August 2018

Technical Network Validation Using Open-shift

AUTHOR:
Neha Gupta

MENTORS:
Alexandre Lossent
Alberto Rodriguez Peon

EXECUTIVE SUMMARY

The interest in using containers to package applications is constantly growing in the software development community, especially with new technologies such as Kubernetes, Open-shift being adopted more frequently as well. This project also based on modularising the currently adopted monolithic structure and building different micro-services out of it, one such micro-service that defined the goal for this project was building a technical network validating admission controller. Modularising the application increased resilience, provided robust monitoring and no single point of failures.

The aim of this project is to build a technical network validating admission controller using Golang deployed in Open-shift that will ensure only whitelisted applications that are visible to the technical network, where the whitelisted applications are assumed to be provided be the security team.

TABLE OF CONTENTS

Background

2.1 Openshift

2.1.1 Overview

OpenShift is a layered system designed to expose underlying Docker and Kubernetes concepts as accurately as possible, with a focus on easy composition of applications by a developer. For example, install Ruby, push code, and add MySQL. More flexibility of configuration is exposed after creation in all aspects of the model. The concept of an application as a separate object is removed in favour of more flexible composition of "services", allowing two web containers to reuse a database or expose a database directly to the edge of the network.

2.1.2 What Are the Layers?

Docker provides the abstraction for packaging and creating Linux-based, lightweight containers. Kubernetes provides the cluster management and orchestrates Docker containers on multiple hosts. OpenShift adds:

1. Source code management, builds, and deployments for developers

2. Managing and promoting images at scale as they flow through your system

3. Application management at scale

4. Team and user tracking for organising a large developer organisation

2.1.3 What Is the OpenShift Architecture?

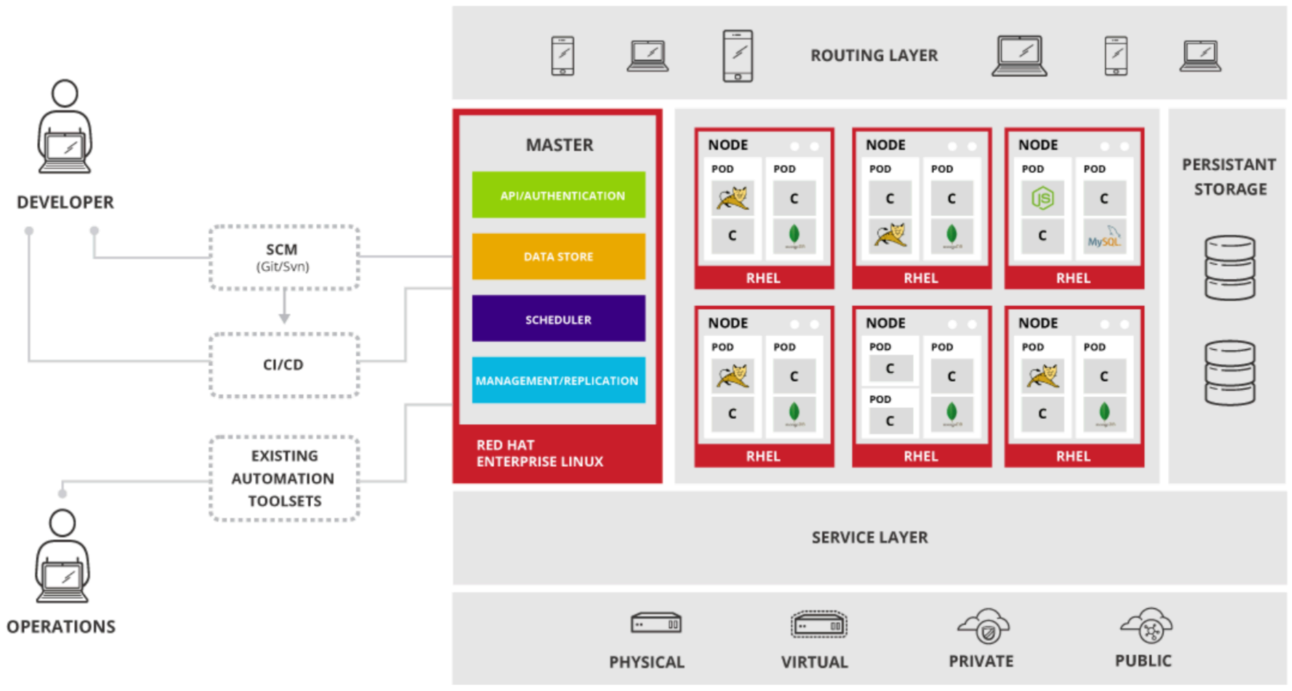
OpenShift has a micro-services-based architecture of smaller, decoupled units that work together. It can run on top of (or alongside) a Kubernetes cluster, with data about the objects stored in etcd, a reliable clustered key-value store. Those services are broken down by function REST APIs, which expose each of the core objects.

