GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Smart Injection of Environment Variables in Kubernetes Pod

Pranay Bhatia

MatrNr: 17935037

Supervisor: Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science

March 24, 2024

Abstract

Automating Kubernetes Pod Restarts with Dynamic Environment Variable Injection using HashiCorp Vault

In the realm of Kubernetes orchestration, automating the dynamic injection of environment variables into running pods has been a longstanding challenge. This research introduces a novel solution that leverages HashiCorp Vault and a Kubernetes liveness probe to facilitate automated restarts of pods upon changes to environment variables.

Traditional solutions for injecting secrets into pods at startup lack the capability to handle dynamic updates without manual intervention. This research addresses the difficulty of achieving automated restarts when environment variables within a pod need to be modified, presenting a comprehensive solution for seamless integration.

Current industry-standard solutions, such as HashiCorp Vault, excel at secret injection during pod initialization. However, they fall short in automating pod restarts when environment variables change, requiring manual restarts or reliance on DevOps processes. This limitation hinders the efficiency and agility of managing dynamic configurations.

Our innovative approach introduces a liveness probe within Kubernetes, ensuring constant monitoring of the environment variable path within the pod. If changes are detected, the liveness probe communicates with HashiCorp Vault, retrieves the updated variables, and triggers a controlled restart of the main container. This self-contained solution eliminates the need for manual interventions and provides a seamless mechanism for automated updates.

The proposed solution has been successfully implemented, demonstrating its effectiveness in automating pod restarts upon changes to environment variables. The evaluation revealed that the solution operates with minimal overhead, ensuring a reliable and efficient mechanism for maintaining up-to-date configurations within a dynamic Kubernetes environment.

In summary, this research introduces an automated solution to a prevalent challenge in Kubernetes orchestration, offering a streamlined process for dynamically updating environment variables and triggering pod restarts. The outcome demonstrates the potential for increased efficiency and autonomy in managing containerized applications, paving the way for more responsive and adaptive Kubernetes deployments.

Contents

1 Introduction

In modern containerized environments orchestrated by Kubernetes, managing environment variables efficiently and securely is essential for ensuring the reliability and security of applications. Environment variables play a crucial role in configuring and customizing containerized applications, providing a flexible and dynamic way to pass configuration information to running containers. However, manually updating environment variables in Kubernetes deployments can be cumbersome and error-prone, particularly in dynamic environments where configuration changes are frequent.

The aim of this report is to propose a novel approach for smartly injecting environment variables into Kubernetes clusters and automating the process of detecting changes and restarting pods accordingly. By leveraging Kubernetes' liveness probe mechanism—a built-in feature for determining the health of containerized applications—we can create a robust and automated solution for managing environment variables seamlessly.

Our approach involves encapsulating the logic for retrieving, validating, and updating environment variables within a liveness probe script, which is embedded within the container's configuration. This script continuously monitors the availability and integrity of environment variables, fetching them from a secure and centralized source, such as HashiCorp Vault. Upon detecting changes or discrepancies in the environment variables, the liveness probe triggers a pod restart, ensuring that the application remains up-to-date with the latest configuration changes.

By automating the injection and updating of environment variables in Kubernetes deployments, our approach offers several benefits, including:

- Improved Reliability: Ensuring that applications always have access to the correct and up-to-date environment variables enhances the reliability and stability of Kubernetes deployments.
- Enhanced Security: By fetching environment variables from a centralized and secure source, such as HashiCorp Vault, we can enforce access controls, encryption, and audit trails, thereby enhancing the security posture of the application. [Vauc]
- Efficient Configuration Management: Automating the process of injecting and updating environment variables simplifies configuration management tasks, reducing the risk of human error and streamlining the deployment process. [**vaulwt-k8s**]

The report begins with an introduction, setting the stage for discussing the proposed solution. It then addresses the problem statement, highlighting the challenges associated with manual environment variable management. Following this, the background section provides insights into HashiCorp Vault and Kubernetes' liveness probe mechanism, crucial components of the solution. Moving on to the architecture, the report details the workings of the proposed approach, including Kubernetes and Vault integration, liveness probe implementation, and deployment restart mechanism. The results section discusses the integration process and identifies potential risks. Finally, the conclusion synthesizes the key findings and outlines future directions. Additionally, the report includes code samples in the appendix for reference.

