

Curso Microservices com Spring Cloud

Curso FJ-33





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	12
1.4 Exercício: Executando o back-end	13
1.5 Exercício: Executando o front-end	13
1.6 Um negócio em expansão	14
2 Decompondo o monólito	15
2.1 Exercício opcional: decomposição em pacotes	15
2.2 Exercício: o monólito modular	15
3 Extraíndo serviços	17
3.1 Criando um microservice de pagamentos	17
3.2 Extraíndo código de pagamentos do monólito	18
3.3 Fazendo a UI chamar novo serviço de pagamentos	20
3.4 Habilitando CORS no serviço de pagamentos	22
3.5 Apagando código de pagamentos do monólito	22
3.6 Exercício: Testando o novo serviço de pagamentos	23
3.7 Criando um microservice de distância	25
3.8 Extraíndo código de distância do monólito	26
3.9 Simplificando o restaurante do serviço de distância	28
3.10 Fazendo a UI chamar serviço de distância	30
3.11 Removendo código de distância do monólito	30
3.12 Exercício: Testando o novo serviço de distância	31
4 Um pouco de Docker	34
4.1 Exercício opcional: criando uma nova instância do MySQL a partir do Docker	34
4.2 Exercício opcional: criando uma instância do MongoDB a partir do Docker	35
4.3 Exercício: gerenciando containers de infraestrutura com Docker Compose	36

5 Migrando dados	39
5.1 Separando schema do BD de pagamentos do monólito	39
5.2 Exercício: migrando dados de pagamento para schema separado	41
5.3 Exercício: migrando dados de pagamento para um servidor MySQL específico	42
5.4 Apontando serviço de pagamentos para o BD específico	44
5.5 Exercício: fazendo serviço de pagamentos apontar para o BD específico	44
5.6 Exercício: migrando dados de restaurantes do MySQL para o MongoDB	45
5.7 Configurando MongoDB no serviço de distância	47
5.8 Exercício: Testando a migração dos dados de distância para o MongoDB	49
6 Integração síncrona (e RESTful)	51
6.1 Cliente REST com RestTemplate do Spring	51
6.2 Exercício: Testando a integração entre o módulo de restaurantes do monólito e o serviço de distância	55
6.3 Cliente REST declarativo com Feign	55
6.4 Exercício: Testando a integração entre o serviço de pagamento e o módulo de pedidos do monólito	57
6.5 Exercício opcional: Spring HATEOAS e HAL	58
6.6 Exercício opcional: Estendendo o Spring HATEOAS	61
7 API Gateway	63
7.1 Implementando um API Gateway com Zuul	63
7.2 Fazendo a UI usar o API Gateway	64
7.3 Exercício: API Gateway com Zuul	65
7.4 Desabilitando a remoção de cabeçalhos sensíveis no Zuul	66
7.5 Exercício: cabeçalhos sensíveis no Zuul	66
7.6 Invocando o serviço de distância a partir do API Gateway com RestTemplate	66
7.7 Invocando o monólito a partir do API Gateway com Feign	68
7.8 Compondo chamadas no API Gateway	69
7.9 Chamando a composição do API Gateway a partir da UI	70
7.10 Exercício: API Composition no API Gateway	71
7.11 LocationRewriteFilter no Zuul para além de redirecionamentos	73
7.12 Exercício: Customizando o LocationRewriteFilter do Zuul	74
7.13 Exercício opcional: um ZuulFilter de Rate Limiting	75
8 Client Side Load Balancing com Ribbon	77
8.1 Detalhando o log de requests do serviço de distância	77
8.2 Exercício: executando uma segunda instância do serviço de distância	77
8.3 Client side load balancing no RestTemplate do monólito com Ribbon	78
8.4 Client side load balancing no RestTemplate do API Gateway com Ribbon	79

8.5 Exercício: Testando o client side load balancing no RestTemplate do monólito com Ribbon	81
8.6 Exercício: executando uma segunda instância do monólito	81
8.7 Client side load balancing no Feign do serviço de pagamentos com Ribbon	82
8.8 Client side load balancing no Feign do API Gateway com Ribbon	82
8.9 Exercício: Client side load balancing no Feign com Ribbon	83
9 Service Registry, Self Registration e Client Side Discovery	84
9.1 Implementando um Service Registry com o Eureka	84
9.2 Exercício: executando o Service Registry	85
9.3 Self Registration do serviço de distância no Eureka Server	85
9.4 Self Registration do serviço de pagamento no Eureka Server	86
9.5 Self Registration do monólito no Eureka Server	87
9.6 Self registration do API Gateway no Eureka Server	88
9.7 Exercício: Testando self registration no Eureka Server	88
9.8 Client side discovery no serviço de pagamentos	89
9.9 Client side discovery no API Gateway	90
9.10 Client side discovery no monólito	90
9.11 Exercício: Testando Client Side Discovery com Eureka Client	90
10 Circuit Breaker e Retry	92
10.1 Exercício: simulando demora no serviço de distância	92
10.2 Circuit Breaker com Hystrix	92
10.3 Exercício: Testando o Circuit Breaker com Hystrix	93
10.4 Fallback no @HystrixCommand	94
10.5 Exercício: Testando o Fallback com Hystrix	95
10.6 Exercício: Removendo simulação de demora do serviço de distância	96
10.7 Exercício: Simulando demora no monólito	96
10.8 Circuit Breaker com Hystrix no Feign	97
10.9 Exercício: Testando a integração entre Hystrix e Feign	97
10.10 Fallback com Feign	98
10.11 Exercício: Testando o Fallback do Feign	99
10.12 Exercício: Removendo simulação de demora do monólito	99
10.13 Exercício: Forçando uma exceção no serviço de distância	99
10.14 Tentando novamente com Spring Retry	100
10.15 Exercício: Testando o Spring Retry	101
10.16 Exponential Backoff	102
10.17 Exercício: Testando o Exponential Backoff	102
10.18 Exercício: Removendo exceção forçada do serviço de distância	103

11 Mensageria e Eventos	104
11.1 Exercício: um serviço de nota fiscal	104
11.2 Exercício: configurando o RabbitMQ no Docker	104
11.3 Publicando um evento de pagamento confirmado com Spring Cloud Stream	105
11.4 Recebendo eventos de pagamentos confirmados com Spring Cloud Stream	107
11.5 Exercício: Evento de Pagamento Confirmado com Spring Cloud Stream	109
11.6 Consumer Groups do Spring Cloud Stream	110
11.7 Exercício: Competing Consumers e Durable Subscriber com Consumer Groups	110
11.8 Configurações de WebSocket para o API Gateway	112
11.9 Publicando evento de atualização de pedido no monólito	113
11.10 Recebendo o evento de atualização de status do pedido no API Gateway	114
11.11 Exercício: notificando novos pedidos e mudança de status do pedido com WebSocket e Eventos	116
12 Contratos	118
12.1 Fornecendo stubs do contrato a partir do servidor	118
12.2 Usando stubs do contrato no cliente	122
12.3 Exercício: Contract Test para comunicação síncrona	125
12.4 Definindo um contrato no publisher	126
12.5 Verificando o contrato no subscriber	130
12.6 Exercício: Contract Test para comunicação assíncrona	132
13 External Configuration	134
13.1 Implementando um Config Server	134
13.2 Configurando Config Clients nos serviços	135
13.3 Exercício: Externalizando configurações para o Config Server	136
13.4 Git como backend do Config Server	136
13.5 Exercício: repositório Git local no Config Server	137
13.6 Movendo configurações específicas dos serviços para o Config Server	139
13.7 Exercícios: Configurações específicas de cada serviço no Config Server	140
14 Monitoramento e Observabilidade	142
14.1 Exercício: expondo endpoints do Spring Boot Actuator	142
14.2 Exercício: configurando o Hystrix Dashboard	144
14.3 Exercício: agregando dados dos circuit-breakers com Turbine	145
14.4 Exercício: agregando baseado em eventos com Turbine Stream	148
14.5 Exercício: configurando o Zipkin no Docker Compose	149
14.6 Exercício: enviando informações para o Zipkin com Spring Cloud Sleuth	150
14.7 Exercício: Spring Boot Admin	151

15 Segurança	153
15.1 Extraindo um serviço Administrativo do monólito	153
15.2 Exercício: um serviço Administrativo	154
15.3 Autenticação e Autorização	155
15.4 Sessões e escalabilidade	156
15.5 REST, stateless sessions e self-contained tokens	157
15.6 JWT e JWS	157
15.7 Stateless Sessions no Caelum Eats	159
15.8 Autenticação com Microservices e Single Sign On	162
15.9 Autenticação no API Gateway e Autorização nos Serviços	162
15.10 Access Token e JWT	162
15.11 Autenticação e Autorização nos Microservices do Caelum Eats	163
15.12 Exercício Opcional: Autenticação no API Gateway	164
15.13 Exercício Opcional: Validando o token JWT e implementando autorização no Monólito	168
15.14 Deixando de reinventar a roda com OAuth 2.0	174
15.15 Roles	175
15.16 Grant Types	175
15.17 OAuth no Caelum Eats	176
15.18 Authorization Server com Spring Security OAuth 2	177
15.19 JWT como formato de token no Spring Security OAuth 2	180
15.20 Exercício: um Authorization Server com Spring Security OAuth 2	182
15.21 Resource Server com Spring Security OAuth 2	185
15.22 Exercício: protegendo o serviço Administrativo	185
15.23 Protegendo serviços de infraestrutura	187
15.24 Confidencialidade, Integridade e Autenticidade com HTTPS	188
15.25 Mutual Authentication	190
15.26 Protegendo dados armazenados	190
15.27 Rotação de credenciais	191
15.28 Segurança em um Service Mesh	192
16 Apêndice: Encolhendo o monólito	194
16.1 Desafio: extrair serviços de pedidos e de administração de restaurantes	194

