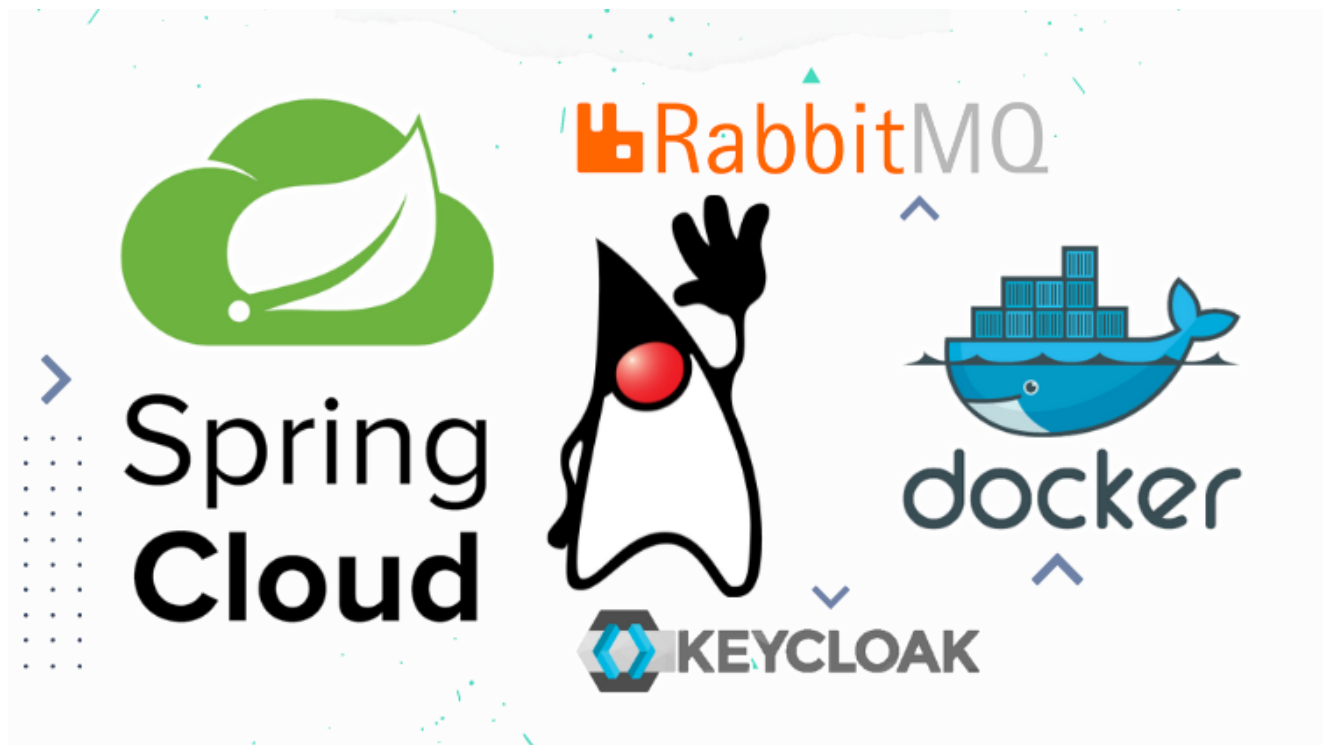


Domine Microservices e Mensageria com Spring Cloud e Docker



Por Dougllas Sousa (<https://cursodsousa.github.io>)

Código fonte no Github:

<https://github.com/cursodsousa/curso-microservices-springcloud>

Configuração e Implementação do Discovery Server (Eureka)

Para implementar o Discovery Server, vamos criar um projeto Spring Boot normal, com a estrutura básica e com a seguinte dependência:

- *Eureka Server*

Sim, apenas esta dependência, além das dependências básicas do Spring Boot, é necessária para criar o discovery server nesse momento. Não é necessário adicionar o Starter Web, pois o Eureka já faz o auto serving.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Ao selecionar o Eureka Server como dependência ou mesmo qualquer outra biblioteca de Spring Cloud, automaticamente o Scaffolding do Spring Boot virá com o Dependency Management do Spring Cloud, a fim de deixar todas as dependências Cloud na mesma versão. Se tiver criado o projeto na mão, adicione-o no seu pom, para visualizar/configurar isto, basta ir no seu pom.xml:

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

Sua versão estará na tag "properties", conforme mostrado a seguir:

