openshift-deep-dive

Contents

Introduction **7**
- Container platform . . . 7
  - Containers in OpenShift . . . 7
  - Orchestrating Containers . . . 8
- Examining the architecture . . . 8
  - Integrating container images . . . 8
  - Accessing applications . . . 8
  - Handling network traffic . . . 9
- Examining an application . . . 9
  - Building an application . . . 9
  - Deploying and serving applications . . . 9
- Use case for platforms . . . 9
- Technology use cases . . . 9
- Businesses use cases . . . 10
- Invalid use cases . . . 10
- Container storage . . . 11
- Scaling applications . . . 11
- Integrating stateful and stateless apps . . . 11

Starting **12**
- Cluster runtimes . . . 12
- Cluster installation . . . 13
- Cluster login . . . 13
- Cluster config . . . 14
- Cluster debugging . . . 15
- Cluster software . . . 16
- First project . . . 16
- Application Components . . . 16
  - Container images . . . 17
  - Build configs . . . 17
  - Deployment configs . . . 17
  - Image stream . . . 18
- Deploying an app . . . 18
- Providing access to apps . . . 19
- Exposing application services . . . 20

Containers **21**
- Defining containers . . . 21
- OpenShift component interaction . . . 21

OpenShift manages deployments . . . 22
Kubernetes schedules applications . . . 22
Docker creates containers . . . 23
Linux isolates resources . . . 23
Working with cluster . . . 23
Listing kernel components . . . 25
    Mount namespace . . . 26
    UTS namespace . . . 30
    PID namespace . . . 31
    Memory namespace . . . 33
    Networking namespace . . . 33
    Summary . . . 35
Cloud native apps . . . 36
    Testing app resiliency . . . 36
Scaling applications . . . 39
Application Health . . . 40
    Creating liveness probes . . . 40
    Creating readiness probes . . . 42
Auto-scaling with metrics . . . 43
    Determining expected workloads . . . 44
    Installing OpenShift metrics . . . 44
    Understanding the metrics . . . 45
    Using pod metrics & autoscaling . . . 45
    Testing the autoscaling setup . . . 47
    Avoiding thrashing . . . 48
Continuous integration & deployment . . . 48
    Container images are the centerpiece . . . 49
    Promoting images . . . 49
    CI/CD:1 Creating an environment - TODO: this needs rewrite . . . 50
    Deployment strategies . . . 51
Stateful applications . . . 53
    Container storage . . . 53
Handling permanency . . . 54
Creating resources . . . 55
    Creating storage . . . 56
    Logging in as `kubeadmin` . . . 57
    Physical volume . . . 58
    Creating volumes . . . 59
    Creating claims . . . 60
    Modifying deployment . . . 61
Volume mounts . . . 63

Stateful applications **64**
    Enabling a headless services . . . 65
    Application clustering with WildFly . . . 65
    Querying the OpenShift server . . . 66
    Verify the data replication . . . 69
    Other cases for direct pod access . . . 69
    Describing sticky sessions . . . 69
    Toggling sticky sessions . . . 70
    Limitations of cookies . . . 71

Shutting down applications . . . 72
Application grace period . . . 72
Container lifecycle hooks . . . 74
Stateful sets . . . 74
Deterministic sequence . . . 75
Examining a stateful set . . . 75
Constant Network identity . . . 78
Consistent persistent storage . . . 79
The Stateful set limitations . . . 79
Non-native stateful applications . . . 79
Cleanup resource limits & quotas . . . 80

Operations & Security **80**
Permissions vs Wild-West . . . 80
Setting up authentication . . . 81
Introduction to htpasswd . . . 81
Creating htpasswd files . . . 81
Changing the provider . . . 82
Working with roles . . . 83
Assigning user roles . . . 83
Creating administrators user . . . 84
Setting default roles . . . 84
Limit ranges . . . 85
Define resource limit ranges . . . 86
Resource quotas . . . 87
Creating compute quotas . . . 88
Creating resource quotas . . . 89
Working with quotas & limits . . . 90
Quotas to existing applications . . . 90
Changing quotas for deployed applications . . . 91
Cgroups for managing resources . . . 91
Overview of the cgroups . . . 92
Identifying container cgroups . . . 93
Confirming cgroups limits . . . 94

Networking **95**
Managing the SDN . . . 95
Configure application node network . . . 96
Linking containers to host interfaces . . . 97
Working with OVS . . . 98
Routing application requests . . . 100
Using the HAProxy service . . . 100
Investigating the HAProxy service . . . 101
HAProxy and request routing . . . 102
Locating services with internal DNS . . . 103
DNS resolution in pod network . . . 104
Configure OpenShift SDN . . . 105
Using the ovs-subnet . . . 105
Using the ovs-multitenant . . . 105
Creating advanced network designs . . . 106
Enabling multi tenant plugin . . . 106

Testing the multi-tenant plugin . . . 108

Security **108**

SELinux core concepts . . . 109

Working with SELinux labels . . . 110

Applying labels with SELinux context . . . 111

Enforcing SELinux with policies . . . 112

Isolating pods with levels . . . 114

Investigating pod security context . . . 114

Examining the `MCS` levels . . . 115

Managing Linux capabilities . . . 116

Controlling the user ID . . . 117

Scanning container images . . . 117

Annotating images with security information . . . 118

- Introduction
  - Container platform
    - Containers in OpenShift
    - Orchestrating Containers
  - Examining the architecture
    - Integrating container images
    - Accessing applications
    - Handling network traffic
  - Examining an application
    - Building an application
    - Deploying and serving applications
  - Use case for platforms
  - Technology use cases
  - Businesses use cases
  - Invalid use cases
  - Container storage
  - Scaling applications
  - Integrating stateful and stateless apps
- Starting
  - Cluster runtimes
  - Cluster installation
  - Cluster login
  - Cluster config
  - Cluster debugging
  - Cluster software
  - First project
  - Application Components
    - Container images
    - Build configs
    - Deployment configs
    - Image stream
  - Deploying an app
  - Providing access to apps
  - Exposing application services
- Containers
  - Defining containers

  - OpenShift component interaction
  - OpenShift manages deployments
  - Kubernetes schedules applications
  - Docker creates containers
  - Linux isolates resources
  - Working with cluster
  - Listing kernel components
    * Mount namespace
    * UTS namespace
    * PID namespace
    * Memory namespace
    * Networking namespace
    * Summary
  - Cloud native apps
    * Testing app resiliency
  - Scaling applications
  - Application Health
    * Creating liveness probes
    * Creating readiness probes
  - Auto-scaling with metrics
    * Determining expected workloads
    * Installing OpenShift metrics
    * Understanding the metrics
    * Using pod metrics & autoscaling
    * Testing the autoscaling setup
    * Avoiding thrashing
  - Continuous integration & deployment
    * Container images are the centerpiece
    * Promoting images
    * CI/CD:1 Creating an environment - TODO: this needs rewrite
    * Deployment strategies
  - Stateful applications
    * Container storage
  - Handling permanency
  - Creating resources
    * Creating storage
    * Logging in as `kubeadmin`
    * Physical volume
    * Creating volumes
    * Creating claims
    * Modifying deployment
  - Volume mounts
- Stateful applications
  - Enabling a headless services
  - Application clustering with WildFly
  - Querying the OpenShift server
  - Verify the data replication
  - Other cases for direct pod access
  - Describing sticky sessions
  - Toggling sticky sessions
  - Limitations of cookies

    - Shutting down applications
        - Application grace period
        - Container lifecycle hooks
    - Stateful sets
        - Deterministic sequence
        - Examining a stateful set
        - Constant Network identity
        - Consistent persistent storage
        - The Stateful set limitations
        - Non-native stateful applications
        - Cleanup resource limits & quotas
- Operations & Security
    - Permissions vs Wild-West
        - Setting up authentication
        - Introduction to htpasswd
        - Creating htpasswd files
        - Changing the provider
    - Working with roles
        - Assigning user roles
        - Creating administrators user
        - Setting default roles
    - Limit ranges
        - Define resource limit ranges
    - Resource quotas
        - Creating compute quotas
        - Creating resource quotas
    - Working with quotas & limits
        - Quotas to existing applications
        - Changing quotas for deployed applications
    - Cgroups for managing resources
        - Overview of the cgroups
        - Identifying container cgroups
        - Confirming cgroups limits
- Networking
    - Managing the SDN
        - Configure application node network
        - Linking containers to host interfaces
        - Working with OVS
    - Routing application requests
    - Using the HAProxy service
        - Investigating the HAProxy service
        - HAProxy and request routing
    - Locating services with internal DNS
        - DNS resolution in pod network
    - Configure OpenShift SDN
        - Using the ovs-subnet
        - Using the ovs-multitenant
        - Creating advanced network designs
        - Enabling multi tenant plugin
        - Testing the multi-tenant plugin
- Security

  - SELinux core concepts
  - Working with SELinux labels
    - Applying labels with SELinux context
    - Enforcing SELinux with policies
    - Isolating pods with levels
    - Investigating pod security context
    - Examining the `MCS` levels
    - Managing Linux capabilities
    - Controlling the user ID
  - Scanning container images
  - Annotating images with security information

Introduction

Containers are changing how everyone in the IT industry does their job. Containers initially entered the scene on developers laptops helping them develop applications more quickly than they could with virtual machines, or by configuring a laptop's operating system. As containers became more common in development environments their use began to expand. Once limited to laptops and small development labs, containers worked their way into enterprise. Within a couple of years containers progressed to the point that they are powering massive production workloads like Github.

Container platform

A container platform is an application platform that uses containers to build deploy serve and orchestrate the application running inside it. OpenShift uses two primary tools to serve applications in containers a container runtime to create containers in Linux and an orchestration engine to manage a cluster of servers or also called nodes, these servers could be actual physical machines, virtual machines or IOT devices, running the containers.

Containers in OpenShift

A container runtime works on a Linux server to create and manage containers. For that to make sense we need to look at how containers function when they are running on a Linux system. In subsequent sections we will dig deeply into how containers isolate applications in OpenShift. To start, you can think of containers as discrete, portable scalable units for applications. Containers hold everything required for the application inside them to function. Each time a container is deployed it holds all the libraries and code needed to its application to function properly. Apps running inside a container can only access the resources in the container. The applications in the container are isolated from anything running in other containers or on the host. Five types of resources are isolated with containers.

- Mounted filesystems.
- Shared memory resources
- Hostname and domain names
- Network resources (IP addresses, MAC addresses, memory buffers)
- Process counters

We will investigate each one of those separately, throughout the next sections. In OpenShift the service that handles the creation and management of containers is docker. Docker is a large active open source project started by Docker, Inc is the company. The resources that docker uses to isolate processes in containers all exist as part of the Linux kernel. These resources include things like SELinux, Linux namespaces and control groups (cgroups), which will be covered later on in the sections in detail. In addition to making these resources much easier to use, docker has also added several features that have enhanced its popularity and growth. Here are some of the primary benefits of docker as container runtime:

- Portability - Earlier attempts at container formats were not portable between hosts running different operating systems. This container format is now standardized as part of the Open Container Initiative.
- Image reuse - any container image can be reused as the base for other container images.
- Application centric API - the API and command line tooling allow developers to quickly create update and delete containers. This is reflected in the API of the docker engine, as well as the API of Kubernetes.
- Ecosystem - Docker Inc, maintains a free public hosting environment for container images it now contains several hundred thousand images.

Orchestrating Containers

Although the docker engine manages containers by facilitating Linux kernel resources, it is limited to a single host operating system. Although a single server running containers is interesting, it is not a platform that you can use to create robust applications. To deploy highly available and scalable applications you have to be able to deploy applications containers across multiple servers. To orchestrate containers across multiple servers effectively you need to use a container orchestration engine, an application that manages a container runtime across a cluster, of hosts to provide a scalable application platform OpenShift uses Kubernetes as its container orchestration engine, Kubernetes is an application open source project that was started by Google, in 2015 it was donated to the Cloud Native Computing Foundation. Kubernetes employs a master/node architecture, Kubernetes master servers maintain the information about the server cluster and nodes run the actual application workloads. It is a great open source project. The community around it is quickly growing and incredibly active. It is consistently one of the most active projects on github. But to realize the full power of a container platform, it needs a few additional components. This is where OpenShift comes in, it uses docker/containerd and Kubernetes, as a starting point for its design. But to be a truly effective container platform it adds a few more tools to provide a better experience for users.

Examining the architecture

OpenShift uses Kubernetes master/node architecture as the base point. From there it expands to provide additional services that a good application platform needs to include out of the box.

Integrating container images

In a container platform like OpenShift container images are created when applications are deployed or updated, to be effective that container image have to be available quickly on all the application nodes in a cluster. To do this OpenShift includes an integrated image registry as part of its default configuration. An image registry is a central location that can serve container images to multiple locations. In OpenShift the integrated registry runs in a container. In addition to providing tightly integrated images access OpenShift works to make access to the applications more efficient.

Accessing applications

In Kubernetes containers are created on nodes using components called pods. There are some distinctions that we will discuss in more depth in next sections, but they are often similar. When an application consists of more than one pod, access to the application is managed through a component called a service. A service is a proxy that connects multiple pods and maps them to an IP address on one or more nodes in the cluster. IP addresses can be hard to manage and share especially when they are behind a fire wall. OpenShift helps to solve this problem by providing an integrated routing layer. The routing layer is a software load balancer. When an application is deployed in OpenShift a DNS entry is created for it automatically. That DNS record is added to the load balancer and the load balancer interfaces with the Kubernetes service to efficiently handle connections between the deployed applications and its users. With applications running in pods across multiple nodes and management requires coming from the master node there a lot of communication between servers in an OpenShift cluster. You need to make sure that traffic is properly encrypted and can be separated when needed.

Handling network traffic

OpenShift uses a software defined networking (SDN) Solution to encrypt and shape network traffic in a cluster. OpenShift SDN, solution that uses `Open vSwitch`. Other SDN solutions are also supported, this will be examined in depth in future sections. Now that you have a good idea of how OpenShift is designed let us look at the life cycle of an application in an OpenShift cluster.

Examining an application

OpenShift has workflows that are designed to help you manage you applications through all phases of its lifecycle - build, deploy, upgrade and retirement.

Building an application

The primary way to build application is to use a builder image. This process is the default workflow in OpenShift and its what you will use in next section to deploy your first application in OpenShift. A builder image is a special container image that includes applications and libraries needed for an application in a given language. In next section we will deploy a PHP web application. The builder image you will use for your first deployment includes the Apache web server and the PHP language libraries - things needs to run this type of application. The build process takes the source code for an application and combines it with the builder image to create a custom application image for the application. The custom application image is stored in the integrated registry, where it is ready to be deployed and served to the application users.

Deploying and serving applications

In the default workflow in OpenShift applications deployment is automatically triggered after the container image is build and available. The deployment process takes a newly created application image and deploys it on one or more nodes. In addition the application pods a service is also created, along with a DNS route in the routing layer. Users are able to access the newly created application through the routing layer after all components have been deployed. App upgrades use the same workflow, when an upgrade is triggered a new container image is created and the new application version is deployed. Multiple upgrade processes are available, we will check them out in future sections.

That is how OpenShift works at a high level we will dig much much deeper into all of these components and mechanisms over the course of future sections. Now that we are armed with the knowledge of OpenShift let us talk about some of the things container platforms are good and not so good at doing

Use case for platforms

The technology in OpenShift is pretty cool, but unless you can tie a new technology to some sort of benefit to your mission it is hard to justify investigating it, in this section, we will take a look at some of the benefits of OpenShift can provide.

Technology use cases

If you stop and think about it for a minute, you can hand the major innovations in IT on a timeline of people seeking more efficient process isolation. Starting with mainframes, we were able to isolate applications more effectively with the client-server model and the x86 revolution. That was followed by the virtualization revolution. Multiple virtual machines can run on a single physical server. This give administrators better density in their datacenters while still isolating processes from each other. With virtual machines each process was isolated in its own virtual machine. Because each virtual machine has a full operating system and a full kernel, must have all the filesystem required for full operating system. That also means it must be patched managed and treated like traditional infrastructure. Containers are the next step in this evolution. An application container holds everything the application needs to run - the source code, the libraries, and the configurations and information about connecting to shared data sources.

What containers do not contain is equally important. Unlike virtual machines, containers are all run on a single, shared kernel. To isolate the application containers use components inside the kernel. Because containers do not have to include a full kernel to serve their application, along with all the dependencies of an operating system, they tend to be much smaller than an equivalent virtual machine. For example whereas a typical virtual machine starts out with a 10GB or larger disk, the container image could be as small as 100MB. Being smaller comes with a couple of advantages. First portability is enhanced. Moving a 100MB from one server to another, is much easier than doing the same for multi gigabyte images. Second because starting a container does not include booting up an entire kernel, the startup process is much faster. Starting a container is typically measured in milliseconds as opposed to seconds or minutes for virtual machines.

Businesses use cases

Modern business solutions must include time or resource savings as part of their design. Solutions today have to be able to use human and computer resource more efficiently than in the past. Containers ability to enable both types of savings is one of the major reasons they have exploded on the scene the way they have.. If you compare a server that is using virtual machines to isolate processes to one that is using containers to do the same thing, you will notice a few key differences:

- Containers consume server resources more effectively. Because there is a single shared kernel for all containers on a host, instead of multiple virtualized kernels, in a virtual machine, more of the servers' resources are used to serve applications instead of for platform overhead.
- App density increases with containers. Because the basic unit used to deploy applications is much smaller than the unit for virtual machines, more applications can fit per server. This means more applications require fewer servers to run.

Invalid use cases

An ever increasing number of workloads are good fit for containers. The container revolution started with pure web applications but now includes command line tools, desktop tools and even relation databases. Even with the massive growth of use cases for containers in some situations they are not the answer. If you have a complex legacy application, be careful when deciding to break it down and convert it to a series of containers. If an application will be around for 18 months and it will take 9 months of work to properly containerized it you may want to leave it where it is. Containers solution began in the enterprise IT world. They are designed to work with most enterprise grade storage systems and network solutions, but they do not work with all of them easily. Some applications are always going to be very large, very resource intensive monolithic applications, examples are software used to run HR departments and some very large relation databases. If a single application will take up multiple servers on its own running it in a container that wants to share resources with other applications on a server does not make any sense

Container storage

Containers are a revolutionary technology but they can not do everything. Storage is an area where containers need to be paired with another solution to deploy production-ready applications. This is because the storage created when a container is deployed is ephemeral. If a container is destroyed or replaced the storage from inside that container is not reused. This is by design to allow containers to be stateless by default. If something goes bad, a container can be removed from your environment completely and new one can be stood up in its place, instantly The idea of a stateless application container is great, but somewhere in your application usually in multiple places data needs to be shared across multiple containers and state needs to be preserved. Here are some examples of these situations:

- Shared data that needs to be available across multiple containers, like uploaded image, for a web application
- Use state information in a complex application which lets users pick up where they leave off during a long running transaction.
- Information that is stored in relational or non-relational databases

In all of these situations, and many others, you need to have persistent storage available to your containers. This storage should be defined as part of your application deployment and should be available from all the nodes in your OpenShift cluster, luckily OpenShift has multiple ways to solve this problem. In future sections we will configure external network storage service, you will then configure it to interact with OpenShift so applications can dynamically allocate and take advantage of its persistent storage volumes. When you are able to effectively integrate shared storage into your application containers you can think about scalability in new ways.

Scaling applications

For stateless application, scaling up and down is straightforward, because there are o dependencies other than what is in the application container and because the transactions happening in the container are atomic by design all you need to do to scale a stateless application is to deploy more instance of it and load balance them together. To make this process even easier OpenShift proxies the connection to each application through a built in load balancer - HAProxy, This allows applications to scale up and down with no change, in how users connect to the application. If your application are stateful meaning they need to store or retrieve shared data, such as a database or data that a user has uploaded then you need to be able to provide persistent storage for them. This storage needs to automatically scale up and down with your application, in OpenShift. For stateful applications persistent storage is a key component that must be tightly integrated into your design. At the end of the day stateful pods are how users get data in and out of your application

Integrating stateful and stateless apps

As you begin separating traditional monolithic apps into smaller services that work effectively in containers you will Begin to view your data needs in a different way. This process is often referred to as designing apps as microservices. For any app you have services that you need to be stateful and others that are stateless, for example the service that provides static web content can be stateless, whereas the service that processes user authentication needs to be able to write information to persistent storage. These services all go together to form your app. Because each service runs in its own container the services can be scaled up and down independently. Instead of having to scale up your entire codebase with containers you can only scale the services in your app that need to process additional workloads. Additionally because only the containers that need access to persistent storage have it, the data going into your container is more secure. That brings us to the end of our initial walkthrough. The benefits provided by OpenShift save time for human and use server resources more efficiently. Additionally the nature of how containers work provides improved scalability and deployment speed versus virtual machines. This all goes together to provide an incredibly powerful app platform that you will work with for the rest of this read.

Starting

There are three ways to interact with OpenShift the command line, the web interface and the REST API. This chapter focuses on deploying apps using the command line, because the command line exposes more of the process that is used to create containerized app in OpenShift. In other sections the examples may use the web interface or even the API. Our intention is to give you the most real world examples of using OpenShift. We want to show you the best tools to get the job done. We will also try our best not to make you repeat yourself. Almost every action in OpenShift can be performed using all three access methods. If something is limited we will do our best to let you know. But we want you to get the best experience possible from using OpenShift. With that said in this section we are going to repeat ourselves, but for a good reason.

The most common task in OpenShift is deploying an app. Because this is the most common task we need to introduce you to it as early as practical using both the command line and the web interface. So place bear with us. This section may seem a little repetitive.

Cluster runtimes

Before we can start using OpenShift you have to deploy it. There a few options, to install OpenShift locally on your machine or on cloud providers. We are going to stop at tools like `Minishift` or `RedHat's CRC`.

Logging in OpenShift must be done, as every action requires authentication. This allows every action to be governed by the security and access rules set up for all users in an OpenShift cluster. We will discuss the various methods of managing authentication in next section, but by default your OpenShift cluster initial configuration is set to allow any user and password. The allow all identity provider creates a user account the first time a user logs in. Each user name is unique and the password can be anything except an empty field. This configuration is safe and recommended only for lab and development OpenShift instances like the one we are setting up.

Using the `oc` command line tool, it is a front facing user level program which is used to interact with the REST server of the running OpenShift cluster, in this case the actual OpenShift server is backed by the Kubernetes REST API server under the hood, but those are implementation details, OpenShift builds on top of the Kubernetes REST API server to bring us more flexibility and robustness. What you need to remember is that you must first use the login command to perform the login action, before any other command can be execute with the `oc` tool

This is done using the following command `oc login -u dev -p dev https://ocp-1.192.168.122.100.nip.io:8443`. What this does is sets the user and password to `dev`, and points the `oc` tool to a valid running cluster REST API server, those 3 fields are mandatory otherwise the tool would not know who to authenticate, or how and where to for that matter.

```
# to obtain the list of configured credentials on the cluster, locally, you
   can use the following command, this will
# list every user that is configured along with the password in a ready to
   use login command to facilitate easier login
crc console --credentials
minishift console --credentials
```

Cluster installation

To install a OpenShift cluster we have really two major options - `minishift` and `crc`. The latter of which is the more modern one which we will be using, this is a distribution provided by RedHat directly, and `crc` supports more modern versions of OpenShift, those would be any version =>4.xx.

The runtime first needs a container runtime on your host machine, that is usually docker, podman or containerd, one of those need to be installed in order for you to be able to install `minishift`. However for the `crc` runtime the process is a bit different, while the `minishift` uses native container technologies to mimic the OpenShift cluster runtime, the `crc` product uses virtualization, meaning that you need to have the virtualization on your host enabled. For any modern windows, there is nothing more required here, as Hyper-V would be used to run the `crc` runtime, however for linux systems, you are required to install `KVM`. For linux systems there are a couple of more dependencies that need to be installed in order for you to be able to run the `crc` runtime:

- `virtsh` - that is a general purpose virtual network daemon, that one is used to bridge the connection between the `crc` runtime running in the virtual machine and your host
- `KVM` - as already stated that is virtualization platform that is required core component to run the `crc` runtime, once installed proceed further

After having all dependencies installed on your platform simply download the `crc` runtime from the official RedHat website, that can be found here - `https://www.redhat.com/en/blog/codeready-containers`. Once you visit this page follow the instructions to setup the orchestration runtime.

Cluster login

To enable ssh login into the virtual machine itself, if using the `crc` distribution, which is running the OpenShift cluster, add the following into your ssh config file located in `~/.ssh/config`. This will allow you to perform a simple ssh login command as such `ssh crc`. Also make sure that the certificate files specified in the configuration below exist, in the specified locations, they should for a local installation of `crc`, but if another local OpenShift cluster distribution is used, replace the paths to point to the locally installed certificate credentials on your machine.

The `minishift` distribution, does not require any special ssh setup, since it provides a command line option to directly login into the cluster using `minishift ssh`. The set of examples below will be using `crc` and the OpenShift version 4.xx which is supported by `crc` version

Creating the projects can now be done using the tool. In OpenShift projects are the fundamental way apps are organized. Projects let users collect their apps into logical groups. They also serve other useful roles around security that we will discuss in future sections. For now though think of a project as a collection of related apps. You will create your first project and then use it to house a handful of apps that you will deploy modify, redeploy and do all sorts of things to over the course of the next few sections.

```
The default project and working with multiple projects - the oc tool's default action is
 to execute the command you run using the current working project. If you create a new
project it automatically becomes your working project. The oc project command changes the
current working project from one to another. To specify a command to be execute against a
specific project regardless of your current working project use the -n parameter with the
 project name you want the command to run against. This is a helpful option when you are
writing scripts that use oc and act on multiple projects
```

There is a quick and dirty way to configure your local ssh client to work with the cluster node, and allow you to very quickly login into the node, by simply providing the name of the host, as shown below in the ssh/config file, this will send the correct identity material form your host machine to the cluster, and allow you to directly login into the cluster node itself, this is very useful, as we will need to interact with the cluster node to configure it in the future.

```
Host crc
    HostName 127.0.0.1
    Port 2222
    User core
    HashKnownHosts yes
    StrictHostKeyChecking no
    IdentityFile ~/.crc/machines/crc/id_ed25519
    UserKnownHostsFile /dev/null
```

The `minishift` has a direct argument that can be used on the command line which is much simpler, no additional configuration is required there - `minishift ssh`. This will do the same as the configuration above, and does not need any manual intervention from the user. You will land directly into the cluster node.

Cluster config

There are certain amount of configuration options to be configured when or before starting the cluster, this is to ensure, that the cluster is going to perform well, or even be able to start, most of the default configurations are okay, however often enough the default resources that `crc` or `minishift` are using for `cpu` and memory resources are not good enough, preventing the clusters from working well or even starting in the first place. This is very important since the more features you enable the more resources you will need. There is no good way to calculate the exact resources that are going to be needed for a local deployment of these options. But a good estimate is that at the very least at least 16 GB or memory for `minishift` and 24 GB of memory for `crc` are required, as far as CPU configuration goes, the minimal requirements should start from 8 cores, for `minishift` and 10 for `crc`. Below are example snippets to set the configuration of the clusters globally, these require the cluster to be restarted if set while the cluster is running.

```
minishift config set --global disk-size 100GB # increase the base size of the
    node, by default that would be 30GB
minishift config set --global memory 16GB # make sure that memory resources
   are not constrained too much
minishift config set --global cpus 8 # enable more cpu cores to allow for
   better performance
```

```
crc config set disk-size 100 # increase the base size of the node, by default
    that would be 30GB
crc config set memory 24512 # make sure that memory resources are not
   constrained too much
crc config set cpus 10 # enable more cpu cores to allow for better
   performance
```

There are other configuration options which will be looked into, which enable different features on the clusters, such as monitoring further down in the sections. There is another level of configuration that can be enabled on the cluster, that is not actually directly related to the cluster tool, that is the patch approach. The patch approach enables OpenShift features that are off by default, meaning that those, unlike the configuration options above, are not strictly related or specific to the clustering solution we are using - `crc` or `minishift`. Rather they are native OpenShift features that we can turn on or off, by patching the default `clusterversion` config object

```
# here is one such example, which actually removes an entry from the
   overrides section of the clusterversion/version
```

```
# object, by doing that we modify the default out of the box behavior,
   whatever the overrides section were enabling or
# disabling, you can run oc describe clusterversion/version to see what this
   object contains by default before doing
# any modifications to it. This command is using the so called json-path, to
   remove the entry from the overrides array
# and more precisely the 0-th index entry from the spec.overrides[]
$ oc patch clusterversion/version --type='json' -p '[{"op":"remove", "path
   ":"/spec/overrides/0"}]'
```

In future sections you will also see that the cluster node itself, contains several files under /etc which can be used to configure the cluster in more direct and manual way by directly editing these files, you will be able to modify the behavior of the cluster, directly. The files stored under /etc in the cluster provide more flexibility but should not be messed with too much unless you are aware of the changes you are going to be doing.

Cluster debugging

In case you would like to ever access the OpenShift cluster API, form the node itself, using the system user, you can do the following as shown below.

```
# this variable is used by the oc tool, to look for a config file to use,
   what this file contains is the certificate
# material that can be used to login into the cluster, without providing any
   credentials, the idea is that we will be
# logging in wit ha special user called system:node:crc
export KUBECONFIG=/var/lib/kubelet/kubeconfig

# now running any oc command will actually run using the config above,
   meaning that we will be executing command in the
# context of the system user, for crc that user is system:node:crc
$ oc <command>
```

The same can be used when working with `minikube`, however the configuration does not live in a virtual machine just like for `crc`, but rather the local host `$HOME/.kube` config folder is used, therefore the file in question is under - `$HOME/.kube/config`. Again run the export expression in your shell, on the host, and you can use `oc to login` as a system user. If your host is Windows, you can use SET instead, while the general approach remains the same

To allow you to login as admin and have access to the default password of the cluster root user, one can also take advantage of the enable emergency login config, this is not always needed, but good to know. By default you can always assume sudo rights in the host by doing `sudo -i` in the shell. That would not require password by default

```
# make sure to configure the cluster first to enable emergency login, this
   will generate a passwd file in the .crc
# directory on your host machine, this file will contain the password to the
   core user in the crc cluster node, which is
# quite useful when you wish to run some sudo enabled commands in the node,
   after you run this config, restart the cluster
# after the cluster is started search for the file - $HOME/.crc/machines/crc/
   passwd, and fetch the password for the core user
crc config set enable-emergency-login true
```

Cluster software

There are ways to install software on our cluster, using the dnf or yum binaries, which are the package manages for the underlying linux distribution that the OpenShift cluster node is using which is CentOS

```
$ ssh crc # here are few preliminary steps, first login into the cluster node
$ sudo -s # login as root user, this will make subsequent commands easier to
   execute directly
$ mount -o remount,rw /usr # remount the /usr file system as read/write, it
   is read-only by default

# modify the dnf config file to look like this, this will remove some hard
   guard set rules, which might interfere with
# installing software on the cluster
$ vi /etc/dnf/dnf.conf
> [main]
> gpgcheck=0
> diskspacecheck=0
> clean_requirements_on_remove=True
> best=True
> skip_if_unavailable=True

$ subscription-manager register # will prompt you to enter your RedHat
   credentials to register this node with RedHat
$ subscription-manager refresh # refresh the indexes and allow us to interact
    with the RedHat registries from the node
$ dnf clean all && dnf repolist -C # clean any left over artifacts, and also
   update the repository list of dnf manager
$ dnf -y install httpd httpd-tools iptables-services \
        nfs-utils go gcc python3 python-pip --nogpgcheck # install some
           utility applications to the master cluster node
```

First project

To create a project you need to run the `oc new-project` command and provide a project name. For the first project use `image-uploader` as the project name

As already mentioned the project term in OpenShift is actually what Kubernetes refers to as the so called namespaces, the OpenShift Projects are a way to separate different application contexts more effectively and also allow for a more general permission control over these contexts, by restricting which namespace or a project a user has access to easily

```
# to create a new project just run the following, that would immediately
   change the context to that project, as well
$ oc new-project image-uploader --display-name='Image Uploader Project'
```

Application Components

Apps in OpenShift are not monolithic structures, they consist of a number of different components in a project that will work together to deploy update and maintain your app through its lifecycle. Those components are as follows

- Container images

- Image streams
- App pods
- Build configs
- Deployment configs
- Deployments
- Services

Container images

Each app deployment in OpenShift creates a custom container image to serve your app. This image is created using the app source code and custom base image called builder image. For example the PHP builder image contains the Apache web server and the core PHP language libraries. The image build process takes the builder image you choose integrates your source code and creates the custom container image that will be used for the app deployment. Once created all the container images along with all the builder images are stored in OpenShift integrated container registry which we discussed in first section. The component that controls the creation of your app containers is the build config.

Build configs

A build config contains all the information needed to build an app using its source code. This includes all the information required to build the app container image.

- URL for the app source code
- Name of the builder image to use
- Name of the app container image that is created
- Events that can trigger a new build to occur

After the build config does its job, it triggers the deployment config that is created for your newly created app.

Deployment configs

If an app is never deployed it can never do its job. The job of deploying and upgrading the app is handled by the deployment config component. Deployment configs track several pieces of information about an app.

- Currently deployed version of the app
- Number of replicas to maintain for the app
- Trigger events that can trigger a redeployment. By default configuration changes to the deployment or changes to the container image trigger an automatic app redeployment
- Upgrade strategy app-cli uses the default rolling upgrade strategy
- App deployments

A key feature of app running in OpenShift is that they are horizontally scalable. This concept is represented in the deployment config by the number of replicas. The number of replicas specified in a deployment config is passed into a Kubernetes object called a replication controller. This is a special type of Kubernetes pod that allows for multiple replicas - copies of the app pod to be kept running at all time. All pods in OpenShift are deployed by replication controllers by default. Another feature that is managed by a deployment config is how apps upgrades can be fully automated. Each deployment for an app is monitored and available to the deployment config component using deployments.

In OpenShift a pod can exist in one of five phases at any given time in its lifecycle. These phases are described in detail in the Kubernetes Documentation. The following is a brief summary of the five pod phases.

- `Pending` - the pod has been accepted by OpenShift but its is not yet schedule on one of the app nodes.
- `Running` - the pod is scheduled on a node and is confirmed to be up and running.
- `Succeeded` - all containers in a pod have terminated successfully and wont be restarted
- `Unknown` - something has gone wrong and OpenShift can not obtain a more accurate status for the pod.

Failed and Succeeded are considered terminal states for a pod in its lifecycle. Once a pod reaches one of these states it wont be restarted. You can see the current phase for each pod in a project by running the `oc get pods` command Pod lifecycle will become important when you begin creating project quotas.