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:

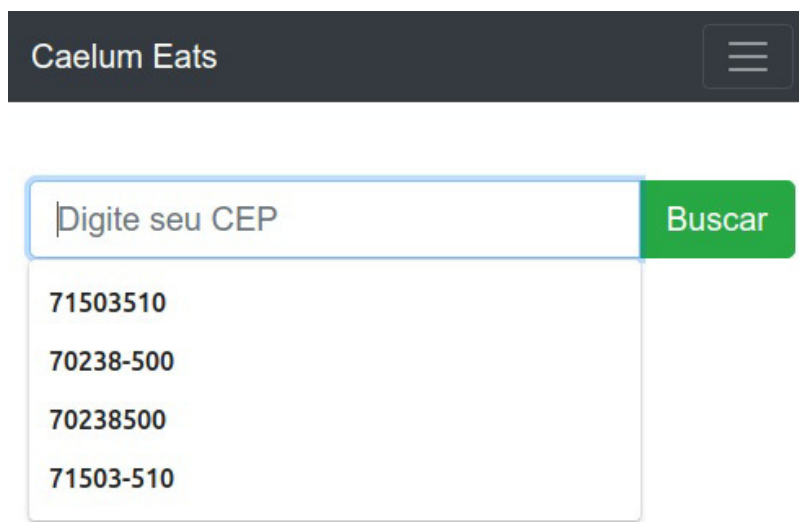
- o cliente, que efetua um pedido
- o dono do restaurante, que mantém os dados do restaurante e muda os status de pedidos pendentes
- o administrador do Caelum Eats, que mantém os dados básicos do sistema e aprova novos restaurantes

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.



A imagem mostra a interface de usuário do Caelum Eats. No topo, há uma barra escura com o texto "Caelum Eats" e um ícone de menu. Abaixo, há um campo de entrada com o placeholder "Digite seu CEP" e um botão verde "Buscar". Abaixo do campo, há uma lista de sugestões de CEPs: "71503510", "70238-500", "70238500" e "71503-510".

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.

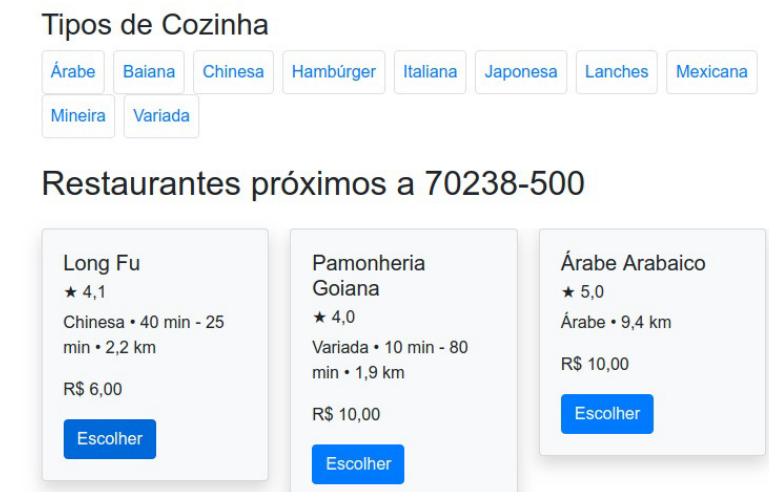


Figura 1.2: Cliente vê os restaurantes mais próximos

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

Observação

AdicionarCancelar

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
1x Gyoza Bovino - 6 unidades R\$ 23,50 <small>Sem sal</small>	Forma de pagamento <div>Ticket Restaurante ▼</div>
Taxa de entrega: R\$ 6,00	Nome no cartão <div>JOAQUIM</div>
Total: R\$ 29,50	Número do cartão <div>8989 8098 0989 0890</div>
	Data de expiração <div>February 2022</div>
	Código de Segurança <div>909</div>
	<div>Criar pagamento</div>

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	
<div>Confirmar pagamento</div>	<div>Cancelar pagamento</div>

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, CPNJ, 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:

Taxa de entrega (em R\$)

Tempo de entrega mínimo (em minutos):

Tempo de entrega máximo (em minutos):

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!



The image shows a login form with a light blue background. It contains two input fields: one for the username labeled 'Usuário:' with the text 'admin' entered, and another for the password labeled 'Senha:' with six dots representing the masked password. Below these fields is a blue button with the text 'Logar' in white.

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

Adicionar

Nome

Árabe

Editar

Remover

Baiana

Editar

Remover

Chinesa

Editar

Remover

Hambúrguer

Editar

Remover

Italiana

Editar

Remover

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	<button>Editar</button> <button>Remover</button>
Amex	Cartão de Crédito	<button>Editar</button> <button>Remover</button>
MasterCard	Cartão de Crédito	<button>Editar</button> <button>Remover</button>
MasterCard Maestro	Cartão de Débito	<button>Editar</button> <button>Remover</button>
Ticket Restaurante	Vale Refeição	<button>Editar</button> <button>Remover</button>

Figura 1.19: Administrador cadastra 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

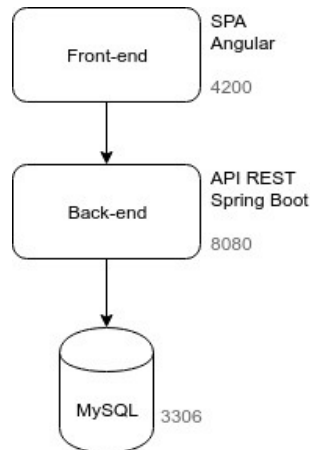


Figura 1.21: Arquitetura do Caelum Eats

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 a 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.

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 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 a execute com *CTRL+F11*.
5. Certifique-se que o projeto `fj33-eats-ui` esteja sendo executado. Acesse `http://localhost:4200` e teste algumas das funcionalidades. Tudo deve funcionar como antes!
6. Analise o projeto. Veja quais classes e interfaces são públicas e quais são *package private*. Observe as dependências entre os pacotes.

2.2 EXERCÍCIO: O MONÓLITO MODULAR

1. 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
```

2. Crie o novo workspace `/home/<usuario-do-curso>/workspace-monolito-modular` no Eclipse, clicando em *File > Switch Workspace > Other*. Troque `<usuario-do-curso>` pelo login do curso.
3. Importe, pelo menu *File > Import > Existing Maven Projects* do Eclipse, o projeto `fj33-eats-monolito-modular`.
4. Para executar a aplicação, acesse o módulo `eats-application` e execute a classe `EatsApplication` com *CTRL+F11*. Certifique-se que as versões anteriores do projeto não estão sendo executadas.
5. Com o projeto `fj33-eats-ui` no ar, teste as funcionalidades por meio de

`http://localhost:4200` . Deve funcionar!

6. Observe os diferentes módulos Maven. Note as dependências entre esses módulos, declaradas nos `pom.xml` de cada módulo.

EXTRAINDO SERVIÇOS

3.1 CRIANDO UM MICROSERVICE 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=<SUA 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 <SUA SENHA> devem ser trocados pelos valores do MySQL do monólito.

3.2 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.porIdComItens(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;

```

3.3 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 executandos, 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.

3.4 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.

3.5 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.

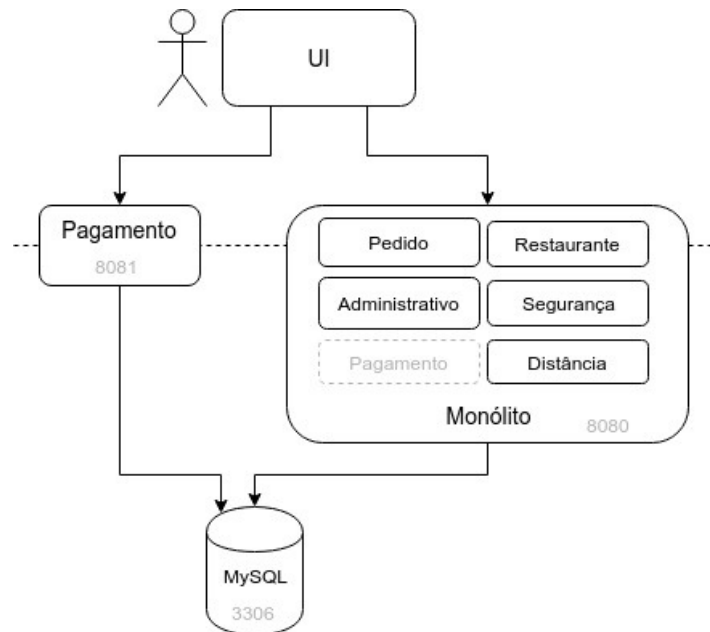


Figura 3.1: Serviço de pagamentos extraído do monólito

3.6 EXERCÍCIO: TESTANDO 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":51.80, "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-extraí-pagamento-service`, que já tem o serviço de pagamentos extraído.

```
cd ~/Desktop/fj33-eats-monolito-modular
```

```
git checkout -f cap3-extraí-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-extraí-pagamento-service`, que contém as alterações necessárias para invocar o novo serviço de pagamentos.

```
cd ~/Desktop/fj33-eats-ui
git checkout -f cap3-extraí-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.

3.7 CRIANDO UM MICROSERVICE DE DISTÂNCIA

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

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

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

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

Em *Dependencies*, adicione:

- Web
- DevTools
- Lombok
- JPA
- MySQL

Clique em *Generate Project*.