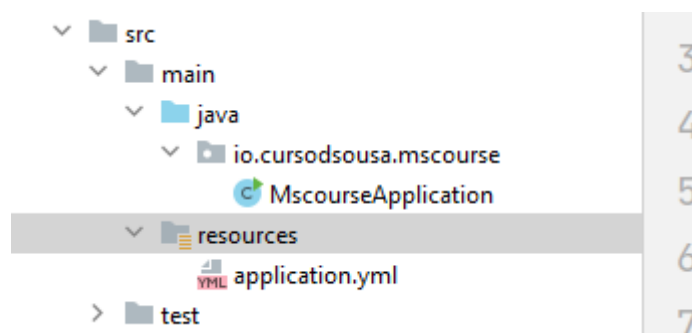
```
<properties>
  <java.version>11</java.version>
  <spring-cloud.version>2021.0.0</spring-cloud.version>
</properties>
```

Agora vamos configurar nosso Eureka Server.

Para configurar, iremos primeiramente anotar a classe de Application com a annotation `@EnableEurekaServer`, essa annotation faz parte do pacote `org.springframework.cloud.netflix.eureka.server`. Feito isso, sua classe Application ficará mais ou menos assim:

```
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.boot.autoconfigure.SpringBootApplication;
5  import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
6
7  @SpringBootApplication
8  @EnableEurekaServer
9  public class MscourseApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(MscourseApplication.class, args);
13     }
14
15 }
16
```

Por enquanto, apenas esta classe será necessária para subir o eureka server. Agora iremos configurar o `application.yml`, o arquivo fica em `src/main/resources`:



caso seu arquivo ainda esteja com a extensão *.properties* (application.properties), renomeie para a extensão *.yml*, para isto você pode clicar com o botão direito do mouse em cima do arquivo, selecionar a opção "Refactor" > "Rename.." - substitua para a nova extensão.

Configurando o application.yml do Discovery Server Eureka

A configuração é relativamente simples. Primeiro iremos configurar o nome da aplicação - esta configuração será de extrema importância ao trabalharmos com microservices e spring cloud, pois é através deste nome que os microservices serão registrados no discovery server.

A propriedade é: *spring.application.name*

No arquivo yml, ficará da seguinte forma (respeitar o espaçamento que regula a hierarquia das configurações):

```
1  spring:
2    application:
3      name: ms-eureka-server
4
```

O nome fica a seu critério, estou usando "ms-eureka-server" como exemplo.

Depois de configurado o nome, vamos configurar a porta onde será servido nosso Eureka Server, escolha uma porta livre, esta porta será fixa. Por convenção a porta do Eureka Server será a "8761", mas nada impede que você utilize alguma outra porta, a configuração no yml é "server.port" e fica desta forma:

```
5  server:
6    port: 8761
```

Por último, iremos adicionar a configuração de *client* do Eureka, para que ele não seja registrado nele mesmo como um serviço, caso não insira esta configuração, ele vai se auto registrar como microservice nele mesmo e, como ele é um server, não faria sentido e até poderia causar algum problema, segue a configuração:

```
8  eureka:
9    client:
10      register-with-eureka: false
11      fetch-registry: false
12
```

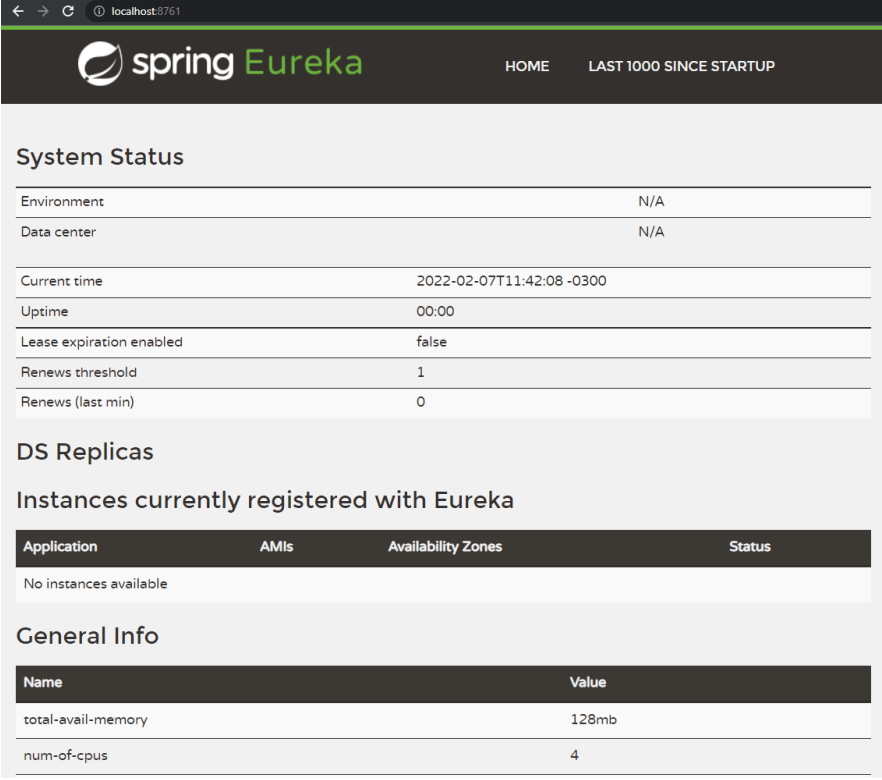
Dessa forma vamos ver o arquivo *application.yml* completo:

```
1  spring:
2    application:
3      name: ms-eureka-server
4
5  server:
6    port: 8761
7
8  eureka:
9    client:
10     register-with-eureka: false
11     fetch-registry: false
12
```

