

CASO 1

O DEPLOY DE SEXTA-FEIRA

LINUXTIPS



O CASO

"É SÓ UMA LINHA DE CÓDIGO"

Sexta-feira, 4 da tarde. O time de produto pede uma alteração pequena, quase cosmética, em um dos micro-serviços. "É só mudar um texto, super rápido", eles dizem. Você faz a alteração, roda os testes locais, tudo verde. Parece seguro. Você abre o terminal, confiante, e aplica o novo manifesto do Deployment.



```
$ kubectl apply -f deployment.yaml  
deployment.apps/meu-servico configured
```

Pronto. Você avisa no canal do Slack: "Deploy da feature XPTO em produção!" e se prepara para fechar o notebook e começar o fim de semana. Mas então, seu celular vibra. E vibra de novo. É o PagerDuty. O canal de alertas do time começa a piscar em vermelho no Slack. O dashboard do Grafana, antes um mar de tranquilidade verde, agora parece uma árvore de natal desgovernada.

Seu colega manda uma mensagem: "Cara, o serviço X caiu depois do seu deploy". Aquele café que você ia tomar vai ter que esperar.

O que aconteceu? Como uma mudança de uma linha de texto derrubou um serviço inteiro?

O DIAGNÓSTICO:

RESPIRA FUNDO E

COMEÇA DO BÁSICO

Nessas horas, o pânico não ajuda. A primeira regra de um bom profissional SRE ou Dev é: respira fundo e comece do básico. Não adianta chutar. Precisamos de dados. Nossa missão é seguir um roteiro lógico de troubleshooting para encontrar a causa raiz, corrigir o problema e, claro, entender por que ele aconteceu para que não se repita.



PASSO 1: QUAL O ESTADO DA APLICAÇÃO? (KUBECTL GET PODS)

O primeiro comando é sempre o **kubectl get pods**. Ele é o nosso primeiro contato com a realidade, a visão geral do campo de batalha. Precisamos saber o status dos nossos Pods.

```
$ kubectl get pods
```

O resultado já mostra que a situação não é boa

```
meu-servico-5f... Running  
meu-servico-5f... Running  
meu-servico-7d... CrashLoopBackOff  
meu-servico-7d... Pending
```

Ok, temos um cenário misto. Alguns pods da versão antiga (hash **5f...**) ainda estão no ar, mas os da nova versão (hash **7d...**) estão sofrendo. Um está em um loop de crash e outro nem consegue iniciar. Isso já nos diz que o problema está na nova versão.

PASSO 2: APROFUNDANDO NOS LOGS E EVENTOS (KUBECTL DESCRIBE POD)

Agora que temos um suspeito (**meu-servico-7d...**), vamos interro-gá-lo. O **kubectl describe pod** é a nossa principal ferramenta de investigação para entender o histórico de um Pod.

```
$ kubectl describe pod meu-servico-7d...
```

Analizando a seção **Events**, vemos duas mensagens que acendem um alerta:

Warning Unhealthy Readiness probe failed: O Kubernetes tentou verificar se a aplicação estava pronta, mas a checagem falhou. Isso significa que, mesmo que o Pod estivesse Running, ele não estava apto a receber tráfego. (Conceito do Day-4)

E no status do container, a razão da última terminação: **Reason: OOMKilled. Out of Memory Killed.** O processo dentro do container usou mais memória do que o limite e o Kernel do Node o matou. (Conceito do Day-2)

Aha! Temos nossa primeira causa concreta. A aplicação está consumindo mais memória do que o esperado, e as checagens de saúde estão falhando.

PASSO 3: COMO FOI A TRANSIÇÃO? (KUBECTL GET REPLICASET)

Um Deployment não gerencia Pods diretamente. Ele usa ReplicaSets para isso. Cada vez que você faz um deploy, um novo ReplicaSet é criado. Entender o que aconteceu com eles nos ajuda a entender o rollout.

```
$ kubectl get replicaset
```

O resultado mostra o ReplicaSet da versão antiga, com 2 pods, e o da nova versão, com apenas 1 pod READY. O Deployment está tentando criar os novos, mas como eles morrem (OOMKilled), o ReplicaSet não consegue atingir o número de réplicas desejado. Ele está fazendo o trabalho dele, mas os Pods não colaboram.
(Conceito do Day-4)



PASSO 4: REVISANDO

O MANIFESTO

(KUBECTL DESCRIBE

DEPLOYMENT)

Se os Pods estão com problemas e o ReplicaSet está sofrendo, a causa raiz provavelmente está na especificação, no "DNA" deles. Vamos olhar a configuração do Deployment em si.

```
$ kubectl describe deployment meu-servico
```

Analizando a especificação do template do Pod, a verdade vem à tona. Na seção Containers, os campos resources (requests e limits) e probes (liveness, readiness, startup) estão completamente ausentes.