2 Problem Statement

In Kubernetes deployments, there can be enormous variables which need to be set before container deployment. This task is carried out by dev-ops colleagues. Manual tweaking and adjustment of variables are required to get a container up and running. This can lead to errors, inconsistencies, and downtime, particularly in dynamic environments where configuration changes are frequent. The problem statement revolves around the need for a robust and automated solution to intelligently inject environment variables into Kubernetes clusters and automatically trigger pod restarts upon detecting changes or updates to the configuration.

Key challenges include:

1. **Manual Configuration Management:** Manually updating environment variables in infrastructure is time-consuming, error-prone, and can result in configuration drifts and inconsistencies across pods.
2. **Dynamic Environment:** In dynamic Kubernetes environments where pods scale up, scale down, or migrate across nodes, ensuring that all pods have access to the latest environment variables becomes challenging.
3. **Application Reliability:** Changes to environment variables, such as API keys or database credentials, may require pod restarts to take effect, impacting application availability and reliability if not handled efficiently.
4. **Scalability:** As Kubernetes deployments scale to accommodate varying workloads, the process of injecting and updating environment variables must scale accordingly to meet the demands of dynamic environments.

Addressing these challenges requires the development of an automated solution that integrates with Kubernetes deployments, intelligently manages environment variables, and ensures the reliability, security, and scalability of applications running in Kubernetes clusters.

3 Background

3.1 HashiCorp Vault

Introduction: HashiCorp Vault is a popular open-source tool for managing secrets and sensitive data in modern cloud-native environments. It provides a centralized platform for securely storing, accessing, and managing secrets such as API keys, passwords, certificates, and encryption keys. Vault offers a robust set of features and capabilities designed to address the challenges of secrets management in dynamic and distributed systems. [Vauc]

Use and Benefits: The primary use of HashiCorp Vault is to securely manage secrets and sensitive data in cloud-native applications and infrastructure. By centralizing the storage of secrets, Vault helps organizations improve security, compliance, and operational efficiency.

Thanks to Built-in APIs which allow developers and operators to interact with Vault programmatically. These APIs provide access to Vault's functionality, including the ability to retrieve, create, update, and delete secrets. [Vaua]

Key/Value Secrets Engine API is the API allows users to store and retrieve secrets as key/value pairs within Vault's hierarchical data store. This also provides metadata, which defines what is the current of secret of defiled in each Key/Value secret engine. [KV]

Additionally, Vault provide a User interface, which developers can use to add, modify, delete and update the secrets which are inside kubernetes container. [Vaub]

In summary, HashiCorp Vault is a powerful tool for managing variables in cloud-native environments. Its rich set of features and flexible APIs makes it an essential component for our architecture.

3.2 Liveness Probes

Functionality of Liveness Probe

What is Liveness Probe: In Kubernetes, a liveness probe is a mechanism used to determine the health of a containerized application running in a pod. It periodically checks the application's state to ensure that it is running as expected. The liveness probe performs this check by sending requests to a specified endpoint within the container and evaluating the response. If the application responds successfully, indicating that it is healthy, the liveness probe considers the container to be in a good state. However, if the application fails to respond or returns an error, indicating that it is unhealthy, the liveness probe takes action, such as restarting the container. [Kub] [Zah23]

Benefits: The use of liveness probes offers several benefits for managing containerized applications in Kubernetes deployments:

- **Improved Reliability:** By continuously monitoring the health of containerized applications, liveness probes help ensure that unhealthy containers are detected and replaced promptly, minimizing downtime and enhancing application reliability. [Oda22]
- **Automatic Recovery:** Liveness probes facilitate automatic recovery from application failures by restarting unhealthy containers, thereby maintaining the desired state of the application and preserving service availability. [Kub]
- **Scalability:** Liveness probes enable Kubernetes clusters to dynamically scale resources based on workload demands, as unhealthy containers are replaced with healthy ones, allowing the cluster to adapt to changing conditions efficiently. [Zah23]
- **Simplified Operations:** By automating the detection and recovery of unhealthy containers, liveness probes reduce the need for manual intervention and streamline operations, making it easier to manage large-scale Kubernetes deployments. [Zah23]

