# **通过 Java 后端调用 OpenShift API 获取运行中代码版本**

## **1. 引言**

### **1.1. 目的**

本报告旨在为 Java 开发人员提供一份技术指南，阐述如何通过 Java 后端应用程序，利用 OpenShift API 以编程方式检索在不同 OpenShift 环境中运行的应用程序部署所对应的容器镜像版本（标签或摘要）。

### **1.2. 背景**

在 Kubernetes 及 OpenShift 生态系统中，部署中实际运行的“代码版本”与 Pod 规范中引用的具体容器镜像标签（tag）或摘要（digest）直接关联 1。理解这一点至关重要。OpenShift 中的 Deployment（或 DeploymentConfig）负责管理 Pod 的生命周期，确保指定数量的 Pod 副本按照预定义的配置运行，而 Pod 规范则定义了其中运行的容器及其使用的镜像 2。因此，要获取运行时的代码版本，实质上就是要获取相关 Pod 中容器所使用的镜像标识符。

### **1.3. 重要性**

以编程方式获取运行中的镜像版本具有多种实际应用价值。例如，可以实现自动化部署验证，确保线上环境运行的是预期版本；可以进行环境审计，追踪不同环境部署的一致性；可以构建自定义的监控仪表盘，展示各服务的版本信息；还可以将版本信息与安全扫描结果关联，进行漏洞管理。

### **1.4. 方法**

本报告将遵循以下结构进行阐述：首先，介绍与版本信息相关的关键 OpenShift/Kubernetes 对象；其次，比较并推荐合适的 Java 客户端库；接着，详细说明认证机制及实现；然后，提供使用 Java 实现查询逻辑的具体步骤和代码示例；之后，明确所需的 RBAC（基于角色的访问控制）权限；最后，讨论错误处理和最佳实践。

## **2. 理解用于版本信息的 OpenShift/Kubernetes 对象**

要从 OpenShift 环境中准确获取运行的代码版本，首先需要理解涉及定义和运行容器化应用程序的关键对象 2。

### **2.1. Deployments (apps/v1)**

* **角色:** Deployment 是 Kubernetes 中用于管理无状态应用的标准资源。它通过维护一组相同的 Pod 副本，确保应用程序达到期望的状态 2。
* **结构重点:** 获取镜像信息的路径嵌套在 Deployment 结构中：Deployment -> spec -> template (PodTemplateSpec) -> spec (PodSpec) -> containers (容器列表 List<Container>) -> image (镜像名称字符串)。这个路径是定位实际运行镜像的关键 4。
* **API 参考:** 可以通过 Kubernetes API 端点 GET /apis/apps/v1/namespaces/{namespace}/deployments 列出指定命名空间下的所有 Deployment 5。

### **2.2. Pods (v1)**

* **角色:** Pod 是 Kubernetes 中可创建和管理的最小部署单元，包含一个或多个紧密耦合的容器，共享存储和网络资源 2。通常，Pod 由 Deployment 或 DeploymentConfig 等控制器管理。
* **结构重点:** 在 Pod 对象内部，获取镜像信息的路径为：Pod -> spec (PodSpec) -> containers (容器列表 List<Container>) -> image (镜像名称字符串) 4。
* **API 参考:** 可以通过 Kubernetes API 端点 GET /api/v1/namespaces/{namespace}/pods 列出指定命名空间下的所有 Pod。

### **2.3. OpenShift DeploymentConfigs (apps.openshift.io/v1)**

* **角色:** DeploymentConfig 是 OpenShift 特有的资源，作为 Deployment 的替代方案，提供了额外的功能，例如基于 ImageStream 变更或配置变更的触发器，以及更复杂的部署策略。
* **结构重点:** 获取镜像信息的路径与 Deployment 类似：DeploymentConfig -> spec -> template (PodTemplateSpec) -> spec (PodSpec) -> containers -> image。
* **环境兼容性考量:** 虽然 Kubernetes Deployment 是标准，但在许多 OpenShift 环境中，特别是遗留系统或利用了 OpenShift 特定触发器功能的应用，仍然广泛使用 DeploymentConfig。这是因为 DeploymentConfig 在 Kubernetes Deployment 功能成熟之前就已经存在于 OpenShift 中。因此，一个健壮的、旨在查询 OpenShift 环境的 Java 后端解决方案，应该能够处理这两种资源类型，或者至少允许配置目标资源类型。这增加了实现的复杂性，但确保了在不同 OpenShift 项目中的广泛适用性。

### **2.4. OpenShift ImageStreams (image.openshift.io/v1)**

* **角色:** ImageStream 是 OpenShift 的一个核心概念，用于追踪镜像标签随时间的变化及其对应的摘要（digest）8。它通常与 DeploymentConfig 结合使用，当 ImageStream 中关注的标签指向新的镜像时，可以自动触发新的部署。
* **相关性:** Pod 规范中定义了*当前运行*的镜像。查询与 DeploymentConfig 关联的 ImageStream 可以提供历史版本信息，或者了解某个标签（如 latest）当前*意图指向*的版本 8。然而，要确定 Pod *实际正在运行*的版本，最可靠的方法是直接检查 Pod 的规范。
* **准确性考量:** 直接查询 Deployment 或 DeploymentConfig 管理的 Pod 的规范是获取*实际运行*版本的最可靠方法，正如一些脚本化解决方案所采用的策略 9。仅仅依赖 ImageStream 的标签（尤其是像 :latest 这样的可变标签 1）可能会产生误导。例如，如果一个 DeploymentConfig 引用了 myapp:latest，但新的镜像推送后触发的部署尚未完成、失败或被暂停，或者 Pod 的 imagePullPolicy 不是 Always 1，那么实际运行的 Pod 可能仍然使用旧镜像的摘要。Pod 规范反映了运行时的真实情况，而 ImageStream 的标签则代表了一个（可能是过时的）意图或指针。

