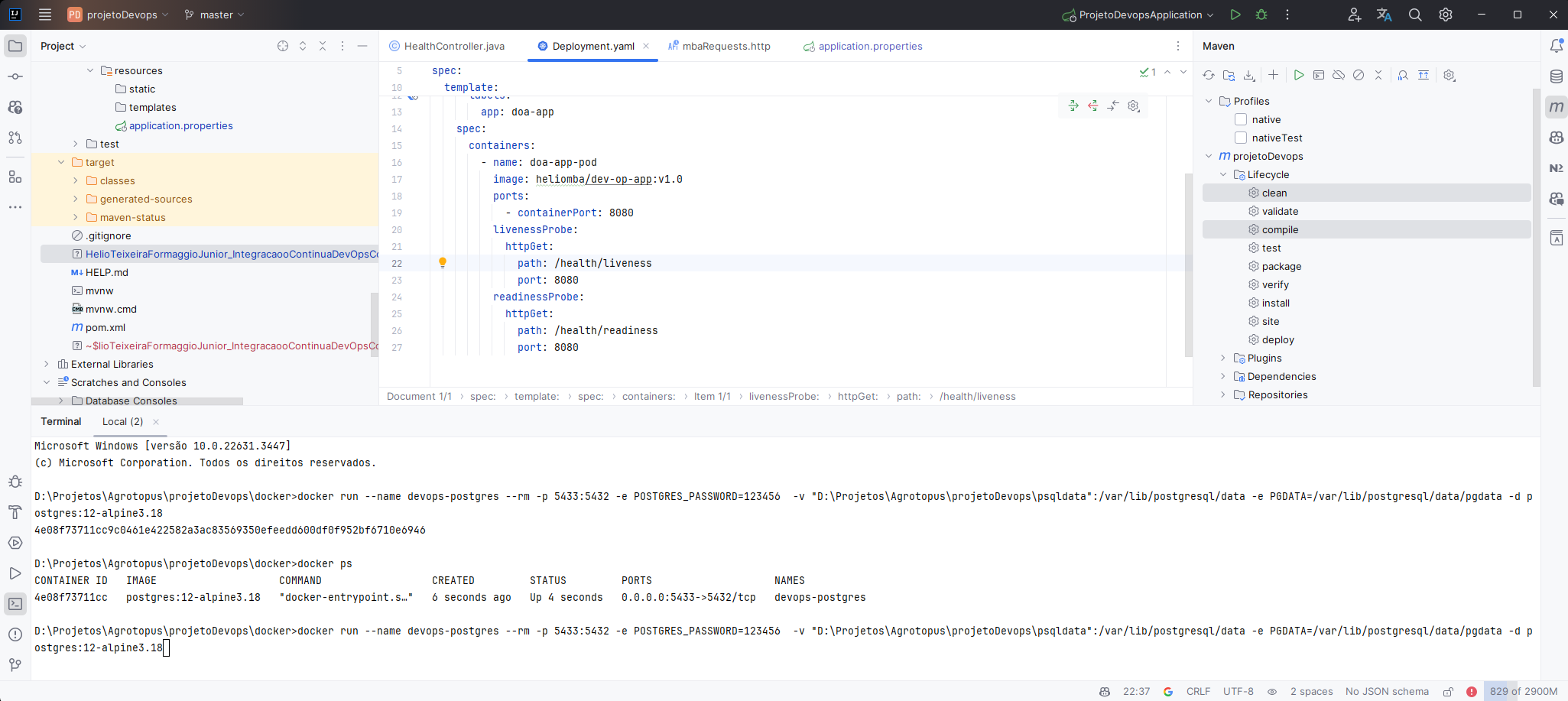
### Integração Contínua, DevOps e Computação em Nuvem [24E1\_3]

### Helio Teixeira Formaggio Junior

# Sobre o Projeto da Disciplina

Através deste documento irei descrever, passo a passo, o deploy de minha aplicação teste em um cluster **Kubernetes** segundo as premissas definidas na disciplina. Utilizo as ferramentas da **Jetbrains** como ponto focal de tudo que desenvolvo, neste caso o **Intellij**. As telas serão, em sua maioria, capturadas por esta ferramenta.

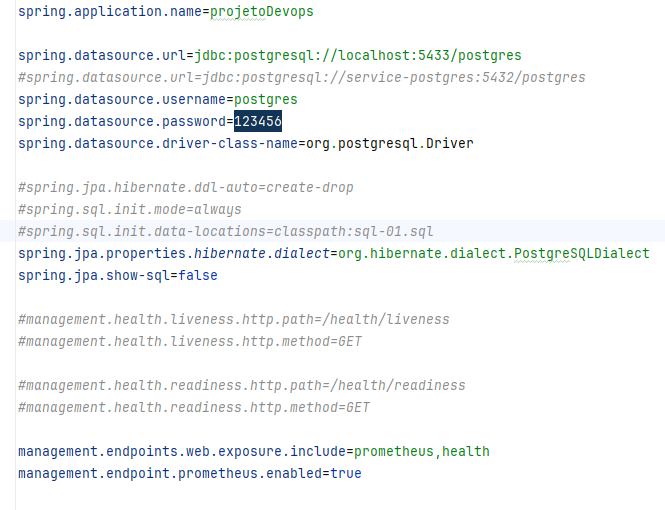
  
Figura 1: Imagem de Exemplo da ferramenta

# Sobre a aplicação utilizada

Estou usando uma aplicação muito básica que foi feita para a disciplina anterior na forma de um micro serviço. Ela expõe apenas as compras de produtos através de **orders**.

O arquivo application.properties ficou assim:

(deixarei códigos comentados para mostrar caminhos que já tentei)

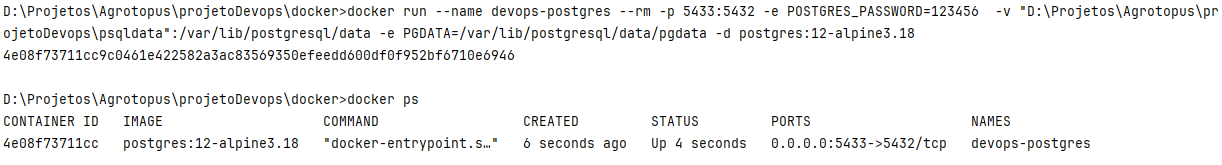
  
Figura 2: arquivo: application.properties

Como pode perceber, tentei usar a configuração de liveness e readness que já está embutida no SpringBoot, mas decidi fazer meus endpoints manualmente. Sobre o endereço do banco, está comentado o que vai no POD, deixei como **localhost** para testes locais.

Para isso usei uma imagem do **Postgresql** e rodei de forma local, colocando a persistência do banco em uma pasta em específico. E utilizei para teste este comando:

docker run --name devops-postgres --rm -p 5433:5432 -e POSTGRES\_PASSWORD=123456 -v "D:\Projetos\Agrotopus\projetoDevops\psqldata":/var/lib/postgresql/data -e PGDATA=/var/lib/postgresql/data/pgdata -d postgres:12-alpine3.18

Resultando em uma imagem como na figura 3.

  
Figura 3

Para o teste da aplicação utilizei três end-points:

GET http://localhost:8080/health/liveness

GET http://localhost:8080/health/readness

GET http://localhost:8080/api/ordes/

Enquanto as duas primeiras enviam como resposta o código HTTP 200, as **orders** listam todas as ordens do sistema no formato como na imagem parcial abaixo.

  
Figura 4

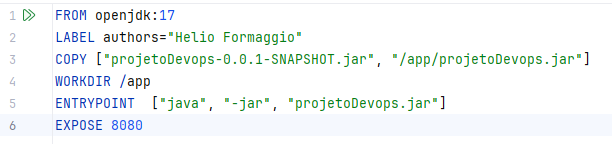
## Publicando a imagem

Para todos os arquivos relacionados ao Docker criei uma pasta *docker* na raiz do projeto. Onde o arquivo Dockerfile está localizado.

Esta pasta contem um arquivo *.bat* onde salvei alguns comandos que fui utilizando durante a preparação do projeto. Incluindo testes com o **Postgresql**, **Prometheus** e **Grafana**.

Para a construção da imagem eu copio o arquivo **.jar** para a pasta pelo motivo de ter tentado utilizar endereços completos absolutos e relativos sem sucesso.

O Arquivo ficou assim:

  
Figura 5

Então utilizei o comando a seguir para a construção:

docker build -t heliomba/dev-op-app:v1.0 -f Dockerfile .