Each time a new version of an app is created by its build config, a new deployment is created and tracked by the deployment config. A deployment represents a unique version of an app. Each deployment references a version of the app image that was created and creates the replication controller to create and maintain the pod to serve the app. New deployments can be created automatically in OpenShift by managing how apps are upgraded which is also tracked by the deployment config.

The default app upgrade method in OpenShift is to perform a rolling upgrade rolling upgrades create new versions of an app allowing new connections to the app to access only the new version. As traffic increases to the new deployment the pods for the old deployment are removed from the system.

New app deployments can be automatically triggered by events such as configuration changes to your app, or a new version of a container image being available. These sorts of trigger events are monitored by image streams in OpenShift.

Image stream

Image stream are used to automate actions in OpenShift. The consist of links to one or more container images. Using image streams you can monitor apps and trigger new deployments when their components are updated.

Deploying an app

Apps are deployed using the `oc new-app` command. When you run this command to deploy the `image uploader` app, into the `image-uploader` project. You need to provide three prices of information.

- The type of the image stream you want to use - OpenShift ships with multiple container images called builder images, that you can use as a starting point for apps.
- A name for your app - in this example use app-cli because this version of your app will be deployed from the command line.
- The location of your app source code - OpenShift will take the source code and combine it with the PHP builder image to create a custom container image for your app deployment

```
# here is a new app deployment from a github hosting
$ oc new-app --image-stream=php --code=https://github.com/OpenShiftInAction/
   image-uploader.git --name=app-cli
```

After you run the `oc new-app` command you will see a long list of output, as shown above. This is OpenShift building the image out of all the components needed to make your app work properly. Now if you visit the web console you will be able to browse the project structure as well, and you will also see that there are quite a few new objects created for the new app in the new project. With the triggering of new `oc new-app` command various new objects are created in the cluster for the current project, this is usually not the way we would do this in the real world, each of those new objects will be manually defined in manifest files by the developers where the properties of these objects can be fine tuned. But for the sake of demonstration and ease of use OpenShift provides the users with quick and dirty ways to deploy apps, this is very useful for development purposes where we just want to put our app into use, and avoid the hassle of manual configuration of OpenShift objects.

```
# list some of the objects which would have been automatically created by the
    OpenShift environment
$ oc get services && oc get pods && oc get deployments

# here is what the output might look like, there are multiple pods that were
   created, you may notice that there was a
# pod called build - which is basically spun up to build the image from the
   source, and then it is uploading that
# image into the internal OpenShift registry # and deployed as an actual pod
   which is in Running state. There is
# also a new service created for our app.
NAME             TYPE      CLUSTER-IP    EXTERNAL-IP PORT(S)           AGE
app-cli          ClusterIP 172.30.51.80 <none>      8080/TCP,8443/TCP 55s

NAME             READY     STATUS        RESTARTS    AGE
app-cli-1-build 0/1        Completed     0           54s
app-cli-1-vsk5q 1/1        Running       0           12s

NAME             REVISION  DESIRED       CURRENT     TRIGGERED BY
app-cli          1         1             1           config,image(app-cli:
   latest)
```

Providing access to apps

In future sections we will explore multiple ways to force OpenShift to redeploy app pods. In the course of a normal day this happens all the time, for any number of reasons, you are scaling apps up and down, apps pods stop responding correctly, nodes are rebooted or have issues, human error, and so on. Although pods may come and go there needs to be a consistent presence for your app in OpenShift. That is what a service does. A service uses labels applied to application pods when they are created to keep track of all pods associated with a given app. This allows a service to act as an internal proxy for your app. You can see information about the service for app-cli by running the `oc describe svc/app-cli command`.

```
$ oc describe svc/app-cli

Name:                      app-cli
Namespace:                 image-uploader
Labels:                    app=app-cli
                           app.kubernetes.io/component=app-cli
                           app.kubernetes.io/instance=app-cli
                           app.kubernetes.io/name=php
Annotations:               openshift.io/generated-by: OpenShiftNewApp
Selector:                  deployment=app-cli
Type:                      ClusterIP
IP Family Policy:          SingleStack
IP Families:               IPv4
IP:                        10.217.4.162
IPs:                       10.217.4.162
Port:                      8080-tcp  8080/TCP
TargetPort:                8080/TCP
Endpoints:
Port:                      8443-tcp  8443/TCP
```

```
TargetPort:                 8443/TCP
Endpoints:
Session Affinity:           None
Internal Traffic Policy:    Cluster
```

Now note that there are the fields which are IP addresses, these are the IP addresses that each service gets, these are the cluster virtual IP addresses that are only routable within the OpenShift cluster. Other information that is maintained includes the IP address of the service and the TCP ports to connect to the in the pod.

`Most components in OpenShift have a shorthand that can be used on the command line to save time and avoid misspelled components names. The previous command uses svc/app-cli to get information about the service for the app-cli app. Build configs can be accessed with the bc/<app-name>`

Services provide a consistent gateway into your app deployment, but the IP addresses of a service is available only in your OpenShift cluster, to connect users to your app and make DNS work properly you need one or more app components Next you will create a route to expose app-cli externally from your OpenShift cluster

Exposing application services

When you install your OpenShift cluster, one of the services that is created is the HAProxy service running in a container on OpenShift, the HAProxy is an open source software load balancer, we will look at this service in depth in next sections,. To create a route for the app-cli run the following command -

```
$ oc expose svc/app-cli
route.route.openshift.io/app-cli exposed
```

As we discussed earlier, OpenShift uses projects to organize applications. An application project is included in the URL that is generated when you create an application route. Each application UR takes the following format - `<application-name>-<project-name>-.<cluster-app.domain>`. This is actually the default format, coming from kubernetes, but slightly modified by OpenShift to include the name of the project, in Kubernetes in place of the project name in that format is actually the namespace for the app, in OpenShift the projects are actually implemented internally as kubernetes namespaces, but very much enhanced.

When you did deploy OpenShift in previous sections you specified the application domain, by default all application in OpenShift are served using the HTTP protocol when you pull all this together the URL for app-cli should be as follows - `http://app-cli-image-uploader.<cluster-app.domain>`. You can get more information about the route you just created by running the `oc describe route/app-cli` command

```
$ oc describe route/app-cli

Name:                   app-cli
Namespace:              image-uploader
Created:                <time> ago
Labels:                 app=app-cli
                        app.kubernetes.io/component=app-cli
                        app.kubernetes.io/instance=app-cli
                        app.kubernetes.io/name=php
Annotations:            openshift.io/host.generated=true
Requested Host:         app-cli-image-uploader.apps-crc.testing
                           exposed on router default (host router-default.
                              apps-crc.testing) <time> ago
Path:                   <none>
```

```
TLS Termination:        <none>
Insecure Policy:        <none>
Endpoint Port:          8080-tcp

Service:        app-cli
Weight:         100 (100%)
Endpoints:      <none>
```

The output tells you that the host configuration added to the HAProxy the service associated with the route and the endpoints for the service, to connect to when handling requests for the route, how that we have created the route to you application go ahead and verify that by visiting the IP address in a web browser.

Focusing on the component that deploy and deliver the app-cli application, you can see the relationship between the service, the newly created route, and the end users. We will cover this in more depth in next sections, but in summary the route is tied to the app-cli service, users access the application pod through the route. This chapter is about relationships. In OpenShift multiple components work in concert to build deploy and manage application. We spend the rest of the discussing the different aspects of these relationships in depth, that fundamental knowledge of how container platforms operate is incredibly valuable.

Containers

In the previous sections you deployed your first app, in OpenShift in this chapter we will look deeper into your OpenShift cluster and investigate how these containers isolate their processes on the application node. Knowledge of how containers work in a platform like OpenShift is some of the most powerful information in IT right now. This fundamental understanding of how a container actually works as part of the Linux kernel and server informs how systems are designed and how issues are analyzed when they inevitable occur. This is a challenging section, not because of a lot of configuration and making complex changes, but because we are talking about the fundamental layers of abstractions that make a container a container in the modern kernel world.

Defining containers

You can find five different container experts and ask them to define what a container is and you are likely to get five different answers, the following are some our personal favorites all of which are correct from a certain perspective.

- A transportable unit to move apps around. This is a typical developer answer
- A fancy linux process
- A more effective way to isolate processes on a linux system, this is a more operations centered answer.

What we need to untangle is the fact that they are all correct, depending on your point of view. In section 1, we talked about how OpenShift uses Kubernetes and docker to orchestrate and deploy apps in a container in your cluster. But we have not talked much about which application component is created by each of these services, before we move forward it is important for you to understand these responsibilities as you begin interacting with application components directly.

OpenShift component interaction

When you deploy and application in OpenShift the request starts in the OpenShift API server. To really understand how containers isolate the process within them we need to take a more detailed look at how these services work together to deploy your application. The relationship between OpenShift Kubernetes docker and ultimately the Linux kernel is a chain of dependencies. When you deploy an application in OpenShift the process starts with the OpenShift services.

OpenShift manages deployments

Deploying application begin with application components that are unique to OpenShift the process is as follows:

1. OpenShift creates a custom container image using your source code and the builder image template you specified.
2. This image is uploaded to the OpenShift container image registry
3. OpenShift creates a build config to document how your – is built. This includes which image was created the builder image used the location of the source code and other information.
4. OpenShift creates a deployment config to control deployments and deploy and update your application. Information in deployment configs includes the number replicas the upgrade method, and application specific variables and mounted volumes.
5. OpenShift creates a deployment which represents a single deployed version of an application. Each unique application deployment is associated with your application deployment config component.
6. The OpenShift internal load balancer is updated with an entry for the DNS record for the application. This entry will be linked to a component that is created by Kubernetes which we will get to shortly.
7. OpenShift creates an image stream component, in OpenShift an image stream monitors the builder image, deployment config, and other components for changes, if a change is detected image streams can trigger application re-deployments to reflect changes

The build config creates an application specific custom container image using the specified builder image and source code, that image is stored in the OpenShift image registry. The deployment config component creates an application deployment that is unique for each version of the app. The image stream is created and monitors for changing to the deployment config and related images in the internal registry. The DNS route is also created and will be linked to the Kubernetes object

Kubernetes schedules applications

Kubernetes is the orchestration engine, at the heart of OpenShift, in many ways an OpenShift cluster is a kubernetes cluster. When you initially deploy app-cli, Kubernetes created several application components.

- Replication controller - scales the application as needed in Kubernetes. This component also ensures that the desired number of replicas in the deployment config is maintained at all times.
- Service - Exposes the application . A kubernetes service is a single IP address that is used to access all the active pods for an application deployment. When you scale an application up or down the number of pods changes, but they are all accessed through a single service proxy object.
- Pods - represent the smallest scalable unit in OpenShift.

The replication controller dictates how many pods are created for an initial application deployment is linked to the OpenShift deployment component. Also linked to the pod components is a Kubernetes service. The service represents all the pods deployed by a replication controller. It provides a single IP address in OpenShift to access your application as it scaled up and down on different nodes in your cluster. The service in the internal IP address that is referenced in the route created in the OpenShift load balancer.

Docker creates containers

Docker is a container runtime. A container runtime is the application on a server that creates, maintains and removes containers. A container runtime can act as a stand alone tool on a laptop or a single server, but it is at its most powerful when being orchestrated across a cluster by a tool like kubernetes. Kubernetes controls docker to create containers that house the app. These containers use the custom base image as the starting point for the files that are visible to application in the container. Finally the docker container is associated with the Kubernetes pod. To isolate the libraries and application in the container image along with other server resources docker uses Linux kernel components. These kernel level resources are the components that isolate the application in your container from everything else on the application node. Let us check them out

Linux isolates resources

We are down to the core of what makes a container a container in OpenShift , and Linux. Docker uses three Linux kernel components to isolate the application running in containers it creates and limit their access to resources on the host machine or cluster node in our case.

- Linux namespaces - provide isolation for the resources running in the container. Although the term is the same this is a different concept than Kubernetes namespaces, which are roughly analogous to n OpenShift project. We will discuss these in more depth in next sections. For the sake of brevity in this section when we reference namespaces, we are talking about Linux namespaces
- Control groups - Provide maximum guaranteed access to limits for CPU and memory on the app node, or cluster node.
- SELinux contexts - prevents the container application from improperly accessing resources on the host or in other containers. An SELinux context is a unique label that is applied to a container resources on the application node. This unique label prevents the container from accessing anything that does not have a matching label on the host.

The docker daemon creates these kernel resources dynamically when the container is created, these resources are associated with the application that are launched for the corresponding container your application is now running in a container. Apps in OpenShift are run and associated with these kernel components they provide the isolation that you see from inside a container in upcoming sections, we will discus how you can investigate a container from the application node. From the point of view of being able inside the container, an application only has the resources allowed and allocated to it.

Working with cluster

To first make sure we can extract the resources from the cluster, we have to be able to login into the cluster, what that means, is that we would like to be able to interact with the virtual machine that is being created locally when you stood up your OpenShift cluster. This means we have to login into the cluster, this is done by using ssh and the following command template - `ssh -i <path-to-private-key> -o <options> -p 2222 core@127.0.0.1`. The ssh command here is setting up a secure connection to the cluster, by making sure it is using a valid private key that the cluster trusts, and the port and host are by default 2222, and 127.0.0.1 for the local cluster, however the same ssh command can be used to login into a provided cluster. As long as you have a locally setup private key which is trusted by the cluster's ssh agent

```
# this will allow you to login into the crc virtual machine, which represents
    the actual single cluster node that is
# being simulated locally, refer to the beginning of this document to setup
   permanent ssh host config for crc
$ ssh -i ~/.crc/machines/crc/id_ed25519 -o StrictHostKeyChecking=no -o
   UserKnownHostsFile=/dev/null -p 2222 core@127.0.0.1
```

To extract the process id of the running container we have to do a few more things, first we can list all running processes on the cluster by doing the following command

```
# that will list all pods along side their ids, names, and further provide
   more information about the running pods, the
# container and image information and more
$ sudo crictl ps | head

# this is a sample of the data that you might see from the crictl ps above,
   we have to look for the app-cli container
# information from this output, and get the container id of that row in the
   table, note that the table is abridged
# version, more columns actually are provided by the output of crictl, like
   pod-id, creation date and other...
CONTAINER       IMAGE
67fb60f6d592d   quay.io/crcont/routes-controller@sha256:9
   a66245c7669a8741da0db9be13cf565548b0fa93ca97ae1db6b8400d726aa71
7c2083341d04d   image-registry.openshift-image-registry.svc:5000/image-
   uploader/app-cli@sha256:85
   e47f7e1eefedd6aa1c09c494fbe98eeefe22aa8945ce7665568ee3175a74e6
06e7b456c7f39   quay.io/openshift-release-dev/ocp-v4.0-art-dev@sha256:
   c34a7d4a5ba7d78debe6b3961498a19e7c2416b2a8a2c10e5086a02934b4e956
c57498c3c0319   quay.io/openshift-release-dev/ocp-v4.0-art-dev@sha256:
   c34a7d4a5ba7d78debe6b3961498a19e7c2416b2a8a2c10e5086a02934b4e956
290e91fe31ae4   quay.io/openshift-release-dev/ocp-v4.0-art-dev@sha256:
   c34a7d4a5ba7d78debe6b3961498a19e7c2416b2a8a2c10e5086a02934b4e956
```

Now that we can see the container is of the app-cli pod/container we can use that to inspect it and obtain the PID of it, the command below will show the details of the container, we would like to find the PID which must be under the “info” property

```
# inspect the container id, in this case this id corresponds to the app-cli,
   see the table above, would provide us with
# a super details spec output/dump of the container's definition
$ sudo crictl inspect 7c2083341d04d

# here is the output of the inspect, the output is again abridged, it is
   quite big, but we care about the first few
# rows, most notably, the PID property, which we can see here, is 7825
{
  "info": {
    "checkpointedAt": "0001-01-01T00:00:00Z",
    "pid": 7825,
    "privileged": false,
    "restored": false,
    "runtimeSpec": { .... }
    }
  ....
}
```

Listing kernel components

Armed with the process id of the current app-cli we can begin to analyze how containers isolate process resources with Linux namespaces. Earlier in this section we discussed how kernel namespaces are used to isolate the application in a container from the other processes on the host. Docker creates a unique set of namespaces to isolate the resources in each container looking again the application is linked to the namespaces because they are unique for each container. Cgroups and SELinux are both configured to include information for a newly created container but those kernel resources are shared among all containers running on the application node. To get a list of the namespaces that were created for the app-cli use the `lsns` command. You need the POD for the application to pass as a parameter to `lsns`.

OpenShift uses the five linux namespaces to isolate processes and resources on application nodes. Coming up with a concise definition for exactly what a namespace does is a little difficult, two analogies best describe their most important properties

- Namespaces are like paper walls in the linux kernel, they are lightweight and easy to stand up and tear down, but they offer sufficient privacy when they are in place.
- Namespaces are similar to two way mirrors, from within the container only the resources in the namespace are available but with proper tooling you can see what is in a namespace from the host system.

The following snippet lists all namespaces for the app-cli with `lsns`. The command requires the process id first, that means that we have to make sure that the process for the correct running container is extracted first, this is described above in detail, but in summary it has to be done by first logging into the cluster

```
# here we can directly list the namespaces for the target process id, in our
   case we can see that this process id was
# extracted from the cluster, using the crictl command above, where we
   inspected the spec of the container
$ sudo lsns -p 7825
NS          TYPE   NPROCS     PID USER        COMMAND
4026531834 time       424       1 root        /usr/lib/systemd/systemd --
   switched-root --system --deserialize 28
4026531837 user       424       1 root        /usr/lib/systemd/systemd --
   switched-root --system --deserialize 28
4026534705 uts         13    7825 1000660000 httpd -D FOREGROUND
4026534706 ipc         13    7825 1000660000 httpd -D FOREGROUND
4026535152 net         13    7825 1000660000 httpd -D FOREGROUND
4026535628 mnt         13    7825 1000660000 httpd -D FOREGROUND
4026535698 pid         13    7825 1000660000 httpd -D FOREGROUND
4026535703 cgroup      13    7825 1000660000 httpd -D FOREGROUND
```

And from the output above, we can clearly see that the process id which we extracted for the pod and by proxy the container app-cli, which was `7825` that there are multiple namespaces attached to this process, the type column signifies that there are mount, cgroup and net, along with `UTS`, and other namespaces created for the process. The five namespaces that OpenShift uses to isolate the apps are as follows:

- Mount - ensures that only the correct content is available to apps in the container
- Network - gives each container its own isolated network stack
- PID - provides each container with its own set of PID counters
- IPC - provides shared memory isolation for each container
- UTS - gives each container its own hostname and domain name

There are currently two additional namespaces in the Linux kernel that are not used by OpenShift

- Cgroup - are used as shared resource on the OpenShift node, so this namespace is not required for effective isolation.
- User - this namespace can map a user in a container to a different user on the host, for example a user with ID 0 in the container could have user ID 5000, when interacting with resources on the host. This feature can be enabled in OpenShift but there are issues with performance and node configuration that fall out of scope for our example cluster, if you like more information on enabling the user namespace to work with docker and thus with OpenShift see the article `Hardening Docker Hosts with User Namespaces - by Chris Binnie`

Mount namespace

The mount namespace isolated file system content, ensuring that content assigned to the container by OpenShift is the only content available to the process, running in the container, the mount namespace for the app-cli container allows the app in the container to access only the content in the custom app-cli container image, and any information stored on the persistent volume associated with the persistent volume claim (PVC) for the app-cli.

`Apps always need persistent storage, persistent storage allows data to persist when a pod is removed from the cluster, it also allows data to be shared between multiple pods when needed, you will learn how to configure and use persistent storage on an NFS server with OpenShift in future section`

The root file system based on the app-cli container image is a little more difficult to uncover, but we will do that next. When you configured OpenShift you specified a block device for docker to use for container storage. Your OpenShift configuration uses logical volume management on this device for container storage. Each container gets its own logical volume when it is created. This storage solution is fast and scales well for large production clusters. To view all logical volumes created by docker on your host, run the `lsblk` command. This command shows all block devices on your host, as well as any logical volumes. It confirms that docker has been creating logical volumes for containers

```
# enter sudo mode, with the core user, to give us more access and command
   execution permissions
$ sudo -i

# list block devices, that lists information about all available or the
   specified block devices on the system, the
# command reads the sysfs file system and udev database to gather this
   information, by default print all block devices
# except any RAM disks, in a tree list format.
$ lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
vda    252:0    0   31G  0 disk
vda1 252:1    0    1M  0 part
vda2 252:2    0  127M  0 part
vda3 252:3    0  384M  0 part /boot

# to list all the active pods, the output is abridged, but we can see what we
    care interested in, this is the app-cli
# pods, which are visible and list-able from the command below, we can see
   that the ps command shows the id of the
# container, along with the pod it is connected to, this way we can inspect
   the container, by id
```

```
$ crictl ps
CONTAINER     IMAGE
                                                                        CREATED
            STATE   NAME            ATTEMPT ID            POD
3b014dbd532c3 quay.io/openshift-release-dev/ocp-v4.0-art-dev@
                           11 minutes ago  Running registry-server 0
   e5017350a2c71 redhat-operators-dtxxg
1ea26899fa493 image-registry.openshift-image-registry.svc:5000/image-uploader
   /app-cli@ 15 minutes ago  Running app-cli        0       d4c372b145163
   app-cli-5fdd99b58d-dplv6

# after we can inspect the container, we can see from the inspect output,
   that there are a few mount targets, a few are
# generic ones such as the /etc/hosts, however we can see that there is one
   which is specific
$ crictl inspect -f '{{ .GraphDriver.Data.DeviceName }}' 1ea26899fa493

# here we can see two important things, first is the process id of the
   container, the PID 1, which we have already seen
# once before, we can use this process id to inspect the state of the process
    from the host, it is a unique process which
# directly ties all information on the host to the container, the host does
   not see pod id, or container id, it sees the
# process id
  "info": {
    "checkpointedAt": "0001-01-01T00:00:00Z",
    "pid": 33398,
    "privileged": false,
    "restored": false,

# this section shows us how and where part of the root file system is mounted
   , the root file system of a container is
# built on the basis of layers, but, certain parts of it may be mounted to
   completely different parts outside of the
# container, in a way these are very similar to shared directories, on a
   network. These mount points refer to hostPath and
# containerPath, which are very much self explanatory
....
    "mounts": [
      {
        "containerPath": "/etc/hosts",
        "gidMappings": [],
        "hostPath": "/var/lib/kubelet/pods/85c76562-91f8-4857-a075-42
           f7089b23e6/etc-hosts",
        "propagation": "PROPAGATION_PRIVATE",
        "readonly": false,
        "recursiveReadOnly": false,
        "selinuxRelabel": true,
        "uidMappings": []
      },
      {
        "containerPath": "/dev/termination-log",
        "gidMappings": [],
        "hostPath": "/var/lib/kubelet/pods/85c76562-91f8-4857-a075-42
           f7089b23e6/containers/app-cli/7e8ccd6b",
        "propagation": "PROPAGATION_PRIVATE",
        "readonly": false,
        "recursiveReadOnly": false,
        "selinuxRelabel": true,
        "uidMappings": []
      },
      {
        "containerPath": "/var/run/secrets/kubernetes.io/serviceaccount",
        "gidMappings": [],
        "hostPath": "/var/lib/kubelet/pods/85c76562-91f8-4857-a075-42
           f7089b23e6/volumes/kubernetes.io~projected/kube-api-access-f7lcd",
        "propagation": "PROPAGATION_PRIVATE",
        "readonly": true,
        "recursiveReadOnly": false,
        "selinuxRelabel": true,
        "uidMappings": []
      }
    ],
...

# here having the PID we can list the namespaces of the container process for
    our application, and we can also see the
# multitude of different namespaces that it has, we want to see and inspect
   the mount namespace
$ lsns -p 33398
        NS TYPE   NPROCS   PID USER       COMMAND
4026531834 time      524     1 root       /usr/lib/systemd/systemd --switched
   -root --system --deserialize 28
4026531837 user      523     1 root       /usr/lib/systemd/systemd --switched
   -root --system --deserialize 28
4026535352 mnt        13 33398 1000660000 httpd -D FOREGROUND
4026535353 pid        13 33398 1000660000 httpd -D FOREGROUND
4026535354 cgroup     13 33398 1000660000 httpd -D FOREGROUND
4026535522 uts        13 33398 1000660000 httpd -D FOREGROUND
4026535523 ipc        13 33398 1000660000 httpd -D FOREGROUND
4026535524 net        13 33398 1000660000 httpd -D FOREGROUND

# we can now enter the mount namespace of the container, note that if you now
    execute ls / you will directly see the
# contents of the container itself, be able to access the root file system of
    the container
$ nsenter --mount --target 33398 /bin/sh

# to exit the mount namespace, simply use exit
$ exit

# now let us see where this is mounted on the host, we can use the process id
```

```
    we have obtained from above, to see
# where the root file system is mounted, on the host, or in this case the
   host is the cluster node itself, take a note
# of the output and the overlay paths, these reference paths on the host root
    file system, placed in the /var directory
grep -w "/" /proc/33398/mountinfo
25787 13562 0:853 / / rw,relatime - overlay overlay rw,context="system_u:
   object_r:container_file_t:s0:c5,c26",lowerdir=/var/lib/containers/storage/
   overlay/l/K565QXNBE4UPBQO2PDIBJBW5U5:/var/lib/containers/storage/overlay/l
   /QXL4VNC6IZOL
VYH23JQKKXAMIE:/var/lib/containers/storage/overlay/l/57
   MMERXGQHPVYCAGGCVVRXUE7F:/var/lib/containers/storage/overlay/l/
   HITMXQH3G2NMN7M5U4GT2XIYE5:/var/lib/containers/storage/overlay/l/4
   H26VNEWY45JGAAVUESP7J4CKD,upperdir=/var/lib/contai
ners/storage/overlay/
   f2d783dad2cc2779ef613259e008670c46fa1252439afff3fa10ee8cbde00567/diff,
   workdir=/var/lib/containers/storage/overlay/
   f2d783dad2cc2779ef613259e008670c46fa1252439afff3fa10ee8cbde00567/work,
   volatile
```

What are these layers, above, referring to. The file system in a container runtime, is made up of layers, meaning that it is not one big chunk of data, but rather, it is layered, for example the base image itself is one layer, but any subsequent changes to the file system in the container will generate a new layer which will stack on top of the previous parent layer. For example adding a new file, creating directories, copying or removing data from the container, will add layers on top of each other. This is similar to how source control systems work, where we have commits, which are based on other commits, and if you follow the commits you can go between different states of the repository, here the idea is very much the same, the container's file system is layered in a very similar fashion, you can think of layers being like commits in a source control system

Above we have shown how we can actually enter the mount namespace, detect where the root file system of a container is mounted on the host, and even explore and operate within the mount namespace, effectively allowing us to modify or work with the root file system with the container, all things that are usually hidden behind commands such as `exec`. Under the hood, exec does the same thing, it enters the namespace of the process allowing you to execute processes in the context of the container's namespace

Each image is build from read-only layers, or also called Copy on Write. And it is working similarly to source control systems as Git. Here is how it works, each image is built from read only layers, if you take the following Dockerfile

```
FROM ubuntu:22.04          # Base layer (Layer A)
RUN apt-get update         # Layer B
COPY app.py /opt           # Layer C
CMD ["python", "/opt/app.py"]
```

Each command creates a new read only layer, layers are stacked on top each other, what does that mean we have the following layers in the given Dockerfile above - Layer C -> Layer B -> Layer A, meaning that similarly to commits in a source control system each layer is based on the previous one, in this case Layer C is based off off Layer B and so on. When you run a container read only layers from the image are mounted as `lowerdir - OverlayFS`, a new writeable layer `upperdir` is added on top, container specific changes, the merged view what the container sees is the `upperdir` + `lowerdir`. How are changes handled in this case, modifying a file from the read only layer - the file is copied from the `lowerdir` to the `upperdir`, changes are applied to the copy in `upperdir`. New files on the other hands are directly written on the `upperdir`. Deleting a file, or a whiteout file, is created in `upperdir` to hide the file in lower layers, the file is actually obscured or occluded in the lower layers, not deleted from those layers.

The read only layers are shared between many different images and by proxy containers, however the writeable layers, are only unique to the container in question, these layers are ephemeral, when the container is deleted so are they, the merged view is what the container sees between the read-only layers and its own writeable layers. As we said each modification or change in the container file system will generate a new writeable layer. The read-only layers are only created during the creation of the container image.

The key implications of this is that this is quite efficient model - multiple containers share the same read only base layers, saves disk space, and since a lot of images are based on the same base layers, such as base images like CentOS, Alpine etc, these layers are shared. Performance is also good since we have copy on write, only the modified filed are copied into new layers. The changes in the container is isolated well by the existence of the `upperdir` layer, the writeable layer is ephemeral by default, deleted when the container is deleted, use the volume to persist data across the containers, note that volumes live outside of the file system layers, they are simply mount point in the container, meaning that they are not part of the file system layers at all

UTS namespace

`UTS` stands for UNIX time sharing in the Linux kernel. The `UTS` namespace lets each container have its own hostname and domain name. It can be confusing to talk about time sharing when the `UTS` namespace has nothing to do with managing the system clock. Time sharing originally referred to multiple users sharing time on system simultaneously. Back in the 70s when the concept was created it was a novel idea. The `UTS` data structure in the Linux kernel had its beginnings then. This is where the hostname domain name and other system information are retained. If you would like to see all the information in that structure run `uname -a` on a Linux server. That command queries the same data structure.

The easiest way to view the hostname for a server is to run the hostname command. You could use `nsenter` to enter the `UTS` namespace for the app-cli container the same way you entered the mount namespace in the previous section. But there are additional tools that will execute a command in the namespace for a running container. One of those tools is the docker exec command. To get the hostname value for a running container pass docker exec a container's short ID and the same hostname command you want to run in the container. Docker executes the specified command for you in the container's namespaces and returns the value. The hostname for each OpenShift container is its pod name: `docker exec <container-id> hostname`

Each container has its own hostname because of its unique `UTS` namespace. If you scale up app-cli the container in each pod will have a unique hostname as well. The value of this is identifying the data coming from each container in a scale up system. To confirm that each container has a unique hostname log into your cluster as your developer user `oc login -u developer -p developer <cluster-url>`. The `oc` command line tools has a functionality that's similar to docker exec, instead of passing in the short ID for the container however you can pass it the pod in which you want to execute the command. After logging in to your `oc` client, scale the app-cli application to two pods with the following command `oc scale deployment/app-cli --replicas=2`, in this case the `dc` in the command stands for deployment config, this is the shorthand name for the object type. The deployment config is the old 3.xx version of the deployment object which was simply renamed in the 4.xx version to deployment. The command with newer versions would look like a little bit different, but pretty much the same - `oc scale deployment/app-cli --replicas=2`

This will cause an update to your app-cli deployment config and trigger the creation of a new app-cli pod. You can get the new pod's name by running the command `oc get pods | grep "Running"`. The grep call prevents the output of pods in a completed state so you see only active pods in the output. Because the container hostname is its corresponding pod name in OpenShift you know which pod you were working with using docker directly.

```
# first make sure that we scale up the deployment of the app, which would
   mean that we have to alter the deployment,
# that would cause OpenShift to bring two new pods, when they are ready the
   old ones will be removed,
$ oc scale deployment/app-cli --replicas=2
deployment.apps/app-cli scaled

# we grep only the ones in running state, making sure to avoid additional
   unwanted pods in completed state which will
# only pollute the output
$ oc get pods | grep "Running"

# we have two running pods in this case since we have set the replicas count
   to 2, now we can resolve the actual
# hostname of the pod
app-cli-5b9c58956d-p7jpn   1/1     Running                           1          23h
app-cli-5b9c58956d-p8jpn   1/1     Running                           1          23h
```

To get the hostname from your new pod, use the `oc exec` command targeting the new pod, it is similar to docker exec, but instead of the container's short id you use the pod name to specify where you want the command to run. The hostname for your new pod matches the pod name, just like your original pod. The command may look something like that:

```
# note that the project here is not passed because by default we are always
   in a project config, we may use oc -n to
# specify temporary project namespaces/name for the current command only,
   this avoids having to first do the oc project
# <project-name> first before having to run the command, and does not change
   the current context permanently
$ oc exec app-cli-5b9c58956d-p7jpn -- hostname

# the output of this command must always be the same like the pod-id, meaning
    that the hostname will always match the id
# of the pod as shown by the command - oc get pods for example, the full id
   that is, therefore the command above will
# print the same text as the hostname as the pod id we have used to call the
   exec command with
app-cli-5b9c58956d-p7jpn
```

```
Remember that in docker, kubernetes and OpenShift, the hostname of the container in
essence is always the unique container identifier as provided by the orchestrator or
docker
```

PID namespace

Because PID are how one app sends signals and information to other apps isolating visible PID in a container to only the app in it is an important security feature. This is accomplished using the PID namespaces. On a linux server the `ps` command shows all running processes along with their associated PID on the host. This command typically has a lot of output on a buys system. The `--ppid` option limits the output to a single PID and any child processes it has spawned, from your app node, run the `ps` command with the `--ppid` option and include the PID you obtained for your app-cli container. Here you can see that the process for PID 7825 is httpd and that it has spawned several other processes:

```
# we run this command on the cluster node itself, this will provide us with a
    list of processes started by this
# particular process id, this process id however is the process id of the
   container, we obtained a few sections earlier
$ ps --ppid 7825

PID TTY          TIME CMD
8938 ?        00:00:00 cat
8943 ?        00:00:00 cat
8951 ?        00:00:00 cat
8959 ?        00:00:00 cat
9033 ?        00:00:09 httpd
9034 ?        00:00:08 httpd
9035 ?        00:00:08 httpd
9064 ?        00:00:08 httpd
9070 ?        00:00:08 httpd
9071 ?        00:00:08 httpd
9077 ?        00:00:08 httpd
9082 ?        00:00:08 httpd
```

Use `oc` exec to get the output of `ps` for the app-cli pod that matches the PID you collected, If you've forgotten you can compare the hostname in the docker compare to the pod name. From inside the container do not use the `--ppid` option, because you want to see all the PID visible from within the app-cli container.

```
# notice that here we use the pod's directly
$ oc exec app-cli-5b9c58956d-p7jpn -- ps

PID TTY          TIME CMD
    1 ?        00:00:01 httpd
    28 ?        00:00:00 cat
    29 ?        00:00:00 cat
    30 ?        00:00:00 cat
    31 ?        00:00:00 cat
    32 ?        00:00:09 httpd
    33 ?        00:00:08 httpd
    34 ?        00:00:08 httpd
    62 ?        00:00:08 httpd
    68 ?        00:00:08 httpd
    69 ?        00:00:08 httpd
    75 ?        00:00:08 httpd
    80 ?        00:00:08 httpd
330 ?        00:00:00 ps
```

There are three main differences in the output.