### **2.5. 镜像标识：标签 (Tag) vs. 摘要 (Digest)**

* **标签 (Tags):** 标签是人类可读的、指向特定镜像版本的可变指针，例如 v1.2.0 或 latest 1。在生产环境中使用可变标签（如 :latest）存在风险，因为它们可能指向不同的镜像内容，导致版本追踪困难和回滚复杂化 1。
* **摘要 (Digests):** 摘要是基于镜像内容的、不可变的标识符（通常是 SHA256 哈希值），例如 sha256:1ff6... 1。摘要唯一地标识了一个特定的镜像层配置。
* **最佳实践:** 强烈建议在部署规范中使用并记录完整的镜像字符串，该字符串可能同时包含标签和摘要（例如 myimage:v1.2.0@sha256:...）。如果同时指定了标签和摘要，Kubernetes 在拉取镜像时会优先使用摘要 1。依赖摘要可以确保部署的确定性和可重复性，避免因标签被移动而导致的版本混乱 1。
* **信息提取:** Java 代码需要从 Pod 规范的 container.image 字段中获取完整的字符串。这个字符串本身就包含了最精确的版本信息。如果应用程序后续需要单独使用标签或摘要，可以对该字符串进行解析。但首要任务是获取并存储完整的镜像标识符，因为它保留了最精确的信息，允许应用程序基于可用的最精确标识符做出决策。

## **3. 选择 Java 客户端库**

直接使用 REST API 与 Kubernetes 或 OpenShift 交互虽然可行，但通常不推荐，因为需要手动处理认证、请求构建、响应解析以及对象序列化/反序列化等复杂任务。使用客户端库可以极大地简化这些工作 11。

### **3.1. 主要选项**

对于 Java 开发者，主要有两个流行的 Kubernetes 客户端库：

* **官方 Kubernetes Java 客户端 (kubernetes-client/java):** 由 Kubernetes API Machinery 特别兴趣小组（SIG）维护 11。
* **Fabric8 Kubernetes 客户端 (fabric8io/kubernetes-client):** 一个历史悠久、广泛使用的社区客户端，以其流式 DSL（领域特定语言）和丰富的扩展而闻名 11。

### **3.2. 比较**

| **功能** | **Fabric8 Kubernetes 客户端** | **官方 Kubernetes Java 客户端** |
| --- | --- | --- |
| **DSL** | 提供流式（Fluent）、链式的 DSL，代码可读性较高 14 | 更直接地映射 Kubernetes API，可能显得更冗长 16 |
| **OpenShift 支持** | 内建 OpenShift 扩展和特定的 OpenShiftClient 接口，更好地处理 OpenShift 特有资源和认证 7 | 提供通用的 Kubernetes 支持，处理 OpenShift 特有类型可能需要额外工作 |
| **可扩展性** | 支持多种扩展，如 Tekton, Knative, Istio 等 7 | 主要关注核心 Kubernetes API |
| **自定义资源 (CRDs)** | 提供良好的 CRD 支持，通常无需额外依赖或代码生成 11 | 支持 CRD，但可能需要代码生成或更复杂的设置 |
| **维护状态** | 社区驱动，kubernetes-client 子项目仍然活跃（见下文澄清），尽管更广泛的 Fabric8 平台已停止 7 | 由 Kubernetes SIG 官方维护 |
| **流行度/使用情况** | 非常流行，被 Quarkus 20, Spring Cloud 21, Apache Camel 17 等项目使用 | 作为官方标准，采用率正在增长 |

关于 Fabric8 状态的澄清:

需要特别注意，虽然有资料提到“fabric8 已停止” 22，但这指的是最初的、更广泛的 Fabric8 微服务集成平台 22。fabric8io/kubernetes-client 这个具体的 Java 客户端库本身仍然是活跃的，由社区持续维护和开发，并且被广泛使用和记录 7。Eclipse JKube 是一个相关的项目，但它似乎与客户端库本身是不同的实体 22。这种表面上的不连续性与实际的活跃状态之间的差异，凸显了在选择技术栈时进行深入调研的重要性。开发者不应仅凭顶层公告就否定一个选项，而应关注具体组件（如客户端库）的近期提交历史、发布频率和社区支持情况。来自 Red Hat 开发者门户 14、Quarkus 集成 20 以及活跃的 Stack Overflow 讨论 19 都证明了该客户端库的持续相关性。本报告必须提供这种细致的澄清，以防用户基于不完整的信息放弃一个可行的选项。

### **3.3. 推荐**

考虑到用户查询的目标是 OpenShift 环境，并且希望通过 Java 后端进行交互，**推荐使用 Fabric8 Kubernetes 客户端**。主要原因在于其流式 DSL 提高了代码的可读性 16，并且它提供了对 OpenShift 对象和认证机制的显式支持 7，这与用户的需求高度契合。后续的代码示例将基于 Fabric8 客户端。

## **4. Java 客户端认证**

与 OpenShift/Kubernetes API 的任何交互都需要经过认证，以确认发起请求的客户端（即 Java 应用程序）的身份，并为其后续操作进行授权 3。

### **4.1. 认证策略概述**

对于 Java 客户端，常见的认证方法包括：

* **Kubeconfig 文件:** 客户端读取用户主目录下的 ~/.kube/config 文件或 KUBECONFIG 环境变量指定的路径下的文件。该文件包含了集群访问端点、用户凭证（如 token、客户端证书等）以及上下文信息 18。这是在集群外部（如开发环境）运行应用程序或工具时的理想选择。
* **服务账户令牌 (Service Account Token):** 当应用程序作为 Pod 在 OpenShift/Kubernetes 集群内部署运行时，Kubernetes 会自动将该 Pod 关联的服务账户的令牌挂载到 Pod 文件系统中的预定位置 (/var/run/secrets/kubernetes.io/serviceaccount/token) 7。这是集群内部应用访问 API 的标准且推荐的方式。
* **OAuth/Bearer 令牌:** 应用程序直接向客户端库提供一个有效的 Bearer 令牌。该令牌可以通过 oc login、oc whoami -t 31 或标准的 OAuth 流程从 OpenShift OAuth 服务器获取 24。适用于特定的集成场景，或在集群外部运行且没有 Kubeconfig 文件时使用 19。
* **用户名/密码:** 这种方式对于应用程序直接访问 API 来说不太常见，通常用于交互式登录（如 Web 控制台或 oc login），登录成功后会获取到一个令牌 24。Fabric8 客户端支持此方式，但对于程序化访问，其安全性通常低于基于令牌的方式 32。
* **客户端证书:** 需要管理证书文件，相比令牌认证，在典型的应用开发中较少使用 3。

