



Kubernetes e Java: Integração Avançada com Suporte a LTS 8, 11 e 17, Exemplos Técnicos e Boas Práticas

Por EducaCiência FastCode

A crescente adoção de contêineres e microsserviços trouxe à tona a necessidade de gerenciar eficientemente aplicações distribuídas. Kubernetes (K8s) tornou-se a plataforma líder em orquestração de contêineres, fornecendo automação e escalabilidade para aplicações distribuídas. Quando combinada com Java, uma linguagem robusta e madura, as soluções em Kubernetes podem ser adaptadas a diversas versões do JDK, especialmente as versões de Suporte de Longo Prazo (LTS): Java 8, 11 e 17.

Este artigo técnico abordará como configurar e implantar aplicações Java no Kubernetes, com exemplos de código otimizados para JDK 8, 11 e 17. Também apresentaremos boas práticas para garantir a eficiência e a escalabilidade de serviços Java em produção.

Compatibilidade de Java com Kubernetes

Cada versão do Java LTS traz avanços significativos, e é essencial entender como essas melhorias impactam a integração com Kubernetes. Para ilustrar isso, usaremos exemplos práticos que abrangem diferentes versões LTS, desde a tradicional Java 8 até a moderna e otimizada Java 17.

Recursos e Melhorias nas Versões LTS

- **Java 8:** Introduziu a API de Streams e melhorias na sintaxe de lambdas. É amplamente utilizado em aplicações legadas.
- **Java 11:** Trouxe melhorias na modularidade com o Java Platform Module System, novas APIs (como HttpClient) e uma abordagem mais ágil para aplicações em nuvem.
- **Java 17:** É a versão LTS mais recente, oferecendo desempenho otimizado, suporte para novos recursos como pattern matching e registro, além de melhorias para ambientes de contêineres.



Exemplo Técnico 1: Deploy de uma Aplicação Java 8 no Kubernetes

Primeiro, um exemplo de uma aplicação Spring Boot simples com Java 8 que será empacotada e executada em Kubernetes.

Código Java (Compatível com Java 8)

java

```
package com.fastcode.k8s;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class FastCodeK8sApplication {

    public static void main(String[] args) {
        SpringApplication.run(FastCodeK8sApplication.class, args);
    }

    @RestController
    class KubernetesController {
        @GetMapping("/status")
        public String status() {
            return "Aplicação Java 8 no Kubernetes - Executando!";
        }
    }
}
```

Dockerfile para Java 8

dockerfile

```
# Usando imagem base com JDK 8
FROM openjdk:8-jdk-alpine
VOLUME /tmp
COPY target/fastcode-k8s-0.0.1-SNAPSHOT.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Manifestos Kubernetes

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fastcode-k8s-app-java8
  labels:
    app: fastcode-k8s-app-java8
spec:
```



```
replicas: 2
selector:
  matchLabels:
    app: fastcode-k8s-app-java8
template:
  metadata:
    labels:
      app: fastcode-k8s-app-java8
spec:
  containers:
    - name: fastcode-k8s-app-java8
      image: fastcode-k8s-app-java8:latest
      ports:
        - containerPort: 8080
      livenessProbe:
        httpGet:
          path: /status
          port: 8080
        initialDelaySeconds: 15
        periodSeconds: 20
      readinessProbe:
        httpGet:
          path: /status
          port: 8080
        initialDelaySeconds: 10
        periodSeconds: 10
```

Exemplo Técnico 2: Deploy de uma Aplicação Java 11 no Kubernetes

Agora, um exemplo de aplicação Spring Boot otimizada para Java 11, utilizando APIs modernas.

Código Java (Compatível com Java 11)

java

```
package com.fastcode.k8s;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class FastCodeK8sApplication {

    public static void main(String[] args) {
        SpringApplication.run(FastCodeK8sApplication.class, args);
    }

    @RestController
    class KubernetesController {
        @GetMapping("/status")
        public String status() {
            return "Aplicação Java 11 no Kubernetes - Executando!";
        }
    }
}
```



```
}  
}
```

