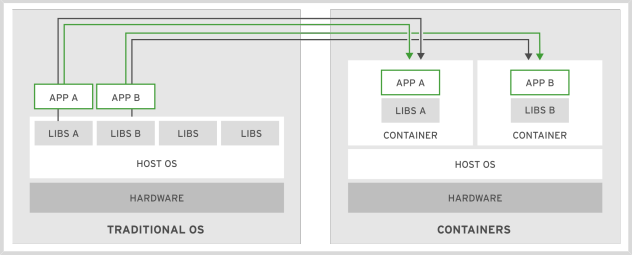
**Understanding Document OpenShift**

**Topics**

* Overview of Container Technology
  + Containerized Application
  + Advantage
* Overview of Container Architecture
  + Introducing Container History
  + Describing Linux Container Architecture
  + Managing Containers with podman
* Overview of kubernetes and OpenShift
  + Limitations of Containers
  + Kubernetes Overview
  + Kubernetes Features
  + OpenShift Overview
  + OpenShift Features
* Provisioning Containerized Services
  + Fetching Container Images with Podman
  + Running Containers
  + Using the Red Hat Container Catalog
  + Example
* Managing Containers
  + Managing the Life Cycle of Containers
  + Creating Containers
  + Running Commands in a Container
  + Managing Containers
  + Attaching Persistent Storage to Containers
* Accessing Containers
  + Introducing Networking with Containers
  + Mapping Network Ports
  + Example
* Managing Container Images.
  + Public Registries
  + Private Registries
  + Configuring Registries in Podman
  + Accessing Registries
    - Searching for Images in Registries.
    - Registry HTTP API
    - Registry Authentication
    - Pulling Images
    - Listing Local Copies of Images
    - Image Tags
  + Manipulating Container Images
    - Saving and Loading Images
    - Deleting Images
    - Deleting all Images
    - Modifying Images
    - Tagging Images
    - Removing Tags from Images
    - Publishing Images to a Registry
    - Example: Creating a Custom Apache Container Image
* Creating Custom Container Images
  + Reusing Existing Dockerfiles
  + Working with the Red Hat Software Collections Library
  + Finding Dockerfiles from the red Hat Software Collections Library
  + Container Images in Red Hat Container Catalog (RHCC)
  + Searching for Images Using Quay.io
  + Finding Dockerfiles on Docker Hub
  + Describing How to use the OpenShift Source-to-Image Tool
  + Building Custom Container Images with Dockerfile
    - Building Base Containers
      * Create a Working Directory
      * Write the Dockerfile Specification
    - CMD and ENTRYPOINT
    - ADD and COPY
    - Layering Image
    - Building Images with Podman
    - Example: Creating a Basic Apache Container Image
* Deploying Containerized Application on OpenShift
  + Describing Kubernetes and OpenShiftArchitecture
    - Kubernetes and OpenShift
      * Kubernetes Terminology
      * OpenShift Terminology
    - New Features in RHOCP 4
    - Describing Kubernetes Resource Types
      * Pods (po)
      * Services (svc)
      * Replication Controllers (rc)
      * Persistent Volumes (pv)
      * Persistent Volume Claims (pvc)
      * ConfigMaps (cm) and Secrets
    - OpenShift Resource Types
      * Deployment config (dc)
      * Build config (bc)
      * Routes
    - Networking
  + Creating Kubernetes Resources
    - The Red Hat OpenShift Container Platform (RHOCP) Command-line Tool
    - Describing Pod Resource Definition Syntax
    - Describing Service Resource Definition Syntax
    - Discovering Services
    - Creating New Applications
    - Managing OpenShift Resources at the Command Line
    - Labelling resources
    - Example: Deploying a Database Server on OpenShift
  + Creating Routes
    - Working with Routes
    - Creating Routes
    - Leveraging the Default Routing Service
    - Example: Exposing a Service as a Route
  + Creating Application with Source-to-Image
    - Describing Image Streams
    - Building an Application with S2I and the CLI
    - Relationship Between Build and Deployment Configurations
    - Example: Creating a Containerized Application with Source-to-Image
  + Creating Applications with the OpenShift Web Console
    - Accessing the OpenShift Web Console
    - Managing Projects
    - Navigating the Web Console
    - Creating New Applications
    - Managing Application Builds
    - Managing Deployed Applications
    - Other Web Console Features
    - Example: Creating an Application with the Web Console
* Deploying Multi-Container Application.
  + Considerations for Multi-Container Applications
    - Leveraging Multi-Container Applications
    - Discovering Services in a Multi-Container Application
    - Comparing Podman and Kubernetes
    - Describing the To Do List Application
    - Example: Deploying the Web Application and MySQL Containers
  + Deploying a Multi-Container Application on OpenShift
    - Examining the Skeleton of a Template
      * Parameters
    - Processing a Template Using the CLI
    - Configuring Persistent Storage for OpenShift Applications
      * Configuring Persistent Storage with Templates
    - Example: Creating an Application with a Template
* Troubleshooting S21 Builds and Deployments
  + Describing Common Problems
    - Troubleshooting Permission Issues
  + Troubleshooting Containerized Applications
    - Forwarding Ports for Troubleshooting
    - Enabling Remote Debugging with Port Forwarding
    - Accessing Container Logs
    - OpenShift Events
    - Overriding Container Binaries
    - Transferring Files To and Out of Containers

**Overview of Container Technology**

* **Containerized Application:**
  + Software applications typically depend on other libraries, configuration files, or services that are provided by the runtime environment. The traditional runtime environment for a software application is a physical host or virtual machine, and application dependencies are installed as part of the host.
  + For example, consider a Python application that requires access to a common shared library that implements the TLS protocol. Traditionally, a system administrator installs the required package that provides the shared library before installing the Python application.
  + The major drawback to traditionally deployed software applications is that the application's dependencies are entangled with the runtime environment. An application may break when any updates or patches are applied to the base operating system (OS)
  + For example, an OS update to the TLS shared library removes TLS 1.0 as a supported protocol.This breaks the deployed Python application because it is written to use the TLS 1.0 protocol for network requests. This forces the system administrator to roll back the OS update to keep the application running, preventing other applications from using the benefits of the updated package. Therefore, a company developing traditional software applications may require a full set of tests to guarantee that an OS update does not affect applications running on the host.
  + Furthermore, a traditionally deployed application must be stopped before updating the associated dependencies.
  + To minimize application downtime, organizations design and implement complex
  + systems to provide high availability of their applications. Maintaining multiple applications on a single host often becomes cumbersome, and any deployment or update has the potential to break one of the organization's applications.
  + Alternatively, a software application can be deployed using a container. A container is a set of one or more processes that are isolated from the rest of the system.
  + Containers provide many of the same benefits as virtual machines, such as security, storage, and network isolation. Containers require far fewer hardware resources and are quick to start and terminate. They also isolate the libraries and the runtime resources (such as CPU and storage) for an application to minimize the impact of any OS update to the host OS.
  + The use of containers not only helps with the efficiency, elasticity, and reusability of the hosted applications, but also with application portability.
  + The Open Container Initiative provides a
  + set of industry standards that define a container runtime specification and a container image specification. The image specification defines the format for the bundle of files and metadata that form a container image. When you build an application as a container image, which complies with the OCI standard, you can use any OCI-compliant container engine to execute the application.
  + There are many container engines available to manage and execute individual containers, including Rocket, Drawbridge, LXC, Docker, and Podman. Podman is available in Red Hat Enterprise Linux 7.6 and later,



* **Advantage**
  + Low hardware footprint
  + Environment isolation
  + Quick deployment
  + Multiple environment deployment
  + Reusability
  + Often, a software application with all of its dependent services (databases, messaging, file systems) are made to run in a single container. This can lead to the same problems associated with traditional software deployments to virtual machines or physical hosts. In these instances, a multi container deployment may be more suitable.
  + Furthermore, containers are an ideal approach when using microservices for application development. Each service is encapsulated in a lightweight and reliable container environment that can be deployed to a production or development environment. The collection of containerized services required by an application can be hosted on a single machine, removing the need to manage a machine for each service.
  + In contrast, many applications are not well suited for a containerized environment. For example, applications accessing low-level hardware information, such as memory, file systems, and devices may be unreliable due to container limitations

**Overview of Container Architecture**

* **Introducing Container History**
  + Containers have quickly gained popularity in recent years. However, the technology behind containers has been around for a relatively long time. In 2001, Linux introduced a project named VServer. VServer was the first attempt at running complete sets of processes inside a single server with a high degree of isolation
  + From VServer, the idea of isolated processes further evolved and became formalized around the following features of the Linux kernel:
    - Namespaces: The kernel can isolate specific system resources, usually visible to all processes, by placing the resources within a namespace. Inside a namespace, only processes that are members of that namespace can see those resources. Namespaces can include resources like network interfaces, the process ID list, mount points, IPC resources, and the system's hostname information.
    - Control groups(cgroups): Control groups partition sets of processes and their children into groups to manage and limit the resources they consume.Control groups place restrictions on the amount of system

resources processes might use. Those restrictions keep one process from using too many resources on the host.