### **4.2. Fabric8 客户端的认证处理**

Fabric8 客户端的设计旨在简化认证过程：

* **自动配置检测:** 默认情况下，Fabric8 客户端会按照特定优先级顺序尝试自动加载配置：系统属性 -> 环境变量 -> Kubeconfig 文件 -> 服务账户令牌和挂载的 CA 证书 18。
* **便利性:** 在许多常见场景下，只需简单地调用 new KubernetesClientBuilder().build() 即可成功初始化客户端，无论是在本地使用 Kubeconfig 文件，还是在集群内部使用服务账户令牌 7。
* **显式配置:** 对于需要更精细控制的场景，可以通过 ConfigBuilder 类来显式配置客户端，例如指定 Master URL、提供令牌、设置特定的 Kubeconfig 文件路径等 7。

### **4.3. Fabric8 客户端认证方法总结**

| **方法** | **描述/用例** | **Fabric8 配置 (ConfigBuilder / 自动检测)** | **安全考量** |
| --- | --- | --- | --- |
| **Kubeconfig 文件** | 本地开发，集群外运行的工具 | 自动检测 (~/.kube/config 或 KUBECONFIG 环境变量)，或 Config.fromKubeconfig(), withKubeconfigFile() | 依赖于 Kubeconfig 文件本身的安全存储和权限 |
| **集群内服务账户** | **推荐**：在 OpenShift/Kubernetes Pod 内部运行的应用程序 | 自动检测 (/var/run/secrets/kubernetes.io/serviceaccount/) 18 | 依赖于为服务账户配置的 RBAC 权限（应遵循最小权限原则） |
| **显式 Bearer/OAuth 令牌** | 特定集成，无 Kubeconfig 的集群外访问，需要预先获取令牌 | new ConfigBuilder().withOauthToken(token).withMasterUrl(url) | 令牌需要安全地存储和传输，通常具有时效性 |
| **用户名/密码** | 主要用于交互式登录，**不推荐**用于应用程序直接访问 API | new ConfigBuilder().withUsername(user).withPassword(pass) | 凭证硬编码或存储风险高，安全性较低 |
| **客户端证书** | 需要管理证书和密钥文件，对于普通应用开发不如令牌方便 | new ConfigBuilder().withClientCertFile(path).withClientKeyFile(path) | 需要安全地管理证书文件和私钥 |

### **4.4. 认证代码示例 (Fabric8)**

Java

import io.fabric8.kubernetes.client.Config;  
import io.fabric8.kubernetes.client.ConfigBuilder;  
import io.fabric8.kubernetes.client.KubernetesClient;  
import io.fabric8.kubernetes.client.KubernetesClientBuilder;  
import io.fabric8.kubernetes.client.KubernetesClientException;  
import java.io.FileNotFoundException;  
import java.io.IOException;  
  
public class AuthenticationExamples {  
  
 public static void main(String args) {  
  
 // 1. 默认配置 (尝试 Kubeconfig 或 集群内服务账户)  
 try (KubernetesClient client = new KubernetesClientBuilder().build()) {  
 System.out.println("成功使用默认配置初始化客户端: " + client.getMasterUrl());  
 } catch (KubernetesClientException e) {  
 System.err.println("默认配置初始化失败: " + e.getMessage());  
 }  
  
 // 2. 指定 Kubeconfig 文件路径  
 String kubeconfigPath = "/path/to/your/kubeconfig"; // 替换为你的 Kubeconfig 文件路径  
 try {  
 // 注意：Config.fromKubeconfig() 在较新版本中可能不直接可用或推荐  
 // 使用 ConfigBuilder 更为通用  
 Config configFromFile = new ConfigBuilder().withKubeconfigFile(kubeconfigPath).build();  
 try (KubernetesClient client = new KubernetesClientBuilder().withConfig(configFromFile).build()) {  
 System.out.println("成功使用指定 Kubeconfig 文件初始化客户端: " + client.getMasterUrl());  
 }  
 } catch (KubernetesClientException e) {  
 System.err.println("使用 Kubeconfig 文件 (" + kubeconfigPath + ") 初始化失败: " + e.getMessage());  
 } catch (Exception e) { // 处理可能的其他文件读取异常  
 System.err.println("加载 Kubeconfig 文件时出错: " + e.getMessage());  
 }  
  
  
 // 3. 使用显式 Bearer Token  
 String masterUrl = "https://your-openshift-api-server:6443"; // 替换为你的 API Server URL  
 String token = "your\_bearer\_token\_here"; // 替换为你的 Token  
 try {  
 Config configWithToken = new ConfigBuilder()  
 .withMasterUrl(masterUrl)  
 .withOauthToken(token)  
 // 可能需要禁用其他自动配置源，或配置信任证书等  
 //.withTrustCerts(true) // 仅用于测试环境！  
 .build();  
 try (KubernetesClient client = new KubernetesClientBuilder().withConfig(configWithToken).build()) {  
 System.out.println("成功使用 Bearer Token 初始化客户端: " + client.getMasterUrl());  
 }  
 } catch (KubernetesClientException e) {  
 System.err.println("使用 Bearer Token 初始化失败: " + e.getMessage());  
 }  
 }  
}

### **4.5. 安全性与可预测性**

Fabric8 的自动配置功能 7 虽然方便，但也意味着应用程序的权限会根据其运行环境（本地开发者的 Kubeconfig 或 Pod 的服务账户）动态变化。这种隐式行为可能导致在不同环境中的行为不一致。例如，一个在本地使用管理员 Kubeconfig 测试通过的应用，部署到集群后可能会因为 Pod 的默认服务账户权限不足而失败。反之，意外地以过高权限运行也是一个安全风险。因此，对于生产环境或需要严格控制权限的场景，推荐采用**显式配置**。即使自动检测可能有效，也应考虑为应用程序创建专用的、具有最小必要权限的服务账户，并在 Java 代码中显式配置客户端使用该服务账户的令牌。这提供了更好的可预测性和安全性，确保应用程序始终以预期的、受限的权限运行。

## **5. 实现：使用 Fabric8 Java 客户端查询 OpenShift**

本节将详细介绍使用 Fabric8 Java 客户端实现查询 OpenShift 中 Deployment 或 DeploymentConfig 的容器镜像版本的具体步骤和代码。