Agora basta rodar a classe main (Application) e abrir o browser na url:

http://localhost:8761

Você deverá ver a tela a seguir:



The screenshot shows the Spring Eureka web interface. The header includes the Spring Eureka logo and navigation links for HOME and LAST 1000 SINCE STARTUP. The main content area is divided into three sections: System Status, DS Replicas, and General Info.

System Status

Property	Value
Environment	N/A
Data center	N/A
Current time	2022-02-07T11:42:08 -0300
Uptime	00:00
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	128mb
num-of-cpus	4

Concluída então a configuração inicial do Eureka Discovery Server. Irei falar sobre isto novamente mais a frente, mas o primeiro componente a subir dentro desta arquitetura deverá ser sempre o Discovery Server, não esquecer este detalhe. Vamos para o primeiro microservice.

Implementação do Microservice de Clientes

Nesta implementação, iremos construir mais uma aplicação Spring Boot, desta vez iremos adicionar dependências comuns e necessárias para rodar o nosso microservice. Crie um projeto Spring Boot comum e adicione as seguintes dependências:

- *Spring Boot Starter Web*
- *Spring Boot Starter Data JPA*
- *H2 Database*
- *Lombok*
- *Spring Boot DevTools*
- *Eureka Client*

Seguem as dependências no arquivo pom.xml:

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

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

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

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Continuação:

```

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

<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>

```

Não esquecer de adicionar/conferir o Dependency Management do Spring Cloud, para que o Eureka Server tenha referência de versão:

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

```

As properties do seu pom.xml devem ficar assim:

```

<properties>
  <java.version>11</java.version>
  <spring-cloud.version>2021.0.0</spring-cloud.version>
</properties>

```

Agora, na classe de Application, adicione a annotation `@EnableEurekaClient`, esta annotation pertence ao pacote `org.springframework.cloud.netflix.eureka`. Sua classe Application ficará mais ou menos assim:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

5 usages  @ Dougllas Sousa *
@SpringBootApplication
@EnableEurekaClient
public class MsclientesApplication {
    @ Dougllas Sousa
    public static void main(String[] args) {
        SpringApplication.run(MsclientesApplication.class, args);
    }
}
```

Agora iremos configurar o arquivo `application.properties`, que deverá renomeá-lo para `application.yml`. A primeira configuração obrigatória, é o nome da aplicação, com a propriedade `spring.application.name`, este será o nome do seu microservice e será desta forma que ele será referenciado no discovery server e também por outros microservices. No exemplo abaixo estou dando o nome de "msclientes":

```
spring:
  application:
    name: msclientes
```

Iremos fazer a configuração do Eureka Client, observe as properties e iremos comentar a seguir:

```
server:
  port: 0

eureka:
  instance:
    instance-id: ${spring.application.name}:${spring-cloud.application.instance_id:${random.value}}
  client:
    register-with-eureka: true
    service-url:
      defaultZone: http://localhost:8761/eureka
```


Vamos comentar as properties:

"server.port":

Nesta configuração, temos a opção de subir o microservice com uma porta comum (ex: 8080), mas estamos colocando o valor 0 (zero), isto indica para o Spring Boot que gostaríamos de subir este microservice em uma porta randômica.

Qual o objetivo de subir em uma porta randômica? É bem simples: queremos poder subir várias instâncias do mesmo microservice e, se colocarmos uma porta fixa, ao subir a segunda instância seu server irá reclamar que aquela porta está ocupada, então desta forma, com porta fixa, toda vez que for subir você deverá alterar a porta dentro do seu arquivo application.yml antes. Para evitar este tipo de configuração, estamos dizendo ao Spring que suba nosso microservice em uma porta randômica. Desta forma toda instância do nosso microservice irá subir em uma porta diferente e não ocupada na máquina host.