* + - Seccomp: Developed in 2005 and introduced to containers circa 2014, Seccomp limits how processes could use system calls. Seccomp defines a security profile for processes, whitelisting the system calls, parameters and file descriptors they are allowed to use.
    - SELinux: SELinux (Security-Enhanced Linux) is a mandatory access control system for processes. Linux kernel uses SELinux to protect processes from each other and to protect the host system from its running processes. Processes run as a confined SELinux type that has limited access to host system resources.
  + All of the above innovations and features focus around a basic concept: enabling processes to run isolated while still accessing system resources.
  + This concept is the foundation of container technology and the basis for all container implementations.
  + Nowadays, containers are processes in Linux kernel making use of those security features to create an isolated environment. This environment forbids isolated processes from misusing system or other container resources.
  + A common use case of containers is having several replicas of the same service (for example, a database server) in the same host. Each replica has isolated resources (file system, ports, memory), so there is no need for the service to handle resource sharing. Isolation guarantees that a malfunctioning or harmful service does not impact other services or containers in the same host,nor in the underlying system.
* **Describing Linux Container Architecture**
  + From the Linux kernel perspective, a container is a process with restrictions. However, instead of running a single binary file, a container runs an image.
  + An image is a file-system bundle that contains all dependencies required to execute a process: files in the file system, installed packages, available resources, running processes, and kernel modules.
  + Like executable files are the foundation for running processes, images are the foundation for running containers. Running containers use an immutable view of the image, allowing multiple containers to reuse the same image simultaneously. As images are files, they can be managed by versioning systems, improving automation on container and image provisioning.
  + Container images need to be locally available for the container runtime to execute them, but the images are usually stored and maintained in an image repository.
  + An image repository is just a service - public or private - where images can be stored, searched and retrieved. Other features provided by image repositories are remote access, image metadata, authorization or image version control.
  + There are many different image repositories available, each one offering different features:
    - Red Hat Container Catalog [<https://registry.redhat.io>]
    - Docker Hub [<https://hub.docker.com>]
    - Red Hat Quay [<https://quay.io/>]
    - Google Container Registry [<https://cloud.google.com/container-registry/>]
    - Amazon Elastic Container Registry [<https://aws.amazon.com/ecr/>]
* **Managing Containers with podman**
  + Containers, images, and image registries need to be able to interact with each other.
  + For example, you need to be able to build images and put them into image registries. You also need to be able to retrieve an image from the image registry and build a container from that image.
  + Podman is an open source tool for managing containers and container images and interacting with image registries. It offers the following key features:
    - It uses image format specified by the Open Container Initiative [https://

www.opencontainers.org] (OCI). Those specifications define a standard, community-driven, non-proprietary image format.

* + - Podman stores local images in the local file-system. Doing so avoids unnecessary client/server architecture or having daemons running on local machines.
    - Podman follows the same command patterns as the Docker CLI, so there is no need to learn a new toolset.
    - Podman is compatible with Kubernetes. Kubernetes can use Podman to manage its containers.
    - Currently, Podman is only available on Linux systems. To install Podman in Red Hat Enterprise Linux, Fedora or similar RPM-based systems, run sudo yum install podman or sudo dnf install podman.

**Overview of kubernetes and OpenShift**

* **Limitations of Containers**
  + Containers provide an easy way to package and run services. As the number of containers managed by an organization grows, the work of manually starting them rises exponentially along with the need to quickly respond to external demands
  + When using containers in a production environment, enterprises often require:
    - Easy communication between a large number of services.
    - Resource limits on applications regardless of the number of containers running them.
    - Respond to application usage spikes to increase or decrease running containers.
    - React to service deterioration.
    - Gradually roll out a new release to a set of users.
* Enterprises often require a container orchestration technology because container runtimes (such as Podman) do not adequately address the above requirements.
* **Kubernetes Overview**
  + Kubernetes is an orchestration service that simplifies the deployment, management, and scaling of containerized applications.
  + The smallest unit manageable in Kubernetes is a pod. A pod consists of one or more containers with its storage resources and IP address that represent a single application. Kubernetes also uses pods to orchestrate the containers inside it and to limit its resources as a single unit.
* **Kubernetes Features**
  + Kubernetes offers the following features on top of a container infrastructure:
  + **Service discovery and load balancing**
    - Kubernetes enables inter-service communication by assigning a single DNS entry to each set of containers. This way, the requesting service only needs to know the target's DNS name, allowing the cluster to change the container's location and IP address, leaving the service unaffected.
    - This permits load-balancing the request across the pool of containers providing the service. For example, Kubernetes can evenly split incoming requests to a MySQL service taking into account the availability of the pods.
  + **Horizontal scaling**
    - Applications can scale up and down manually or automatically with configuration set either with the Kubernetes command-line interface or the web UI.
  + **Self-healing**
    - Kubernetes can use user-defined health checks to monitor containers to restart and reschedule them in case of failure.
  + **Automated rollout**
    - Kubernetes can gradually roll updates out to your application's containers while checking their status. If something goes wrong during the rollout, Kubernetes can roll back to the previous iteration of the deployment.
  + **Secrets and configuration management**
    - You can manage configuration settings and secrets of your applications without rebuilding containers. Application secrets can be user names, passwords, and service endpoints; any configuration settings that need to be kept private.
  + **Operators**
    - Operators are packaged Kubernetes applications that also bring the knowledge of the application's life cycle into the Kubernetes cluster. Applications packaged as Operators use the Kubernetes API to update the cluster's state reacting to changes in the application state.
* **OpenShift Overview**
  + Red Hat OpenShift Container Platform (RHOCP) is a set of modular components and services built on top of a Kubernetes container infrastructure. RHOCP adds the capabilities to provide a production PaaS platform such as remote management, multitenancy, increased security, monitoring and auditing, application life-cycle management, and self-service interfaces for

developers

* + Beginning with Red Hat OpenShift v4, hosts in an OpenShift cluster all use Red Hat Enterprise Linux CoreOS as the underlying operating system.
* **OpenShift Features**
  + OpenShift adds the following features to a Kubernetes cluster:
  + **Integrated developer workflow**
    - RHOCP integrates a built-in container registry, CI/CD pipelines, and S2I; a tool to build artifacts from source repositories to container images.
  + **Routes**
    - Easily expose services to the outside world.
  + **Metrics and logging**
    - Include built-in and self-analyzing metrics service and aggregated logging.
  + **Unified UI**
    - OpenShift brings unified tools and a UI to manage all the different capabilities.