Esse serviço foi criado há muito tempo, provavelmente por alguém que não seguiu as melhores práticas. Ele sempre funcionou na sorte. A pequena mudança de código que você fez, por algum motivo, aumentou um pouco o consumo de memória, e foi a gota d'água que fez o castelo de cartas desmoronar.

A CAUSA RAIZ: UMA DÍVIDA TÉCNICA QUE VENCEU

O problema não foi a sua linha de código. O problema foi uma dívida técnica esperando para ser cobrada. A aplicação estava rodando sem as garantias mínimas de estabilidade que o Kubernetes oferece. Foi uma combinação de fatores:

FALTA DE GESTÃO DE RECURSOS (DAY-2):

Sem requests e limits, o Pod era uma bomba-relógio. Qualquer pequena variação no consumo de memória poderia levá-lo a ser morto pelo sistema (OOMKilled).

ESTRATÉGIA DE ROLLOUT OTIMISTA (DAY-3):

A estratégia RollingUpdate padrão é boa, mas sem readiness probes, o Kubernetes não sabia se o novo Pod estava realmente pronto antes de matar o antigo, causando a instabilidade durante o deploy.

AUSÊNCIA DE VERIFICAÇÕES DE SAÚDE (DAY-4):

A falta das Probes foi o principal problema. Sem a Readiness Probe, o Kubernetes direcionava tráfego para Pods que ainda estavam iniciando ou já estavam quebrados. Sem a Liveness Probe, um Pod que travasse ficaria no ar indefinidamente, como um "zumbi".



A SOLUÇÃO: O HOTFIX SALVADOR

Hora de corrigir o problema da forma certa. Você cria um novo branch, hotfix/add-resources-and-probes, e prepara um manifesto robusto, aplicando todo o seu conhecimento.

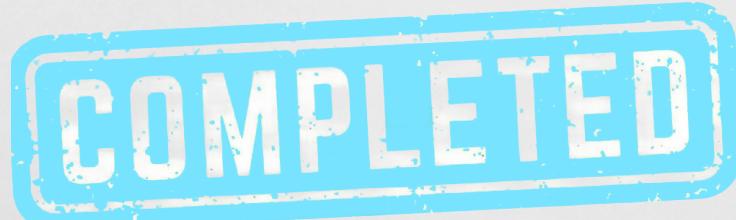
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: meu-servico
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    spec:
      containers:
        - name: meu-servico-container
          image: minha-imagem:v1.1-hotfix
          resources: # Definindo uma "cota" de
recursos justa
          requests:
            cpu: "250m"
            memory: "256Mi"
          limits:
            cpu: "500m"
            memory: "512Mi"
        startupProbe: # Dando um tempo para a
aplicação iniciar
```



```
httpGet:  
  path: /healthz  
  port: 8080  
failureThreshold: 30  
periodSeconds: 10  
livenessProbe: # Checando se a aplicação  
não travou  
  httpGet:  
    path: /healthz  
    port: 8080  
readinessProbe: # Verificando se está  
pronta para o trabalho  
  httpGet:  
    path: /readyz  
    port: 8080
```



Você aplica o hotfix. Desta vez, você observa o rollout com kubectl rollout status deployment/meu-servico. Os Pods sobem um a um, de forma controlada. O dashboard volta a ficar verde. Você respira aliviado.



LIÇÕES APRENDIDAS:

UM POST-MORTEM

PARA O TIME

Na segunda-feira, na reunião de time, você apresenta o post-mortem. Aquele deploy de sexta-feira se tornou uma lição valiosa para todos. As principais conclusões foram:

RECURSOS NÃO SÃO OPCIONAIS (DAY-2):

Toda aplicação em produção DEVE ter requests e limits definidos. É a base da estabilidade.

DEPLOY SEGURO É DEPLOY COM ESTRATÉGIA (DAY-3):

A estratégia RollingUpdate é poderosa, mas só funciona bem em conjunto com as probes.

SUA APLICAÇÃO PRECISA SABER CONVERSAR (DAY-4):

As Probes são a linguagem que a sua aplicação usa para dizer ao Kubernetes como ela está se sentindo. Sem elas, o Kubernetes opera às cegas.

O que começou como um susto de sexta-feira terminou com um sistema mais robusto e um time mais experiente. E essa é uma das melhores partes de trabalhar com tecnologia.

Problema resolvido!