Controllers, which read those APIs, apply changes to other objects, and report status or write back to the object. One make calls to the REST API to change the state of the system. Controllers use the REST API to read the user's desired state, and then try to bring the other parts of the system into sync. For example, when a user requests a build they create a "build" object. The build controller sees that a new build has been created, and runs a process on the cluster to perform that build. When the build completes, the controller updates the build object via the REST API and the user sees that their build is complete.

The controller pattern means that much of the functionality in OpenShift is extensible. The way that builds are run and launched can be customised independently of how images are managed, or how deployments happen. The controllers are performing the "business logic" of the system, taking user actions and transforming them into reality. By customising those controllers or replacing them with your own logic, different behaviours can be implemented. From a system administration perspective, this also means the API can be used to script common administrative actions on a repeating schedule. Those scripts are also controllers that watch for changes and take action. OpenShift makes the ability to customise the cluster in this way a first-class behaviour.

To make this possible, these controllers leverage a reliable stream of changes to the system to sync their view of the system with what users are doing. This event stream pushes changes from etcd to the REST API and then to the controllers as soon as changes occur, so changes can ripple out through the system very quickly and efficiently.

However, since failures can occur at any time, the controllers must also be able to get the latest state of the system at startup, and confirm that everything is in the right state. This resynchronisation is important, because it means that even if something goes wrong, then the operator can restart the affected components, and the system double checks everything before continuing. The system should eventually converge to the user's intent, since the controllers can always bring the system into sync.



2.2 **Docker**

2.2.1 **Overview**

The basic units of OpenShift applications are called containers. Many application instances can be running in containers on a single host without visibility into each others' processes, files, network, and so on. Typically, each container provides a single service (often called a "micro-service"), such as a web server or a database, though containers can be used for arbitrary workloads. OpenShift and Kubernetes add the ability to orchestrate Docker containers across multi-host installations.

Though we do not directly interact with Docker tools when using OpenShift, understanding Docker's capabilities and terminology is important for understanding its role in OpenShift and how this project's controller applications functions inside of containers.