"eureka.instance.instance-id":

Aqui estamos nomeando a instância do microservice, no valor colocamos a seguinte expressão:

```
${spring.application.name}:${spring-cloud.application.instance_id:${random.value}}
```

O que estamos fazendo aqui é criar um nome único para cada instância do microservice, este nome de instância irá aparecer dentro do eureka server e deve obrigatoriamente ser único. Assim acabamos por randomizar o processo de criação do nome da instância, que irá ser composto por "Nome do microservice : nome da instancia : valor randômico ", perceba que estamos utilizando *expression language* com `${ value }` para referenciar outras properties previamente setadas pelo Spring Boot.

Agora vamos criar um Endpoint para fazer o teste mais a frente com o Gateway, crie uma classe chamada ClientesResource, em um pacote chamado "application" (a partir do pacote principal da sua aplicação), a classe ficará mais ou menos assim:

```
package io.github.cursodsousa.msclientes.application;

import org.springframework.web.bind.annotation.*;

@ Dougllas Sousa *
@RestController
@RequestMapping("clientes")
public class ClientesResource {

    @ Dougllas Sousa *
    @GetMapping
    public String status() {
        return "ok";
    }
}
```

Com isso criamos uma requisição para retornar o status da aplicação ("ok").

Configuração e Implementação do Gateway

Para implementar o Gateway, vamos criar um projeto Spring Boot normal, com a estrutura básica e com as seguintes dependências (não utilizar o starter-web normal, ele só funciona com o WEBFLUX):

- starter-webflux
- eureka-client
- starter-gateway

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

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

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Ao selecionar o Eureka Cliente e o Gateway como dependências ou mesmo qualquer outra biblioteca de Spring Cloud, automaticamente o Scaffolding do Spring Boot virá com o Dependency Management do Spring Cloud, a fim de deixar todas as dependências Cloud na mesma versão. Se tiver criado o projeto na mão, adicione-o no seu pom, para visualizar/configurar isto, basta ir no seu pom.xml:

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

As properties do seu pom.xml devem ficar assim:

```
<properties>
  <java.version>11</java.version>
  <spring-cloud.version>2021.0.0</spring-cloud.version>
</properties>
```

Na classe Application, devemos ter as seguintes annotations:

@EnableEurekaClient
@EnableDiscoveryClient

além da @SpringBootApplication

```
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  
import org.springframework.cloud.gateway.route.RouteLocator;  
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;  
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;  
import org.springframework.context.annotation.Bean;
```

1 usage 👤 Douglas Sousa

```
@SpringBootApplication  
@EnableEurekaClient  
@EnableDiscoveryClient  
public class MsccloudgatewayApplication {  
  
    👤 Douglas Sousa  
    public static void main(String[] args) {  
        SpringApplication.run(MsccloudgatewayApplication.class, args);  
    }  
}
```

Agora vamos configurar o application.yml, ele deve ficar da seguinte forma (renomeie seu application.properties para application.yml):

```
1  spring:  
2    application:  
3      name: msccloudgateway  
4    cloud:  
5      gateway:  
6        discovery:  
7          locator:  
8            enabled: true  
9            lower-case-service-id: true  
10   server:  
11     port: 8080  
12  
13   eureka:  
14     client:  
15       fetch-registry: true  
16       register-with-eureka: true  
17       service-url:  
18         defaultZone: http://localhost:8761/eureka
```

em spring.application.name colocamos o nome do nosso Gateway, logo abaixo podemos visualizar as configurações específicas do Gateway, onde habilitamos o discovery locator,

para que ele consiga identificar os microservices que estão registrados no eureka, e a segunda configuração "*lower-case-service-id*" para true indica que os nomes dos microservices serão configurados em lowercase dentro do gateway.

Mais abaixo temos o registro do Gateway no Eureka, para que ele identifique os microservices registrados e também para fazer o balanceamento de carga. Perceba que o Gateway também possui uma porta fixa, pois é através dele que será a porta de entrada dos nossos microservices.

Para finalizar nosso Gateway vamos configurar o Roteador, dentro da classe Application mesmo, iremos criar um @Bean e configurá-lo da seguinte maneira:

```
@Bean
public RouteLocator routes(RouteLocatorBuilder builder){
    return builder
        .routes()
        .route( r -> r.path( ...patterns: "/clientes/**").uri("lb://msclientes") )
        .build();
}
```