Parameters: Liveness probes in Kubernetes are configured using several parameters that define how the probe operates and when it should take action:

- **Initial Delay:** This parameter specifies the amount of time to wait after the container starts before performing the first liveness probe. It allows the application to initialize before being checked for health. [Kub]

- **Period:** The period parameter defines the frequency at which the liveness probe should be executed, indicating how often the application's health should be assessed.[Kub]
- **Timeout:** This parameter sets the maximum amount of time the probe should wait for a response from the application. If the application fails to respond within this timeframe, the probe considers it unhealthy.[Kub]
- **Failure Threshold:** Conversely, the failure threshold parameter defines the number of consecutive failed probe results that indicate the application is unhealthy and should be restarted.[Kub]

Leveraging Liveness Probe for Conditional Restart of Cluster: By leveraging the functionality of liveness probes, Kubernetes deployments can implement conditional restarts of clusters based on the health status of containerized applications. We will use liveness probe to verify if the secret version of a key/vault secret engine matches to current deployment version. If both version matched, we will classify pod as healthy. In case of mismatch pod should restart itself automatically and fetch latest version of secrets from the vault.

4 Architecture

To start, the code implementation can be accessed at `https://github.com/pranay-bh/helm-keycloak` for complete architectural setup.

4.1 Working

The above diagram illustrates the sequence of events within a Kubernetes environment, depicting the initialization and operation of containers, as well as interactions with external services such as Vault for secret management. On start, Kubernetes startup triggers the activation of the `MyAppInitContainer`, responsible for generating a `.env` file at `/tmp/.env`. Subsequently, the `.env` file is passed to the `MyAppMainContainer`, which initiates normal application functionality, such as content serving via Nginx. Concurrently, the liveness probe is activated by `MyAppMainContainer`. The probe then communicates with Vault to verify the availability of a specific path (`http://vault/secret/myapp`). Depending on the response received, different paths are followed: if the path is absent (`404`), Vault initializes with environment variables and creates a version file; if present (`200`), Vault compares the current version with the locally stored version. In case of a version mismatch, Vault updates the version and `.env` files, leading to the failure of the liveness probe, triggering a restart of `MyAppMainContainer`. Upon restart, the updated `.env` file is utilized, ensuring seamless operation with the new environment. (Figure 1).

4.2 Kubernetes and Vault Integration

The script provided facilitates the integration between Kubernetes and HashiCorp Vault. It begins by checking if the `VAULT_ENABLE` environment variable is set to "true". If enabled, it proceeds to fetch secrets from Vault using the specified token and address constructed based on environment variables (`VAULT_SERVICE_HOST` and `VAULT_SERVICE_PORT`).

