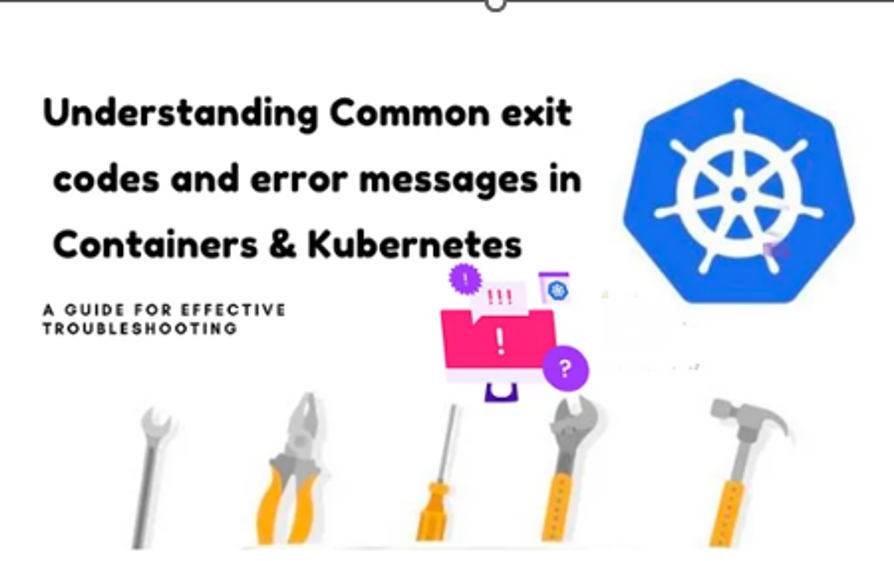
**Understanding Common exit codes and error messages in Containers & Kubernetes**



**Introduction:**

When working with containers and Kubernetes, understanding exit codes and common error messages is important for better troubleshooting and maintaining the health of your apps.

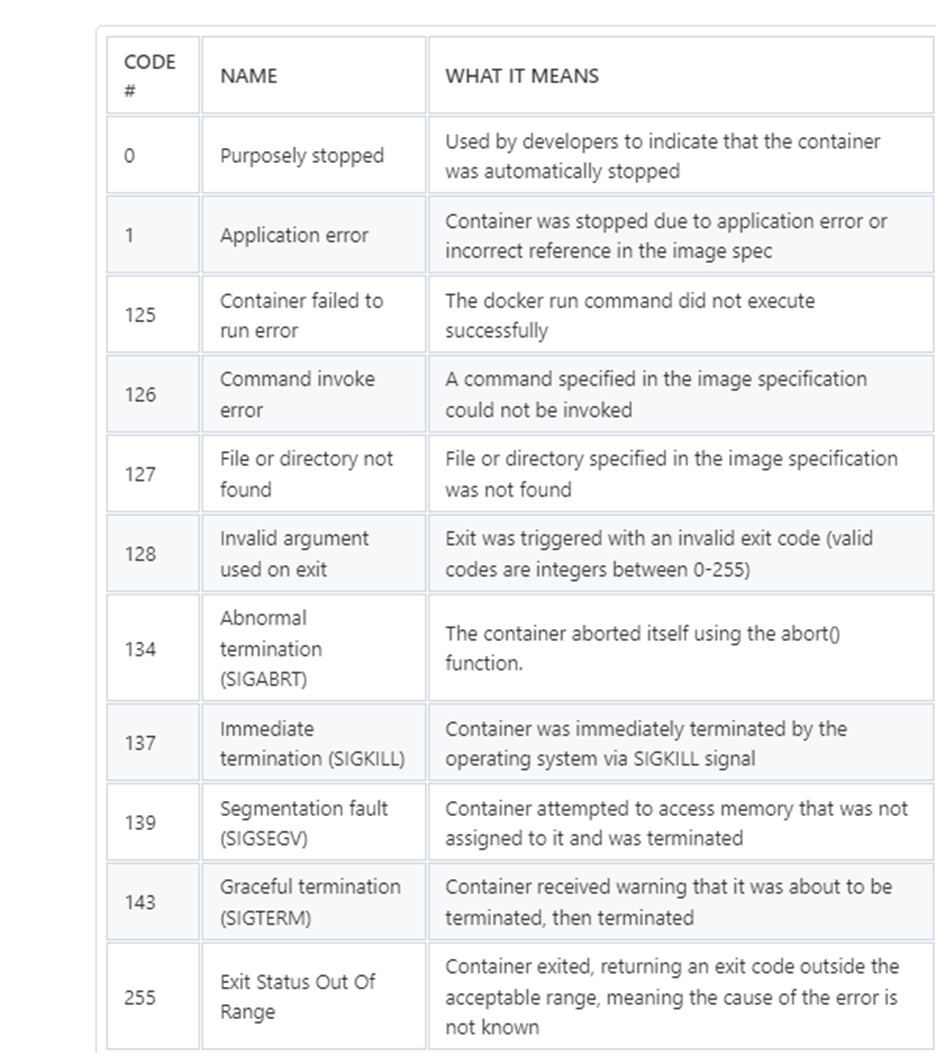
Exit codes are used by container engines to indicate the reasons for container termination, providing valuable insights into the root causes of pod failures.