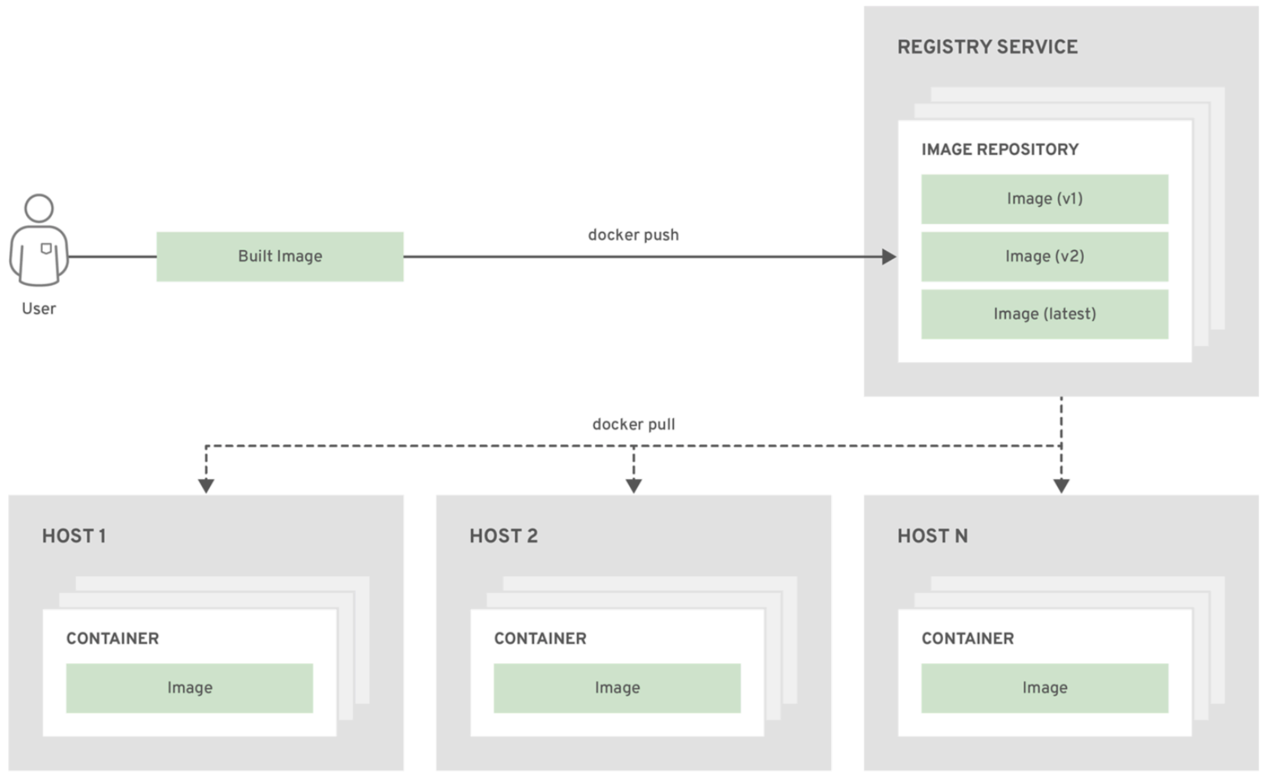
Docker containers are based on Docker images. A Docker image is a binary that includes all of the requirements for running a single Docker container, as well as metadata describing its needs and capabilities. We can think of it as a packaging technology. Docker containers only have access to resources defined in the image, unless we give the container additional access when creating it. By deploying the same image in multiple containers across multiple hosts and load balancing between them, OpenShift can provide redundancy and horizontal scaling for a service packaged into an image.

We can use Docker directly to build images, but OpenShift also supplies builders that assist with creating an image by adding your code or configuration to existing images. Since applications develop over time, a single image name can actually refer to many different versions of the "same" image. Each different image is referred to uniquely by its hash.

2.2.2 **Docker Registry**

A Docker registry is a service for storing and retrieving Docker images. A registry contains a collection of one or more Docker image repositories. Each image repository contains one or more tagged images. Docker provides its own registry, the Docker Hub, but we can also use private or third-party registries. OpenShift can also supply its own internal registry for managing custom Docker images.

The relationship between containers, images, and registries is depicted in the following diagram:



2.3 AdmissionController

2.3.1 Overview

Admission web-hooks call web-hook servers to either mutate pods upon creation, such as to inject labels, or to validate specific aspects of the pod configuration during the admission process.

Admission web-hooks intercept requests to the master API prior to the persistence of a resource, but after the request is authenticated and authorised.

2.3.2 Admission Web-hooks

In OpenShift Container Platform we can use admission web-hook objects that call web-hook servers during the API admission chain. There are two types of admission web-hook objects we can configure:

1. **Mutating admission** web-hooks allow for the use of mutating web-hooks to modify resource content before it is persisted.

2. **Validating admission** web-hooks allow for the use of validating web-hooks to enforce custom admission policies.

Configuring the web-hooks and external web-hook servers is beyond the scope of this report. However, the web-hooks must adhere to an interface in order to work properly with OpenShift Container Platform.

When an object is instantiated, OpenShift Container Platform makes an API call to admit the object. During the admission process, a mutating admission controller can invoke web-hooks to perform tasks, such as injecting affinity labels. At the end of the admissions process, a validating admission controller can invoke web-hooks to make sure the object is configured properly, such as verifying affinity labels. If the validation passes, OpenShift Container Platform schedules the object as configured.

