Technical Artifacts considerations

A guide to make correct use of Kubernetes artifacts

Author: Liran Cohen

licohen@redhat.com

TABLE OF CONTENTS

PREFACE

ABOUT THIS DOCUMENT

This document aims to depict the main guidelines when producing, storing, testing and using programming products and making sure the outcome of every run on every Kubernetes cluster will be identical.

AUDIENCE

The intended audience for this document are developers and DevOps personnel wishing to establish a line of product stream which is trusted, stable, secure and persistent throughout environments.

REVISIONS

Name	Version	Date	Comments
Liran Cohen	1.0	10.7.2019	Final

Table I-b: Revisions

Executive summary

When producing applications it is customary to safe the output of a successful compilation in an artifact repository for use on any environment and to maintain revisions of the product produced. The saved objects are called artifacts and they are stored in an artifact repository to make them accessible to anyone who is allowed and wishes to deploy these artifacts on their environment.
Kubernetes based applications are following a similar discipline while the artifacts are the containers being run on the Kubernetes cluster itself. These artifacts must be built, tested and signed (the latter is optional but highly recommended) in order to keep the same deployment consistent across clusters and environments.

Software artifacts

The term artifact comes from the Lating word combination: arte (by or using art) and factum (something made) in software terms an artifact is a finalized version of a programming process, as described in the following suggested process:

1. The developer writes code.
2. testing the code and performing validation.
3. Building (compiling) the code if required.
4. Packing the output file(s) into a product bundle.
5. Deploying the output product bundle onto a test environment.
6. signing the output product bundle.
7. Upload the product bundle onto an artifact repository.

The above pipeline describes a suggested CI pipeline at the end of which a trusted, tested and signed artifact is produced and saved for use by others (for example CD pipelines). The main advantage is the fact that using this artifact along with the corresponding configuration will result in a trusted deployment.

Kubenrtes Artifact definitions

In the container world and in Kubernetes clusters in particular, the stage in which the output files are packaged is even more advanced and requires building a container imageready to deploy on any Kubernetes cluster.
The above process will help achieve ephemerality of applications with accordance to the 12 factor app guidelines (https://12factor.net/) mainly Build-release-run and config points.
this methodology turn the microservice based application into a set of trusted building blocks, tested and signed for use on any Kubernetes cluster.

Kubernetes artifact provisioning

The above guidelines, while seem reasonable may require additional tools in order to make a container packed artifact work in the environment on which it is deployed.
For example, imagine you have an external database with which your microservice communicated, the database credentials can indeed be set in a Kubernetes secret and the rest of the connection configuration (URL, database port, database name etc.) can be injected into the container (which will run in a Pod on Kubernetes) using a configmap or directly using environment variables. but what if provisioning the database schema for the first time is required by the microservice? , in such situations there are several tools to help you achieve such provisioning abilities without changing the artifact itself at any stage.
For example using init containers which are containers that run to completion before the main container runs and can be used for provisioning, dynamic configuration etc.

The risk of changing container content

Several methodologies evolved when using third party applications or cluster specific software which may cause instability of running applications and raise the probability of unwanted container \ Pod behavior.
for example, injecting a telemetry collection agent into a container just before deploying it (using init containers, sidecars or any other method) will “break the seal” and create a whole new artifact different than the original (as intended by the developer).
This new artifact was not tested by the application developer and was not approved or verified.

The outcome is a new Pod with the original application and additional code or infra-container configuration changes which, should a problem occur, make it very difficult to find the root cause and like with every product, breaking the “seal” voids the manufacturer warranty if one wishes to point to a similar scenario from the retail market.

While in the VM world it was customary to install all kinds of agents, collectors, scanners and many other software types on the VM on which the application is running, in the containers world and Kubernetes in particular, it is more than frowned upon mainly because containers (and Pods) are not VMs , they behave differently, run differently and fail differently, to add to the containers “cauldron”, when running container on Kubernetes, one does not even know where the container is running or even how many instances of the same container are running, hence investigating a malfunction may prove to be a daunting task if reproduction is impossible and the running artifact is different from the original one.

Artifact versioning

The concept of version deployment also underwent a substantial change. The concept of rolling forward if done correctly, minimizes the probability of malfunctions as a result of version mismatch between containers.
The term rollback, although still exists, is strongly discouraged.

In case a hotfix is required, a new sealed version of the existing container is fabricated, given a newer version number, and deployed into the Kubernetes cluster as a newer version. This way, the “new” container version which is in fact the existing container with the desired fix, will be tested using the same environment in which the existing container we wish to issue the hotfix for thus avoiding an older version from being deployed onto an existing, newer version environment.
tempering with containers after they are packed in any way may result in the unwanted scenario where an older version of an injected piece of software breaks the container and the application itself as a result of version mismatch.

Artifact security

As a container is released, it should be tested in various aspects including security. after passing such tests the end user can trust the container to be certified by the author in all aspects and especially security wise.
Injecting an additional piece of software \ data \ credentials etc. into the container after it is signed by the author, other than those provided by the Kubernetes cluster (such as configMaps, secrets etc.) may expose the application and the data which it processes to various security vulnerabilities emanating from multiple situations starting with the vulnerabilities in the injected data all the way through to application interaction with the injected data \ application vulnerabilities.

Artifact profiling

When creating a deployment yaml for kubernetes, one has to perform a comprehensive application profiling and state various configurations to be deployed along with the container. some examples are the resource request, resource limits, serviceaccount etc.

Injecting additional software into a container will probably require changing these values as the requirements change accordingly. an agent installed onto a container for example may not require much memory or cpu at idle times, but when the load increases even a 5% increase may cause failures and in extreme cases, even trigger an eviction process for that Pod.
Another example is a case where an injected agent requires elevated privileges and as a result the whole Pod gets these elevated permissions significantly increasing the risk of running such a Pod in production (especially with user-facing Pods such as dashboards, APIs etc.)

Summary

In order to keep artifacts and applications in tact through normal operations, version deployment and allow for easier troubleshooting and smoother application deployment, container must be kept sterile as much as possible, sealed and signed to verify content and application integrity throughout deployments in the same cluster and \ or different clusters. The long term stability of a distributed container based application is highly dependant to the content being consistent and stable.