Destination Earth Core Service Platform

DESP Integration Procedure: Runtime

Destination Earth
Funded by the European Union
Implemented by ECMWF esa EUMETSAT

Reference: DEST-SRCO-TN-2300355
Date: 14/09/2023
Version:1.0

Role/Title	Name	Signature	Date
Author	DESP Team		14/09/2023
Verified	DESP Service Manager		14/09/2023
Approved	DESP Contract Manager		14/09/2023

Change register

Version/Rev.	Date	Change	Reason
1.0	14/09/2023		First release of the document

Table of Contents

Index of Figures

Index of Tables

1. Introduction

1.1 Scope

This document provides the procedure to deploy a Service on DESP Runtime Platform for the "*Destination Earth – DestinE Core Service Platform Framework – Platform & Data Management Services*".

1.2 Purpose

The purpose of this document is to provide a guide for any Service Provider who intends to deploy a service on the DESP Runtime Platform.

This document follows the Dev and Integration Guidelines [RD-1] provided by TAS to document the deployment procedure of the DESP Platform Management Services.

1.3 Applicable Documents

Ref.	Title	Reference and Version
AD-1	[DP-SOW] Statement of Work - Destination Earth – Destine Core Service Platform Framework – Platform & Data Management Services	ESA-EOPG-EOPGD-SOW-10, v 1.0
AD-2	[AD-DSP-TSR] DESP Framework – Platform & Data Management Services – Technical and Service Requirements	ESA-EOPG-EOPGD-RS-10, v1.0
AD-3	[AD-DDL-DP] DestinE – System Framework – Data Portfolio	EUM/TSS/DOC/22/1279455, v1G, 09/09/2022
AD-4	[AD-DSP-SR] DESP Framework – Platform & Data Management Services – Security Requirements	ESA-ESO-SSRS-2022-0111, v1.0
AD-5	Space engineering – Software	ECSS-E-ST-40C, 06/03/2009

1.4 Reference Documents

Ref.	Title	Reference and Version
RD-1.	Dev and Integration Guidelines	ESA-DESP-TAS-001 v01.00 draft
RD-2.	DESP SDD and Master ICD	DEST-SRCO-DD-2300317
RD-3.	Security Coding Rules	83231328-DDQ-TAS-EN

1.5 Acronyms and Abbreviations

Acronym	Definition
AD	**A**pplicable **D**ocument
API	**A**pplication **P**rogramming **I**nterface
CaaS	**C**ontainer **a**s **a** **S**ervice
CSM	**C**yber **S**ecurity **M**anager
CVE	**C**ommon **V**ulnerabilities and **E**xposure
DB	**D**ata**B**ase

DESP	**D**estin**E** core **S**ervice **P**latform
DIAS	**D**ata and **I**nformation **A**ccess **S**ervices
ECSS	**E**uropean **C**ooperation for **S**pace **S**tandardization
ESA	**E**uropean **S**pace **A**gency
E2E	**E**nd to **E**nd
GDPR	**G**eneral **D**ata **P**rotection **R**egulation
GUI	**G**raphical **U**ser **I**nterface
IaaS	**I**nfrastructure **a**s **a** **S**ervice
IAM	**I**dentity and **A**ccess **M**anagement
IVV	**I**ntegration, **V**erification and **V**alidation
PaaS	**P**latform **a**s **a** **S**ervice
PVC	**P**ersistent **V**olume **C**laim
RD	**R**eference **D**ocument
SSO	**S**ingle **S**ign **O**n
TBC	**T**o **B**e **C**onfirmed
TBD	**T**o **B**e **D**efined

2. Platform Overview

The Runtime Platform is the platform that hosts DESP Core Services and that can host also now upcoming DESP Services. It is composed of a Container-as-a-Service solution, based on Kubernetes and Rancher with DESP Platform Services integrated on top. The DESP Platform Services integrated in the runtime platform are:

- Identity and Access Management Service
- Service Registry
- Service Operations Monitoring Dashboard Service
- Executive Dashboard
- Web Portal
- Accounting Service.

This runtime will be provided in stable versions for Data Management Services and new upcoming services to integrate in it. The runtime allows the Services to use the underlying OVH infrastructure in a simple way, granting to them by default the scalability, availability, and security of the required infrastructure.

For the DESP program, two separate projects are to be provisioned in a single OVH account. These two projects are called "Staging" and "Production".

The **"Staging"** project embeds the End2End integration environment used by Serco for integration and validation testing of the DESP Data Management Services and of new upcoming Services in the DESP Platform. It includes the last stable version of the Runtime platform. It is also used to be representative of the production environment during investigation of issues occurring on the Production environment. It could also be used as backup environment of the Production environment in case of major outage. This environment is hosted in a Public Cloud Project on the OVH public cloud infrastructure (IaaS).

The **"Production"** project will host the operational DESP environment including Runtime (which corresponds to the CaaS and PaaS of DESP) and Core services. The Production environment is hosted in a Public Cloud Project on the OVH public cloud infrastructure (IaaS), in a different datacentre with respect to the Staging environment.

Each Data Management Service and each new upcoming Service hosted on the Runtime in the DESP will have a dedicated namespace in E2E project and will be hosted in a dedicated Public Cloud Project, separated from the Public Cloud Project that hosts the main Runtime Platform with the integrated Platform Management Services.

3. How to deliver a Service for integration

To deploy a new Service in DESP, a Service Provider may use the following pre-configured resources:

- A Gitlab Server with Gitlab CI templates to build and analyse binaries,
- Rancher Fleet to deploy apps on a set of clusters.

The following documentation sections give a full example of the deployment of a sample app.

The whole process is represented here in below:

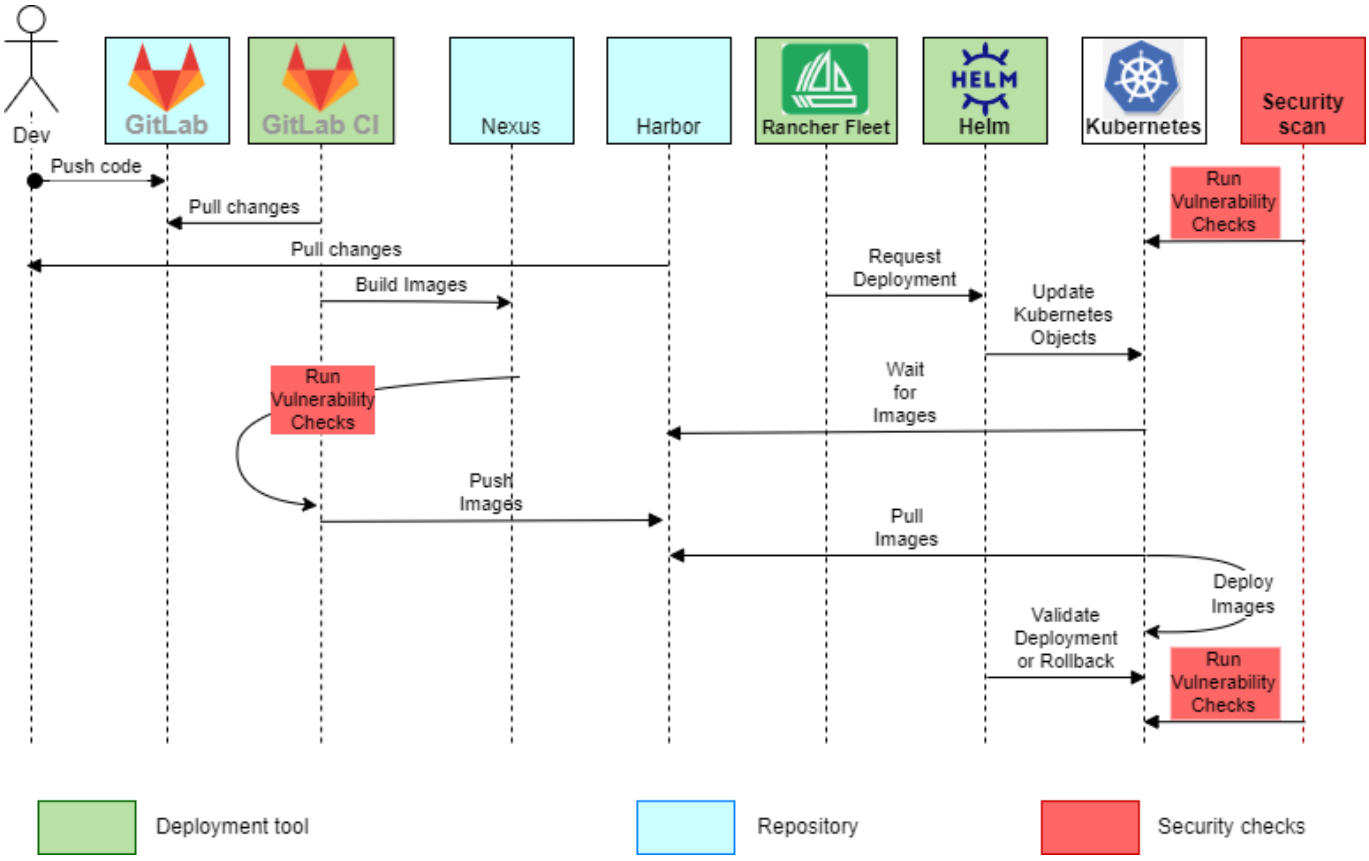

Figure 1 - Sequence diagram of build, checks and deployment pipelines

Security is an important part of DESP build and deployment process. Security checks are mandatory:

- at build phase (with Gitlab CI),
- at runtime deployment phase.

Any detected vulnerability will have to be managed by the Serco/TAS team according to an agreed frame concerning fixation / rejection / temporary pass through of vulnerabilities. Any critical or major (CVE with CVSS score > 7) vulnerability will drive to a rejection of the related Data Management Service.

DESP Services that cannot release code in GitLab will be able to deposit container images directly into a dedicated Nexus repository (which will populate the Harbor repository thanks to CI/CD pipelines, between the Gitlab CI and Harbor).

Nevertheless, CVE scans will be performed on containers deposited into the Nexus and a report will be produced to inform about severity of detected vulnerabilities and exposure. The process mentioned in the previous paragraph will be applicable to the concerned containers and associated libraries.

Concerning the code checks that will not be possible to perform in such a case, the Service Provider shall provide the CSM with a release document signifying that it has complied with the rules set out by the CSM (including the provided coding rules). The Service Provider shall provide the SAST report from their internal factory, including a clear traceability to the coding rules.

It will be the responsibility of the Service Provider to enable or not the code quality check on each of its provided component (it will be part of the deployment configuration to be provided by the Service Provider at delivery).The Service Provider can be laid responsible in case of missing or inconsistent code check that leads to a compromising of DESP.

Verification evidence shall be provided to IVV team before activating the IVV steps.

3.1 Repository organization

Each Service is provided with a private registry implemented as a "Harbor" repository.

3.2 GITLab structure

Each Service is hosted in a dedicated GitLab project, to which identity and job role restrained access management is applicable.

All DESP services are delivered in an overall GROUP = DESP.

In this group, we will use the structure below:

[DESP]

/[Function]/ … /[subcomponent]

Including:

- *<deployment> directory (see below)*
- *<images> directory: it contains the dockerfiles required to build the container images*
- *gitlab-ci.yml: it contains the CI/CD*
- *readme.md*

<Deployment> directory will be composed at least of:

- *charts*
- *fleet.yml*
- *HELM files: Chart.yaml, templates/, etc...*
- *values.<env>.yaml*
- *readme.md specific to the chart*

Example:

DESP/SAMPLE-APP/sample-app

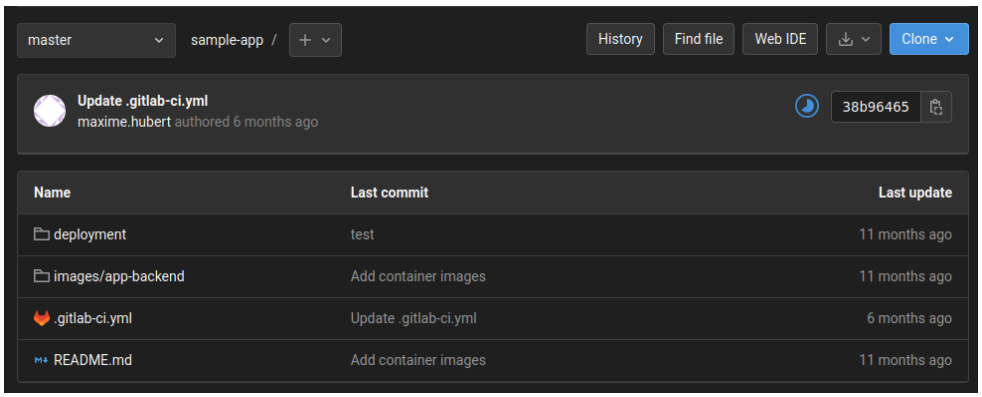

Figure 2 - Example of a repository matching the structure

3.3 Harbor organization

Each Service will have its own Harbor project, where container shall be pushed using the Gitlab CI templates.

Developers will be given a set of credentials to access their Harbor project in the E2E environment.

Example:

We want to publish images for the Sample App Service. It features two images: image-a and image-b.