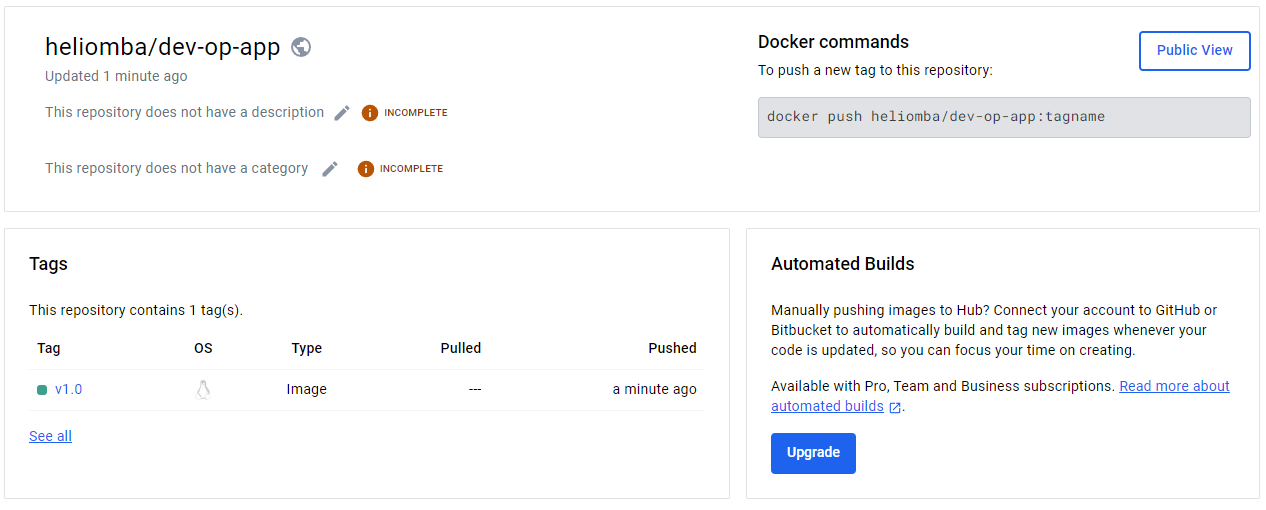
E o resultado foi:

  
Figura 6

O próximo passo foi enviar a imagem para meu repositório do DockerHub com o seguinte comando:

docker push heliomba/dev-op-app:v1.0

O resultado:

  
Figura 7  
  
Figura 8

O endereço do meu repositório é <https://hub.docker.com/repository/docker/heliomba/dev-op-app/general>.

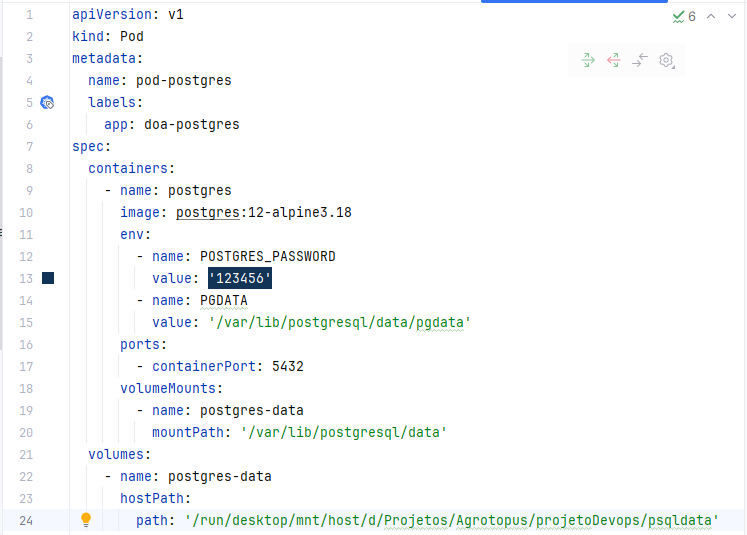
# Preparação do banco no cluster Kubernetes

Utilizarei o Docker Windows como cluster de Kubernetes.

Para o banco, precisei mapear o mesmo endereço de dados utilizado no teste. Este endereço está disponível na raiz do projeto para propósitos de reprodução, e necessita que o endereço absoluto seja ajustado.

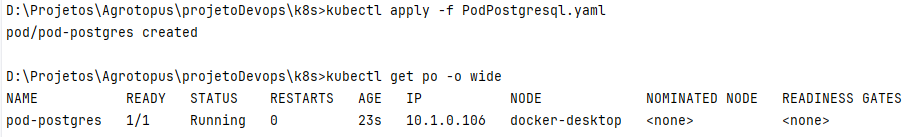
Todos os arquivos relacionados ao Kubernetes estão na pasta **k8s** na raiz do projeto.

Para o banco preparei o seguinte arquivo domo um POD diretamente, já que neste exemplo terá apenas uma instância.

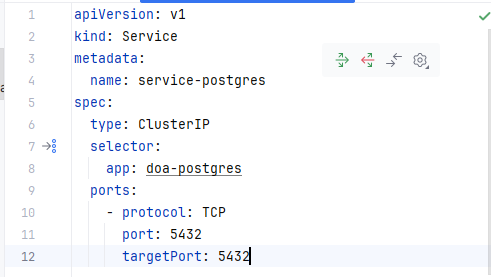
  
*Figura 9*

O endereço que está na propriedade **path:** é o endereço relativo do sistema do host olhando do ponto de vista interno do cluster.

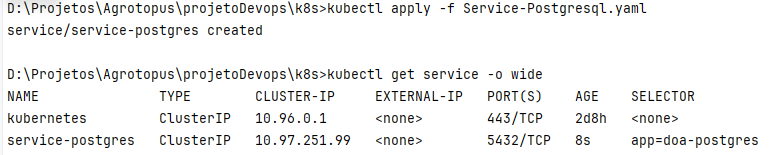
Aplicando a configuração do POD temos o seguinte resultado:

  
*Figura 10*

Para que a aplicação não se preocupe com os IPs do cluster, criamos um **service** do tipo **ClusterIP** como no arquivo:

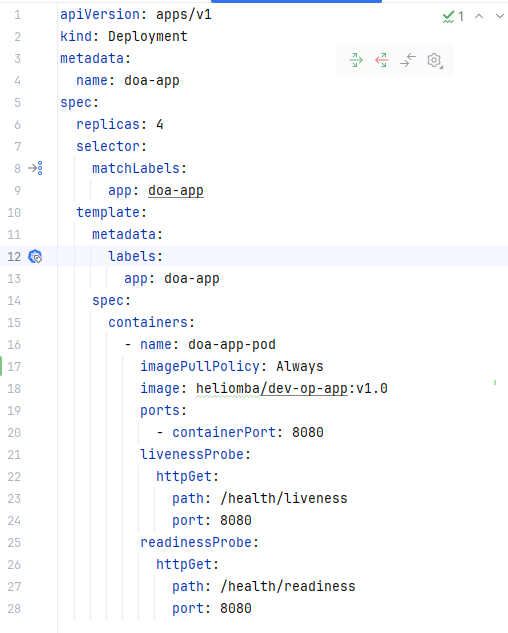
  
*Figura 11*

Aplicando a configuração do serviço temos o seguinte resultado.

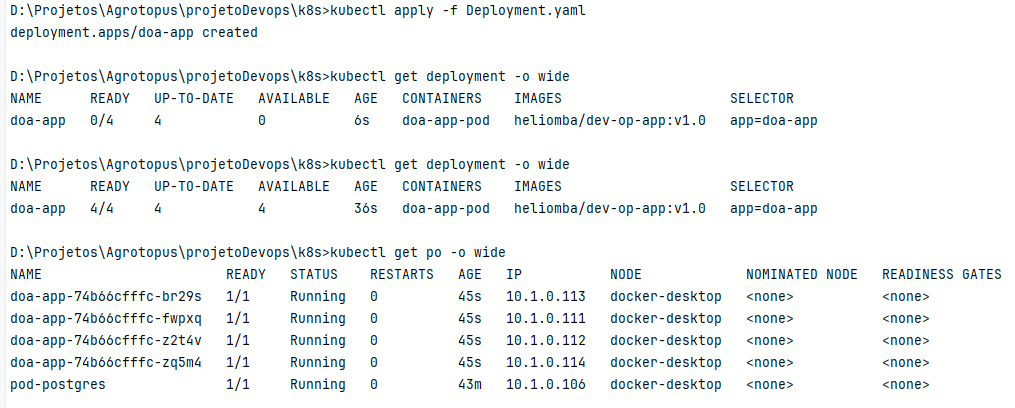
  
*Figura 12*

# Preparação da Aplicação

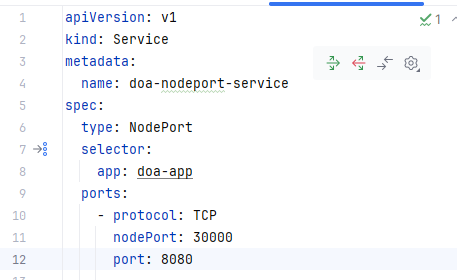
Para a aplicação, utilizamos uma configuração de Deployment onde eu o nomeio **doa-app** (DevOpsApp). O arquivo ficou assim:

  
*Figura 13*

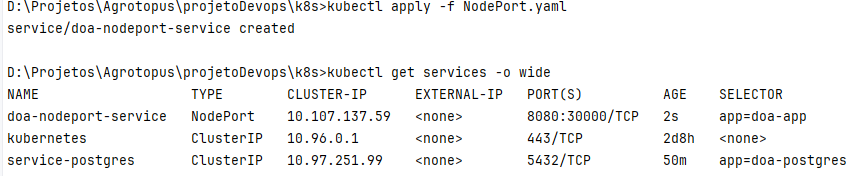
E aplicando o Deployment temos este resultado:

  
Figura 14

Agora precisamos expor a aplicação para fora do cluster. Para isto utilizaremos um NodePort e um Service. Vou configurar para a porta 30000 como na imagem abaixo.