**Provisioning Containerized Services**

* **Fetching Container Images with Podman**
  + Running a containerized application, that is, running an application inside a container, requires a container image, a file system bundle providing all application files, libraries, and dependencies the application needs to run.
  + Container images can be found in image registries: services that allow users to search and retrieve container images.
  + Podman users can use the search subcommand to find available images from remote or local registries:

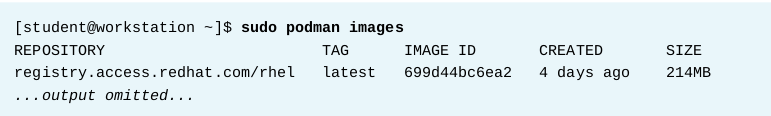
$ sudo podman search rhel

* + After you have found an image, you can use Podman to download it. When using the pull subcommand, Podman fetches the image and saves it locally for future use:

$ sudo podman pull rhel

* + Container images are named based on the following syntax:registry\_name/user\_name/image\_name:tag
    - First registry\_name, the name of the registry storing the image. It is usually the FQDN of the registry.
    - user\_name stands for the user or organization the image belongs to.
    - The image\_name should be unique in the user namespace.
    - The tag identifies the image version. If the image name includes no image tag, the latest is assumed.
    - After retrieval, Podman stores images locally and you can list them with the images subcommand:

$ sudo podman images



* **Running Containers**
  + The podman run command runs a container locally based on an image. At a minimum, the command requires the name of the image to execute in the container
  + The container image specifies a process that starts inside the container known as the entry point. The podman run command uses all parameters after the image name as the entry point command for the container. The following example starts a container from a Red Hat Enterprise Linux image. It sets the entry point for this container to the echo "Hello world" command.

$ sudo podman run ubi7/ubi:7.7 echo 'Hello!'

* + To start a container image as a background process, pass the -d option to the podman run.
  + Running a containerized Apache HTTP server in the background.

$ sudo podman run -d rhscl/httpd-24-rhel7:2.4-36.8

* + Podman inspect command to retrieve the container's internal IP address

from container metadata

* + $ sudo podman inspect -l \

> -f "{{.NetworkSettings.IPAddress}}"

* + Finally, uses the IP address to fetch the root page from Apache HTTP

server.

$ curl <http://10.88.0.68:8080>

* + This response proves the container is still up and running after the podman run command.



* + Most Podman subcommands accept the -l flag (l for latest) as a replacement for

the container id. This flag applies the command to the latest used container in any Podman command.

* + If the image to be executed is not available locally when using the podman run

command, Podman automatically uses pull to download the image.

* + When referencing the container, Podman recognizes a container either with the container name or the generated container id.
  + Use the --name option to set the container name when running the

container with Podman. Container names must be unique. If the podman run command includes no container name, Podman generates a unique random name.

* + If the images require interacting with the user with console input, Podman can redirect container input and output streams to the console. The run subcommand requires the -t and -i flags (or, in short, -it flag) to enable interactivity.
  + -t is equivalent to --tty, meaning a pseudo-tty (pseudo-terminal) is to be

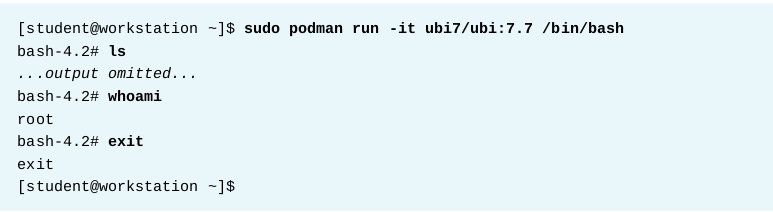
allocated for the container.

* + -i is the same as --interactive. When used, standard input is kept open into

the container.

* + -d, or its long form --detach, means the container runs in the background

(detached). Podman then prints the container id.



* + Some containers need or can use external parameters provided at startup.
  + The most common approach for providing and consuming those parameters is through environment variables.
  + Podman can inject environment variables into containers at startup by adding the -e flag to the run subcommand

$ sudo podman run -e GREET=Hello -e NAME=RedHat \

> rhel7:7.5 printenv GREET NAME

Hello

RedHat

[student@workstation ~]$

or

$ sudo podman run --name mysql-custom \

> -e MYSQL\_USER=redhat -e MYSQL\_PASSWORD=r3dh4t \

> -d rhmap47/mysql:5.5

* **Using the Red Hat Container Catalog**
  + Red Hat maintains its repository of finely tuned container images. Using this repository provides customers with a layer of protection and reliability against known vulnerabilities, which could potentially be caused by untested images.
  + The standard podman command is compatible with the Red Hat Container Catalog. The Red Hat Container Catalog provides a user-friendly interface for

searching and exploring container images from the Red Hat repository.

* + The Container Catalog also serves as a single interface, providing access to different aspects of all the available container images in the repository. It is useful in determining the best image among multiple versions of container images given health index grades. The health index grade indicates how current an image is, and whether it contains the latest security updates.
  + The Container Catalog also gives access to the errata documentation of an image. It describes the latest bug fixes and enhancements in each update. It also suggests the best technique for pulling an image on each operating system.
  + Link : https://catalog.redhat.com/
* **Example:**
  + Start a container from the Red Hat Software Collections Library MySQL image.

$ sudo podman run --name mysql-basic \

> -e MYSQL\_USER=user1 -e MYSQL\_PASSWORD=mypa55 \

> -e MYSQL\_DATABASE=items -e MYSQL\_ROOT\_PASSWORD=r00tpa55 \

> -d rhscl/mysql-57-rhel7:5.7-3.14

* + - This command downloads the MySQL container image with the 5.7-3.14 tag, and then starts a container-based image. It creates a database named items, owned by a user named user1 with mypa55 as the password. The database administrator password is set to r00tpa55 and the container runs in the background.
  + Verify that the container started without errors

$ sudo podman ps --format "{{.ID}} {{.Image}} {{.Names}}"

* + Access the container sandbox by running the following command

$ sudo podman exec -it mysql-basic /bin/bash

bash-4.2$

* + Show the database

bash-4.2$ mysql -uroot

mysql> show databases;

* **Example:**
  + Start a container named httpd-basic in the background, and forward port 8080 to port 80 in the container. Use the quay.io/redhattraining/httpd-parent container image with the 2.4 tag.

$ sudo podman run -d -p 8080:80 \

> --name httpd-basic quay.io/redhattraining/httpd-parent:2.4

* + Test the httpd-basic container.

$ curl <http://localhost:8080>

* + Customize the httpd-basic container to display Hello World as the message. The

container's message is stored in the file /var/www/html/index.html.

* + - Start a Bash session inside the container.

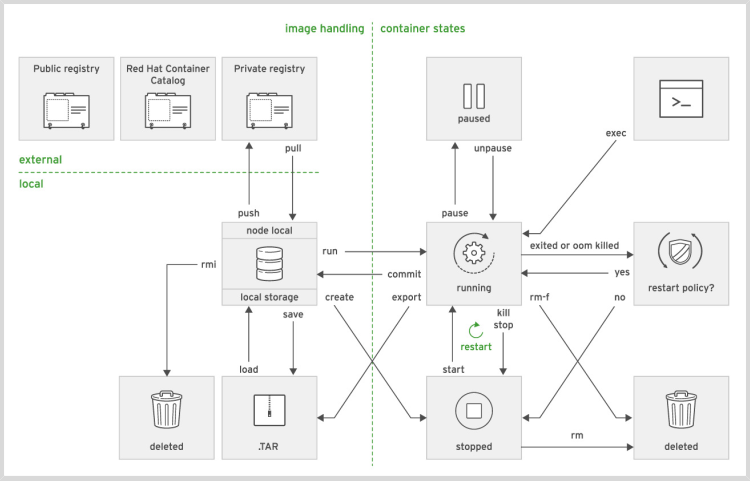
$ sudo podman exec -it httpd-basic /bin/bash

* + - Change the index.html file to contain the text Hello World, replacing all of the existing content.

