openshift-deep-dive

Contents

  - 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 a dev-environment

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 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 options

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
```

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.

```
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` 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 the `minishift` version along with an older version of `openshift` - 3.11 instead of the, OpenShift version 4.xx which is mostly supported by `crc`

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`

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

```
$ oc new-project image-uploader --display-name='Image Uploader Project'

Now using project "image-uploader" on server "https://192.168.42.124:8443".
You can add applications to this project with the 'new-app' command. For
   example, try:
    oc new-app centos/ruby-22-centos7~https://github.com/openshift/ruby-ex.
       git
to build a new example application in Ruby.
```

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

```
$ oc new-app --image-stream=php --code=https://github.com/OpenShiftInAction/
   image-uploader.git --name=app-cli

    Apache 2.4 with PHP 7.1
    -----------------------
    PHP 7.1 available as container is a base platform for building and
       running various PHP 7.1 applications and frameworks. PHP is an HTML-
       embedded scripting language. PHP attempts to make it easy for
       developers to write dynamically ge
nerated web pages. PHP also offers built-in database integration for several
   commercial and non-commercial database management systems, so writing a
   database-enabled webpage with PHP is fairly simple. The most common use of
    PHP coding
is probably as a replacement for CGI scripts.

    Tags: builder, php, php71, rh-php71

    * The source repository appears to match: php
    * A source build using source code from https://github.com/
       OpenShiftInAction/image-uploader.git will be created
      * The resulting image will be pushed to image stream "app-cli:latest"
      * Use 'start-build' to trigger a new build
    * This image will be deployed in deployment config "app-cli"
    * Ports 8080/tcp, 8443/tcp will be load balanced by service "app-cli"
      * Other containers can access this service through the hostname "app-
         cli"

--> Creating resources ...
    imagestream "app-cli" created
    buildconfig "app-cli" created
    deploymentconfig "app-cli" created
    service "app-cli" created
--> Success
    Build scheduled, use 'oc logs -f bc/app-cli' to track its progress.
    Application is not exposed. You can expose services to the outside world
       by executing one or more of the commands below:
     'oc expose svc/app-cli'
    Run 'oc status' to view your app.
```

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
Annotations:       openshift.io/generated-by=OpenShiftNewApp
Selector:          app=app-cli,deploymentconfig=app-cli
Type:              ClusterIP
IP:                172.30.51.80
Port:              8080-tcp  8080/TCP
TargetPort:        8080/TCP
Endpoints:         172.17.0.5:8080
Port:              8443-tcp  8443/TCP
TargetPort:        8443/TCP
Endpoints:         172.17.0.5:8443
```

```
Session Affinity:  None
Events:            <none>
```

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 "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:                19 seconds ago
Labels:                 app=app-cli
Annotations:            openshift.io/host.generated=true
Requested Host:         app-cli-image-uploader.192.168.42.124.nip.io
                          exposed on router router 19 seconds ago
Path:                   <none>
TLS Termination:        <none>
Insecure Policy:        <none>
Endpoint Port:          8080-tcp

Service:        app-cli
Weight:         100 (100%)
```

```
Endpoints:      172.17.0.5:8443, 172.17.0.5:8080
```

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 minishift cluster instance
$ minishift ssh
```

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 1074347
{
  "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 1074347
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 1074347 1000660000 httpd -D FOREGROUND
4026534706 ipc        13 1074347 1000660000 httpd -D FOREGROUND
4026535152 net        13 1074347 1000660000 httpd -D FOREGROUND
4026535628 mnt        13 1074347 1000660000 httpd -D FOREGROUND
4026535698 pid        13 1074347 1000660000 httpd -D FOREGROUND
4026535703 cgroup     13 1074347 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 `1074347` 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 shraed 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

```
$ sudo 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
vda4 252:4    0 30.5G  0 part /var/lib/kubelet/pods/22534047-9293-4e30-a3c2-6
   d1d1244b417/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
```

The LC device that the app-cli container uses for storage is recorded in the information from docker inspect. To get `LV` for your app-cli container run the following command, `docker inspect -f '{{ .GraphDriver.Data.DeviceName }}' <container-id>`

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 dc/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. 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.

```
# 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 beelow
$ 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 dc/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 rc`

```
$ oc describe rc/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=2025-05-22 17:07:31
   +0000 UTC
              openshift.io/deployer-pod.created-at=2025-05-22 17:07:29 +0000
                 UTC
              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>
Events:
  Type    Reason            Age   From                    Message
  ----    ------            ----  ----                    -------
  Normal  SuccessfulCreate  5m    replication-controller  Created pod: app-
     cli-1-vsk5q
  Normal  SuccessfulCreate  1m    replication-controller  Created pod: app-
     cli-1-gkg9h
```