O RouterLocator é a configuração de roteamento do Gateway, ou seja, onde vemos acima "*r.path("/clientes/**")*" estamos definindo que, quando chegar uma requisição no Gateway com esta URL, ele irá redireccionar para a uri definida no segundo parametro: "*.uri("lb://msclientes")*". Agora entendendo o parametro "*uri*": conforme mostrado acima, estamos utilizando um protocolo "*lb*", que significa "*Load Balancer*" - Balanceamento de Carga, e dentro temos o nome do microservico de clientes ("*msclientes*"), que foi o *spring.application.name* do microservice de clientes, isso indica que, quando as requisições chegarem na url "*/clientes/***" elas serão roteadas para o microservice de clientes fazendo o balanceamento de carga, ou seja, se eu tiver mais de uma réplica do microservice de clientes, ele vai distribuir as requisições entre elas, não concentrando apenas em uma instancia. A classe main completa ficará mais ou menos assim (imports não estão no print):

```
@SpringBootApplication
@EnableEurekaClient
@EnableDiscoveryClient
public class MsccloudgatewayApplication {

    @ Dougllas Sousa
    public static void main(String[] args) {
        SpringApplication.run(MsccloudgatewayApplication.class, args);
    }

    @ Dougllas Sousa *
    @Bean
    public RouteLocator routes(RouteLocatorBuilder builder){
        return builder
            .routes()
            .route( r -> r.path( ...patterns: "/clientes/**").uri("lb://msclientes") )
            .build();
    }
}
```

Agora para subir a arquitetura, devemos primeiro subir o Eureka Server, logo após, o microservice de Clientes, e, por ultimo, o Gateway. As requisições irão ser feitas para o Gateway (<http://localhost:8080>), para testar uma requisição para o status do cliente, basta fazer um GET para a URL:

<http://localhost:8080/clientes>

Construindo Imagens Docker para aplicações Spring Boot

Antes de construirmos nossa imagem é necessário entender o processo de BUILD e RUN de nossa aplicação Spring Boot.

Para rodar uma aplicação Spring Boot, primeiro é necessário empacotar a app, utilizando o seguinte comando do Maven:

mvn clean package

Ao executar este comando será gerado uma pasta "target" na raiz da aplicação e dentro desta pasta um arquivo ".jar", este é o fat jar que pode ser utilizado para dar start na aplicação Spring Boot.

Para dar start neste JAR, basta executar o comando a seguir a partir da pasta target (digamos que o nome do seu jar seja "application.jar"):

java -jar application.jar

Agora sim, depois de entender o processo, podemos partir para construção da Imagem Docker. Para construir uma imagem Docker a partir de uma aplicação Spring Boot, é necessário criar um arquivo com o nome "Dockerfile" (sem extensão) na raiz da sua app Spring com o seguinte conteúdo:

```
1  >> FROM maven:3.8.5-openjdk-17 as build
2  WORKDIR /app
3  COPY . .
4  RUN mvn clean package -DskipTests
5
6  FROM openjdk:17
7  WORKDIR /app
8  COPY --from=build ./app/target/*.jar ./app.jar
9
10 ENTRYPOINT java -jar app.jar
```

Vamos agora entender um pouco sobre cada comando deste Dockerfile, linha a linha:

Linha 1: Todo Dockerfile começa com o comando FROM, onde iremos indicar qual a imagem base da nossa imagem, no caso acima, estamos utilizando o Maven com Open JDK 17, no final utilizamos o alias "build" para indicar que a nossa imagem será construída em 2 fases: o build e o run.

Linha 2: O comando WORKDIR é utilizado para criar uma pasta dentro do container onde serão executados os próximos comandos

Linha 3: Nesta linha estamos utilizando o comando COPY, que serve para copiar um arquivo da pasta atual (a mesma pasta onde está o Dockerfile) para a pasta indicada no WORKDIR (no exemplo acima é para a pasta "app").

Linha 4: Podemos executar linhas de comando dentro de uma imagem docker a partir do comando RUN, no exemplo acima, estamos rodando o build da aplicação Spring Boot, como o comando "mvn clean package -DskipTests", a flag "-DskipTests" indica que o passo de executar os testes do maven não será executado.

Linha 6: Agora iremos iniciar o run de nossa app buildada na fase "build". Vamos utilizar uma imagem do OpenJDK 17 para rodar nossa app Spring Boot.

Linha 7: Indicando que iremos usar a pasta "app" com o comando WORKDIR.

Linha 8: Agora o comando COPY novamente, só que com uma flag diferente, desta vez iremos copiar o arquivo JAR que foi buildado na fase "build" para a pasta "app" desta camada. A flag "--from" recebe um parametro que é de onde irei copiar o arquivo, estamos passando o alias "build", referenciando a fase acima, dela estamos copiando qualquer arquivo JAR (por isso o *), isso é interessante usar pois o seu jar pode mudar de nome a cada build dependendo da versão do seu projeto, se toda vez você modificar a versão do seu projeto (ex: clientes-app-1.0-SNAPSHOT.jar), o arquivo jar virá com nomenclatura diferente, então no COPY estamos pegando qualquer arquivo com a extensão .jar e renomeando para "app.jar".