When the API request comes in, the mutating or validating admission controller uses the list of external web-hooks in the configuration and calls them in parallel.

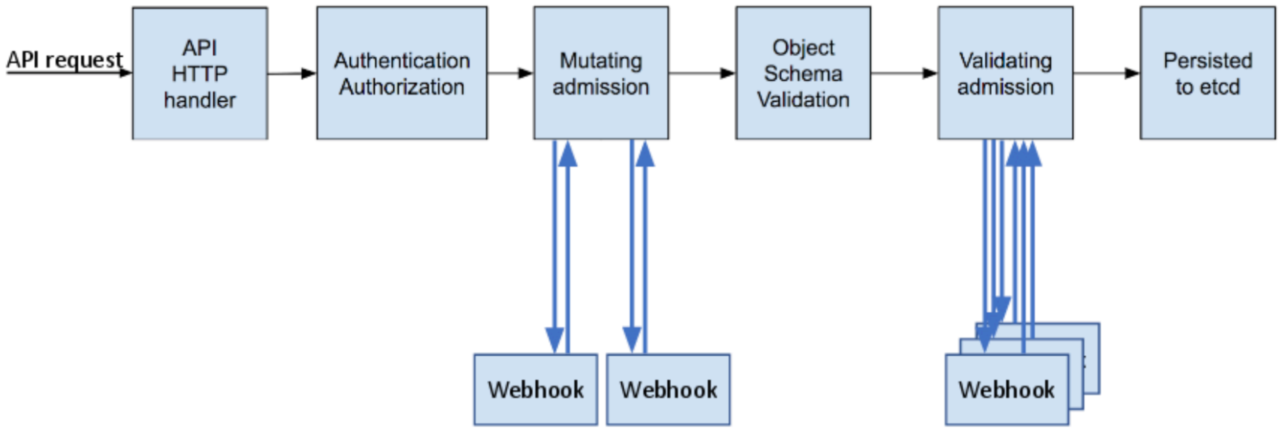
1. If all of the web-hooks approve the request, the admission chain continues.

2. If any of the web-hooks deny the request, the admission request is denied, and the reason for doing so is based on the first web-hook denial reason.
If more than one web-hook denies the admission request, only the first will be returned to the user.

3. If there is an error encountered when calling a web-hook, that request is ignored and is be used to approve/deny the admission request.

The communication between the admission controller and the web-hook server needs to be secured using TLS. Generate a CA certificate and use the certificate to sign the server certificate used by the web-hook server. The PEM-formatted CA certificate is supplied to the admission controller using a mechanism, such as Service Serving Certificate Secrets.

The following diagram illustrates this process with two admission web-hooks that call multiple web-hooks.



2.3.3 Validating Admission Web-hooks

Validating admission web-hooks are invoked during the validation phase of the admission process. This phase allows the enforcement of invariants on particular API resources to ensure that the resource does not change again. The Pod Node Selector is also an example of a validation admission, by ensuring that all nodeSelector fields are constrained by the node selector restrictions on the project.

Reasons for switching to micro-services

A variant of service-oriented architecture (SOA), micro-services is an architectural style in which applications are decomposed into loosely coupled services. With fine-grained services and lightweight protocols, micro-services offers increased modularity, making applications easier to develop, test, deploy, and, more importantly, change and maintain.

With micro-services, the code is broken into independent services that run as separate processes. Output from one service is used as an input to another in an orchestration of independent, communicating services. Microservices is especially useful for businesses that do not have a pre-set idea of the array of devices its applications will support.

1. Increased resilience

With micro-services, the entire application is decentralised and decoupled into services that act as separate entities. Unlike the monolithic architecture wherein a failure in the code affects more than one service or function, there is minimal impact of a failure using micro-services. Even when several systems are brought down for maintenance, the users won’t notice it.

2. Improved scalability

Scalability is the key aspect of micro-services. Because each service is a separate component, we can scale up a single function or service without having to scale the entire application. Business-critical services can be deployed on multiple servers for increased availability and performance without impacting the performance of other services.

3. The ability to use the right tool for the right task

With micro-services, we don't have to get tied up with a single vendor. Instead, we have the flexibility to use the right tool for the right task. Each service can use its own language, framework (in this project we are using Golang), or ancillary services while still being able to communicate easily with the other services in your application.