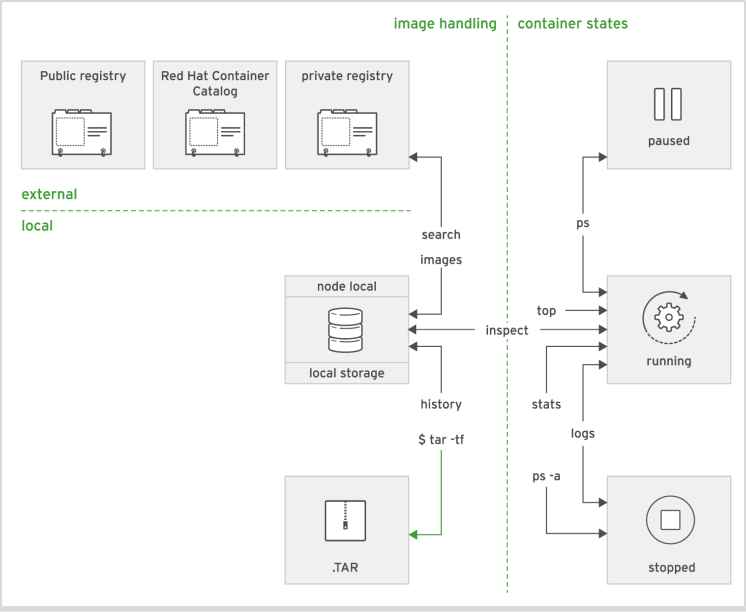
bash-4.4# echo "Hello World" > /var/www/html/index.html

**Managing Containers**

* **Managing the Life Cycle of Containers**
  + Podman allows you not only to run containers, but also to make them run in the background, execute new processes inside them, and provide them with resources such as file system volumes or a network.
  + Podman, implemented by the podman command, provides a set of subcommands to create and manage containers. Developers use those subcommands to manage container and container images life cycle.



* + Podman also provides a set of useful subcommands to obtain information about running and stopped containers.



* **Creating Containers**
  + The podman run command creates a new container from an image and starts a process inside the new container.
  + If the container image is not available locally, this command attempts to

download the image using the configured image repository.

$ sudo podman run rhscl/httpd-24-rhel7

* + The container was started with a non-interactive process (without the -it option) and is running in the foreground because it was not started with the -d option.
  + Stopping the resulting process with Ctrl+C (SIGINT) therefore stops both the container process as well as the container itself.
  + Podman identifies containers by a unique container ID or container name. The podman ps command displays the container ID and names for all actively running containers:

$ sudo podman ps

* + - The container ID is unique and generated automatically.
    - The container name can be manually specified, otherwise it is generated automatically. This name must be unique or the run command fails.
  + The podman run command automatically generates a unique, random ID. It also generates a random container name. To define the container name explicitly, use the --name option when running a container

$ sudo podman run --name my-httpd-container

rhscl/httpd-24-rhel7

* + - The name must be unique. Podman throws an error if the name is already in use,including stopped containers.
  + Another important feature is the ability to run the container as a daemon process in the background. The -d option is responsible for running in detached mode. When using this option, Podman returns the container ID on the screen, allowing you to continue to run commands in the same terminal while the container runs in the background.

$ sudo podman run --name my-httpd-container -d

rhscl/httpd-24-rhel7

* + The container image specifies the command to run to start the containerized process, known as the entry point. The podman run command can override this entry point by including the command after the container image.

$ sudo podman run rhscl/httpd-24-rhel7 ls /tmp

anaconda-post.log

* + - The specified command must be executable inside the container image.
  + Some containers need to run as an interactive shell or process. This includes containers running processes that need user input (such as entering commands), and processes that generate output through standard output.

$ sudo podman run -it rhscl/httpd-24-rhel7 /bin/bash

* + - The -t and -i options enable terminal redirection for interactive text-based programs. The -t option allocates a pseudo-tty (a terminal) and attaches it to the standard input of the container.
    - The -i option keeps the container's standard input open, even if it was detached, so the main process can continue waiting for input.
* **Running Commands in a Container**
  + When a container starts, it executes the entry point command. However, it may be necessary to execute other commands to manage the running container.
    - Executing an interactive shell in an already running container.
    - Running processes that update or display the container's files.
    - Starting new background processes inside the container.
  + The podman exec command starts an additional process inside an already running container

$ sudo podman exec 7ed6e671a600 cat /etc/hostname

* + - Uses the container ID to execute the command.
  + Podman remembers the last container used in any command. Developers can skip writing this container's ID or name in later Podman commands by replacing the container id by the -l option

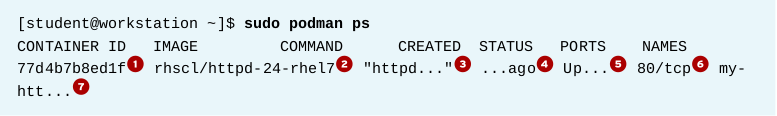
$ sudo podman exec my-httpd-container cat /etc/hostname

>> 7ed6e671a600

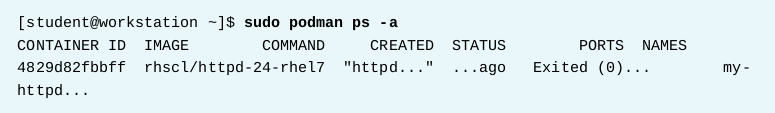
$ sudo podman exec -l cat /etc/hostname

>> 7ed6e671a600

* **Managing Containers**
  + Creating and starting a container is just the first step of the container's life cycle. A container's life cycle also includes stopping, restarting, or finally removing it. Users can also examine the container status and metadata for debugging, updating, or reporting purposes.
  + Podman provides the following commands for managing containers.
  + **podman ps**



* + - 1 Each container, when created, gets a container ID, which is a hexadecimal number. This ID looks like an image ID but is unrelated.
    - 2 Container image that was used to start the container.
    - 3 Command executed when the container started.
    - 4 Date and time the container was started.
    - 5 Total container uptime, if still running, or time since terminated.
    - 6 Ports that were exposed by the container or any port forwarding that might be configured.
    - 7 The container name.
    - Podman does not discard stopped containers immediately. Podman preserves their local file systems and other states for facilitating postmortem analysis. Option -a lists all containers, including stopped ones



* + While creating containers, Podman aborts if the container name is already in

use, even if the container is in a “stopped” status. This option can help to avoid

duplicated container names.

* + **podman inspect:** This command lists metadata about a running or stopped container. The command produces JSON output:

$ sudo podman inspect my-httpd-container

* + - This command allows formatting of the output string using the given Go template with the -f option. For example, to retrieve only the IP address, use the following command.

$ sudo podman inspect \

> -f '{{ .NetworkSettings.IPAddress }}' my-httpd-container

>> 172.17.0.9

* + **podman stop:** This command stops a running container gracefully.

$ sudo podman stop my-httpd-container

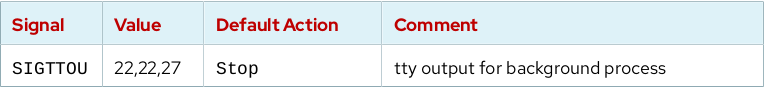
* + - Using podman stop is easier than finding the container start process on the host OS and killing it
  + **podman kill:** This command sends Unix signals to the main process in the container. If no signal is specified, it sends the SIGKILL signal, terminating the main process and the container.

$ sudo podman kill my-httpd-container

* + - You can specify the signal with the -s option

$ sudo podman kill -s SIGKILL my-httpd-container





* + - Term: Terminate the process.
    - Core: Terminate the process and generate a core dump.
    - Ign: Signal is ignored.
    - Stop: Stop the process.
  + **podman restart:** This command restarts a stopped container.

$ sudo podman restart my-httpd-container

* + - The podman restart command creates a new container with the same container ID, reusing the stopped container state and file system.
  + **podman rm:** This command deletes a container and discards its state and file system.

$ sudo podman rm my-httpd-container

* + - The -f option of the rm subcommand instructs Podman to remove the container even if not stopped. This option terminates the container forcefully and then removes it. Using -f option is equivalent to podman kill and podman rm commands together
    - You can delete all containers at the same time. Many podman subcommands accept the -a option. This option indicates using the subcommand on all available containers or images. The following example removes all containers

$ sudo podman rm -a

* + - Before deleting all containers, all running containers must be in a “stopped” status. You can use the following command to stop all containers:

$ sudo podman stop -a

* + The inspect, stop, kill, restart, and rm subcommands can use the container

ID instead of the container name.

* + **Example:**
    - Download the MySQL database container image, and attempt to start it. The container does not start because several environment variables must be provided to the image.