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

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

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

```
server.port = 8082
```

```
#DATASOURCE CONFIGS
```

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

```
spring.datasource.username=<SEU USUARIO>
```

```
spring.datasource.password=<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 USUARIO>` e `<SUA SENHA>` pelos valores do BD.

3.8 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 tipoDeCozinhaId, String cep) {
    TipoDeCozinha tipo = new TipoDeCozinha();
    tipo.setId(tipoDeCozinhaId);

    List<Restaurante> aprovadosDoTipoDeCozinha = restaurantes.findAllByAprovadoAndTipoDeCozinha(true, tipo, 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;
```

3.9 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;
```

3.10 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 ...
}
```

3.11 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.

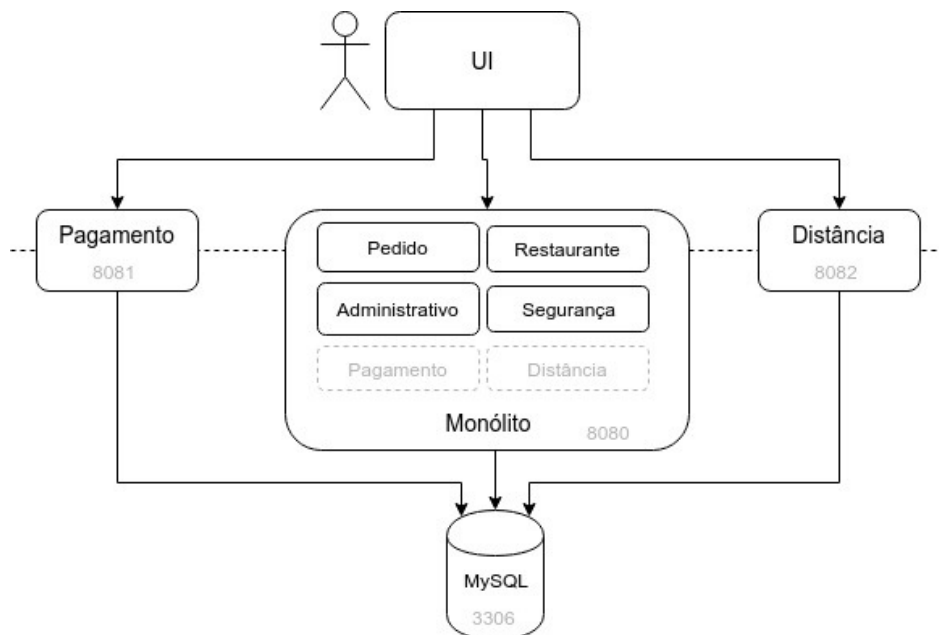


Figura 3.2: Serviço de distância extraído do monólito

3.12 EXERCÍCIO: TESTANDO O NOVO SERVIÇO DE DISTÂNCIA

1. Em um Terminal, clone o projeto do serviço de distância para o seu Desktop:

```
cd ~/Desktop
git clone https://gitlab.com/aovs/projetos-cursos/fj33-eats-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-extraí-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-extraí-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-extraí-distancia-service`, que contém as mudanças para chamar o novo serviço de distância:

```
cd ~/Desktop/fj33-eats-ui
git checkout -f cap3-extraí-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!

UM POUCO DE DOCKER

4.1 EXERCÍCIO OPCIONAL: CRIANDO UMA NOVA INSTÂNCIA DO MYSQL A PARTIR DO DOCKER

1. Abra um Terminal e baixe a imagem do MySQL 5.7 para sua máquina com o seguinte comando:

```
docker pull mysql:5.7
```

2. Suba um container do MySQL 5.7 com o seguinte comando:

```
docker 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 ao sair.
- `-d`, ou `--detach`, para rodar o container no background, imprimindo o id do container e liberando o Terminal para outros comandos.
- `-p`, ou `--publish`, que associa a porta do container ao host. No nosso caso, associamos a porta 3307 do host à porta padrão do MySQL (3306) do container.
- `--name`, define um apelido para o container.
- `-e`, ou `--env`, define variáveis de ambiente para o container. No caso, definimos a senha do usuário root por meio da variável `MYSQL_ROOT_PASSWORD`. Também definimos um database a ser criado na inicialização do container e seu usuário e senha, pelas variáveis `MYSQL_DATABASE`, `MYSQL_USER` e `MYSQL_PASSWORD`, respectivamente.

Mais detalhes sobre essas opções podem ser encontrados em: <https://docs.docker.com/engine/reference/commandline/run/>

3. Liste os containers que estão sendo executados pelo Docker com o comando:

```
docker ps
```

Deve aparecer algo como:

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

```
ats.mysql
```

É possível formatar as informações, deixando a saída do comando mais enxuta. Para isso, use a opção `--format` :

```
docker ps --format "{{.Image}}\t{{.Names}}"
```

O resultado será semelhante a:

```
mysql:5.7      eats.mysql
```

4. Acesse os logs do container `eats.mysql` com o comando:

```
docker logs eats.mysql
```

5. 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 exec -it eats.mysql mysql -upagamento -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` .

6. Pare a execução do container `eats.mysql` com o comando a seguir:

```
docker stop eats.mysql
```

4.2 EXERCÍCIO OPCIONAL: CRIANDO UMA INSTÂNCIA DO MONGODB A PARTIR DO DOCKER

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

```
docker pull mongo:3.6
```

2. Execute o MongoDB 3.6 em um container com o comando:

```
docker 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 .

3. Liste os containers, obtenha os logs de eats.mongo e pare a execução. Use como exemplo os comandos listados no exercício do MySQL.

4.3 EXERCÍCIO: GERENCIANDO CONTAINERS DE INFRAESTRUTURA COM DOCKER COMPOSE

1. No seu Desktop, defina um arquivo docker-compose.yml com o seguinte conteúdo:

```
# 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"
```

Observação: mantenha os TABs certinhos. São muito importantes em um arquivo .yaml . Em caso de dúvida, peça ajuda ao instrutor.

Caso não queira digita, o conteúdo do docker-compose.yml pode ser encontrado em:

<https://gitlab.com/snippets/1859850>

2. Mude para o diretório do docker-compose.yml , o Desktop, no caso. Suba ambos os containers, do MySQL e do MongoDB, com o comando:

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

A opção -d , ou --detach , roda os containers no background, liberando o Terminal.

Observe os containers sendo executados com o comando do Docker:

```
docker 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. 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
```

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. 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
```

5. 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` .

MIGRANDO DADOS

5.1 SEPARANDO SCHEMA DO BD DE PAGAMENTOS DO MONÓLITO

O Flyway será usado como ferramenta de migração de dados do `eats-pagamento-service`. Deve ser adicionada uma dependência no `pom.xml`:

```
# 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.

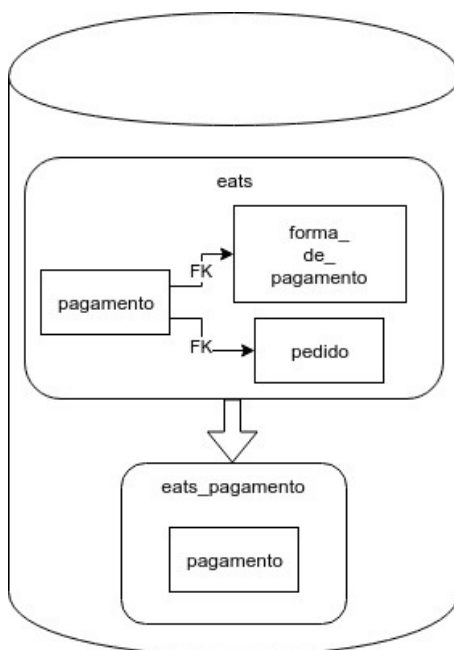


Figura 5.1: Schema separado para o BD de pagamentos

5.2 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`.

5.3 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` .

5.4 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_USUARIO>  
spring.datasource.username=pagamento
```

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

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

5.5 EXERCÍCIO: FAZENDO SERVIÇO DE PAGAMENTOS APONTAR PARA O BD ESPECÍFICO

1. Obtenha as alterações no datasource do serviço de pagamentos da branch `cap5_apontando_pagamentos_para_BD_proprio` :

```
cd ~/Desktop/fj33-eats-pagamento-service  
git checkout -f cap5_apontando_pagamentos_para_BD_proprio
```

Reinicie o serviço de pagamentos, executando a classe `EatsPagamentoServiceApplication` .

2. Abra um Terminal e crie um novo pagamento:

```
curl -X POST  
-i  
-H 'Content-Type: application/json'  
-d '{ "valor": 9.99, "nome": "MARIA DE SOUSA", "numero": "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!

5.6 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.

5.7 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.

O atributo `aprovado` pode ser removido, já que a migração dos dados foi feita de maneira que o database de distância do MongoDB só contém restaurantes já aprovados.

