Documentação Trabalho Devops

Aluno: Matheus Pereira da Silva

Disciplina: Devops

Visão Geral

Esta documentação descreve a arquitetura do projeto de devops desenvolvido em Java 17, abrangendo serviços individuais, suas funções, responsabilidades e interações.

Serviços

Micro Serviço de Frete

- Nome: frete-service
- Repositório: frete-service
- Função: Gerenciamento de todas as operações relacionadas ao transporte de cargas.
- Responsabilidades:
    - Gerenciamento dos detalhes da entrega.

Micro Serviço de Motorista

- Nome: motorista-service
- Repositório: motorista-service
- Função: Administração do cadastro e disponibilidade dos motoristas.
- Responsabilidades:
    - Fornecer informações sobre os motoristas para o frete-service durante a atribuição de entregas.

Bancos de Dados

Banco de Dados Frete

- Nome: db-frete
- Repositório: db-frete
- Função: Persistência de informações de fretes
- Responsabilidades:
    - Gerenciar o banco de dados de fretes.

Banco de Dados Motorista

- Nome: db-motorista
- Repositório: db-motorista
- Função: Persistência de informações de motoristas
- Responsabilidades:
    - Gerenciar o banco de dados de motoristas, incluindo informações como nome e CNH.

Kubernetes

- Nome: tms-kubernetes
- Repositório: tms-kubernetes
- Função: Gerenciamento de arquivos yaml kubernetes para deploy

Parte 1: Docker

- Utilize o Docker para criar uma imagem personalizada de alguma aplicação previamente feita por você.
- Publique a sua imagem no Dockerhub.

Nesta parte do trabalho, utilizaremos o Docker para criar imagens personalizadas de quatro aplicações previamente desenvolvidas: Frete Service, Database Frete, Motorista Service e Database Motorista.

O objetivo é empacotar cada aplicação em um contêiner Docker para facilitar a implantação e a execução em diferentes ambientes. Além disso, iremos publicar essas imagens no Dockerhub, um repositório público de imagens Docker, para que possam ser facilmente acessadas e utilizadas por outros desenvolvedores.

A seguir, detalharemos os passos para construir cada imagem, fazer o push para o Dockerhub e executar os contêineres correspondentes. Essas etapas garantirão que nossas aplicações estejam prontas para implantação em um ambiente Kubernetes.

Frete Service

1. Fazer build do jar

1. `mvn clean package`

2. Fazer build da imagem docker

1. `docker build -t matheuspsilva29/frete-service:1.0.3 .`

3. Fazer push da imagem

1. `docker push matheuspsilva29/frete-service:1.0.3`

4. Rodar a imagem

```bash
 docker run -d \
   --name frete-service \
          --network tms \
          -p 8082:8082 \
          matheuspsilva29/frete-service:1.0.3
```

Database Frete

1. Fazer build da imagem docker

1. `docker build -t matheuspsilva29/db-frete:1.0.2 .`

2. Fazer push da imagem

1. `docker push matheuspsilva29/db-frete:1.0.2`

3. Rodar a imagem

```bash
docker run -d \
  --name db-frete \
  --network tms \
  -e POSTGRES_USER=postgres \
  -e POSTGRES_PASSWORD=postgres \
  -e POSTGRES_DB=fretes \
  -p 5432:5432 \
  matheuspsilva29/db-frete:1.0.2
```

Motorista Service

1. Fazer build do jar

1. `mvn clean package`

2. Fazer build da imagem docker

1. `docker build -t matheuspsilva29/motorista-service:1.0.1 .`

3. Fazer push da imagem

1. `docker push matheuspsilva29/motorista-service:1.0.1`

4. Rodar a imagem

```bash
 docker run -d \
   --name motorista-service \
          --network tms \
          -p 8081:8081 \
          matheuspsilva29/motorista-service:1.0.1