$sudo podman run --name mysql-db rhscl/mysql-57-rhel7

>> You must either specify the following environment variables:

MYSQL\_USER (regex: '^[a-zA-Z0-9\_]+$')

MYSQL\_PASSWORD (regex: '^[a-zA-Z0-9\_~!@#$%^&\*()-=<>,.?;:|]+$')

* + - * If you try to run the container as a daemon (-d), the error message about the required variables is not displayed. However, this message is included as part of the container logs, which can be viewed using the following command

$ sudo podman logs mysql-db

* + - Create a new container named mysql, and specify each required variable using the -e parameter.
    - $ sudo podman run --name mysql \

> -d -e MYSQL\_USER=user1 -e MYSQL\_PASSWORD=mypa55 \

> -e MYSQL\_DATABASE=items MYSQL\_ROOT\_PASSWORD=r00tpa55\

> rhscl/mysql-57-rhel7

* + Verify that the mysql container started correctly. Run the following command

$ sudo podman ps --format="{{.ID}} {{.Names}} {{.Status}}"

* + Inspect the container metadata to obtain the IP address for the MySQL.

$ sudo podman inspect \

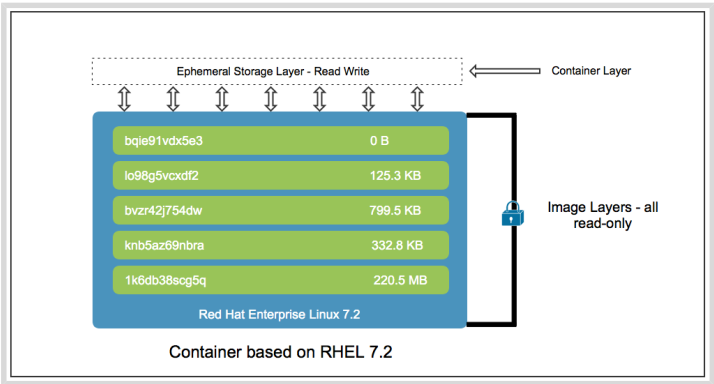
> -f '{{ .NetworkSettings.IPAddress }}' mysql

* + The podman inspect command provides additional important information.

For example, the Env section displays environment variables, such as the

MYSQL\_ROOT\_PASSWORD variable.

* **Attaching Persistent Storage to Containers**
  + Container storage is said to be ephemeral, meaning its contents are not preserved after the container is removed. Containerized applications work on the assumption that they always start with empty storage, and this makes creating and destroying containers relatively inexpensive operations.
  + Previously in this course, container images were characterized as immutable and layered, meaning that they are never changed, but rather composed of layers that add or override the contents of layers below.
  + A running container gets a new layer over its base container image, and this layer is the container storage. At first, this layer is the only read/write storage available for the container, and it is used to create working files, temporary files, and log files. Those files are considered volatile. An application does not stop working if they are lost. The container storage layer is exclusive to the running container, so if another container is created from the same base image, it gets another read/write layer. This ensures that each container's resources are isolated from other similar containers.
  + Ephemeral container storage is not sufficient for applications that need to keep data over restarts, such as databases. To support such applications, the administrator must provide a container with persistent storage.



* + Containerized applications should not try to use the container storage to store persistent data, because they cannot control how long its contents will be preserved. Even if it were possible to keep container storage indefinitely, the layered file system does not perform well for intensive I/O workloads and would not be adequate for most applications requiring persistent storage.
  + **Reclaiming Storage**
    - Podman keeps old stopped container storage available to be used by troubleshooting operations, such as reviewing failed container logs for error messages.
    - If the administrator needs to reclaim old container storage, the container can then be deleted using podman rm container\_id. This command also deletes the container storage. The stopped container IDs can be found using podman ps -a command.
  + **Preparing the Host Directory**
    - Podman can mount host directories inside a running container. The containerized application sees these host directories as part of the container storage, much like regular applications see a remote network volume as if it were part of the host file system. But these host directories' contents are not reclaimed after the container is stopped, and they can be mounted to new containers whenever needed.
    - For example, a database container can use a host directory to store database files. If this database container fails, Podman can create a new container using the same host directory, keeping the database data available to client applications. To the database container, it does not matter where this host directory is stored from the host point of view; it could be anything from a local hard disk partition to a remote networked file system.
    - A container runs as a host operating system process, under a host operating system user and group ID, so the host directory needs to be configured with ownership and permissions allowing access to the container.
    - In RHEL, the host directory also needs to be configured with the appropriate SELinux context, which is container\_file\_t.
    - Podman uses the container\_file\_t SELinux context to restrict which files of the host system the container is allowed to access. This avoids information leakage between the host system and the applications running inside containers.
  + **Steps to setup host directory**
    - Create a directory with owner and group root

$ sudo mkdir /var/dbfiles

* + - The user running processes in the container must be capable of writing files to the directory.If the host machine does not have exactly the same user defined, the permission should be defined with the numeric user ID (UID) from the container. In the case of the Red Hat-provided MySQL service, the UID is 27:

$ sudo chown -R 27:27 /var/dbfiles

* + - Apply the container\_file\_t context to the directory (and all subdirectories) to allow containers access to all of its contents.

$ sudo semanage fcontext -a -t container\_file\_t

'/var/dbfiles(/.\*)?'

* + - Apply the SELinux container policy that you set up in the first step to the newly created directory:

$ sudo restorecon -Rv /var/dbfiles

* + **Mounting a Volume**
    - After creating and configuring the host directory, the next step is to mount this directory to a container. To bind mount a host directory to a container, add the -v option to the podman run command, specifying the host directory path and the container storage path, separated by a colon

(:).

* + - For example, to use the /var/dbfiles host directory for MySQL server database files, which are expected to be under /var/lib/mysql inside a MySQL container image named mysql, use the following command:

$ sudo podman run -v /var/dbfiles:/var/lib/mysql

rhmap47/mysql

* + **Example:**
    - Create the /var/local/mysql directory with the correct SELinux context and

permissions.

$ sudo mkdir -pv /var/local/mysql

* + - Add the appropriate SELinux context for the /var/local/mysql directory and its contents.

$ sudo semanage fcontext -a \

> -t container\_file\_t '/var/local/mysql(/.\*)?'

* + - Apply the SELinux policy to the newly created directory.

Command: sudo restorecon -R /var/local/mysql

* + - Verify that the SELinux context type for the /var/local/mysql directory is

container\_file\_t

$ ls -ldZ /var/local/mysql

* + - Change the owner of the /var/local/mysql directory to the mysql user and
    - mysql group.

$ sudo chown -Rv 27:27 /var/local/mysql

* + - The user running processes in the container must be capable of writing files to the directory. If the host machine does not have exactly the same user defined, the permission should be defined with the numeric user ID (UID) from the container. For the MySQL service provided by Red Hat, the UID is 27.
    - Create a MySQL container instance with persistent storage.
      * Pull the MySQL container image.

$ sudo podman pull rhscl/mysql-57-rhel7

* + - * Create a new container specifying the mount point to store the MySQL database data