### **5.1. 环境设置**

* **依赖管理:** 首先，需要在 Java 项目中添加 Fabric8 Kubernetes 客户端的依赖。
  + **Maven (pom.xml):**  
    XML  
    <dependency>  
     <groupId>io.fabric8</groupId>  
     <artifactId>kubernetes-client</artifactId>  
     <version>6.10.0</version>  
    </dependency>  
    *(请根据 Fabric8 官方文档检查并使用最新的兼容版本)* 14
  + **Gradle (build.gradle):**  
    Gradle  
    dependencies {  
     implementation 'io.fabric8:kubernetes-client:6.10.0' // 检查最新版本  
     // implementation 'io.fabric8:openshift-client:6.10.0' // 如果需要  
    }  
    通常情况下，kubernetes-client 依赖已经包含了处理核心 Kubernetes 资源（如 Deployment, Pod）以及通过通用 API 处理 CRD 的能力，对于本报告的目标（获取 Deployment/Pod 中的镜像信息）可能已经足够 7。如果需要直接操作 DeploymentConfig 等 OpenShift 特定类型，添加 openshift-client 会更方便。

### **5.2. 客户端初始化**

在代码中实例化 KubernetesClient。强烈建议使用 try-with-resources 语句来管理客户端实例，以确保在使用完毕后自动关闭客户端并释放资源 7。

Java

import io.fabric8.kubernetes.client.KubernetesClient;  
import io.fabric8.kubernetes.client.KubernetesClientBuilder;  
import io.fabric8.kubernetes.client.KubernetesClientException;  
  
//...  
  
try (KubernetesClient client = new KubernetesClientBuilder().build()) {  
 // 在这里执行与 Kubernetes/OpenShift API 的交互  
 System.out.println("Kubernetes 客户端已初始化，连接到: " + client.getMasterUrl());  
 //... 后续代码...  
} catch (KubernetesClientException e) {  
 System.err.println("初始化 Kubernetes 客户端时出错: " + e.getMessage());  
 // 处理异常，例如记录日志  
}

这里使用了默认构造函数，它会尝试自动配置认证（如前文所述）。可以根据需要替换为显式配置的方式。

### **5.3. 查询 Deployments 或 DeploymentConfigs**

确定目标命名空间，然后使用客户端对象来查询资源。

Java

String namespace = "your-target-namespace"; // 替换为目标命名空间  
  
try (KubernetesClient client = new KubernetesClientBuilder().build()) {  
  
 // 查询 Deployments  
 System.out.println("在命名空间 '" + namespace + "' 中查询 Deployments...");  
 client.apps().deployments().inNamespace(namespace).list().getItems().forEach(deployment -> {  
 System.out.println("找到 Deployment: " + deployment.getMetadata().getName());  
 extractImageInfoFromDeployment(deployment); // 调用下面定义的辅助方法  
 });  
  
 // 查询 DeploymentConfigs (如果需要)  
 // 需要确保添加了 openshift-client 依赖，或者使用通用 API  
 // 方式一：使用 OpenShiftClient (需要 openshift-client 依赖)  
 /\*  
 System.out.println("\n在命名空间 '" + namespace + "' 中查询 DeploymentConfigs...");  
 try {  
 client.adapt(OpenShiftClient.class).deploymentConfigs().inNamespace(namespace).list().getItems().forEach(dc -> {  
 System.out.println("找到 DeploymentConfig: " + dc.getMetadata().getName());  
 extractImageInfoFromDeploymentConfig(dc); // 类似的辅助方法  
 });  
 } catch (Exception adaptException) {  
 System.err.println("无法适配到 OpenShiftClient，请确保添加了 openshift-client 依赖: " + adaptException.getMessage());  
 }  
 \*/  
 // 方式二：使用通用 API (不需要 openshift-client 依赖，但类型处理稍繁琐)  
 /\*  
 client.genericKubernetesResources("apps.openshift.io/v1", "DeploymentConfig")  
 .inNamespace(namespace)  
 .list()  
 .getItems()  
 .forEach(genericResource -> {  
 System.out.println("找到 DeploymentConfig (通用 API): " + genericResource.getMetadata().getName());  
 // 需要手动将 genericResource 转换为 DeploymentConfig 或提取字段  
 });  
 \*/  
  
  
} catch (KubernetesClientException e) {  
 System.err.println("查询资源时出错: " + e.getMessage());  
}

代码中展示了如何列出指定命名空间下的所有 Deployment 7。同时注释掉了查询 DeploymentConfig 的两种方式：一种是适配到 OpenShiftClient（需要额外依赖）18，另一种是使用通用 API。根据实际需要选择。

### **5.4. 提取镜像信息**

获取到 Deployment 或 DeploymentConfig 对象后，需要深入其结构以找到容器镜像信息。

Java

import io.fabric8.kubernetes.api.model.Container;  
import io.fabric8.kubernetes.api.model.PodSpec;  
import io.fabric8.kubernetes.api.model.PodTemplateSpec;  
import io.fabric8.kubernetes.api.model.apps.Deployment;  
// 如果使用 OpenShiftClient，可能需要导入: import io.fabric8.openshift.api.model.DeploymentConfig;  
import java.util.List;  
import java.util.Optional;  
  
// 辅助方法：从 Deployment 中提取镜像信息  
private static void extractImageInfoFromDeployment(Deployment deployment) {  
 Optional<PodTemplateSpec> templateSpec = Optional.ofNullable(deployment)  
 .map(Deployment::getSpec)  
 .map(ds -> ds.getTemplate()); // Java 8 Optional 链式调用避免 NullPointerException  
  
 if (templateSpec.isPresent()) {  
 Optional<PodSpec> podSpec = Optional.ofNullable(templateSpec.get().getSpec());  
 podSpec.ifPresent(spec -> {  
 // 处理常规容器  
 List<Container> containers = spec.getContainers();  
 if (containers!= null &&!containers.isEmpty()) {  
 System.out.println(" 常规容器镜像:");  
 containers.forEach(container -> {  
 System.out.println(" - 容器名: " + container.getName() + ", 镜像: " + container.getImage());  
 parseImageString(container.getImage()); // 可选：解析标签和摘要  
 });  
 } else {  
 System.out.println(" 未找到常规容器定义。");  
 }  
  
 // 处理 Init 容器 (同样重要，它们也运行代码)  
 List<Container> initContainers = spec.getInitContainers();  
 if (initContainers!= null &&!initContainers.isEmpty()) {  
 System.out.println(" Init 容器镜像:");  
 initContainers.forEach(initContainer -> {  
 System.out.println(" - 容器名: " + initContainer.getName() + ", 镜像: " + initContainer.getImage());  
 parseImageString(initContainer.getImage()); // 可选：解析  
 });  
 }  
 });  
 } else {  
 System.out.println(" 无法找到 Deployment 的 Pod 模板。");  
 }  
}  
  