Linha 10: O comando ENTRYPOINT serve para dar start no container, ele deve conter o comando que inicializa sua app buildada, um detalhe importante é que este comando, diferente do comando RUN, que pode ser adicionado várias vezes dentro de um Dockerfile, só pode ser adicionado uma vez e deve ser no final do Dockerfile. Estamos colocando então o comando para dar start na aplicação Spring Boot.

Para buildar esta imagem, utilize o seguinte comando a partir da mesma pasta onde está seu Docker file:

docker build -tag aplicacao:1.0 .

O Comando "docker build" é utilizado para construir uma imagem docker a partir de um arquivo Dockerfile, os outros parametros são:

-tag: neste parâmetro, indicamos qual será o nome da imagem e sua tag, no caso acima, o nome da imagem é "aplicacao" e sua tag é "1.0", a tag é a versão da imagem, logo, colocamos ela junto com o nome da imagem, separando-os por dois pontos (":").

O segundo parametro é o “.” (ponto) no final do comando, este parametro é a localização do Dockerfile, e quando passamos o “.” (ponto), indicamos que o Dockerfile está na pasta atual. Caso não passe este parametro, o build não irá acontecer.

Depois de executar o comando de build, você conseguirá visualizar sua imagem através do comando:

docker images

Este comando listará todas as imagens docker em sua instalação, é mostrado mais ou menos assim:

```
PS G:\dev\projetos\curso micro services\projeto-cursoms> docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
cursoms-gateway	latest	03f63e8f9f98	3 days ago	705MB
cursoms-cartoes	latest	a3e93335df5b	10 days ago	729MB
cursoms-avaliadorcredito	latest	5730ce656171	10 days ago	712MB
cursoms-clientes	latest	01a9457121fb	10 days ago	727MB
<none>	<none>	aaafe0d29a62	11 days ago	705MB
cursoms-eureka	latest	61acf938556a	4 weeks ago	708MB
rabbitmq	3.9-management	5903ebf60c32	7 weeks ago	257MB
quay.io/keycloak/keycloak	18.0.0	a669b057e631	8 weeks ago	562MB

“REPOSITORY” é o nome da imagem e “TAG” é a versão (tag) que você indicou no comando de build, caso não tenha indicado uma versão (tag), a sua imagem ficará com a tag “latest”.

Rodando Containers a partir da sua imagem Docker

Agora vamos iniciar um container a partir desta imagem, estamos assumindo para este exemplo que, sua imagem se chama “aplicacao” e possui tag “1.0”, vamos colocar o comando para iniciar containers a partir desta imagem:

docker run - -name minha-aplicacao -p 8081:8080 aplicacao:1.0

Este é o comando básico para dar start na imagem buildada no passo anterior. Vamos entender como funciona este comando:

docker run é o comando utilizado para criar e startar containers a partir de imagens docker. Este deve ser o primeiro comando quando você quiser fazer isto.

os outros parâmetros:

Comando - - **name**: este é comando utilizado para nomear um container docker, se você não colocar este parâmetro o docker irá atribuir um nome aleatório para seu container, então é