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

```
@Document(collection = "restaurantes") // adicionado
@Entity-
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Restaurante {

    @Id
```

```

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

private String cep;

private Boolean aprovado;

@Column(nullable = false)
private Long tipoDeCozinhaId;
}

```

Os 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/br/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/br/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=<SUA_SENHA>

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

```

O database padrão do MongoDB é `test`. A porta padrão é `27017`.

Para saber sobre outras propriedades, consulte: <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

5.8 EXERCÍCIO: TESTANDO A MIGRAÇÃO DOS 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.

INTEGRAÇÃO SÍNCRONA (E RESTFUL)

6.1 CLIENTE REST COM RESTTEMPLATE DO SPRING

No `eats-distancia-service`, crie um Controller chamado `RestaurantesController` no pacote `br.com.caelum.eats.distancia` com um método que insere um novo restaurante e outro que atualiza um restaurante existente. Defina mensagens de log em cada método.

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

```
@RestController
@AllArgsConstructor
@Slf4j
class RestaurantesController {

    private RestauranteRepository repo;

    @PostMapping("/restaurantes")
    ResponseEntity<Restaurante> adiciona(@RequestBody Restaurante restaurante, 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:

```
# fj33-eats-monolito-modular/eats/eats-application/src/main/resources/application.properties
```

```
configuracao.distancia.service.url=http://localhost:8082
```

No módulo `eats-application` do monólito, crie uma classe `RestClientConfig` no pacote `br.com.caelum.eats`, que fornece um `RestTemplate` do Spring:

```
# fj33-eats-monolito-modular/eats/eats-application/src/main/java/br/com/caelum/eats/RestClientConfig.java
```

```
@Configuration
class RestClientConfig {

    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

Faça os imports adequados:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;
```

No módulo `eats-restaurant` do monólito, crie uma classe `RestauranteParaServicoDeDistancia` no pacote `br.com.caelum.eats.restaurant` que contém apenas as informações adequadas para o serviço de distância. Crie um construtor que recebe um `Restaurante` e popula os dados necessários:

```
# fj33-eats-monolito-modular/eats/eats-restaurant/src/main/java/br/com/caelum/eats/restaurant/RestauranteParaServicoDeDistancia.java
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
class RestauranteParaServicoDeDistancia {

    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?

6.2 EXERCÍCIO: TESTANDO A INTEGRAÇÃO ENTRE O MÓDULO DE RESTAURANTES DO MONÓLITO E O SERVIÇO DE DISTÂNCIA

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();
```

6.3 CLIENTE REST DECLARATIVO COM FEIGN

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

```
# fj33-eats-monolito-modular/eats/eats-pedido/src/main/java/br/com/caelum/eats/pedido/PedidoController.java
```

```
@PutMapping("/pedidos/{id}/pago")
void pago(@PathVariable("id") Long id) {
    Pedido pedido = repo.porIdComItens(id);
    if (pedido == null) {
        throw new ResourceNotFoundException();
    }
    pedido.setStatus(Pedido.Status.PAGO);
    repo.atualizaStatus(Pedido.Status.PAGO, pedido);
}
```

No arquivo `application.properties` de `eats-pagamento-service`, adicione uma propriedade `configuracao.pedido.service.url` que contém a URL do monólito:

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

```
configuracao.pedido.service.url=http://localhost:8080
```

No `pom.xml` de `eats-pagamento-service`, adicione uma dependência ao *Spring Cloud* na versão `Greenwich.SR2`, em `dependencyManagement`:

```
# fj33-eats-pagamento-service/pom.xml
```

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Greenwich.SR2</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Feito isso, adicione o *starter* do `OpenFeign` como dependência:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Anote a classe `EatsPagamentoServiceApplication` com `@EnableFeignClients` para habilitar o `Feign`:

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

```
}
```

6.4 EXERCÍCIO: TESTANDO A INTEGRAÇÃO ENTRE O SERVIÇO DE

PAGAMENTO E O MÓDULO DE PEDIDOS DO MONÓLITO

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**.

6.5 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;
}

@DeleteMapping("/{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;
}
}

```

3. Reinicie o serviço de pagamentos e obtenha o pagamento de um id já cadastrado:

```
curl -i http://localhost:8081/pagamentos/1
```

A resposta será algo como:

```

HTTP/1.1 200
Content-Type: application/hal+json;charset=UTF-8
Transfer-Encoding: chunked
Date: Tue, 28 May 2019 19:04:43 GMT

{
  "id":1,
  "valor":51.80,
  "nome":"ANDERSON DA SILVA",
  "numero":"1111 2222 3333 4444",
  "expiracao":"2022-07",
  "codigo":"123",
  "status":"CRIADO",
  "formaDePagamentoId":2,
  "pedidoId":1,
  "_links":{
    "self":{

```

```

        "href": "http://localhost:8081/pagamentos/1"
      },
      "confirma": {
        "href": "http://localhost:8081/pagamentos/1"
      },
      "cancela": {
        "href": "http://localhost:8081/pagamentos/1"
      }
    }
  }
}

```

Teste também a criação, confirmação e cancelamento de novos pagamentos.

4. Altere o código do front-end para usar os *link relations* apropriados ao confirmar ou cancelar um pagamento:

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

```

confirma(pagamento): Observable<any> {
  this.ajustaIds(pagamento);

  const url = pagamento._links.confirma.href; // adicionado

  return this.http.put(`${this.API}/${pagamento.id}`, 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!

6.6 EXERCÍCIO OPCIONAL: ESTENDENDO O SPRING HATEOAS

1. Crie uma classe `LinkWithMethod` que estende o `Link` do Spring HATEOAS e define um atributo adicional chamado `method`, que armazenará o método HTTP dos links. Defina um construtor que recebe um `Link` e uma `String` com o método HTTP:

```

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

@Getter
public class LinkWithMethod extends Link {

```

```

private static final long serialVersionUID = 1L;

private String method;

public LinkWithMethod(Link link, String method) {
    super(link.getHref(), link.getRel());
    this.method = method;
}
}

```

Os imports são os seguintes:

```

import org.springframework.hateoas.Link;
import lombok.Getter;

```

2. Na classe `PagamentoController`, adicione um `LinkWithMethod` na lista para os links de confirmação e cancelamento, passando o método HTTP adequado.

Use o trecho abaixo nos métodos `detalha` e `cria` de `PagamentoController`:

```

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

Link confirma = linkTo(methodOn(PagamentoController.class).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. Ajuste o código do front-end para usar o `method` de cada *link relation*:

```

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

confirma(pagamento): Observable<any> {
    const url = pagamento._links.confirma.href;

    return this.http.put(url, null);

    const method = pagamento._links.confirma.method;
    return this.http.request(method, url);
}

cancela(pagamento): Observable<any> {
    const url = pagamento._links.cancela.href;

    return this.http.delete(url);

    const method = pagamento._links.cancela.method;
    return this.http.request(method, url);
}

```

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

API GATEWAY

7.1 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
```

7.2 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 ...

}
```

7.3 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?

7.4 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=
```

7.5 EXERCÍCIO: CABEÇALHOS SENSÍVEIS NO ZUUL

1. Pare o API Gateway.

Obtenha o código da branch `cap7-cabecalhos-sensíveis-no-zuul` do projeto `fj33-api-gateway` :

```
cd ~/Desktop/fj33-api-gateway
git checkout -f cap7-cabecalhos-sensíveis-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!

7.6 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`.

7.7 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.

7.8 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`

```
# fj33-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/restaurant/1` terá como resposta um JSON com os dados do restaurante e de distância mesclados.

7.9 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}/restaurant/${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.

7.10 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
```

Reinicie 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.

7.11 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 = RequestContext.getCurrentContext().getResponseStatusCode();
                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.

7.12 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.

7.13 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.

CLIENT SIDE LOAD BALANCING COM RIBBON

8.1 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
```

8.2 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)` .

8.3 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* `distancia`, 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://distancia
```

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

8.4 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://distancia` do Ribbon. Essa propriedade será usada no `DistanciaRestClient`.

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

```
zuul.routes.distancia.url=http://localhost:8082
```

```
zuul.routes.distancia.path=/distancia/**
```

```
distancia.ribbon.listOfServers=http://localhost:8082,http://localhost:9092
```

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

Modifique a anotação `@Value` do construtor de `DistanciaRestClient` para que use a propriedade `configuracao.distancia.service.url`:

```
# fj33-api-gateway/src/main/java/br/com/caelum/apigateway/DistanciaRestClient.java
```

```
class DistanciaRestClient {  
  
    // código omitido ...  
  
    DistanciaRestClient(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/br/com/caelum/apigateway/RestClientConfig.java
```

```
@Configuration  
class RestClientConfig {  
  
    @LoadBalanced // adicionado  
    @Bean  
    RestTemplate restTemplate() {  
        return new RestTemplate();  
    }  
  
}
```

Lembrando que o import correto é:

```
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
```

8.5 EXERCÍCIO: TESTANDO O 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` .

8.6 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!

8.7 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 monólito 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 ...

}
```

8.8 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
```

8.9 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 REGISTRY, SELF REGISTRATION E CLIENT SIDE DISCOVERY

9.1 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` :

```
# fj33-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
```

9.2 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.

9.3 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` .

9.4 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/}
```

9.5 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/}
```

9.6 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/}
```

9.7 EXERCÍCIO: TESTANDO SELF REGISTRATION NO EUREKA SERVER

1. Interrompa a execução do monólito, dos serviços de pagamentos e distância e do API Gateway.

Faça o checkout da branch `cap9-self-registration-no-eureka-server` nos projetos do monólito, do API Gateway e dos serviço de pagamentos e distância:

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap9-self-registration-no-eureka-server

cd ~/Desktop/fj33-api-gateway
git checkout -f cap9-self-registration-no-eureka-server
```

```
cd ~/Desktop/fj33-eats-distancia-service
git checkout -f cap9-self-registration-no-eureka-server

cd ~/Desktop/fj33-eats-pagamento-service
git checkout -f cap9-self-registration-no-eureka-server
```

2. Pare as instâncias do serviço de distância.

Execute a *run configuration* `EatsDistanciaApplication` .

Acesse o Eureka Server pelo navegador, na URL `http://localhost:8761/` . Observe que a aplicação *DISTANCIA* aparece entre as instâncias registradas com Eureka.

Então, execute a segunda instância do serviço de distância, usando a *run configuration* `EatsDistanciaApplication` (1) .

Recarregue a página do Eureka Server e note que são indicadas duas instâncias, com suas respectivas portas. Em *Status*, deve aparecer algo como `UP (2) - 192.168.0.90:distancia:9092` , `192.168.0.90:distancia:8082` .

3. Pare o serviço de pagamento.

Em seguida, execute novamente a classe `EatsPagamentoServiceApplication` .

Com o serviço em execução, vá até a página do Eureka Server e veja que *PAGAMENTOS* está entre as instâncias registradas.

4. Pare as duas instâncias do monólito.

A seguir, execute novamente a *run configuration* `EatsApplication` .

Observe *MONOLITO* como instância registrada no Eureka Server.

Execute a segunda instância do monólito com a *run configuration* `EatsApplication` (1) .

Note o registro da segunda instância no Eureka Server, também em *MONOLITO*.

5. Pare o API Gateway.

Logo após, execute novamente `ApiGatewayApplication` .

Note, no Eureka Server, o registro da instância *APIGATEWAY*.

9.8 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
```

9.9 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
```

9.10 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
```

9.11 EXERCÍCIO: TESTANDO 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.

CIRCUIT BREAKER E RETRY

10.1 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!