In this guide, we will explore the significance of exit codes, and common error messages, and how to interpret them in the context of Kubernetes.

**Container exit codes:**

Container exit codes are used by container engines to indicate the reasons for container termination. When a container terminates, it reports why it was terminated through an exit code. Understanding these exit codes can help in diagnosing the root cause of pod failures.

The most common exit codes used by containers are:



Exit codes serve as a way to inform the user, operating system, and other applications about why the process was terminated. Each code is a number ranging from 0 to 255.

***Codes below 125 have application-specific meanings, while codes above 125 are reserved for system signals.***

Understanding these exit codes is essential for troubleshooting and resolving issues in Kubernetes clusters, nodes, containers, or pods. By identifying the exit code, one can take appropriate steps to diagnose and fix the underlying problems.

**Interpreting Common Container Exit Codes:**

1. ***Exit Code 0 (Purposefully Stopped):***

Exit code 0 denotes a deliberate termination of the container, often initiated by developers or automated processes. Technically, it signifies a clean exit without any errors. When a container receives this exit code, it implies that the foreground process has completed its task successfully or that an intentional stop signal was issued.

***2. Exit Code 1 (Application Error or Invalid Reference):***

Exit code 1 typically arises from application errors or misconfigurations within the container environment. This could include runtime exceptions, segmentation faults, or other critical failures encountered by the application process. Additionally, an invalid reference in the container’s specifications, such as an incorrect image name or missing dependencies, can trigger this exit code.

***3. Exit Code 125 (Command Execution Issue):***

Exit Code 125 indicates a failure in executing the command specified during container initialization. This failure might occur due to various reasons, including incorrect command syntax, insufficient permissions, or resource limitations such as memory or CPU constraints. Detailed examination of container logs and runtime environments is essential to pinpoint the root cause of this issue.

***4. Exit Code 126 (Command Invocation Issue):***

A container receiving Exit Code 126 indicates that the command specified in its execution environment could not be invoked successfully. This failure typically stems from missing dependencies or incompatible runtime environments required for command execution. Troubleshooting this issue involves examining the container’s environment variables, ensuring proper installation of dependencies, and verifying compatibility with the runtime environment.

***5. Exit Code 127 (Command Not Found):***

Exit Code 127 signals that a command referenced in the container’s specification is not found within the container’s filesystem. This could occur due to various reasons, such as a missing executable file, an incorrect command path, or a typo in the command name. Identifying and rectifying these discrepancies requires a thorough inspection of the container’s filesystem and environment configuration.

***6. Exit Code 128 (Invalid Argument to Exit):***

Exit Code 128 indicates a successful termination of the container process, typically after fulfilling its intended task. Unlike other exit codes that signify errors or failures, Code 128 denotes a graceful exit without encountering any exceptional conditions. It confirms that the container’s main process completed its execution without encountering errors or exceptions.

***7. Exit Codes 134, 137, 139, 143, 255 (Signal Terminations):***

These exit codes correspond to specific signals and their implications, such as out-of-memory conditions or system-generated termination signals.

For instance, Exit Code 137 denotes an immediate termination triggered by the operating system via the SIGKILL signal, often indicating resource exhaustion or critical system failures. Understanding the nuances of each signal termination code is crucial for diagnosing and mitigating underlying issues effectively.

**Common error messages and how to solve them:**

Here are some of the most common Kubernetes errors you are likely to encounter, and quick solutions to try first before you embark on more advanced troubleshooting.

***ImagePullBackOff***:

Occurs when Kubernetes is unable to pull the container image specified in the pod definition. This could be due to issues such as invalid image name, permission problems, or network connectivity issues.

*Example*:

The Kubernetes cluster fails to pull the Docker image from the specified registry due to authentication failure or network timeout.