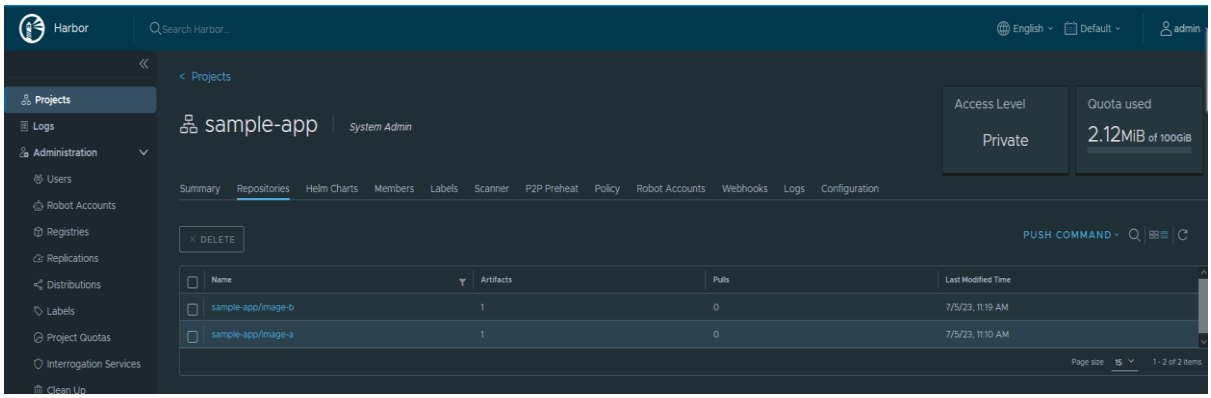

Figure 3 - Example of a Harbor project for the Sample App component with two images

The path to these images shall be:

<service-name>/image/path:version

A secret store (using the project name) and an external secret (using the Harbor token) must be created in the Service namespace to access the Harbor images.

3.4 Access rights

Full rights to each registry are granted by default to representatives of the companies in charge of the development of the Service linked to that registry. More users can be added by Serco upon request. Each single user shall have a unique and personal account.

In addition, full access rights are provided to Serco representatives on each registry.

3.5 Write code and build images

The recommended setup is to the following:

Start from an empty git repository. Create one in Gitlab or with the command line.

Use a .gitlab-ci.yml file that points to CI libraries instead of writing it from scratch

3.5.1 Create a git repository

To create a new git repository:

```
$ git init sample-app
$ cd sample-app
$ touch README.md
```

At the end, the directory should look like this:

```
$ ls
images  README.md
```

Request a remote repository on the Gitlab server and push code there.

3.5.2 Build and deploy container images

To build a container image, place the Dockerfile in the git repository:

```
$ mkdir images/app-backend
$ vi images/app-backend/Dockerfile
```

Test the builds locally:

```
$ docker build ./images/app-backend/ -t app-backend:v0.1
```

If this works, add the following configuration to a .gitlab-ci.yml file:

```
include:
  - project: $DEVSECOPS_TEMPLATES_REPOSITORY
    ref: $DEVSECOPS_TEMPLATES_REF
```

```
    file: "sample-app-devsecops.yml"
```

Create variables for each image to build:

```
variables:
  BACKEND_APP_IMAGE: "sample-app/sample-app-backend:1.0.0"
```

Add BUILD block to build the image:

```
build-sample-app-backend:
  variables:
    DOCKER_BUILDPATH: images/app-backend/
    TEST_IMAGE: "$BACKEND_APP_IMAGE"
  extends: .build-e2e
```

Add RELEASE block to release the image on an environment:

```
release-e2e-sample-app-backend:
  needs:
    - job: "build-sample-app-backend"
  extends: .release-e2e
        variables:
          TEST_IMAGE: "$BACKEND_APP_IMAGE"
```

This file makes sure that once the code is pushed an image is built from "images/app-backend/", with the tag "registry.xxx/components/sample-app-backend:1.0.0"

This image is deployed in E2E environment.

The docker release jobs may extend the following release environments:

```
.release-e2e
.release-prod
```

Note: If several images are under build in a single repository, go to settings à "CI/CD Settings" and uncheck "Skip outdated deployment jobs".

At the end, the directory should look like this:

```
$ ls -a
. .. deployment .git .gitlab-ci.yml images README.md
```

Commit and push the code, and check the CI pipeline that has been triggered:

Figure 4 - Image has now been built and released in the configured environment

3.6 Deploy the service image with HELM and Rancher Fleet

Helm is used through the "charts" definition to ease and accelerate with two commands, the deployment of an application over Kubernetes clusters.

The recommended setup is to the following:

Create a new application using "helm init", even to deploy an existing helm chart. This will give flexibility with the configuration.

$ helm create sample-app

$ mv sample-app deployment

At the end, the directory should look like this:

$ ls

deployment images README.md

3.6.1 Fleet deploy

Rancher Fleet is the tool used to deploy Helm templates on Kubernetes clusters. To use it, there are two prerequisites:

- An access to the Rancher server: it will be possible to see and configure deployments in the "Continuous Deployment" menu.
- A fleet.yml file alongside the helm template files in the git repository.

Example:

To deploy the sample app in the E2E environment:

- Create the following file in "deployment/fleet.yaml":

```
defaultNamespace: sample-app
# Custom helm options
helm:
  # How long for helm to wait for the release to be active. If the value
  # is less than or equal to zero, we will not wait in Helm
  timeoutSeconds: 0
targetCustomizations:
```

```
- name: e2e
  clusterSelector:
    matchLabels:
      customer: e2e
  helm:
    chart: .
    valuesFiles:
      - ./values.e2e.yaml
```

- Connect to Rancher:

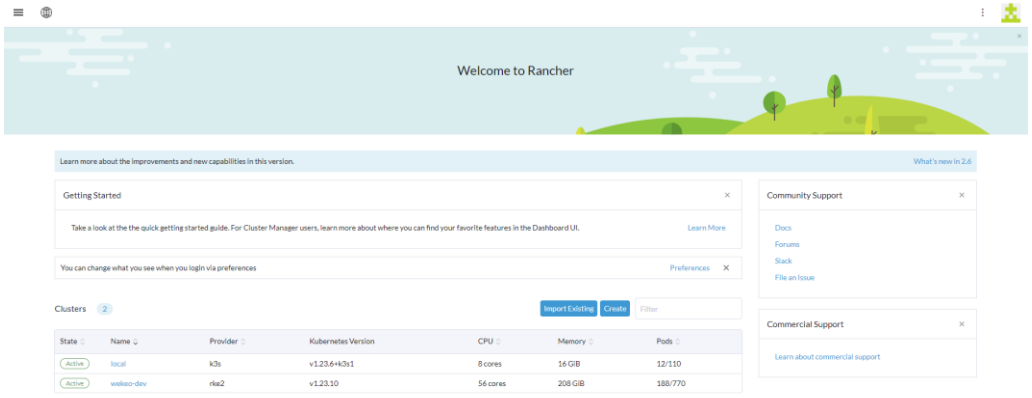

Figure 5 - Rancher welcome page

- Navigate to "Continuous Delivery" on left menu

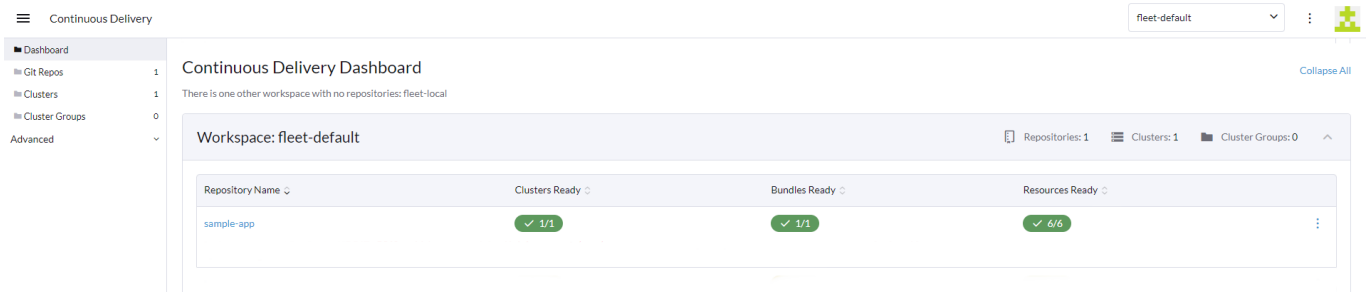

Figure 6 - Rancher CD Dashboard

- Configure a Git Repo, set the following parameters:
  - URL and branch or tag to point to the repository
  - Add a new path and set it to "deployment"
  - Set the credentials used to pull the git repository
  - Check the TLS Certificate Verification parameter

