080/parceiros/restaurantes/1
```

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": "Unauthorized", "message": "Você não está autorizado a acessar esse recurso.", "path": "/parceiros/restaurantes/1"}
```

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 .

14.15 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.

14.16 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.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

14.17 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 *serve-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.

14.18 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.

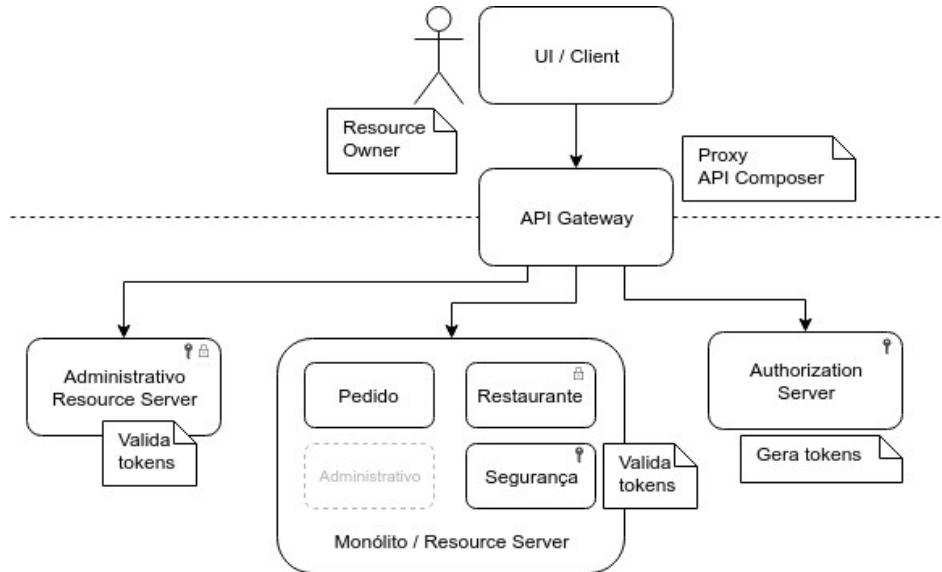


Figura 14.4: Roles OAuth no Caelum Eats

14.19 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=any'
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", "refresh_token": "0780c97f-f1d1-4a6f-82cb-c17ba5624caa", "expires_in": 43199, "scope": "any"}
```

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=bdb22855-5705-4533-b925-f1091d576db7
```

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_ADMIN"], "client_id": "eats", "scope": ["any"]}
```

Erros comuns

Se as credenciais do Client estiverem incorretas

```
curl -i -X POST --basic -u eats:SENHA_ERRADA -H 'Content-Type: application/x-www-form-urlencoded' -k -d 'grant_type=password&username=admin&password=123456&scope=any' http://localhost:8085/oauth/token
```

receberemos um status 401 (Unauthorized):

```
HTTP/1.1 401
...
>{"timestamp":"2019-08-28T14:39:58.413+0000","status":401,"error":"Unauthorized","message":"Unauthorized","path":"/oauth/token"}
```

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: application/x-www-form-urlencoded' -k -d 'grant_type=password&username=admin&password=SENHA_ERRADA&scope=any' http://localhost:8085/oauth/token
```

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: application/x-www-form-urlencoded' -d 'grant_type=password&username=admin&password=123456' http://localhost:8085/oauth/token
```

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 or the user is not allowed the requested scopes)"}
```

Se omitirmos o grant type

```
curl -i -X POST --basic -u eats:eats123 -H 'Content-Type: application/x-www-form-urlencoded' -d 'username=admin&password=123456&scope=any' http://localhost:8085/oauth/token
```

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: application/x-www-form-urlencoded' -d 'grant_type=NAO_EXISTE&username=admin&password=123456&scope=any' http://localhost:8085/oauth/token
```

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 type: NAO_EXISTE"}
```

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"}
```

Saber inglês é muito importante em TI



cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos

[Pratique seu inglês na Alura Língua.](#)

14.20 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 AuthorizationServerConfigurerAdapter {

    private final AuthenticationManager authenticationManager;
    private final OAuth2ClientProperties clientProperties;
    private final String jwtSecret;

    public OAuthServerConfiguration(AuthenticationManager authenticationManager,
                                    OAuth2ClientProperties clientProperties,
                                    @Value("${jwt.secret}") String jwtSecret) {
        this.authenticationManager = authenticationManager;
        this.clientProperties = clientProperties;
        this.jwtSecret = jwtSecret;
    }

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
            .withClient(clientProperties.getClientId())
            .secret(clientProperties.getClientSecret());
    }

    @Override
    public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {
        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$1YJxJHAbtsSCeyqgN7S1gurPZ8NSmTVA33dgPq6NqElU6qjzlpk0a
```

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: application/x-www-form-urlencoded' -d 'grant_type=password&username=admin&password=123456&scope=any' http://localhost:8085/oauth/token
```

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.eyJleHAiOjE1NjcwNTkwODUsInVzZXJfbmFtZSI6ImFkbWluiIiwiYXV0aG9yaXRpZXMiOlsciUk9MRV9BRE1JTiJdLCJqdGkiOiI20DlkMGE0ZS0xZjRmLTQ50GMt0GMzMS05YjV1YjMyZWYxYjgiLCJjbGllbnRfaWQiOjJ1YXRzIiwigc2NvcGUiOlsciYW55Il19.ZtYpX3GJPYU8UnhHRtmEtQ7SLiiZdZ0rdCRJt64ovF4", "token_type": "bearer", "expires_in": 43199, "scope": "any", "jti": "689d0a4e-1f4f-498c-8c31-9b5eb32ef1b8"}
```

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:

```
eyJleHAiOjE1NjcwNTkwODUsInVzZXJfbmFtZSI6ImFkbWluiIiwiYXV0aG9yaXRpZXMiOlsciUk9MRV9BRE1JTiJdLCJqdGkiOiI20DlkMGE0ZS0xZjRmLTQ50GMt0GMzMS05YjV1YjMyZWYxYjgiLCJjbGllbnRfaWQiOjJ1YXRzIiwigc2NvcGUiOlsciYW55Il19
```

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:

```
ZtYpX3GJPYU8UNhHRtmEtQ7SLiiZdZ0rdCRJt64ovF4
```

Como usamos o algoritmo HS256 , um algoritmo de chaves simétricas, a chave privada setada em signingKey precisa ser conhecida para validar a assinatura.