// 类似地，可以为 DeploymentConfig 创建一个 extractImageInfoFromDeploymentConfig 方法  
  
// 可选的辅助方法：解析镜像字符串以区分标签和摘要  
private static void parseImageString(String image) {  
 if (image == null |  
| image.isEmpty()) return;  
  
 String imageBase = image;  
 String tag = "latest"; // 默认标签  
 String digest = null;  
  
 // 检查并分离摘要  
 int digestSeparator = image.lastIndexOf('@');  
 if (digestSeparator > 0) {  
 imageBase = image.substring(0, digestSeparator);  
 digest = image.substring(digestSeparator + 1);  
 }  
  
 // 检查并分离标签  
 int tagSeparator = imageBase.lastIndexOf(':');  
 // 确保 ':' 不在 '/' 之后（例如 registry.com:5000/image），并且不是摘要分隔符之前的部分  
 if (tagSeparator > 0 && imageBase.substring(tagSeparator + 1).indexOf('/') == -1) {  
 tag = imageBase.substring(tagSeparator + 1);  
 imageBase = imageBase.substring(0, tagSeparator);  
 } else if (digest == null) {  
 // 如果没有摘要且没有显式标签，则假定为 latest  
 tag = "latest";  
 } else {  
 // 如果有摘要但没有显式标签  
 tag = "<none>"; // 或保持为 null  
 }  
  
 System.out.println(" (解析: 基础镜像=" + imageBase + ", 标签=" + tag + ", 摘要=" + (digest!= null? digest : "无") + ")");  
}

这段代码演示了如何安全地导航 Deployment 结构以获取 PodSpec 4。关键在于，它不仅处理了 spec.containers（常规容器），还处理了 spec.initContainers（初始化容器）2。这很重要，因为初始化容器也运行代码，并且它们的版本信息对于全面的环境审计同样关键。简单地只看第一个容器或只看常规容器是不够的。代码还包含了一个可选的 parseImageString 方法，用于从完整的镜像字符串中分离出基础镜像名、标签和摘要。

### **5.5. 完整代码示例**

Java

import io.fabric8.kubernetes.api.model.Container;  
import io.fabric8.kubernetes.api.model.PodSpec;  
import io.fabric8.kubernetes.api.model.PodTemplateSpec;  
import io.fabric8.kubernetes.api.model.apps.Deployment;  
import io.fabric8.kubernetes.client.KubernetesClient;  
import io.fabric8.kubernetes.client.KubernetesClientBuilder;  
import io.fabric8.kubernetes.client.KubernetesClientException;  
// import io.fabric8.openshift.client.OpenShiftClient; // 如果需要查询 DeploymentConfig  
  
import java.util.List;  
import java.util.Optional;  
  
public class GetOpenShiftImageVersions {  
  
 public static void main(String args) {  
 // 目标命名空间，可以从配置或环境变量读取  
 String namespace = System.getenv().getOrDefault("TARGET\_NAMESPACE", "default");  
  
 System.out.println("尝试连接到 Kubernetes/OpenShift 集群...");  
  
 try (KubernetesClient client = new KubernetesClientBuilder().build()) {  
 System.out.println("连接成功! Master URL: " + client.getMasterUrl());  
 System.out.println("将在命名空间 '" + namespace + "' 中查询...");  
  
 // 查询 Deployments  
 System.out.println("\n--- 查询 Deployments ---");  
 List<Deployment> deployments = client.apps().deployments().inNamespace(namespace).list().getItems();  
 if (deployments.isEmpty()) {  
 System.out.println("在命名空间 '" + namespace + "' 中未找到 Deployments。");  
 } else {  
 deployments.forEach(deployment -> {  
 System.out.println("\nDeployment: " + deployment.getMetadata().getName());  
 extractImageInfoFromDeployment(deployment);  
 });  
 }  
  
 // 可选：查询 DeploymentConfigs (取消注释并确保依赖存在)  
 /\*  
 System.out.println("\n--- 查询 DeploymentConfigs ---");  
 try {  
 List<io.fabric8.openshift.api.model.DeploymentConfig> dcs = client.adapt(OpenShiftClient.class)  
 .deploymentConfigs().inNamespace(namespace).list().getItems();  
 if (dcs.isEmpty()) {  
 System.out.println("在命名空间 '" + namespace + "' 中未找到 DeploymentConfigs。");  
 } else {  
 dcs.forEach(dc -> {  
 System.out.println("\nDeploymentConfig: " + dc.getMetadata().getName());  
 // extractImageInfoFromDeploymentConfig(dc); // 实现类似的提取方法  
 });  
 }  
 } catch (Exception adaptException) {  
 System.err.println("无法查询 DeploymentConfigs，请检查 OpenShiftClient 适配和依赖: " + adaptException.getMessage());  
 }  
 \*/  
  
 } catch (KubernetesClientException e) {  
 System.err.println("\n与 Kubernetes API 交互时出错:");  
 System.err.println(" 状态码: " + e.getCode());  
 System.err.println(" 消息: " + e.getMessage());  
 // 在实际应用中，这里应该进行更详细的日志记录或错误处理  
 e.printStackTrace();  
 } catch (Exception e) {  
 System.err.println("\n发生意外错误: " + e.getMessage());  
 e.printStackTrace();  
 }  
 }  
  
 // 辅助方法：从 Deployment 中提取镜像信息  
 private static void extractImageInfoFromDeployment(Deployment deployment) {  
 Optional<PodTemplateSpec> templateSpec = Optional.ofNullable(deployment)  
 .map(Deployment::getSpec)  
 .map(ds -> ds.getTemplate());  
  
 if (templateSpec.isPresent()) {  
 Optional<PodSpec> podSpec = Optional.ofNullable(templateSpec.get().getSpec());  
 if (podSpec.isPresent()) {  
 PodSpec spec = podSpec.get();  
 // 处理常规容器  
 List<Container> containers = spec.getContainers();  
 if (containers!= null &&!containers.isEmpty()) {  
 System.out.println(" 常规容器镜像:");  
 containers.forEach(container -> {  
 System.out.println(" - 容器名: " + container.getName() + ", 镜像: " + container.getImage());  
 parseImageString(container.getImage());  
 });  
 } else {  
 System.out.println(" 未找到常规容器定义。");  
 }  
  