10.2 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;
```

10.3 EXERCÍCIO: TESTANDO O 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	7	
Processing:	75	381 360.8	275	1557	
Waiting:	67	375 359.0	270	1527	
Total:	75	382 360.9	275	1558	

10.4 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.

10.5 EXERCÍCIO: TESTANDO O FALLBACK COM HYSTRIX

1. Acesse repetidas vezes, em um navegador, a URL a seguir:

<http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1>

Deve ocorrer, em algumas das vezes, uma exceção `HystrixRuntimeException` com as mensagens:

- *porCepEId timed-out and fallback failed.*
- *porCepEId short-circuited and fallback failed.*

2. No projeto `fj33-api-gateway`, obtenha o código da branch `cap10-fallback-no-hystrix-command`:

```
cd ~/Desktop/fj33-api-gateway
git checkout -f cap10-fallback-no-hystrix-command
```

Reinicie o API Gateway.

3. Tente acessar várias vezes a URL testada anteriormente:

<http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1>

Observe que não ocorre mais uma exceção, mas não há a informação de distância. Apenas os detalhes do restaurante são retornados.

Algo semelhante a:

```
{
```

```

    "id": 1,
    "cnpj": "98444252000104",
    "nome": "Long Fu",
    "descricao": "O melhor da China aqui do seu lado.",
    "cep": "71503510",
    "endereco": "ShC/SUL COMERCIO LOCAL QD 404-BL D LJ 17-ASA SUL",
    "taxaDeEntregaEmReais": 6,
    "tempoDeEntregaMinimoEmMinutos": 40,
    "tempoDeEntregaMaximoEmMinutos": 25,
    "aprovado": true,
    "tipoDeCozinha": {
        "id": 1,
        "nome": "Chinesa"
    },
    "restauranteId": 1
}

```

10.6 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);
    }

}

```

10.7 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 ...

}

```

10.8 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.

10.9 EXERCÍCIO: TESTANDO A 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
```

Reinicie 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.*

10.10 FALLBACK COM FEIGN

No Feign, definimos de maneira declarativa o cliente REST, por meio de uma interface.

A estratégia de Fallback na integração entre Hystrix e Feign é fornecer uma implementação para essa interface. Engenhoso!

No `api-gateway`, crie uma classe `RestauranteRestClientFallback`, que implementa a interface `RestauranteRestClient`. No método `porId`, deve ser fornecida uma lógica de fallback para o detalhamento de um restaurante. Anote essa nova classe com `@Component`, para que seja gerenciada pelo Spring.

```
# fj33-api-gateway/src/main/java/br/com/caelum/apigateway/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);
}
```

10.11 EXERCÍCIO: TESTANDO O FALLBACK DO 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/restaurant/1>

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

10.12 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/restaurant/1>

Os detalhes do restaurante devem voltar a ser exibidos!

10.13 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);
    }

}
```

10.14 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;
```

10.15 EXERCÍCIO: TESTANDO O SPRING RETRY

1. Faça o checkout da branch `cap10-retry` do monólito:

```
cd ~/Desktop/fj33-eats-monolito-modular
git checkout -f cap10-retry
```

Reinicie 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
$InternalServerError: 500 null] with root cause

org.springframework.web.client.HttpServerErrorException$InternalServerError: 500 null
    at org.springframework.web.client.HttpServerErrorException.create(HttpServerErrorException.java:7
9) ~[spring-web-5.1.4.RELEASE.jar:5.1.4.RELEASE]
    ...

```

10.16 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;
```

10.17 EXERCÍCIO: TESTANDO O EXPONENTIAL BACKOFF

1. Vá até a branch `cap10-backoff` do projeto `fj33-eats-monolito-modular` :

```
``sh 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
$InternalServerError: 500 null] with root cause
...

```

10.18 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

11.1 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.

11.2 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*.

11.3 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 ...
}

```

11.4 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;
```

11.5 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
```

Reinicie o serviço de pagamento.

Inicie o serviço de nota fiscal executando a classe `EatsNotaFiscalServiceApplication`.

2. Certifique-se que o service registry, o serviço de pagamento, o serviço de nota fiscal e o monólito estejam sendo executados.

Confirme um pagamento já existente com o cURL:

```
curl -X PUT -i http://localhost:8081/pagamentos/1
```

Observação: para facilitar testes durante o curso, a API de pagamentos permite reconfirmação de pagamentos. Talvez não seja o ideal...

Acesse a UI de gerenciamento do RabbitMQ, pela URL `http://localhost:15672`.

Veja nos gráficos que algumas mensagens foram publicadas. Veja `pagamentosConfirmados` listado em *Exchange*.

Observe, nos logs do serviço de nota fiscal, o XML da nota emitida. Algo parecido com:

```
<xml>
<loja>314276853</loja>
<nat_operacao>Almoços, Jantares, Refeições e Pizzas</nat_operacao>
<pedido>
```

```

<items>
  <item>
    <descricao>Yakimeshi</descricao>
    <un>un</un>
    <codigo>004</codigo>
    <qtde>1</qtde>
    <vlr_unit>21.90</vlr_unit>
    <tipo>P</tipo>
    <class_fiscal>21069090</class_fiscal>
  </item>
  <item>
    <descricao>Coca-Cola Zero Lata 310 ML</descricao>
    <un>un</un>
    <codigo>004</codigo>
    <qtde>2</qtde>
    <vlr_unit>5.90</vlr_unit>
    <tipo>P</tipo>
    <class_fiscal>21069090</class_fiscal>
  </item>
</items>
</pedido>
<cliente>
  <nome>Isabela</nome>
  <tipoPessoa>F</tipoPessoa>
  <contribuinte>9</contribuinte>
  <cpf_cnpj>169.127.587-54</cpf_cnpj>
  <email>isa@gmail.com</email>
  <endereco>Rua dos Bobos, n 0</endereco>
  <complemento>-</numero>
  <cep>10001-202</cep>
</cliente>
</xml>

```

11.6 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

```

11.7 EXERCÍCIO: COMPETING CONSUMERS E DURABLE SUBSCRIBER COM CONSUMER GROUPS

1. Pare o serviço de nota fiscal e confirme alguns pagamentos pelo cURL.

Note que, mesmo com o serviço consumidor parado, a mensagem é publicada no MOM.

Suba novamente o serviço de nota fiscal e perceba que as mensagens publicadas enquanto o serviço estava fora do ar **não** foram recebidas. Essa é a característica de um *non-durable subscriber*.

2. Execute uma segunda instância do serviço de nota fiscal na porta 9093 .

No workspace dos microservices, acesse o menu *Run > Run Configurations...* do Eclipse e clique

com o botão direito na configuração `EatsNotaFiscalServiceApplication` e depois clique em *Duplicate*.

Na configuração `EatsNotaFiscalServiceApplication` (1) que foi criada, acesse a aba *Arguments* e defina `9093` como a porta da segunda instância, em *VM Arguments*:

```
-Dserver.port=9093
```

Clique em *Run*. Nova instância do serviço de nota fiscal no ar!

3. Use o cURL para confirmar um pagamento. Algo como:

```
curl -X PUT -i http://localhost:9999/pagamentos/1
```

Note que o XML foi impresso nos logs das duas instâncias, `EatsNotaFiscalServiceApplication` e `EatsNotaFiscalServiceApplication` (1). Ou seja, todas as instâncias recebem todas as mensagens publicadas no exchange `pagamentosConfirmados` do RabbitMQ.

4. Em um Terminal, vá até a branch `cap11-consumer-groups` do serviço de nota fiscal:

```
cd ~/Desktop/fj33-eats-nota-fiscal-service
git checkout -f cap11-consumer-groups
```

Reinicie 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*.

11.8 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

@EnableWebSocketMessageBroker
@Configuration
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!

11.9 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;
```

11.10 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;
```

11.11 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

12.1 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 direito 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 `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.

12.2 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`.

Invoke 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`:

```
# fj33-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;

aServiceUrl) {
    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]
```

12.3 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!

12.4 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 direito 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` :

```
# fj33-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.

12.5 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 direito 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!

12.6 EXERCÍCIO: CONTRACT TEST PARA COMUNICAÇÃO ASSÍNCRONA

1. Abra um Terminal e vá até a branch `cap12-contrato-cliente-servidor` do projeto do serviço de pagamentos:

```
cd ~/Desktop/fj33-eats-pagamento-service
git checkout -f cap12-contrato-cliente-servidor
```

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-cliente-servidor`:

```
cd ~/Desktop/fj33-eats-nota-fiscal-service
git checkout -f cap12-contrato-cliente-servidor
```

Faça o refresh do projeto no Eclipse.

Clique com o botão direito na classe `ProcessadorDePagamentosTest`, 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!

EXTERNAL CONFIGURATION

13.1 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/}
```

13.2 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_URI: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`.

13.3 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!

13.4 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

13.5 EXERCÍCIO: REPOSITÓRIO GIT LOCAL NO CONFIG SERVER

1. Faça checkout da branch `cap13-repositorio-git-no-config-server` do projeto do Config Server:

```
``sh cd ~/Desktop/fj33-config-server git checkout -f cap13-repositorio-git-no-config-server
```

Reinicie 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.