```

Database Motorista

1. Fazer build da imagem docker

1. `docker build -t matheuspsilva29/db-motorista:1.0.0 .`

2. Fazer push da imagem

1. `docker push matheuspsilva29/db-motorista:1.0.0`

3. Rodar a imagem

```bash
docker run -d \
  --name db-motorista \
  --network tms \
  -e POSTGRES_USER=postgres \
  -e POSTGRES_PASSWORD=postgres \
  -e POSTGRES_DB=motoristas \
  -p 5433:5432 \
  matheuspsilva29/db-motorista:1.0.0
```

Parte 2: Kubernetes

Suba sua imagem em algum cluster kubernetes seguindo as seguintes especificações.:

- Utilize Deployment para subir sua aplicação com 4 réplicas.
- Exponha sua aplicação de modo que ela fique acessível fora do Cluster. (NODEPORT)
- Se sua aplicação fizer uso de banco de dados crie um POD com o mesmo e deixe-o acessível através do ClusterIP.
- Se sua aplicação não fizer uso de um BD suba uma imagem do Redis e crie um ClusterIP para o mesmo.
- Crie algum probe para sua aplicação (Readness ou Liveness.)

Nesta etapa do trabalho, o foco é subir imagens de aplicação em um cluster Kubernetes, seguindo especificações detalhadas. Utilizaremos o recurso Deployment para implementar cada aplicação com 4 réplicas, garantindo alta disponibilidade e escalabilidade.

Além disso, faremos com que as aplicações sejam acessíveis fora do cluster, utilizando o tipo de serviço NodePort. Se uma aplicação necessitar de um banco de dados, criaremos um POD correspondente e o tornaremos acessível por meio do ClusterIP. Caso contrário, faremos o uso de uma imagem do Redis e criaremos um ClusterIP para ele.

Para realizar essas tarefas, optamos por utilizar o Kind, uma ferramenta que simplifica a criação de clusters Kubernetes usando containers Docker como "nós". Isso proporciona um ambiente de desenvolvimento ágil e eficiente.

Antes de prosseguir, é importante garantir que o Docker e o Kind estejam instalados na máquina local. Em seguida, configuramos o Cluster Kubernetes com o Kind, criando um cluster com o nome "tms".

Após a configuração do cluster, subiremos os pods Kubernetes utilizando arquivos de manifesto, que descrevem os recursos necessários para cada aplicação, como deployments e serviços. Esses passos nos permitirão implantar e testar nossas aplicações em um ambiente Kubernetes de forma eficiente e padronizada.