- The initial httpd command is listed in the output
- The `ps` command is listed in the output
- The PID are completely different

Each container has a unique PID namespaces. That means from inside the container the initial command that started the container (PID 7825) is viewed as PID 1, this is the parent process, from which all others get forked. All the processes it spawned also have PID, in the same container specific namespace. Apps that are created by a process already in a container automatically inherit the container's namespace. This makes it easier for app in the container to communicate. So far we have discussed how filesystems hostnames and PID are isolated in a container, next let us take a quick look at how shared memory resources are isolated.

Memory namespace

Apps can be designed to share memory resources. For example app A can write a value into a special shared section of a system memory and the value can be read and used by app B. The following shared memory resource are isolated for each container in OpenShift

- POSIX message queue interfaces in `/proc/sys/fs/mqueue`
- IPC interfaces in `/proc/sysvipc`
- Memory parameters like - `msgmax, msgmnb, msgmni etc`

If a container is destroyed shared memory resources are destroyed as well. Because these resources are app specific you will work with them more in next sections. When you deploy a stateful app, the last namespace to discuss in the network namespace.

Networking namespace

The fifth kernel namespace that is used by docker to isolate containers in OpenShift is the network namespace there is nothing funny about the name for this namespace. The networking namespace isolated network resources and traffic in a container the resources in this definition mean the entire TCP/IP stack, is used by apps in the container. Future chapters are solely dedicated to going deep into the OpenShift software defined networking, but we need to illustrate in this section how the view from within the container is drastically different than the view from your host. The PHP builder image you used to create `app-cli` and `app-gui` doesn't have the IP utility installed. You could install it into the running container using yum. But a faster way to is to use `nsenter`. Earlier you used `nsenter` to enter the mount namespace of the docker process so you could view the root filesystem for app-cli.

```
It would be great if we could go through the OSI model here. Unfortunately it is out the
 scope for now. In short it is a model to describe how data travels in a TCP/IP network
. These are seven layers, you will often hear about layer 3 devices, or a layer 2 switch
, when someone says that, they are referring to the layer of the OSI model on which a
particular device operates. Additionally, the OSI model is a great tool to use any time
 you need to understand how data moves through any system or app. If you haven't read up
 on the OSI model before, it is work your time to look at the article - "The OSI model
explained, how to understand and (Remember)the 7 layer Network Model.
```

If you run `nsenter` and include a command as the last argument, then instead of opening an interactive session in that namespace the command is executed, in the specified namespace, and returns the result. Using this tool you can run the IP command from your server's default namespace in the network namespace of your app-cli container. If you compare this to the output from running `/sbin/ip` a command on your host the differences are obvious. Your app node will have 10 or more active network interfaces. These represent the physical and software defined devices that make OpenShift function securely. But in the app-cli container you have a container specific loopback interface device and a single network interface with a unique MAC IP address:

```
# running the following on the host/cluster node can yield a big output
   similar to the output below
$ ip a

# the output below is the abridged version of the actual output, suffice to
   say that the cluster node has many many more
# software and hardware network devices active, compared to the container
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
   default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    group default qlen 1000
    link/ether 52:54:00:bc:15:3a brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.66/24 brd 192.168.122.255 scope global noprefixroute
       dynamic eth0
       valid_lft 3187sec preferred_lft 3187sec
    inet6 fe80::5054:ff:febc:153a/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    group default qlen 1000
    link/ether 52:54:00:53:34:7a brd ff:ff:ff:ff:ff:ff
    inet 192.168.42.124/24 brd 192.168.42.255 scope global noprefixroute
       dynamic eth1
       valid_lft 3146sec preferred_lft 3146sec
    inet6 fe80::5054:ff:fe53:347a/64 scope link
       valid_lft forever preferred_lft forever
5: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    group default
    link/ether 02:42:be:ed:e5:72 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:beff:feed:e572/64 scope link
       valid_lft forever preferred_lft forever
11: veth3dab0ee@if10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   noqueue master docker0 state UP group default
    link/ether f6:a1:76:bd:cc:1d brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::f4a1:76ff:febd:cc1d/64 scope link
       valid_lft forever preferred_lft forever
15: veth8baab20@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   noqueue master docker0 state UP group default
    link/ether e2:10:11:22:12:87 brd ff:ff:ff:ff:ff:ff link-netnsid 2
    inet6 fe80::e010:11ff:fe22:1287/64 scope link
       valid_lft forever preferred_lft forever
23: veth1a268b0@if22: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   noqueue master docker0 state UP group default
    link/ether 6e:f8:23:2d:6d:36 brd ff:ff:ff:ff:ff:ff link-netnsid 6
    inet6 fe80::6cf8:23ff:fe2d:6d36/64 scope link
       valid_lft forever preferred_lft forever
25: veth98336dc@if24: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   noqueue master docker0 state UP group default
    link/ether 8e:f8:d3:78:6f:27 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::8cf8:d3ff:fe78:6f27/64 scope link
       valid_lft forever preferred_lft forever
31: vethe2d8c8f@if30: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
```

```
   noqueue master docker0 state UP group default
    link/ether f2:80:d0:94:f5:af brd ff:ff:ff:ff:ff:ff link-netnsid 3
    inet6 fe80::f080:d0ff:fe94:f5af/64 scope link
       valid_lft forever preferred_lft forever
```

Here is the other command which uses the same user binary `ip` from the host, to run in the context of the container (or as we know a container is a fancy process / PID) namespace.

```
# this we again run on the host/cluster node, however as you can see we are
   first entering the container namespace, in
# this case, then we run the IP command, what is cool is that since the
   container is running on the host/cluster node, we
# can use the same binary (ip) located in sbin, to obtain information about
   the network namespace but in the context of
# the container, instead of the host.
$ sudo nsenter -t 7825 -n /sbin/ip a

# This is pretty much the entire un-edited output from the command when run
   in the context of the container namespace,
# as you can see the number of software / hardware devices here are much much
    less, which is what we would expect, and
# as mentioned, one loopback - lo, and one regular interface - eth0
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
   default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc noqueue state
   UP group default
    link/ether 0a:58:0a:d9:00:43 brd ff:ff:ff:ff:ff:ff link-netns 312e9fcc-4
       ccb-4a32-8488-91b6d821d62e
    inet 10.217.0.67/23 brd 10.217.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::858:aff:fed9:43/64 scope link
       valid_lft forever preferred_lft forever
```

Summary

The network namespace is the first component in the OpenShift networking solution. We'll discuss how network traffic gets in and out of containers in next sections, when we cover OpenShift networking in depth, in OpenShift isolating processes doesn't happen in the app- or even in the userspace, on the app node. This is a key difference between other types of software clusters, and even some other container based solutions. In OpenShift isolation and resource limits are enforced in the linux kernel on the app nodes, isolation with kernel namespaces provides a much smaller attack surface. An exploit that would let someone break out from a container would have to exist in the container runtime or the kernel itself. With OpenShift as we'll discuss in depth in next section when we examine security principles in OpenShift configurations of the kernel and the container runtime is tightly controlled. The last point we would like to make in this section echoes how we began the discussion. Fundamental knowledge of how containers work and use the Linux kernel is invaluable. When you need to manage your cluster or troubleshoot issues when they arise, this knowledge lets you think about containers in terms of what they are doing all the way to the bottom of the Linux kernel. That makes solving issues and creating stable configurations easier to accomplish. Before you move on clean up to a single replica.

Cloud native apps

Cloud native is how the next generation of apps is being created. In this part of the book, we'll discuss the technologies in OpenShift that create the continuously deploying self-healing auto-scaling behaviors we all expect in a cloud native app. Previous chapters focused on working with and modifying the services in OpenShift. This section also walks you through creating problem for your app to ensure that they're always functioning correctly.

Testing app resiliency

When you deployed the Image Uploader app in previous sections, one pod was created for each deployment. If that pod crashed, the app would be temporarily unavailable until a new pod was created to replace it. If your app became more popular, you would not be able to support new users past the capacity of a single pod. To solve this problem and provide scalable apps, OpenShift deploys each app with the ability to scale up and down. The app component that handles scaling app pods is called the replication controller.

Replication controller The replication controller main function is to ensure that the desired number of identical pod is running all the time. If a pod exits or fails, the replication controller deploys a new one to ensure a healthy app is always available. In other words OpenShift takes care of maintaining the `desired state, as configured by the developers or app managers`. You can think of the replication controller as a pod monitoring agent that ensures certain requirements are met across the entire OpenShift cluster. You can check the current status of the replication controller (RC) for the app-cli deployment by running the `oc describe`. Note that the individual deployment is specified not the name of the app. The information tracked about the RC helps to establish its relationships to the other components that make up the app.

```
# to make sure that we create, a replication controller instances, which will
    be done when we use scale, on the existing
# deployment, the replication controller is an evolution of the replica
   controller, but it is basically the same object
# that is representing the actual replicas being managed by the orchestrator
$ oc scale deployment/app-cli --replicas=2
deploymentconfig.apps.openshift.io "app-cli" scaled
```

- Name of the RS/RC, which is the same as the name of the deployment, it is associated with
- Image name used to create pods for the RC
- Labels and selectors for the RC
- Current and desired number of pod replicas running in the RC
- Historical pod status information i for the RC, including how many pods are waiting to be started or have failed since the creation of the RC

The labels and selectors in the next listing are key-value pairs, that are associated with all OpenShift components. They are used to create and maintain the relationships and interactions between apps. We will discuss them in more depth in next sections.

Also need to note that the command below will return the list of RC only if there were ever any replicas created, if the app pods were never replicated, meaning that there was only ever one pod for the deployment config, then no RC object will be created, make sure that you have had created replicas for the given deployment, otherwise you might not receive the expected result, from `oc get rs`

```
# describe the replication controller
$ oc describe rs/app-cli-1

Name:          app-cli-1
Namespace:     image-uploader
Selector:      app=app-cli,deployment=app-cli-1,deploymentconfig=app-cli
Labels:        app=app-cli
               openshift.io/deployment-config.name=app-cli
Annotations:   openshift.io/deployer-pod.completed-at=<date>
               openshift.io/deployer-pod.created-at=<date>
               openshift.io/deployer-pod.name=app-cli-1-deploy
               openshift.io/deployment-config.latest-version=1
               openshift.io/deployment-config.name=app-cli
               openshift.io/deployment.phase=Complete
               openshift.io/deployment.replicas=1
               openshift.io/deployment.status-reason=config change
               openshift.io/encoded-deployment-config={"kind":"
                  DeploymentConfig","apiVersion":"v1","metadata":{"name":"app-
                  cli","namespace":"image-uploader","selfLink":"/apis/apps.
                  openshift.io/v1/namespaces/image-up...
Replicas:      2 current / 2 desired
Pods Status:   2 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:        app=app-cli
                 deployment=app-cli-1
                 deploymentconfig=app-cli
  Annotations:   openshift.io/deployment-config.latest-version=1
                 openshift.io/deployment-config.name=app-cli
                 openshift.io/deployment.name=app-cli-1
                 openshift.io/generated-by=OpenShiftNewApp
  Containers:
   app-cli:
    Image:         172.30.1.1:5000/image-uploader/app-cli@sha256:478
       fe6428546186cfbb0d419b8cc2eab68af0d9b7786cc302b2467e5f11661db
    Ports:         8443/TCP, 8080/TCP
    Host Ports:    0/TCP, 0/TCP
    Environment:   <none>
    Mounts:        <none>
  Volumes:         <none>
```

```
Note that more recent versions of openshift will by default use the ReplicationController
 instead of the ReplicationController object, this is because the new set object is more
 feature full, and the old ReplicationController is being phased out, ultimately it is
 the same object, with some new nice features added on top of the manifest and object
specification
```

Labels and selectors As we go forward, it is important that you understand the following, regarding how labels and selectors are used in OpenShift

- When an pp is deployed in OpenShift every object that is created is assigned a collection of labels. Labels are unique per project, just like apps names. That means in Image Uploader, only one app can be named app-cli.

- Labels that have been applied to an object are attributes that can be used to create relationships in OpenShift. But relationships are two way streets, if something can have a label, something else must be able to state a need for a resource with that label. The other side of this relationship exists in the shape of label selectors
- Label selectors are used to define the labels that are required when work needs to happen

Let's examine this in more depth, using the app-cli app you deployed in the second section. In the next example you will remove a label from a deployed pod. This is one of the few times we will ask you to do something to intentionally break an app. Removing a label from a pod will break the relationship with the RC and other app components. The purpose of this example is to demonstrate the RC in action - and to do that you need to create a condition it need to remedy.

As we have already mentioned selectors in an app component are the labels it uses to interact with other components, and link up to them, they are `not simply meant for tagging human readable information to app components`. The app-cli RC will create and monitor apps pods with the following labels:

```
# this is the labels and selector sections from a describe command run
   against the deployment of the app-cli, you can
# see that in our case the deployment object is tagged with many labels, and
   the selector in this case matches the
# in the pod template. The labels for the dpeloyment are mostly kubernetes
   specific, and only one is really the user
# defined one which is app=app-cli
Name:                   app-cli
Namespace:              image-uploader
CreationTimestamp:      <date>
Labels:                 app=app-cli
                        app.kubernetes.io/component=app-cli
                        app.kubernetes.io/instance=app-cli
                        app.kubernetes.io/name=php
Selector:               deployment=app-cli

# pod contains only one label, which is matched against the selector section
   in the deployment above, that is how they
# are interlinked, and that is how OpenShift knows which pods belong to which
    deployments, the same rule is true and
# followed for any other OpenShift object, that is how relations between them
    are build in Kubernetes and OpenShift
Pod Template:
  Labels:       deployment=app-cli
```

The fastest way to delete the pods for the app-cli deployments is through the command line. This process shouldn't be part of your normal app workflow, , but it can come in handy when you're troubleshooting issues in an active cluster. From the command line run the following `oc delete pod -l app=app-cli` what this does is it will delete all pods that have a matching selector `app-cli`, the app selector was automatically attached when we created the app-cli in the first place by OpenShift, but usually in the real world one would decide how to name the key (app) and value (app-cli) pair of the label tag. What is important to note is that labels are fundamental part of how OpenShift works, and Kubernetes as well for that matter. If we can make an analogy it would be that the labels are like table relations in a relational database.

There is also another command that can be used to delete and cleanup all resources related to and attached to a given label, we can use the following - `oc delete all --selector app=app-cli`. This will make sure to delete all resources, which are attached or related and associated to the given label.

Returning back to the pod for which we removed or detached the label, you might be wondering whether the abandoned pod will still receive traffic from users. It turns out that the service object, responsible for network traffic, also works on the concept of labels and selectors. To determine whether the abandoned pod would have served traffic, you need to look at the Selector field in the service object. You can get this selector information about the app-cli service by running the following `oc describe svc app-cli | grep Selector` this will print out the selector label for the service, in this case something like `app=app-cli,deploymentconfig=app -cli`, as you can see there are more than one selectors which can be applied to a OpenShift object, the key and value are usually enough to know how that label or selector will be used, here we can see that there is one generic top level selector which is the `app=app-cli` which basically says that this service object is linked to all `app-cli` objects, in a way tagging them with a namespace of sorts, then there is a specific selector which links the service to the deployment config which is in our case not relevant. Because we have deleted the selector label from the original pod, its label are not longer match for the app-cli service, since there is no other pod, that means that not traffic will be routed to the original pod, selectors would no longer receive traffic requests.

Kubernetes was born out of many lessons learned at Google from running containers at scale for 10+ years. The main two orchestration engines internally at Google during this time have been Borg and its predecessor, Omega. One of the primary lessons learned from the two systems was that control loops of decoupled API objects were far preferable to large centralized stateful orchestration. This type of design is often called control through choreography. Here are just a few of the ways it is implemented in Kubernetes.

- Decoupled API components
- Avoided stateful information
- Looping through control loops against various micro services

By running through control loops instead of maintaining a large state diagram, the resiliency of the system is considerably improved. If a controller crashes, i,t reruns the loop when it's restarted whereas a state machine can become brittle when there are errors or the system starts to grow in complexity. In our specific examples, this holds true because the RC loops through pods with the labels in its Selector field as opposed to maintaining a list of pods that it is supervising

Replication controllers ensure that properly configured apps pods are always available in the proper number. Additionally the desired replica counts can be modified manually or automatically. In the next section we will discuss how to scale app deployments

Scaling applications

An app can consist of many different pods, all communicating together to do work for the app users. Because different pods need to be scaled independently, each collection of identical pods is represented by a service components, as we initially discussed. More complex apps can consist of multiple services of independently scaled pods. A standard app design uses three tiers to separate concerns in the app.

- Presentation layer - Provides the user interface, styling and workflows, for the user. This is typically where the website lives.
- Login layer - Handles all the required data processing for an app. This is often referred to as the middleware layer.
- Storage layer - Provides persistent storage for app data. This is often database, filesystems or a combination of both

So basically the app code runs in the pods for each app layer. That code is accessed directly from the routing layer. Each app service communicates with the routing layer and the pods it manages. This design results in the fewest network hops between the user and the app. This design also allows each layer in the three tier design to be independently scaled to handle its workload effectively without making any changes to the overall app configuration, also changes in any of app layers are not going affect the other layers they interact with, e.g. `scaling` or generally modifying the midtier end layer will impact the frontend tier

Application Health

In most situations app pods run into issues because the code in the pod stops responding.

This is typically where the website lives, login layer handles all the required data processing for an this is often referred to as the middleware layer, storage layer provides persistent storage for data, this is often database filesystems or a combination of both, so basically the code runs in the pods layer, that code is accessed directly form the routing layer, each service communication with the routing layer. The first step is building a resilient app is to run automated health and status checks on your pods, restarting them when necessary without manual intervention. Creating probes to run the needed checks on apps to make sure they are healthy is built into Open shift. The first type of problem we will look at is the liveness probe.

In OpenShift you define a liveness probe as a parameter for specific containers in the deployment config. The liveness probe configuration then then propagates down to individual containers created in pods running as part of the deployment. A service on each node running the container is responsible for running the liveness probe that is defined in the deployment config. If the liveness probe was created as a script then it is run inside the container. If the liveness probe was created as HTTPS response or TCP socket based probe then it is run by the node connecting to the container. If a liveness probe fails for a container then the pod is restarted.

Note that the service that executed liveness probe checks is called the kubelet service. This is the primary service that runs on each app node in the OpenShift cluster, and it actually responsible for many other things, among which is running the liveness status of the node.

Creating liveness probes

In OpenShift the liveness probe component is a simple powerful concept that checks to be sure an app pod is running and healthy. Liveness probes can check container health three ways:

- HTTP checks if a given URL endpoint served by the container, and evaluates the HTTPS status response code
- Container execution check - a command typically a script that is run at intervals to verify that the container is behaving as expected. Non zero exit code from the command results in a liveness check failure.
- TCP socket check - Checks that a TCP connection can be established on a specific TCP port in the app pod.

Note that the HTTP response code is a three digit number supplied by the server as part of the HTTP response headers in a web request. A 2xx response indicates that a successful connection is made, and a 3xx response indicated that HTTP redirect. You can find more bout the response codes on the `IETF` website

As a best practice always create a liveness probe unless your app is intelligent enough to exit when it hits an unhealthy state. Create liveness probes that not only check the internal components of the app, but also isolate problems from external service dependencies. Fr example a container shouldn't fail its liveness probe because another service that it needs isn't functional. Modern apps should have code to gracefully handle missing service dependencies. If you need an app to wait for a missing service dependency you can use readiness probes., which are covered later on in this section. For legacy apps that require an ordered startup sequence of replicated pods, you can take advantage of a concept called stateful sets, which we will cover later on. To make creating probes easier, a health check wizard is built in the OpenShift web interface, using the wizard will help you avoid formatting issues that can result from creating the raw YAML template by hand.

```
$ oc set probe deployment/app-cli --liveness --get-url=http://:8080/ --
   initial-delay-seconds=5
deployment.apps/app-cli probes updated
```

After one adds the liveness probe, there are ways to check how the liveness probe has been reflected in the final deployment configuration by simply inspecting the deployment object, and we will notice that there is indeed a new line in there which represents the liveness probe, of type HTTP, which check on the root context path of the app, there are other minor things that we can also take a note of such that - timeout, the time to wait for a response from the endpoint, then there is the period, meaning that every N number seconds a new request will be made, the threshold is 1 success response means the container is alive, at most 3 failures means that the pod will be restarted and so will be the container

```
# here is the command which will inspect and show the liveness probe, that
   was recently configured, most of the data for
# the deployment was omitted, just to show the probe info from the otherwise
   big deployment describe command dump
$ oc describe deployment -l app=app-cli
Name:                   app-cli
Pod Template:
  Labels:       deployment=app-cli
  Annotations:  openshift.io/generated-by: OpenShiftNewApp
  Containers:
   app-cli:
    Image:         image-registry.openshift-image-registry.svc:5000/image-
       uploader/app-cli@sha256:17
       c9ec389e0e130b3891e25e64b53cda350a6223732f62f56032b42cb361ffd1
    Ports:         8080/TCP, 8443/TCP
    Host Ports:    0/TCP, 0/TCP
    Liveness:      http-get http://:8080/ delay=5s timeout=1s period=10s #
       success=1 #failure=3
    Environment:   <none>
    Mounts:        <none>
  Volumes:         <none>
  Node-Selectors:  <none>
  Tolerations:     <none>
```

Here is how the actual YAML manifest file, or at least the part that would contain the `livenessProbe` configuration would look like, had you configured it manually, the information that we can see here is the same as one can observe in the describe command above

```
livenessProbe:
    httpGet:
        path: /
        port: 8080
        scheme: HTTP
    timeoutSeconds: 5
    periodSeconds: 10
    successThreshold: 1
    failureThreshold: 3
```

Creating readiness probes

Many apps need to perform any combination of the following before they are able to receive traffic, which increases the amount of time before an app is ready to do work. Some common tasks include

- Loading classes into memory
- Initializing datasets and databases
- Performing internal checks
- Establishing a connection to other containers or external service
- Finishing a startup sequence or other workflow

Fortunately, OpenShift also supports the concept of readiness probes, which ensures that the container is ready to receive traffic before marking the pod as active. Similar to the liveness probe a readiness probe is run at the container level in a pod and supports the same HTTPS, container execution, and TCP socket based checks like the liveness probe does. Unlike the liveness probe however, a failed readiness check doe not result in a new pod being deployed, if a readiness check fails the pod remains running while not receiving traffic. Let's run through an example of adding a readiness probe to the app-cli app, using the command line. For this readiness probe, you will tell OpenShift to look for a non-existent endpoint (that would simulate a startup delay between the container running, but the potentially heavy to start and setup app spinning up into a started state).

Looking for a URL that does not exist in your app-cli deployment will cause the readiness probe to fail. This exercise illustrates how OpenShift probe works when it runs into an undesired condition. Until a deployment passes a readiness probe it will not receive user requests. If it never passes the readiness probe, as in this example the deployment will fail and never be made available to users. To create the readiness probe use the command line and run the command:

```
# this will again as any other deployment update cause the old pods to be
   deleted and new ones will be spun up in their
# place which would represent the new state
$ oc set probe deployment/app-cli --readiness --get-url=http://:8080/notreal
   --initial-delay-seconds=5
deployment.apps/app-cli probes updated
```

The output includes a message that the deployment was updated. Just like a liveness probe, creating a readiness probe triggers the creation of a new app-cli deployment. Check to see whether the new pods were deployed by running the `oc get pods`

```
$ oc get pods

NAME                       READY STATUS    RESTARTS AGE
app-cli-1-build            0/1   Completed 0        8d
app-cli-59c7ff6b74-h7q8r 1/1   Running   0        14m
app-cli-59c7ff6b74-pnldz 1/1   Running   0        14m
app-cli-6d69c87c88-lm4tc 0/1   Running   0        67s
```

Note below that the liveness probe is fine, and is reporting that there are no failures while that is not true for the readiness probe, which is reporting a failed state, which is expected after having it configured with a non existing endpoint to check against

```
Name:                    app-cli
Pod Template:
  Labels:       deployment=app-cli
  Annotations:  openshift.io/generated-by: OpenShiftNewApp
  Containers:
   app-cli:
    Image:          image-registry.openshift-image-registry.svc:5000/image-
       uploader/app-cli@sha256:17
       c9ec389e0e130b3891e25e64b53cda350a6223732f62f56032b42cb361ffd1
    Ports:          8080/TCP, 8443/TCP
    Host Ports:     0/TCP, 0/TCP
    Liveness:       http-get http://:8080/ delay=5s timeout=1s period=10s #
       success=1 #failure=3
    Readiness:      http-get http://:8080/notreal delay=5s timeout=1s period
       =10s #success=1 #failure=3
    Environment:   <none>
    Mounts:         <none>
  Volumes:           <none>
  Node-Selectors:   <none>
  Tolerations:       <none>
```

The new pod is running, but will never be ready (1/1), that is because our probe was configured to check for a non existent URL, it is not ready to receive traffic. The previous pod is still running and receiving any incoming requests. Eventually the readiness probe will fail two more times and will meet the readiness probe `failureThreshold` metrics, which were set to 3 by default. As we discussed in the previous section, `failureThreshold` for a readiness or liveness probe sets the number of times a probe will be attempted before it is considered a failure

```
Note that the readiness probe will take 10 minutes to trigger a failed deployment. When
this happens the pod will be deleted, and the deployment will roll back to the old working
 configuration, resulting in a new pod without the readiness probe. You can modify the
default timeout parameters by changing the timeoutSeconds parameter as part of the spec.
strategy.*params in the deployment object. Deployment strategies are covered in greater
detail in future chapters
```

Once all three failures occur, the deployment is marked as failed, and OpenShift automatically reverts back to the previous deployment. The reason for the failure will be shown in the event view in OpenShift, that can be shown from the command line tool or the web console Because events are easier to read through the web console user interface, let us check that out there. Expand the panel in the events view, and you will be able to see the deployment has failed, and why that was, the reason will be stated as well.

```
# make sure to restore the readiness probe to point to the correct, endpoint,
    that is at least valid, that would drop
# the old pods, and make sure the newly created ones are all in a valid state
    where both the liveness and readiness probe
$ oc set probe deployment/app-cli --readiness --get-url=http://:8080/ --
   initial-delay-seconds=5
```

Auto-scaling with metrics

In the last section, we learned about the health and status of an app. You learned that OpenShift deployment use replication controllers - also called replication controllers, under the covers to ensure that a static number of pods is always running, that the desired state, configured by us, matches the actual state the cluster has. Readiness probes and liveness probes make sure running pods start as expected and behave as expected. The number of pods servicing a given workload can also be easily modified to anew static number with a single command or the click of a button.

This new deployment model gives you much better resource utilization than the traditional virtual machine model, but it is not a silver bullet, for operational efficiency. One of the big IT challenges with virtual machines is resource utilization. Traditionally when deploying virtual machines developers ask for much higher levels of CPU and RAM than are actually needed. Not only is making changes to the machine resources challenging, but many developers typically have no idea what types of resources are needed to run the app. Even at large companies like Google and Netflix predicting app workload demand is so challenging that tools are often used to scale the apps as needed.

Determining expected workloads

Imagine that you deployed a new app and it unexpectedly exploded in popularity, external monitoring tools notify you that you need more pods to run your app. Without any historical context there is no data to indicate how many new pods are needed tomorrow, next week or next month. A great example is Pokemon GO a popular mobile app that runs on Kubernetes. Within minutes of its release demand spiked well past expectations and over the opening weekend it became an international sensation. Without the ability to dynamically provision pods on demand, the game likely would have crashed, as millions of users started to overload the system. In OpenShift triggering horizontal pod scaling without human intervention is called autoscaling. Developers can set limits and objective measures to scale pods up and down on demand, and administrators can limit the number of pods to a defined range. The indicators that OpenShift uses to determine if the app needs more of fewer pods are based on pod metrics such as CPU and memory usage. But those pod metrics are not available out of the box, to use metrics in OpenShift the administrators must deploy the OpenShift metrics tack. This metric stack comprises several popular open source technologies, like `Prometheus, Hawkular, Heapster, and Apache Cassandra`. Once the metric stack is installed, OpenShift autoscaling has the objective measures it needs to scale pods up and down on demand.

The metric stack can also be deployed with the initial OpenShift installation by using the advanced installation option. The latest versions of OpenShift also have the option to deploy Prometheus, a popular open source monitoring and altering solution, to provide and visualize cluster metrics, in the future, Prometheus may be used as the default metric solution.

Installing OpenShift metrics

Installing the OpenShift metrics stack is straightforward, by default the pods that are used to collect and process metrics run in the open shift infra project, that was created by default during the installation. Switch to the open shift infra project, from the command line

```
# before running that command make sure you have switched and logged into the
    administrator user, otherwise the regular
# user does not have access and will not even be able to see system infra
   projects
$ oc console --credentials
$ oc login <cluster-ip> -u <kubeadmin> -p <adminpassword>

# this is the important bit, that would change the default cluster version
   object, which controls certain flags/switches
# determining which features are enabled or not, in this case we remove the
   first entry from the default overrides, which would restore the default
   behavior, and enable metrics
$ oc describe clusterversion/version
$ oc patch clusterversion/version --type='json' -p '[{"op":"remove", "path
   ":"/spec/overrides/0"}]'
$ oc get pods -n openshift-monitoring
```

```
alertmanager-main-0                                          0/6     Pending   0
           8m4s
cluster-monitoring-operator-6db6cb4c67-6gcmn                 1/1     Running   0
           8m23s
kube-state-metrics-79fb78866f-dvf4q                          3/3     Running   0
           8m5s
metrics-server-7d57d94644-2hbkw                              1/1     Running   0
           8m
monitoring-plugin-d9d9ccfc8-97z98                            0/1     Pending   0
           7m59s
node-exporter-xl7nq                                          2/2     Running   0
           8m5s
openshift-state-metrics-5476c76b54-qc49r                     3/3     Running   0
           8m5s
prometheus-k8s-0                                             0/6     Pending   0
           7m59s
prometheus-operator-55955cbfbc-c9q8n                         2/2     Running   0
           8m15s
prometheus-operator-admission-webhook-7cb5b87c4b-92jcz       1/1     Running   0
           8m19s
telemeter-client-bb985cdff-95fp8                             0/3     Pending   0
           2m51s
thanos-querier-67d44b859-29klf                               0/6     Pending   0
           8m3s
```

Understanding the metrics

In the previous section you successfully deployed the OpenShift metrics stack. Three types of pods were deployed to make this happen each with a different purpose, using technologies like Prometheus. But none of the pods generate metrics themselves. Those come from kubelets, a kubelet is a process that runs on each OpenShift node and coordinated which tasks the node should execute with the OpenShift master. As an example of an replication controller request that a pod be started, the OpenShift scheduler which runs on a master eventually tasks an OpenShift node to start the pod. The command to start the pod is passed to the kubelet process running on the assigned OpenShift node. One of the additional responsibilities of the kubelet is to expose the local metrics available, to the Linux kernel through an HTTPS endpoint. The OpenShift metrics pods use the metrics exposed by the kubelet on each OpenShift node as their data source. Although the kubelet exposes the metrics for individual nodes through HTTPS, no built in tools are available to aggregate this information and present a cluster wide view. This is where Prometheus comes in handy, it acts as the back end for metrics deployment, it queries the API server for the list of nodes and then queries each individual node to get the metrics for the entire cluster. It stores the metrics in its internal store data set. On the frontend the Prometheus pod processes the metrics. All metrics are exposed in the cluster through a common REST API to pull metrics into the OpenShift console. The API can also be used for integration into the third party tools or other monitoring solutions.

Using pod metrics & autoscaling

To implement pod autoscaling based around metrics you need a couple of simple things - first you need a metrics stack to pull and aggregate the metrics from the entire cluster and then make those metrics easily available. So far so good. Second you need an object to monitor the metrics and trigger the pod up and down. This object is called a Horizontal Pod Auto-scaler - HPA (Remember that abbreviation). And its main job is to define when OpenShift should change the number of replicas in an app deployment.

Creating the HPA object OpenShift provides a shortcut from the CLI to create the HPA object. This shortcut is available through the `oc autoscale command`. Switch to the CLI and use the following command

```
# note the scaling factor and conditions, we define the min and max number of
    replicas, and we also define where the
# replicas should be created, in this case when the CPU usage reaches a
   certain threshold
$ oc autoscale deployment/app-cli --min 2 --max 5 --cpu-percent=75
```

A couple of things happen when you run that command. First you trigger an automatic scale up to two app-cli pods by setting the minimum number of pods to 2. Run the following command to verify the number of app-cli pods. - `oc get pods`. We should be seeing at least two running pods in the ready state, immediately, even if we have not had any or just one replica. Second the HPA object was created for you. By default it has the same name as the Deployment object, app-cli this command gets the name of the HPA object created by the `oc autoscale`

```
# this will list all HPA objects, in our case only one deployment has it, and
    we also have only one deployment anyway.
$ oc get hpa
NAME      REFERENCE            TARGETS              MINPODS   MAXPODS
   REPLICAS   AGE
app-cli   Deployment/app-cli   cpu: <unknown>/75%   2         5         2
            6d

$ oc describe hpa/app-cli

Name:                                                    app-cli
Namespace:                                               image-uploader
Labels:                                                  <none>
Annotations:                                             <none>
CreationTimestamp:                                       <date>
Reference:                                               Deployment/app-cli
Metrics:                                                 ( current / target )
  resource cpu on pods  (as a percentage of request):   <unknown> / 75%
Min replicas:                                            2
Max replicas:                                            5
Deployment pods:                                         2 current / 2 desired
Conditions:
  Type           Status  Reason                   Message
  ----           ------  ------                   -------
  AbleToScale    True    SucceededGetScale        the HPA controller was able
      to get the target's current scale
  ScalingActive  False   FailedGetResourceMetric  the HPA was unable to
     compute the replica count: failed to get cpu utilization: missing
     request for cpu in container app-cli of Pod app-cli-d97b4c84b-vbmsg

Events:
  Type     Reason                        Age                         From
                           Message
  ----     ------                        ----                      ----
                            -------
  Warning  FailedGetResourceMetric       8s                        horizontal-
     pod-autoscaler  failed to get cpu utilization: missing request for cpu
     in container app-cli of Pod app-cli-d97b4c84b-vbmsg
```

The events sections displays some errors, especially around reporting the fact that the CPU utilization was not computed, note that we can see the `<unknown>` state in the table when we listed the HPA object above. The message states that: `failed to get cpu utilization: missing request for cpu in container app-cli of Pod app-cli-d97b4c84b-vbmsg`, meaning that the pods themselves were not configured to have any CPU resource limits, remember that above, we configured the auto-scaler, however that only tells OpenShift when to scale the pods, but since the pods have to resources restriction, how would any percentage of the CPU utilization be computed, we have to set a CPU utilization as well

In OpenShift a resource request is a threshold you can set that affects scheduling and quality of service. It essentially provides the minimum of resources guaranteed to the pod. For example a user can set a CPU request of four-tenths of a core written - 400 `milli cores` or 400m. This tells OpenShift to schedule the pod on nodes that can guarantee that three will always be at least 400m of CPU time. CPU is measured in units called milli cores. By default pods do not get individual cores they get time slices of CPU sharing the cores on the node with other pods. If a particular node has four CPU assigned to it, then 4000m are available, to all the running pods in that node.