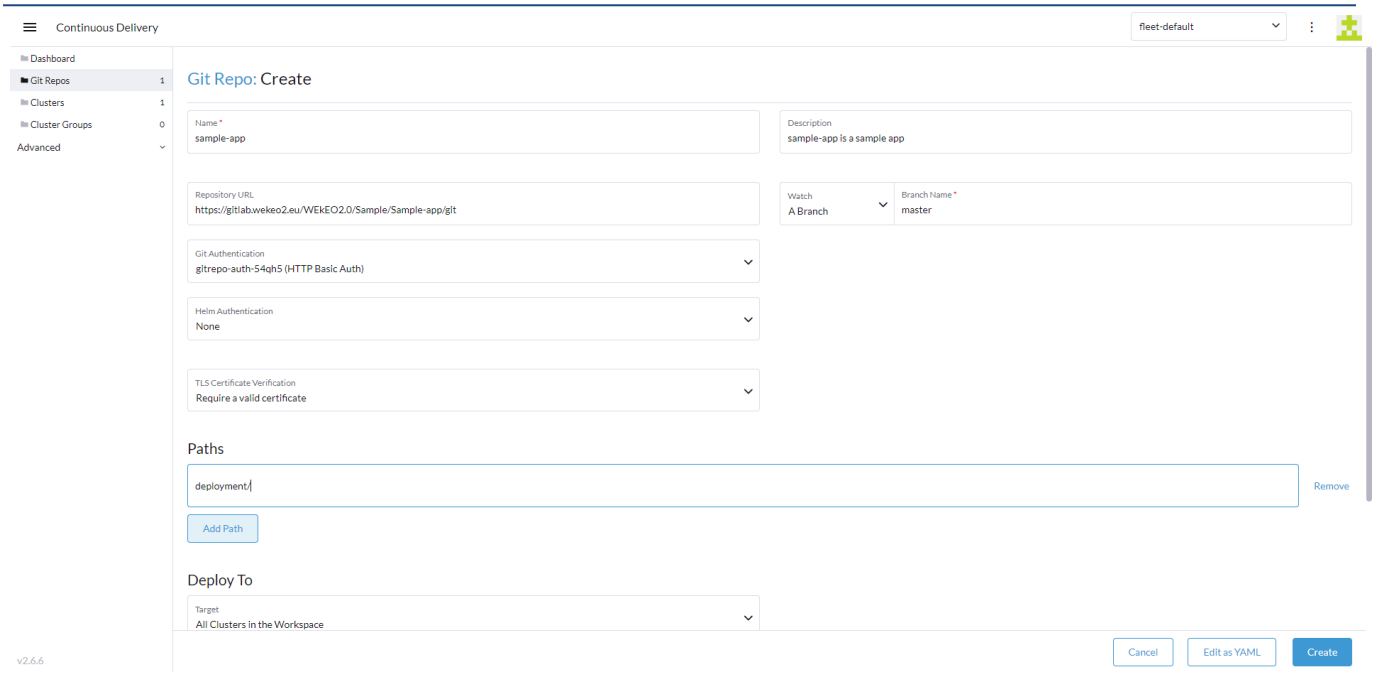

Figure 7 - Rancher Repo creation

Check the progress of the deployment in "Continuous Delivery".
The app will be updated anytime the code is updated in the repository.
More information in the official documentation: https://fleet.rancher.io

3.6.2 Pull images on the target cluster

Once uploaded in the project under the platform registry, the images are private. This section explains how to configure the Service to pull these images from the private platform registry.

3.6.2.1 Create the imagePullSecret in the namespace

Create the following files under the templates/ directory of the helm charts. The entries to configure are the following:

- *my-path* is the name of a vault key-value store configured by the service administrator
- *my-role* is the name of a vault role configured by the service administrator

```
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
    name: vault-backend
spec:
    provider:
        vault:
            server: "http://vault.vault.svc:8200"
```

```
        path: "my-path"
        version: "v2"
        auth:
          kubernetes:
            mountPath: "kubernetes"
            role: "my-role"
            serviceAccountRef:
              name: "default"
---
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: pull-secret
spec:
  dataFrom:
  - extract:
      conversionStrategy: Default
      decodingStrategy: Base64
      key: pull-secret
  refreshInterval: 1m
  secretStoreRef:
    kind: SecretStore
    name: vault-backend
  target:
    creationPolicy: Owner
    deletionPolicy: Retain
    name: harbor-image-pull
    template:
      engineVersion: v2
      mergePolicy: Replace
      type: kubernetes.io/dockerconfigjson
```

More information: https://external-secrets.io/latest/guides/common-k8s-secret-types/#dockerconfigjson-example

3.6.2.2 Validate the configuration

Check that the configuration has been deployed properly in the namespace:

```
$ kubectl get externalsecrets.external-secrets.io -n my-namespace
NAME          STORE          REFRESH INTERVAL STATUS          READY
pull-secret   vault-backend  1m               SecretSynced    True
```

```
$ kubectl get secrets harbor-image-pull -n my-namespace
NAME                TYPE                             DATA    AGE
harbor-image-pull   kubernetes.io/dockerconfigjson   1       56m
```

It is possible to now use the harbor-image-pull Secret as an imagePullSecret for the application. For a running example, see the Sample App project.

