Implantação de Aplicação Multicamada com Kubernetes e Helm: PlannerRun

Versão 2.0 - Orquestração e Automação

Autor:
Eduardo Henrique Spinelli - RA 800220

Disciplina:
DevOps

Professor:
Prof. Dr. Delano Medeiros Beder

Centro de Ciências Exatas e de Tecnologia
Departamento de Computação
São Carlos - SP
17 de julho de 2025

Conteúdo

1 Introdução

Este documento detalha a evolução do projeto PlannerRun, uma aplicação web multicamada projetada para gerar planos de treinamento de corrida personalizados. A primeira versão do projeto focou na conteinerização dos seus componentes utilizando Docker e na orquestração local com Docker Compose.

O objetivo desta segunda fase do projeto é migrar a infraestrutura da aplicação para um ambiente de orquestração mais robusto e escalável, utilizando **Kubernetes**. Para automatizar e gerenciar a complexidade da implantação, foi desenvolvido um **Helm Chart**, que define todos os recursos necessários de forma parametrizada e reutilizável.

A aplicação continua dividida em três componentes principais:

- **Frontend:** Uma interface web em Next.js responsável pela interação com o usuário.
- **Backend:** Uma API em Flask (Python) que lida com a lógica de negócio, integração com a API de pagamentos Stripe e comunicação com o banco de dados.
- **Banco de Dados:** Uma instância do PostgreSQL para a persistência dos dados dos clientes e da aplicação.

A migração para Kubernetes visa trazer benefícios como escalabilidade, alta disponibilidade, automação de deploy e gerenciamento simplificado de configurações e segredos, alinhando o projeto com as práticas modernas de DevOps.

2 Arquitetura no Kubernetes

A arquitetura da aplicação no Kubernetes foi desenhada para garantir a separação de responsabilidades, a escalabilidade e a segurança dos componentes. A comunicação entre os serviços é gerenciada internamente pelo Kubernetes, enquanto o acesso externo é controlado por um Ingress.

Figura 1: Diagrama da arquitetura da aplicação no Kubernetes.

Os principais artefatos Kubernetes utilizados para construir esta arquitetura são:

- **Deployments:** Gerenciam os Pods de cada aplicação (frontend, backend, db), garantindo que o número desejado de réplicas esteja sempre em execução.
- **Services:** Expõem os Deployments como um serviço de rede interno, permitindo que os componentes se comuniquem de forma estável (ex: o frontend se conecta ao backend via ‘backend-service‘).
- **Ingress:** Gerencia o acesso externo à aplicação, roteando o tráfego HTTP da URL ‘k8s.local‘ para o serviço do frontend. Ele também roteia as chamadas de API (‘/api/*‘) para o serviço do backend.
- **PersistentVolumeClaim (PVC):** Solicita armazenamento persistente para o banco de dados, garantindo que os dados não sejam perdidos caso o Pod do PostgreSQL seja reiniciado.
- **ConfigMap:** Armazena o script de inicialização do banco de dados (‘init.sql‘), que é montado no container do PostgreSQL para criar a tabela ‘clientes‘ na primeira execução.
- **Secret:** Armazena dados sensíveis, como senhas de banco de dados, chaves de API do Stripe e credenciais de e-mail, de forma segura e separada do código da aplicação.

Essa estrutura modular permite que cada componente seja atualizado e escalado de forma independente, uma das principais vantagens do uso de Kubernetes.

3 Artefatos Kubernetes Detalhados

A implantação da aplicação é definida por um conjunto de manifestos YAML, gerenciados pelo Helm. Abaixo, detalhamos os principais artefatos.

3.1 Deployment

Um Deployment é responsável por manter um conjunto de réplicas de um Pod em execução. Para o PlannerRun, temos três Deployments distintos.