Resource requests also can be combined with a resource limit which is similar to a request but sets the maximum amounts of resources guaranteed to the pod. Setting requests and limits also allows the user to set a quality of service level by default.

- `BestEffort` - Neither a resource nor a limit is specified this is for low priority apps that can live with very low amounts of processing and memory resources.
- `Burstable` - a request is set, indicating a minimum amount of resources allocated to the pod
- `Guaranteed` - a request and a limit are both set to the same number. This is for the highest priority apps that need the most consistent amount of computing power

Setting a lower quality of service gives the schedule more flexibility by allowing it to place more pods in the cluster. Setting a higher quality of service limits flexibility but, give apps more consistent resources. Because choosing the quality of service is about finding reasonable defaults most app should fall into the `Burstable`

Setting a CPU request, can be done by using the `oc set resources deployment/app-cli --requests=cpu=400m`. As with other changes to the deployment, config, this results in a new deployment config object, it will in turn create new pods which will then replace the current ones once the set of new pods produced by the new deployment go into the ready state. Now we can list again the HPA objects and see if there are any issues. After we have run the above, we should now be able to test the auto scaling

Testing the autoscaling setup

To demonstrate that autoscaling works as expected you need to trigger the CPU threshold that we have previously set. To help reach this mark use the Apache benchmark instance that comes pre-installed with `CentOS`, and is already available in your path. Before you run the benchmarking test, make sure you are logged in the open shift console in another window, so you can switch over to see pods being spun up and down. Then go to the overview page for the `image-uploader` project and run the command in the following:

```
# this will execute a total number of 50000 requests towards the pod, which
   will certainly cause overwhelming CPU usage,
# and force the OpenShift and monitoring service to scale more pods
$ ab -n 100000 -c 1000 http://app-cli-image-uploader.apps-crc.testing:8080/
```

```
# now if you are lucky and the threshold was hit we can describe the HPA
   object, and see the following in the events
# section, which shows that the pods were auto-scaled to 3 and then 4 based
   on CPU utilization requirements, the actual
# number will really depend on your local systems capabilities and such
$ oc describe hpa/app-cli
Normal SuccessfulRescale 105s horizontal-pod-autoscaler  New size: 3; reason:
    cpu resource utilization (percentage of request) above target
Normal SuccessfulRescale 105s horizontal-pod-autoscaler  New size: 4; reason:
    cpu resource utilization (percentage of request) above target

# eventually those will be down scaled, but that would take time, this is
   described in the following section, which resolves the issue with
# thrashing, the process of spinning up and down pods way too quickly over a
   short period of time , which OpenShift tries to avoid, do not
# wonder why those 3 or 4 pods that were auto-scaled might stay 3 or 4 and
   take some time to get back to the original 2 or 1 replicas that
# your deployment describes
```

Avoiding thrashing

When the Apache benchmark test kicked off, the OpenShift auto-scaler detected very high CPU usage on the deployed pods which violated the HAP constraints. This caused a new pods to be spun up on demand. Behind the scenes the deployment was modified creating a new number of replicas. After the tests were completed the CPU usage on the pods went back when the CPU spiked and the new pods spun up quickly, it tool several minutes for new pods to spin down. By default HAP synchronizes with the Prometheus metrics every 30 seconds. You can modify this sync period in the `master-config.yml`

This time window is by design to avoid something called thrashing, in OpenShift that is the constant starting and sopping of pods unnecessarily, Thrashing can cause wasted resource consumption because deploying new pods uses resources to schedule and deploy a pod on a new node, which often includes things like loading apps and libraries into memory. After OpenShift triggers an initial scale there is a forbidden window of time to prevent trashing. The rationale is that in practice if there is a need to constantly scale up and scale down within a matter of minutes, it is probably less expensive to keep the pods running than it is to continuously trigger scaling changes. In versions of OpenShift up to 3.6, the forbidden window is hard coded at 5 minutes to scale down the pods and 3 minutes to scale up the pods, in OpenShift versions higher than that, the default values are still 5 and 3 minutes respectively, but they can be modified via the `controllerManagerArgs`, and the `horizontal-pod-autoscaler-downscale-delay`.

Continuous integration & deployment

Deploying software into production is difficult one major challenge is adequately testing apps before they make it into production. And adequate testing requires one of the longest standing challenges in IT - consistent environments. For many organizations, it is time consuming to stand up new environments are for development, testing and QA and more. When the environments are finally in place they are often inconsistent. These inconsistencies develop over time due to poor configuration, management, partial fixes and fixing problems upstream such as directly making a patch in a production environment. Inconsistent environment can lead to unpredictable software. To eliminate the risk, organizations often schedule maintenance window during software deployments and then cross their fingers.

Over the last 15 years there have been many attempts to improve software processes. Most notable has been the industry wide effort to move from the waterfall method of deploying software to flexible approaches such as Agile that attempts to eliminate risk of performing many small iterative deployments as opposed to the massive software rollouts common with the waterfall method. But Agile falls short in several areas, because it focuses on software development and does not address the efficiency of the rest of the stakeholders in the organization. For example code may get to operations mode quickly but a massive bottleneck may result because operations now has to deploy code more frequently.

Many organizations are trying to solve these problems with a modern DevOps approach that brings together all the stakeholders to work jointly throughout the software development lifecycle. DevOps is now almost synonymous with automation, and Continuous integration (CI) & Continuous deployment (CD). Often short-handed to CI/CD. The delivery mechanism for implementing this is often called a software deployment pipeline in addition to technology that is cheaper focused largely on the technology aspect and how it related to containers

Container images are the centerpiece

From a technology perspective containers are becoming the most important technology in the software deployment pipeline. Developers can code apps and services without the need to design or even care about the underlying infrastructure. Operations teams can spend fewer resources designing the installation of apps. Apps and services can easily be moved not only between environments like QA testing and so on, in the software development pipeline but also between on premises and public cloud environments such as Amazon Web Services - `AWS`, `Microsoft Azure` and `Google Compute Platform (GCP)`.

When apps need to be modified developers package new container images, which include the app, configuration and runtime dependencies. The container then goes through the software deployment pipeline, automated, testing and processing. Using container images in a software deployment pipeline reduces risk because the exact same binary is run in every environment. If a change need to be made then it begin in the sandbox or development environments and the entire deployment process starts over. Because running containers are created from the container images, there is no such ting as fixing things upstream. If a developer operator attempted to circumvent the deployment process and path directly into production the change would not persist. The change must be made to the underlying container image. By making the container the centerpiece of the deployment pipeline system stability and app resiliency are greatly increased. When failures occur, identifying issues and rolling back software is quicker because the container can be rolled back to a previous version. This is much different than in previous approaches, where entire app servers and databases may need to be reconfigured in parallel to make the rollback of the entire system possible.

In addition, containers let developer run more meaningful testing earlier in the development cycle, because they have environments that mimic production, on their laptops. A developer can reasonable simulate a production environment load and performance testing on the container during development. The result is higher quality more reliable software updated. Better more efficient, testing also leads to less work in progress and fewer bottle necks, which means faster updates.

Promoting images

In this section you will build a full pipeline in OpenShift. To keep the promise of using the same binary in every environment you will build your image just once in your development environment. You will then use image tagging to indicate that the image is ready to be promoted to other projects. To facilitate this process you will use Jenkins and some additional OpenShift concepts which you will learn about as you go. Jenkins is an open source automation server that is commonly used as the backbone for CI/CD pipelines because it has many plugins for existing tools and technologies. Jenkins often becomes a Swiss army knife that is used to integrate disparate technologies into one pipeline

CI/CD:1 Creating an environment - TODO: this needs rewrite

The first part of any CI/CD pipeline is the development environment. Here, container images are built tested and then tagged if they pass their tests. All container build happen in this environment. You will use pre-built template to spin a up a simple app that runs on `Python`, and uses `MongoDB`, as a database. The template also provides an open source Git repository called `Gogs`, which comes pre-installed with the app already in it. `PostgresSQL`is also provided as a database for `Gogs`.

This section will make heavy use of OpenShift templates to install apps. An OpenShift template is essentially an array of objects, that can be parameterized and pun up on demand. In most cases the API objects created as part of the template are all part of the same app, but that is not a hard requirement. Using OpenShift templates provides several features that are not available if you manually import objects:

- Parametrized values can be provided at creation time.
- Values can be created dynamically based on regex values, such as randomly generated database password
- Messages can be displayed to the user in the console or on the CLI. Typically message include information on how to use the app
- You can create labels that can applied to all objects in the template.
- Part of the OpenShift API allows templates to be instantiated programmatically, and without a local copy of the template

OpenShift comes with many templates out of the box that you can see through the service catalog or by running `oc get templates - n openshift`. To see the raw templates files navigate to `/usr/share/openshit /examples`

```
# here is the abridged version of the list of the command mentioned above,
   that pulls the list of templates which are
# installed in the default OpenShift distribution, note that this list is
   about 1/3rd of the default templates, but
# there are even more on platforms like github distributed by regular people
   that can be installed and setup in your
# OpenShift cluster instance, proprietary ones certainly also exist
NAME                                          DESCRIPTION

   PARAMETERS        OBJECTS
jenkins-ephemeral                             Jenkins service, without
   persistent storage....                                    12 (all set)
        7
jenkins-ephemeral-monitored                   Jenkins service, without
   persistent storage. ...                                   13 (all set)
        8
jenkins-persistent                            Jenkins service, with
   persistent storage....                                       14 (all set)
        8
jenkins-persistent-monitored                  Jenkins service, with
   persistent storage. ...                                      15 (all set)
        9
mariadb-ephemeral                             MariaDB database service,
   without persistent storage. For more information ab...   8 (3 generated)
     3
mariadb-persistent                            MariaDB database service, with
   persistent storage. For more information about...   9 (3 generated)   4
mysql-ephemeral                               MySQL database service, without
```

```
    persistent storage. For more information abou...   8 (3 generated)   3
mysql-persistent                                      MySQL database service, with
   persistent storage. For more information about u...   9 (3 generated)   4
nginx-example                                         An example Nginx HTTP server
   and a reverse proxy (nginx) application that ser...   10 (3 blank)      5
nodejs-postgresql-example                             An example Node.js application
   with a PostgreSQL database. For more informati...   18 (4 blank)      8
nodejs-postgresql-persistent                          An example Node.js application
   with a PostgreSQL database. For more informati...   19 (4 blank)      9
openjdk-web-basic-s2i                                 An example Java application
   using OpenJDK. For more information about using t...   9 (1 blank)       5
postgresql-ephemeral                                  PostgreSQL database service,
   without persistent storage. For more information...   7 (2 generated)   3
postgresql-persistent                                 PostgreSQL database service,
   with persistent storage. For more information ab...   8 (2 generated)   4
rails-pgsql-persistent                                An example Rails application
   with a PostgreSQL database. For more information...   23 (4 blank)      9
rails-postgresql-example                              An example Rails application
   with a PostgreSQL database. For more information...   22 (4 blank)      8
react-web-app-example                                 Build a basic React Web
   Application                                                9 (1 blank)
         5
redis-ephemeral                                       Redis in-memory data structure
   store, without persistent storage. For more in...   5 (1 generated)   3
.....
```

At the command line lets create your development environment, by running the following, this will create the project and also on top of that will make sure to setup the necessary template to our CI/CD pipeline.

```
# first create the new project, that will be used throughout this section
$ oc new-project dev --display-name="ToDo App - DEV"

# we need to configure the template, this template is included in a file
   which we can simply apply
$ oc create -f openshift/dev-todo-app-template.json -n dev

# now to instantiate the template itself we can simply do this will basically
    create a new app using all the objects
# defined in the template, the template is nice because we can re-use it to
   duplicate apps easily, or use it as a way to
# replicate common app setups and configurations
$ oc new-app --template="dev-todo-app-flask-mongo-gogs"
```

Deployment strategies

So far in this section, you have learned how to build a container image and automate the promotion, of that image across different environments using native OpenShift automation in addition to Jenkins integration. But we have not discussed the exact sequence for ow the new version of the application is rolled out, The way you update your application in OpenShift is called a deployment strategy it is a critical component of supporting a wide variety of application in the platform. OpenShift supports several deployment strategies including the following:

- Rolling - the default strategy. When pods consisting of the new image become ready by passing their readiness checks, they slowly replace the old images, one by one, setting this deployment strategy is done in the deployment config object
- Re-create - scales down to zero pods consisting of the old image and then begin to deploy the new pods. This strategy has the cost of a brief downtime while waiting for the new pods to be spun up. Similar to rolling in order to use this strategy it must be set in the deployment configuration object.
- Blue/Green focuses on reducing risk by standing up the pods consisting of new images while the pods with the old images remain running, this allows the user to test their code in a production environment. When the code has been fully tested all new requests are sent to the new deployment. OpenShift implements this strategy using routes.
- Canary - adds checkpoints to the blue/green strategy by rolling out a fraction of the new images at a time and stopping. The user can test the application adequately before rolling out more pods. As with blue/green deployments this strategy is implemented using OpenShift routes.
- Dark launches - rolls out new code but does not make it available to users. By testing how the new code works in production, the users can then later enable the features when its determined to be safe. This strategy has been made popular at place like Facebook and Google. To accomplish dark launches the application code must have checks for certain environment variables that are used to enable the new features. In OpenShift you can take advantage of that code by toggling the new features on or off by setting the appropriate environment variables for the application deployment.

  There are many considerations for choosing a deployment strategy. The rolling strategy upgrades your application the most quickly while avoiding downtime, but it runs your old code side-by-side with your new code. For many stateful application such as clustered application and databases this can be problematic. For example imagine that your new deployment has any of the following characteristics

- It is rolling out a new database schema
- it has a long running transaction
- it shared persistent storage among all the pods in the deployment
- it uses clustering to dynamically discover other pods,

In these cases it makes sense to use a re-create strategy instead of a rolling update. Databases almost always use the re-create strategy. You can check the strategy for the pod by doing a quick inspect of the deployment object for the target app. Stateless applications is a good fir for a rolling upgrade whereas databases make more sense to run using the re-create strategy. Both the rolling and the re-create strategies have extensible options including various parameters to determine the timing of the rollouts they also provide lifecycle hooks which allow code to be injected during the deployment process. Many users also choose to add blue green and canary style deployment strategies by combining the power of OpenShift routes with a rolling update or re create strategies, for application using rolling deployment adding a blue/green or canary style deployment allows the OpenShift user to reduce risk by providing a more controlled rollout using checkpoints. For application using the re-create deployment strategy adding blue/green or canary features lets the application avoid downtime. Both the blue/green and canary deployments use OpenShift routes to manage traffic across multiple services. To implement these deployment strategies an entire copy of the application is created with the new code. This copy includes the objects to run the application the deployment, service, replication controller, pods and so on. When adequate testing has been performed on the new code, the OpenShift route is patched to point to the service containing the new code. A blue/green deployment has the added benefit of testing code in production and because the old code is still running the app can be rolled back to the old code, if something breaks. Once downside to using blue/green deployments is that they require more infrastructure, because your application needs to double the resources while both versions of the code are running at the same time. A canary deployment is similar to a blue/green one except that whereas blue/green switches the route between services all at once, canary uses, weights to determine what percentage of traffic should go to the new and old services. You can modify the weights for the rollout using the OpenShift command line interface tool or the console.

Here is a brief summary of the topics we have covered, we can say: That image streams enable automation and consistency for container images, you can use OpenShift triggers for even based image builds and deploys you can use DNS for service discovery. You can use environment variables for service discovery if dependencies are installed first. Image tagging automates the promotion of images between environments. Secrets mask sensitive data that needs to be decoupled from an image. Config maps provide startup arguments environment variables or files mounted in an image, OpenShift provides a Jenkins instance or a template with many useful plugins pre-installed, allowing regular Jenkinsfile pipelines to be executed and monitored form the OpenShift console, OpenShift supports many types of deployment strategies for a wide variety of applications.

Stateful applications

The very first application we have deployed in OpenShift, the php web app that we have deployed, is the `image uploader`, Here are a few additional application features:

- Shows you the full size image when you click it.
- Shows you these images as thumbnails on the application page
- Uploads images from your workstation
- Verifies that what you are uploading is a standard image format

It is not the next Instagram, but it is a simple enough to live in a couple of files of source code and be easy to edit and manipulate in OpenShift. You will use that simplicity to your advantage in this section. If you have not already go ahead and test out the image app, and upload a few images. After you do that your application should show these images on the home page as thumbnails

When you deploy an application in OpenShift you can specify the minimum number of replicas instances of the application to keep running all the times If you do not specify a number, OpenShift will always keep at least one instance of your app running at all times. We initially discussed this in earlier sections, and used the feature in a more recent sections to scale up the image to more than one replicas. None of these replicas had persistent storage though. If one of the application pods was deleted or scaled down any data it had written would be gone as well. We can test this one.

Container storage

After logging into your OpenShift cluster from the command line with the oc command line tool you can use the `oc get pods` command to get a list of all your running pods. The output of that will show that the app-cli has a few running pods, and a few completed ones, which are the build ones. The running ones are the actual application pods, as we already know. After that, lets delete one of the pods, or even both, using the `oc delete pod <pod-id>`

Even after we delete one OpenShift will make sure that the desired state, meaning at least 2 replicas is fulfiled, therefore it will scale up the application, creating a new pod in place of the one we have just deleted, If we run the `oc get pods` one more time we will see that the new pod has a new `age` which is about 10 or so seconds, this is the time it took us to list the pods again, and in between that time OpenShift already created the new pod in place of the one we deleted

```
# here is a short snippet that demonstrates the steps from above.
$ oc login <cluster-ip> -u <kubeadmin> -p <adminpassword>
$ oc get pods
$ oc delete pod
$ oc get pods
```

Now that would seem to be the answer to just about everything wouldn't it. Applications automatically restart themselves when they go down. But before we close up shop take another look at your application web page. We can see that the images we have initially uploaded earlier are nowhere to be found , When a pod is deployed in OpenShift the storage that it used for the file system is ephemeral - it does not carry over from one instance of the application to the next. When application needs to be more permanent, you need to set up a persistent storage for use in OpenShift

Handling permanency

In OpenShift persistent storage is available for data that needs to be shared with other pods or needs to persist pas the lifetime of any particular pod. Creating such permanent storage in pods is handles by persistent volumes - PV. These `PVs` in OpenShift use industry standard network based storage solutions to manage persistent data. OpenShift can use a long list of storage solutions to create persistent volumes - including, but not limited to the following:

- `CSI` - the container storage interface defines a standard interface for container orchestration systems, like Kubernetes, to expose arbitrary storage systems to their container workloads. Once a CSI compatible volume driver is deployed on a Kubernetes cluster, users may use the CSI volume type to attach or mount the volumes exposed by the CSI driver. A CSI volume can be used in a Pod just like any other, we can have the pod reference a persistent volume claim, which itself is referencing a persistent volume which itself defines the CSI driver to use.

- `NFS` - these types of volumes allow an existing NFS (Networking File System) share to be mounted into a Pod, Unlike `emptyDir` which is erased when a Pod is removed the contents of an NFS volume are preserved and the volume is merely unmounted. This means that an NFS volume can be pre-populated with data and that data can be shared between pods. NFS can be mounted by multiple writers simultaneously. You must have your own NFS server running with the share exported before you can use it, you can not specify NFS mount options in the Pod spec, these have to be specified in the persistent volume document itself, for the volume.

- `HostPath` - local directories on the OpenShift nodes themselves, which are NOT ephemeral, `hostPath` volume mounts a file or directory from the host node file system into your pod. That is not something that most Pods will need, but it offers a powerful escape hatch for some application

- `Local` - a local volume represents a mounted local storage device such as a disk partition or directory, Local volumes can only be used as statically created persistent volume objects, dynamic provisioning is not supported, compared to `hostPath` volumes, local volumes are used in a durable and portable manner without manually scheduling pods to nodes. These system is aware of the volume's node constraints by looking at the node affinity on the Persistent volume, however local volumes are subject to the availability of the underlying node and are not suitable for all applications. The pod using this volume is unable to run. Applications using the local volumes must be able to tolerate this reduced availability as well as potential data loss, depending on the durability of the underlying disk

Note that using the `HostPath` volume type presents many security risk. If you can avoid using a `hostPath` volume you should. For example define a local `PersistentVolume` instead, and use that. If you are restricting the access to specific directories on the node, using admission time validation, that restriction is only effective when you additionally require that any mounts of that `hostPath` volume are read only. If you allow a read-write mount of any host path by an untrusted Pod, the containers in that pod may be able to subvert the read-write host mount. Take care when using `hostPath` volumes, whether these are mounted as read-only or as read-write because:

- Access to the host file system can expose privileged system credentials such as for the kubelet or privileged API keys.

- Pod with identical configuration such as created from a Pod template, may behave differently on different nodes due to different file on the nodes
- `hostPath` volume usage is not treated as ephemeral storage usage. You need to monitor the disk usage by yourself because excessive `hostPath` disk usage will lead to disk pressure on the node

In the next chapter we will configure a persistent volume in OpenShift using the network file system - HostPath

Creating resources

In OpenShift they rely on the listed types of network storage to make the storage available across all nodes in a cluster. For the examples in the next few sections you will use a persistent volume built with NFS storage. First we have to export a NFS volume on your OpenShift master. As we discussed early on in the sections your OpenShift cluster is currently configured to allow any user to log into the system as long as their password is not empty. Each new username is added to a local database, at first login. You created a user named dev and used that user to create a project and deployed the app-cli. The dev user can create projects and deploy apps but it does not have the proper permissions to make a cluster wide changes like attaching a persistent volume. We will take a much deeper look at how users are managed in OpenShift in future sections, but to create a persistent volume we need an admin level access user in the OpenShift cluster, luckily we can get that.

```
# just to ensure we do not run into issues with some commands
$ sudo -i

# list the available disks, this is just for information purposes, we can see
    that we have one main partition, which
# is vda4, the rest is system storage, for the virtual machine, such as /boot
$ lsblk
vda1 252:1    0    1M  0 part
vda2 252:2    0  127M  0 part
vda3 252:3    0  384M  0 part /boot
vda4 252:4    0 99.5G  0 part /var/lib/kubelet/pods/6994529f-6383-4141-8f1c
   -48cd7189ee3c/volumes/kubernetes.io~csi/pvc-3849f616-7f27-47d4-b0f8-7
   c98897aa529/mount
                                /var/lib/kubelet/pods/14c7b420-5ed8-49d6-829f
                                   -af7f14474200/volume-subpaths/nginx-conf/
                                   networking-console-plugin/1
                                /var
                                /sysroot/ostree/deploy/rhcos/var
                                /usr
                                /etc
                                /
                                /sysroot

# make the directory where we will store the network file system data, also
   remount /var to be extra sure
$ mount -o remount,rw /var
$ mkdir /var/nfs-share

# we care about the block identifier for our storage, in this case as already
    mentioned that would be vda4
$ blkid | grep -i vda4
```

```
/dev/vda4: LABEL="root" UUID="3ee0fdbd-7a71-4158-a0f8-db54b07fa6af" TYPE="xfs
   " PARTLABEL="root" PARTUUID="a8432eef-31a4-7b42-a1fd-768a79c7c61d"

# check the firewall rules from iptable, if you do not get any output, the
   second command, that will expose the NFS
# server port in the firewall rules
$ iptables -L -v -n | grep 2049
$ iptables -I INPUT -p tcp --dport 2049 -j ACCEPT

# NFS is actually a collection of four services that you need to enable and
   start:
# - -rpcbindNFS uses the RPC protocol to transfer data.
# - nfs--serverThe NFS server service.
# - nfs--lockHandles file locking for NFS volumes.
# - nfs--idmapHandles user and group mapping for NFS volumes.
systemctl enable rpcbind nfs-server nfs-lock nfs-idmap;
systemctl start rpcbind nfs-server nfs-lock nfs-idmap;
```

Creating storage

Now that we have created the persistent volumes it is time to take advantage of them in OpenShift application consume persistent storage using persistent volume claims. A persistent volume claim - PVC - can bedded into an application as volume using the command line or through the web interface, let us create one PVC first, on the command line and add it to the application

The way the persistent volume claims match up with persistent volumes, first you need to know how persistent volumes and persistent volume claims match up to each other. In OpenShift PV represent the available storage, while the PVC, represent an application need for that storage - the claim for the storage. When you create a PVC, OpenShift look for the best fit among the available PV and reserves it for use by the PVC. In the example environment matches are based on two criteria :

- Persistent Volume Size - OpenShift tries to take best advantage of available resources, when a PVC is created it reserves the smallest PV available that satisfies its needs.
- Data Access Mode - when a PVC is created OpenShift look for an available PV with at least the level of access required if an exact match is not available it reserves a PV with more privileges that still satisfies the requirements for example if a PVC is looking for a PV with a RWO (read/write) access mode it will use a PV with a RWX (read/write many) access mode if one is available.

Because all the persistent volumes in your environment are the same size matching them to a PVC will be straightforward, next you will create a PVC for your application to use.

First have to enable the NFS capabilities on the server, meaning that we have to create, and then enable the mount paths, that we are going to be using in the persistent volume. The script below does just that, you will notice that it creates 5 directories which can be used by 5 different PVC, they are simply named starting from `pv01` to `pv05`, and are under the directory - `/var/nfs-share`.

```
#!/bin/sh

# create a file with the following contents, on the crc node itself, name it
   nfs.sh, and execute it, make the file
# executable with chmod # +x nfs.sh, run this script as sudo, like so sudo ./
   nfs.sh. This will make the needed
```

```
# directories and export them as nfs volumes which can later on be mounted
   and used by our persistent volume and the
# persistent claims
NUM_PVS=5
PREFIX="app-cli"

# go over the predefined number of persistent volumes to create, make a new
   directory for each one of them, under the
# specified path, note that this path is matched exactly in the creation of
   persistent volume object above
for i in $(seq -f "%02g" 1 ${NUM_PVS})
do
    mkdir -p "/var/nfs-share/${PREFIX}-pv${i}"
    echo "/var/nfs-share/${PREFIX}-pv${i} *(rw,sync,no_root_squash)" >> /etc/
       exports
done

# make sure that the permissions for the directories are lax, to ensure that
   the directories can be written to and read
# from, otherwise the container might not be able to save or read any data at
    all
chmod -R 777 /var/nfs-share
chown -R nfsnobody:nfsnobody /var/nfs-share
# show what was exported at the end, you should see all the directories that
   we exported above, note that these are
# also going to be used later on in the manifest files for storage
exportfs
```

```
# make sure to make the script executable after which use sudo to execute the
    script as shown below
$ chmod +x nfs.sh
$ sudo ./nfs.sh

# to checkout what was exported, you can cat out the file exports, and see
   all the volumes in there
$ cat /etc/exports
/var/nfs-share/app-cli-pv01 *(rw,sync,no_root_squash)
/var/nfs-share/app-cli-pv02 *(rw,sync,no_root_squash)
/var/nfs-share/app-cli-pv03 *(rw,sync,no_root_squash)
/var/nfs-share/app-cli-pv04 *(rw,sync,no_root_squash)
/var/nfs-share/app-cli-pv05 *(rw,sync,no_root_squash)
```

Logging in as `kubeadmin`

When an OpenShift cluster is installed it creates a config file for a special user names kubeadmin or system:admin, depending on which OpenShift distribution we are using. The admin user is authenticated using an SSL certificate regardless of the authentication provider that is configured. Admin user has full administrative privileges on an OpenShift cluster. The key certificate for admin are placed in the Kubernetes config files in `~/.kube`, when the OpenShift cluster is installed., this makes it easier to run commands as admin. It is also possible to list the credentials of the users from the command line directly, or you can even observe the credentials being logged out to the terminal when the cluster is being started

```
# you might see something like this when the cluster is being started for the
    first time, otherwise refer to the help
# documentation of the cluster tool you are using, that will point you to the
    right sub command to use to correctly
# extract the credentials for the admin user

The server is accessible via web console at:
  https://console-openshift-console.apps-crc.testing

Log in as administrator:
  Username: kubeadmin
  Password: pottH-ZrwmV-KscNd-CZheg

Log in as user:
  Username: developer
  Password: developer
```

The example above, shows the example output for the `crc` deployment method of the OpenShift cluster, but similar output will be observed for the `minishift` deployment method, you have to take a note of the IP address of the cluster, since that will your point of entry to the API server and the user interface and console

If you are unable to locate the password for the admin or developer, and you are using the `crc` or `minishift` methods, checkout your home directory for folders named `.crc` and `.minishift`, therein you will be able to find the details about the credentials stored in files. These are usually located in the following locations on your host machine - `$HOME/.crc/machines/crc/kubeadmin-password` or `$HOME/.minishift/machines/minishift/TODO`

Physical volume

To create a resources from a YAML template use the `oc create` or `oc apply` command along with the `-f parameter`, which will specify the template file you want to apply or process. To create the persistent volume for this example you will use the template block above, save it to a file, and run the command. Therefore the command might look something like that - `oc apply -f <directory/pv01.yml>`. What are some of the options used in the template above.

The Access mode of the persistent volume, dictates how the different pods can access the persistent volume we have created, this is mostly to be able to reasonably deal with race conditions and resource locks, that might occur while data is being accessed - in any capacity, read or write

- Read/Write (RWO) - This volume can be mounted as read write by a single node, in the OpenShift cluster, this is a useful when you have workloads where a single application pod will be writing data. An example is a relational database, when you know that all the writes to the persistent data will come from a single pod, that would be the pod that is running database server/service
- Read/Only Many (ROX) - This is for volumes where the volume can be mounted as read only by multiple OpenShift nodes. An example of where this type of access mode is useful is when a horizontally scalable web application needs access to the same static content such as images.
- Read/Write Many (RWX) - That access mode is the option you will use for the persistent volume in this section. It allows multiple nodes to mount this volume read from it and write to it. The `image uploader` application is a good example. When you scale up the application in the next section multiple nodes will need to be able to read and write to the persistent storage you are about to create

A reclaim policy dictates how the persistent volume reclaims space after a storage claim on the persistent volume is no longer required. Two options are available:

- Retain - with this reclaim policy all data is retained in the volume, reclaiming space is a manual process
- Recycle - this reclaim policy automatically removed data when the claim is deleted, this one is the one we will be using for the persistent volume created for this section

Creating volumes

OpenShift makes extensive use of the configuration files written in YAML. These files are a human readable language that is often used for configuration files and to serialize data in a way that is easy for both humans and computers to consume. YAML is the default way to push data into and get data into Kubernetes and by proxy into OpenShift. In previous sections we have talked about OpenShift resources that are created when an application is built and deployed. These resources have documented YAML formatted templates so you can create and manage the resources easily. In later sections you will use several of these templates to create or change resources in OpenShift. For the application deployments you created in earlier sections, these templates were automatically generated and stored by OpenShift, when we created the new application or in other words when we run the `new-app` command.

In this section you will use the template to create a persistent volume, this template is more like a specification, that even provides a version, that version tells Kubernetes which version of the specification revision we are using to create the given resource object, different revision versions might have some differences in the general layout and structure of the YAML specification file.

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
    name: pv01
spec:
    capacity:
        storage: 2Gi
    accessModes:
        - ReadWriteMany
    nfs:
        # The IP address here refers to the IP address of the NFS server
        # In our case we can use the address of the cluster node we
        # can try nslookup from the node - nslookup api.crc.testing
        # server: <cluster-ip-address>
        # path: <path-inside-cluster-node>
        server: 192.168.127.2
        path: /var/nfs-share/app-cli-pv01
    persistentVolumeReclaimPolicy: Recycle
```

```
# we can now see the newly created persistent volume which points to the nfs-
   directory which would be setup in the
# next section, make sure that these directories indeed exist after
   completing the next section where they are setup
$ oc get pv

NAME  CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
pv01  2Gi        RWX            Recycle          Available
```

So what type of information does this file or template contain really:

- At the very start as we already mentioned, is the version of the object, in this case it is under revision `v1`.
- First it is the type of resource the template will create different resources have different templates configurations in this case you are creating a persistent volume
- A name for the resource to be created, this is simply `pv01`, this is mandatory from a Kubernetes point of view, each object of a given type/kind needs a unique name
- Storage capacity for the persistent volume, that will be created measured in GB, in this example, each of the persistent volumes is 2GB
- Next is the access mode of the volume, we will dig into deeper later on
- NFS path for this persistent volume
- NFS server for this persistent volume , if you used another IP address for your master or used another server, you will need to edit this value,
- Recycle policy for the persistent volume that will be created. These policies dictate what and how data will be disposed of once it is no longer being used by an application, we will dig deeper into those in the next sections

Creating claims

Now we will do something very similar to what we have already done for the persistent volume, create it from a template, again using YAML as the base manifest format, that will be fill up all the necessary fields that are required to attach our PVC to the application, and by proxy match that PVC with the persistent volume that we have already created for the cluster. There are few important parameters to take a note of:

- The name of the PVC to be created, this is important, as it is core parameter for all Kubernetes and by proxy/extension OpenShift objects, we can not really make one without an unique name
- The access mode of the PVC, in this example the PVC will request the RWX (read/write many) this aligns with the volumes we have already crated in previous section, which were all RWX.
- The size of the storage required, this example creates a 2GB storage request which matches the size of the persistent volume that you created in the previous section

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
    name: pvc01
spec:
    # we are forcing this claim to bind to the pv01, this is usually not
       needed, but we would like to make sure that the
    # claim will correctly be using the right persistent volume we have
       already created, the storageClassName is mandatory
    # since otherwise this will error out on running apply
    volumeName: pv01
    storageClassName: ""
    accessModes:
        - ReadWriteMany
    resources:
        requests:
            storage: 2Gi
```

We can then use the regular `oc` command to apply this configuration, first save that one into a file, and then we can simply run the `oc apply -f pv01.yml`, note that unlike the persistent volumes we are not required to use an admin user, the volumes are meant to be created by admins, but the claims are created for pods, by developers, usually what happens when a new deployment config is created is that the pod will refer to the volume claim by name, which should be created by the developers first.