13.6 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-segredo-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-segredo-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-
```

13.7 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-segredo-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.

MONITORAMENTO E OBSERVABILIDADE

14.1 EXERCÍCIO: EXPONDO ENDPOINTS DO SPRING BOOT ACTUATOR

1. Adicione, a todos os projetos, uma dependência ao *starter* do Spring Boot Actuator:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Essa dependência deve ser adicionada ao `pom.xml` do:

- módulo `eats-application` do monólito
- `eats-pagamento-service`
- `eats-distancia-service`
- `eats-nota-fiscal-service`
- `api-gateway`
- `service-registry`
- `config-server`

2. Adicione, ao `application.properties` do `config-repo`, uma configuração para expor todos os *endpoints* do Actuator disponíveis:

`config-repo/application.properties`

```
management.endpoints.web.exposure.include=*
```

Não deixe de comitar a alteração:

```
git commit -am "expõe endpoints do Spring Boot Actuator em todos os serviços"
```

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
3. Para fazer com as requisições ao Actuator do API Gateway não 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
```

4. Exponha 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=*
```

5. Reinicie 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>

14.2 EXERCÍCIO: CONFIGURANDO O HYSTRIX DASHBOARD

1. 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*.

2. Extraia o `hystrix-dashboard.zip` e copie a pasta para seu Desktop.
3. No Eclipse, no workspace de microservices, importe o projeto `hystrix-dashboard` , usando o menu *File > Import > Existing Maven Projects*.
4. 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;
```

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

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

```
server.port=7777
```

6. Execute a classe `HystrixDashboardApplication`.
7. Acesse o Hystrix Dashboard, pelo navegador, com a seguinte URL:

<http://localhost:7777/hystrix>

Coloque, na URL, o endpoint de Hystrix Stream Actuator do API Gateway:

<http://localhost:9999/actuator/hystrix.stream>

Clique em *Monitor Stream*.

Em outra aba, acesse URLs do API Gateway como as que seguem:

- <http://localhost:9999/restaurantes/1>, que exibirá o circuit breaker do monolito
- <http://localhost:9999/pagamentos/1>, que exibirá o circuit breaker do serviço de pagamentos
- <http://localhost:9999/distancia/restaurantes/mais-proximos/71503510>, que exibirá o circuit breaker do serviço de distância
- <http://localhost:9999/restaurantes-com-distancia/71503510/restaurante/1>, que exibirá os circuit breakers relacionados a composição de chamadas feita no API Gateway

14.3 EXERCÍCIO: AGREGANDO DADOS DOS CIRCUIT-BREAKERS COM TURBINE

1. 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*.

2. Extraia o `turbine.zip` e copie a pasta para seu Desktop.
3. No Eclipse, no workspace de *microservices*, importe o projeto `turbine`, usando o menu *File > Import > Existing Maven Projects*.
4. 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;
```

5. 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'
```

6. 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
```

7. Execute a classe `TurbineApplication`.

Então, acesse o Turbina 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#porId(Long)","reportingHosts":1,"propertyValue_executionIsolationThreadPoolKeyOverride":"null","propertyValue_executionIsolationThreadTimeoutInMilliseconds":1000,"propertyValue_executionTimeoutInMilliseconds":1000}
```

8. Vá novamente ao Hystrix Dashboard, pela 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, como nos passos anteriores.

Depois disso, veja os status dos circuit breakers.

14.4 EXERCÍCIO: AGREGANDO BASEADO EM EVENTOS COM TURBINE STREAM

1. No pom.xml do projeto turbine , troque a dependência ao starter do Turbine pela do Turbine Stream. Adicione também o binder do Spring Cloud Stream ao RabbitMQ:

fj33-turbine/pom.xml

```
<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>
```

2. 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'
```

3. Troque a anotação @EnableTurbine por @EnableTurbineStream na classe TurbineApplication :

fj33-turbine/src/main/java/br/com/caelum/turbine/TurbineApplication.java

```
@EnableTurbine  
@EnableTurbineStream  
@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.netflix.turbine.EnableTurbine;
```

```
import org.springframework.cloud.netflix.turbine.stream.EnableTurbineStream;
```

4. 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>
```

5. 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
```

6. Reinicie o API Gateway e o Turbine.

Acesse, pelo navegador, o Hystrix Dashboard:

<http://localhost:7777/hystrix>

Aponte para a URL do Turbine:

<http://localhost:7776/turbine.stream>

Chame o API Gateway em outra janela do navegador, como as dos exercícios anteriores.

Observe informações sobre os circuit breakers no Hystrix Dashboard.

14.5 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/>

14.6 EXERCÍCIO: ENVIANDO INFORMAÇÕES PARA O ZIPKIN COM SPRING CLOUD SLEUTH

1. 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

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
```

Não esqueça de comitar a alteração:

```
git commit -m "configuração do Spring Cloud Sleuth"
```

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 e, em *Service Name*, selecione um serviço. 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).

14.7 EXERCÍCIO: SPRING BOOT ADMIN

1. 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*.

2. Extraia o `admin-server.zip` e copie a pasta para seu Desktop.
3. No Eclipse, no workspace de microservices, importe o projeto `admin-server` , usando o menu *File* > *Import* > *Existing Maven Projects*.
4. 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;
```

5. No arquivo `application.properties` , modifique a porta para `8084` :

fj33-admin-server/src/main/resources/application.properties

```
server.port=8084
```

6. 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
```

7. Execute a classe `AdminServerApplication`.

Pelo navegador, acesse a URL:

<http://localhost:8084>

Veja informações sobre as aplicações e instâncias.

Em *Wallboard*, há uma visualização interessante do status dos serviços.

SEGURANÇA

15.1 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 `eats-administrativo` do monólito para o novo serviço `eats-administrativo-service`:

- `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 .

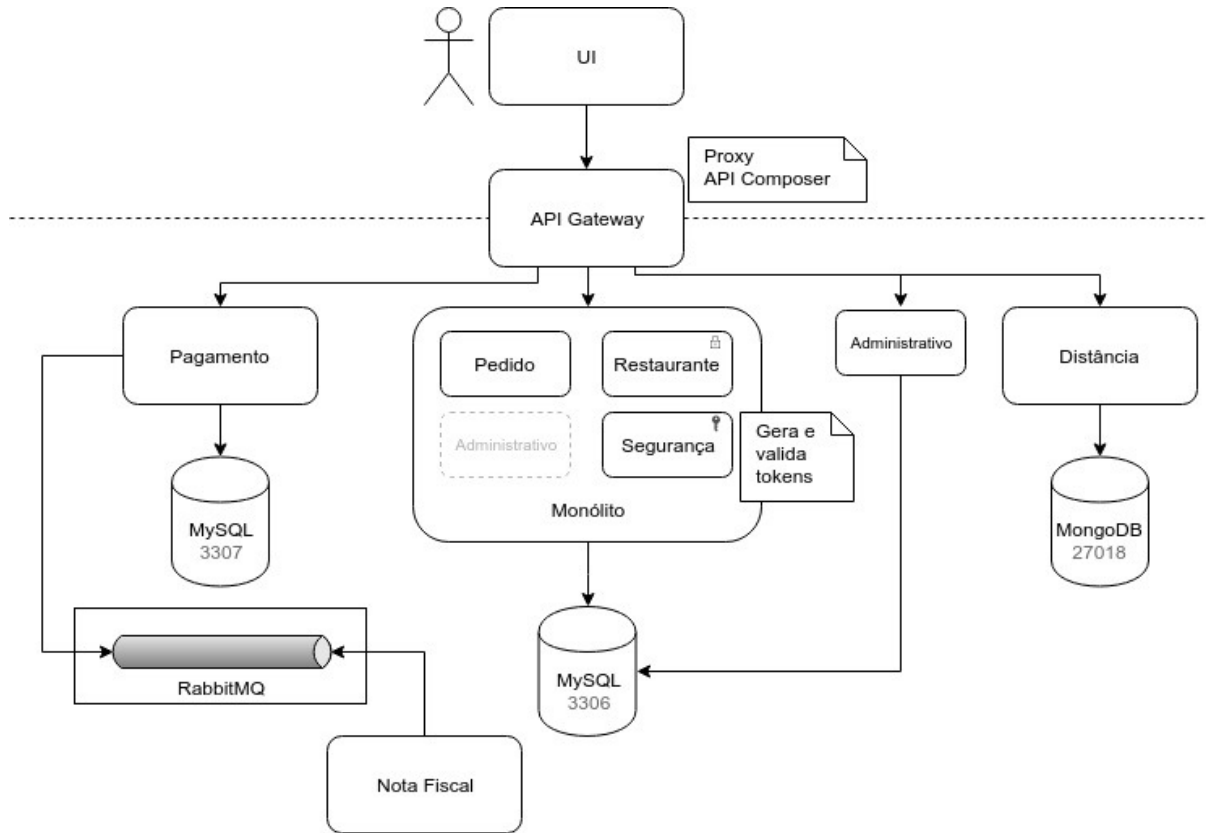


Figura 15.1: Serviço administrativo extraído do monólito

15.2 EXERCÍCIO: UM SERVIÇO ADMINISTRATIVO

1. 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
```

2. 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=
```

3. 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>
```

4. 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>
```

5. 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*.
6. 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("/atuador/**").permitAll()
            .antMatchers("/admin/**").hasRole(Role.ROLES.ADMIN.name())
            // código omitido ...
    }
}
```

15.3 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.

15.4 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.

15.5 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?

15.6 JWT E JWS

JWT (JSON Web Token) é um formato de token compacto e self-contained que serve para 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)
```

Um exemplo de um JWS usado no Caelum Eats seria o seguinte:

Os trechos anteriores podem ser decodificados de Base64 para texto normal usando um site como:

O primeiro trecho, `eyJhbGciOiJIUzI1NiJ9`, é o cabeçalho do JWS. Quando decodificado, é:

O valor de `alg` indica que foi utilizado o algoritmo HMAC (hash-based message authentication code) com SHA-256 como função de hash. Nesse algoritmo, há uma chave secreta (um texto) simétrica, e 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 livremente.