 // 处理 Init 容器  
 List<Container> initContainers = spec.getInitContainers();  
 if (initContainers!= null &&!initContainers.isEmpty()) {  
 System.out.println(" Init 容器镜像:");  
 initContainers.forEach(initContainer -> {  
 System.out.println(" - 容器名: " + initContainer.getName() + ", 镜像: " + initContainer.getImage());  
 parseImageString(initContainer.getImage());  
 });  
 }  
 } else {  
 System.out.println(" Pod 模板中未找到 PodSpec。");  
 }  
 } else {  
 System.out.println(" 无法找到 Deployment 的 Pod 模板。");  
 }  
 }  
  
 // 可选的辅助方法：解析镜像字符串  
 private static void parseImageString(String image) {  
 if (image == null |  
| image.isEmpty()) return;  
 String imageBase = image, tag = "latest", digest = null;  
 int digestSeparator = image.lastIndexOf('@');  
 if (digestSeparator > 0) {  
 imageBase = image.substring(0, digestSeparator);  
 digest = image.substring(digestSeparator + 1);  
 }  
 int tagSeparator = imageBase.lastIndexOf(':');  
 if (tagSeparator > 0 && imageBase.substring(tagSeparator + 1).indexOf('/') == -1) {  
 tag = imageBase.substring(tagSeparator + 1);  
 imageBase = imageBase.substring(0, tagSeparator);  
 } else if (digest == null) tag = "latest"; else tag = "<none>";  
 System.out.println(" (解析: 基础=" + imageBase + ", 标签=" + tag + ", 摘要=" + (digest!= null? digest : "无") + ")");  
 }  
}

这个完整的示例展示了如何初始化客户端、查询特定命名空间中的 Deployment、遍历找到的 Deployment、提取常规容器和 Init 容器的名称和镜像字符串，并包含基本的错误处理和资源关闭（通过 try-with-resources）。

## **6. 所需的 RBAC 权限**

仅仅能够通过认证（确认身份）并不足以让 Java 客户端成功查询所需信息。客户端所使用的身份（通常是服务账户）还必须被 Kubernetes/OpenShift 的 RBAC（基于角色的访问控制）系统*授权*执行相应的操作（读取 Deployments 和 Pods）24。认证回答“你是谁？”，授权回答“你能做什么？”。

### **6.1. 最小权限原则**

在配置 RBAC 权限时，应始终遵循最小权限原则，即仅授予执行任务所必需的最少权限，以降低潜在的安全风险。

### **6.2. 所需权限**

为了让 Java 客户端能够获取指定命名空间中 Deployment（或 DeploymentConfig）及其 Pod 的容器镜像信息，服务账户需要以下权限：

* **API 组 (apiGroups):**
  + "apps": 用于访问 Deployment 资源。
  + "apps.openshift.io": 如果需要访问 DeploymentConfig 资源。
  + "" (空字符串): 表示核心 API 组，用于访问 Pod 资源。
* **资源 (resources):**
  + "deployments"
  + "deploymentconfigs" (如果需要)
  + "pods"
* **动词 (verbs):**
  + "get": 获取单个资源的详细信息。
  + "list": 列出指定类型的所有资源。 *(如果需要持续监控变化，可能还需要 "watch" 权限，但对于一次性查询，”get” 和 “list” 已足够)*

### **6.3. 实现方式**

权限通过在目标命名空间中创建 Role 和 RoleBinding 对象来授予：

* **Role:** 定义一组权限（允许对哪些资源的哪些 API 组执行哪些动词）。
  + **示例 Role YAML (pod-deployment-reader-role.yaml):**  
    YAML  
    apiVersion: rbac.authorization.k8s.io/v1  
    kind: Role  
    metadata:  
     # 必须在需要访问的目标命名空间中创建此 Role  
     namespace: your-target-namespace # <--- 替换为目标命名空间  
     name: pod-deployment-reader  
    rules:  
    - apiGroups: ["apps"] # 允许访问 Deployment  
     resources: ["deployments"]  
     verbs: ["get", "list"]  
    - apiGroups: [""] # 允许访问 Pod (核心 API 组)  
     resources: ["pods"]  
     verbs: ["get", "list"]  
    # 如果需要访问 DeploymentConfig，添加以下规则：  
    # - apiGroups: ["apps.openshift.io"]  
    # resources: ["deploymentconfigs"]  
    # verbs: ["get", "list"]  
    34
* **RoleBinding:** 将一个 Role 授予一个或多个主体（用户、组或服务账户）。
  + **示例 RoleBinding YAML (pod-deployment-reader-binding.yaml):**  
    YAML  
    apiVersion: rbac.authorization.k8s.io/v1  
    kind: RoleBinding  
    metadata:  
     # 必须在与 Role 相同的命名空间中创建此 RoleBinding  
     namespace: your-target-namespace # <--- 替换为目标命名空间  
     name: pod-deployment-reader-binding  
    subjects:  
    - kind: ServiceAccount  
     name: your-java-app-service-account # <--- 替换为 Java 应用使用的服务账户名  
     # 服务账户也属于这个命名空间  
     namespace: your-target-namespace # <--- 替换为目标命名空间  
    roleRef:  
     kind: Role # 引用的是 Role (命名空间级别)  
     name: pod-deployment-reader # <--- 引用上面创建的 Role 名称  
     apiGroup: rbac.authorization.k8s.io  
    34

将这两个 YAML 文件应用到目标 OpenShift/Kubernetes 集群（例如，使用 oc apply -f <filename> 或 kubectl apply -f <filename>），即可为指定的服务账户授予在特定命名空间中读取 Deployment 和 Pod 信息的权限。

### **6.4. 权限范围考量**

需要注意的是，Role 和 RoleBinding 都是**命名空间范围**的对象 34。这意味着上述配置仅授予服务账户在 your-target-namespace 这个命名空间内的权限。如果 Java 应用程序需要查询**多个**命名空间，则必须在**每一个**目标命名空间中都创建相应的 Role 和 RoleBinding。另一种选择是使用 ClusterRole 和 ClusterRoleBinding，它们授予的是集群范围的权限。虽然 ClusterRole 可以简化对多命名空间访问的管理，但它也显著扩大了权限范围，增加了服务账户令牌泄露时的潜在风险。因此，除非确实需要集群范围的访问权限并且充分理解其影响，否则**强烈建议优先使用命名空间级别的 Role 和 RoleBinding**，以更好地遵循最小权限原则。