The one rule to remember is that the PVC needs to be created in the same project as the project for which it will provide storage. When the PVC is created it queries OpenShift to get available persistent volumes. It uses the criteria described to find the best match and then reserves that persistent volume, once that is done it can take a minute or so depending on the size of your cluster, for the PVC to become available to be used in an app as persistent storage. The following command shows how you can use the `oc` command line tool to provide information about the active persistent volume claims, in an OpenShift project.

```
# Note the following event, once the persistent volume claim is created, it
   will be lazily initialized, meaning that it
# will not be 'created' per se, until at least one pod is actually using it,
   therefore it will stay in pending status
# until that happens

  Type    Reason                  Age                     From
                             Message
  ----    ------                  ----                    ----
                             -------
  Normal  WaitForFirstConsumer  7s (x14 over 3m16s)  persistentvolume-
     controller  waiting for first consumer to be created before binding
```

```
# note that we have to first make sure that the currently set project is the
   one we are going to work with, either use
# the -n flag when calling the command below, or just simply use oc project <
   project-name>, to first set the currently
# active project context
$ oc get pvc

NAME          STORAGECLASS                     VOLUMEATTRIBUTESCLASS   AGE
pvc-app-cli   crc-csi-hostpath-provisioner   <unset>                 27s
```

A PVC represents reserved storage available to the applications in your project. But it is not yet mounted into an active application. To accomplish that you need to mount your newly created PVC into an application as a volume.

Modifying deployment

In OpenShift a volume is any filesystem file or data mounted into an application pod to provide persistent data. In this section we are concerned with persistent storage volumes. Volumes also are used to provide encrypted data, application configuration and other types of data as you saw in earlier sections. To add a volume you use the `oc volumes command`, The following example takes the newly created PVC and adds it into the app-cli application - the command is broken over multiple lines to make it more readable and understandable

```
# note that we need 3 primary things here, first is we need to tell it the
   type of volume, the name of the volume, and
# where the volume will be mounted, meaning which path in the container will
   be linked to the volume, reading and writing
```

```
# under that path will actually directly write/read to the persistent volume
   in this case the NFS server
$ oc set volume deployment/app-cli --add \
        --claim-name=pvc01 \
        --type=PersistentVolumeClaim \
        --mount-path=/opt/app-root/src/uploads

# after this if we do describe the deployment we will be able to see the
   following output, the rest of it is removed, and only the volume
   information is shown, which allows us to see which claim our deployment is
    working with, and also the name of the volume which is as mentioned
   automatically generated
$ oc describe deployment/app-cli

...
  Volumes:
   volume-rpw2z:
    Type:          PersistentVolumeClaim (a reference to a
       PersistentVolumeClaim in the same namespace)
    ClaimName:     pvc01
    ReadOnly:      false
...
```

By applying the volume to the deployment, for app-cli you can trigger a redeployment of the application automatically to incorporate the new persistent volume, the following parameters specified above are required and mandatory, as mentioned in the comment, they define the minimal number of known arguments that need to be specified for the volume to be created for our application.

Optionally you can also specify a name of the volume itself, with the –name parameter. If it is not set, OpenShift creates on dynamically, typically simply as a sequence of letters and numbers

Using the `oc` describe command to list the details of the deployment we can now see that the volume is attached to the deployment and more precisely to the pod itself

```
# we can see below that the volume is attached to the pod, through the
   deployment
$ oc describe deployment/app-cli

# Here is an excerpt from from the describe above, we can see that the volume
    was created, and it was linked to the
# persistent volume claim we have already created, and in turn that claim is
   linked to the persistent volume object
# which is an NFS volume configured on the cluster node
  Volumes:
   volume-plc22:
    Type:          PersistentVolumeClaim (a reference to a
       PersistentVolumeClaim in the same namespace)
    ClaimName:     app-cli
    ReadOnly:      false
```

```
# and this is how part of the spec of the deployment for the pod will look
   like, a volume is attached to the deployment, in this case a volume is
   given an automatic name, what is important here is that the volume is
   linked to a claim, the claim we already created above
spec:
    volumes:
        - name: volume-plc22
          persistentVolumeClaim:
              claimName: app-cli
```

Because we have created the persistent volume and the claims to allow for multiple reads and writes by using the RWX access mode, when this application scale horizontally each new pod will mount the same PVC, and be able to read data from and write data to it. To sum up you just modified your containerized application to provide horizontally scalable persistent storage.

```
Finally go into the application through your browser, visit the following web page for our
 image upload application - http://app-cli-image-uploader.apps-crc.testing/, upload a new
 image in there, and follow up with the next sections, remember up the name of the file
you upload we will use that as reference to validate that indeed the file was correctly
uploaded to the host / node, under the correct directory we specified in the persistent
claim
```

Volume mounts

Because we are using NFS server, exports as the source for the persistent volume, it stands to reason that somewhere on the OpenShift nodes, those NFS volumes are mounted and created. You can see that this is the case by looking at the following example. SSH into the OpenShift node, locally where the containers are running, run the mount command and search for the mounted volumes from the IP address of the NFS server. The IP address of the OpenShift cluster should be format of `192.168.XXX.XXX`

```
# after having ssh access to the node, we can run the mount command, replace
   the cluster-node-ipaddress with the ip
# address of the cluster node, we can obtain the cluster node ip address as
   shown above, using nslookup against
# the api.crc.testing host, since the /etc/resolv.conf is configured with the
    cluster dns service, therefore the
# host will be resolved to a correct host
$ mount | grep 192.168.127.2

192.168.127.2:/var/nfs-share/app-cli-pv01 on /var/lib/kubelet/pods/952b5bd5-
   c8f2-4da5-9b62-5b58d355f6e5/volumes/kubernetes.io~nfs/pv01 type nfs4 (rw,
   relatime,vers=4.2,rsize=1048576,wsize=1048576,namlen=255,hard,proto=tcp,
   timeo=600,ret
rans=2,sec=sys,clientaddr=192.168.127.2,local_lock=none,addr=192.168.127.2)
192.168.127.2:/var/nfs-share/app-cli-pv01 on /var/lib/kubelet/pods/861fcda4-
   d641-4f8b-9bdf-8c012f5199f2/volumes/kubernetes.io~nfs/pv01 type nfs4 (rw,
   relatime,vers=4.2,rsize=1048576,wsize=1048576,namlen=255,hard,proto=tcp,
   timeo=600,ret
rans=2,sec=sys,clientaddr=192.168.127.2,local_lock=none,addr=192.168.127.2)
```

You will get multiple results, each of those will be for each pod that is running for the application, earlier in this section you used the `oc get pv` command and confirmed that the PV was being used by the app-cli, but that does not explain how the NFS volume is made available in the app-cli container's mount namespace.

Earlier in previous section, we had a look at how the file system in a container is isolated from the rest of the containers that are running on the cluster host, and the cluster host/node itself, through the use of namespaces and more precisely the mount namespace which is one of the corner stone namespaces that the Linux kernel uses when dealing with containerized applications. The persistent volume mount however, is not added to the mount namespace of the app-cli containers. Instead the NFS mount is made available in the container using a technology called `bind mount`

A bind mount, in a Linux is a special type of mounted volume where part fo the file system is mounted in a new new additional location. For app-cli the NFS mount for the persistent volume is mounted using a bind mount at /opt/app-root/src/uploads in the container mount namespace, using ta bind mount makes the content available simultaneously in two locations. A change in one location is automatically reflected in other location.

Bind mounts are used for volumes for two primary reasons. First creating a bind volume mount on a Linux system is a lightweight operation in terms of performance and CPU requirements. That means redeploying a new container to replace an old one does not involve remounting a remote volume. This keeps container creation time low.

Second this approach separates concerns for persistent storage, using bind mounted the container definition does not have to include specific information about the remote volume. The container only needs to define the name of the volume to mount. OpenShift abstract how to access the remote volume and make it available in the containers. The separation of concerns between administration and usage of a cluster is consistent OpenShift design feature.

```
# finally to show the magic in play, from the cluster node, you can navigate
   to the directory /var/nfs-share/app-cli-pv01,
# there you will find the uploaded file / image, now  we have successfully
   mounted a persistent volume claim from the
# host through the use of a persistent volume object
$ cd /var/nfs-share/app-cli-pv01 && ls -la

total 3972
drwxrwxrwx. 2 nfsnobody  nfsnobody      48 Jun 11 14:23 .
drwxrwxrwx. 7 nfsnobody  nfsnobody     106 Jun 11 13:40 ..
-rw-r--r--. 1 1000660000 root      4063862 Jun 11 14:23 image.jpg
```

The goal of this section was to walk you through configuring the components that make a persistent storage available to a container in OpenShift. In the following sections, we will use persistent storage to create more scalable and resilient applications.

Stateful applications

In previous sections we have created persistent storage for the image upload pods, which allowed data to persist past the lifecycle of a single pod. When a pod failed a new pod spun up in its place and mounted the existing persistent volume locally. Persistent storage in OpenShift allows many stateful applications to run in containers. Many other stateful applications still have requirements that are unsatisfied by persistent storage alone, for instance many workloads distribute data through replication which required application level clustering. In OpenShift this type of data replication requires direct pod to pod networking without going through the service layer. It is also very common for stateful applications such as databases to have their own custom load balancing and discovery algorithms, which require direct pod to pod access. Other common requirements for stateful applications include the ability to support sticky services and sessions as well as implement a predictable graceful shutdown.

One of the main goals of the OpenShift container platform is to be a world class platform for stateless and stateful applications. In order to support stateful applications a variety of tools are available to make virtually any application container native. This chapter will walk you through the most popular tools including headless services, sticky sessions pod discovery techniques and stateful sets, just to name a few. At the end of this section you will walk through the power the stateful set, which bring many stateful applications to life on OpenShift

Enabling a headless services

A good example of application clustering in everyday life is demonstrated by amazon's virtual shopping cart. Amazon customers browse for items and add them to a virtual shopping cart so they can potentially be purchased later. If an Amazon user is signed in to their account their virtual shopping cart will be persisted permanently because the data is stored in a database. But for users who are not signed in to an account the shopping cart is temporary. The temporary cart is implemented as in-memory cache in amazon's data centers. By taking advantage of an in-memory caching end users get fast performance which results in a better user experience. Once downside of using in-memory caching is that if a server crashed that data is lost. A common solution to this problem is data replication - when an application puts data in memory that data can be replicated to many different caches, which result in fast performance and redundancy.

Before applications can replicate data among one another, they need a way to dynamically find each other. In previous sections, this concept was covered through the use of the service discovery in which pods use OpenShift service object. The OpenShift service object provides a stable IP and port that can be used to access one or more pods. For most cases having a stable IP and port to access one or more replicated pods is all that is required. But many types of applications such as thsoe that replicate data require the ability to find all the pods in a service and access each one directly, on demand.

One working solution would be to use a single service object for each pod, giving the application a stable IP and port for each pod. Although this works nicely, it is not ideal because it can generate many service object which can become difficult to manage. A better solution is to implement a headless service and discover the application pod using an application specific discovery metrics. A headless service is a service object that does not load balance or proxy between backend pods. it is implemented by the setting - `spec.clusterIP` field to None, in the service API object.

Headless services are most often used for application that need to access specific pods directly without going through the service proxy. Two common examples of headless service are clustered databases and applications that have a client side load balancing login built into them or the code. Later in this chapter we will explore an example of a headless service using MongoDB a popular NoSQL database.

Application clustering with WildFly

In this section we will deploy a classic example of application level clustering in OpenShift using WildFly, a popular application server for Java based application runtimes. You will be deploying new application as part of this chapter so create a new project as follows

```
$ oc new-project stateful-apps

Now using project "stateful-apps" on server "https://api.crc.testing:6443".
You can add applications to this project with the 'new-app' command. For
   example, try:
    oc new-app rails-postgresql-example
to build a new example application in Ruby. Or use kubectl to deploy a simple
    Kubernetes application:
    kubectl create deployment hello-node --image=registry.k8s.io/e2e-test-
       images/agnhost:2.43 -- /agnhost serve-hostname
```

It is important to note that this new example uses cookies stored in your browser to track your session, Cookies aer small pieces of data that servers ask your browser to hold to make your experience better. In this case a cookie will be stored in your browser with a simple unique identifier, a randomly generated string called `JSESSIONID`. When the user initially accesses the web application the server will reply with a cookie containing that java session id field and a unique identifier as the value. Subsequent access to the application will use the `JSESSIONID` to look up all information about the user's session, which is stored in a replication cache. It doe not matter which pod is accessed - the user experience will be the same

The WildFly application that you will deploy will replicate the user data among all the pods in its service. The application will track which user the request comes from by checking the `JSESSIONID`, that is passed from the user's browser cookie. Because the user data will be replicated the end user will have consistent experience even if some pods die and new pods are accessed. Run the following `oc` command to create the new application.

```
# this will create the template that we will use to create the application
   and all required dependencies below
$ oc create -f openshift/stateful-app-template.yml -n stateful-apps

# to see the list of templates, we can now do the following, and use the name
    of our newly created template to create
# the new application from it, as done below
$ oc get templates

# use the template to directly create the components of our new application,
   like so, specify the name of the template
$ oc new-app --template="wildfly-oia-s2i"
```

Now that the application is deployed let us explore application clustering with WildFly on OpenShift. To demonstrate this we will do the following

- add data to your session by registering users on the application page
- make a note of the pod name
- scale the service to two replicated pods. The WildFly application will then automatically replicate your session data in memory between pods
- delete the original pod
- verify that your session data is still active

Querying the OpenShift server

Before we get started we need to modify some permissions for the default service account in your project which will be responsible for running the application pods. From the stateful app project run the following command to add the view role to the default service account. The view role will allow the pods running in the project to query the OpenShift API server directly. In this case the application will take advantage of the ability to query the OpenShift API server to find other WildFly application pods in the project. These other instances will send pod-to-pod traffic directly to each other and use their own application specific service discovery method and load balancing features for communication

```
# this will make sure to add to the service account the view role, the
   service accounts are named based on the project
# name and you can see that in our command below too, that would be system:
   serviceaccount:stateful-apps:default
$ oc policy \
    add-role-to-user \
    view \
    system:serviceaccount:$(oc project -q):default \
    -n $(oc project -q)

clusterrole.rbac.authorization.k8s.io/view added: "system:serviceaccount:
   stateful-apps:default"
```

Start by registering a few users in the WildFly application page. List the pods for the project and remember their ids name. Then scale up these pods with the

```
# we first would like to the pods that we have, we can see that we have one
   running pod, which is where the application
# lives, the rest are the build and deploy pods which are completed we do not
    care about them, we can see that the
# application lives in a pod with id - wildfly-app-1-vr2kz
$ oc get pods
NAME                   READY   STATUS              RESTARTS   AGE
wildfly-app-1-build    0/1     Completed           0          3h25m
wildfly-app-1-deploy   0/1     Completed           0          3h24m
wildfly-app-1-vr2kz    1/1     Running             0          3h24m

# in between here, create a few Wildfly, accounts by visiting the application
    URL, first we can list the routes created
# for our app, and then describe one of the routes, to find where this is
   exposed, after which action we can visit that
# link
$ oc get routes
NAME                   HOST/PORT
   PATH   SERVICES                PORT       TERMINATION     WILDCARD
wildfly-app            wildfly-app-stateful-apps.apps-crc.testing
                     wildfly-app            <all>                        None

# from the description of the route we can see the request host, which in
   this case point to
# http://wildfly-app-stateful-apps.apps-crc.testing, visit that URL and
   create a few accounts first, so we can see how the
# data we create will persist over when later on we scale the pods
$ oc describe route route/wildfly-app
Name:                   wildfly-app
Namespace:              stateful-apps
Labels:                 app=wildfly-oia-s2i
                        app.kubernetes.io/component=wildfly-oia-s2i
                        app.kubernetes.io/instance=wildfly-oia-s2i
                        application=wildfly-app
                        template=wildfly-oia-s2i
Requested Host:         http://wildfly-app-stateful-apps.apps-crc.testing

# we can now scale up the deployment/deployment-config, which will cause the
   old pod to get deleted, and two new ones
# will get created
$ oc scale dc/wildfly-app --replicas=2
```

```
# here we can see that the pods are now two, the old one is still there that
   would be the one with suffix vr2kz, but a
# new with with a suffix 47nwb has now spun up
$ oc get pods
NAME                   READY   STATUS              RESTARTS   AGE
wildfly-app-1-build    0/1     Completed           0          3h33m
wildfly-app-1-deploy   0/1     Completed           0          3h31m
wildfly-app-1-vr2kz    1/1     Running             0          3h31m
wildfly-app-1-47nwb    1/1     Running             0          31s

# now we can delete the original pod, the one ending with vr2kz, if your
   application works correctly, during the scaling
# procedure, the application would have cloned / replicated over the data
   from the vr2kz original pod, to the new one
# which ends with 47nwb, meaning that no data will be lost when the first pod
    get deleted
$ oc delete pod wildfly-app-1-vr2kz

# after we delete this pod, since our deployment config was modified to
   require two replicas a new one would be spun up
# in its place, to keep the pod replicas at two (2) which is okay, the
   desired state is fulfiled, there is one new one
# with suffix k7g28 in place of the one we deleted vr2kz
$ oc get pods
NAME                   READY   STATUS              RESTARTS   AGE
wildfly-app-1-build    0/1     Completed               0          3h37m
wildfly-app-1-deploy   0/1     Completed               0          3h36m
wildfly-app-1-k7g28    1/1     Running                 0          6s
wildfly-app-1-47nwb    1/1     Running                 0          4m41s
```

The two pods discovered each other with the help of a WildFly specific discovery mechanism designed for Kubernetes, the implementation is called KUBE_PING and its part of the `JGroups` project. When the second pod was started it queried the OpenShift API for all the pods in the current project. The API server then returned a list of pods in the current project. The KUBE_PING code in the WildFly server filtered the list of pods for those with special ports labeled ping. If any of the pods in the result set returned from the API server match the filter then the `JGroups` code in WildFly will attempt to join any existing clusters among the pods in the list.

Take a moment to examine the result set from the pod perspective by navigating to any of the pods in the OpenShift console and clicking the terminal tab, then run this command to query the API server for a list of pods in the project matching the label `application=wildfly-app`

```
# first we can simply login into or remote ssh into one of the pods and
   inspect the contents of the environment which
# shows what is the OpenShift config that WildFly would be using to query the
    server, in this case we run the following
# inside the ssh session of one of the pods - env | grep -i kube, and we can
   see that we have these env variables, which
# are used to determine which namespace to query, which is really the project
    name in OpenShift terms, and the labels to
# filter on the pods, in our case to make sure it is looking only for pods
   that are the actual application that we care
```

```
# about.
OPENSHIFT_KUBE_PING_NAMESPACE=stateful-apps
OPENSHIFT_KUBE_PING_LABELS=application=wildfly-app
```

Verify the data replication

Now that two pods a re successfully clustered together delete original pod from the OpenShift console or by the command line. The OpenShift replication controller will notice that a pod has been deleted and will spin up a new one in its place to ensure that there are still two replicas. If clustering gs working properly the original data you entered will still be available, even though it was originally stored in memory in a pod that no longer exists. Double check by refreshing the application in your browser. If the data is not longer there make sure that the policy command above was run correctly, it has to be run first before you do any other changes and before we start scaling and deleting pods

Other cases for direct pod access

A java application server that needs to cluster application is just one common use case for direct pod discovery and access. Another example is an application that has its own load balancing or routing mechanisms such as shared data base. A shared database is one in which large data sets are stored i many small databases as opposed to one large database. Many sharded databases have intelligence built into their clients and drivers that allow for direct access to the correct shard without querying where the data resides. Sharded databases work well with OpenShift and have been implemented using MongoDB and Infinispan

A typical sharded database implementation may include creating the service object as headless service. Once a headless service object is created DNS can be used as another service discovery mechanism or method. A DNS query for a given headless service will return A records for all the pods in the service. More information on the DNS and A records is available in next sections. Applications can then implement custom login to determine which pod to access. One popular application that uses DNS queries to determine which instances to access is Apache Kafka a fast open source messaging broker. Most implementations of Kafka on OpenShift and other kubernetes base platforms use headless services so the messaging brokers can access each other directly to send and replicate messages. The brokers find each other using DNS queries which are made possible by implementing a headless service.

Other common use cases for direct access include more mundane IT workloads such as software agents that are used for backups and monitoring. Backup agents are often run with many traditional database workloads and implement features such as scheduled snapshots and point in time recovery of data. A monitoring agent often provides features such as real time alerting and visualization of an application. Often these agents may either run locally embedded as instrumented code in the application or communicate through direct network access. For many cases direct network access is required because the agents may communicate with more than one application across many servers. In these scenarios, the agents require consistent direct access to applications in order to fulfill their daily functions.

Describing sticky sessions

In the WildFly example data is replicated between the WildFly server instances. A cookie with a unique identifier is generated automatically by the application and stored in your browser. By using a cookie the application can track which end user is accessing the application. This approach works well but has several drawbacks. The most obvious is that if the WildFly server did not support application clustering or did not have a discovery mechanism that works in OpenShift that application would produce uneven user experience. Without application clustering if there were two application pods one with user data and one without user data then the user would see their data like 50% of the time because requests are send round-robin manner between pods in a service

Once common solution to this problem is to use sticky session. In the context of OpenShift enabling sticky sessions ensures that a user making requests into the cluster will consistently receive responses from the same pod for the duration of their session.

This added consistently helps ensure a smooth user experience and allows many applications that store temporarily data locally in the container to be run in OpenShift. By default in OpenShift sticky session are implemented using cookies for HTTP based routes and some types of HTTPS based routes. The OpenShift router can reuse existing cookies or create new cookies. The WildFly application you created earlier created its own cookie so the router will use that cookie for the sticky session implementation. If cookies are disable or can not e used for the route sticky sessions are implemented using a load balancing scheme called source that uses the client's IP address as part of its implementation.

Toggling sticky sessions

Let us see how sticky session work by toggling cookies on and off using the Linux curl command line tool, which can make HTTP requests to a server eight times and print the result. The WildFly application you have deployed has a couple of REST endpoints that have not been explored yet. The first endpoint can be used to print out the pod IP and hostname. Enter the following command to print out that information from 8 sequential HTTP requests to the app's route

```
# here we call curl 8 times against the route of our application, using the /
   rest/serverdata/ip endpoint
for I in $(seq 1 8); \
do \
    curl -s \
    "$(oc get route wildfly-app -o=jsonpath='{.spec.host}')/rest/serverdata/
       ip"; \
    printf "\n"; \
done

# you will get something like that, as an output, we can see the host names
   which are really also corresponding to the
# pod names as well as the IP addresses of those pods,
{"hostname":"wildfly-app-1-tmkqj","ip":"10.217.0.164"}
{"hostname":"wildfly-app-1-tmkqj","ip":"10.217.0.164"}
{"hostname":"wildfly-app-1-tmkqj","ip":"10.217.0.164"}
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
{"hostname":"wildfly-app-1-tmkqj","ip":"10.217.0.164"}
```

The output prints the hostname and the IP address of each pod four times, alternating back and forth between pods in a round robin pattern. That is as you would expect from the default behavior, the curl command does not provide any cookie or identifier method to tell OpenShift to not do the default round robin routing approach.

Fortunately curl can save cookies locally in a text file that can be used for future HTTP requests, to the server. Use the following command to grab the cookie from the WildFly application and save it to a local file called `cookie.txt`

```
# first make sure to save the cookie by doing a regular request, then we will
    use that file to send it over for our 8
```

```
# requests we are going to be doing, just as above, and inspect what we get
   as a result
$ curl -o /dev/null \
    --cookie-jar cookies.txt \
    $(oc get route wildfly-app -o=jsonpath='{.spec.host}')/rest/serverdata/ip

# here is the content of that file after we created it with the request above
   , we can see that this is the data of a
# regular browser cookie, it contains the type of the cookie - http-only, and
    the host we are hitting as well, and some
# other fields which are of no interest at the moment, along with a couple of
    cookie identifiers at the end
$ cat cookie.txt
# Netscape HTTP Cookie File
#HttpOnly_wildfly-app-stateful-apps.apps-crc.testing FALSE / FALSE 0
   e5cd79b5768b9bd942ee273490b6e8ec 38affa384fdb33244245b19937c8d6e8

# here we first also save any cookies that the server provides us back and
   then send them back, to the server with each
# HTTP request, this will ensure that the requests are pinned to a pod
for I in $(seq 1 8); \
do \
    curl -s --cookie cookies.txt \
    "$(oc get route wildfly-app -o=jsonpath='{.spec.host}')/rest/serverdata/
       ip"; \
    printf "\n"; \
done

# and here is the output, now we can see that only one pod is being hit,
   there is no round robin pattern of access for the pods
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
{"hostname":"wildfly-app-1-bhtbm","ip":"10.217.0.165"}
```

Limitations of cookies

One limitation of using cookies for load balancing is that they do not work for HTTPS connections that use the pass-through routing. In pass through routing there is an encrypted connection from the client typically a browser all the way to the application pod. In this scenario cookies wont work, because the connection is encrypted and OpenShift has no way of decrypting it, there is no way for the routing layer in OpenShift to see the request. To solve this problem OpenShift uses the client IP address to implement sticky sessions. But this option has a couple of drawbacks.

First many client IP addresses get translated using Network Address Translation (NAT), before reaching their destination. When a request is translated using NAT it replaces the often private IP address of the client with that of a public IP address. This frequently makes the client IP address the same for all users on a particular home or business network. Imagine a scenario in which you ran three pods to run an application for everyone in your office, but everyone in your office was being routed to the same pod because the requests are translated by NAT and are seen or appeared to be show the same source IP address.

Second OpenShift uses an internal hashing schema based on the client IP address and the number of pods to determine is load balancing schema. When the number of replicas changes such as when you are using auto scaling it is possible to lose sticky sessions.

For the rest of this section you will not need to instances of the WildFly application. So let us scale it back down

```
$ oc scale dc/wildfly-app --replicas=2
deploymentconfig.apps.openshift.io "wildfly-app" scaled
```

Shutting down applications

So far in this section you have learned how to use sticky sessions to ensure that users have consistent experience in OpenShift. You have also learned how to use custom load balancing and service discovery in OpenShift services. To demonstrate custom load balancing you deployed an application that keeps users data in memory and replicates its data to other pods. When looking at clustering you entered data and then scaled up to two pods that replicated the data you entered. You then killed the original pod and verified that your data was still there. This approach worked well but in a controlled and limited capacity. Imagine a scenario in which autoscaling was enabled and the pods were spinning up and down more quickly. How would you know the application data had been replicated before the particular pod was killed

- or even which pod was killed. OpenShift has several ways to solve this issue

Application grace period

The easiest and most straightforward solution is to use a grace period for the pod to gracefully shut down. Normally when an OpenShift deletes a pod it sends the pod a Linux TERM signal, often abbreviated as SIGTERM. The SIGTERM acts as a notification to the process that it needs to finish what it is doing and then exit and pass control to the parent process. One caveat is that the application needs custom code to catch the signal and handle the shutdown sequence. Fortunately many application servers have this code built in. If the container does not exit within a given grace period, OpenShift sends a Linux KILL signal or SIGKILL, that immediately terminates the application.

In this section, we will deploy a new a to demonstrate how OpenShift grace period works. In the same stateful apps project that you are already in run the following command to build and deploy the application

```
# this will create the project directly from the github using the resource
   template
$ oc new-app \
    -l app=graceful \
    --context-dir=dockerfile-graceful-shutdown \
    https://github.com/OpenShiftInAction/chapter8
```

The application may take a minute to build because it may need to pull down the new base image to build the application. Once the application is successfully built and running delete it with a grace period of 10 seconds.

```
# this will prompt the container runtime to wait 10 seconds before forcefully
    killing the container process, this gives
# us a window of 10 seconds, within which we will take a look at the logs
   produces by the pod, see below
```

```
$ oc delete pod -l app=graceful --graceperiod=10
```

When you run delete with a grace period of 10 seconds, OpenShift sends a SIGTERM signal immediately to the pod and then forcibly kills it in 10 seconds if it has not exited by itself within that time period. Quickly run the following command to see this plays our in the logs for the pod

```
# this will pull the logs for the pods that are running our app, we are using
    a sub shell command to select the pod-id
# dynamically to make sure we do not lose time tracking it down, since we
   have 10 seconds to do this, to see the logs,
# there is a way to see the logs of a killed pod as well, but that we will
   leave for future excessive
$ oc logs -f $(oc get pods -l app=graceful -o=jsonpath='{.items[].metadat.nam
   }')

# we can see that these are the logs the pod is printing while it is running,
    since it is an example application purely
# created to test the SIGTERM it is waiting for us to send that signal,
   through the delete command, then after the signal
# was caught it is now running the code that is in the signal hook or
   callback section
Waiting for SIGTERM, sleeping for 5 seconds now...
Waiting for SIGTERM, sleeping for 5 seconds now...
Waiting for SIGTERM, sleeping for 5 seconds now...
...
Caught SIGTERM! Gracefully shutting down now
Gracefully shutting down for 0 seconds
Gracefully shutting down for 1 seconds
Gracefully shutting down for 2 seconds
Gracefully shutting down for 3 seconds
Gracefully shutting down for 4 seconds
Gracefully shutting down for 5 seconds
Gracefully shutting down for 6 seconds
Gracefully shutting down for 7 seconds
Gracefully shutting down for 8 seconds
Gracefully shutting down for 9 seconds
```

The process that is running is a simple bash script that waits for a SIGTERM signal and then prints a message to standard out until it is killed. In this case the pod was given a grace period of 10 seconds and the pods printed logs for approximately 10 seconds before it was forcibly killed. By default, the grace period is set to 30 seconds. If you have an important container, that you never want to kill or be killed, you must set the `terminationGracePeriodSeconds` field in the deployment to -1

As we know in a container the main process runs as process PID - 1. This is very important and when handling linux signals, because only PID 1, receives the signal, although most containers have a single process many containers have multiple processes. In this scenario the main process needs to catch the signal and notify the other process in the container - `systemd` can also be used as a seamless solution. For containers with multiple processes that all need to handle linux signals it is best to use `systemd`, for this implementation.

We can now proceed and delete this example app demo, bu doing `oc delete all -l app=graceful`, which will delete all resources, tagged with the label - graceful

Container lifecycle hooks

Although catching basic linux signal such as SIGTERM is a best practice many application are not equipped to handle the Linux signal.s A nice way to externalize the logic from the application is to use the `preStop` hook and one of two container lifecycle hooks available in OpenShift. Container lifecycle hooks allow users to take predetermined actions during a container management lifecycle event. The two events available in OpenShift are as follows

- `PreStop` - Executes a handler before the container is terminated, this event is blocking, meaning it must finish before the pod is terminated.
- `PostStart` - Executes a handler immediately after the container is started.

Similar to readiness probes and liveness probes the handler can be a command that execute in the container, or it can be an HTTP call to an endpoint exposed by the container. Container lifecycle hooks can be used in conjunction with pod grace periods. If the `preStop` hook are used they take precedence over the pod deletion. SIGTERM will not be sent to the container until the `preStop` hook finishes executing.

Container lifecycle hooks and linux signal handling are often used together but in many cases users decide which method to use for their application. The main benefit of using linux signal handling is that the application will always behave the same way, no matter where the image is run. It guarantees consistent and predictable shutdown behavior because the behavior is coded in the application itself. Sending SIGTERM signals on delete is fundamental not only to all kubernetes or OpenShift platforms but to all runtimes like docker, contaienrd and so on. If the user handles the SIGTERM signal in their application the image will behave consistently even if it is moved outside of OpenShift. Because the `preStop` hooks need to be explicitly added to the deployment or template there is no guarantee that the image will behave the same way in other environments. Many application such as third party application do not handle SIGTERM properly and the end user can not easily modify the code. In this case a `preStop` hook must be used. A good example of this is the NGINX server, a popular and lightweight HTTP server. When NGINX is sent a SIGTERM it exits immediately rather than forking NGINX image itself, and adding code to handle the linux SIGTERM signal an easy solution is to add a `preStop` hook that gracefully shuts down the NGINX server form the command line. A general rule to follow is that i you control the code, you should code your application to handle the SIGTERM if you do not control the code use a `preStop` hook if needed

Stateful sets

So far in this chapter you have learned that OpenShift has many capabilities to support stateful application. These let users make traditional workloads first class citizens on OpenShift, but some application also require even more predictable startup and shutdown sequencing as well as predictable storage and networking identifying information. Imaging a scenario with the WildFly application in which data replication is critical to the user experience but a massive scaling event destroys too many pods at one time while replication is happening, how will the application recover ? Where will the data be replicated to ? Is it going to be lost forever ?

To solve this problem OpenShift has a special object called a stateful set, known as pet set or older versions of OpenShift. A stateful set is a powerful tool in the OpenShift users toolbox to facilitate many traditional workloads in a modern environment. A stateful set object is used in place of a replication controller or replica set (in newer version of OpenShift platform) and it is the underlying implementation to ensure replicas in a service, but it does so in a more controlled way.

A replication controller can not control the order of how pods are created or destroyed. Normally if a user configures a deployment to go from one to five replicas in OpenShift that task is passed to an replica set or controller that starts four new pods all at once. The order in which they are started, and marked as ready, is completely random and unknown

Deterministic sequence

A stateful set brings a deterministic sequential order to pod creation and deletion. Each pod that is created also has an ordinal index number associated with it. The ordinal index indicates the startup order. For instance if the application WildFly application was using a stateful set with three replicas the pods would bee started and names in this order - `wildfly-app-0`, `wildfly-app-1`, `wildfly-app-2`. A stateful set also ensures that each pod is running and ready, (has passed the readiness probe) before the next pod is started, it is sequential. In the previous scenario `wildfly-app-2` would not be started until the `wildfly-app-1` was running and ready

The reverse is also true, a replication set or controller will delete pods at random, when a command is given to reduce the number of replicas. A stateful set can also be used to controller shutdown sequence, it starts with the pod that has the highest ordinal index (n-1 replica) and works its way backward to meet the new replica requirements. A pod will not be shut down until the previous pod has been fully terminate. Refer back to the previous section regarding SIGTERM and lifecycle hooks which will also affect this behavior

The controller shutdown sequence can be critical for many stateful application. In the case of the WildFly application user data is being shared between a number of pod. When the WildFly application is shut down gracefully a data synchronizes process may occur between the remaining pods in the application cluster. This process often be interrupted without the use of a stateful set because the pods are shut down in parallel. By using a predictable one-at-a-time shutdown sequence, the application is less likely to lose any data which results in a better user experience.