3.6.2.3 Troubleshoot

If the status of the pull-secret is SecretSyncedError, it is possible to get more information contacting the Serco team. Provide them with the output of the following command:

```
$ kubectl describe externalsecrets.external-secrets.io pull-secret
```

3.6.3 How to create secret with values from vault

Let's consider that following keys have been added in the vault:

Table 1 - Secret Keys

secretstore	role	secret	key	value
paas	pass-read	s3	access	accessForS3
			secret	secretForS3

The SecretStore object vault-backend allowing access to secretstore paas is also ready. If not, check the previous chapter on SecretStore creation.
An external secret vault-s3-secret must be created, using the SecretStore object vault-backend:

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: vault-s3-secret
spec:
  refreshInterval: 1m
  secretStoreRef:
    kind: SecretStore
    name: vault-backend
  target:
    creationPolicy: Owner
    deletionPolicy: Retain
    name: sample-app-s3-secret
  dataFrom:
  - extract:
```

```
        key: s3
```

it will create the secret sample-app-s3-secret containing the keys:


```
{
    "apiVersion": "v1",
    "data": {
        "access": "YWNjZXNzRm9yUzM=",
        "secret": "c2VjcmV0Rm9yUzM="
    },
    "immutable": false,
    "kind": "Secret",
    "metadata": {
        "annotations": {
            "meta.helm.sh/release-name": "sample-app-test-deployment",
            "meta.helm.sh/release-namespace": "sample-app",
            "objectset.rio.cattle.io/id": "default-sample-app-test-deployment",
            "reconcile.external-secrets.io/data-hash": "5a1c1f22b3445c1634a94757f72bede1"
        },
        "creationTimestamp": "2023-07-28T15:54:17Z",
        "labels": {
            "app.kubernetes.io/managed-by": "Helm",
            "objectset.rio.cattle.io/hash": "55b937b2157ebae4188a133aeb827ddaeaad2f70"
        },
        "name": "sample-app-s3-secret",
        "namespace": "sample-app",
        "ownerReferences": [
            {
                "apiVersion": "external-secrets.io/v1beta1",
                "blockOwnerDeletion": true,
                "controller": true,
                "kind": "ExternalSecret",
                "name": "vault-s3-secret",
                "uid": "52eaea58-2f03-4c05-9dfd-b707eff84a18"
            }
        ],
        "resourceVersion": "9380718",
        "uid": "2cd3f775-2e20-4e7b-91c7-a364d59b982d"
    },
    "type": "Opaque"
}
```


Then in the deployment file, it is necessary to add the keys as environment variables:

```
    - env:
        - name: S3_ACCESS_KEY_VALUE
```

```
      valueFrom:
        secretKeyRef:
          key: access
          name: sample-app-s3-secret
    - name: S3_SECRET_KEY_VALUE
      valueFrom:
        secretKeyRef:
          key: secret
          name: sample-app-s3-secret
```

After the pod is deployed, the ENV variables have been set with the vault values:

```
$ kubectl exec -it -n sample-app sample-app-test-deployment-6d77c69d88-2cfzq -- sh -c 'echo $S3_ACCESS_KEY_VALUE'
accessForS3
```

3.6.4 Edit Helm templates

Edit the files under "deployment". Follow the official Helm documentation to get more information.

It is possible to add several "values.*.yaml" if it is necessary to provide different configurations per environment.

Feel free to verify the helm templates with the following command:

```
$ helm -n my-namespace template app ./deployment/
```

Or if there are several values files:

```
$ helm -n my-namespace  template app ./deployment/ -f ./deployment/values.yaml -f ./deployment/values.E2E.yaml
```

Note: In that case, values from "./deployment/values.E2E.yaml" override the one in "./deployment/values.yaml".

General HELM documentation is available on: https://helm.sh/docs/.

Helm provides several mechanisms that must be used (if needed).

3.6.4.1 Chart hooks

Helm provides a hook mechanism to allow chart developers to intervene at certain points in a release's life cycle. For example, it is possible to use hooks to:

- Load a ConfigMap or Secret during install before any other charts are loaded.
- Execute a Job to back up a database before installing a new chart, and then execute a second job after the upgrade in order to restore data.

- Run a Job before deleting a release to gracefully take a service out of rotation before removing it.

Hooks work like regular templates, but they have special annotations that cause Helm to utilize them differently. In this section, we cover the basic usage pattern for hooks.

The following hooks are defined:

Table 2 - Hook Annotation

Annotation Value	Description
pre-install	Executes after templates are rendered, but before any resources are created in Kubernetes
post-install	Executes after all resources are loaded into Kubernetes
pre-delete	Executes on a deletion request before any resources are deleted from Kubernetes
post-delete	Executes on a deletion request after all of the release's resources have been deleted
pre-upgrade	Executes on an upgrade request after templates are rendered, but before any resources are updated
post-upgrade	Executes on an upgrade request after all resources have been upgraded
pre-rollback	Executes on a rollback request after templates are rendered, but before any resources are rolled back
post-rollback	Executes on a rollback request after all resources have been modified

Writing a Hook

Hooks are just Kubernetes manifest files with special annotations in the metadata section. Because they are template files, it is possible to use all of the normal template features, including reading .Values, .Release, and .Template.

For example, this template, stored in templates/post-install-job.yaml, declares a job to be run on post-install:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: "{{ .Release.Name }}"
  labels:
    app.kubernetes.io/managed-by: {{ .Release.Service | quote }}
    app.kubernetes.io/instance: {{ .Release.Name | quote }}
    app.kubernetes.io/version: {{ .Chart.AppVersion }}
    helm.sh/chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
  annotations:
    # This is what defines this resource as a hook. Without this line, the
```

```
    # job is considered part of the release.
    "helm.sh/hook": post-install
    "helm.sh/hook-weight": "-5"
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  template:
    metadata:
      name: "{{ .Release.Name }}"
      labels:
        app.kubernetes.io/managed-by: {{ .Release.Service | quote }}
        app.kubernetes.io/instance: {{ .Release.Name | quote }}
        helm.sh/chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
    spec:
      restartPolicy: Never
      containers:
      - name: post-install-job
        image: "alpine:3.3"
        command: ["/bin/sleep","{{ default "10" .Values.sleepyTime }}"]
```

What makes this template a hook is the annotation:

```
annotations:
  "helm.sh/hook": post-install
```

One resource can implement multiple hooks:

```
annotations:
  "helm.sh/hook": post-install,post-upgrade
```

Similarly, there is no limit to the number of different resources that may implement a given hook. For example, one could declare both a secret and a config map as a pre-install hook.

When subcharts declare hooks, those are also evaluated. There is no way for a top-level chart to disable the hooks declared by subcharts.

It is possible to define a weight for a hook which will help build a deterministic executing order. Weights are defined using the following annotation:

```
annotations:
  "helm.sh/hook-weight": "5"
```

Hook weights can be positive or negative numbers but must be represented as strings. When Helm starts the execution cycle of hooks of a particular kind it will sort those hooks in ascending order.

Hook deletion policies

It is possible to define policies that determine when to delete corresponding hook resources. Hook deletion policies are defined using the following annotation:

annotations:

"helm.sh/hook-delete-policy": before-hook-creation,hook-succeeded

It is possible to choose one or more defined annotation values:

Table 3 - Hook annotation values for deletion

Annotation Value	Description
before-hook-creation	Delete the previous resource before a new hook is launched (default)
hook-succeeded	Delete the resource after the hook is successfully executed
hook-failed	Delete the resource if the hook failed during execution

If no hook deletion policy annotation is specified, the before-hook-creation behaviour applies by default.

3.6.4.2 Labels and Annotations

The following table defines common labels that Helm charts use. Helm itself never requires that a particular label be present. Labels that are marked REC are recommended and should be placed onto a chart for global consistency. Those marked OPT are optional. These are idiomatic or commonly in use but are not relied upon frequently for operational purposes.