4. Continuous delivery

Because micro-services works with loosely coupled services, we don't need to rewrite the entire codebase to add or modify a feature. We make changes only to a specific service. By developing applications in smaller increments that are independently testable and deployable, you can get your application and services to market quicker.

Unlike monolithic applications, in which dedicated teams work on discrete functions such as UI, database, server-side logic, and technological layers, micro-services uses cross-functional teams to handle the entire life cycle of an application using a continuous delivery model. When developers, operations, and testing teams work simultaneously on a single service, testing and debugging becomes easy and instant. With this approach of incremental development, code is continuously developed, tested and deployed, and we can use code from existing libraries instead of reinventing the wheel.

7. Robust monitoring is a must

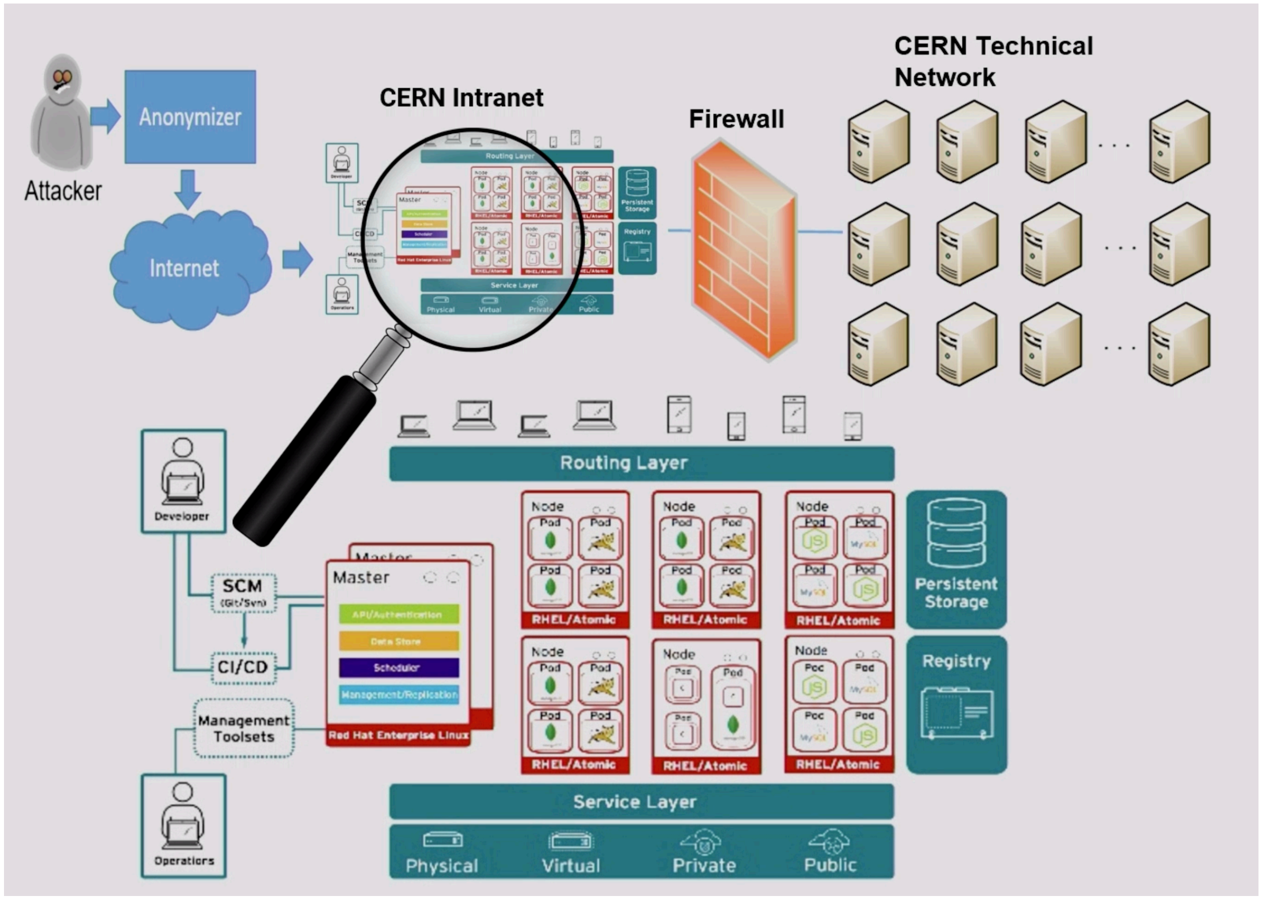
Because each service relies on its own language, platform, and APIs, and we will be orchestrating different entities of the micro-services project, we need robust monitoring to effectively monitor and manage the entire infrastructure, because if we don't know when a service fails or a machine goes down, it may be impossible to track down issues when they arise.