Examining a stateful set

To see how stateful sets work first create a new project

```
# first create a new namespace project that we will be using to do our
   testing in
$ oc new-project statefulset

# then we can create the template from the provided file, which will be later
    used to create the actual app
$ oc create \
    -f openshift/stateful-set-template.yml \
    -n statefulset
template.template.openshift.io/mongodb-statefulset-replication-emptydir
   created

# now create the application from the template we have created above
$ oc new-app --template="mongodb-statefulset-replication-emptydir"

# note how our two pods are correctly labeled with -0 and -1, these are the
   ordinal values of the pods, they were
# created in that exact same order by the stateful set/controller
$ oc get pods
NAME      READY STATUS  RESTARTS AGE
mongodb-0 1/1   Running 0        68s
mongodb-1 1/1   Running 0        37s

# now we can see the logs of the very first pod with index 0, and we can
   actually see how the pod is configured to look for other pods such as in
   this case pod with index 1, to replicate the data, that was added to the
   database, in this case a single entry of user was added, then that data
   was replicated to the other pod, you can clearly see how the mongodb-0 pod
    is trying to access the other pod through the deterministic IP/host
   address which in this case is the mongodb-1.mongodb-internal.statefulset.
   svc.cluster.local
$ oc logs mongodb-0
Successfully added user: { "user" : "oiauser", "roles" : [ "readWrite" ] }
bye
<event-date> I REPL     [conn12] replSetReconfig admin command received from
   client
<event-date> I REPL     [conn12] replSetReconfig config object with 2 members
    parses ok
<event-date> I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
   mongodb-1.mongodb-internal.statefulset.svc.cluster.local:27017
<event-date> I ASIO     [NetworkInterfaceASIO-Replication-0] Successfully
   connected to mongodb-1.mongodb-internal.statefulset.svc.cluster.local
   :27017
<event-date> I REPL     [ReplicationExecutor] New replica set config in use:
   { _id: "rs0", version: 2, protocolVersion: 1, members: [ { _id: 0, host: "
   mongodb-0.mongodb-internal.statefulset.svc.cluster.local:27017", arbi
terOnly: false, buildIndexes: true, hidden: false, priority: 1.0, tags: {},
   slaveDelay: 0, votes: 1 }, { _id: 1, host: "mongodb-1.mongodb-internal.
   statefulset.svc.cluster.local:27017", arbiterOnly: false, buildIndexes:
   true, hidden: fal
se, priority: 1.0, tags: {}, slaveDelay: 0, votes: 1 } ], settings: {
   chainingAllowed: true, heartbeatIntervalMillis: 2000, heartbeatTimeoutSecs
   : 10, electionTimeoutMillis: 10000, getLastErrorModes: {},
   getLastErrorDefaults: { w: 1, wti
meout: 0 }, replicaSetId: ObjectId('6846cec0fa37b01dd820333d') } }
<event-date> I REPL     [ReplicationExecutor] This node is mongodb-0.mongodb-
   internal.statefulset.svc.cluster.local:27017 in the config
<event-date> I REPL     [ReplicationExecutor] Member mongodb-1.mongodb-
   internal.statefulset.svc.cluster.local:27017 is now in state STARTUP
<event-date> I REPL     [ReplicationExecutor] Member mongodb-1.mongodb-
   internal.statefulset.svc.cluster.local:27017 is now in state SECONDARY

# start scaling to 3 replicas, which will simply add one more, to the list
   above, notice that our pods are in the
# ready state, which means that we will be able to scale them up, had the
   mongodb-1 pod been in a non ready 0/1 state,
# our scale command would have failed, that is because the stateful set will
   not allow us to scale up when we have
# pods that are not ready
$ oc scale statefulset/mongodb --replicas=3 && oc get pods
NAME      READY STATUS  RESTARTS AGE
mongodb-0 1/1   Running 0        4m16s
mongodb-1 1/1   Running 0        4m15s
mongodb-2 1/1   Running 0        4s

# now we can also look up the logs of the mongodb-0, we can see that now the
   newest pod which was the mongodb-2, is detected by the mongodb-0 pod,
   which then replicates its data to the mongodb-2 pod, using the same
   process as it did for the pod with ordinal index 1 above.
$ oc logs mongodb-0
<event-date> I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
   mongodb-2.mongodb-internal.statefulset.svc.cluster.local:27017
<event-date> I REPL     [ReplicationExecutor] New replica set config in use:
   { _id: "rs0", version: 3, protocolVersion: 1, members: [ { _id: 0, host: "
   mongodb-0.mongodb-internal.statefulset.svc.cluster.local:27017", arbi
terOnly: false, buildIndexes: true, hidden: false, priority: 1.0, tags: {},
   slaveDelay: 0, votes: 1 }, { _id: 1, host: "mongodb-1.mongodb-internal.
   statefulset.svc.cluster.local:27017", arbiterOnly: false, buildIndexes:
   true, hidden: fal
se, priority: 1.0, tags: {}, slaveDelay: 0, votes: 1 }, { _id: 2, host: "
   mongodb-2.mongodb-internal.statefulset.svc.cluster.local:27017",
   arbiterOnly: false, buildIndexes: true, hidden: false, priority: 1.0, tags
   : {}, slaveDelay: 0, vot
es: 1 } ], settings: { chainingAllowed: true, heartbeatIntervalMillis: 2000,
   heartbeatTimeoutSecs: 10, electionTimeoutMillis: 10000, getLastErrorModes:
    {}, getLastErrorDefaults: { w: 1, wtimeout: 0 }, replicaSetId: ObjectId('
   6846cec0fa3
7b01dd820333d') } }
<event-date> I REPL     [ReplicationExecutor] This node is mongodb-0.mongodb-
   internal.statefulset.svc.cluster.local:27017 in the config
<event-date> I ASIO     [NetworkInterfaceASIO-Replication-0] Connecting to
   mongodb-2.mongodb-internal.statefulset.svc.cluster.local:27017
<event-date> I ASIO     [NetworkInterfaceASIO-Replication-0] Successfully
   connected to mongodb-2.mongodb-internal.statefulset.svc.cluster.local
   :27017
<event-date> I ASIO     [NetworkInterfaceASIO-Replication-0] Successfully
   connected to mongodb-2.mongodb-internal.statefulset.svc.cluster.local
   :27017
<event-date> I REPL     [ReplicationExecutor] Member mongodb-2.mongodb-
   internal.statefulset.svc.cluster.local:27017 is now in state STARTUP
<event-date> I REPL     [ReplicationExecutor] Member mongodb-2.mongodb-
   internal.statefulset.svc.cluster.local:27017 is now in state SECONDARY
```

Unlike previous use of the scale command this time you need to explicitly state that you are scaling a stateful set. In OpenShift console notice that the new pod that was created has a deterministic pod name with the ordinal index associated with it. Also the reason we are using the `statefulset` object to scale, instead of deployment, is because the `statefulset` is a replacement for the regular deployment object when working with stateful application contexts, such as this one. The `StatefulSet` has similar but also different properties in its specification than the `Delpoyment` this is to reflect the features that the `StatefulSet` exposes to the OpenShift users, related to managing application state

Similar to the WildFly application the three MongoDB pods are replicating data to each other. To check that this replication is fully functional click any of the pods on the bottom of the mongodb stateful set Details page and then click the Terminal tab. Any commands executed here will execute in the pod. First log into the mongodb as the admin user, using `mongo admin -u admin -p oiaadminpassword`. After that check the status of the MongoDB replica set by typing `rs.status()` after the login was successful

Constant Network identity

Stateful sets also provide a consistent means of determining the host-naming scheme for each pod in the set. Each predictable hostname is also associated with a predictable DNS entry. Examine the pod hostname for mongodb-0 by executing this command -

```
# we are again pulling the id of each pod or its name to exec a cat command
   from the context of each node, printing the
# contents of the /etc/hostname file, which will show us the actual hostname
   of the pod
$ for statefulpod in $(oc get pods -l name=mongodb -o=jsonpath='{.items[*].
   metadata.name}'); \
    do \
        oc exec $statefulpod -- cat /etc/hostname; \
    done

# these are the hostnames, these will correspond to the names of the pods
   themselves as we know, from the very firs
# sections when we discussed the relationship between the container,
   container runtime, hostname and the orchestrator
mongodb-0
mongodb-1
mongodb-2
```

We know that the hostname is the same as the pod name, that is something we have seen and investigated earlier in previous sections. The stateful set also ensures a DNS entry for each pod running in a stateful set. This can be found by executing the dig command using the DNS entry name for each pod. Find the IP addresses by executing the following command from one of the OpenShift nodes. Because the command relies on the OpenShift provided DNS it must be run from within the OpenShift environment to work properly

`When you are using stateful sets the pod hostname in the DNS is listed in the format <pod name>.<service name>.<namespace>.svc.cluster.local`

Because this example also contains a headless service, there are DNS A records for the pods associated with the headless service. Ensure that the pod IP into DNS match the previous listing by running this command from the OpenShift nodes

```
# login into the node itself first, through ssh
$ ssh <cluster-node-host>

# from within the cluster node, we can then execute the following piece of
   shell script which will use dig - DNS lookup
# utility. Is a flexible tool for interrogating DNS name servers. It performs
    DNS lookups and displays the answers that
# are returned from the name server(s) that were queried. Most DNS
   administrators use dig to troubleshoot DNS problems
# because of its flexibility, ease of use, and clarity of output. Other
   lookup tools tend to have less
# functionality than d
$ for statefulpod in $(oc get pods -l name=mongodb -o=jsonpath='{.items[*].
   metadata.name}'); \
    do \
        dig +short $statefulpod.mongodb-internal.statefulset.svc.cluster.
           local; \
    done
```

Consistent persistent storage

Pods running as part of a stateful set can also have their own persistent volume claims associated with each pod. But unlike a normal persistent volume claim, they remain associated with a pod and its ordinal index, as long as the stateful set exists. In the previous example you deployed an ephemeral stateful set without persistent storage. Imagine that the previous example was using persistent storage, and the pods were writing log files that included the pod hostname. You wold not want the persistent volume claim to later be mapped to the volume of a different pod with a different hostname because it would be hard to make sense of those log files, for debugging and auditing purposes. Stateful sets solve this problem by providing a consistent mapping through the use of a volume claim template which is a template the persistent volume associates with each pod. If a pod dies or is reschedules to a different node, then the persistent volume claim will be mapped only to the new pod that starts in its place with the same hostname as the old pod. Providing a separate and dedicate persistent volume claim for each pod in the stateful set is crucial for many different types of stateful applications which cannot use the typical deployment config model of sharing the same persistent volume claims across many applications instances.

The Stateful set limitations

Under normal circumstances pods controller by a stateful set should not need to be deleted manually. But there are a few scenarios in which a pod being controller by a stateful set could be deleted by an outside force. For instance if the kubelet or node is unresponsive, then the API server may remove the pod after a given amount of time and restart it somewhere else in the cluster. A pod could also exit accidentally or could be manually removed by a user. In those cases it is likely that the ordinal index will be broken, New pods will be created with the same hostname and DNS entries, as the old pods but the IP addresses may be different. For this reason any application that relies on hard coded IP addresses is not a good idea, or fit for stateful sets. If the application can not be modified to use DNS or hostnames instead of IP addresses you should use a single service per pod for a stable IP address.

Another limitation is that all the pods in a stateful set are replicas of each other which of course makes sense when you want to scale, But that would not help any situation in which disparate applications need to be started in a particular order. A classic example is a Java or NET application that throws errors if a database is unavailable, once the database is started then the application also needs to be restarted to refresh the connections. In that scenario, a stateful set would not help the order between the two disparate services

Non-native stateful applications

One of the reasons OpenShift has gained so much market adoption is that traditional IT workloads work just as well as modern stateless applications. Yet there is still work to be done, one of the biggest promises of using containers is that applications will behave the same way between environments. Containers start form well known image binaries that contain the application and the configuration it needs to run. If a container dies a new one is started form the previous image binary that is identical to how the previous container was started. Once major problem with this model occurs for applications that are changed on the fly and store their information in a way that makes it difficult to re-create

A good example of this issue can be seen with `wordpress` an extremely popular blogging application that was designed many years before containers became popular. In a given `wordpress` workflow a blogger might go to the admin portion of their web-site add some text and then save it, `wordpress` saves all that text in a database, along with any HTML and styling. When the blogger has completed this action the container has drifted form its original image. Container drift is normal for most applications but in this case, if the container crashed the blog would be lost, persistent storage can be used to ensure that the data is persisted. When a new `wordpress` pod starts it could map to the database and would have all the blogs available.

But promoting such a snapshot of a database among various environments is a major challenge. There are many examples of using an event driven workflow that can be triggered to export and import a database after a blogger publishes content, but it is not easy nor is it native to the platform. Containers start from well known immutable container images, but engineering a reverse workflow in which images are created form running container instances is more error-prone and rigid. Other examples that have worked with some engineering include - applications that open a large number of ports applications that rely on hard coded IP addresses, and other legacy applications that rely on older Linux technologies

Cleanup resource limits & quotas

After you are done with this chapter, make sure to either delete the resource quota and limit ranges objects we have created or edit them to be more lenient, in case you wish to up-scale your replicas for the app-cli project, this will vastly restrict your capabilities to do so since the limits are not enough for more than 2 replicas for this project. Be wary of this !

Operations & Security

This section focuses on cluster wide concepts and knowledge you will need to effectively manage an OpenShift cluster at scale. These are some of the skills required for any operations teams managing an OpenShift cluster. In previous chapter is all about working with OpenShift integrated role based access control, you will change the authentication provider for your cluster, users and work with the system accounts build into OpenShift along with the default roles for different user types. Other sections focus on the software define network that deployed as part of OpenShift this is how containers communicate with each other and how service discovery works in an OpenShift cluster. The we will bring everything together and look at OpenShift from security perspective. We will discuss how SELinux is used in OpenShift and how you can work with security policies to provide the most effective level of access for your application.

Permissions vs Wild-West

A platforms like OpenShift are not effective for multiple users without robust access and permissions management for various applications in OpenShift components. If every user had a full access to all your OpenShift resources it would truly be the wild west. Conversely if it was difficult to access resources, OpenShift would not be a good for much either. OpenShift has a robust authentication and access control system, that provides a good balance of self-service workflows to keep productivity up while limiting users to only what they need to get their job done. When you first deployed OpenShift the default configuration allowed any user name and non empty password field to log in. This authentication method uses the allow-all identity provider that comes with OpenShift.

In OpenShift the identity provider is a plugin that defines how users can authenticate and the backend service that you want to connect for managing user information. Although the allow all provider is good enough when you are learning to use OpenShift when you need to enforce access rules you will need to change to a more secure authentication method. In the next section you will replace the allow-all provider with one that uses a local data base file

We are going to use and configure the OpenShift cluster with an Apache htpasswd database for user access and set up a few users to use with that authentication source You will create the following users:

- developer a user with permissions typical to those given to a developer in an OpenShift cluster
- project-admin - a user with permissions typical of a developer or team lead in an OpenShift cluster

- admin - a user with administrative control over the entire OpenShift cluster

Please go through that now, and then continue with this section. After configuring OpenShift in that way, if you attempt to log in with your original dev or other user that user can not be authenticated because it is not in your htpasswd database, but if you log into using the new developer or any of the new users, you will no longer have access to the previous projects we have created - like the `image-uploader` and so forth, that is due to the fact that the old dev user would still own that namespace or project

Setting up authentication

Many different user databases are available to the IO professionals for managing access and authentication. To interoperate with as many of these as possible, OpenShift provides 11 identity providers that interface with various user databases, including the allow all provider that you have been using so far in the cluster. These providers are as follows

- allow all - allows any username and non-empty password to log in
- deny all - does not allow any usernames and passwords to log in
- htpasswd - authenticates with Apache htpasswd database files
- keystone - user OpenStack Keystone as the authentication source
- LDAP - authenticates using an LDAP provider like OpenLDAP
- Basic - uses the Apache Basic authentication on a remote server to authenticate users
- Request Header - uses custom http header for user authentication
- GitHub - authenticates with github, using OAuth2
- Gitlab - authenticates with gitlab, using OAuth2
- Google - uses google OpenID connect for authentication

Different authentication providers have different options that are specific to each provider unique format, for example the options available for the htpasswd provider are different than those required for the github provider, because these providers access such different user databases

What is htpasswd - that is an utility from the old days, it goes all the way back to the first versions of the Apache web server in the later 90s, back then computers had so much less memory that the name of an application could affect system performance, applications names were typically limited to eight characters, to fit this tight requirement characters were often removed or abbreviated - and thus htpasswd was born.

Introduction to htpasswd

The htpasswd provider uses the Apache style htpasswd files for authentication. These are simple databases that contain a list of usernames and their corresponding password in an encrypted format. Each line in the file represents a user. The user and password sections are separated with a colon (:). The password section includes the algorithms that was used to encrypt the password encapsulated with $ characters, and the encrypted password itself. Here is an example htpasswd file wit two users, admin and developer

```
admin:$apr1$vUqfPZ/D$sTL5RCy1m5kS73bC8GA3F1
developer:$apr1$oKuOUw1t$CEJSFcVXDH5Jcq7VDF5pU/
```

You create htpasswd files using the htpasswd command line tool, by default the htpasswd tool uses a custom encryption algorithms base on the `md5` hashing.

Creating htpasswd files

To create an htpasswd database file, you need to ssh into your master server or cluster node, on the master server the configuration files for the OpenShift master process are in the /etc/origin/master. There you will create the htpasswd file called `openshift.htpasswd` with three users - developer, project-admin, and admin - to act as the database for the htpasswd provider to interact with

You need to run the htpasswd command to add each user, the first time you run the command be sure to include the -c option to create the new htpasswd file. First make sure that the htpasswd utility is installed, you can do these operations on your host machine, since it is the file we are interested in, the file that will be produced by the commands below

```
# execute the following command to add the different users, and also create
   the htpasswd database file, you see that we
# still keep the kubeadmin here, just change the password of that user, this
   is because that user is the only
# cluster-admin, until we have another one we can not remove it from the
   htpasswd file, otherwise we will not be able to
# execute any cluster level operations which we need to still, to add roles
   to the other users and the new admin, which is
# going to be named just - admin, after that we can safely delete the
   kubeadmin user.
$ mkdir -p /etc/origin/master && touch openshift.htpasswd
$ htpasswd -b -c -B /etc/origin/master/openshift.htpasswd kubeadmin admin
$ htpasswd -b -B /etc/origin/master/openshift.htpasswd admin admin
$ htpasswd -b -B /etc/origin/master/openshift.htpasswd developer developer
$ htpasswd -b -B /etc/origin/master/openshift.htpasswd project-admin project-
   admin
```

Changing the provider

Before we add the new database with username and password, we can actually take a look at how the `crc` does it by default, it is using the htpasswd file database, too and we can actually see how with this command

```
# by default we can see that there is an object called htpass-secret, that is
    the default one which actually contains
# the two users, and their username and password, with the command below we
   can extract the contents of the secret to the
# standard output the secret is an opaque object that contains the htpasswd
   file encoded in base64, meaning it is by no
# means encrypted and we can see the content of the file
$ oc extract secret/htpass-secret --to=- --confirm -n openshift-config

developer:$2a$10$QVaaReC08iHwTJ5fUv2Kj.YdVbjtuNKgf89X/0uwsmQN17cUyR.vW
kubeadmin:$2a$10$GskveI9lKitkLXj.yoB46ueCoHp50ytLofjhQN5/LSy7bFs.UW8ta
```

First we need to define a new secret, we can use the contents of the htpasswd file after which we can apply that to the cluster like so, we will use a new secret name to avoid having to delete the old one, but the process will follow the same configuration steps

```
# note that we are creating a new secret in the openshift-config namespace,
   this is where all the openshift config
# lives, we will use the database file we created above, the secret command
   will base64 encode the file and create the
# secret from it, we use the new file we generated with the new users and
   passwords
$ oc create secret generic htpass-secret-custom --from-file=htpasswd=
   openshift/htpasswd-new-users -n openshift-config

secret/htpass-secret-custom created
```

Here is the document we have to actually update, by default there is already a cluster level OAuth provider, if we run an –apply with this new provider content, what will happen is that it will get updated and the name of the `fileData` field will be replace with the new secret that we created, which is the `htpass-secret-custom`

```
apiVersion: config.openshift.io/v1
kind: OAuth
metadata:
    name: cluster
spec:
    identityProviders:
        - name: htpasswd_custom_provider
          mappingMethod: claim
          type: HTPasswd
          htpasswd:
              fileData:
                  name: htpass-secret-custom
```

```
# run the following command with the content of the OAuth object above, to
   update the cluster config, then logout of
# existing sessions and try to login again, with some of the users, we have
   set the passwords to be the same as the
# usernames, by default all users will be developers, and will have access to
    no projects, we will fix this in next
# section where we give them permissions from the cluster node itself
$ oc apply -f misc-tech-topic/openshift/oauth-cluster-htpass.yml

oauth.config.openshift.io/cluster configured
```

Make sure to log-out and log back in after the changes, use the new password for the `kubeadmin` user which is simply `admin`

Working with roles

Roles are used to define permission for all users in an OpenShift cluster, in previous sections, you used the special admin to configure physical volumes on your cluster, the admin is a special user account. To work with roles you will use a new command line tool name `oadm` (short for OpenShift administration). It is installed by default on your cluster

On your master node, the OpenShift deployment program setup up the root user, we can see the user information set up for the root user by running the following command as the root user on your master node OpenShift server - `oadm config view`. This allows administrators with access to the root user on an OpenShift master node server to have cluster administration access by default. It is useful, but it also means you have to make sure everyone who has root access ot your master server should be able to control your OpenShift cluster. For a smaller cluster like the one you have built this will work fine. But for a large cluster, the people who should have root access to your server and the people who should be able to administer OpenShift probably wont match. You can distribute this administrative certificate as needed for your cluster administrator workstations.

Assigning user roles

Remember those users you created ? The developer user need permissions to view and add new content to the `image-uploader` project. To accomplish that first make sure you are working in the context of the `image-uploader` project by running the following command `oc project image-uploader` In the project namespace you need to add the edit role to your developer user. This role gives user permission to add. Adding a role to a new user for a project or even the entire OpenShift cluster is called binding a role to a user.

```
# make sure that your are first logged in the cluster node, do not execute
   this form the host machine, rather it has to
# be done directly from the node / master cluster server
$ oadm policy add-role-to-user edit developer
```

To confirm that your new role is applied log in again through the web UI or the command line as the developer user. You should now have access to the `image-uploader` project, and the deployed application there. That takes care of the developer user. Let us give your admin user a little more access in the next section you will give the admin user administrator level access to your entire OpenShift cluster

Creating administrators user

So far the OpenShift cluster has a single project, as an OpenShift cluster grows it typically has dozens or even hundreds of projects at any given time. To manage this effectively you need users who can administer a project or even hundreds of projects at any given time.

Creating the project admin - for the `image-uploader` project you will make the project admin user an administrator for the project only. You can do so much the same way you have the developer user the ability to edit, instead of binding the edit role to the project admin user, however you need to bind the admin role. This role will give the project admin user full administrative privileges in the `image-uploder` namespace project. Run the following command as root on your master server

```
# this gives the
$ oadm policy add-role-to-user admin project-admin
```

You now have developer user who can work in the `image-uploader` project and a project-admin who can administer the project. The next user role you need is one who can manage the entire OpenShift cluster.

Creating the cluster admin - the cluster admin role is important. A cluster admin can not only administer projects but also manage all OpenShift internal configuration and state, To create a cluster admin run the following command as root on your master node:

```
# note the difference here, we are giving a cluster-role, not just a role to
   the user, and the cluster role name is -
# cluster-admin, that should be enough for you to tell the difference between
    this command and the one we did for the other
# two users above.
$ oadm policy add-cluster-role-to-user cluster-admin admin
```

This command binds the admin role to the admin user we already created in our htpasswd database file, instead of binding that role for a single project, it binds it for every project or namespace in OpenShift. Everything you have donein this section until now will hep you edit existing users and make sure they have the correct privileges to access what their job requires. But what happens when you add new users ? In the next section we will configure OpenShift bind the edit role to new users by default when they are created

Setting default roles

OpenShift has three default groups. These groups are configured during OpenShift installation and define whether a user is authenticated. You can use these groups to target users for additional actions, but the groups themselves can not be modified.

- system:authenticated - any user who has successfully authenticated through the web UI or command line, or via the API
- system:authenticated:oauth - any user who has been authenticated by OpenShift internal OAuth2 server. This excludes system accounts
- system:unauthenticated - users who have failed authentication or not attempted to do one.

In your cluster it will be helpful to allow any authenticated user to access the `image-uploader` project. You can accomplish this by running the following `oadm` policy command which binds the edit role for the `image-uploader` project specified by the -n option to the system:authenticated group

```
# now any user who has successfully authenticated and logged in will now have
    direct access to this project.
$ oadm policy add-role-to-group edit -n image-uploader system:authenticated
```

When do we use other default groups - this example uses the system:authenticated group, depending on what you need to accomplish the other groups can be used in a similar fashion. The `system:authenticated:oauth` groups excludes the system accounts that are used to build and deploy applications in OpenShift. We will cover those in future sections, in short this group consists of all the humans and external services accessing OpenShift. System:unauthenticated can be used if you want to provide a level of anonymous access in your cluster. Its most common use however is to route any user currently in that group to the OpenShift login page

To confirm that your new default user role has taken effect add an additional user named `user1` to your htpasswd database file with the following command - `echo user1 | htpasswd --stdin /etc/origin/master /openshift/htpasswd user1`

Log into your cluster with that user and confirm that your new user can use the `image-uploader` project by default, that user should have the ability to work in the `image-uploader` project from the first login

Any time you have a shared environment you need processes in place to ensure that one user or project does not take up too many resources, or privileges in your cluster- either accidentally or on purpose. Limit ranges and resource quotas are the processes that manage this potential problem. In OpenShift these resources constraints are different for each deployment depending on whether explicit resources quotas are requested.

For previous applications that we deployed we did not specify any resources, either processor or memory to be allocated for either deployment these best-effort deployments do not request such specific resources and are assigned a best-effort quality of service can govern default values at the project level in OpenShift by using limit ranges. In the next section, we will discuss limit ranges in more depth and you will create your own and apply them to the `image-uploader` project.

Limit ranges

For each project in OpenShift a limit range defined as a `LimitRange` when working with the OpenShift API provides resources constraints, for most objects that exist in a project. The objects are the types of OpenShift components that users deploy to serve applications and data. Limit ranges apply to the maximum processing and memory resources and total object count for each components. The limits for each component are outlined here

Project component	Limits
Pod	CPU and memory per pod, and total pods per project
Container	CPU and memory per container, default memory and CPU, maximum burstable ratio per container, and total containers per project

Project component	Limits
`ImagesMaximum`	image size for the internal registry
Image	`streamMaximum` image tag references and image references per image stream
Persistent	volume `claimsMinimum` and maximum storage request size per PVC

Before an application is deployed or scaled up the project limit ranges are analyzed to confirm that the request is within the limit range. If a project limit range does not allow the desired action then it does not happen.

For example if a project limit range defines the memory per pod as being between 50 MB and 1,000 MB a request for a new application deployment with a defined quota of 1,500 MB will fail because it is outside the pod memory limit range for that project. Limit ranges have the additional benefit of being able to define default compute resource values for a project. When you deployed `app-gui` and `app-cli` you had not yet defined a limit range for the `image-uploader` project and did not specify the resources for each deployment so each application pod was deployed with no resource constraints. In OpenShift a deployment with no defined resources quota.

If users start accessing the gui heavily it can consume resources to the point that the performance of the app-cli deployment is affected for a busy cluster with multiple users and running application that is a major problem, with limit ranges you can define the default compute resources for a project that does not specify a quota to prevent this from happening in your cluster.

Define resource limit ranges

Limit ranges define the minimum and maximum RAM and CPU an application can be allocated when its deployed. In addition to the top and bottom of the range you can specify default request value and limits. The difference between an application requested value, and its maximum value limit is called the burstable range

Let us create a limit range for the `image-uploader` project using the template, create the file containing that content and apply it to the

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
    name: "core-resource-limits"
spec:
    limits:
        - type: "Pod"
          max:
              cpu: "2"
              memory: "1Gi"
          min:
              cpu: "100m"
              memory: "4Mi"
        - type: "Container"
          max:
              cpu: "2"
              memory: "1Gi"
          min:
              cpu: "100m"
```

```
                memory: "4Mi"
            default:
                cpu: "300m"
                memory: "200Mi"
            defaultRequest:
                cpu: "200m"
                memory: "100Mi"
            maxLimitRequestRatio:
                cpu: "10"
```

To define a limit range a user need to have the cluster admin role. To log in as the admin user run the following command, since we have already setup a user like that, we can use the following command to do so

```
# first make sure that we are logged in with a user that has a cluster-admin
   role
$ oc login -u kubeadmin -p <kubeadmin-password> <cluster-host-address>

# set the resource limits for the project or namespace alone
$ oc apply -f openshift/limit-ranges.yml -n image-uploader

# we can then list them and optionally describe the object for more details
$ oc get limitranges -n image-uploader

NAME                   CREATED AT
core-resource-limits   <date>
```

After which we can simply run the apply to create the new resource limit range for the project or namespace, in this case we are using the `image-uploader` project, other project can be used instead just change the value of the -n argument in the command.

Throughout this section and the rest of the document we will need to create YAML template files that are referenced in the `oc` commands .We use relative file name paths to keep the examples easy to read, but if you are not running `oc` from the directory where those files are created, be sure to reference the full path when you run the command

You can use the command line to confirm that the `image-uploader` project limit range was created and to confirm that the settings you specified in the template were accurately read. As with every other resource in the OpenShift cluster we can do this by using the get command - `oc get limitrange`. Those can be combined with a describe command to list the details for each of these resource limits

You can also use the web interface to confirm that the limit range you just set. Using the Resources > Quotas. Limit ranges act on a components in a project. They also provide default resource limits for deployments that do not provide any specific values themselves But they do not provide project wide limits to specify maximum resource mounts. For that you will need to define a resource quota for `image-uploader` project. In the next section that is exactly what will be done

Resource quotas

Nobody likes noise neighbor, and OpenShift users are no different. If one project users were able to consume more than their fair share of the resources in an OpenShift cluster all manner of resource availability issues wold occur, for example a resource hungry development project could stop applications in a production level project in the same cluster from scaling up when their traffic is increased. To solve this problem OpenShift uses project quotas to provide resource caps and limits at the project level. Whereas limit ranges provide maximum resource limits for an entire project, quotas on the other hand fall into three primary categories:

- compute & storage resources - memory, processing (CPU) etc
- object counts - services, storage claims, config maps, secrets, replication controllers etc

In one of the very first sections, we discussed the pod life-cycles, project quotas apply only to pods that are not in a terminal phase. Quotas apply to any pod in a pending running or unknown state. Before an application deployment is started or a deployed application is changed OpenShift evaluates the project quotas.

In the next section you will create a compute resource quota for the `image-uploader` project

Creating compute quotas

Compute resource quotas apply to CPU and memory allocation, They are related to limit ranges because they represent quotas against totals for requests and limits for all application in a project. You can set the following six values with compute resource quotas.

- cpu, request.cpu - total of all CPU requests in a project typically measured in cores or millicores CPU and requests.cpu are synonyms and can be used interchangeable.
- memory, request.memory - total of all memory requests in a project typically expressed in mega or gigabytes or memory and requests.memory are synonyms and can be used interchangeable
- limits.cpu - total for all CPU limits in a project
- limits.memory - total for all memory limits in a project

In addition to the quotas you can also specify the scope the quota applies to. There are four quotas scopes in OpenShift.

- `Terminating` - Pods that have a defined life cycle. Typically these are builder and deployment pods.

- `NotTerminating` - Pods that do not have a defined life cycles. This scopes include application pods like the app-gui and app-cli and most other applications you will deploy in OpenShift.
- `BestEffort` - Pods that have a best-effort quality of service, for processing and memory. Best-effort deployment are those that did not specify a request or limit when they were created.
- `NotBestEffort` - Pods that do not have a best effort quality of service, for processing and memory, that is the inverse of `BestEffort` this scope is useful when you have a mixture of low priority transient workloads that have been deployed with best effort quality of service and higher priority workloads with dedicated resources

To create an new quota for a project cluster admin privileges are required. That means you need to be logged in as the admin user to run this command, because the developer user has only the edit role bound to it, for the `image-uploader` project and has no privileges for the rest of the cluster. To log in as the admin user follow the previous section, where we already did that for the range limits

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
    name: compute-resources
spec:
    hard:
        pods: "10"
        requests.cpu: "2"
        requests.memory: 2Gi
        limits.cpu: "3"
        limits.memory: 3Gi
    scopes:
        - NotTerminating
```

Save the template to a file, and execute the commands below to apply the resource quota object, this will ensure that the `image-uploader` project is now properly restricted by both limit ranges, and now by the resource quotas. Which will prevent us from:

- due to the limit ranges - we will never be able to deploy something that might hog or request too many resources in the `image-uploader` project
- due to the resource quotas - during the active execution of our containers inside the `image-uploader` project they will never be able to take more runtime resources than allowed

```
# first make sure that we are logged in with a user that has a cluster-admin
   role
$ oc login -u kubeadmin -p <kubeadmin-password> <cluster-host-address>

# first create the resource quota from the template
$ oc apply -f openshift/resource-quotas.yml -n image-uploader

# we can then list them and optionally describe the object for more details
$ oc get resourcequotas -n image-uploader

NAME                AGE    REQUEST
                                                   LIMIT
compute-resources   106s   pods: 2/10, requests.cpu: 0/2, requests.memory:
   0/2Gi   limits.cpu: 0/3, limits.memory: 0/3Gi
```

Creating resource quotas

Resource quotas track all resources in a project that are deployed by Kubernetes. Core components in OpenShift like deployment configs and build configurations are not covered by quotas, that is because these components are created on demand for each deployment and controller by OpenShift.