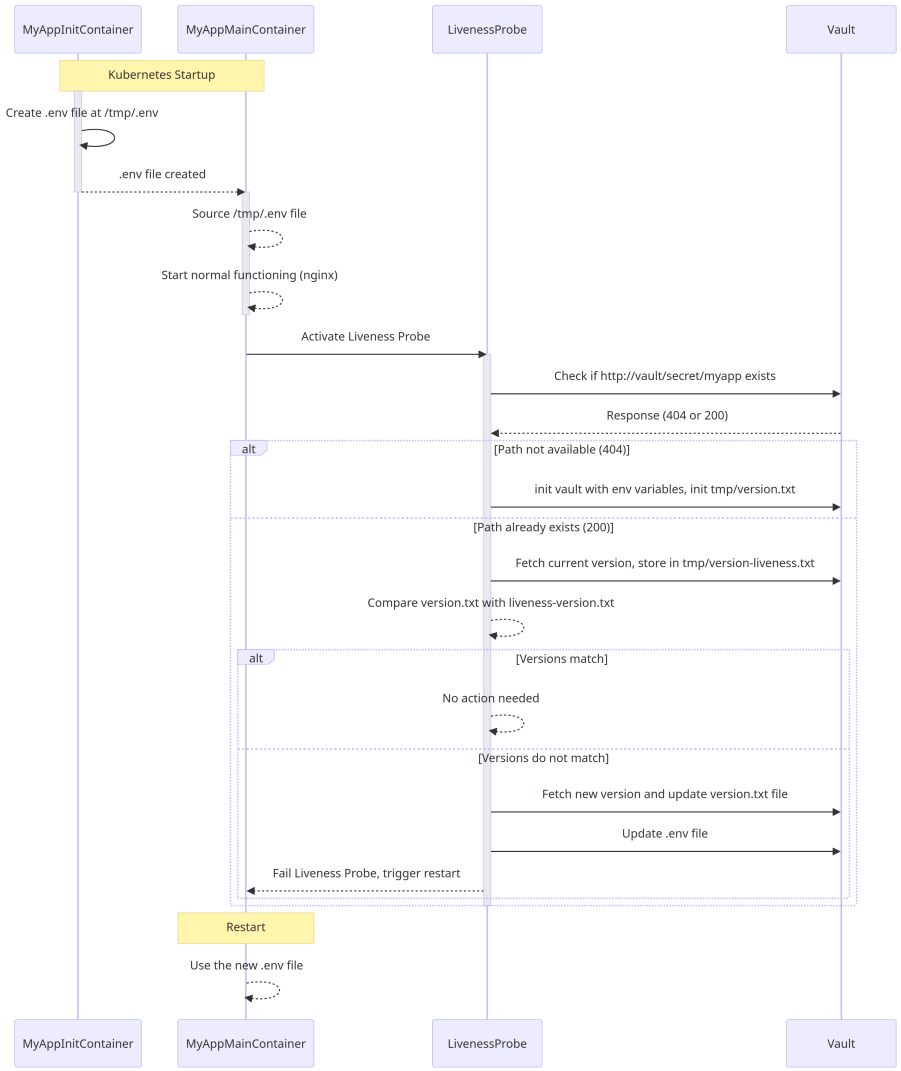

Figure 1: implementation Sequence diagram

4.3 Liveness Probe Implementation

The liveness probe implementation leverages Kubernetes' built-in feature to ensure the health of containerized applications [Kub]. Within the script, an HTTP request is made to the specified Vault address to check the availability of secrets. The response status code is evaluated, and actions are taken accordingly. If the secret store is missing (404), it initializes a new Key-Vault secret path. If the token is incorrect (403), it logs an error. Otherwise, it compares the version of the retrieved secrets with the locally stored version. If a mismatch is detected, it updates the environment variables and exports them for the application to use.

4.4 Deployment Restart Mechanism

The deployment restart mechanism is triggered conditionally based on the outcome of the liveness probe. If the script detects changes in the environment variables fetched from Vault, it exits with a non-zero status code (1), indicating that a restart is required. This triggers Kubernetes to restart the pod, ensuring that the application picks up the

updated environment variables. Conversely, if no changes are detected, the script exits with a zero status code (0), indicating that the environment variables are up to date, and no restart is necessary. This mechanism ensures that the application remains consistent to configuration changes and always uses the latest variables from Vault.

5 Result

The result addresses the challenges posed by manual configuration in Kubernetes deployments, offering a robust, UI Based automated solution to intelligently inject environment variables into Kubernetes clusters. The solution not only eliminates the need for manual tweaking and adjustment but also make developers to do the same task without having knowledge of deployments and infrastructure. It enhances reliabilit and scalability within dynamic environments where configuration changes are frequent. The solution includes a script packaged along with the application container image or as a ConfigMap in Kubernetes, ensuring availability within the container environment and execution as part of the pod lifecycle.

During pod initialization, Kubernetes mounts the script and executes it as a liveness probe. The script communicates with HashiCorp Vault using the specified address and token to fetch secrets as needed. Standard Unix commands such as `curl` and `jq` are utilized to interact with the Vault API and parse JSON responses. Upon detecting changes in retrieved secrets, the script restart the pods and exports new secrets as environment variables within the pod, ensuring the application has access to the latest configuration.