Dockerfile para Java 11

dockerfile

```
# Usando imagem base com JDK 11  
FROM adoptopenjdk:11-jre-hotspot  
VOLUME /tmp  
COPY target/fastcode-k8s-0.0.1-SNAPSHOT.jar app.jar  
EXPOSE 8080  
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Manifestos Kubernetes

yaml

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: fastcode-k8s-app-java11  
  labels:  
    app: fastcode-k8s-app-java11  
spec:  
  replicas: 3  
  selector:  
    matchLabels:  
      app: fastcode-k8s-app-java11  
  template:  
    metadata:  
      labels:  
        app: fastcode-k8s-app-java11  
    spec:  
      containers:  
        - name: fastcode-k8s-app-java11  
          image: fastcode-k8s-app-java11:latest  
          ports:  
            - containerPort: 8080  
          livenessProbe:  
            httpGet:  
              path: /status  
              port: 8080  
            initialDelaySeconds: 15  
            periodSeconds: 20  
          readinessProbe:  
            httpGet:  
              path: /status  
              port: 8080  
            initialDelaySeconds: 10  
            periodSeconds: 10
```



Exemplo Técnico 3: Deploy de uma Aplicação Java 17 no Kubernetes

Por fim, uma aplicação Java 17, que traz otimizações significativas para ambientes de contêineres.

Código Java (Compatível com Java 17)

java

```
package com.fastcode.k8s;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class FastCodeK8sApplication {

    public static void main(String[] args) {
        SpringApplication.run(FastCodeK8sApplication.class, args);
    }

    @RestController
    class KubernetesController {
        @GetMapping("/status")
        public String status() {
            return "Aplicação Java 17 no Kubernetes - Executando!";
        }
    }
}
```

Dockerfile para Java 17

```
dockerfile

# Usando imagem base com JDK 17, ideal para produção
FROM eclipse-temurin:17-jdk-alpine
VOLUME /tmp
COPY target/fastcode-k8s-0.0.1-SNAPSHOT.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Manifestos Kubernetes

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fastcode-k8s-app-java17
  labels:
    app: fastcode-k8s-app-java17
spec:
```



```
replicas: 3
selector:
  matchLabels:
    app: fastcode-k8s-app-java17
template:
  metadata:
    labels:
      app: fastcode-k8s-app-java17
  spec:
    containers:
      - name: fastcode-k8s-app-java17
        image: fastcode-k8s-app-java17:latest
        ports:
          - containerPort: 8080
        livenessProbe:
          httpGet:
            path: /status
            port: 8080
          initialDelaySeconds: 15
          periodSeconds: 20
        readinessProbe:
          httpGet:
            path: /status
            port: 8080
          initialDelaySeconds: 10
          periodSeconds: 10
```

Boas Práticas Operacionais em Kubernetes para Aplicações Java

1. **Configuração Externa com ConfigMaps e Secrets:** Utilize ConfigMaps e Secrets para gerenciar configurações e credenciais de forma segura e flexível, sem embutir essas informações no código-fonte.
2. **Imagens Docker Otimizadas:** Utilize imagens leves, como as baseadas em alpine, e evite incluir bibliotecas ou ferramentas desnecessárias.
3. **Probes de Saúde:** Implemente livenessProbe e readinessProbe para garantir que apenas pods saudáveis recebam tráfego. Esses probes ajudam a detectar e corrigir falhas automaticamente.
4. **Escalonamento Automático com HPA:** Utilize o Horizontal Pod Autoscaler (HPA) para ajustar o número de réplicas automaticamente com base em métricas de utilização de CPU ou memória.
5. **Monitoramento com Prometheus e Grafana:** Integre Prometheus e Grafana para monitoramento



Conclusão

A integração entre Java e Kubernetes oferece uma plataforma poderosa e escalável para o desenvolvimento e implantação de aplicações modernas, especialmente aproveitando os recursos de longa duração das versões LTS (Java 8, 11 e 17). Cada versão do JDK traz melhorias significativas que podem ser otimizadas para ambientes de contêineres, proporcionando benefícios em termos de modularidade, desempenho e eficiência de recursos. Ao utilizar boas práticas como otimização de imagens Docker, configuração externa com ConfigMaps e Secrets, implementação de probes de saúde e escalonamento automático, as aplicações Java podem se beneficiar do ecossistema Kubernetes para garantir resiliência, alta disponibilidade e facilidade de manutenção em produção.

O uso adequado de técnicas como HPA, Prometheus e Grafana complementa esse ambiente, fornecendo insights detalhados e automação inteligente para gerenciar aplicações de maneira eficaz. Portanto, a escolha da versão do JDK deve ser alinhada às necessidades do projeto, levando em consideração a maturidade da base de código, os requisitos de desempenho e o suporte de longo prazo necessário, para maximizar a eficiência operacional e a qualidade do serviço.

EducaCiência FastCode – Construindo um futuro sólido para a tecnologia, com práticas de desenvolvimento que fortalecem a comunidade e a qualidade de software!