Table 4 - Common labels used by Helm charts

Name	Status	Description
app.kubernetes.io/name	REC	This should be the app name, reflecting the entire app. Usually {{ template "name" . }} is used for this. This is used by many Kubernetes manifests, and is not Helm-specific.
helm.sh/chart	REC	This should be the chart name and version: {{ .Chart.Name }}-{{ .Chart.Version | replace "+" "_" }}.
app.kubernetes.io/managed-by	REC	This should always be set to {{ .Release.Service }}. It is for finding all elements of application package managed by Helm.
app.kubernetes.io/instance	REC	This should be the {{ .Release.Name }}. It aids in differentiating between different instances of the same application.
app.kubernetes.io/version	OPT	The version of the app and can be set to {{ .Chart.AppVersion }}.
app.kubernetes.io/component	OPT	This is a common label for marking the different roles that pieces may play in an application. For example, app.kubernetes.io/component: frontend.
app.kubernetes.io/part-of	OPT	When multiple charts or pieces of software are used together to make one application. For example,

		application software and a database to produce a website. This can be set to the top level application being supported.

3.6.4.3 Built-in Objects

Objects are passed into a template from the template engine. And the code can pass objects around (we'll see examples when we look at the with and range statements). There are even a few ways to create new objects within the templates, like with the tuple function we'll see later.

Objects can be simple and have just one value. Or they can contain other objects or functions. For example. the Release object contains several objects (like Release.Name) and the Files object has a few functions.

In the previous section, we use {{ .Release.Name }} to insert the name of a release into a template. Release is one of the top-level objects that it is possible to access in the templates.

- Release: This object describes the release itself. It has several objects inside of it:
- Release.Name: The release name
- Release.Namespace: The namespace to be released into (if the manifest doesn't override)
- Release.IsUpgrade: This is set to true if the current operation is an upgrade or rollback.
- Release.IsInstall: This is set to true if the current operation is an install.
- Release.Revision: The revision number for this release. On install, this is 1, and it is incremented with each upgrade and rollback.
- Release.Service: The service that is rendering the present template. On Helm, this is always Helm.
- Values: Values passed into the template from the values.yaml file and from user-supplied files. By default, Values is empty.
- Chart: The contents of the Chart.yaml file. Any data in Chart.yaml will be accessible here. For example {{ .Chart.Name }}-{{ .Chart.Version }} will print out the mychart-0.1.0.

The built-in values always begin with a capital letter. This is in keeping with Go's naming convention. When names are created, it is possible to use a convention that suits the service team. Some teams, like many whose charts available on Artifact Hub, choose to use only initial lower case letters in order to distinguish local names from those built-in. In this guide, we follow that convention.

More precise documentation is available on https://helm.sh/docs/.

3.6.5 How to set certificate for Ingress

The certificate for ingress must be retrieved from Let's encrypt.

All domain name must end with: desp.space

In order to generate a certificate for the Service, the following annotations must be added in the ingress metadata:

annotations:

cert-manager.io/cluster-issuer: letsencrypt-prod

Then in the spec part, the TLS block must be added with the hostname used by the Service:

- app.example.com: The hostname of the service application
- tls-secret: The name of the secret to create, can be customized.

```
tls:
- hosts:
  - app.example.com
  secretName: tls-secret
```

The certificate and the secret will be automatically generated.

When the service application is accessed, it is possible to then check the certificate is well generated.

3.6.6 Use secrets

Application secrets such as passwords and certificate keys **cannot be stored in the source code**. This section describes how to use these secrets in the application.

Secrets will be created in the vault service for each environment upon request.

3.6.6.1 How to use these secrets in the service application

Secrets consists of key-value pairs, created in a common Vault service, under the following location:

<backend-key-value-store>/<secret-path>/<key>:<value>

These back-ends will be created by the IVV team, and an access to manage them will be provided to developers.

First Method: Pod annotations

In this example, secrets have been populated under wso2/mysql-root and wso2/mysql-wso2carbon, with the key "password".

```
podAnnotations:
  vault.hashicorp.com/agent-inject: "true"
  vault.hashicorp.com/agent-inject-status: update
  vault.hashicorp.com/agent-inject-secret-mysql-password-root: wso2/mysql-root
  vault.hashicorp.com/agent-inject-template-mysql-password-root: |
    {{`{{- with secret "wso2/mysql-root" -}}
    {{ .Data.data.password }}
    {{- end -}}`}}
  vault.hashicorp.com/agent-inject-secret-mysql-password-wso2carbon: wso2/mysql-wso2carbon
  vault.hashicorp.com/agent-inject-template-mysql-password-wso2carbon: |
```

```
    {{`{{- with secret "wso2/mysql-wso2carbon" -}}
    {{ .Data.data.password }}
    {{- end -}}`}}
  vault.hashicorp.com/role: wso2-mysql-root
```

Second method: External Secrets

In this example, secrets have been populated under openldap/secrets, with the admin_password and config_password properties.

This configuration will synchronize the secret stored in the Vault service with a Kubernetes secret called "ldap-passwords" with the keys "LDAP_CONFIG_PASSWORD" and "LDAP_ADMIN_PASSWORD":

```
---
apiVersion: external-secrets.io/v1alpha1
kind: SecretStore
metadata:
  name: openldap-secrets-store
  namespace: "{{ .Release.Namespace }}"
spec:
  provider:
    vault:
      server: "{{ .Values.externalSecrets.server }}"
      # Path is the mount path of the Vault KV backend endpoint
      path: "openldap"
      version: "v2"
      namespace: "vault"

      auth:
        # Kubernetes auth: https://www.vaultproject.io/docs/auth/kubernetes
        kubernetes:
          mountPath: "kubernetes"
          role: "openldap-password-read"
---
apiVersion: external-secrets.io/v1alpha1
kind: ExternalSecret
metadata:
  namespace: "{{ .Release.Namespace }}"
  name: secrets
```

```
spec:
  # SecretStoreRef defines which SecretStore to use when fetching the secret data
  secretStoreRef:
    name: openldap-secrets-store
    kind: SecretStore
  target:
    name: ldap-passwords
  data:
    - secretKey: LDAP_ADMIN_PASSWORD
      remoteRef:
        key: secrets
        property: admin_password
    - secretKey: LDAP_CONFIG_PASSWORD
      remoteRef:
        key: secrets
        property: config_password
```

3.6.6.2 Project Defined Variable (TBC)

Table 5 - Project Defined Variables

Name	Description
DESP_DOMAIN	
DESP_SAMPLE_APP_AUTH_USER	
DESP_SAMPLE_APP_AUTH_PASS	
DESP_SAMPLE_APP_CACHE_S3_ACCESS	
DESP_SAMPLE_APP _CACHE_S3_SECRET **DESP_SAMPLE_APP _CACHE_S3_URL**	
DESP_SAMPLE_APP _CACHE_S3_BUCKET	

4. Use Kubernetes Storage Classes

4.1 Openstack Cinder storage class

This part is dedicated to Persistent Volume Claims (PVC) for S3 buckets. When creating a PVC with the Openstack Cinder Storage Class, a Cinder Volume is created in the Openstack Cinder Service.
If the PVC is deleted, the following occurs:

- The Cinder volume and its content is deleted if the StorageClass is not Persistent.
- The Cinder volume and its content is kept if the StorageClass is Persistent.