## **7. 错误处理和最佳实践**

与任何网络服务交互时，健壮的错误处理都至关重要。与 OpenShift/Kubernetes API 通信也不例外。

### **7.1. 处理 KubernetesClientException**

Fabric8 客户端在与 API 服务器交互过程中遇到问题时，通常会抛出 KubernetesClientException 或其子类 14。应用程序代码应该准备好捕获并处理这些异常。

Java

try (KubernetesClient client = new KubernetesClientBuilder().build()) {  
 //... API 调用代码...  
} catch (KubernetesClientException e) {  
 System.err.println("与 Kubernetes API 交互时发生错误:");  
 System.err.println(" HTTP 状态码: " + e.getCode()); // 获取 HTTP 状态码  
 System.err.println(" 错误消息: " + e.getMessage());  
 // 根据状态码或错误消息进行具体处理  
 if (e.getCode() == 401) {  
 System.err.println(" 认证失败，请检查 Token 或 Kubeconfig 配置。");  
 } else if (e.getCode() == 403) {  
 System.err.println(" 授权失败，请检查服务账户的 RBAC 权限。"); // 参考第 6 节  
 } else if (e.getCode() == 404) {  
 System.err.println(" 资源未找到。");  
 // 可能需要优雅地处理资源不存在的情况  
 } else if (e.getCode() >= 500) {  
 System.err.println(" API 服务器内部错误或暂时不可用，考虑重试。");  
 // 实现重试逻辑 (见下文)  
 } else {  
 // 其他错误，记录详细信息  
 e.printStackTrace(); // 在生产环境中应使用日志框架记录  
 }  
}

14

* **分析错误:** 检查异常的 getCode()（HTTP 状态码）和 getMessage() 对于理解失败原因至关重要。
* **常见错误及处理:**
  + **401 Unauthorized:** 认证信息无效或缺失。检查客户端初始化和凭证配置（参见第 4 节）。
  + **403 Forbidden:** 身份已认证，但 RBAC 策略禁止该操作。检查服务账户的 Role 和 RoleBinding 配置（参见第 6 节）34。
  + **404 Not Found:** 请求的资源（如命名空间、Deployment）不存在。应用程序应能优雅处理这种情况。
  + **400 Bad Request:** 客户端发送的请求无效（例如，格式错误）。检查 API 调用参数。
  + **5xx Server Errors:** API 服务器端发生错误或暂时过载。这些通常是瞬时错误，可以考虑重试。
* **日志记录:** 记录详细的错误信息（包括堆栈跟踪）对于调试至关重要 14。
* **重试逻辑:** 对于 5xx 错误或网络超时等瞬时问题，可以实现重试机制，例如采用指数退避策略，避免立即重试导致问题恶化 14。可以使用现成的库（如 Resilience4j, Guava Retrying）或手动实现。

### **7.2. 资源管理**

始终使用 try-with-resources 语句来管理 KubernetesClient 实例。这可以确保无论代码是正常执行还是抛出异常，客户端的网络连接和其他资源都能被正确关闭和释放 14。

### **7.3. 镜像标识符最佳实践**

再次强调，为了获得最准确、最不可变的运行版本标识，应优先使用从 container.image 字符串中提取的镜像**摘要**（@sha256:...）1。标签虽然可读性好，但其指向的内容可能改变。

### **7.4. API 速率限制**

需要注意，频繁地轮询 OpenShift/Kubernetes API（尤其是在大型集群或多租户环境中执行大量 list 操作）可能会触发 API 服务器的速率限制（Rate Limiting）机制，导致请求被拒绝（通常返回 HTTP 429 Too Many Requests 错误）。虽然速率限制的具体细节可能因集群配置而异，但这是一个普遍存在的保护机制 23。

* **影响:** 如果 Java 应用程序过于频繁地查询 API，可能会遇到 429 错误，导致功能中断。
* **应对策略:**
  + **合理安排轮询间隔:** 如果需要定期检查版本，请设置合理的、非过于频繁的轮询间隔。
  + **使用 Watch 机制:** 如果需要近乎实时地响应资源变化（例如，部署完成、镜像更新），应优先考虑使用 Kubernetes 的 watch 机制。Fabric8 客户端支持 Watch 操作 7，它比反复轮询 list 效率高得多，因为它允许服务器在资源发生变化时主动通知客户端。
  + **缓存:** 对于不经常变化的信息，可以在应用程序本地进行缓存，减少对 API 服务器的直接请求。

## **8. 结论**

### **8.1. 总结**

本报告详细阐述了如何通过 Java 后端应用程序调用 OpenShift API 来获取运行中容器的镜像版本。关键步骤包括：

1. **理解核心对象:** 识别 Deployment、Pod（以及 OpenShift 中的 DeploymentConfig）是承载运行版本信息的关键资源。
2. **选择客户端库:** 推荐使用 Fabric8 Kubernetes 客户端，因其流式 DSL 和对 OpenShift 的良好支持。
3. **处理认证:** 根据运行环境（集群内或集群外）选择合适的认证方式，对于集群内应用，优先使用自动挂载的服务账户令牌。
4. **配置授权:** 通过创建命名空间级别的 Role 和 RoleBinding，为服务账户授予读取所需资源的最小权限。
5. **实现查询逻辑:** 使用 Fabric8 客户端 API 查询 Deployment/Pod，并从容器规范中提取完整的 image 字符串。
6. **错误处理与最佳实践:** 实现健壮的错误处理，正确管理客户端资源，并注意 API 速率限制。

### **8.2. 核心要点**

最核心的要点是，Deployment 或 DeploymentConfig 中的 Pod 规范是确定*当前实际运行*的容器镜像版本的权威来源。为了获得最精确和不可变的标识，应关注并优先使用从 container.image 字符串中提取的镜像**摘要**（digest）。

### **8.3. 后续考虑**

基于本报告提供的方法，可以进一步探索相关主题：

* **实时监控:** 利用 Kubernetes Watch 机制实现对部署版本变化的实时监控，而非周期性轮询。
* **CI/CD 集成:** 将此版本检查逻辑集成到持续集成/持续部署（CI/CD）流程中，作为部署后验证步骤。
* **多环境管理:** 通过外部化配置（如环境变量、配置文件）来管理不同 OpenShift 环境（开发、测试、生产）的 API 端点和认证信息。
* **扩展查询:** 查询其他相关对象以获取更丰富的上下文信息，例如 Pod 的状态、事件记录或关联的 Service 信息。