There can be some downsides as well with micro-services, like, testing isn't straightforward. Each service has its own dependencies, some direct, others transitive. As features are added, new dependencies will pop up. Keeping tabs on all this quickly becomes impractical. Plus, as the number of services increases, so too does the complexity. Whether it's database errors, network latency, caching issues, or service unavailability, the micro-services architecture better be able to handle a reasonable level of faults. So, resiliency testing and fault injection are a must.

Also, we need to design with failure in mind. Designing for failure is essential. We should be prepared to handle multiple failure issues, such as system downtime, slow service and unexpected responses. Here, load balancing is important, but having a plan B is another important option. When a failure arises, the troubled service should still run in a degraded functionality without crashing the entire system.

Implementation

The go controller takes care of accepting and dropping the change in annotations of any Open-shift application depending upon the whitelist that is allowed to be exposed to CERN technical network. The basic architecture of the network looks like the following:



Logic behind the Controller

```
If route has annotation `router.cern.ch/technical-network-access: true` then
  Check if route annotation `router.cern.ch/network-visibility` has value `Internet`:
    in this case route is requesting visibility on TN+Internet+Intranet;
    otherwise it is requesting TN+Intranet
  Check value of label `router.cern.ch/technical-network-allowed` on the parent namespace:
    If `Intranet`, the project is allowed to expose routes to TN+Intranet;
    if `Internet`, the project is allowed to expose routes to TN+Intranet and TN+Internet+Intranet
    if not present or other value: project is not allowed to expose routes to TN at all
  If the requested route visibility is not allowed by the label on the namespace, reject the route creati
```

```
1. TN : Technical Network

2. Open-shift’s conventions on standard Open-shift route annotations are
followed, so the following spellings in annotation/label values are
equivalent: Internet, internet, INTERNET.

3. Similarly, True/true/TRUE are equal.
```

Deployment

To get started spin up a local Open-shift cluster and login as an admin. Make sure to enable the ValidatingAdmissionWebhook in the open-shift cluster configuration and then deploy the web-hook that is invoked each time a there is a create or update in the annotations of any service running the open shift cluster in the CERN technical network.

Create a new Project, say test and login using the authentication token in local open shift cluster which was spin up and create a deployment and a service. After the deployment and service are rolling, create the ClusterRoleBinding role and a service account.

Note that if the AdmissionController is not working, there might be a case that it’s not enabled in the open-shift cluster configuration, so you can enable it by adding the following code and restart the service and check again.

```
admissionConfig:
  pluginConfig:
        ValidatingAdmissionWebhook:
            configuration: {
              kind: DefaultAdmissionConfig, apiVersion: v1,
              disable: false
              }
```

Conclusion and Future Work

This project was a success in achieving the milestone. We now have a go controller that we can deploy on Open-shift clusters that will take care of accepting and dropping the change in annotations of any Open-shift application depending upon the whitelist that is allowed to be exposed to CERN technical network. There is still some work to be done on the controller for example, it can be integrated with the continuous deployment and integration to reduce the manual interventions, thus lesser scope of errors.

References

The source repository is present at https://gitlab.cern.ch/paas-tools/paas-infra/ValidatingAdmissionController.git or https://github.com/nehagup/ValidatingAdmissionController.git

Openshift Development kit https://docs.okd.io/latest/architecture/core_concepts/

Dynamic Admission Control https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/

Gitlab CI/CD https://docs.gitlab.com/ee/ci/