The result addresses the problem statement by providing an automated solution that intelligently manages environment variables, enhancing the reliability, security, and scalability of applications running in Kubernetes clusters. By automatically triggering pod restarts upon detecting changes or updates to the configuration, the solution minimizes errors, inconsistencies, and downtime, thereby optimizing the operational efficiency of Kubernetes deployments.

5.1 Integration

To start, the code implementation can be accessed at `https://github.com/pranay-bh/helm-keycloak` for basic deployment guideline. To integrate HashiCorp Vault into a generic deployment, follow the outlined procedure:

1. **Deploy HashiCorp Vault:** Begin by deploying HashiCorp Vault within your environment. This provides a UI to change environment variables.

2. **Configure Config Map:** Create a config map containing a shell script named `vault-integration.sh` from the GitHub repository mentioned architecture.

   - This script takes two parameters:
     - **Vault Token:** By default, set to 'admin'.
     - **Vault Path:** This path acts as the location in Vault where the variables will be stored. Adjust this path if needed.

   If Vault is running in a different cluster: Update the Vault Path to reach the URL and Vault token of the external Vault instance.

3. **Values.yaml File:** Ensure that the `values.yaml` file includes the following variables
   Vault_Token Vault_Path Vault_Enabled
4. **Update Deployment Container:**
   - Add two volume mounts:
     - For bash scripts (config and secret YAML file mounts are read-only after Kubernetes 1.19) (in previous script:, `/var/tmp`).
     - For writing the `.env` file and temporary files to compare Vault secret versions (in previous script:, `/tmp`).
   - Create an init container responsible for creating a new `.env` file. This file stores all variables required for the main container's startup. Its absence may cause the main container to fail to start.
   - Mount volumes and pass Vault-specific environment variables to the main container.
   - Create and execute a script from the liveness probe to interact with Vault.

5.2 Risks

The above mentioned deployment structure posses some potential risks. Firstly, Vault serves as the single source of failure within the system. Any downtime or service interruption of the Vault instance could disrupt critical operations reliant on its services.

Moreover, without proper backup mechanisms, such as persistent storage or integration with Consul for Vault, the loss of Vault data upon restart poses a significant risk. To mitigate this, implementing persistence storage with Vault or utilizing Consul for Vault management can enhance data reliability and availability.

Additionally, containerized applications must be configured to correctly source environment variables from the Vault setup. Misconfiguration can lead to applications failing to accessing updated environment variables.

Thus, while Vault integration offers numerous benefits, careful consideration and implementation of mitigating measures are essential to address these potential risks effectively.

6 Conclusion

In conclusion, the developed solution represents a significant advancement in addressing the challenges associated with manual configuration in Kubernetes deployments.The automation of environment configuration not only eliminates the need for manual tweaking and adjustment but also minimizes errors, inconsistencies, and downtime that may arise from manual intervention.

By automatically triggering pod restarts upon detecting changes or updates to the configuration, the solution optimizes operational efficiency and reduce Devops effort, ensuring that applications are always running with the latest configuration parameters. This proactive approach streamlines the deployment process and enhances the overall agility of Kubernetes environments.

In essence, the developed solution provides a foundation for reliable and scalable Kubernetes deployments, empowering organizations to effectively manage complex configurations while maintaining high levels of operational efficiency and automation.

References

[Kub] Kubernetes Probes. *Configure Liveness, Readiness and Startup Probes*. Kubernetes. URL: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/ (visited on 03/20/2024).

[KV] KV - Secrets Engines | Vault | HashiCorp Developer. *KV - Secrets Engines | Vault | HashiCorp Developer*. URL: https://developer.hashicorp.com/vault/docs/secrets/kv (visited on 03/20/2024).

[Oda22] Mehmet Odabasi. *Kubernetes Probes Explained with an Emphasis on Liveness Probe through Hands-on Experience*. Oct. 2022. URL: https://medium.com/@mehmetodabashi/kubernetes-probes-explained-with-an-emphasis-on-liveness-probe-through-hands-on-experience-71ac8192a64c (visited on 03/20/2024).