3.1.1 Backend Deployment

Gerencia os Pods da API Flask. Ele define a imagem a ser utilizada, as portas e, crucialmente, as variáveis de ambiente necessárias, que são injetadas a partir de um Secret.

```
apiVersion apps/v1
kind Deployment
metadata
  name backend-deployment
spec
  replicas {{ .Values.replicaCount.backend }}
  template
    spec
      containers
      - name: backend
        image {{ .Values.backend.image }}{{ .Values.backend.tag }}
        ports
        - containerPort: 5000
        env
        - name: DB_HOST
          value db-service
        - name: DB_PASSWORD
          valueFrom
            secretKeyRef
              name planner-secrets
              key DB_PASSWORD
        # ... outras variaveis de ambiente
```

Listing 1: Trecho do ‘backend-deployment.yaml‘

3.1.2 Frontend Deployment

Gerencia os Pods da aplicação Next.js. A variável `NEXT_PUBLIC_API_URL` aponta para o caminho da API no Ingress, que por sua vez redireciona para o backend.

```
apiVersion apps/v1
kind Deployment
metadata
  name frontend-deployment
spec
  replicas {{ .Values.replicaCount.frontend }}
  template
    spec
      containers
      - name: frontend
        image {{ .Values.frontend.image }}{{ .Values.frontend.tag }}
        ports
        - containerPort: 80
        env
```

```
15          - name: NEXT_PUBLIC_API_URL
16            value http//{{ .Values.ingress.host }}/api
```

Listing 2: Trecho do ‘frontend-deployment.yaml‘

3.2 Service

Um Service fornece um ponto de acesso de rede estável para um conjunto de Pods.

3.2.1 Backend Service

Expõe o Deployment do backend na porta 5000, permitindo que outros serviços no cluster (como o Ingress) possam alcançá-lo pelo nome 'backend-service'.

```
apiVersion v1
kind Service
metadata
  name backend-service
spec
  selector
    app backend
  ports
    - protocol: TCP
      port 5000
      targetPort 5000
```

Listing 3: Arquivo 'backend-service.yaml'

3.2.2 Database Service

Expõe o banco de dados PostgreSQL na porta 5432, permitindo que a API backend se conecte a ele através do nome 'db-service'.

```
apiVersion v1
kind Service
metadata
  name db-service
spec
  selector
    app db
  ports
    - protocol: TCP
      port 5432
      targetPort 5432
```

Listing 4: Arquivo 'db-service.yaml'

3.3 Ingress

O Ingress gerencia o acesso externo aos serviços no cluster. Para o PlannerRun, ele define regras baseadas no host e no caminho da URL.

```
apiVersion networking.k8s.io/v1
kind Ingress
metadata
  name plannerrun-ingress
spec
  rules
    - host: {{ .Values.ingress.host }}
      http
        paths
          - path: /api
            pathType Prefix
            backend
              service
```

```
14                name backend-service
15                port
16                  number 5000
17          - path: /
18            pathType Prefix
19            backend
20              service
21                name frontend-service
22                port
23                  number 80
```

Listing 5: Arquivo ‘ingress.yaml‘

3.4 Persistência e Configuração

3.4.1 PersistentVolumeClaim (PVC)

Para garantir que os dados do PostgreSQL não sejam perdidos, um PVC é usado para solicitar armazenamento persistente.

```
apiVersion v1
kind PersistentVolumeClaim
metadata
  name {{ include "plannerrun-chart.fullname" . }}-db-pvc
spec
  accessModes
    - ReadWriteOnce
  resources
    requests
      storage {{ .Values.database.storage }}
```

Listing 6: Trecho do ‘db-pvc.yaml‘

3.4.2 ConfigMap

O ConfigMap armazena o script ‘init.sql‘, que é montado no container do PostgreSQL para inicializar o banco de dados na primeira execução.

```
apiVersion v1
kind ConfigMap
metadata
  name db-init-script
data
  init.sql |
    CREATE TABLE IF NOT EXISTS clientes (
        id SERIAL PRIMARY KEY,
        -- ... colunas da tabela
    );
```

Listing 7: Trecho do ‘db-configmap.yaml‘

3.4.3 Secret

A gestão de segredos é feita através de um objeto Secret do Kubernetes. O ‘secrets.yaml‘ no Helm Chart é um template que pega valores de um arquivo externo e os codifica em Base64, garantindo que dados sensíveis não sejam expostos no código.

```
apiVersion v1
kind Secret
metadata
  name planner-secrets
type Opaque
data
  DB_NAME {{ .Values.secrets.DB_NAME | b64enc | quote }}
  DB_USER {{ .Values.secrets.DB_USER | b64enc | quote }}
  DB_PASSWORD {{ .Values.secrets.DB_PASSWORD | b64enc | quote }}
  # ... outros segredos
```

Listing 8: Template ‘secrets.yaml‘

4 Helm e Automação do Deploy

Para simplificar a implantação e o gerenciamento da aplicação no Kubernetes, foi criado um Helm Chart. O Helm atua como um gerenciador de pacotes para o Kubernetes, permitindo definir, instalar e atualizar aplicações complexas de forma consistente.