$ sudo podman run --name persist-db \

> -d -v /var/local/mysql:/var/lib/mysql/data \

> -e MYSQL\_USER=user1 -e MYSQL\_PASSWORD=mypa55 \

> -e MYSQL\_DATABASE=items -e MYSQL\_ROOT\_PASSWORD=r00tpa55 \

> rhscl/mysql-57-rhel7

* + - * Verify that the container started correct

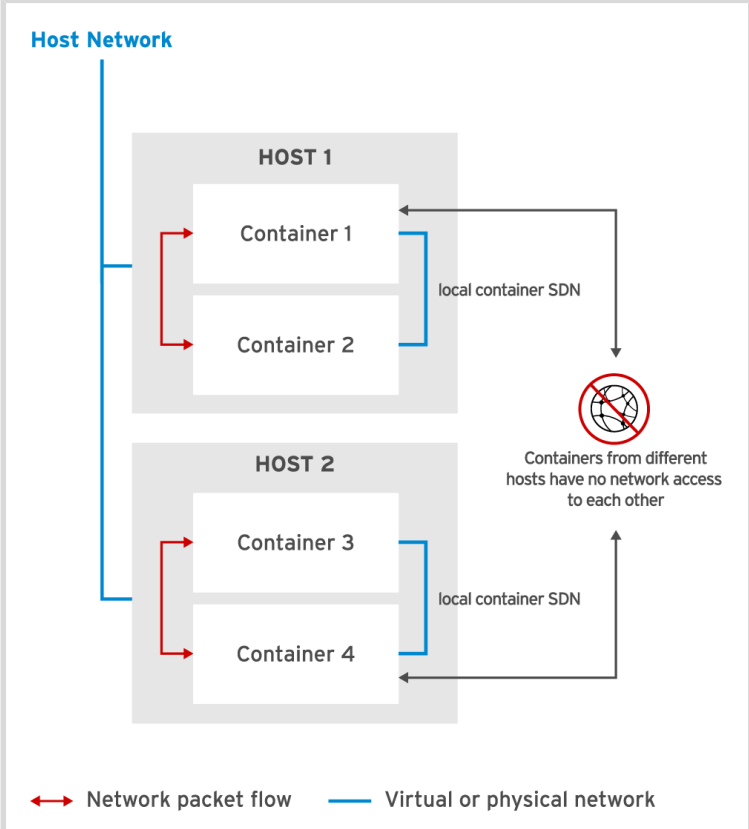
$ sudo podman ps --format="{{.ID}} {{.Names}} {{.Status}}"

* + - Verify that the /var/local/mysql directory contains the items directory.

$ ls -ld /var/local/mysql/items

**Accessing Containers**

* **Introducing Networking with Containers**
  + The Cloud Native Computing Foundation (CNCF) sponsors the Container Networking Interface (CNI) open source project. The CNI project aims to standardize the network interface for containers in cloud native environments, such as Kubernetes and Red Hat OpenShift Container Platform.
  + Podman uses the CNI project to implement a software-defined network (SDN) for containers on each host. Podman attaches each container to a virtual bridge and assigns each container a private IP address. The configuration file that specifies CNI settings for Podman is /etc/cni/net.d/87-podman-bridge.conflist.

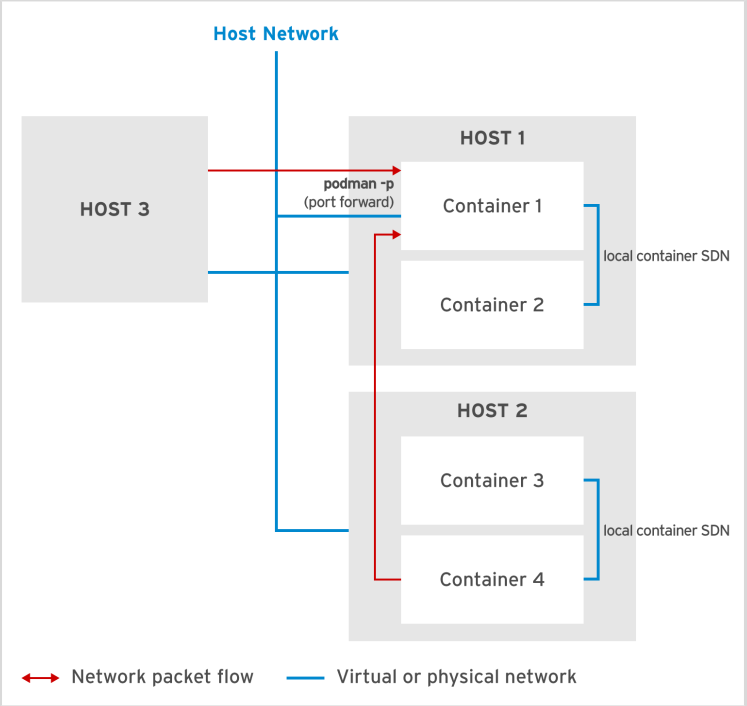
****

* + When Podman creates containers on the same host, it assigns each container a unique IP address and connects them all to the same software-defined network. These containers can communicate freely with each other by IP address
  + Containers created with Podman running on different hosts belong to different software-defined networks. Each SDN is isolated, which prevents a container in one network from communicating with a container in a different network. Because of network isolation, a container in one SDN can have the same IP address as a container in a different SDN.
  + It is also important to note that, by default, all container networks are hidden from the host network. That is, containers typically can access the host network, but without explicit configuration, there is no access back into the container network.
* **Mapping Network Ports**
  + Accessing a container from the host network can be a challenge. A container is assigned an IP address from a pool of available addresses. When a container is destroyed, the container's address is released back to the pool of available addresses. Another problem is that the container software-defined network is only accessible from the container host.
  + To solve these problems, define port forwarding rules to allow external access to a container service. Use the -p [<IP address>:][<host port>:]<container port> option with the podman run command to create an externally accessible container. Consider the following example

$ sudo podman run -d --name apache1 -p 8080:80

rhscl/httpd-24-rhel7:2.4

* + The value 8080:80 specifies that any requests to port 8080 on the host are forwarded to port 80 within the container.



* + You can also use the -p option to only forward requests to a container if those requests originate from a specified IP address

$ sudo podman run -d --name apache2 \

> -p 127.0.0.1:8081:80 rhscl/httpd-24-rhel7:2.4

* + The example above limits external access to the apache2 container to requests from localhost to host port 8081. These requests are forwarded to port 80 in the apache2 container.
  + If a port is not specified for the host port, Podman assigns a random available host port for the container

$ sudo podman run -d --name apache3 -p 127.0.0.1::80

rhscl/httpd-24-rhel7:2.4

* + To see the port assigned by Podman, use the podman port <container name> command

$ sudo podman port apache3

* + If only a container port is specified with the -p option, a random available host port is assigned to the container. Requests to this assigned host port from any IP address are forwarded to the container port.
* Example:
  + Create a MySQL container instance with persistent storage and port forwarding.

$ sudo podman run --name mysqldb-port \

> -d -v /var/local/mysql:/var/lib/mysql/data -p 13306:3306 \

> -e MYSQL\_USER=user1 -e MYSQL\_PASSWORD=mypa55 \

> -e MYSQL\_DATABASE=items -e MYSQL\_ROOT\_PASSWORD=r00tpa55 \

> rhscl/mysql-57-rhel7

* + Verify that the mysqldb-port container started successfully and enables port forwarding.

$ sudo podman ps --format="{{.ID}} {{.Names}} {{.Ports}}"

>> 9941da2936a5 mysqldb-port 0.0.0.0:13306->3306/tcp

* + Populate the database using the provided file. If there are no errors, then the command does not return any output.

$ mysql -uuser1 -h 127.0.0.1 -pmypa55 \

> -P13306 items < /home/student/DO180/labs/manage-networking/db.sql

* + Verify that the database loaded successfully by executing a non-interactive command inside the container.

$ sudo podman exec -it mysqldb-port \

> /opt/rh/rh-mysql57/root/usr/bin/mysql -uroot items -e "SELECT \* FROM Item"

* + Verify that the database loaded successfully by using port forwarding from the local host.

$ mysql -uuser1 -h 127.0.0.1 -pmypa55 \

> -P13306 items -e "SELECT \* FROM Item"

* + Verify that the database loaded successfully by opening an interactive terminal session inside the container.

$ sudo podman exec -it mysqldb-port /bin/bash

* + Verify that the database contains data.

bash-4.2$ mysql -uroot items -e "SELECT \* FROM Item"

* Example:
  + Create the /var/local/mysql directory with the correct SELinux context and permissions
    - Create the /var/local/mysql directory.

$ sudo mkdir -pv /var/local/mysql

* + - Add the appropriate SELinux context for the /var/local/mysql directory and its contents. With the correct context, you can mount this directory in a running container.

$ sudo semanage fcontext -a \

> -t container\_file\_t '/var/local/mysql(/.\*)?'

* + - Apply the SELinux policy to the newly created directory.