[Vaua] Vault | HashiCorp Developer. *HTTP API | Vault | HashiCorp Developer*. URL: https://developer.hashicorp.com/vault/api-docs (visited on 03/20/2024).

[Vaub] Vault | HashiCorp Developer. *Vault UI | Vault | HashiCorp Developer*. URL: https://developer.hashicorp.com/vault/tutorials/getting-started/getting-started-ui (visited on 03/20/2024).

[Vauc] Vault by HashiCorp. *Secrets Management*. URL: https://www.vaultproject.io/use-cases/secrets-management (visited on 03/17/2024).

[Zah23] Adam Zahorscak. *Guide to Understanding Your Kubernetes Liveness Probes Best Practices*. 2023. URL: https://www.fairwinds.com/blog/a-guide-to-understanding-kubernetes-liveness-probes-best-practices (visited on 03/20/2024).

A Code samples

```
if [ "$VAULT_ENABLE" = "true" ]; then

    VAULT_TOKEN="admin"
    VAULT_PATH="myapp"
    VAULT_ADDRESS="http://$VAULT_SERVICE_HOST:$VAULT_SERVICE_PORT/v1/secret/data/$VAULT_PATH"
    echo "address: " $VAULT_ADDRESS
    VAULT_HEADER="X-Vault-Token: $VAULT_TOKEN"

    echo "Installing necessary packages: curl and jq..."
    which apt-get && apt-get update && apt-get install -y curl jq
    curl
    jq

    HTTP_STATUS_CODE=$(curl -s --location "$VAULT_ADDRESS" --header "X-Vault-Token: $VAULT_TOKEN" -o /dev/null -w "%{http_code}")

    if [ "$HTTP_STATUS_CODE" = "000" ]; then
        echo "Service not found on given URL: $VAULT_ADDRESS"
        exit 0
    elif [ "$HTTP_STATUS_CODE" = "404" ]; then
        echo "Store does not exist. Initializing Vault store in $VAULT_PATH for the first time..."
        BODY=$(env | sort | jq -R -n 'reduce inputs as $line ({}; . + ($line | split("=") | {(.[0]): .[1]})) | {"data": .}')
        curl -s -H "$VAULT_HEADER" -H "Content-Type: application/json" -X POST -d "$BODY" "$VAULT_ADDRESS" -o /dev/null
        echo 1 > /tmp/version.txt
        exit 0
    elif [ "$HTTP_STATUS_CODE" = "403" ]; then
        echo "Incorrect token: $VAULT_TOKEN for $VAULT_ADDRESS"
        exit 0
    else
        echo "Status code: $HTTP_STATUS_CODE"
        echo "$(curl -s -H "$VAULT_HEADER" "$VAULT_ADDRESS" | jq -r '.data.metadata.version')" > /tmp/liveness-version.txt

        if cmp -s /tmp/version.txt /tmp/liveness-version.txt; then
            echo "Environment variable version $(cat /tmp/liveness-version.txt) is up to date"
            exit 0
        else
            echo "New environment variable version $(cat /tmp/liveness-version.txt) detected"
```

```
            JSON_DATA="$(curl -s -H "$VAULT_HEADER" "$VAULT_ADDRESS
               ↪ ")"
            echo "$( echo $JSON_DATA | jq -r '.data.metadata.
               ↪ version')" > /tmp/version.txt
            rm /tmp/.env || true
            echo "Exporting variables for $VAULT_PATH with version
               ↪ $(cat /tmp/version.txt)"
            keys=$(echo "$JSON_DATA" | jq -r '.data.data | keys[]')
            for key in $keys; do
                value=$(echo "$JSON_DATA" | jq -r ".data.data[\"
                   ↪ $key\"]")
                export "$key"="$value"
                # echo "Exporting variable: $key=$value"
                echo "$key=\"$value\"" >> /tmp/.env
            done
            exit 1
        fi
    fi
else
    echo "VAULT_ENABLE is not set to true. Doing nothing and moving
       ↪  on as expected."
fi
```

Listing 1: liveness probe bash script