  
*Figura 15 : NodePort*

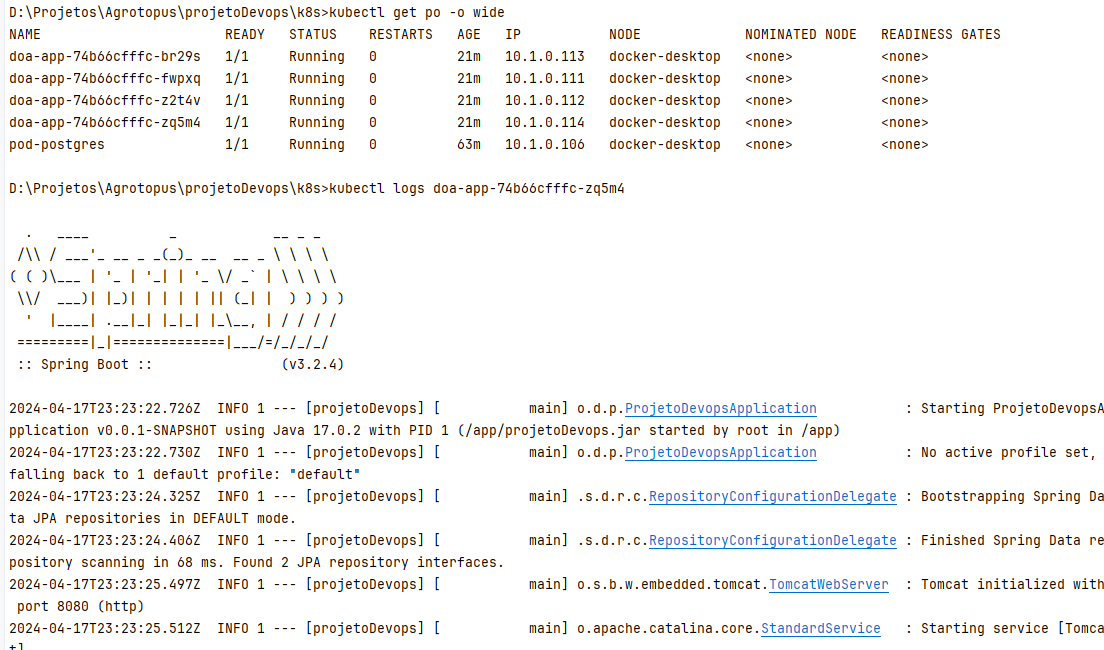
Aplicando as configurações temos o seguinte resultado:

  
*Figura 17*

Para confirmarmos se a aplicação está corretamente configurada e exposta, utilizamos o endpoint de orders citado anteriormente, mas na nova porta 30000. O resultado vemos abaixo nesta imagem parcial:

  
*Figura 18*

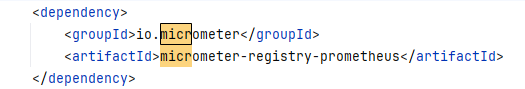
Apenas para uma segunda confirmação, posso verificar o log de inicialização de minha aplicação escolhendo um dos PODs e solicitando o log deste, como na imagem abaixo:

  
*Figura 19*

# Preparação do Prometheus

Iniciamos a preparação para expor as métricas da aplicação com o Prometheus já na compilação da aplicação.

Colocamos a extensão do Micrometer Prometheus no POM da aplicação como abaixo:

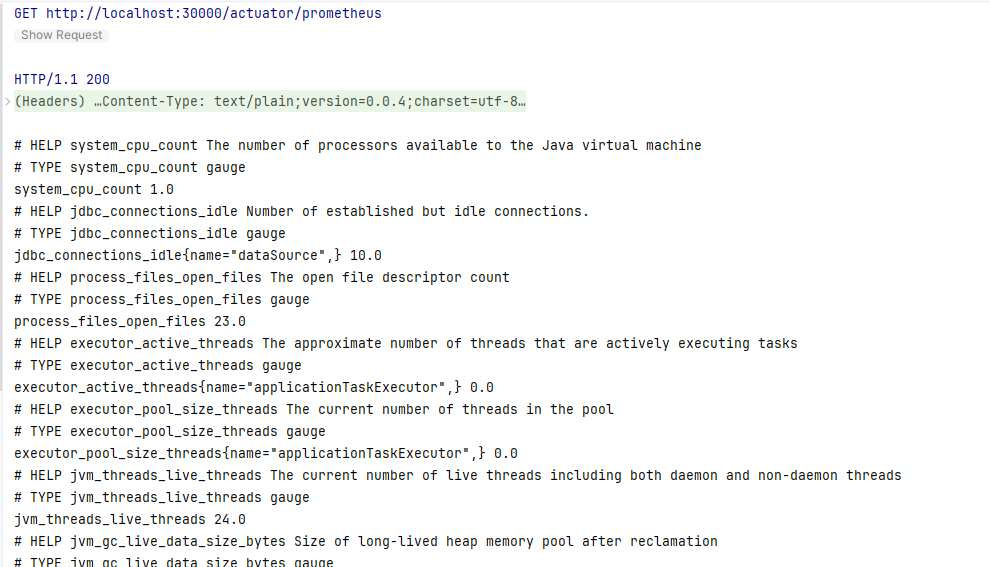
  
*Figura 20*

E configuramos a exposição no arquivo application.properties:

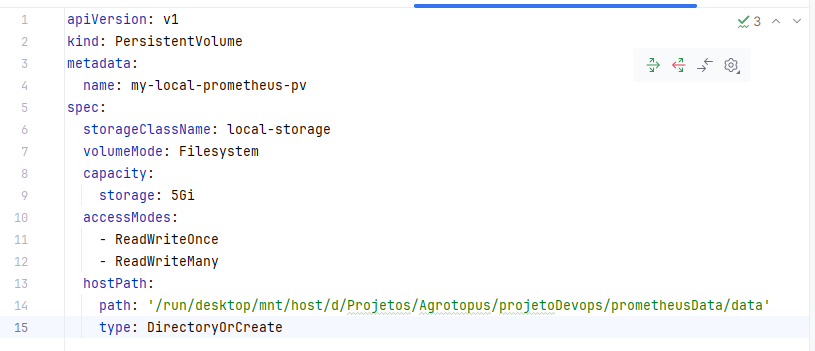
  
*Figura 21*

Existe a exposição do **health** pelas minhas tentativas de expor os **probes** padrão do SpringBoot. Terei de analisar o funcionamento do padrão do framework mais profundamente.

Assim conseguimos os seguintes resultados ao consumir as métricas:

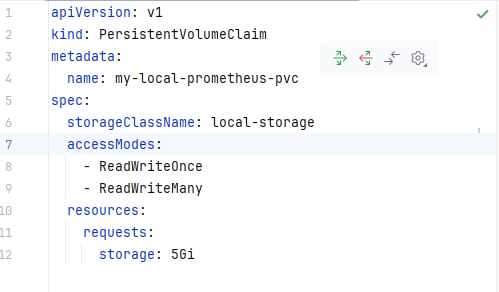
  
*Figura 22*

Agora prepararemos a aplicação do Prometheus para salvar as métricas. Precisamos que os dados não sejam voláteis, sendo assim criaremos um volume externo ao Cluster para salvar os dados. Utilizaremos o seguinte PersistentVolume:

  
*Figura 23*

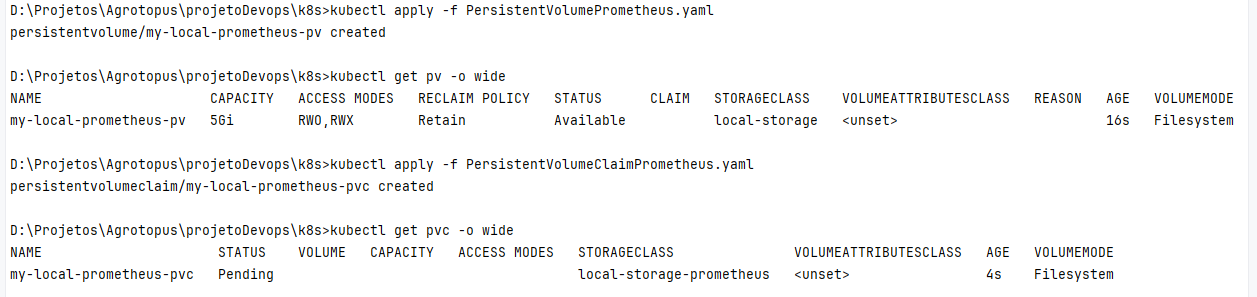
Para propósito de testes foi utilizado um “local storage”. Se precisar replicar, ajustar para a o endereço absoluto da pasta disponibilizada no projeto.

Agora precisamos de uma forma de solicitar este volume. Sendo assim, criei este PersistentVolumeClaim:

  
*Figura 24*

Sobre os “accessModes”, deixei os dois porque estava fazendo alguns testes da diferença entre eles. E verifiquei que se trata de quantos PODs podem acessar o volume. Se apenas um , ou vários. Deixei porque neste teste não haveria diferença com apenas um POD do Prometheus.

Aplicando estas configurações temos os seguintes resultados:

  
*Figura 25*

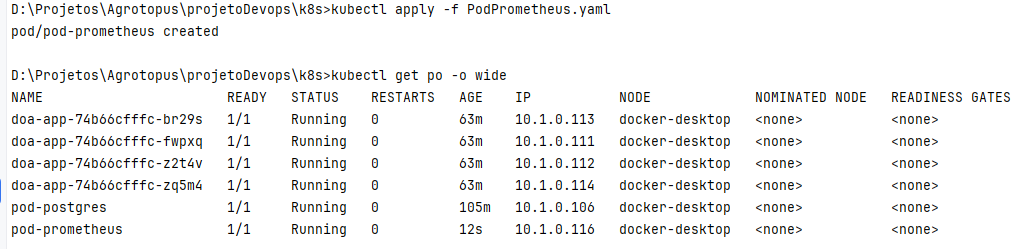
Agora precisamos preparar o POD do Prometheus para monitorar a aplicação, como demonstra este arquivo:

*Figura 26*

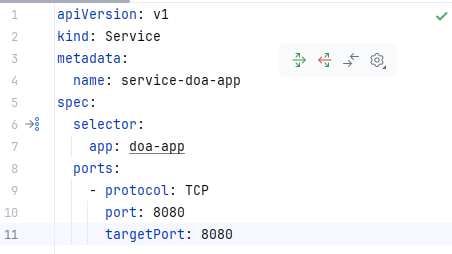
Observe que para a configuração eu fiz um volume externo direto, sem utilizar um PV ou PVC. Eu queria ter feito com o PVC, mas infelizmente percebi que a forma dos *Binds* entre os volumes e quais volumes o Claim vai escolher é um pouco mais complexa e não senti que entenderia a tempo. Ficou mais um ponto para aprofundar que parece bastante importante.