Pré-requisitos

- Docker instalado na sua máquina
- Kind instalado na sua máquina

Configurando o Cluster Kubernetes com Kind

1. **Criando um cluster Kind:**

   `kind create cluster --name tms`

2. **Verificando se o cluster foi criado corretamente:**

   `kubectl cluster-info --context kind-tms`

3. **Subindo pods kubernetes**

   `kubectl apply -f k8s/deployment`

   `kubectl apply -f k8s/service`

Parte 3: Monitoramento e Dashboard

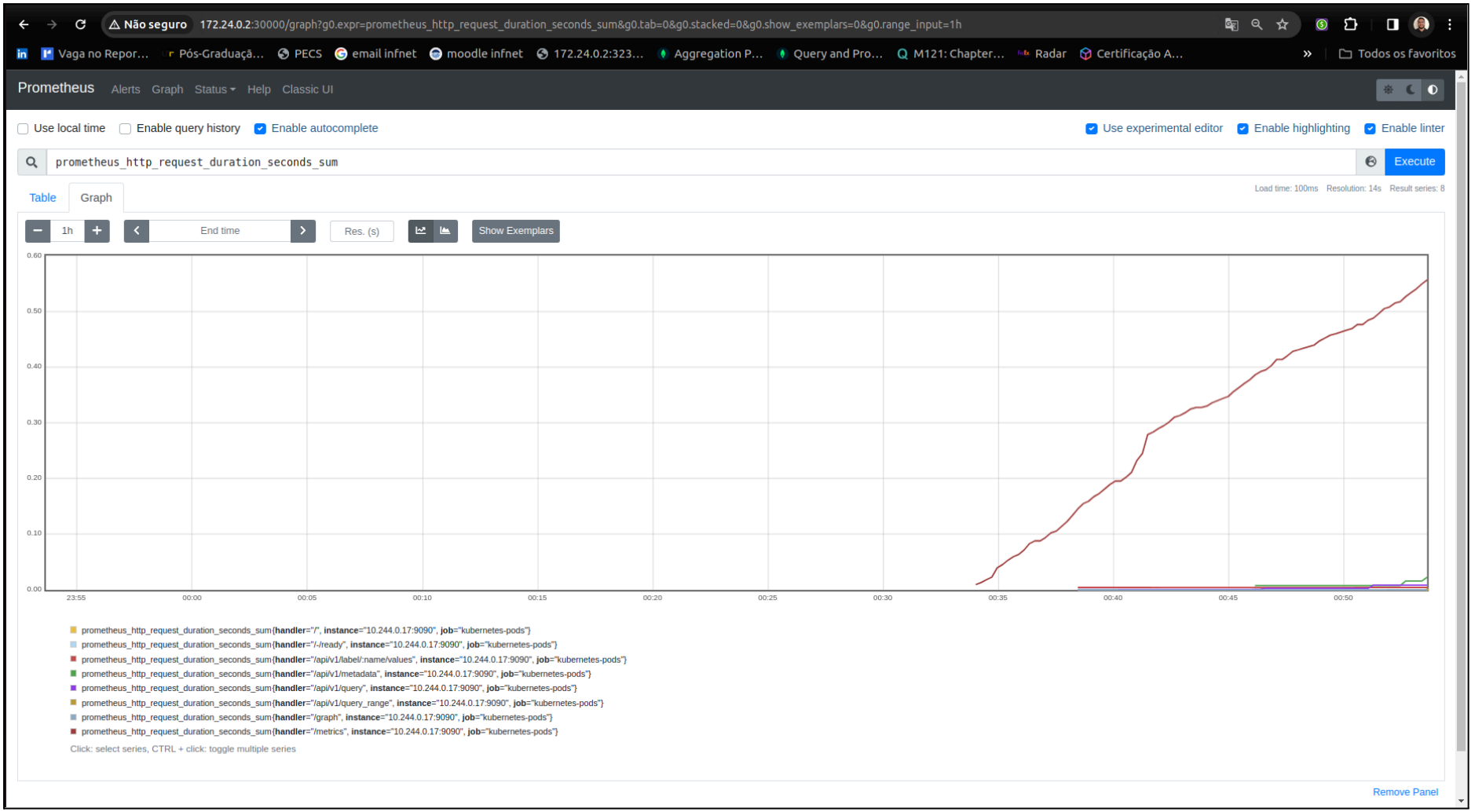
- Crie a estrutura para monitorar sua aplicação com o Prometheus e o Grafana (Ou qualquer ferramenta a sua escolha[Você deve ter um servidor de métricas e alguma ferramenta para dashboards.])
- Apenas o Grafana deverá ficar acessível para fora do Cluster.
- Utilize um PVC para escrever os dados do Prometheus de maneira persistente.
- Crie dashboards do Grafana que exponha dados sensíveis da sua aplicação. (Memória, cpu, etc.)

Para configurar o monitoramento com Prometheus e Grafana no seu cluster Kubernetes, siga estas etapas:

Nesta etapa do trabalho, estabeleceremos a infraestrutura necessária para monitorar nossas aplicações utilizando o Prometheus e o Grafana. Essas ferramentas nos permitirão coletar métricas importantes de desempenho e criar dashboards personalizados para análise.