14.21 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-authorization-server.git
```

2. No workspace de microservices do Eclipse, accesse *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$1YJxJHAbtsSCeyqgN7S1gurPZ8NSmTVA33dgPq6NqElU6qjzlpk0a
```

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: application/x-www-form-urlencoded' -d 'grant_type=password&username=admin&password=123456&scope=any' http://localhost:8085/oauth/token
```

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.eyJleHAiOjE1NjcwNTkwODUsInVzZXJfbmFtZSI6ImFkbWluIiwiYXV0aG9yaXRpZXMiOlsiuK9MRV9BRE1JTiJdLCJqdGkiOjI2OD1kMGE0ZS0xZjRmLTQ50GMt0GMzMS05YjV1YjMyZWYxYjgiLCJjbGllbnRfaWQiOjJlYXRzIiwic2NvcGUiOlsiYWh55Il19.ZtYpX3GJPYU8UNhHRtmEtQ7SLiiZdZ0rdCRJt64ovF4",
  "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:

- o `AuthenticationController`
- o `AuthenticationDto`
- o `JwtTokenManager`
- o `PasswordEncoderConfig`
- o `Role`
- o `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.

14.22 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.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.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

14.23 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/com/caelum/eats/administrativo/EatsAdministrativoServiceApplication.java

@EnableResourceServer // adicionado
@EnableDiscoveryClient
@SpringBootApplication
public class EatsAdministrativoServiceApplication {

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

O import correto é o seguinte:

```
# fj33-eats-administrativo-service/src/main/java/com/caelum/eats/administrativo/EatsAdministrativoServiceApplication.java
```

```
import org.springframework.security.oauth2.config.annotation.web.configuration.EnableResourceServer;
```

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-secreto
```

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 .

```
# fj33-eats-administrativo-service/src/main/java/br/com/caelum/eats/administrativo/OAuthResourceServerConfig.java
```

```
@Configuration
class OAuthResourceServerConfig extends ResourceServerConfigurerAdapter {

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/tipos-de-cozinha/**", "/formas-de-pagamento/**").permitAll()
            .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/OAuthResourceServerConfig.java
```

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.oauth2.config.annotation.web.configuration.ResourceServerConfigurerAdapter;
```

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": 3, "tipo": "CARTAO_CREDITO", "nome": "American Express"}' http://localhost:9999/admin/formas-de-pagamento/3
```

Como resposta, teríamos um erro 401 (Unauthorized) :

```
HTTP/1.1 401
```

```
{"error": "unauthorized", "error_description": "Full authentication is required to access this resource"}  
}
```

Será necessário passar um token obtido do Authorization Server que contém o role `ADMIN` para que a chamada anterior seja bem sucedida.

14.24 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-oauth-2
```

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_CREDITO", "nome": "Amex Express"}, {"id": 2, "tipo": "CARTAO_CREDITO", "nome": "MasterCard"}, {"id": 6, "tipo": "CARTAO_DEBITO", "nome": "MasterCard Maestro"}, {"id": 5, "tipo": "VALE_REFEICAO", "nome": "Ticket Restaurante"}, {"id": 1, "tipo": "CARTAO_CREDITO", "nome": "Visa"}, {"id": 7, "tipo": "CARTAO_DEBITO", "nome": "Visa Débito"}]
```

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": 3, "tipo": "CARTAO_CREDITO", "nome": "American Express"}' http://localhost:9999/admin/formas-de-pagamento/3
```

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", error_description="Full authentication is required to access this resource"  
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 required to access this resource"}
```

Devemos incluir, no cabeçalho `Authorization`, o token JWT obtido anteriormente:

```
curl -i -X PUT -H 'Content-type: application/json' -H 'Authorization: Bearer TOKEN-JWT-AQUI' -d '{ "id": 3, "tipo": "CARTAO_CREDITO", "nome": "Amex Express"}' http://localhost:8084/admin/formas-de-pagamento/3
```

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"}
```

14.25 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:eurekaPassword@localhost:8761/eureka/
```

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

14.26 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 2048 -keystore eats-keystore.jks -validity 3650
```

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
```

14.27 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
```

14.28 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.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

14.29 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-password@tcp(192.168.33.10:3306)/"
vault write mysql/config/lease lease=1h lease_max=24h
vault write mysql/roles/readonly sql="CREATE USER '{{name}}'@'%' IDENTIFIED BY '{{password}}';GRANT SELECT ON *.* TO '{{name}}'@'%';"
```

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/>

14.30 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

15.1 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.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

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>

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>

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. 2018. Em: <https://twitter.com/pcalcado/status/963183090339385345>

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>

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 (Technical 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>

DUSSEAU, 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. *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. *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. *Richardson Maturity Model*. 2010. Em:
<https://martinfowler.com/articles/richardsonMaturityModel.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. *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>

GRIGORIK, Ilya. *High Performance Browser Networking*. O'Reilly, 2013. 400 p. Em:
<https://hpbn.co/>

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>

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. *μServices*. 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.seradio.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, Teodhor 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. *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/>

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 CIO*. 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>

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>

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>