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.

158 15.6 JWT E JWS

No site <https://jwt.io/> conseguimos obter o algoritmo utilizado, os dados do payload e até validar um JWT.

Se soubermos a chave secreta, podemos verificar se a assinatura bate com o payload do JWS. Se bater, o token é válido. Dessa maneira, conseguimos garantir que não houve manipulação dos dados e, portanto, sua integridade.

Um detalhe importante é que um JWS não garante a confidencialidade dos dados. Se houver algum software bisbilhotando os dados trafegados na rede, o payload do JWS pode ser lido, já que é apenas codificado em Base64 URL encoded. A confidencialidade pode ser reforçada por meio de TLS no canal de comunicação ou por meio de JWE.

Uma grande desvantagem de um JWT é que o token é irrevogável antes de sua expiração. Isso implica que, enquanto o token não estiver expirado será válido. Por isso, implementar um mecanismo de logout pelo usuário passa a ser complicado. Poderíamos trabalhar com intervalos pequenos de expiração, mas isso afetaria a experiência do usuário, já que frequentemente a expiração levaria o usuário a efetuar novo login. Uma maneira comum de implementar logout é ter um cache com JWT invalidados. Porém, isso nos leva novamente a uma solução *stateful*.

15.7 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-seguranca` do monólito. No método `authenticate`, é gerado e retornado um token JWS.

O token JWS é armazenado em um `localStorage` no front-end. Há um *interceptor* do Angular que, antes de cada requisição AJAX, adiciona o cabeçalho `Authorization: Bearer` com o valor do token armazenado.

No back-end, a classe `JwtAuthenticationFilter` é executada a cada requisição e o token JWS é extraído dos cabeçalhos HTTP e validado. Caso seja válido, é recuperado o `sub` (Subject) e obtido o usuário do BD com seus ROLES (`ADMIN` ou `PARCEIRO`), setando um `Authentication` no contexto de segurança:

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

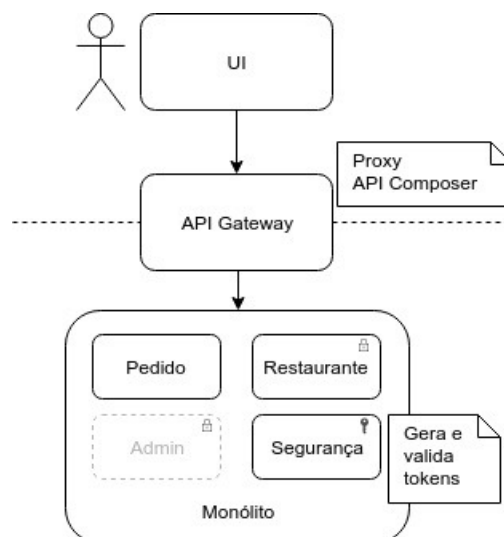


Figura 15.2: Geração e validação de tokens no módulo de segurança do monólito

15.8 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 os 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).

15.9 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.

15.10 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: ACCESS 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.

15.11 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.

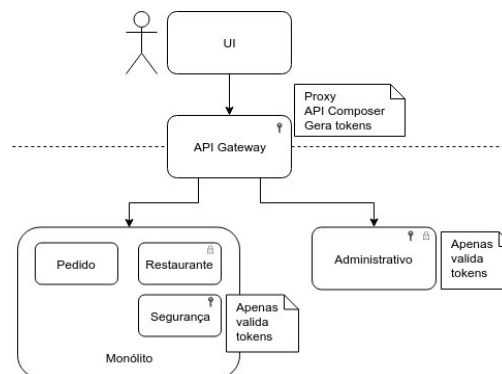


Figura 15.3: Geração de tokens no API Gateway e validação no módulo de segurança do monólito e no serviço Administrativo

15.12 EXERCÍCIO OPCIONAL: AUTENTICAÇÃO NO API GATEWAY

1. Poderíamos ter um BD específico para conter dados de usuários nas tabelas `user`, `role` e `user_authorities`. Porém, para simplificar, vamos manter os dados de usuários no BD do próprio monólito.

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>
```

2. 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-segredo-bem-secreto
jwt.expiration = 604800000
```

3. Mova as classes a seguir do módulo `eats-seguranca` do monólito para o API Gateway, no pacote `br.com.caelum.apigateway`, apagando do pacote original:

- `AuthenticationController.java`
- `AuthenticationDto.java`
- `UserInfoDto.java`
- `UserRepository.java`
- `UserService.java`
- `PasswordEncoderConfig.java`

Além dessas, copie as seguintes classes, mantendo-as também no módulo `eats-seguranca` do monólito:

- `Role.java`

- User.java

Copie também a seguinte classe do módulo `eats-application` do monólito:

- CorsConfig.java

Não esqueça de ajustar o pacote das classes copiadas.

Essas classes fazem geração e validação de tokens JWT, assim como o cadastro de novos donos de restaurante.

4. 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(final AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userService).passwordEncoder(passwordEncoder);
    }

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

5. Defina, no mesmo pacote do API Gateway, uma classe `JwtTokenManager`, responsável pela geração dos tokens. A validação e extração de informações de um token serão responsabilidade de cada serviços.

É importante adicionar o username e os roles do usuário aos *claims* do JWT.

fj33-api-gateway/src/main/java/br/com/caelum/apigateway/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();
    }
}
```

6. 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 que direciona todas as requisições para o monólito.

7. Execute `ApiGatewayApplication`, certificando-se que o Service Registry e o Config Server estão no ar.

Então, abra o terminal e simule a autenticação do administrador:

```
curl -i -X POST -H 'Content-type: application/json' -d '{"username":"admin", "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
```

```
{"username": "admin", "roles": ["ADMIN"], "token": "eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJkdYVWsdW0gRWF0cyIsInN1YiI6IjEiLCJyb2x1cyI6WyJBRElJTjIjLdCjE2c2VybmFtZSI6ImFkbWluIiwiaWF0IjoxNTY2NTE4NzIyLCJleHAiOiE1Njc5MjM1MjJ9.FmH2QkryLBxWZjt2DMKHsCmjQNCmk3hrRAC0keam5_w"}
```

15.13 EXERCÍCIO OPCIONAL: VALIDANDO O TOKEN JWT E IMPLEMENTANDO AUTORIZAÇÃO NO MONÓLITO

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

Não deixe de importar:

## 15.13 EXERCÍCIO OPCIONAL: VALIDANDO O TOKEN JWT E IMPLEMENTANDO AUTORIZAÇÃO NO MONÓLITO

2. 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-seguranca/src/main/java/br/com/caelum/eats/seguranca/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 ...
```

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

```
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Role implements GrantedAuthority {

 // código omitido...

 @Id
 private String authority;

 // restante do código...
```

- i. Altere a classe `SecurityConfig` do módulo de segurança do monólito, removendo código associado a autenticação e cadastro de novos usuários:

```
fj33-eats-monolito-modular/eats/eats-seguranca/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 BCryptPasswordEncoder bCryptPasswordEncoder;

// código omitido...

@Override
protected void configure(final AuthenticationManagerBuilder auth) throws Exception {
auth.userDetailsService(userService).passwordEncoder(bCryptPasswordEncoder);
}

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

```

3. 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
}

```

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 findById(Long userId); // modificado

 // código omitido...
}

```

Faça com que a classe `RestauranteAuthorizationTargetService` use o novo método do repository:

```

fj33-eats-monolito-modular/eats/eats-
restaurante/src/main/java/br/com/caelum/eats/restaurante/RestauranteAuthorizationService.java

``java @Service @AllArgsConstructor class RestauranteAuthorizationTargetService {

 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...

}

```

6. 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.

```
config-repo/monolito.properties
```

```
```properties
jwt.expiration = 604800000
```

1. 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/formas-de-pagamento>

ou

<http://localhost:8080/formas-de-pagamento>

Deve funcionar e retornar algo como:

```
[{"id":4,"tipo":"VALE_REFEICAO","nome":"Alelo"}, {"id":3,"tipo":"CARTAO_CREDITO","nome":"Amex"}, {"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"}]
```

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.

Abra um Terminal e tente modificar o nome de uma forma de pagamento usando o cURL:

```
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>

A resposta será um erro HTTP 401 (Unauthorized), com uma mensagem de acesso negado. Algo como:

```
HTTP/1.1 401
Date: Fri, 23 Aug 2019 00:50:11 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":"2019-08-23T00:50:11.652+0000","status":401,"error":"Unauthorized","message":"Você não está autorizado a acessar esse recurso.","path":"/admin/formas-de-pagamento/3"}
```

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 -X PUT -H 'Content-type: application/json' -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJDYWVsdW0gRWF0cyIsInN1YiI6IjEiLCJyb2x1cyI6WyJBRE1JTjJdLCJ1c2VybmFtZSI6ImFkbWluIiwiaWF0IjoxNTY2NTE4NzIyLCJleHAiOiJlNjc0MjM1MjJ9.FmH2QkryLBxWZjt2DMKHsCmjQNCmk3hrRAC0keam5_w' -d '{"id": 3, "tipo": "CARTAO_CREDITO", "nome": "American Express"}' http://localhost:9999/admin/formas-de-pagamento/3
```

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":3,"tipo":"CARTAO_CREDITO","nome":"American Express"}
```

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 ADMIN.