Todos os arquivos estão na pasta *prometheusData* do projeto.

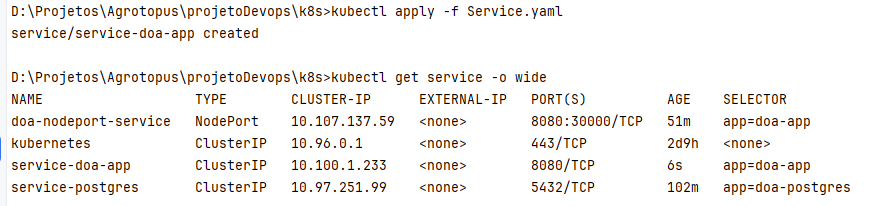
Aplicando a configuração do POD temos este resultado:

  
*Figura 27*

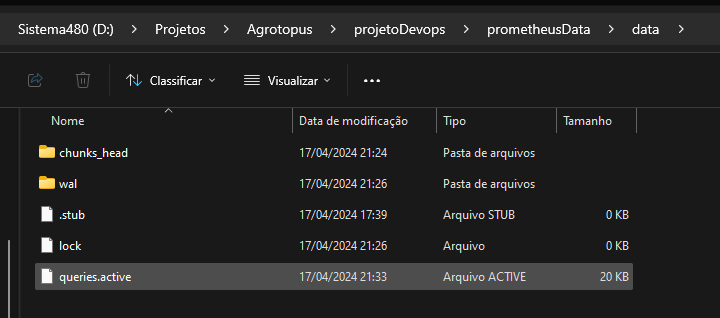
Para o acesso mais fácil da aplicação pelo **Prometheus**, criei um service básico:

  
*Figura 28*

Aplicando a configuração temos este resultado:

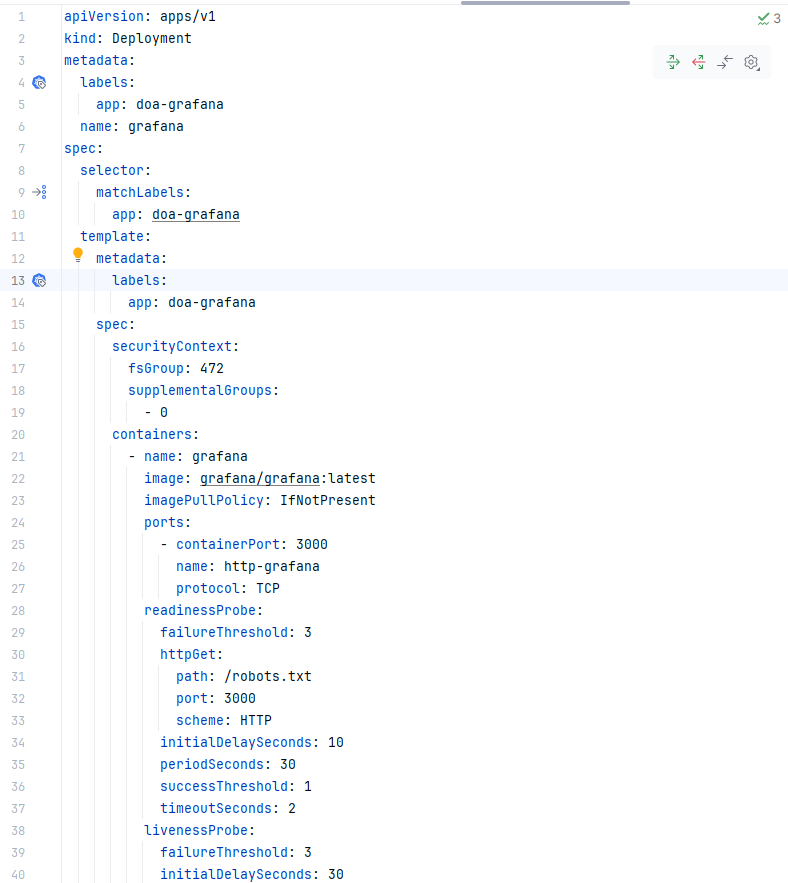
  
*Figura 29*

Com isso, já reparei que os arquivos na pasta “data” já começaram a serem gerados como mostro na figura abaixo:

  
*Figura 30*

# Preparação do Grafana

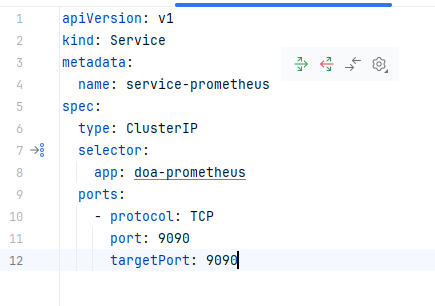
Para o Grafana criei esta configuração de DEPLOYMENT seguindo o padrão recomendado pela documentação do site oficial.

  
  
*Figura 31*

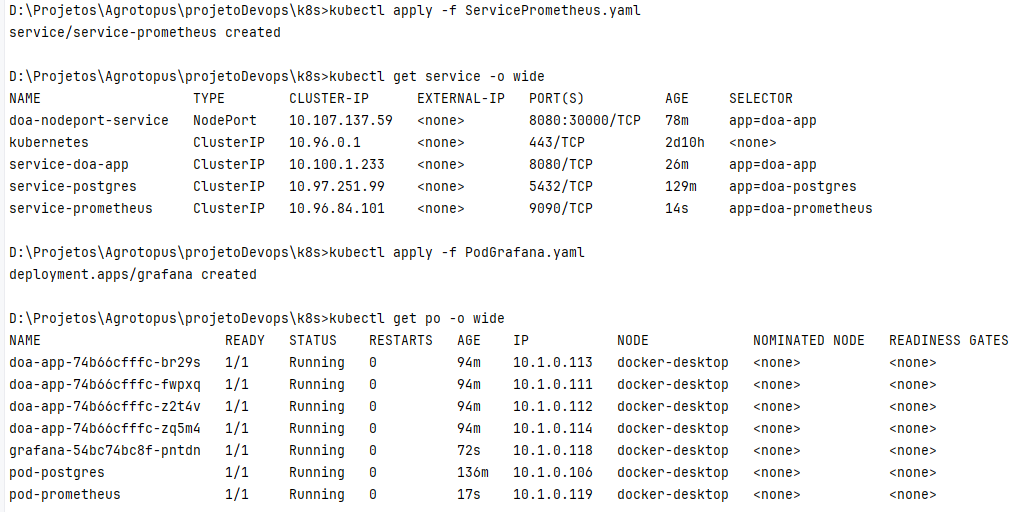
Um pouco extenso, mas é bom para termos a curiosidade de saber para que serve cada uma das propriedades envolvidas.

Novamente temos um volume externo básico para testes do tipo *fileSystem* onde acessamos diretamente a pasta de dados. Como dito anteriormente, ainda preciso me aprofundar nos PV e PVC para saber qual vai solicitar quem. Por este motivo anexei de forma direta.

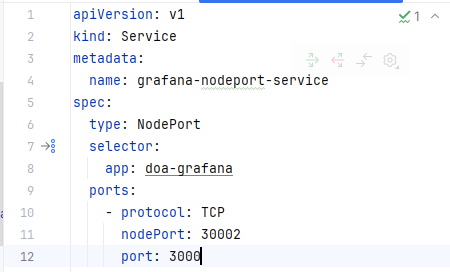
E para o acesso facilitado do Prometheus pelo Grafana, criei um simples service como abaixo:

  
*Figura 32*

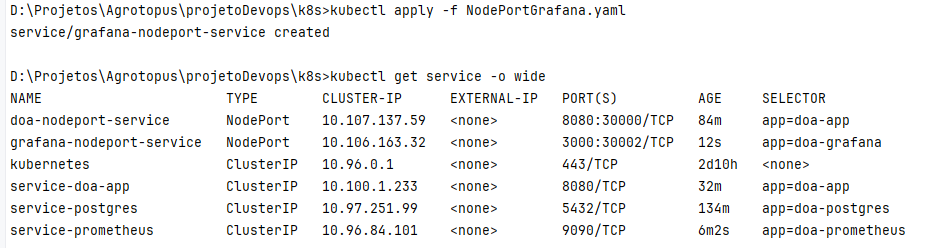
Aplicando as configurações do **Service** para o Prometheus e do **DEPLOYMENT** para o Grafana, temos estes resultados:

  
*Figura 33*

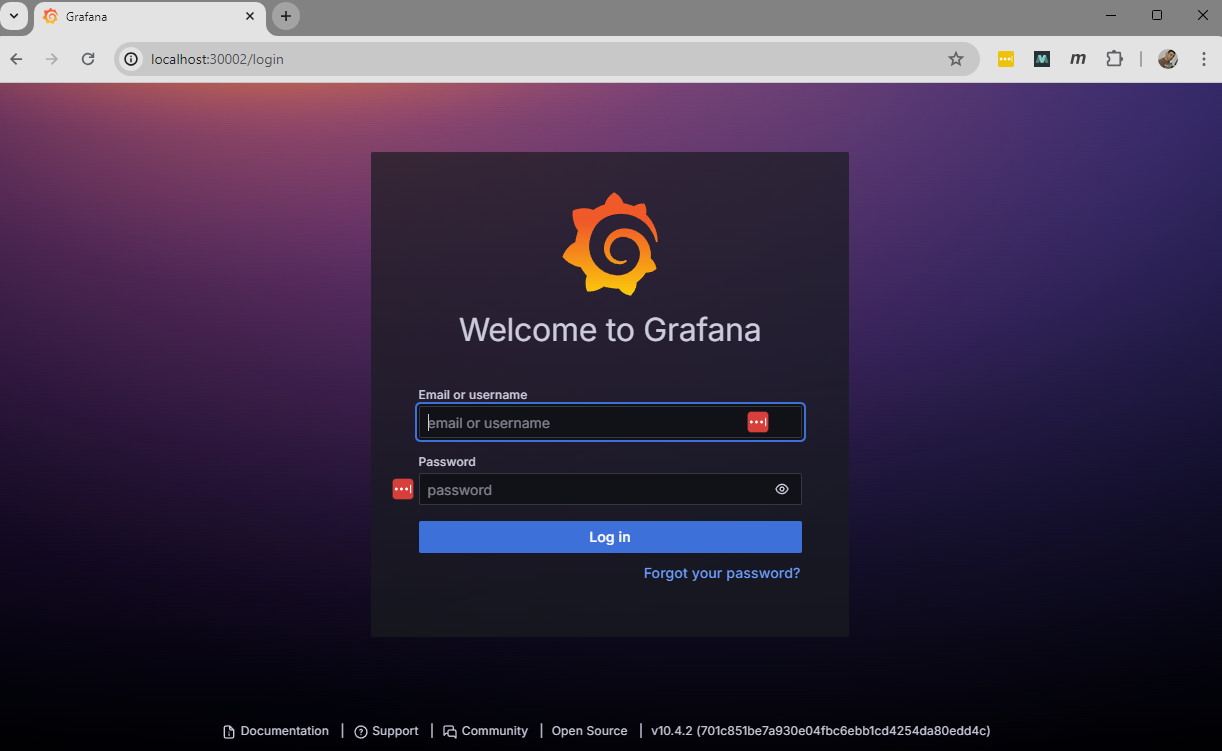
Agora precisamos expor o Grafana para fora do Cluster para utiliza-lo. Usaremos este serviço **NodePort**:

  
*Figura 34*

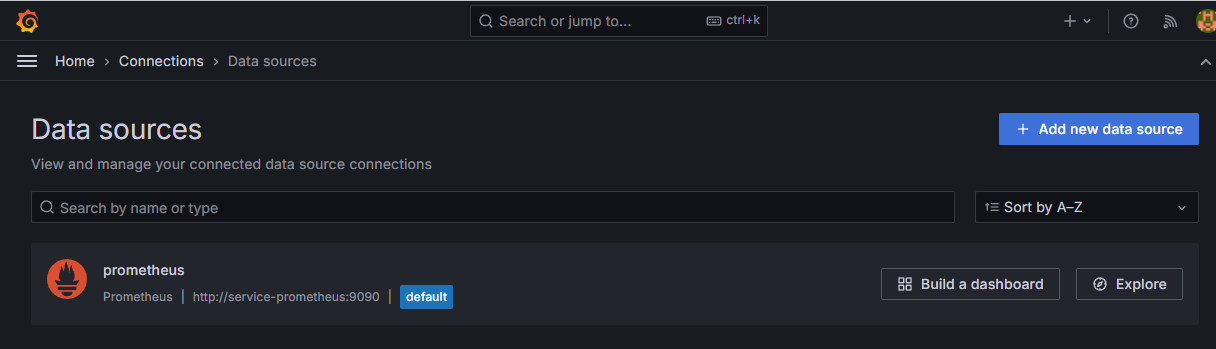
Aplicando o serviço, temos este resultado:

  
*Figura 35*

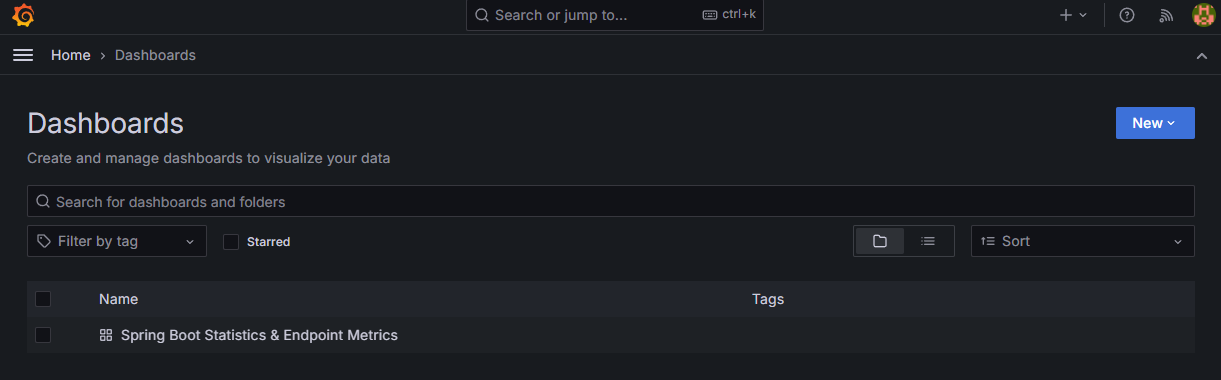
O resultado é o acesso do Grafana através de um Browser.

  
*Figura 36*

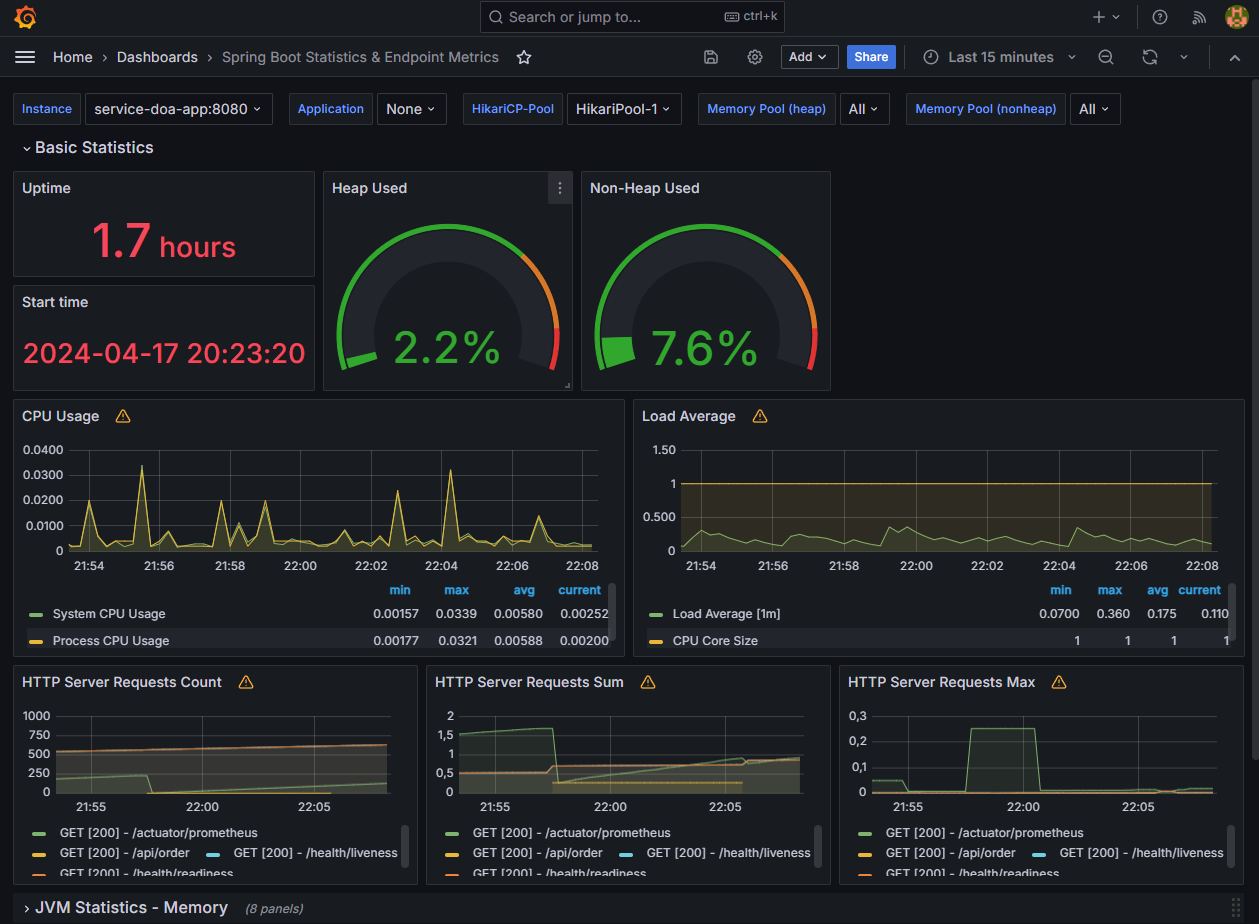
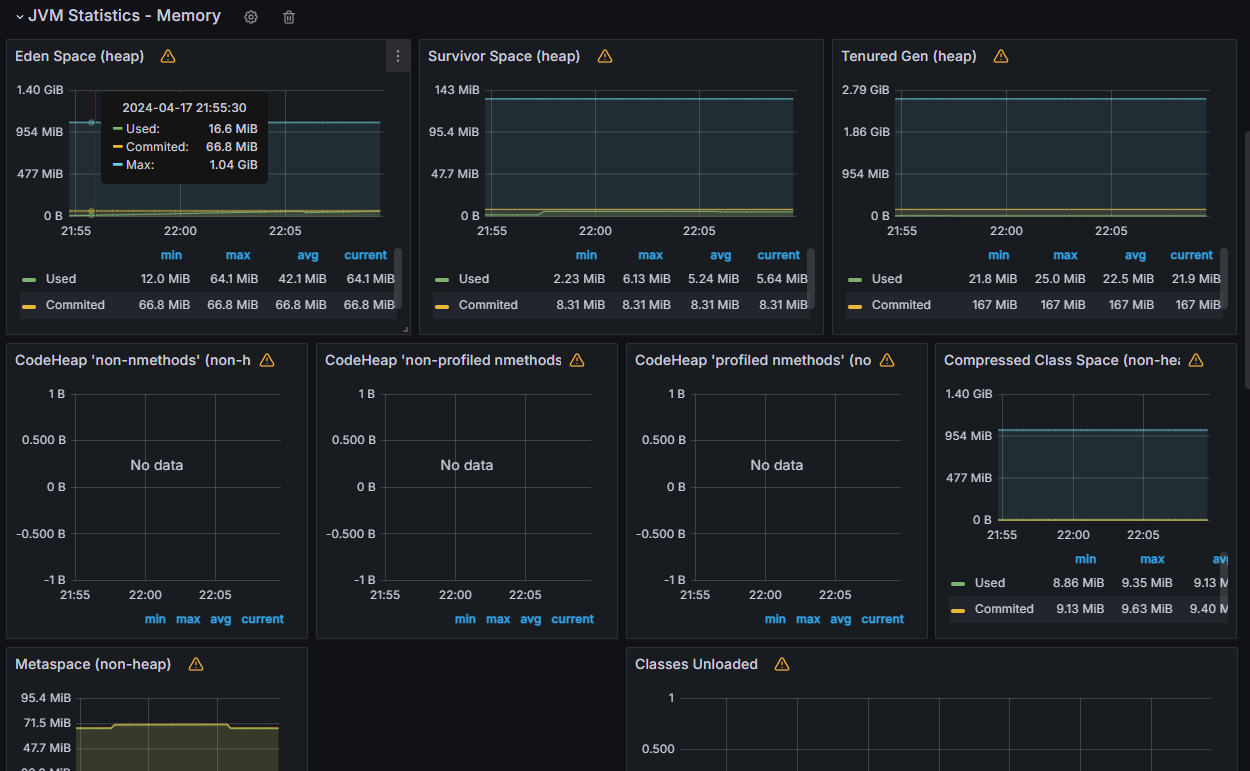
Criamos uma “DataSource” ou fonte de dados:

  
*Figura 37*

Criamos um Dashboard baseado em um template:

  
*Figura 38*

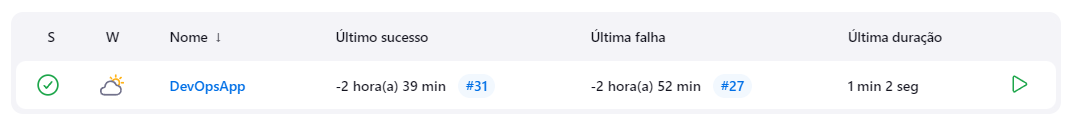
O resultado é um DashBoard com estas informações:

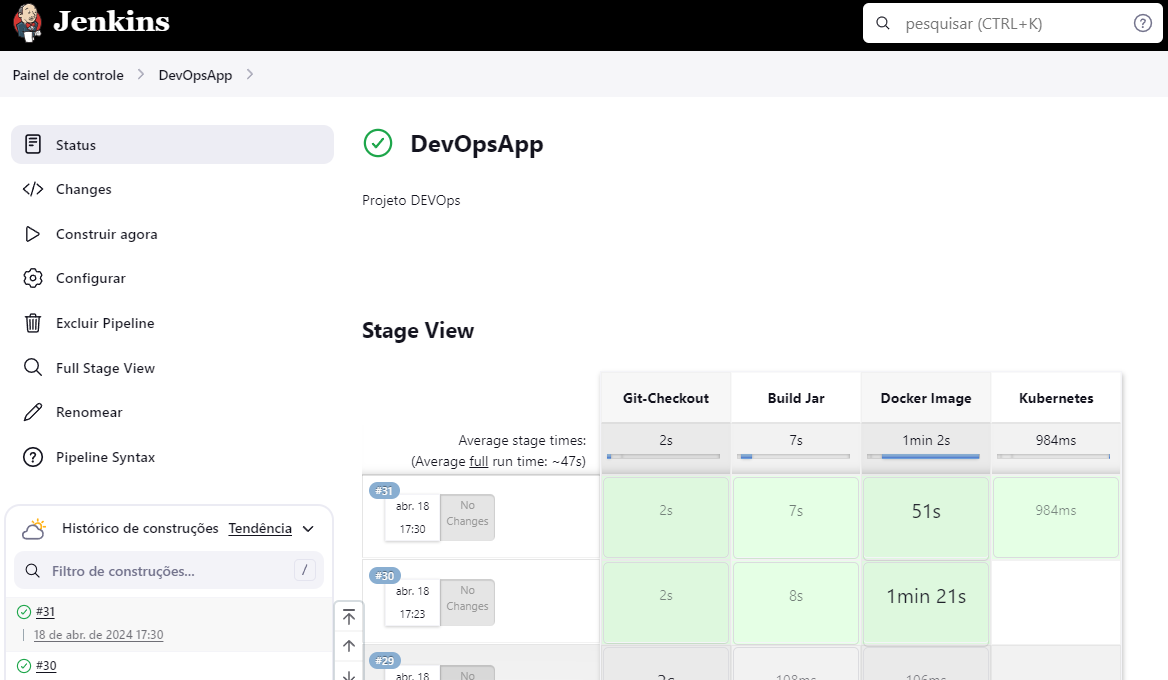
  
  
*Figuras 38 e 39*

# Preparação do Jenkins

Para a esteira de deploy utilizando o Jenkins, preparei a aplicação em minha máquina, onde já estava preparado o Docker Windows, o Maven e o Java 17, necessários para a construção e deploy.

Para esta esteira criei um projeto estilo Pipeline como nas imagens.

  
*Figura 40*

  
Figura 41

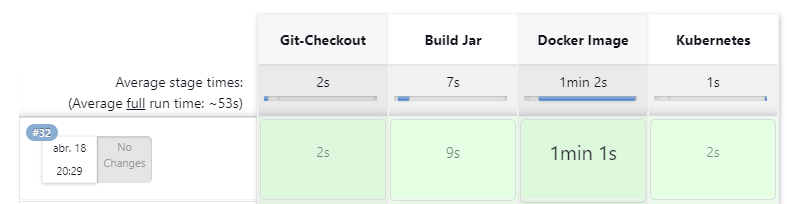
Fiz de uma forma simples o Pipeline a seguir:

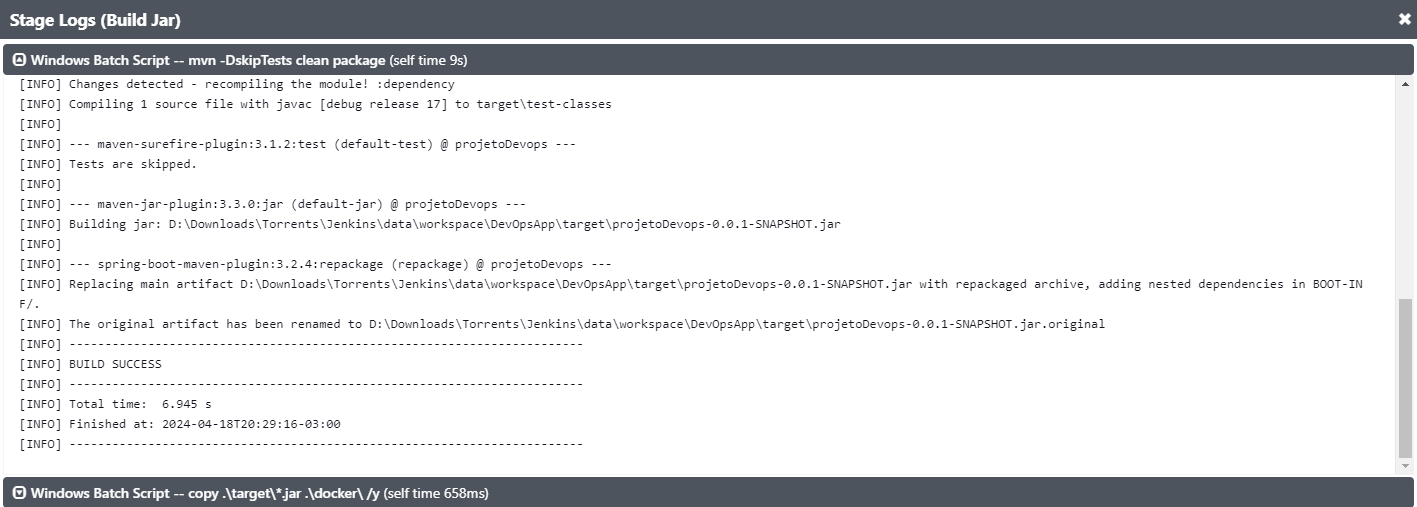
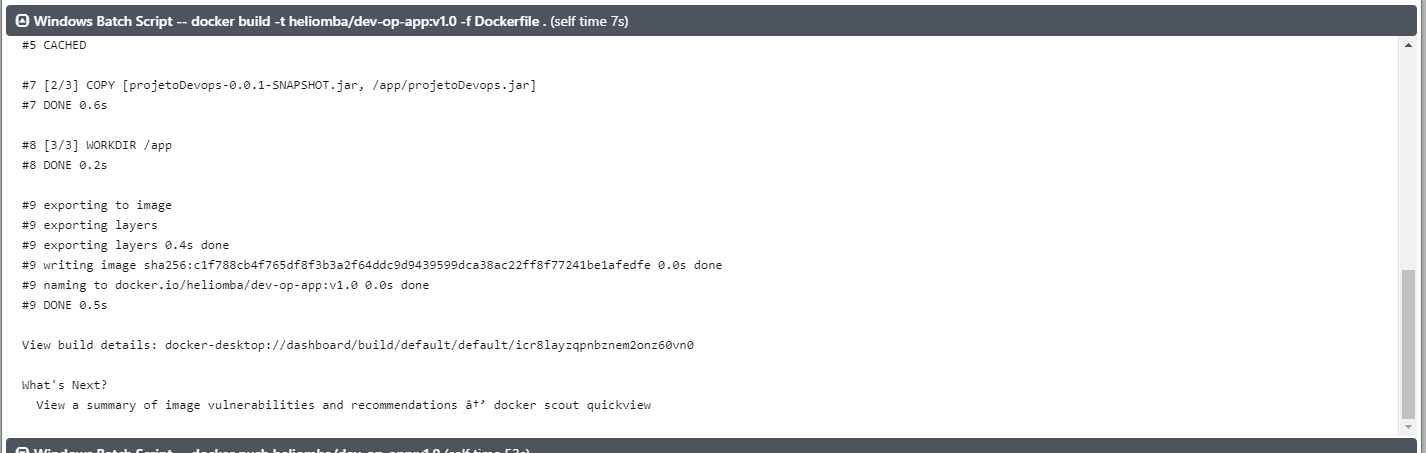
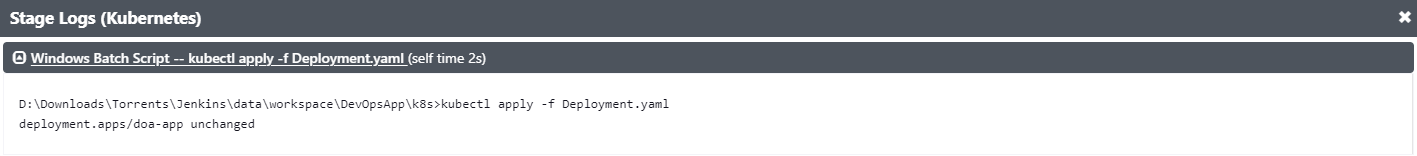
  