recomendado que sempre nomeie seus containers com este comando, no caso acima o nome do container é “minha-aplicacao”, este nome poderá ser utilizado como referência para outro comando docker, como por exemplo “docker stop minha-aplicacao” que seria o comando para parar este container e “docker start minha-aplicacao” para iniciá-lo novamente sem precisar criar do zero.

Comando **-p**: este comando serve para indicar uma porta onde na máquina HOST (a sua máquina que está rodando os comandos) irá escutar uma porta do container, no exemplo acima estamos indicando que a porta 8081 da máquina HOST irá escutar a porta 8080 do container, ou seja, neste parâmetro, a primeira porta é a da máquina HOST e a segunda, a do container, se você não adicionar este parâmetro, não será possível acessar o container pela porta dele.

Por ultimo, em um comando run do docker, colocamos o nome + tag da imagem que iremos utilizar pra criar o container, no exemplo acima é “aplicacao:1.0”

Depois de executado este comando, seu container chamado “minha-aplicacao” irá iniciar de acordo com a forma que foi colocada no Dockerfile. Logo, sendo uma aplicação Spring Boot irá startar o app.jar e irá ficar escutando na porta 8081. Então, para visualizar sua aplicação, acesse:

<http://localhost:8081>

Para visualizar seu(s) container(s) rodando na sua instância do docker, utilize o comando:

docker ps

Outros comandos úteis do Docker (todos comandos começam com “docker”):

docker pull: baixa uma imagem docker da internet.

docker images: lista as imagens.

docker image rm <id ou nome da imagem>: deleta uma imagem.

docker run <nome da imagem>: inicializa um container a partir de uma imagem.

docker ps: mostra todos os containers docker rodando na sua instancia.

docker ps -a: mostra todos os containers, parados ou rodando na sua instância.

docker container rm <id ou nome do container>: deleta um container.

docker stop <id ou nome do container>: para um container (sem deletá-lo, você poderá reiniciá-lo novamente mais tarde).

docker start <id ou nome do container>: inicializa um container parado anteriormente.

docker logs <id ou nome do container>: visualizar os logs de um container.

Inicializando o RabbitMQ

Antes de iniciar, o que é o RabbitMQ? É um servidor de mensageria open source desenvolvido em Erlang, implementado para trabalhar com mensagens em um protocolo denominado Advanced Message Queuing Protocol (AMQP). Ele possibilita lidar com o tráfego de mensagens de forma rápida e confiável, além de ser compatível com diversas linguagens de programação, possuir interface de administração nativa e ser multiplataforma. Em suma é um serviço de Mensageria bastante utilizado e bem simples de aprender.

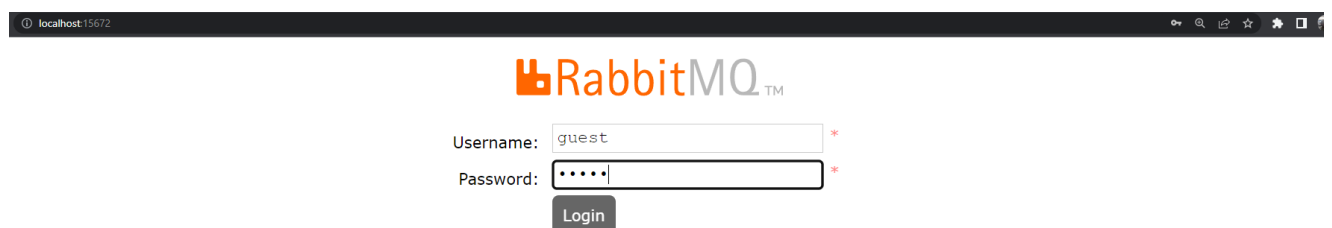
Para iniciá-lo a partir do docker utilize o comando:

```
docker run --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.9-management
```

Este comando irá inicializar um container a partir da versão 3.9 do RabbitMQ, na sua máquina HOST o painel irá escutar na porta 15672, logo, depois de iniciar o container, você pode acessar o painel através do endereço:

<http://localhost:15672>

Por padrão o painel do RabbitMQ pode ser acessado com o login e senha “guest”, ou seja, coloque “guest” tanto no login como na senha:



Perceba que o RabbitMQ possui 2 portas que podem ser expostas pra fora do container: a 15672 e a 5672, a primeira é a porta onde acessamos o painel administrativo mostrado acima e o segundo é a porta onde as aplicações se conectam ao serviço de mensageria do RabbitMQ, então é necessário expor as duas para conseguir utilizá-lo.

Inicializando o Keycloak

Patrocinado pela Red Hat, o Keycloak é um software open source de um servidor JBoss feito para trabalhar em conjunto com sua aplicação em implementações mais comuns de autenticação e autorização. Caso as configurações padrão não te atendam, existem várias configurações e customizações que podem ser feitas para adequar o funcionamento ao seu sistema.

Para subir uma instância do Keycloak a partir do docker, utilizamos o seguinte comando:

```
docker run -p 8080:8080 -e KEYCLOAK_ADMIN=admin -e  
KEYCLOAK_ADMIN_PASSWORD=admin quay.io/keycloak/keycloak:18.0.1 start-dev
```

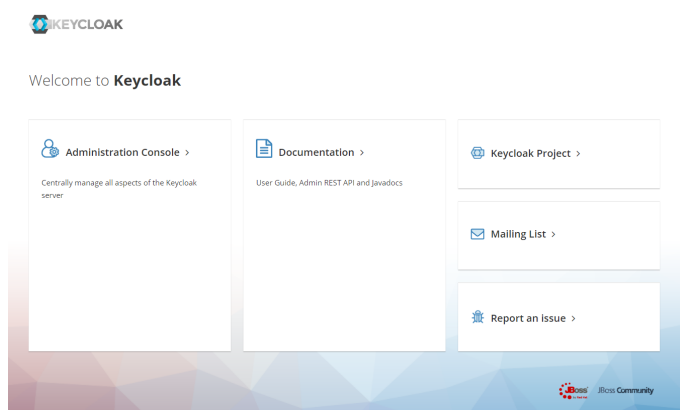
Aqui neste comando docker run, temos um parâmetro diferente, que é o “-e”, este parâmetro significa variável de ambiente (e de environment). Neste container do Keycloak estamos passando 2 variável de ambiente para dentro do container, a variável “KEYCLOAK_ADMIN” que serve para indicar qual será o login do usuário administrador e a segunda variável é o “KEYCLOAK_ADMIN_PASSWORD”, que seta qual será a senha do administrador, guarde elas para acessar o painel do Keycloak. Normalmente o nome da imagem fica no final do comando docker run, porém, a imagem deste comando é a seguinte:

quay.io/keycloak/keycloak:18.0.1

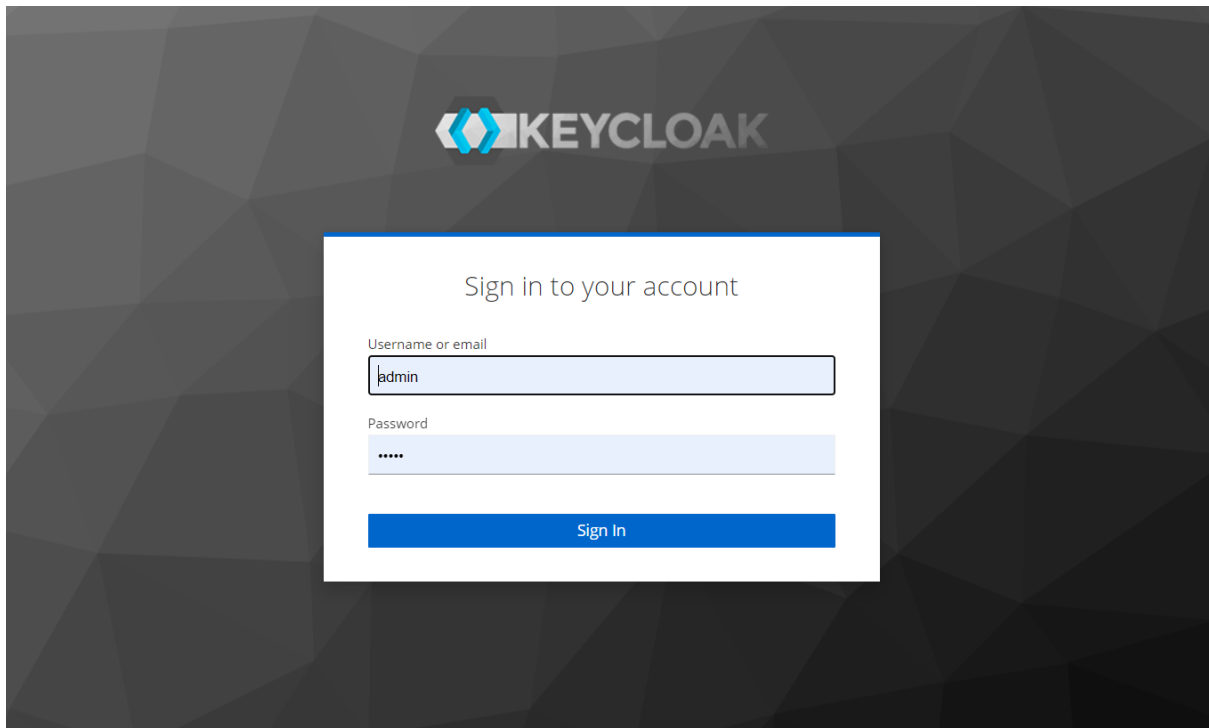
Logo após o nome da imagem temos o comando “start-dev”, este é um parâmetro de inicialização do container, ou seja, ele irá startar a partir da imagem e irá rodar em modo “dev” (desenvolvimento). Este é um parâmetro obrigatório.

Para acessar o painel do Keycloak acesse a porta 8080 da sua máquina HOST ou a porta que você indicou no parâmetro “-p”:

http://localhost:8080



Para logar clique em “Administration Console”:



Utilize os usuário e senha “admin”, conforme configurado nas variáveis de ambiente de inicialização do container.

Sites e referências:

Site oficial do Spring Cloud:

<https://spring.io/projects/spring-cloud>

Site oficial do Spring Boot:

<https://spring.io/projects/spring-boot>

Site oficial do RabbitMQ:

<https://www.rabbitmq.com/>

Site oficial do Keycloak:

<https://www.keycloak.org/>

Site oficial do Docker:

<https://www.docker.com/>