The components that are managed by resource quotas are the primary resources in an OpenShift cluster that consume storage and compute resources, keeping track of a project's resources is important when you need to plan how to grow and manage your OpenShift cluster to accommodate your application. The following components are tracked by resource quotas

- config maps - we discussed config maps in previous sections, they provide a way to configure and define data for containers
- persistent volume claims - applications requests for persistent storage
- resource quotas - the total number of quotas per project
- replication controller - the number of controllers in a project. This is typically equal to the number of deployed applications but you can also manually deploy applications using different workloads that could make this number change.
- secrets - we discussed them in previous section, they are a variation of the config maps
- services - the total number of services in a project
- image streams - the total number of image streams in a project

Most of the items in this list should look familiar, we have been discussing them for several sections at this point. The following listing shows the resource quotas template that you need to apply to the `image-uploader` project to do this apply the following file

```
apiVersion: v1
```

```
kind: ResourceQuota
metadata:
    name: object-counts
spec:
    hard:
        configmaps: "10"
        persistentvolumeclaims: "5"
        resourcequotas: "5"
        replicationcontrollers: "20"
        secrets: "50"
        services: "10"
        openshift.io/Image streams: "10"
```

Save the template to a file, and execute the commands below to apply the resource quota object, this will ensure that the `image-uploader` project is now properly restricted by OpenShift object or resource counts as well

```
# first create the resource quota from the template
$ oc apply -f openshift/storage-resource-quotas.yml -n image-uploader

# we can then list them and optionally describe the object for more details
$ oc get resourcequotas -n image-uploader
```

Working with quotas & limits

Now that the `image-uploader` project has limit ranges and quotas it is time to put them through their paces. The compute quota for the app is not being reflected yet and your first task is to fix that

Quotas to existing applications

When you deployed the apps in previous sections no quotas or limits were defined for the `image-uploader` project. As we mentioned when you were creating limit ranges back then you cluster was essentially the wild west and any deployed application could consume any amount of resources in the cluster. If an application is created and there are no limit ranges to reference an no resources were requested as when you deployed the metrics pod. The linux kernel components that define the resource constraints for each container are created with unlimited values for the resources limits. This is what happened when you deployed the app-cli and the app-gui and why their CPU and memory quotas are not reflected in OpenShift.

Now that you have applied limit ranges and quotas to the `image-uploader` project you have OpenShift to re-create the containers for these applications to include these constraints, The easiest way to to do this is to delete the current pods for each application. When you run the following `oc delete command` OpenShift will automatically deploy new pods that contain the default limit ranges, that you defined in the previous section.

```
# this will delete all pod resources related to those labels, note that  we
   are only deleting the pods, nothing else,
# the deployment and in particular the replication set will ensure that the
   pods are re-created with the correct limits
$ oc delete pod -l deployment=app-cli
$ oc delete pod -l deployment=app-gui
```

Because you did not specify specific resource values your new app-gui and app-cli pods inherit the default request values defined in the core-resource-limits limit range object. Each pod was assigned 200 millicores and 100 MIB of RAM. You can see that in the previous output that the consumed CPU and memory quotas for the `image-uploader` project are twice the default request.

It is definitely not a best practice to start using projects without having set limits and resources first, but we had to start somewhere, and if the very first few sections was all about quotas you would never have gotten to this section, so for teaching purposes, we began using OpenShift without discussing proper configuration rules

Changing quotas for deployed applications

When you deploy a new application you can specify limits and quotas as part of its definition. You can also edit the YAML definition for an existing deployment config directly from the command line. To edit the resource limits for your deployment run the following `oc edit command` which lets you edit the current YAML definition for the application.

```
# this will open your default system editor, and allow you to edit the
   contents of the manifest as if it was being done
# directly and interactively
$ oc edit deployment/app-cli
```

To edit the resource limits you need to find the `spec.containers.resources` section of the configuration. This section is currently empty, because nothing was defined for the application when it was initially deployed, we will change that.

```
resources:
    requests:
        cpu: "750m"
        memory: "500Mi"
    limits:
        cpu: "1"
        memory: "1000Mi"
```

This defines our pod as burstable, because the maximum limits that are defined are higher than the request, meaning that the pod can burst up to 1 CPU or that would mean 1000 millicores, as we have only requested 3/4 of that - 750 millicores, and the memory limit is twice as much as requested

Savings the new configuration will trigger a new deployment for the app-cli this new deployment will incorporate your new resource requests and limits. Once the build completes your deployment will be available with more guaranteed resources, you can also verify this with the regular describe command, or through the web UI console.

You can edit a deployment config to make complex changes to deployed applications but it is manual process. For new applications deployments your project should use the default limit ranges whenever possible to inherit default values

While your resource requests and limit ranges are new and fresh in your mind let us dig a little deeper and discuss how these constraints are enforced in OpenShift by the Linux kernel and the container runtime - like docker or containerd, using cgroups

Cgroups for managing resources

Cgroups are Linux kernel components, that provide per process limits for CPU, memory and network bandwidth and block storage bandwidth. In an OpenShift cluster they enforce the main limits and quotas configured for applications and projects.

Overview of the cgroups

Cgroups are defined in a hierarchy in the `/sys/fs/cgroup/` directory on the application node. Within this directory is a directory for each type of cgroup controller that is available. A controller represents a specific system resource that can be controller by cgroups. In this section we are focusing on the cpu and memory cgroups controllers. In the directories for the cpu and memory controllers is a directory named `kubepod.slice`. Cgroups slices are used to create subdivisions within the cgroups controller. Slices are used as logical dividers in a controller and define resource limits for groups of resources below them in the cgroup hierarchy.

```
# login into the node, and navigate to the directory, then we can actually
   see the structure of this directory
$ ls /sys/fs/cgroup

-r--r--r--.  1 root root 0 Jun 11 16:03 cgroup.controllers
-rw-r--r--.  1 root root 0 Jun 11 17:24 cgroup.max.depth
-rw-r--r--.  1 root root 0 Jun 11 17:24 cgroup.max.descendants
-rw-r--r--.  1 root root 0 Jun 11 16:03 cgroup.procs
-r--r--r--.  1 root root 0 Jun 11 17:24 cgroup.stat
-rw-r--r--.  1 root root 0 Jun 11 17:22 cgroup.subtree_control
-rw-r--r--.  1 root root 0 Jun 11 17:24 cgroup.threads
-r--r--r--.  1 root root 0 Jun 11 16:03 cpu.stat
-r--r--r--.  1 root root 0 Jun 11 16:03 cpuset.cpus.effective
-r--r--r--.  1 root root 0 Jun 11 17:24 cpuset.cpus.isolated
-r--r--r--.  1 root root 0 Jun 11 17:24 cpuset.mems.effective
drwxr-xr-x.  2 root root 0 Jun 11 16:03 dev-hugepages.mount
drwxr-xr-x.  2 root root 0 Jun 11 16:03 dev-mqueue.mount
drwxr-xr-x.  2 root root 0 Jun 11 16:03 init.scope
-r--r--r--.  1 root root 0 Jun 11 16:03 io.stat
drwxr-xr-x.  4 root root 0 Jun 11 16:04 kubepods.slice <- here is the
   kubepods directory we care about
drwxr-xr-x.  3 root root 0 Jun 11 16:03 machine.slice
-r--r--r--.  1 root root 0 Jun 11 17:24 memory.numa_stat
--w-------.  1 root root 0 Jun 11 17:24 memory.reclaim
-r--r--r--.  1 root root 0 Jun 11 16:03 memory.stat
-r--r--r--.  1 root root 0 Jun 11 17:24 misc.capacity
-r--r--r--.  1 root root 0 Jun 11 17:24 misc.current
drwxr-xr-x.  2 root root 0 Jun 11 16:40 proc-fs-nfsd.mount
drwxr-xr-x.  2 root root 0 Jun 11 16:03 sys-fs-fuse-connections.mount
drwxr-xr-x.  2 root root 0 Jun 11 16:03 sys-kernel-config.mount
drwxr-xr-x.  2 root root 0 Jun 11 16:03 sys-kernel-debug.mount
drwxr-xr-x.  2 root root 0 Jun 11 16:03 sys-kernel-tracing.mount
drwxr-xr-x. 34 root root 0 Jun 11 17:02 system.slice
drwxr-xr-x.  3 root root 0 Jun 11 16:19 user.slice

# if we navigate to that directory we will see that, indeed it has two
   directories, which we need to look at
$ ls kubepod.slice | grep kube

kubepods-besteffort.slice
kubepods-burstable.slice
```

The `kubepods` slice is where the configuration to enforce OpenShift requests and limits are located. Within the `kubepod.slice` are two slices `kubepods-besteffort.slice` and `kubepods-burstable.slice`. These two slices are how resource limits for best-effort and burstable quality of service levels that we have discussed in this section are enforced. Because you defined resource requests for app-cli and app-gui they both will be defined in `kubepods-burstable.slice`. Within the `kubepod-besteffort.slice` and the `kubepods-burstable.slice` are multiple additional slices. There is not an immediate identifier to tell you which slice contains the resource information for a give container, but you can get that information directly from docker on your application node.

Identifying container cgroups

To determine which cgroup slice controls the resources for your deployment, we need to get the cgroup information from the container runtime. The cgroup slice that each container belongs to is listed in the information from the inspect command. To obtain filter on the `cgroupsPath` element accessor. This limits the output to only the cgroup slice information. In your example the cgroup slice for the app-cli is the following long id, which signifies the type is burstable as well, we can see that from the value for the `cgroupsPath` key in the inspect command -

```
# here you will notice that we get two outputs that is because we have two
   different replicas for our deployment, but
# that will vary depending on your current state of the deployment
$ crictl ps | grep app-cli | awk '{print $1'} | xargs -I'{}' crictl inspect
   {} | grep cgroupsPath

"cgroupsPath": "kubepods-burstable-pod70aff84f_0aa8_47ed_9b8c_cd3d6a185708.
   slice:crio:7
   f4ec5610c6a8f4110f43ac2d415d6d20449ff916bc8eb81407bfbde3438680a"
"cgroupsPath": "kubepods-burstable-pod58448c42_a7bd_4996_9012_849df2cffa5c.
   slice:crio:2
   f1a0714812de8170f791ddf3dc008b39c9a020028f1b42418ed47133092859e"
```

As we mentioned, we are in the burstable slice still. The slice defined in the app-cli inspect output that is. Slices do not define resource constraints for individual containers but they can set default values for multiple containers. That is why the hierarchy of slices look a little excessive here. You have one more layer to go to get to the resource constraints for the app-cli container. In the lowest slice is a scope directory. Each scope is named after the full hash that a container's short IO is based on. In our example app-cli resource constraints are defined in the scope named

```
# navigate to the burstable slice sub directory, and check out the output of
   the following ls command, we can see that
# for the two pods that we have with the respective IDs that start with 80aff
    and 58448, there are two slices
$ cd /sys/fs/cgroup/kubepods.slice/kubepods-burstable.slice

# grep the directory content for the two pods, based on the pod id, we can
   see
$ ls | grep "pod\(70aff\|58448\)"
kubepods-burstable-pod58448c42_a7bd_4996_9012_849df2cffa5c.slice
kubepods-burstable-pod70aff84f_0aa8_47ed_9b8c_cd3d6a185708.slice
```

Cgroups configurations are created on an OpenShift application nodes using this process. It is a little complex and because cgroups are listed according to the cgroup controller and not the PID they manage, troubleshooting them can be a challenge on a busy system. When you need to see the cgroup configuration for a single container, the process is more straightforward. In the next section we will look at how the cgroup information from the host is mounted in each container that is created

Confirming cgroups limits

When a container runtime creates a container, it mounts the cgroup scope that applies to it in the container, in the `/sys/fs/cgroup` directory, it truncates the slices and scope so the container appears to have only a single cgroup controller. We are going to focus on the limits that enforce CPU and memory constraints for the app-cli container. Let us begin with the limits for the CPU consumption. To start an interactive shell prompt in your running container, run the following command edited to reference your container short ID

```
# first make sure to select one of the containers that are currently being
   active for the deployment, that could be
# either one of those, or you can scale down your deployment to one replica
   in case this is confusing
$ crictl ps | grep app-cli | awk '{print $1'}
7f4ec5610c6a8
2f1a0714812de

# we use the crictl, or the containerd runtime to log into the container
   through the interactive use of /bin/bash
$ crictl exec -it 7f4ec5610c6a8 /bin/bash
```

As we discussed earlier in this chapter CPU resources in OpenShift are allocated in millicores or one-thousands of the CPU resources available on the server. For example if your application node has two processors, a total of 2,000 millicores is available for the containers on the node. The ration expressed here is what is represented in the cpu cgroup. The actual number is not expressed in the same units, but the ratios are always the same. The app-cli container has request of 750 millicores, with the limit of 1,000 millicores or one CPU. You need the following two values form `/sys/fs/cgroup/cpu.max` to build a ratio that confirms that the limit for the app-cli container is correct configured. The file is one line, which defines two values, divided by space, the meaning of the two values is described below:

- `cat cpu.max | awk '{print $1}'` - the time period in microseconds during which the cgroup quota for the processor access is measured and reallocated, this can be manipulated to create different processing quota ratios for different applications.
- `cat cpu.max | awk '{print $2}'` - the time in microseconds that the cgroup is allowed to access the processor during the defined period, the period of time is adjustable. For example if that `value` value is 100, the cgroup will be allowed to access the processor 100 microseconds during the set period, if that period is also 100 , that means the cgroup ha unlimited access to the processor, on the system. If the period were set to 1000, the process would have access to the processor for 100 microseconds out of every 1,000.

For the app-cli this cgroup limits the container access to 1 CPU during 100,000 out of every 100,000 microseconds. If you convert these values to a ratio app-cli is allocated a maximum of 1,000 millicores of 1 CPU. That is the limit we have set for app-cli. This is how CPU time limits are managed for each container in an application deployment Next let us look at how the request values are controller by cgroups.

The request limit for app-cli is managed by the value in `/sys/fs/cgroup/cpu/cpu.weight`. This value is a ratio of CPU resources relative to all the cores on the system.

The memory limit for the app-cli is controlled by the value in /sys/fs/cgroup/memory/memory.current, there are some other files like memory.max memory.min and memory.peak, which are mostly self explanatory, the one that is more interesting is the memory.max, which defines the maximum amount of allowed memory that the container can take up which in this case matches perfectly with the limits configuration, meaning that it has a value of - `1048576000` - which is as configured a maximum amount of 1GB, The memory.current, varies and depends on the app that is running for app-cli that value is at 350MB - `34635776`

Resource limits for OpenShift containers are enforced with kernel cgroups. The only exception is hte memory request value. There is no cgroup to control the minimum amount of RAM available to a process this value is primarily used to determine which node a pod is assigned to in your OpenShift cluster.

This section covered a lot of what is required to create and maintain a healthy OpenShift cluster. We have gone far down into the Linux kernel to confirm how container resources limits are enforced. Although limits requests and quotas are not the most exciting things to work through they are absolutely critical and essential component of OpenShift ready to handle production workloads effectively.

You cluster is now connected with an authentication database and the project you have been working on has effective resource limits and quotas. In the following sections we will keep building on that momentum

To summarize this is how the values are distributed, note that this table is using cgroup v1 values as reference, the formula for cgroup v2 is a bit different. It is worth noting that during the last couple of years the container runtimes have moved to the cgroups v2, which change a few things among which are the default values for different limits, such as the cpu.weight, which in the older cgroups v1 were called cpu.shares. The shares value and weight have different scaling and they are calculate very differently, most container runtimes still use the old cgroups v1 as reference, and they adjust and scale the value to match cgroups v2 specification when creating the containers. The reason being that the OCI - Open container reference spec, was written with cgroups v1 in mind

OpenShift Value	cgroup v1 File	cgroup v2 File
cpu: 1000m limit	cpu.max	cpu.max
cpu: 750m request	cpu.cfs_quota_us, cpu.cfs_period_us	cpu.weight
memory: 500Mi request	N/A	N/A
memory: 1000Mi limit	memory.max	memory.max

Networking

The importance of the network design configuration in an OpenShift cluster can not be overstated, it is the fabric of what binds you cluster together. With that perspective in mind OpenShift does a lot of work to make sure its networking configuration is stable performs well and is highly configurable and available. Those principles are what we will cover in this section. Let us start with an overview of how the network stack in OpenShift is designed

Managing the SDN

`OVS` is an enterprise grade scalable high performance software defined network - in OpenShift its the default SDN used to create the pod network in your cluster, it is installed by default when you deploy OpenShift. `OVS` runs as a service on each node in the cluster, you can check the status of the service by running the following `systemctl` command on any node

```
# firs login into the cluster node, and then query systemd for the status of
   the service on the node
$ sudo -i
$ systemctl status ovs-vswitchd
```

```
# you will see something like this, which shows that the service is running
   and enabled out of the box
 ovs-vswitchd.service - Open vSwitch Forwarding Unit
     Loaded: loaded (/usr/lib/systemd/system/ovs-vswitchd.service; static)
    Drop-In: /etc/systemd/system/ovs-vswitchd.service.d
             10-ovs-vswitchd-restart.conf
     Active: active (running) since <date>; <time> ago
   Main PID: 1204 (ovs-vswitchd)
      Tasks: 15 (limit: 152745)
     Memory: 57.1M
        CPU: 1min 32.243s
     CGroup: /system.slice/ovs-vswitchd.service
             1204 ovs-vswitchd unix:/var/run/openvswitch/db.sock -vconsole:
                emer -vsyslog:err -vfile:info --mlockall --user openvswitch:
                hugetlbfs --no-chdir --log-file=/var/log/openvswitch/ovs-
                vswitchd.log --pidfile=/var/run/openvswi>
```

```
The service is automatically enabled on cluster nodes as part of the OpenShift deployment
. The configuration file for the OVS service is located at /etc/systconfig/opensvwitch
and each node's local OVS database is located in the /etc/openswitch directory For day-to
-day operations. OVS should be transparent. Its configuration and updates are controller
 by OpenShift. Using OVS provides several advantages to OpenShift. jthis transparent
operation is possible because OpenShift uses the Kubernetes container network interface
 - the container network interface provide a plugin architecture to integrate different
solutions to create and mange the pod network. OpenShift uses OVS as its default but it
can function with other network providers as well.
```

The `OVS` used in your OpenShift cluster is the communication backbone for all your deployed pods, traffic in an out of ever pod is affected by it, in the OpenShift cluster. For that reason you need to know how it works and how to effectively use it for your needs. Let us start with the network configuration of your OpenShift application node.

Configure application node network

When a node is added to an OpenShift cluster several network interfaces are created in addition to the standard loopback interface and `eth0` physical interface. For our purposes we will call `eth0` the physical interface even though you are using a virtual machine for your cluster. That is because OpenShift creates the following additional virtual interface.

- `br0` - An `OVS` bridge all OpenShift interfaces are associated with. `OVS` creates this interface when the node is added to the OpenShift cluster.
- `tun0` attached to `br0`. Acts as the default gateway for each node. Traffic in and out of your OpenShift cluster is routed through this interface.
- `vxlan_sys_4789` - also attached to `br0` this virtual extensible local area network is encrypted and used to route traffic to containers on other nodes in your cluster. It connects the nodes in your OpenShift cluster to create your pod network.

Additionally each pod has a corresponding virtual Ethernet interface that is linked to the `eth0` interface in the pod by the Linux kernel. Any network traffic that is sent either interface in this relationship is automatically presented to the other. All of these relationships are

```
What are Linux bridges, TUN interfaces, and VXLAN - a Linux bridge is a virtual interface
```

```
 that is used to connect other interfaces together, If two interfaces on a host are
attached to abridge they can communicate with each other without routes needing to be
created. This help with communication speed as well as keeping networking configuration
 simple on the host and in the container. A VXLAN is a protocol that acts as an overlay
 network between teh nodes in your OpenShift cluster, an overlay network is a software
defined networks that is deployed on top of another network. The VXLAN used in OpenShift
are deployed on top of the networking configuration of the host To communicate securely
between pods, the VXLAN encapsulates pod network traffic in an additional layer of network
 information so it can be delivered to the proper pod on the proper sever by IP address.
The overlay network is the pod network in your OpenShift cluster. The VXLAN interface on
each node provide access to and from that network. You can find the full definition and
specification for VXLAN and at its RFC doc.
```

You can see these interfaces on your application nodes by running the IP command. The following sample output has been trimmed with a little command line magic for brevity and clarity:

```
$ ip a | egrep '^[0-9].*:' | awk '{print $1 $2}'

1:lo:
2:ovs-system:
3:ovn-k8s-mp0:
4:br-int:
5:eth10:
6:tap0:
8:br-ex:
9:cf3abd918c8e682@if2:
10:9943d81586b22a8@if2:
12:363ff29e19ebff4@if2:
14:b2cf78b2daa4f7b@if2:
...
```

The networking configuration for the master node is essentially the same as an application node. The master node uses the pod network to communicate with pods on the application nodes as they are deployed deliver their application and are eventually deleted. In the next section we will look at more deeply at how the interface in the container is linked to a corresponding `veth` interface on the cluster node.

Linking containers to host interfaces

In previous sections we talked about the network namespace and how each container contains a unique loopback and `eth0` interface for network communication. Form the perspective of application in a container these two interface are the only networks on the host, to get network traffic in an out of the container the `eth0` interface in the container is linked in the Linux kernel to a corresponding `veth` interface in the host's default network namespace

The ability to link two interfaces is a feature of the linux kernel. To determine which `veth` interface a container is linked to you need to log into the application node where the container is running you can figure this out in just a few steps let us use the app-cli as an example

Any virtual interface on a Linux system can be linked by the kernel to another virtual or physical interface. When an interface is linked to another the kernel makes them essentially the same interface. If something happens to one interface it automatically happens to its linked interface in an interface `iflink` file a file created and maintained by the running Linux kernel at `/sys/class/net/<interface-name>/iflink` is the index number for its linked interface. To find the linked interface number for the app-cli container run the following oc exec command making sure to use the pod ID for you app-cli deployment. This command uses the cat command line tool to echo the contents of the iflink file.

```
# the file contains only a single number which in this case represents the
   number of the linked interface
$ oc exec app-cli-7976b4c888-rdbv7 -- cat /sys/class/net/eth0/iflink
115
```

The eth0 interface in the app-cli pod is linked to interface 115, on the application node. But which veth interface is number 115 ? That information is available in the output form the IP command. The link ID also called the `ifindex` for each interface is the number at the beginning of each interface listed in the command. For each eth0 interface in a container its `iflink` value is the `ifindex` value of its corresponding veth interface.

```
# first make sure you have entered the cluster node in the first place
$ ssh crc

# grep for the virtual interface, that exactly contains and starts with the
   number we extracted from the iflink file
# above, that would be 115 in our example, but you result and interface link
   number will be different
$ ip a | grep -A 3 '^115.*:'
115: 8a4aad65b71dfe8@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc
   noqueue master ovs-system state UP group default
    link/ether 5a:2a:70:d2:bc:92 brd ff:ff:ff:ff:ff:ff link-netns fc30e1db
       -0655-46e5-bfac-157def87be33
    inet6 fe80::582a:70ff:fed2:bc92/64 scope link
       valid_lft forever preferred_lft forever
```

We have now confirmed that the app-cli pod is linked by the Linux kernel to the virtual interface [8a4aad65b71dfe8] - on the cluster node. This is how network traffic enters and exist containers in general. Next let us confirm that this veth on the node is connected to the cluster's pod network so network traffic can get in and out of the OpenShift cluster

Working with OVS

The command line tool to work with `OVS` directly is `ovs-vsctl`. To use this tool you need to be logged in on to the host cluster you are looking for information about. In these examples we are logged in the cluster node already.

We mentioned earlier that all OpenShift SDN interfaces are attached to an `OVS` ridge named `br0`. We make this distinctions of calling it an `OVS` bridge because it is a bridge interface that is created and controller by `OVS` itself. You can also create a bridge interface with the Linux kernel. A linux bridge is created and managed using the `brctl` command. You can confirm that the `bridge` interfaces are being controller by `OVS` by running the following `ovs-vsctl` command to list active `OVS` bridges

```
$ ssh crc
$ sudo - i
$ ovs-vsctl list-br

# we have two interfaces, from the names we can deduce that one is internal
   and the other is called external.
ovs-vsctl list-br
br-ex
br-int
```

```
# let us see what the internal one contains first, the output is abridged but
    these are the identities of the veth
# interface for each pod that is running in the cluster node, cluster has a
   lot of operator and utility pods that
# run besides our own app pods
$ ovs-vsctl list-ifaces br-int

0549f003ca15713
07ef7817039d9ff
0dde308ad503bef
...............
1fd82e5f2d18b18
2427839e331de3f
252a3d9203d2239
363ff29e19ebff4
388c3b826f0b846
3d2158eb7ef5b90
46022cd19537541
496573f934b3956
4dc687e9870a5ac
50023d8075b388c
503b86107070900
51386e7bf3410e9
5859aa3a89d522d
646690890e9b6d1
76f5605e245b613
7c5e9807644e8cb
80c6c6f7b10f830
...............
8b4e358d3c1e190
8c88cf893b34724
8f2491e5cd6475b
ovn-k8s-mp0
patch-br-int-to-br-ex_crc <- take a good note of this interface

$ ovs-vsctl list-ifaces br-ex

patch-br-ex_crc-to-br-int <- take a good note of this interface
tap0
```

If you have not used Linux bridges before it can seem confusing when you know a bridge should be present bu none appears when you run `brctl`, because they are being managed by `OVS` The node has a single `OVS` bridges named `br-ex and br-int`

The output of the command above, lists all interfaces connected to this bridge br-int. This is how OpenShift SDN function, when a new pod is deployed a new veth interface is created and attached to br-int. At that point the pod can send and receive network traffic on the pod network. It can communicate outside the cluster through the br-int and br-ex

In the next section we will put the bridge interfaces and the SDN to work by digging deeper into the how application traffic is routed and how application communicate in your cluster. Let us start at the beginning with a request for the app-cli deployment

Routing application requests

When you browse to the route of app-cli at - http://app-cli-image-uploader.apps-crc.testing/ your request goes to the node, first on port 80, the default HTTP port. Log in to the cluster node and run the following netstat command to determine which service is listening to port 80

```
$ netstat -tpl --numeric-ports | grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
        73357/haproxy
```

There is an HAProxy service running on the port in the node, that is interesting. HAProxy is the front door to your application in OpenShift. HAProxy is an open source software defined load balancer and proxy application. In OpenShift it takes the URL route associated with an application and proxies those requests into the proper pod to get the requested data back to the requesting user. Those requests into the proper pod to get the requested data back to the requesting user. We wont dig too much into all that HAProxy can do we are focusing on how OpenShift uses HAProxy.

Using the HAProxy service

The routed pod runs in the project named openshift-ingress in OpenShift. The router pod handles incoming user requests for your OpenShift cluster application and proxies them to the proper pod to be served to the user. The router pod listens directly on the host interface for the node its deployed on and uses the pod network to proxy requests for different applications to the proper pod. This session then returns to the user from the pod host through the TUN interface.

1. The user requests information for app-cli by the route's URL which connect to the OpenShift cluster node
2. The HAProxy pod takes the request URL and maps it to its corresponding pod
3. HAProxy uses the pod network to proxy the connection to a node where an app-cli pod is deployed
4. The request goes through the pod network and is passed into the app-cli pod
5. The TUN interface attached to the bridge routes traffic to the host network interface
6. The app-cli processes the request and sends the response through its host TUN interface back to the user

Because the router listens directly on the host interface its configured differently than a typical pod in OpenShift. In the next section we will investigate the HAProxy in more detail

How does HAProxy always deploy to the same node, in OpenShift it is possible to tag an entire cluster application node with a label, then when deploying specific pods you can tell them on which cluster node to be deployed based on this label, similarly to how the names of the namespaces can be used to deploy applications in different project namespaces

This is done using the `nodeSelector` value in the manifest file, in the deployment configuration component. The default OpenShift router has a node selector that specifies the node with the matching region=infra label, you can see this node selector in the router deployment config like so

```
$ oc get deployment -n openshift-ingress

NAME                READY   UP-TO-DATE   AVAILABLE   AGE
router-default      1/1     1            1           87d
routes-controller   1/1     1            1           20h

```

```
# the node selector for the deployment router-default might look something
   like this
nodeSelector:
    region: infra
    kubernetes.io/os: linux
    node-role.kubernetes.io/master: ""
```

Investigating the HAProxy service

The `lsns` tool you used in previous sections displays the namespaces associated with the HAProxy process listening on port 80. The following `lsns` command woks in your example cluster. First we have to find the PID of your app pod

```
$ sudo -i
# here is what we can do to extract the pid of the router-default pod, this
   will inspect the pod, and fetch the pid
$ crictl ps | grep router-default | awk '{print $1'} | head -n1 | xargs -I'{}
   ' crictl inspect {} | grep "\"pid\":"

# we might get a result like that, which points to the PID of that container,
    we can then use it to list the namespaces
# bound to this process
> "pid": 7702,

# here we list all namespaces which this process has ownership of, take a
   good look at the first 3 namespaces, and note
# that they are different than the others, that is because these reference
   namespaces directly on the host, you can see
# that they are owned by PID 1, on the host, that is. Another giveaway is
   that the number of processes NPROCS is quite
# large for the very first 3 namespaces, which kind of should also ring some
   alarm bells, while the other namespaces owned
# by our process 7702, has just a couple of NPROCS running inside that
   namespace
$ lsns -p 7702

        NS TYPE   NPROCS   PID USER       COMMAND
4026531834 time      560     1 root       /usr/lib/systemd/systemd --system
   --deserialize 41
4026531837 user      560     1 root       /usr/lib/systemd/systemd --system
   --deserialize 41
4026531840 net       435     1 root       /usr/lib/systemd/systemd --system
   --deserialize 41
4026533678 uts         2  7702 1000560000 /usr/bin/openshift-router --v=2
4026533679 ipc         2  7702 1000560000 /usr/bin/openshift-router --v=2
4026533715 mnt         2  7702 1000560000 /usr/bin/openshift-router --v=2
4026533716 pid         2  7702 1000560000 /usr/bin/openshift-router --v=2
4026533722 cgroup      2  7702 1000560000 /usr/bin/openshift-router --v=2
```

Note something strange in the output, we can see that the PID for the time, user and net namespaces are 1, which means that they are using the namespaces from the host, the PID column shows who owns this namespace, PID 1 is always always the very first process that is started on the host, that is not not the router pod PID which is 7702. Using the host network namespace lets HAProxy listen directly on the host interfaces for incoming requests. Listening on the host interface means HAProxy receives application requests directly acting as OpenShift front door for application traffic. The router pod has its own mount namespace, which means that config files for HAProxy are isolated in the container. To enter the router pod, run the following `oc rsh` command, substitute the name of your router pod, this will initialize an `ssh` like session into the pod

```
# change the context namespace first and then get the list of pods that are
   of interest to us
$ oc project openshift-ingress
$ oc get pods
NAME                                 READY   STATUS    RESTARTS   AGE
router-default-6fcb8bbdff-p9sqk      1/1     Running   0          20h
routes-controller-75bcb6d4c4-dbhjh   1/1     Running   0          20h

# this will establish an ssh like session into the pod
$ oc rsh router-default-6fcb8bbdff-p9sqk

# now after the remote session into the pod is established run the following
   from the pod
$ ip a

# this is an abridged output, but take a good look at the very first
   interface, that is the interface 115, we already
# saw in the host, for the app-cli pod, now the same interface is also
   visible from the router pod, not only that one but
# actually all of them, why ? Well the ip command is executed in the
   namespace of the host, not the pod, that is, why we
# can access interfaces on the host
......
115: 8a4aad65b71dfe8@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc
   noqueue master ovs-system state UP group default
    link/ether 5a:2a:70:d2:bc:92 brd ff:ff:ff:ff:ff:ff link-netnsid 63
    inet6 fe80::582a:70ff:fed2:bc92/64 scope link
       valid_lft forever preferred_lft forever
500: f595c598b9740a2@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc
   noqueue master ovs-system state UP group default
    link/ether f6:f0:4d:04:bd:b9 brd ff:ff:ff:ff:ff:ff link-netnsid 44
    inet6 fe80::f4f0:4dff:fe04:bdb9/64 scope link
       valid_lft forever preferred_lft forever
511: d3138ecde33decd@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc
   noqueue master ovs-system state UP group default
    link/ether 42:7f:38:ab:b8:07 brd ff:ff:ff:ff:ff:ff link-netnsid 40
    inet6 fe80::407f:38ff:feab:b807/64 scope link
       valid_lft forever preferred_lft forever
.....
```

HAProxy and request routing

The config file for HAProxy is in the pod at `/var/lib/haproxy/conf/haproxy.config`. This config file is maintained by OpenShift cluster and operator / controllers. Any time an application is deployed, updated or deleted, OpenShift updates this config file and has the HAProxy process reload it. Let us see this in action

```
# while still in the router pod, we can cat this file out
$ cat /var/lib/haproxy/conf/haproxy.config | grep app-cli

# here is the magic, you can actually see the config for both pods, that we
   have running for this app at the moment
backend be_http:image-uploader:app-cli
  server pod:app-cli-7976b4c888-zqx5l:app-cli:8080-tcp:10.217.0.86:8080
     10.217.0.86:8080 cookie 24406d64b9747a68dacf13f017996cea weight 1 check
     inter 5000ms
  server pod:app-cli-7976b4c888-rdbv7:app-cli:8080-tcp:10.217.0.87:8080
     10.217.0.87:8080 cookie a4765a9ead710ffdc2e669795c0ea2b4 weight 1 check
     inter 5000ms

# from your host machine, run the following command to make the pod replicas
   less
$ oc scale deployment/app-cli --replicas=1 -n image-uploader

# from the router pod now do execute the command again, we can see that only
   one entry is now present, which matches
# the 1 replica we have for this project after the scaling was performed
   above
$ cat /var/lib/haproxy/conf/haproxy.config | grep app-cli
backend be_http:image-uploader:app-cli
  server pod:app-cli-7976b4c888-rdbv7:app-cli:8080-tcp:10.217.0.87:8080
     10.217.0.87:8080 cookie a4765a9ead710ffdc2e669795c0ea2b4 weight 1
```

We will not go into depth on what all of these fields mean, however you can clearly see that there are the ip addresses of the pods, as well as the names and ids of these pods. HAProxy takes the request from the user maps the requested URL to a defined route in the cluster and proxies the request to the IP address for a pod in the service associated with that route. All this traverses the pod network created by OpenShift SDN.. This process works in concert with iptables on each host. OpenShift uses a complex dynamic iptables configurable to make sure requests on the pod network are routed to the proper application pod. IPtables are a complex topic that we do not have the space to cover here.