#### Works cited

1. Images - Kubernetes, accessed May 3, 2025, <https://kubernetes.io/docs/concepts/containers/images/>
2. Pods - Kubernetes, accessed May 3, 2025, <https://kubernetes.io/docs/concepts/workloads/pods/>
3. Using the OpenShift REST API | Red Hat Product Documentation, accessed May 3, 2025, <https://docs.redhat.com/en/documentation/openshift_container_platform/3.5/html-single/using_the_openshift_rest_api/index>
4. How to retrieve a specific Kubernetes deployment image tag using ..., accessed May 3, 2025, <https://stackoverflow.com/questions/72239591/how-to-retrieve-a-specific-kubernetes-deployment-image-tag-using-java-api>
5. Using the OpenShift API is there a way to get deployments in a project - Stack Overflow, accessed May 3, 2025, <https://stackoverflow.com/questions/70040777/using-the-openshift-api-is-there-a-way-to-get-deployments-in-a-project>
6. List All Container Images Running in a Cluster - Kubernetes, accessed May 3, 2025, <https://kubernetes.io/docs/tasks/access-application-cluster/list-all-running-container-images/>
7. kubernetes-client/doc/CHEATSHEET.md at main · fabric8io ... - GitHub, accessed May 3, 2025, <https://github.com/fabric8io/kubernetes-client/blob/main/doc/CHEATSHEET.md>
8. Image APIs | API reference | OpenShift Container Platform 4.9 - Red Hat Documentation, accessed May 3, 2025, <https://docs.openshift.com/container-platform/4.9/rest_api/image_apis/image-apis-index.html>
9. How do I find out the application version that is currently deployed in OpenShift as a container image? - Stack Overflow, accessed May 3, 2025, <https://stackoverflow.com/questions/71265087/how-do-i-find-out-the-application-version-that-is-currently-deployed-in-openshif>
10. Using container image digests in Kubernetes manifests - Google Cloud, accessed May 3, 2025, <https://cloud.google.com/kubernetes-engine/docs/tutorials/using-container-image-digests-in-kubernetes-manifests>
11. What is the Kubernetes Java client? - Red Hat, accessed May 3, 2025, <https://www.redhat.com/en/topics/cloud-computing/what-is-kubernetes-java-client>
12. Client Libraries | Kubernetes, accessed May 3, 2025, <https://kubernetes.io/docs/reference/using-api/client-libraries/>
13. rohanKanojia/fabric8-official-java-client-comparison - GitHub, accessed May 3, 2025, <https://github.com/rohanKanojia/fabric8-official-java-client-comparison>
14. How to use Fabric8 Java Client with Kubernetes | Red Hat Developer, accessed May 3, 2025, <https://developers.redhat.com/articles/2023/01/04/how-use-fabric8-java-client-kubernetes>
15. Kubernetes Client for Java: Fabric8 introduction - Marc Nuri, accessed May 3, 2025, <https://blog.marcnuri.com/kubernetes-client-java-fabric8-introduction>
16. Fabric8 Kubernetes Java Client - ncona.com, accessed May 3, 2025, <https://ncona.com/2022/09/fabric8-kubernetes-java-client>
17. Getting started with the fabric8 Kubernetes Java client | Red Hat ..., accessed May 3, 2025, <https://developers.redhat.com/blog/2020/05/20/getting-started-with-the-fabric8-kubernetes-java-client>
18. kubernetes-client/README.md at main · fabric8io/kubernetes-client ..., accessed May 3, 2025, <https://github.com/fabric8io/kubernetes-client/blob/master/README.md>
19. How to get kubernetes service account access token using fabric8 java client?, accessed May 3, 2025, <https://stackoverflow.com/questions/46046871/how-to-get-kubernetes-service-account-access-token-using-fabric8-java-client>
20. Kubernetes Client - Quarkus, accessed May 3, 2025, <https://quarkus.io/guides/kubernetes-client>
21. Spring Cloud Kubernetes vs. Kubernetes API Client - Stack Overflow, accessed May 3, 2025, <https://stackoverflow.com/questions/68500321/spring-cloud-kubernetes-vs-kubernetes-api-client>
22. fabric8: open source Integrated Development Platform for Kubernetes, accessed May 3, 2025, <https://fabric8.io/>
23. Client Libraries for Kubernetes - Go client, Python, Fabric8, JKube & Java Operator SDK, accessed May 3, 2025, <https://nubenetes.com/kubernetes-client-libraries/>
24. Understanding authentication | Authentication and authorization | OpenShift Container Platform 4.12, accessed May 3, 2025, <https://docs.openshift.com/container-platform/4.12/authentication/understanding-authentication.html>
25. Authentication - Additional Concepts | Architecture | OpenShift Container Platform 3.11, accessed May 3, 2025, <https://docs.openshift.com/container-platform/3.11/architecture/additional_concepts/authentication.html>
26. Overview | REST API Reference | OKD 3.7, accessed May 3, 2025, <https://docs.okd.io/3.7/rest_api/index.html>
27. Authenticating | Kubernetes, accessed May 3, 2025, <https://kubernetes.io/docs/reference/access-authn-authz/authentication/>
28. Access Clusters Using the Kubernetes API, accessed May 3, 2025, <https://kubernetes.io/docs/tasks/administer-cluster/access-cluster-api/>
29. Create a Token-based Kubeconfig - Knowledge Base - Leaseweb, accessed May 3, 2025, <https://kb.leaseweb.com/kb/users-roles-and-permissions-on-kubernetes-rbac/kubernetes-users-roles-and-permissions-on-kubernetes-rbac-create-a-token-based-kubeconfig/>
30. Adding a Service Account Authentication Token to a Kubeconfig File - Oracle Help Center, accessed May 3, 2025, <https://docs.oracle.com/en-us/iaas/Content/ContEng/Tasks/contengaddingserviceaccttoken.htm>
31. How to get Openshift session token using rest api calls - Stack Overflow, accessed May 3, 2025, <https://stackoverflow.com/questions/49501133/how-to-get-openshift-session-token-using-rest-api-calls>
32. How do I authenticate and authorize when accessing Kubernetes API using the Fabric8 library? What are the common authentication methods to choose from? - yifan-online.com, accessed May 3, 2025, <https://yifan-online.com/en/km/article/detail/16316>
33. How to Discover OpenShift by REST API - Universal Discovery Content Pack, accessed May 3, 2025, <https://docs.microfocus.com/doc/407/2020.11/openshiftdscvr>
34. Using RBAC Authorization | Kubernetes, accessed May 3, 2025, <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>