$ sudo restorecon -R /var/local/mysql

* + - Change the owner of the /var/local/mysql directory to match the mysql user and mysql group for the rhscl/mysql-57-rhel7 container image

$ sudo chown -Rv 27:27 /var/local/mysql

* + Deploy a MySQL container instance using the following characteristics:
    - Name: mysql-1
    - Run as daemon: yes
    - Volume: from /var/local/mysql host folder to /var/lib/mysql/data container folder.
    - Container image: rhscl/mysql-57-rhel7
    - Port forward: no
    - Environment variables:
      * MYSQL\_USER: user1
      * MYSQL\_PASSWORD: mypa55
      * MYSQL\_DATABASE: items
      * MYSQL\_ROOT\_PASSWORD: r00tpa55

$ sudo podman run --name mysql-1 \

> -d -v /var/local/mysql:/var/lib/mysql/data \

> -e MYSQL\_USER=user1 -e MYSQL\_PASSWORD=mypa55 \

> -e MYSQL\_DATABASE=items -e MYSQL\_ROOT\_PASSWORD=r00tpa55 \

> rhscl/mysql-57-rhel7

* + - Verify that the container was started correctly.

$ sudo podman ps --format="{{.ID}} {{.Names}}"

* + Load the items database using the /home/student/DO180/labs/manage-review/

db.sql script.

sudo podman inspect \

> -f '{{ .NetworkSettings.IPAddress }}' mysql-1

* + Load the database using the SQL commands in /home/student/DO180/labs/

manage-review/db.sql. Use the IP address you find in the previous step as the

database server's host IP.

$ mysql -uuser1 -h CONTAINER\_IP \

> -pmypa55 items < /home/student/DO180/labs/manage-review/db.sql

* + Use an SQL SELECT statement to output all rows of the Item table to verify that the Items database is loaded.
    - You can add the -e SQL parameter to the mysql command to execute an SQL instruction.

$ mysql -uuser1 -h CONTAINER\_IP -pmypa55 items \

> -e "SELECT \* FROM Item"

* + Stop the container gracefully.
    - This step is very important because a new container will be created sharing the same volume for database data. Having two containers using the same volume can corrupt the database. Do not restart the mysql-1 container

$ sudo podman stop mysql-1

* + Create a new container with the following characteristics:
    - Name: mysql-2
    - Run as a daemon: yes
    - Volume: from /var/local/mysql host folder to /var/lib/mysql/data container folder.
    - Container image: rhscl/mysql-57-rhel7
    - Port forward: yes, from host port 13306 to container port 3306
    - Environment variables:
      * MYSQL\_USER: user1
      * MYSQL\_PASSWORD: mypa55
      * MYSQL\_DATABASE: item
      * MYSQL\_ROOT\_PASSWORD: r00tpa55

$ sudo podman run --name mysql-2 \

> -d -v /var/local/mysql:/var/lib/mysql/data \

> -p 13306:3306 \

> -e MYSQL\_USER=user1 -e MYSQL\_PASSWORD=mypa55 \

> -e MYSQL\_DATABASE=items -e MYSQL\_ROOT\_PASSWORD=r00tpa55 \

> rhscl/mysql-57-rhel7

* + - Verify that the container was started correctly.

$ sudo podman ps --format="{{.ID}} {{.Names}}"

* + - Save the list of all containers (including stopped ones) to the /tmp/my-containers file.Save the information with the following command

sudo podman ps -a > /tmp/my-containers

* + Access the Bash shell inside the container and verify that the items database and the Item table are still available. Confirm also that the table contains data.

$ sudo podman exec -it mysql-2 /bin/bash

bash-4.2$ mysql -uroot

mysql> show databases;

mysql> use items;

mysql> show tables;

mysql> SELECT \* FROM Item;

exit

* + Using port forwarding, insert a new row into the Item table. The row should have a description value of Finished lab, and a done value of 1.
    - Connect to the MySQL database.

$ mysql -uuser1 -h workstation.lab.example.com \

> -pmypa55 -P13306 items

**Managing Container Images**

* **Public Registries**
  + Image registries are services offering container images to download. They allow image creators and maintainers to store and distribute container images to public or private audiences
  + Podman searches for and downloads container images from public and private registries. Red Hat Container Catalog is the public image registry managed by Red Hat. It hosts a large set of container images, including those provided by major open source projects, such as Apache, MySQL, and Jenkins. All images in the Container Catalog are vetted by the Red Hat internal security team, meaning they are trustworthy and secured against security flaws.
  + Red Hat container images provide the following benefits:
    - Trusted source
    - Original dependencies
    - Vulnerability-free
    - Runtime protection: All applications in container images run as non-root users, minimizing the exposure surface to malicious or faulty applications.
    - Red Hat Enterprise Linux (RHEL) compatible.
    - Red Hat support.
  + Quay.io is another public image repository sponsored by Red Hat. Quay.io introduces several exciting features, such as server-side image building, fine-grained access controls, and automatic scanning of images for known vulnerabilities
  + While Red Hat Container Catalog images are trusted and verified, Quay.io offers live images regularly updated by creators.
  + Quay.io users can create their namespaces, with fine-grained access

control, and publish the images they create to that namespace.

* + Container Catalog users rarely or never push new images, but consume trusted images generated by the Red Hat team.
* **Private Registries**
  + Some image creators or maintainers want to make their images publicly available. Other image creators, however, prefer to keep their images private due to:
    - Company privacy and secret protection.
    - Legal restrictions and laws.
    - Avoidance of publishing images in development.
  + Private registries give image creators the control about their images placement, distribution and usage.
* **Configuring Registries in Podman**
  + To configure registries for the podman command, you need to update the /etc/containers/registries.conf file. Edit the registry entry in the [registries.search] section, adding an entry to the values list.

[registries.search]

registries = ["registry.access.redhat.com", "quay.io"]

* + Use an FQDN and port number to identify a registry. A registry that does not include a port number has a default port number of 5000. If the registry uses a different port, it must be specified. Indicate port numbers by appending a colon (:) and the port number after the FQDN.
  + Secure connections to a registry require a trusted certificate. To support insecure connections,add the registry name to the registries entry in [registries.insecure] section of /etc/containers/registries.conf file:

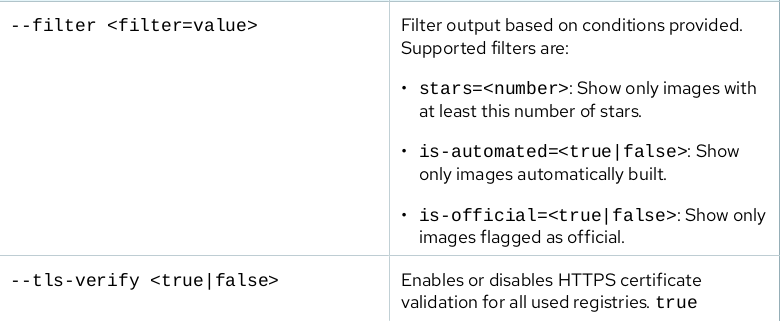
[registries.insecure]

registries = ['localhost:5000']

* **Accessing Registries**
  + **Searching for Images in Registries.**
    - The podman search command finds images by image name, user name, or description from all the registries listed in the /etc/containers/registries.conf configuration file. The syntax for the podman search command is shown below

$ sudo podman search [OPTIONS] <term>





* + **Registry HTTP API**
    - A remote registry exposes web services that provide an application programming interface (API) to the registry.
    - Podman uses these interfaces to access and interact with remote repositories.
    - Many registries conform to the Docker Registry HTTP API v2 specification, which exposes a standardized REST interface for registry interaction. You can use this REST interface to directly interact with a registry, instead of using Podman.
    - Some samples using this API with curl commands are shown below:
      * To list all repositories available in a registry, use the /v2/\_catalog endpoint. The n parameter is used to limit the number of repositories to return.

curl -Ls https://myserver/v2/\_catalog?n=3

{"repositories":["centos/httpd","do180/custom-httpd","hello-openshift"]}

* + - * If Python is available, use it to format the JSON response:

curl -Ls https://myserver/v2/\_catalog?n=3 \

> | python -m json.tool

* + - * The /v2/<name>/tags/list endpoint provides the list of tags available for a single image

curl -Ls \

> https://quay.io/v2/redhattraining/httpd-parent/tags/list \

> | python -m json.tool

* + - Quay.io offers a dedicated API to interact with repositories beyond what is specified in Docker Repository API. See https://docs.quay.io/api/ for details.
  + **Registry Authentication**
    - Some container image registries require access authorization. The podman login command allows username and password authentication to a registry