The method for routing requests in OpenShift works well. But it poses a problem when you deploying applications that depend on each other to function, if a new pod is added to an application or a pod is replaced and it receives a new IP address the change would require all applications that reference it to be updated and redeployed. This is not a serviceable solution. Luckily OpenShift incorporates a DNS service on the pod network. Let us examine that next

Locating services with internal DNS

Applications depend on each other to deliver information to users. Middleware apps depend on databases. Web presentation layers depend on middleware. In tan application spanning multiple independently scalable pods these relationships are complex to manage to make this easier OpenShift deployed `SkyDNS` - when the cluster is deployed, and makes it available on the pod network, is a DNS service that uses etcd, the primary Kubernetes database to store DNS records. Also known as zone files, are config files where DNS records are recorded for a domain controller by a DNS server, in OpenShift `SkyDNS` controls the zone files for several domains that exist only on the pod network. `cluster.local` - top level domain for everything in your OpenShift cluster, `svc.cluster.local` - domain for all services running in your cluster.

Domains for each project are also created. For example `image-uploader.svc.cluster.local` - used to access all the services created in the `image-uploader project`. A DNS A record is created in `SkyDNS` for each service in OpenShift when an application is deployed a service represents all the deployed pods for an application. To view the services for the `image-uploader project` run the following `oc` command

```
# we can see the services here, however what is more interesting is the route
   , which was created earlier by the expose command,
# and is directly linked to the service, the router object is told which
   service to serve on a given path, port and so on
$ oc get services -n image-uploader
NAME      TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)             AGE
app-cli   ClusterIP   10.217.5.54   <none>        8080/TCP,8443/TCP   21h

# here we can see in the route manifest that the route is linked to the
   service, here is also some more interesting
# parts of the spec, like the `targetPort`, and the `host`, which points to
   the <app-name>-<project-name>-<cluster-host>
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  labels:
    app: app-cli
    app.kubernetes.io/component: app-cli
    app.kubernetes.io/instance: app-cli
    app.kubernetes.io/name: php
  name: app-cli
spec:
  host: app-cli-image-uploader.apps-crc.testing
  port:
    targetPort: 8080-tcp
  to:
    # the name of the service is part of the route manifest, this is what
       links the service with the route, and the
    # route is what links our service to the outside world, the pods are
       linked through the service, and our app/container is
    # linked through the pod
    kind: Service
    name: app-cli
    weight: 100
  wildcardPolicy: None
```

We can see that the following relationship is established between the app and the outside world/traffic - ingress/traffic -> route -> service -> pod -> container -> application

DNS resolution in pod network

When a pod is deployed the `/etc/resolv.conf` file from the application node is mounted in the container in the same location. In linux `/etc/resolv.conf` configures the servers and is used for DNS service name resolution. By default `/etc/resolv.conf` on the application node is configured with the IP address for the node itself. DNS requests on each application node are forwarded to the `SkyDNS` running on the master server node.

The search parameter in `/etc/resolv.conf` is also updated when its mounted in the container it is updated to include `cluster.local svc.cluster.local` and all other domains manged by the `SkyDNS` service. Any domain defined in the search parameter in `resolv.conf` are used when a fully qualified domain name is not used for a hostname. FQDN are defined with an RFC document, but the general gist of what they are is that they are a complete address on a network. The domain `server.domain.com` is fully qualified where `server` only is not a complete domain name. The search parameter provides one or more domains that are automatically appended to the non FQDN to use for DNS queries

When a request comes in from your cluster those requests are automatically forwarded to the master server where `SkyDNS` handles requests. Let us test this in action. The format is `service_name.project_name.svc -cluster.local:port`. The following example is run from the node, you can run the same command from within a pod without specifying the FQDN because `/etc/resolv.conf` has the `SkyDNS` search domains added. Using the `oc rsh` you can enter the namespace for the app-cli pod and use curl to download the index page from app-cli and the default page for the router service

```
TODO
```

Configure OpenShift SDN

When you deploy OpenShift the default configuration for the pod network topology is a single flat network. Every pod in every project is able to communicate without restrictions. OpenShift SDN uses a plugin architecture that provides different network topologies in OpenShift. There are currently three OpenShift plugins that can be enabled in the OpenShift config without making large changes to your cluster.

- `ovs-subnet` - enabled by default, creates a flat pod network allowing all pods in all projects to communicate with each other.
- `ovs-multitenant` - separates the pod by project the application deployed in a proEct can only communicate with pods deployed in the same project, you will enable this one layer on in this section
- `OVS-networkpolicy` - provides fine grained ingress and egress rules for application. This plugin provides a lot of config power, but the rules can be complex. This plugin is out of scope

The Kubernetes container network interface accepts different networking plugins. OpenShift SDN is the default CNI plugin in OpenShift it configures and manages the pod network for your cluster, let us review the available OpenShift SDN plugins

Using the ovs-subnet

Earlier you were able to communicate directly with an application from the stateful-apps project from a pod in the `image-uploader` project you could do so because of how the `ovs-subsnet` plugins configured in the pod network. A flat network topology for all pods in all projects lets communication happen between any deployed applications

With this setup, an OpenShift cluster is deployed like a single tenant, with all resources available to one another. If you need to separate network traffic for multiple tenants you can use the multitenant plugin.

Using the ovs-multitenant

The `ovs-multitenant` network plugins isolated pod communications at the project level. Each pod for each application deployment can communicate only with pods and services in the same project on the pod network. For example the app-gui and app-cli pods can communicate directly because they are both in the same `image-uploader` project namespace. But they are isolated from the `wildfly-app` application in the `stateful-apps` project in your cluster. This isolation relies on two primary tools in `Open vSwitch`

- VXLAN - network identifier - acts like in a fashion similar to a VLAN in a traditional network. It is a unique identifier that can be associated with an interface and used to isolate communication to interfaces with the same VNID.
- OpenFlow - is a communication protocol that can be used to map network traffic across a network infrastructure. OpenFlow is used in OpenShift to help define which interfaces can communicate and when to route traffic through the vxlan0 and tun0 interfaces on each node.

When the ovs-multitenant plugin is enabled each project is assigned a VNID. The VNID for each project is maintained in the etcd database on the OpenShift master node. When a pod is created its linked veth interface is associated with the project's VNID and OpenFlow rules are created to make sure it can communicate only with pods in the same project. The router and registry pods in the default project are assigned VNID-0 this is a special VNID that can communicate with tall other VNID on a system. If a pod needs to communicate with a pod on another host, the VNID is attached to each packet

With the ovs-multitenant plugin enabled if a pod needs to communicate with a pod in another project the request must be routed off the pod network and connect to the desired application through its external route like any other external request this is not always the most efficient architecture. The OpenShift SDN ovs-networkpolicy plugin provides more fine grained control over how applications communicate across projects.

Creating advanced network designs

The ovs-networkpolicy plugin provides fine grained access control for individual applications regardless of the project they are in these rules can become complex very quickly we do not have the space to cover this here, but you can learn more about them on the official documentation site of OpenShift

Enabling multi tenant plugin

To enable the multi tenant plugin you need to ssh into your master node and application nodes (if you have more than one node cluster configured) and edit a cluster network config

```
# on the master server, login with a correct user that has cluster-admin
   roles, first lets see what is in there
$ oc describe network/cluster

# this describes the network cluster object that is currently deployed on the
    master server, you can see some of the
# details we had already mentioned below in the output
Name:          cluster
Namespace:
Labels:        <none>
Annotations:   <none>
API Version:   config.openshift.io/v1
Kind:          Network
Metadata:
  Creation Timestamp:  <date>
  Generation:          6
  Resource Version:    26060
  UID:                 bc505ed9-75c3-4f2a-bc0f-db8a0769e5cd
Spec:
  Cluster Network:
    Cidr:          10.217.0.0/22
    Host Prefix:  23
  External IP:
    Policy:
  Network Diagnostics:
    Mode:
    Source Placement:
    Target Placement:
  Network Type:  OVNKubernetes
  Service Network:
    10.217.4.0/23
Status:
  Cluster Network:
    Cidr:               10.217.0.0/22
    Host Prefix:        23
  Cluster Network MTU:  1400
  Conditions:
    Last Transition Time:  <date>
    Message:
    Observed Generation:   0
    Reason:                AsExpected
    Status:                True
    Type:                  NetworkDiagnosticsAvailable
  Network Type:            OVNKubernetes
  Service Network:
    10.217.4.0/23
Events:  <none>

# first make sure to apply the following network policies, these are a
   prerequisite, to allow us to continue with
# the creation and setup of the multitenant plugin
$ oc apply -f openshift/network-policy-multitenant.yml
networkpolicy.networking.k8s.io/allow-from-openshift-ingress created

# here after applying the configuration from above you should be able to see
   the effects of the net network policies
# in effect, created on the master cluster node
$ oc describe networkpolic
Name:         allow-from-openshift-ingress
Namespace:    openshift-ingress
Created on:   <date>
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     <none> (Allowing the specific traffic to all pods in this
     namespace)
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      NamespaceSelector: policy-group.network.openshift.io/ingress=
  Not affecting egress traffic
  Policy Types: Ingress
```

```
Name:          allow-from-openshift-monitoring
Namespace:     openshift-ingress
Created on:    <date>
Labels:        <none>
Annotations:   <none>
Spec:
  PodSelector:     <none> (Allowing the specific traffic to all pods in this
     namespace)
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      NamespaceSelector: network.openshift.io/policy-group=monitoring
  Not affecting egress traffic
  Policy Types: Ingress

Name:          allow-same-namespace
Namespace:     openshift-ingress
Created on:    <date>
Labels:        <none>
Annotations:   <none>
Spec:
  PodSelector:     <none> (Allowing the specific traffic to all pods in this
     namespace)
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      PodSelector: <none>
  Not affecting egress traffic
  Policy Types: Ingress
```

After these changes are committed and done you have to make sure to restart the master node server, you can do this with the command line on your host machine like so `crc stop` and then followed by `crc stop`, the changes will take effect after the server is restarted

```
TODO
```

Testing the multi-tenant plugin

Previously in the section you logged in to the app-cli pod using the `oc rsh` and downloaded the web index pages for the other applications, now we are going to try to do the same. We will connect to the pods again, and try to download the index pages for both applications again, from the two different projects, one of the apps will be from the current project `image-uploader` project and the other from the `wildfly-app` project, but it does not really matter which one you choose as long as its a different one than the `image-uploader` project

Security

Each topic in this chapter is specific to security and to making OpenShift a secure platform for your application. This chapter is not a comprehensive summary of OpenShift security features that would take many lines of words or and is a great idea for a standalone document. What we will do in this section is walkthrough the fundamentals of OpenShift we want to give you examples of what we think are the most crucial concepts and we will do our best to point you in the right direction, for the topics we do not have room to cover

We begin discussing important security concepts and making OpenShift secure not long after the very first section,

- Understanding OpenShift role in your environment
- Deploying applications with specific users
- Diving deep into how container processes are isolated
- Confirming application health and status
- Autoscaling applications to automate resilience
- CI/CD pipeline so humans do not have to be involved
- Working with persistent storage
- Controlling access to pods and handling interactions between pods
- Using identity providers and working with roles, limits and quotas
- Creating a secure stable networking

We may be using a broad definition of security her but every section in this documentation, contributes to your understanding of OpenShift and how to deploy it in an automated and secure fashion. Automation and security go hand in hand, because humans are not good at repetitive task. The more you can automate task for your application the more secure you can make those applications Even though we have already covered a lot of ground regarding security we will still need to devote this entire section to security specific topics

OpenShift has layers of security form the Linux kernel on each application node through the routing layer that delivers applications to end user, we will begin this discussion with the linux kernel and work our way up through the application stack. For containers and OpenShift, security begins in the Linux kernel with SELinux

SELinux core concepts

SELinux is a linux kernel module, that is used to enforce mandatory access control - MAC. This is a set of access levels that are assigned to users by the system. Only users with root-level permissions privileges can alter them. For typical users including the automated user accounts in OpenShift that deploy applications, the SELinux config specified for a deployment is an immutable fact. MAC is a contrast to discretionary access control - in linux. `DAC` is the system of users and file ownership access modes that we all use every day on Linux hosts. If only `DAC` were tin effect in your OpenShift cluster users could allow full access to their container resources by changing the ownership or the access mode for the container process or storage resources. One of the key security features of OpenShift is that SELinux automatically enforces MAC policies that can not be changed by unprivileged users for pods and other resources even if they deployed the application

We need to take a few lines to discuss some of the fundamental information that we will use throughout the section. As with security in general this will not be a full SELinux introduction, Entire books have been written on that topic including the SELinux coloring book. But the following information, will help you understand how OpenShift uses SELinux to create a secure platform. We will focus on the following SELinux concepts

- Labels - SELinux labels are applied to all objects on a linux server
- Context - SELinux contexts apply labels to object based on file system location
- Policies - SELinux policies are rules that control interactions between objects with different SELinux labels

Working with SELinux labels

SELinux labels are applied to all objects on your OpenShift servers as they are created. An SELinux label dictates how an object on a linux sever interacts with the SELinux kernel module. We are defining an object in this context as anything a user or process can create or interact with on a server, such as the following.

- files
- directories
- TCP ports
- unix sockets
- shared memory resources

Each object in SELinux label has four section separated by colons:

- User which SELinux user has access to the object with the SELinux label,
- Role the SELinux role that can access the objects with the matching SELinux
- Type SELinux type for each label. This is the section where most common SELinux rules are written
- multi category security - often called the `MCS` bit, Unique for each container what we will spend the most time on

`Open vSwitch` for communication on your OpenShift cluster nodes. `/var/run/open-vswitch/db.sock`. To view this label run the following ls command using the -Z option flag to include the SELinux information in its output.

```
# besides the regular file permission, group and user owner, we can also see
   the labels for the sock object are also
# displayed
$ ls -alZ /var/run/openvswitch/db.sock
srwxr-x---. 1 openvswitch hugetlbfs system_u:object_r:openvswitch_var_run_t:
   s0 0 Jun 11 16:03 /var/run/openvswitch/db.sock

# The SELinux label for the open v-switch socket object, the format looks
   something like this, we can see from above,
# that we have multiple targets in the label
system_u:object_r:openvswitch_var_run_t:s0

system_u - the SELinux user is used for the ``MCS`` inmplementation
object_r - the SELinux role is used primarily for ``MCS`` implementation
openvswitch_var_run_t - the SELinux type is used in type enformcement
   policies to define interactions between objects on a Linux host
s0 - objects are assigned an ``MCS`` value to distinguish between different
   category levels on the Linux system.
```

In addition to the standard POSIX attributes of mode, owner and group ownership, the output also includes the SELinux label for `/var/run/openvswitch/db.sock`.. Next lets examine how SELinux labels are applied to files and other objects, when they are created.

```
# Most commands have the -Z option that will include the commands in SELinux
   labels common command line tools like
# ls,ps,netstat and others accept the -Z option, to include the SELinux
   information in their output, because objects are
# presented in the linux operating system as files their SELinux labels are
   stored in their filesystem extended
# attributes. You can view these attributes directly for the Open vSwitch
   socket using the following getfattr command:
```

```
$ getfattr -d -m - /var/run/openvswitch/db.sock
# file: var/run/openvswitch/db.sock
security.selinux="system_u:object_r:openvswitch_var_run_t:s0"

# if you are looking for a full SELinux documentation a great place to start
   is the Red Hat enterprise linux 7 SELinux
# guide, that you can find on the official site of RedHat
```

Applying labels with SELinux context

Labels are applied to files using SELinux contexts rules that are used to apply labels, to object on a linux system, contexts use regular expression to apply labels depending on where the object exists in the file system. One of the worst things a system admin can hear is a developer telling the that the SELinux breaks their application. In reality application is almost certainly creating objects on the linux server that do not have a defined SELinux context. If SELinux does not know how to apply the correct label it does not know how to treat the application objects. This often results in SELinux policy denials that lead to frantic calls and requests to disable SELinux because its breaking an application.

To query the context for a system use the `semanage` command, and filter it using grep. You can use `semanage`to search for context that apply to any label related to any file or directly, including the Openv`Switch`socket. A search for`openvswitch`in the`semanage`output shows that the context`system_u:obejct_r:poenvswitch_var_run_t:s0`, is applied to any object created in the `/var/run/openvswitch` directory

```
$ semanage fcontext -l | grep openvswitch
/etc/openvswitch(/.*)?                             all files
   system_u:object_r:openvswitch_rw_t:s0
/run/ovn(/.*)?                                     all files
   system_u:object_r:openvswitch_var_run_t:s0
/usr/bin/neutron-openvswitch-agent                 regular file
   system_u:object_r:neutron_exec_t:s0
/usr/bin/ovs-appctl                                regular file
   system_u:object_r:openvswitch_exec_t:s0
/usr/bin/ovs-vsctl                                 regular file
   system_u:object_r:openvswitch_exec_t:s0
/usr/bin/quantum-openvswitch-agent                 regular file
   system_u:object_r:neutron_exec_t:s0
/usr/lib/systemd/system/openvswitch.service        regular file
   system_u:object_r:openvswitch_unit_file_t:s0
/usr/sbin/ovs-vswitchd                             regular file
   system_u:object_r:openvswitch_exec_t:s0
/usr/sbin/ovsdb-ctl                                regular file
   system_u:object_r:openvswitch_exec_t:s0
/usr/sbin/ovsdb-server                             regular file
   system_u:object_r:openvswitch_exec_t:s0
/usr/share/openvswitch/scripts/ovs-ctl             regular file
   system_u:object_r:openvswitch_exec_t:s0
/usr/share/openvswitch/scripts/ovs-kmod-ctl        regular file
   system_u:object_r:openvswitch_load_module_exec_t:s0
/var/lib/openvswitch(/.*)?                         all files
   system_u:object_r:openvswitch_var_lib_t:s0
```

```
/var/log/openvswitch(/.*)?                          all files
   system_u:object_r:openvswitch_log_t:s0
/var/run/openvswitch(/.*)?                          all files
   system_u:object_r:openvswitch_var_run_t:s0
```

Properly applied SELinux labels create policies that control how objects with different labels can interact with each other. Let us discuss those next

Enforcing SELinux with policies

SELinux policies are a complex thing. They are heavily optimized and compiled so they can be interpreted quickly by the linux kernel. Creating one or looking at the code that creates one is outside the scope here, but lets look at the basic example of what an SELinux policy would do. For this we will use an example that most people are familiar with, the Apache web server. Apache is a common everywhere and has long established SELinux policies that we can use as an example

```
# if you have followed the install setup steps from above, you will be able
   to see/install the httpd binary in /sbin/httpd,
# which is the Apache web server binary, if not installed, use dnf -y install
    httpd on your master cluster node
$ which httpd
/sbin/httpd

# run the following ls command to inspect the SELinux policies on the
   executable, use the -Z flag, to do so
$ ls -zlZ /sbin/httpd
-rwxr-xr-x. 1 root root system_u:object_r:httpd_exec_t:s0 589576 Jan 29 17:56
    /sbin/httpd
```

The executable file for the Apache web server is `/usr/sbin/htpd`. This httpd executable has an SELinux labels of `system_u:object_r:httpd_exec_t:s0`. On CentOS and Red Hat systems the default Apache web content directory is `/var/www/html`. This directory has an SELinux label of `system_u:object_r:httpd_sys_content_t:s0`, The default `cgi-script` directory for Apache is `/var/www/cgi-bin`, and it has the SELinux label of `system_u:object_r:httpd_sys_script_exec_t:s0`. There is also the httpd_port_t label for the following TCP port numbers - 80, 8008, 8009, 8433, 9000, 81, 443, 488.an SELinux policy enforces the following rules using these labels for the `httpd_exec_t` object type

```
$ ls -alZ /var/www/cgi-bin
drwxr-xr-x. 2 root root system_u:object_r:httpd_sys_script_exec_t:s0  6 Jan
   29 17:56 .
drwxr-xr-x. 4 root root system_u:object_r:httpd_sys_content_t:s0     33 Jun
   15 12:40 ..

$ ls -alZ /var/www/html
drwxr-xr-x. 2 root root system_u:object_r:httpd_sys_content_t:s0  6 Jan 29
   17:56 .
drwxr-xr-x. 4 root root system_u:object_r:httpd_sys_content_t:s0 33 Jun 15
   12:40 ..
```

An SELinux policy enforces the following rules using these labels for the httpd_exec_t object type:

- `httpd_exec_t` - can write only to objects with an `httpd_sys_content_t` type
- `httpd_exec_t` - can execute scripts only with the `httpd_sys_script_exec_t`

- `httpd_exec_t` - can read from directories with `httpd_sys_script_exec_t`
- `httpd_exec_t` - can open and bind only to ports with the `httpd_port_t`

This means even if Apache is somehow compromised by a remote user it can read content from `/var/www/html` and run scripts from `/var/www/cgi-bin`. It also can not write to `/var/www/cgi-win`. All of this is enforced by the Linux kernel, regardless of the ownership or permission of the binary or these directories, and which user owns the httpd process. The default SELinux loaded on a Linux system is the targeted type.

The rules in the targeted SELinux type are applied only to objects that have matching context. Every object on a server is assigned a labels based on the SELinux context it matches. If an object does not match the context it is assigned an unconfined_t type in its SELinux labels. The unconfined_t type ha no contexts or policies associated with it. Interactions between objects that are not covered by a policy in targeted SELinux are allowed to run with no interference.

To summarize - the httpd executable can only find to specific ports as listed above, with the matching `httpd_port_t` type. Then the binary can only execute scripts with the `httpd_sys_script_exec_t` type, and also the binary can only read from directories with the `httpd_sys_script_exec_t`, and can server content from and write to directories tagged with the `httpd_sys_content_t` type.

For CentOS and Red Hat Enterprise Linux the default policies use type enforcement. Type enforcement uses the type value from SELinux labels to enforce the interaction between objects. Let us review what we have talked about so far up until this point

- SELinux is used to enforce the MAC in your OpenShift cluster. MAC provides access controls at the deeper level than a traditional user/group ownership and access mode. It also applies to objects on the operating system that are not traditional files and directories.
- Every object on an OpenShift node is assigned an SELinux label including a type.
- Labels are assigned according to a SELinux context as objects are created,
- With labels applied SELinux policies enforce interaction between objects. SELinux uses type enforcement policies on your OpenShift cluster to ensure proper interactions between objects

This SELinux configuration is standard for any CentOS or RedHat system running with SELinux in enforcing mode. Just as in the Apache web server process we have been running discussing you know what container is essentially a process. Each container process is assigned an SELinux label when its created and that label dictates the policies that affect the container. To confirm the SELinux label that is used for containers in OpenShift get the container PID, form the container runtime, and use the `ps` command with the -Z command parameter searching for that PID with grep

```
$ crictl ps | grep router-default | awk '{print $1'} | head -n1 | xargs -I'{}
   ' crictl inspect {} | grep "\"pid\":"
"pid": 7702,

$ ps -axZ | grep 7702
system_u:system_r:spc_t:s0         7702 ?        Ssl    2:03 /usr/bin/
   openshift-router --v=2
```

OpenShift hosts operate with SELinux enforcing mode. Enforcing mode means that the policy engine that controls how objects can interact is full activated, if an object attempts to do something that is against the SELinux policies present on the system, that action is not allowed, and the attempt is logged by the kernel, to confirm that the SELinux is in enforcing mode run the following command `getenforce` command

```
# you will get the following minimal output from this command, which
   basically confirms that we are indeed in enforcing
# mode running in the master node
```

```
$ getenforce
> Enforcing
```

In other servers, tools like virus scanners can cause issues with SELinux. A virus scanner is designed to analyze files on a server that are created and managed by other services. That makes writing an effective SELinux policy for a virus scanner a significant challenge, another typical issue is that when application and their data are placed in location on the file system that do not match their corresponding SELinux context. If the Apache web server is trying to access content from /data on server, it will be denied by SELinux because /data does not match any SELinux context associated with Apache. These sorts of issues lead to some people deciding to disable SELinux.

The user and role portions of the label are not used for type enforcement policies. The `svirt_lxc_net_t` type is used in SELinux policies that control which resources on the system container can interact with. We have not discussed the fourth part of the SELinux label - the `MCS` level, which isolates pods in OpenShift lets examine how that works next

Isolating pods with levels

The original purpose of the `MCS` bit was to implement the `MCS` security standards on linux servers. These standards control data access for different security levels on the same servers. For example secret and top secret data could exist on the same server. A top secret level process should be able to to access secret level data, a concept called data dominance. But secret processes should never be able to access the top secret level data, because that data has higher `MCS` level. This is the security feature you can use to prevent a pod from accessing data its not authorized to access on the host.

OpenShift uses the `MCS` level for each container process to enforce security as part of the pod security context. A pod security context is all the information that describes how its secured on its application node. Let us look at the security context for the app-cli pod

Investigating pod security context

Each pod security context contains information about its security posture. You can find full documentation on the possible fields that can be defined at the official documentation in OpenShift. In OpenShift the following parameters are configured by default.

- Capabilities - defines an application ability to perform various tasks on the host. Capabilities can be added to or dropped from each pod. We will look at these in depth in this section
- Privileged - specifies whether the container is running with any of the host namespaces
- RunAsUser - UID with which to run the container process. This can be configured which we will also checkout in this section, it is often used in Dockerfile images.
- SELinuxOptions - SELinux options for the pod, Normally the only needed option is to st the SELinux level.

You can view the security context for a pod in the GUI by choosing the Pods, and then selecting the pod you want the information about, and then choosing Actions -> Edit -> YAML. From the command line that might look like that

```
# make sure to specify the correct pod id, based on the output from the oc
   get pods, the output is abridged of course,
# but the detailed output for a pod would always include the `securityContext
   ` field, which you can see below, it contains
# the `seLinuxOptions` levels
$ oc get -n image-uploader -o yaml pod/app-cli-7976b4c888-rdbv7
```

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
  creationTimestamp: "<date>"
  labels:
    deployment: app-cli
    pod-template-hash: 7976b4c888
  name: app-cli-7976b4c888-rdbv7
  namespace: image-uploader
spec:
  securityContext:
    fsGroup: 1000650000
    seLinuxOptions:
      level: s0:c26,c0 # <- take a note the following c0 `MCS` level rule
    seccompProfile:
      type: RuntimeDefault
  serviceAccount: default
  serviceAccountName: default
```

Examining the MCS levels

The structure of the MCS level consists of sensitivity level s0 and two categories c8 and c7 as shown in the following output from the previous command. You may have noticed that the order of the categories is reversed in the oc output compared with the pc command. This makes no difference in how the Linux kernel reads and acts on the MCS level.

A detailed discussion of how different MCS levels can interact is out of scope. OpenShift assumes that application deployed in the same project will need to interact with each other. With that in the pods in a project have the same MCS level. Sharing an MCS level lets applications share resources easily and simplifies the security configuration you need to make for your cluster.

Let us examine the SELinux configuration for pod in different project. You already know the MCS level for app-cli. Because the app-cli and app-gui are in the same project, they should have the same MCS level. T get the MCS level of the app-gui pod use the same `oc` get command

```
# first make sure to extract the pods for the project, and then we will pull
   the manifest, grepping on the relevant
# field, to verify that the levels are indeed the same, in both pods in the
   same project, just as described above
$ oc get pods -n image-uploader
app-cli-7976b4c888-rdbv7   1/1     Running     0          3d20h
app-gui-5d5dc97869-8r2wl   1/1     Running     0          96s

$ oc get -o yaml pod/app-cli-7976b4c888-rdbv7 | grep "level:"
      level: s0:c26,c0

$ oc get -o yaml pod/app-gui-5d5dc97869-8r2wl | grep "level:"
      level: s0:c26,c0
```

This confirms what we have sated earlier the levels for the app-gui and app-cli are the same because they are deployed in the same project. Use the `wildfly-app` you deployed in the earlier chapter, to get the name of the deployed pod running run the following `oc` command, get the pods of that project and compare the security options of the SELinux labels

```
# we can also display the abridged output for the manifest for the `stateful-
   apps`, `wildfly-app` as well, we can see
# that here the levels are completely different
$ oc get pods -n stateful-apps
NAME                   READY   STATUS      RESTARTS   AGE
wildfly-app-1-snffs    1/1     Running     0          108s

# inspect the manifest file for the pod, and note that `securityContext`
   section, and see that the levels in this case are
# different for this project compared to the `image`-uploader
$ oc get -n stateful-apps -o yaml pod/wildfly-app-1-snffs
apiVersion: v1
kind: Pod
metadata:
  annotations:
    openshift.io/deployment-config.latest-version: "1"
    openshift.io/deployment-config.name: wildfly-app
    openshift.io/deployment.name: wildfly-app-1
    openshift.io/generated-by: OpenShiftNewApp
    openshift.io/scc: restricted-v2
    seccomp.security.alpha.kubernetes.io/pod: runtime/default
  creationTimestamp: "<date>"
  generateName: wildfly-app-1-
  labels:
    application: wildfly-app
    deployment: wildfly-app-1
    deploymentConfig: wildfly-app
    deploymentconfig: wildfly-app
  name: wildfly-app-1-snffs
spec:
  securityContext:
    fsGroup: 1000670000
    seLinuxOptions:
      level: s0:c26,c10 # <- take a note the following c10 `MCS` level rule
    seccompProfile:
      type: RuntimeDefault
  serviceAccount: default
  serviceAccountName: default
```

Each project uses a unique `MCS` level for deployed applications this `MCS` level permits each project applications to communicate only with resources in the same project. Let us continue looking at pod security context components with pod capabilities

Managing Linux capabilities

The capabilities listed in the app-cli security context are Linux capabilities that have been removed from the container process. Linux capabilities are permissions assigned to, or removed from processes by the Linux kernel:

```
securityContext:
    capabilities:
        drop:
        - KILL
        - MKNOD
        - STGID
        - SETUID
        - SYS-CHROOT
```

Capabilities allow a process to perform administrative task on the system. The root user on a Linux server can run commands with all Linux capabilities by default. That is why the root user can perform task like opening TCP ports below 1024 which is provided by the `CAP_NET_BIND_SERVICE` capability, and loading modules into the Linux kernel, which is provided by the `CAP_SYS_MODULE` capability.

You can add capabilities to a pod if it needs to be able to perform a specific type of task. Add them to the capabilities .add list in the pod's security context. To remove default capabilities from pods, add the capabilities you want to remove, add them to the drop list. This is the default action in OpenShift. The goal is to assign the fewest possible capabilities for a pod to fully function. This least privileged model ensures that pods can not perform tasks on the system that are not related, to their application proper function. The default value for the privileged option is False; setting the privileged option to True is the same as giving the pod the capabilities of the root user on the system. Although doing so should not be common practice privileged pods can exist and be useful under certain circumstances. A great example is the HAProxy pod we already talked about. It runs a s a privileged container so it can bind to port 80 on its node to handle incoming application requests. When an application needs access to host resources that can not be easily provided to the pod, running a privileged container may help.

The last value in the security context that we need to look at is what controls the user ID that the pod is run with - `runAsUser` parameter

Controlling the user ID

In OpenShift by default each project deploys pods using a random UID. Just like the `MCS` level the UID is common for all pods in a project to allow easier interactions between pods when needed. The UID for each pod is listed in the security context in the `runAsUser` parameter. By default OpenShift does not allow application to be deployed using UID 0, which is the default UID for the system's root user. There are not any known ways for UID 0 to break out of a container, but being UID 0 in a container means you must be incredibly careful about taking away capabilities and ensuring proper file ownership on the system. In an ounce of prevention that can prevent the need for a pound of a cure down the road

The components in a pod or a container security context are controller by the security context constraints `SCC` - assigned to the pod when its deployed. An `SCC` is a configuration applied to pods that outlines the security context components it will operate with. We will discuss the `SCC` in more depth in the next section when you deploy an application in your cluster that needs a more privileged security context than the default one. This application is a container image scanning utility that looks for security issues in container images in your OpenShift registry.

Scanning container images

OpenShift is only as secure as the containers it deploys. Even if your container images are built using proven vetted base images supplied by vendors or created using your own secure workflows, you will need a process to ensure that the image you are using do not have any security issues as they age in your cluster. The most straightforward solution for this challenge is to scan you container images. We are going to scan a single container image on demand in this section in a production environment image scanning should be an integral component in your application deployment workflow. Companies like Black Duck Software and `Twistlock` have image scanning and compliance tools that integrate with OpenShift. You must be able to trust what is running in your containers and quickly fix issues when they are found. An entire industry has sprung up in the past few years that provides container image scanning products to help make this an every day reality. These scanning utilities have the capabilities to annotate or tag images with metadata, that metadata, is then used to determine if the image is deemed a security risk or not, in the next section we will see how we can manually annotate images ourselves. The process is the same that an automated tool image scanner would take.

Annotating images with security information

OpenShift is configured with image policies that control which images are allowed to run on your cluster. The full documentation can be found at the official RedHat OpenShift documentation page. Annotations in the image metadata enforce image policies you can add these annotation manually. The deny-execution policy prevents an image from running on the cluster under any condition. To apply this policy to the image you can use the annotate command on the `oc` command line tool for OpenShift

```
# this will annotate the image with the sha-id with the deny execution
$ oc annotate image <image-sha-id> images.openshift.io/deny-execution=true
```

Image policies do not affect running pods, but they prevent an image with the deny execution annotation from being used for deployments. To see this in action, delete the active pods for the annotated image. Normally the replication controller for the deployment will automatically deploy a new version of the pod based on the correct base image. But no new pod will be deployed in this case, the replication controller will be stopped in its tracks , when it checks and sees that the image is annotated therefore it will not be allowed to execute the container runtime and start a new container for the target image. Looking at the events for the app project you can see that the image policies in OpenShift are reading the annotation that was added to the image and preventing a new pod from being deployed.

```
# see the events for the given namespace or project, you might notice
   something like that in the output of the events
$ oc events -n <namespace-project>

# the following output is abridged, but that is what one would expect from
   the events list for the namespaces / project
> Warning FailedCreate
> Error creating: Pod "" is invalid: spec.containers[0].image: Forbidden:
> this image is prohibited by policy
```

This process manually scans a container image and adds an annotation to it if security issues are found. The annotation is read by the OpenShift image-policy engine and prevents any new pods from being deployed using that image. Automated solutions like Black Duck and `Twistlock` handle this dynamically, including annotations about the security findings and information about the scan. These annotations can be used for security reporting and to 3ensure that the most secure application are deployed in OpenShift at all times. We started this section with SELinux and worked our way up to the security context that define how pods are assigned security permissions in OpenShift. As we said at the start of this section this is not a comprehensive list or a complete security workflow. Our goal has been to introduce you to what are the most important security concepts in OpenShift and give you enough information to begin to use and customize them as you gain experience with OpenShift