*Troubleshooting*:

· Verify the image name and registry credentials specified in the pod definition.

· Check network connectivity between the Kubernetes cluster and the container registry.

· Ensure that the necessary permissions are set to pull the image from the registry.

Here is an example of how you could resolve an ImagePullBackOff error by checking the image pull policy and the image repository credentials:

Get the name of the pod with the ImagePullBackOff error:

$ kubectl get pods

Verify the image pull policy is set to “Always” or “IfNotPresent”:

$ kubectl describe pod [pod-name]

If the policy is set correctly, check if the image repository requires authentication.

If authentication is required, verify that you have the correct credentials.

If the image repository requires authentication, add the secrets to your Kubernetes cluster:

$ kubectl create secret docker-registry [secret-name] –docker-server=[repository-url] –docker-username=[username] –docker-password=[password]

Update the deployment file to use the newly created secret:

$ kubectl edit deployment [deployment-name]

In the deployment file, under the spec section, add the following line under the template section and imagePullSecrets:

– name: [secret-name]

Save the changes and reapply the deployment:

$ kubectl apply -f [deployment-file].yaml

**Crash Loop BackOff:**

This indicates that a pod is repeatedly crashing immediately after startup, triggering Kubernetes to back off and delay restarting the pod.

*Example:*

An application running in a container encounters a fatal error or a misconfiguration, causing the pod to crash and restart in a loop.

*Troubleshooting:*

· Check the pod resource requests and limits and adjust them if needed

· Verify that all required environment variables are set correctly

· Check the logs of the pod and the application for any errors or crash messages.

***NodeNotReady:***

This signifies that a node in the Kubernetes cluster is not ready to accept pods due to various reasons such as network issues, resource shortages, or node failures.

*Example:*

The node experiences network connectivity issues or hardware failures, causing it to become unresponsive.

*Troubleshooting:*

· Check the status of the node and review node conditions for any errors or warnings, using the kubectl describe node command.

· Check the logs of the relevant system daemons and processes to see if they indicate the cause of the failure.

· Investigate network connectivity between the node and the rest of the cluster.

· Address any resource shortages or hardware failures affecting the node’s readiness.

· Monitor the node’s system resource usage (e.g. memory, CPU) and increase the resources if necessary.

· If the node is undergoing maintenance or has failed, you may need to drain and evict the pods from the node and then repair or replace the node.

***Pod Pending:***

This indicates that the pod has been accepted by the Kubernetes system but is not yet running due to scheduling constraints or resource shortages.

*Example:*

The pod is awaiting the allocation of required resources, such as CPU or memory before it can be scheduled onto a node.

*Troubleshooting:*

· Check resource requests and limits specified in the pod definition.

· Ensure that there are available resources on the cluster nodes.

· Review scheduling constraints and node conditions to identify any issues affecting pod scheduling.

***PodCrashExitCode:***

It indicates that a pod has terminated due to an application error or an unexpected exit code.

*Example:*

The application running in the pod encounters a runtime exception or a critical error, causing the pod to crash and terminate.

*Troubleshooting:*

· Review container logs to identify the cause of the crash.

· Investigate application code and dependencies for any bugs or misconfigurations.

· Ensure that the container runtime environment is properly configured and that all required dependencies are available.

***Forbidden:***

This indicates that the user or service account does not have the necessary permissions to perform the requested operation.

*Example:*

A user attempts to create or modify a resource in the Kubernetes cluster, but their role or role binding does not grant them the required permissions.

*Troubleshooting:*

· Review role-based access control (RBAC) policies and ensure that the user or service account has the appropriate roles and role bindings assigned.

· Check for any restrictions or permissions conflicts that may be preventing the operation.

These are just a few examples of common error messages in Kubernetes, along with their definitions and troubleshooting steps. Understanding these errors and their implications is essential for effectively managing and troubleshooting Kubernetes clusters.

**Conclusion:**

In conclusion, Kubernetes, despite its advanced capabilities, is susceptible to errors such as *ImagePullBackOff*, *CrashLoopBackOff*, *Exit Codes*, and *Node NotReady*. To address these errors, understanding their root causes is essential.

Familiarizing yourself with these errors and their resolution steps is important. Troubleshooting involves identifying the root cause, evaluating available information, and taking appropriate steps to fix the problem.

This may include updating configurations, restarting failed pods, or addressing network connectivity issues. By employing best practices, log analysis, and automated tools, administrators can effectively pinpoint and resolve issues, ensuring a reliable and high-performing Kubernetes environment.