2. Altere o payload do JWT, definindo um valor diferente para o `sub`, o Subject, que indica o id do usuário.

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:

```
eyJpc3MiOiJDYWVsdW0gRWF0cyIsInN1YiI6IjEiLCJyb2x1cyI6WyJBRE1JTjJdLCJ1c2VybmFtZSI6ImFkbWluIiwiaWF0IjoxNTY2NTE4NzIyLCJleHAiOiJlNjc0MjM1MjJ9
```

Será exibido em *Plain Text*, um JSON parecido com:

```
{"iss":"Caelum Eats","sub":"1","roles":["ADMIN"],"username":"admin","iat":1566518722,"exp":1567123522}
```

Altere o `sub` para `2`, simulando um usuário malicioso tentando forjar um token para roubar a identidade de outro usuário, de id diferente. O texto codificado em Base 64 URL Encoding será algo como:

```
eyJpc3MiOiJDYWVsdW0gRWF0cyIsInN1YiI6IjEiLCJyb2x1cyI6WyJBRE1JTjJdLCJ1c2VybmFtZSI6ImFkbWluIiwiaWF0IjoxNTY2NTE4NzIyLCJleHAiOiJlNjc0MjM1MjJ9
```

Observe que a codificação é quase idêntica: apenas o 39º caractere foi modificado de `E` para `I`.

Através de um Terminal, use o cURL para tentar alterar uma forma de pagamento utilizando o payload modificado do JWT:

```
curl -i -X PUT -H 'Content-type: application/json' -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJDYWVsdW0gRWF0cyIsInN1YiI6IjEiLCJyb2x1cyI6WyJBRE1JTjJdLCJ1c2VybmFtZSI6ImFkbWluIiwiaWF0IjoxNTY2NTE4NzIyLCJleHAiOiJlNjc0MjM1MjJ9.FmH2QkryLBxWZjt2DMKHsCmjQNCmk3hrRAC0keam5_w' -d '{"id": 3,
```

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

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":"/admin/formas-de-pagamento/3"}
```

Teste também com o cURL o acesso direto ao monólito, usando a porta 8080 . O acesso deve ser negado, da mesma maneira.

15.14 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.

15.15 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.

15.16 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 cadastra 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.

15.17 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.

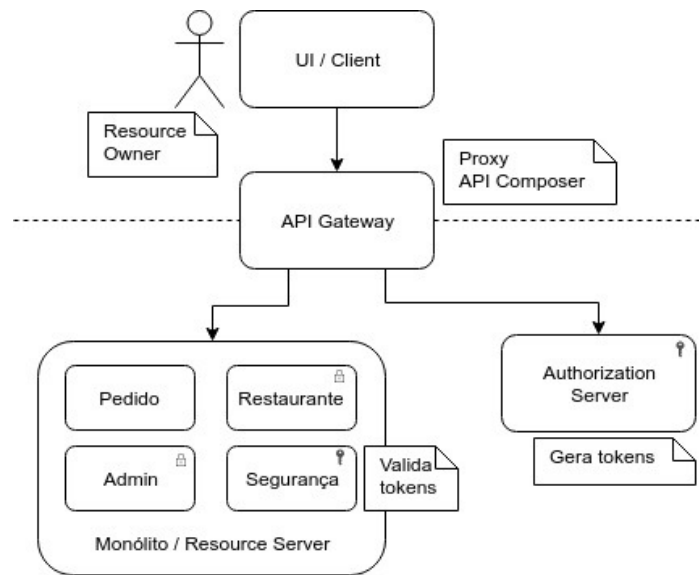


Figura 15.4: Roles OAuth no Caelum Eats

15.18 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"}
```

15.19 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$1YJxJHAbtsSCeyqgN7S1gurPZ8NSmTVA33dgPq6NqElU6qjz1pk0a
```

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.eyJleHAiOjE1NjcwNTkwODUsInVzZXJfbmFtZSI6ImFkbW1uIiwiaXV0aG9yaXRpZXMiOiJ1Uk9MRV9BRE1JTjJdLCJqdGkiOiI2ODlkMGE0ZS0xZjRmLTQ5OGMtOGMzMS05YjVlYjMyZWYxYjgi

```

```
LCJjbGllbnRfawQiOiJlYXRzIiwic2NvcGUiOiYw55Il19.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:

```
eyJleHAiOiE1NjcwNTkwODUsInVzZXJfYmFtZSI6ImFkbWluIiwiaXV0aG9yaXRpZXMiOiUk9MRV9BRE1JTjJdLCJqdGkiOiI2ODlkMGEOZS0xZjRmLTQ5OGMtOGMtMS05YjVlYjMyZWYxYjgiLCJjbGllbnRfawQiOiJlYXRzIiwic2NvcGUiOiYw55Il19
```

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.

15.20 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, acesse *File > Import > Existing Maven Projects* e clique em *Next*. Em *Root Directory*, aponte para o diretório clonado anteriormente.

Veja o código das classes `AuthorizationServerApplication` e `OAuthServerConfig`, além dos

arquivos `bootstrap.properties` e `application.properties`.

Note que o `spring.application.name` é `authorizationserver` . A porta definida para o `Authorization Server` é `8085` .

3. Crie o arquivo `authorizationserver.properties` no `config-repo`, com o seguinte conteúdo:

```
#DATASOURCE CONFIGS
spring.datasource.url=jdbc:mysql://localhost/eats?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=

jwt.secret = rm'!@N=Ke!~p8VTA2ZRK~nMDQX5Uvm!m'D&]{@Vr?G;2?Xhbc:Qa#9#eMLN\}x3?JR3.2zr~v)gYF^8\>:X
fB:Ww75N/emt9Yj[bQMNCww\J?N,nvH.<2\.r~w]*e~vgak)X"v8H`MH/7"2E`,^k@n<vE~wD3g9JWPy;CrY*.Kd2_] )=><D
?YhBaSua5hw%{2}_FVxzB9`8FH^b[X3jzVER&:jw2<=c38=>L/zBq`}C6tT*cCSVc^c]-Lj)&/

security.oauth2.client.client-id=eats
security.oauth2.client.client-secret=$2a$10$1YJxJHAbtsSCeyqgN7S1gurPZ8NSmTVA33dgPq6NqElU6qjz1pk0a
```

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.
    eyJleHAiOiE1NjcwNTkwODUsInVzZXJfcmFtZSI6ImFkbWluIiwiaXV0aG9yaXRpZXMiOiJsUk9MRV9BRE1JTjJdLCJqdGkiOiI2ODlkMGE0ZS0xZjRmLTQ5OGMtOGMzMSc0YyYjVlYjMyZWYxYjgiLCJjbGllbnRfawQiOiJlYXRzIiwic2NvcGUI0IisiYW55Ii11
    9.
    ZtYpX3GJJPYU8UNhHRtmEtQ7SLiizdZ0rdCRJt64ovF4",
  "token_type": "bearer",
  "expires_in": 43199,
  "scope": "any",
  "jti": "689d0a4e-1f4f-498c-8c31-9b5eb32ef1b8"
```

```
}
```

Pegue o conteúdo da propriedade `access_token` e analise o cabeçalho e o payload em: <https://jwt.io>

O payload deverá conter algo semelhante a:

```
{
  "exp": 1567059085,
  "user_name": "admin",
  "authorities": [
    "ROLE_ADMIN"
  ],
  "jti": "689d0a4e-1f4f-498c-8c31-9b5eb32ef1b8",
  "client_id": "eats",
  "scope": [
    "any"
  ]
}
```

5. Remova o código de autenticação do API Gateway.

Para isso, delete as seguintes classes do API Gateway:

- ~~AuthenticationController~~
- ~~AuthenticationDto~~
- ~~JwtTokenManager~~
- ~~PasswordEncoderConfig~~
- ~~Role~~
- ~~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.

15.21 RESOURCE SERVER COM SPRING SECURITY OAUTH 2

Para definir um Resource Server com o Spring Security OAuth 2, que consiga validar e decodificar os tokens (opacos ou JWT) emitidos pelo Authorization Server, basta anotar a aplicação ou uma configuração com `@EnableResourceServer`.

Podemos definir, na configuração `security.oauth2.resource.token-info-uri`, a URI de validação de tokens opacos.

No caso de token self-contained JWT, devemos definir a propriedade `security.oauth2.resource.jwt.key-value`. Pode ser a chave simétrica, no caso de algoritmos como o HS256, ou a chave pública, como no RS256. A chave pública em um algoritmo assimétrico pode ser baixada do servidor quando definida a propriedade `security.oauth2.resource.jwt.key-uri`.

Por padrão, todos os endereços requerem autenticação. Porém, é possível customizar esse e outros detalhes fornecendo uma implementação da interface `ResourceServerConfigurer`. É possível herdar da classe `ResourceServerConfigurerAdapter` para facilitar as configurações.

15.22 EXERCÍCIO: PROTEGENDO O SERVIÇO ADMINISTRATIVO

1. 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>
```

2. Anote a classe `EatsAdministrativoServiceApplication` com `@EnableResourceServer`:

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

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



```

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

```

3. Adicione ao `admin.properties` do `config-repo`, a mesma chave usada no Authorization Server na propriedade `security.oauth2.resource.jwt.key-value`:

`config-repo/admin.properties`

```
security.oauth2.resource.jwt.key-value = um-segredo-bem-secreto
```

4. 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/admin/OAuthResourceServerConfig.java`

```

@Configuration
public 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();
    }
}

```

5. Abra um terminal e tente listar 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"}
```

15.23 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 agrega 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/
```

15.24 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 tem 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
```

15.25 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
```

15.26 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 entrarem 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.

15.27 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 de 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 S
ELECT 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/>

15.28 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 passá-íamos 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

16.1 DESAFIO: EXTRAIR SERVIÇOS DE PEDIDOS E DE ADMINISTRAÇÃO DE RESTAURANTES

Objetivo

Extraia os módulos `eats-pedido` e `eats-restaurant` 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.