Passo 1: Implantação do Prometheus

Iniciamos criando um arquivo YAML para implantar o Prometheus no cluster Kubernetes. Este arquivo contém as configurações essenciais para o funcionamento do Prometheus, incluindo a definição de jobs de scrape para coleta de métricas das aplicações. Utilizamos o comando `kubectl apply -f prometheus.yaml` para aplicar as configurações e implantar o Prometheus no cluster.



Prometheus Request Duration

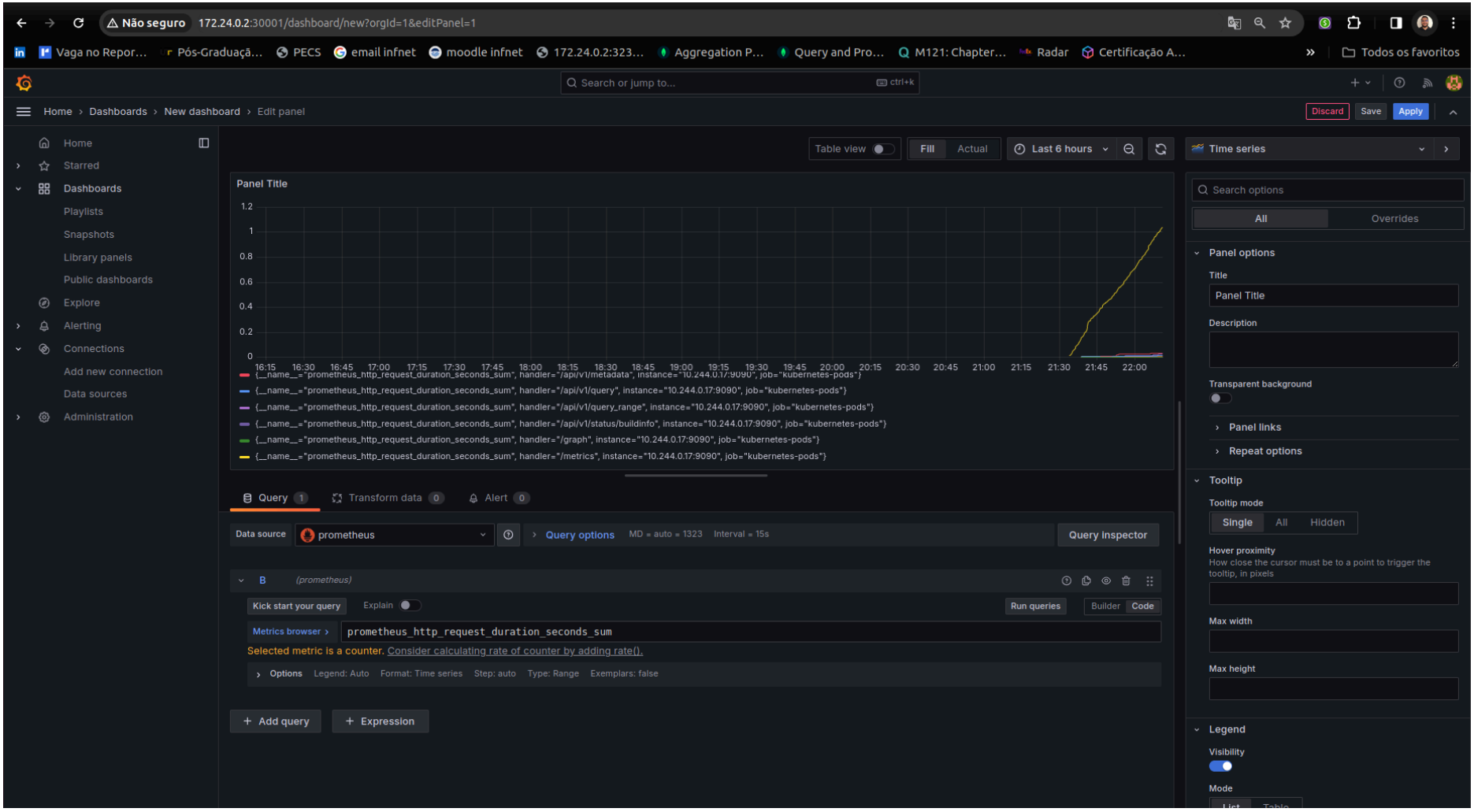
Passo 2: Configuração do Grafana

Em seguida, criamos outro arquivo YAML para implantar o Grafana no cluster Kubernetes. Similar ao passo anterior, este arquivo define as configurações necessárias para o Grafana e utiliza o comando `kubectl apply -f grafana.yaml` para realizar a implantação.

Passo 3: Exposição do Grafana fora do cluster

Para acessar o Grafana externamente ao cluster, criamos um serviço do tipo NodePort. Isso permite que o Grafana seja acessado através de um endereço IP e porta específicos. Utilizamos o comando `kubectl apply -f grafana-service.yaml` para criar e aplicar as configurações do serviço.

Com esses passos, estabelecemos uma estrutura sólida para monitoramento de nossas aplicações, garantindo acesso fácil e seguro às métricas coletadas pelo Prometheus e a criação de dashboards personalizados no Grafana.



Grafana Request Duration

Parte 4: Jenkins

- Utilize o Jenkins (ou qualquer ferramenta) para criar um pipeline de entrega do seu projeto.

Nesta etapa, utilizamos o Jenkins para criar um pipeline de entrega contínua para o projeto. O pipeline automatiza o processo de construção, teste e implantação das aplicações em um ambiente Kubernetes. As etapas principais do pipeline incluem:

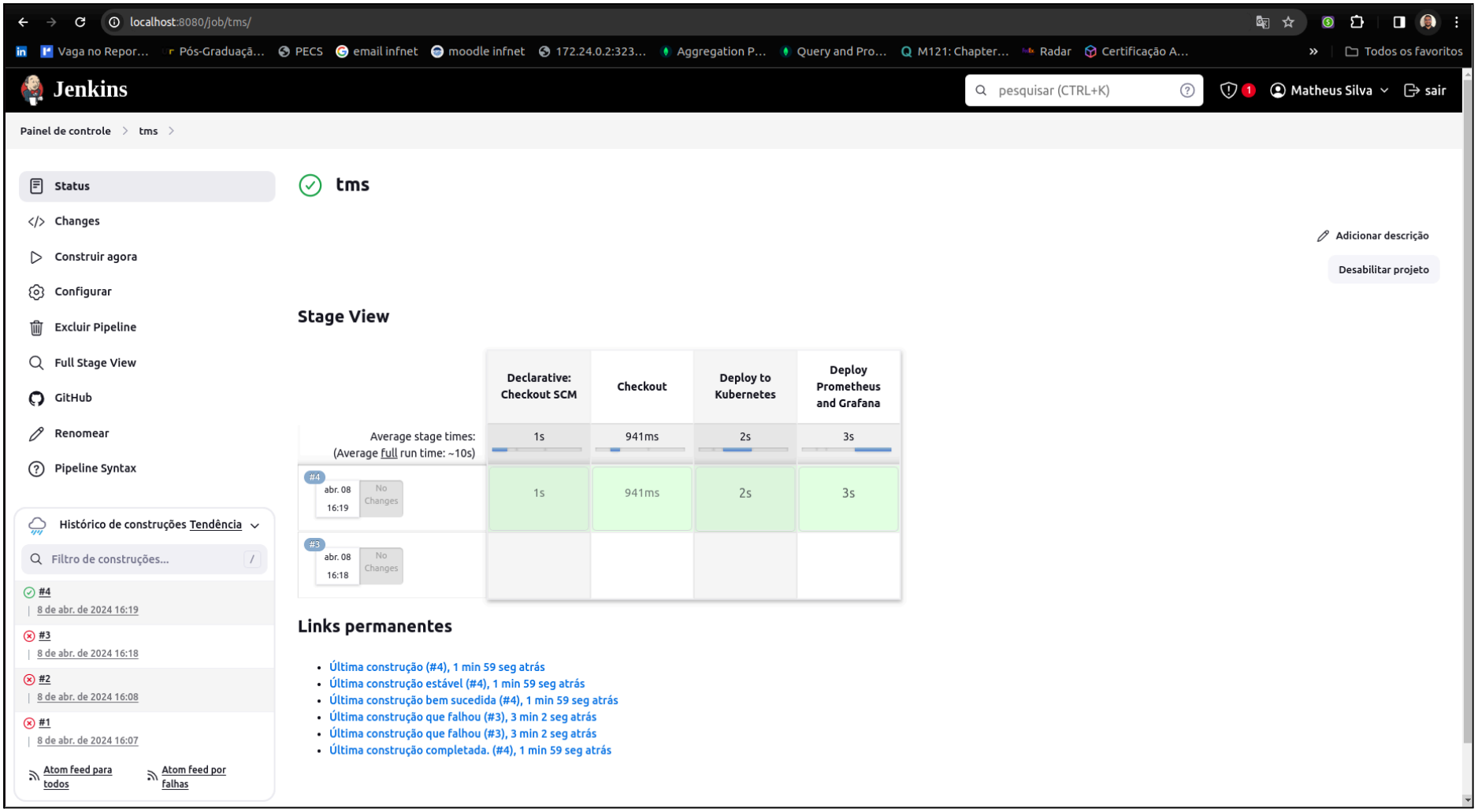
- Checkout do Código-fonte: O pipeline começa verificando o código-fonte do repositório configurado no Jenkins(https://github.com/Matheuspsilva/tms-kubernetes).
- Implantação no Kubernetes: O pipeline utiliza o `kubectl` para implantar os recursos Kubernetes necessários para as aplicações, como deployments e services.
- Implantação do Prometheus e Grafana: Além das aplicações, o pipeline também implanta os recursos do Prometheus e Grafana para monitoramento e visualização de métricas das aplicações.

Ao automatizar esses processos, o pipeline de entrega contínua torna a implantação de novas versões das aplicações mais eficiente e confiável, garantindo uma entrega rápida e consistente de software.

```
pipeline {
   agent any
   stages {
       stage('Checkout') {
           steps {
               checkout scm
           }
       }

       stage('Deploy to Kubernetes') {
           steps {
               script {
                   // Apply Kubernetes manifests
                   sh 'kubectl apply -f k8s/deployment'
                   sh 'kubectl apply -f k8s/service'
               }
           }
       }

       stage('Deploy Prometheus and Grafana') {
           steps {
               script {
                   // Apply Prometheus and Grafana manifests
                   sh 'kubectl apply -f prometheus/prometheus.yaml'
                   sh 'kubectl apply -f prometheus/prometheus-service.yaml'
                   sh 'kubectl apply -f k8s/deployment/grafana-deployment.yaml'
                   sh 'kubectl apply -f k8s/service/grafana-service.yaml'
               }
           }
       }
   }
}
```