4.1 Estrutura do Helm Chart

O Chart do PlannerRun está localizado em ‘helm/plannerrun-chart/‘ e possui a seguinte estrutura:

- **Chart.yaml:** Arquivo de metadados que descreve o chart, como nome e versão.
- **values.yaml:** Arquivo que contém os valores padrão e configuráveis para a implantação. Segredos e dados sensíveis são omitidos deste arquivo e gerenciados separadamente.
- **templates/:** Diretório que contém os manifestos Kubernetes parametrizados (templates) para todos os recursos da aplicação (Deployments, Services, etc.).

4.2 Script de Automação

O processo de implantação foi totalmente automatizado com o script ‘deploy-to-minikube.sh‘. Este script executa uma sequência de tarefas essenciais:

1. **Validação:** Verifica se o arquivo de segredos ‘secrets.values.yaml‘ existe.
2. **Inicialização do Cluster:** Inicia o Minikube e habilita o addon de Ingress.
3. **Configuração do Docker:** Aponta o ambiente Docker local para o daemon do Minikube, permitindo que as imagens construídas localmente sejam usadas pelo cluster.
4. **Build das Imagens:** Constrói as imagens do frontend e do backend.
5. **Configuração de Rede:** Adiciona uma entrada no arquivo ‘/etc/hosts‘ para mapear a URL ‘k8s.local‘ ao IP do Minikube.
6. **Deploy com Helm:** Executa o comando ‘helm upgrade –install‘, que instala ou atualiza a aplicação no cluster. Ele utiliza o arquivo ‘secrets.values.yaml‘ para injetar os segredos de forma segura, sem expô-los na linha de comando.

```
# ... (inicializacao e build)

# Valida se o arquivo de segredos existe
if [ ! -f "$SECRETS_FILE" ]; then
  echo "Erro: Arquivo de segredos '$SECRETS_FILE' nao encontrado."
  exit 1
fi

# Instala/Atualiza o Helm Chart com segredos de um arquivo
helm upgrade --install "$CHART_NAME" "$CHART_PATH" \
  -f "$SECRETS_FILE" \
  --set ingress.host="$HOST_NAME" \
  --wait --timeout 5m
```

Listing 9: Trecho do script ‘deploy-to-minikube.sh‘

5 Conclusão

A migração da aplicação PlannerRun de um ambiente Docker Compose para uma infraestrutura gerenciada por Kubernetes e Helm representa um avanço significativo em termos de maturidade de DevOps.

Os principais resultados alcançados foram:

- **Infraestrutura como Código (IaC):** Todos os recursos da aplicação são definidos de forma declarativa nos manifestos do Helm Chart, permitindo implantações consistentes e reproduzíveis.
- **Escalabilidade e Resiliência:** A arquitetura com Deployments e Services permite que cada componente seja escalado horizontalmente de forma independente e que o Kubernetes gerencie automaticamente a recuperação de Pods em caso de falhas.
- **Automação do Deploy:** O script ‘deploy-to-minikube.sh‘ simplifica todo o processo de implantação, desde a configuração do ambiente até a instalação da aplicação, reduzindo a chance de erros manuais.
- **Gerenciamento Seguro de Segredos:** A adoção de um arquivo de valores separado para segredos (‘secrets.values.yaml‘), não versionado no Git, garante que as credenciais sensíveis sejam mantidas seguras e fora do código-fonte.

Este trabalho prático demonstrou com sucesso a aplicação dos conceitos de orquestração de contêineres, automação e gerenciamento de configuração, consolidando as bases para um ciclo de vida de desenvolvimento e operações mais eficiente e seguro.