Table 6 - Cinder classes

Name	Persistent	Shared FS
csi-cinder-sc-delete	No	No
csi-cinder-sc-retain	Yes	No

Full documentation

Example:

```
$ cat << EOF | kubectl apply --insecure-skip-tls-verify  -f -
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-claim
spec:
  storageClassName: csi-cinder-sc-delete
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Mi
EOF
persistentvolumeclaim/test-claim created
```

A new volume called pvc-746d9d0f-88d7-4bf1-9f29-4154f863d7a8 has been created. It is possible to see it in the Openstack UI

```
$ kubectl get pvc
NAME         STATUS   VOLUME                                     CAPACITY   ACCESS MODES
STORAGECLASS          AGE
test-claim   Bound    pvc-746d9d0f-88d7-4bf1-9f29-4154f863d7a8   1Gi        RWX
csi-cinder-sc-delete      47s
```

Upon deletion, the volume is deleted in Openstack, and it is possible to validate it in the Openstack UI.

$ kubectl delete pvc test-claim

persistentvolumeclaim "test-claim" deleted

The same test can be performed with the csi-cinder-sc-retain storage class. The volume will remain in Openstack.

4.2 Dynamic NFS Storage class

This part is dedicated to Persistent Volume Claims (PVC) for NFS storage class toward Kubernetes. When creating a PVC with the NFS Storage Class, a sub-directory is created in the internal NFS Server. If the PVC is deleted, the following occurs:

- The sub-directory and its content is deleted if the StorageClass is not Persistent.
- The sub-directory and its content is kept if the StorageClass is Persistent.

Table 7 - NFS Classes

Name	Persistent	Shared FS
temporary-nfs-storage	Yes	Yes
persistent-nfs-storage	No	Yes
persistent-nfs-storage-with-path	Yes	Yes

Paths

When creating a PVC with these storage classes, the following paths are created in the NFS server:

- temporary-nfs-storage: A path under ${.PVC.namespace}-${.PVC.name} is created in the NFS server
- persistent-nfs-storage: A path under ${.PVC.namespace}-${.PVC.name} is created in the NFS server
- persistent-nfs-storage-with-path: A path under ${.PVC.annotations.nfs.io/storage-path} is created: it means that it is possible to customize the path with a nfs.io/storage-path: custom/directory annotation in the PVC.

Full documentation

Example:

$ cat << EOF | kubectl apply --insecure-skip-tls-verify -f -

kind: PersistentVolumeClaim

apiVersion: v1

metadata:

* name: test-claim*

```
spec:
  storageClassName: temporary-nfs-storage
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Mi
EOF
persistentvolumeclaim/test-claim created
```

The - sub-directory is created in the nfs server:

```
$ ls /mnt/nfs-server/
default-test-claim
```

Upon deletion, the directory is deleted:

```
$ kubectl delete pvc  test-claim
persistentvolumeclaim "test-claim" deleted

$ ls /mnt/nfs-server/default-test-claim/
ls: cannot access '/mnt/nfs-server/default-test-claim/': No such file or directory
```

The same test can be performed with the persistent-nfs-storage storage class. The sub-directory will remain in the NFS server.

5. DEVSECOPS & IVV Processes

This section describes the processes implemented in the DESP E2E environment to assure our deliveries with respect to DEVSECOPS rules.

5.1 Environment

5.1.1 DEVSECOPS Build environment

DevSecOps is defined as a set of practices that combine development and operations teams with security teams to secure the application development process from the beginning.

One of the critical components of DevSecOps is continuous integration/continuous delivery (CI/CD).

CI/CD helps project automate the application delivery process, from code development to product deployment. This can help project speed up the delivery of new features and fixes while reducing the risk of errors and security vulnerabilities.

5.1.2 E2E Environment

The End2End (E2E) environment is used for integration of DESP Data Management Services and new upcoming services on the last stable version of the integrated DESP Platform (Runtime + DESP Platform Management Services). Here integration, validation and verification tests are executed by the actors.

DESP Data Management Service or a new DESP Service can be deployed in the E2E environment in order to start its integration and validation in a fully functionally representative environment of DESP Platform.

The DESP Data Management Services are here integrated with the last stable version of the Runtime Platform integrated with the DESP Platform Management Services, that is deployed in the E2E environment from the IVV environment after successful integration and verification.

The E2E environment is thus used to run scenarios implying several applicative Services and to run end to end tests. This validation process receives continuously improvements of the Services either to increase the functionality of the system or its quality (better performance, bug-free).

Once the system configuration has reached the required level for the associated upgrade, it is proposed for deployment on the operational environment ('PROD').

5.2 Integration Verification Validation and Deployment

This section describes the processes put in place in the frame of the Integration Verification and Validation Strategy.

The new paradigm introduced with the notions of continuous integration, increments and the service approach allows having an optimised verification process and continuous improvement of the service both in terms of quality of service (SLAs) and in terms of functional scope.

5.2.1 Integration

The release integration is performed in a five steps approach.

1. The Service will be released for system integration on E2E.

2. Integration, verification and end-to-end validation tests are executed to check the correct system behaviour of the Services. This is performed by running scenarios implying several applicative Services and to run end to end system tests.

3. Each issue detected during these phases is managed by creating a service problem report (SPR) in Gitlab and is reviewed in dedicated review board (following appropriate workflow).

4. Solved and known SPRs are listed in the release note of each release to be delivered at each delivery level.

5. Each delivery/deployment during these phases are managed using GIT tag.

Annex 1. Semantic Versioning Specification (SemVer) 2.0.0

Summary

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version : incompatibility with API changes
- MINOR version: functionality added in a backward compatible manner
- PATCH version: backward compatible bug fixes

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Introduction

In the world of software management there exists a dreaded place called "dependency hell." The bigger the system grows and the more packages shall be integrates into the software, the more likely a set of problems arise.

In systems with many dependencies, releasing new package versions can quickly become a nightmare. If the dependency specifications are too tight, there is the risk of version lock (the inability to upgrade a package without having to release new versions of every dependent package). If dependencies are specified too loosely, there will be version promiscuity (assuming compatibility with more future versions than is reasonable). Dependency hell happens when version lock and/or version promiscuity prevent from easily and safely moving the project forward.

As a solution to this problem, we propose a simple set of rules and requirements that dictate how version numbers are assigned and incremented. These rules are based on but not necessarily limited to pre-existing widespread common practices in use in both closed and open-source software. For this system to work, it is necessary to first declare a public API. This may consist of documentation or be enforced by the code itself. Regardless, it is important that this API be clear and precise. Once the service public API is identified, changes shall be communicated to it with specific increments to the version number. Consider a version format of X.Y.Z (Major.Minor.Patch). Bug fixes not affecting the API increment the patch version, backward compatible API additions/changes increment the minor version, and backward incompatible API changes increment the major version.

We call this system "Semantic Versioning." Under this scheme, version numbers and the way they change convey meaning about the underlying code and what has been modified from one version to the next.