Execução do pipeline no jenkins com sucesso

Parte 5: Stress test

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="5.0" jmeter="5.6.3">
 <hashTree>
   <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="motorista-serice">
     <elementProp name="TestPlan.user_defined_variables" elementType="Arguments" guiclass="ArgumentsPanel"
testclass="Arguments" testname="Variáveis Definidas Pelo Usuário">
       <collectionProp name="Arguments.arguments"/>
     </elementProp>
   </TestPlan>
   <hashTree>
     <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup" testname="Grupo de Usuários">
       <intProp name="ThreadGroup.num_threads">99</intProp>
       <intProp name="ThreadGroup.ramp_time">1</intProp>
       <boolProp name="ThreadGroup.same_user_on_next_iteration">true</boolProp>
       <stringProp name="ThreadGroup.on_sample_error">continue</stringProp>
       <elementProp name="ThreadGroup.main_controller" elementType="LoopController"
guiclass="LoopControlPanel" testclass="LoopController" testname="Controlador de Iteração">
         <stringProp name="LoopController.loops">1</stringProp>
         <boolProp name="LoopController.continue_forever">false</boolProp>
       </elementProp>
     </ThreadGroup>
     <hashTree>
       <HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy" testname="Requisição HTTP">
         <stringProp name="HTTPSampler.domain">172.24.0.2 </stringProp>
```

```
        <stringProp name="HTTPSampler.port">31149</stringProp>
        <stringProp name="HTTPSampler.path">/metrics</stringProp>
        <boolProp name="HTTPSampler.follow_redirects">true</boolProp>
        <stringProp name="HTTPSampler.method">GET</stringProp>
        <boolProp name="HTTPSampler.use_keepalive">true</boolProp>
        <boolProp name="HTTPSampler.postBodyRaw">false</boolProp>
        <elementProp name="HTTPsampler.Arguments" elementType="Arguments" guiclass="HTTPArgumentsPanel"
testclass="Arguments" testname="Variáveis Definidas Pelo Usuário">
          <collectionProp name="Arguments.arguments">
            <elementProp name="" elementType="HTTPArgument">
              <boolProp name="HTTPArgument.always_encode">false</boolProp>
              <stringProp name="Argument.value"></stringProp>
              <stringProp name="Argument.metadata">=</stringProp>
              <boolProp name="HTTPArgument.use_equals">true</boolProp>
            </elementProp>
          </collectionProp>
        </elementProp>
      </HTTPSamplerProxy>
      <hashTree/>
    </hashTree>
  </hashTree>
 </hashTree>
</jmeterTestPlan>
```

Teste motorista-service com Jmeter

Ferramentas e Tecnologias

- Java 17: Linguagem de programação principal.
- Spring Boot: Framework para simplificar a configuração e execução dos microsserviços.
- Spring Cloud: Para facilitar o desenvolvimento de alguns padrões comuns em sistemas distribuídos (configuração centralizada, descoberta de serviços, etc).
- Docker: Plataforma de conteinerização que facilita a criação, implantação e execução de aplicações em ambientes isolados.
- Kubernetes: Orquestrador de contêineres para automação, implantação, dimensionamento e gerenciamento de aplicações em contêineres.
- Prometheus: Sistema de monitoramento e alerta de código aberto para métricas de sistemas e serviços.
- Grafana: Plataforma de análise e visualização de métricas que permite criar dashboards e alertas personalizados.
- Jenkins: Ferramenta de automação de integração contínua e entrega contínua (CI/CD) para construir, testar e implantar software de forma automatizada.
- Kind (Kubernetes in Docker): Ferramenta para criação de clusters Kubernetes usando containers Docker como "nós" Kubernetes, útil para desenvolvimento e testes locais.
- Micrometer: Biblioteca utilizada para coletar métricas de aplicações e integrar com sistemas de monitoramento, como o Prometheus.
- Actuator do Spring Boot: Módulo que fornece endpoints para monitoramento e gerenciamento de aplicações Spring Boot.

- PostgreSQL: Sistema de gerenciamento de banco de dados relacional utilizado para armazenar dados de forma estruturada.
- Git: Sistema de controle de versão distribuído usado para rastrear mudanças no código-fonte durante o desenvolvimento de software.

Repositórios

https://github.com/Matheuspsilva/motorista-service

https://github.com/Matheuspsilva/frete-service

https://github.com/Matheuspsilva/eureka-server

https://github.com/Matheuspsilva/api-gateway-frete

https://github.com/Matheuspsilva/tms-kubernetes