*Figura 42*

Como o ambiente é Windows, utilizei para os consoles o “**bat**”. E utilizei os arquivos gerados para eu criar o ambiente manualmente. Percebi que se eu tivesse feito através de comandos, poderia inserir variáveis no Pipeline.

Encontrei algumas ferramentas que inserem variáveis dentro do **.yml** como o *Helm* e o *Kustomize.io*. São interessantes para trocar as tags e outros valores de interesse.

Ao executar o Pipeline, temos as seguintes respostas:

  
*Figura 43*

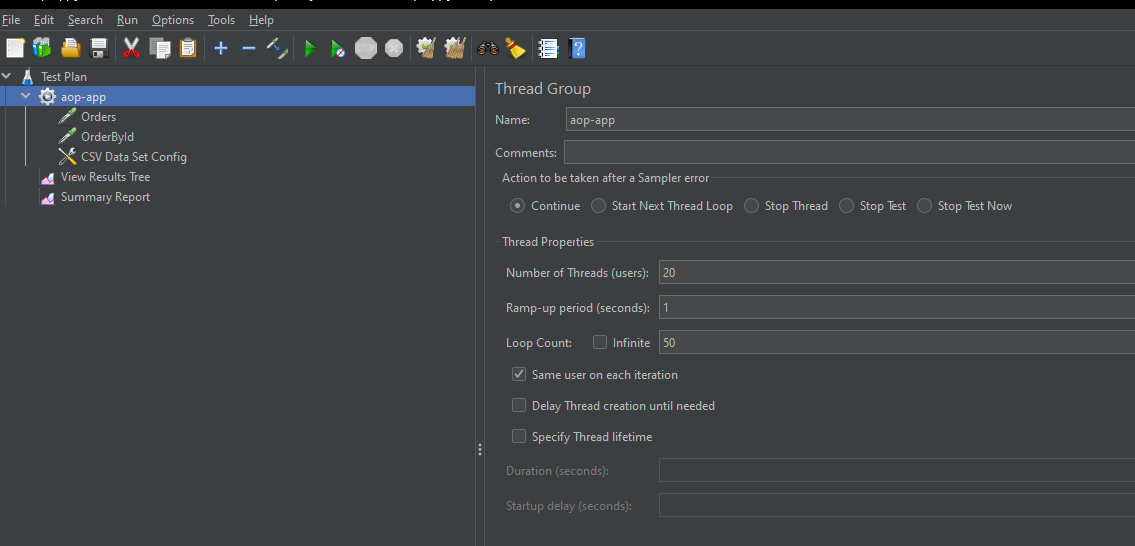
  
  
  
  
  
*Grupo de figuras 44*

Com o **imagePullPolicy** do Deployment fará que a atualização force o download e atualize os Pods.

Podemos criar gatilhos de necessário.

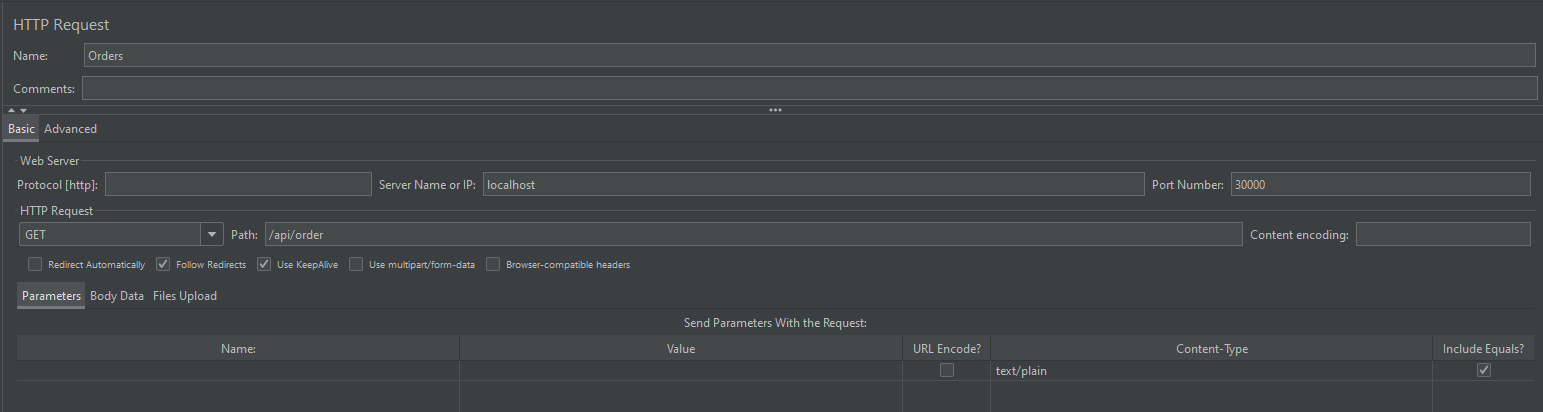
# Teste de Stress usando o jMeter

Fiz um pequeno teste de stress usando o Jmeter. Criei uma configuração como a seguir.

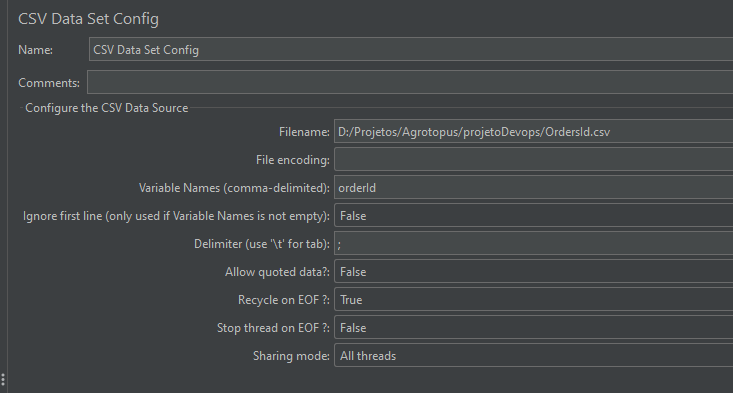
  
*Figura 45*

Criei dois *requests* simples.

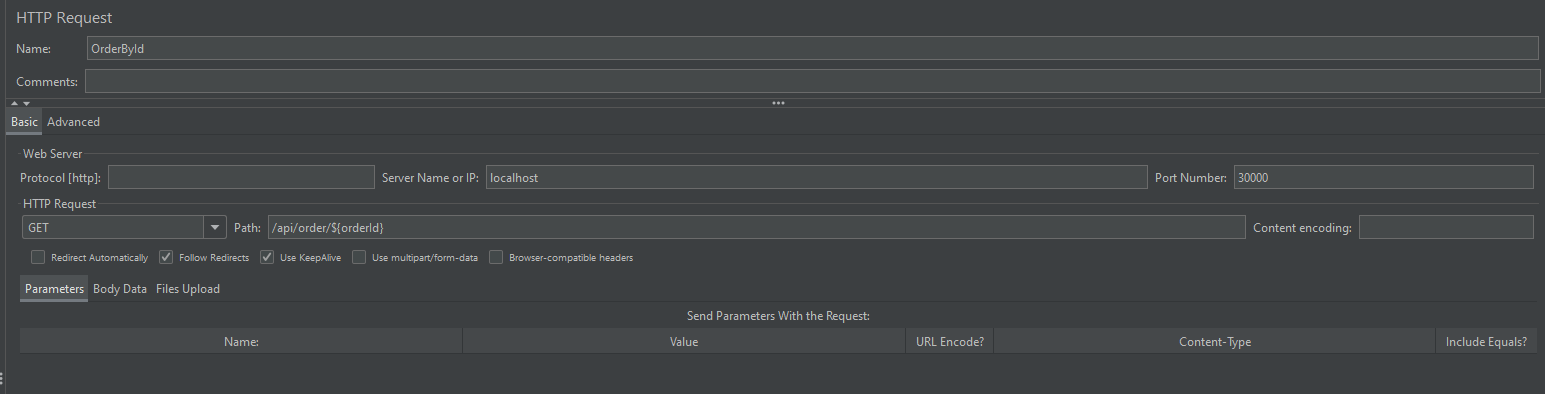
Um é o **Orders** que lista todos os registros de pedidos:

  
*Figura 46*

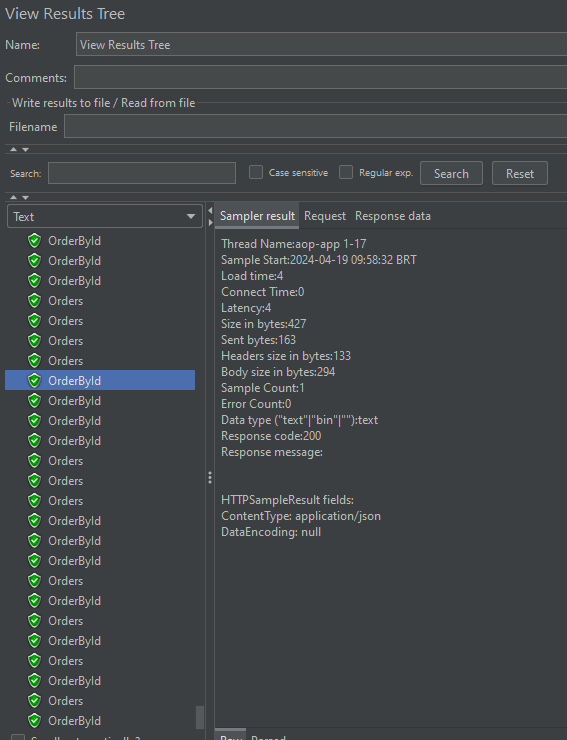
O outro é a solicitação pelo **id** do pedido. Para isso criei uma variável randômica utilizando um **CSV** como base. Este arquivo contem uma lista de valores que representam o identificador único do registro. Tenho o costume de usar sempre **UUID**s como identificadores.

  
*Figura 47*

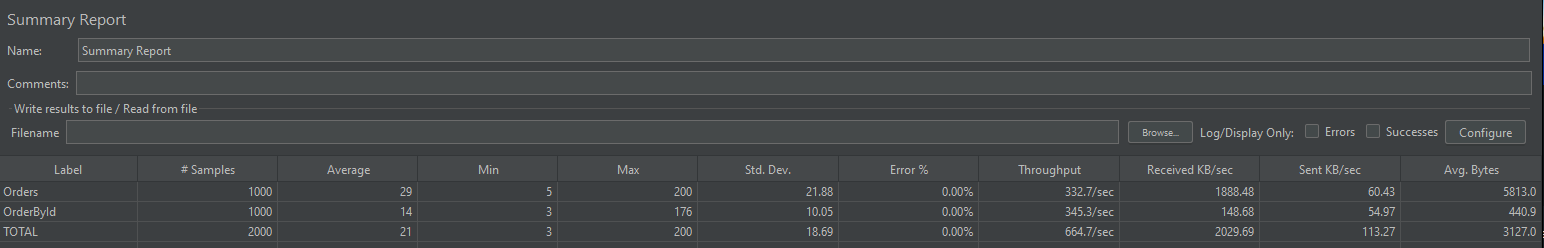
O request está abaixo:

  
*Figura 48*

Rodando o stress com 20 threads e 50 requests, temos a seguinte resposta no result tree:

  
*Figura 49*

E esta resposta no sumário:

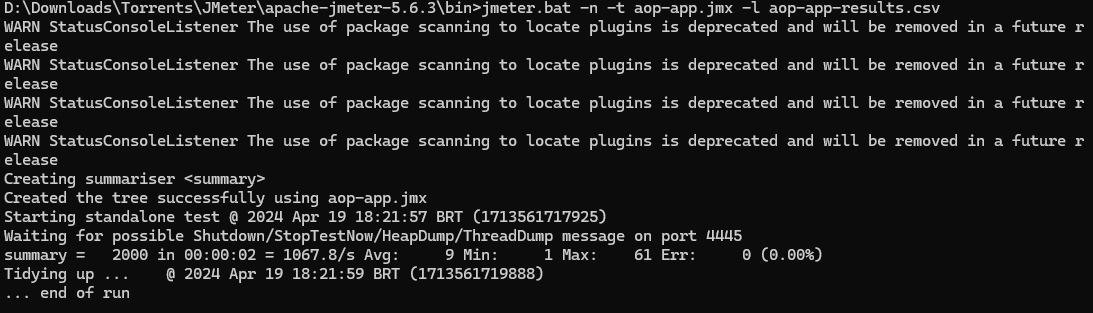
  
*Figura 50*

O JMeter também dá a possibilidade de rodar via comando para usarmos em scripts. Neste caso user o seguinte comando na pasta ***bin*** do jMeter:

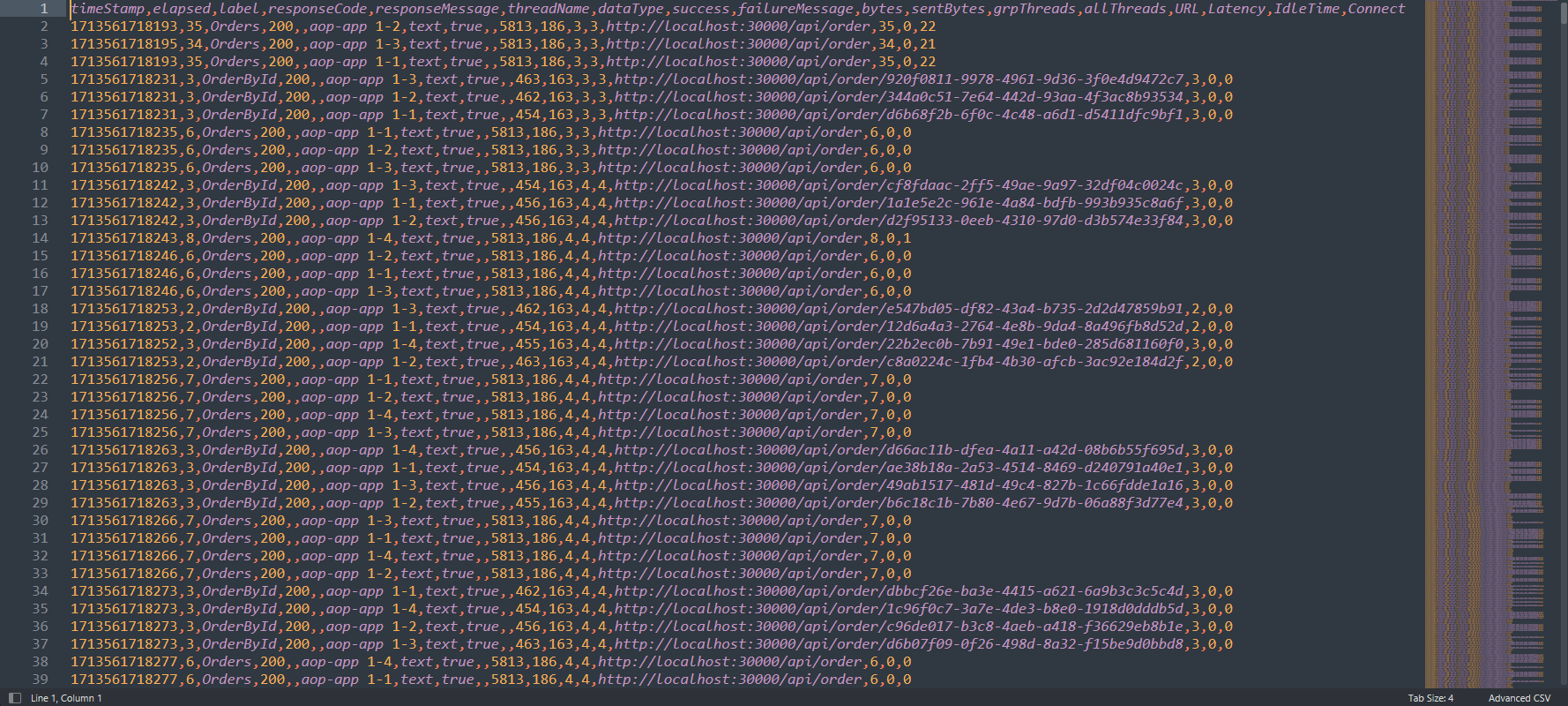
jmeter.bat -n -t aop-app.jmx -l aop-app-results.csv

O parâmetro **-n** é o NON\_GUI (sem interface), o **-t** é para indicar quané o arquivo de teste, no caso o **aop-app.jmx** e o **-l** é para indicar o nome do arquivo de resultado final.

O resultado da tela de comando é esse:

  
*Figura 51*

Indicando que o processo ocorreu com sucesso. E gerando este arquivo com os dados necessários.

  
*Figura 52*

Desta forma concluímos o teste de carga final.

# Considerações

Todos os arquivos do projeto, inclusive o banco do Postrgres e dados do Prometeus e Grafana estão no seguinte repositório público do GitHub.

<https://github.com/GodOfSun/mbaDevops>

Documento gerado para o projeto da disciplina de Integração Contínua, DevOps e Computação em Nuvem [24E1\_3], com o professor Leonardo Silva da Glória,

por Helio Teixeira Formaggio Junior.