Semantic Versioning Specification (SemVer)

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

1. Software using Semantic Versioning MUST declare a public API. This API could be declared in the code itself or exist strictly in documentation. However it is done, it SHOULD be precise and comprehensive.
2. A normal version number MUST take the form X.Y.Z where X, Y, and Z are non-negative integers, and MUST NOT contain leading zeroes. X is the major version, Y is the minor version, and Z is the patch version. Each element MUST increase numerically. For instance: 1.9.0 -> 1.10.0 -> 1.11.0.

3. Once a versioned package has been released, the contents of that version MUST NOT be modified. Any modifications MUST be released as a new version.
4. Major version zero (0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.
5. Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes.
6. Patch version Z (x.y.Z | x > 0) MUST be incremented if only backward compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.
7. Minor version Y (x.Y.z | x > 0) MUST be incremented if new, backward compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.
8. Major version X (X.y.z | X > 0) MUST be incremented if any backward incompatible changes are introduced to the public API. It MAY also include minor and patch level changes. Patch and minor versions MUST be reset to 0 when major version is incremented.
9. A pre-release version MAY be denoted by appending a hyphen and a series of dot separated identifiers immediately following the patch version. Identifiers MUST comprise only ASCII alphanumerics and hyphens [0-9A-Za-z-]. Identifiers MUST NOT be empty. Numeric identifiers MUST NOT include leading zeroes. Pre-release versions have a lower precedence than the associated normal version. A pre-release version indicates that the version is unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version. Examples: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92, 1.0.0-x-y-z.--.
10. Build metadata MAY be denoted by appending a plus sign and a series of dot separated identifiers immediately following the patch or pre-release version. Identifiers MUST comprise only ASCII alphanumerics and hyphens [0-9A-Za-z-]. Identifiers MUST NOT be empty. Build metadata MUST be ignored when determining version precedence. Thus two versions that differ only in the build metadata, have the same precedence. Examples: 1.0.0-alpha+001, 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85, 1.0.0+21AF26D3----117B344092BD.
11. Precedence refers to how versions are compared to each other when ordered.
12. Precedence MUST be calculated by separating the version into major, minor, patch and pre-release identifiers in that order (Build metadata does not figure into precedence).
13. Precedence is determined by the first difference when comparing each of these identifiers from left to right as follows: Major, minor, and patch versions are always compared numerically. Example: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1.
14. When major, minor, and patch are equal, a pre-release version has lower precedence than a normal version:
    Example: 1.0.0-alpha < 1.0.0.
15. Precedence for two pre-release versions with the same major, minor, and patch version MUST be determined by comparing each dot separated identifier from left to right until a difference is found as follows:
16. Identifiers consisting of only digits are compared numerically.
17. Identifiers with letters or hyphens are compared lexically in ASCII sort order.
18. Numeric identifiers always have lower precedence than non-numeric identifiers.
19. A larger set of pre-release fields has a higher precedence than a smaller set, if all of the preceding identifiers are equal.
    Example: 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0.

About

The Semantic Versioning specification was originally authored by Tom Preston-Werner, inventor of Gravatar and cofounder of GitHub.

License

Annex 2. HELM template file examples

Deployment file with Helm built-in variables

TBC

Config Map

TBC

Annex 3. DESP applicable requirements

System Design Requirements

See [RD-2].

DEV Requirements

Reminder below of applicable requirements for Development activities:

Table 8 - DEV requirements table

Req.ID	Requirement Description	Requirement Note	Thematic
DESP-DEV-001	Docker container shall be in dedicated namespace of Service in respect of the namespace naming convention		Development rule
DESP-DEV-002	All data for the configuration of the Service shall be contained in Helm values files, and then passed to the containers using configmaps or environment variables file		Development rule
DESP-DEV-003	All committed Service source code provided shall not contain any hardcoded login and/or password sensitive information such as login/passwords, API keys, or secret keys.		Development rule
DESP-DEV-004	Service provided shall not contain any hardcoded parameters		Development rule
DESP-DEV-005	Docker design and implementation shall be serverless? Stateless maybe?		Development rule
DESP-DEV-006	All Services shall be compliant from Kubernetes v1.23.6		Development rule
DESP-DEV-007	All Services shall be deployable using Helm. Helm templates and values files shall be provided.		Development rule Configuration/Deployment
DESP-DEV-008	Docker Service design and implementation shall support replica and Horizontal Pod Autoscalling features of Kubernetes.		Development rule
DESP-DEV-009	Helm file to deploy the Service shall be provided. Note: Only helm will be used to deploy the Services		Configuration/Deployment
DESP-DEV-010	Service provided shall implement liveness probe		Development rule
DESP-DEV-011	Service provided shall implement healthiness probe		Development rule
DESP-DEV-013	Service design and implementation shall be stateless compliant.		Development rule
DESP-DEV-014	Service provided shall use applicative account with generic operation email		Configuration/Deployment
DESP-DEV-015	Service provided shall use environment variable configuration file for deployment		Configuration/Deployment
DESP-DEV-016	Service provided shall expose all logs in stdout/stderr		Development rule

DESP-DEV-017	The partner shall use development infrastructure and tools provided by the Customer.		Development rule
DESP-DEV-018	Logs and events shall be sent to the site central log server within 5 minutes of the log event creation for the provision of detailed usage statistics and monitoring and alerting.		Monitoring
DESP-DEV-033	Once deployed, the Kubernetes application shall only download Docker images from a private project Docker registry.		Development rule
DESP-DEV-034	Helm charts shall allow to specify the download source of all container images in their values.yaml file.		Development rule
DESP-DEV-035	The application shall set all confidential data (credentials, certificate keys, etc...) using Kubernetes Secrets or pods annotations for the Vault service. These secrets will be generated and managed by the Customer in the Vault service provided by the IVV team.		Development rule
DESP-DEV-036	Dependencies and Dockerfiles shall be provided for each image deployed by the Kubernetes application.		Development rule
DESP-DEV-037	Any Services provided by the Partner shall successfully pass the DevSecOps analysis provided by the Customer.		Development rule
DESP-DEV-040	A Git flow shall be provided to integrate each application (tag / branches for each features / Merge on master branch ...)		Development rule
DESP-DEV-042	The Partner shall apply for its development the coding rules provided by the Customer.		Development rule
DESP-DEV-043	The Partner shall provide automatic tests scripts covering DESP requirements that is compliant with the Customer including test data.		Development rule
DESP-DEV-045	Development and Deployment rules document shall be applicable for the Partner for its development and the document could evolve along the project including the extensions periods (options).		Development rule
DESP-DEV-046	The Partner shall provide resource requirements consumption(CPU, RAM, Storage) needed for each Service that he provides for DESP		
DESP-DEV-049	All pre-install or post-install actions (sql commands, shell or curl commands) shall be declared using initContainers or Helm hooks.		Configuration/Deployment
DESP-DEV-050	All deployments of docker images or Helm packages shall be automated by extending the provided Gitlab CI templates.		Configuration/Deployment
DESP-DEV-051	All Services assets (source code, container images) should pass the security scans part of the Gitlab CI templates. All CVEs detected in vulnerability scans with a HIGH or		Development rule

	CRITICAL score should be fixed or accepted by TAS security officer.		

Security requirements

See [RD-3].