`Note that more recent versions of openshift will by default use the ReplicationSet 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:

```
Labels: app=app-cli
        deployment=app-cli-1
        deploymentconfig=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 obect, 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. a 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 isues that can result from creating the raw YAML template by hand.

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 dc -l app=app-cli

Name:           app-cli
Namespace:      image-uploader
Created:        8 minutes ago
Labels:         app=app-cli
Annotations:    openshift.io/generated-by=OpenShiftNewApp
Latest Version: 1
Selector:       app=app-cli,deploymentconfig=app-cli
Replicas:       2
Triggers:       Config, Image(app-cli@latest, auto=true)
Strategy:       Rolling
Template:
Pod Template:
  Labels:       app=app-cli
                deploymentconfig=app-cli
  Annotations:  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>

Deployment #1 (latest):
        Name:           app-cli-1
        Created:        7 minutes ago
        Status:         Complete
        Replicas:       2 current / 2 desired
        Selector:       app=app-cli,deployment=app-cli-1,deploymentconfig=app
           -cli
        Labels:         app=app-cli,openshift.io/deployment-config.name=app-
           cli
        Pods Status:    2 Running / 0 Waiting / 0 Succeeded / 0 Failed

Events:
```

```
  Type          Reason                          Age     From
                              Message
  ----          ------                          ----    ----
                              -------
  Normal        DeploymentCreated               7m      deploymentconfig-
     controller     Created new replication controller "app-cli-1" for
     version 1
  Normal        ReplicationControllerScaled     3m      deploymentconfig-
     controller     Scaled replication controller "app-cli-1" from 1 to 2
```

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:

```
$ oc set probe dc/app-cli --readiness --get-url=http://:8080/notreal --
   initial-delay-seconds=5
deploymentconfig "app-cli" 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          9m
app-cli-1-gkg9h     1/1       Running     0          4m
app-cli-1-vsk5q     1/1       Running     0          8m
app-cli-2-deploy    1/1       Running     0          24s
app-cli-2-h65v4     0/1       Running     0          22s
```

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.

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 `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 this project,
   note that the admin password is none, it is
# not required for the local development version of the minishift instance
$ oc login https://192.168.42.124:8443 -u system:admin
$ oc project openshift-infra

Now using project "openshift-infra" on server "https://192.168.42.124:8443".
```

OpenShift provides an `Ansible` playbook called `openshift-metrics.yml` to install the OpenShift metrics stack. The playbook comes with reasonable default settings but can also be customized by passing environment variables, on the command line, Switch to the virtual machine/cluster that is running OpenShift on your host first and then run the following command

```
# login into the cluster virtual machine, first from there we can run the
   setup stage for the metrics stack, to install
$ minishift ssh
```

If all the tasks run properly the end of the output should show no errors, or failed tasks as in that would indicate that the stack was installed just fine. If the deployment fails, double check that the environment variables you passed to `openshift-metrics.yml` are accurate

After the playbook completes check from the command line that the metric stack is running. Similar to other features in OpenShift the stack is deployed as several different pods. You may have to wait a couple of minutes for the system to pull down the metrics container images and start. You can use the watch command to check the results of `oc get pods` every two seconds

```
# this will make sure to watch the output of the get pods command, which will
    terminate once the command returns
```

```
$ watch oc get pods
```

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 po 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 dataset. 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 dc/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:                                                   app-cli
Namespace:                                              image-uploader
Labels:                                                 <none>
```

```
Annotations:                                                  <none>
CreationTimestamp:                                            Thu, 22 May 2025
   20:39:15 +0300
Reference:                                                    DeploymentConfig/app-
   cli
Metrics:                                                      ( current / target )
  resource cpu on pods  (as a percentage of request):  <unknown> / 75%
Min replicas:                                                 2
Max replicas:                                                 5
Events:                                                       <none>
```

The description 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.

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 combines 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 dc/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

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 50000 -c 500
```

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 shorthanded 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 a dev-environment

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 aray of objects, that can be parameterized and pun up on demand. In most cases the API obEcts 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 https://raw.githubusercontent.com/OpenShiftInAction/chapter6/
   master/openshift-cicd-flask-mongo/OpenShift/templates/dev-todo-app-flask-
   mongo-gogs.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/dev/-toto-app-flash-mongo-gogs"
```

The pods will take a few minutes to deploy, First the `Gogs`, PostgreSQL and MongoDB pods are deployed. A separate pod called `install-gogs` also automates the installation of `Gogs` by initializing PostgreSQL and cloning the remote Git repository, locally. When `Gogs` is fully installed with a local copy of the remote Git repository the `install-gogs` pod configures a webhook an event drive HTTPS callback that you will use to automate new builds in OpenShift. `Gogs` will recognize the event and send HTTP POST to the OpenShift API telling the OpenShift to start a new source to image. Every time there is a new commit the app will be rebuilt, once the `install-gogs` pod finishes it tasks it exits.

Next open the OpenShift console in your browser and navigate to the app by choosing App - Routes - . You will see the app home page, verify that `Gogs` is running properly by navigating to the `gogs` route, if the `install-gogs` pod is not finished and marked completed you will see the database initialization instead. Once the pod has completed the installation process you will see the `Gogs` home page and will be ready to proceed.

Because the app is using Python the build process is very fast. For languages like Java that may take some time, to build using tools like Maven, you can enable incremental builds that allow the build pod to reuse build artifacts such as the Maven JAR and Maven POM that were imported during the build process. This avoids the redundancy of having to install the same dependencies multiple times for every build.

You now have a full development environment. By making some app code changes you can see the environment in action, and demonstrate many of the OpenShift automation features. Here you will edit the main landing page of the app.