$ sudo podman login -u username \

> -p password registry.access.redhat.com

* + - The registry HTTP API requires authentication credentials. First, use the Red Hat Single Sign On (SSO) service to obtain an access token

$ curl -u username:password -Ls \

>"https://sso.redhat.com/auth/realms/rhcc/protocol/redhat-docker-v2/auth?service=docker-registry"

* + - Then, include this token in a Bearer authorization header in subsequent requests

[student@workstation ~]$ curl -H "Authorization: Bearer eyJh...mgL4" \

> -Ls https://registry.redhat.io/v2/rhscl/mysql-57-rhel7/tags/list \

> | python -mjson.tool

* + **Pulling Images**

$ sudo podman pull [OPTIONS] [REGISTRY[:PORT]/]NAME[:TAG]

* + - The podman pull command uses the image name obtained from the search subcommand to pull an image from a registry. The pull subcommand allows adding the registry name to the image. This variant supports having the same image in multiple registries.
    - Example:

$ sudo podman pull quay.io/bitnami/nginx

* + - If the image name does not include a registry name, Podman searches for a matching container image using the registries listed in the /etc/containers/registries.conf configuration file. Podman search for images in registries in the same order they appear in the configuration file.
  + **Listing Local Copies of Images**
    - By default, Podman stores container images in the /var/lib/containers/

storage/overlay-images directory.

* + - Podman provides an images subcommand to list all the container images stored locally.

$ sudo podman images

* + **Image Tags**
    - An image tag is a mechanism to support multiple releases of the same image.

sudo podman pull rhscl/mysql-57-rhel7:5.7

* **Manipulating Container Images**
  + There are various ways to manage image containers while adhering to DevOps principles. For example, a developer finishes testing a custom container in a machine and needs to transfer this container image to another host for another developer, or to a production server. There are two ways to do this:
    - Save the container image to a .tar file.
    - Publish (push) the container image to an image registry.
  + **Saving and Loading Images**
    - Existing images from the Podman local storage can be saved to a .tar file using the podman save command.
    - The generated file is not a regular TAR archive; it contains image metadata and preserves the original image layers. Using this file, Podman can recreate the original image exactly as it was.

$ sudo podman save [-o FILE\_NAME] IMAGE\_NAME[:TAG]

* + - Podman sends the generated image to the standard output as binary data. To avoid that, use the -o option.
    - Example: $ sudo podman save \

> -o mysql.tar registry.access.redhat.com/rhscl/mysql-57-rhel7:5.7

* + - Use the .tar files generated by the save subcommand for backup purposes. To restore the container image, use the podman load command.

$ sudo podman load [-i FILE\_NAME]

Example: $ sudo podman load -i mysql.tar

* + - Gzip using the --compress parameter. The load subcommand uses the gunzip command before importing the file to the local storage.
  + **Deleting Images**
    - Podman keeps any image downloaded in its local storage, even the ones currently unused by any container. However, images can become outdated, and should be subsequently replaced.
    - Any updates to images in a registry are not automatically updated. The image must be removed and then pulled again to guarantee that the local storage has the latest version of an image.
    - To delete an image from the local storage, run the podman rmi command. The syntax for this command is as follows

$ sudo podman rmi [OPTIONS] IMAGE [IMAGE...]

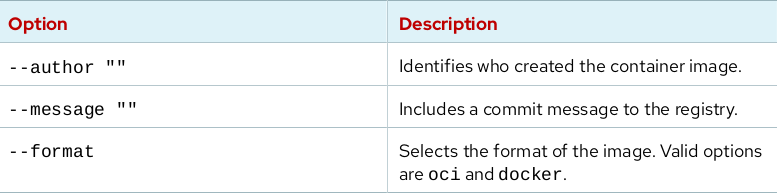
* + **Deleting all Images**
    - To delete all images that are not used by any container.

$ sudo podman rmi -a

* + **Modifying Images**
    - Ideally, all container images should be built using a Dockerfile, in order to create a clean,lightweight set of image layers without log files, temporary files, or other artifacts created by the container customization. However, some users may provide container images as they are, without a Dockerfile. As an alternative approach to creating new images, change a running container in place and save its layers to create a new container image. The podman commit command provides this feature.

$ sudo podman commit [OPTIONS] CONTAINER \

> [REPOSITORY[:PORT]/]IMAGE\_NAME[:TAG]



* + - The --message option is not available in the default OCI container format.
    - To find the ID of a running container in Podman, run the podman ps command

$ sudo podman ps

* + - Eventually, administrators might customize the image and set the container to the desired state. To identify which files were changed, created, or deleted since the container was started, use the diff subcommand. This subcommand only requires the container name or container ID

$ sudo podman diff mysql-basic

* + - The diff subcommand tags any added file with an A, any changed ones with a C, and any deleted file with a D.
    - The diff command only reports added, changed, or deleted files to the container file system. Files that are mounted to a running container are not considered part of the container file system.
    - To retrieve the list of mounted files and directories for a running container, use the podman inspect command.

$ sudo podman inspect \

> -f "{{range .Mounts}}{{println .Destination}}{{end}}" CONTAINER\_NAME/ID

* + - To commit the changes to another image, run the following command

$ sudo podman commit mysql-basic mysql-custom

* + **Tagging Images**
    - A project with multiple images based on the same software could be distributed, creating individual projects for each image, but this approach requires more maintenance for managing and deploying the images to the correct locations.
    - Container image registries support tags to distinguish multiple releases of the same project. For example, a customer might use a container image to run with a MySQL or PostgreSQL database, using a tag as a way to differentiate which database is to be used by a container image
    - Usually, the tags are used by container developers to distinguish between multiple versions of the same software. Multiple tags are provided to identify a release easily.
    - The official MySQL container image website uses the version as the tag's name (5.5.16). Also, the same image has a second tag with the minor version, such as 5.5, to minimize the need to get the latest release for a specific version.
    - To tag an image, use the podman tag command

$ sudo podman tag [OPTIONS] IMAGE[:TAG] \

> [REGISTRYHOST/][USERNAME/]NAME[:TAG]

* + - The IMAGE argument is the image name with an optional tag, which is managed by Podman. The following argument refers to the new alternative name for the image. Podman assumes the latest version, as indicated by the latest tag, if the tag value is absent. For example, to tag an image, use the following command

$ sudo podman tag mysql-custom devops/mysql

* + - The mysql-custom option corresponds to the image name in the container registry.
    - To use a different tag name, use the following command instead.

$ sudo podman tag mysql-custom devops/mysql:snapshot

* + **Removing Tags from Images**

$ sudo podman rmi devops/mysql:snapshot

* + - Because multiple tags can point to the same image, to remove an image referred to by multiple tags, first remove each tag individually.
  + Podman automatically adds the latest tag if you do not specify any tag, because Podman considers the image to be the latest build. However, this may not be true depending on how each project uses tags. For example, many open source projects consider the latest tag to match the most recent release, but not the latest build.
  + **Publishing Images to a Registry**
    - To publish an image to a registry, it must reside in Podman's local storage and be tagged for identification purposes. To push the image to the registry the syntax of the push subcommand

$ sudo podman push [OPTIONS] IMAGE [DESTINATION]

* **Example: Creating a Custom Apache Container Image**
  + Log into your Quay.io account and start a container by using the image available at quay.io/redhattraining/httpd-parent. The -p option allows you to specify a redirect port. In this case, Podman forwards incoming requests on TCP port 8180 of the host to TCP port 80 of the container

$ sudo podman login quay.io

Username: your\_quay\_username

Password: your\_quay\_password

$ sudo podman run -d --name official-httpd \

> -p 8180:80 quay.io/redhattraining/httpd-parent

* + Create an HTML page on the official-httpd container.

$ sudo podman exec -it official-httpd /bin/bash

bash-4.4# echo "DO180 Page" > /var/www/html/do180.html

bash-4.4# exit

* + Ensure that the HTML file is reachable from the workstation VM by using the curl

command.

$ curl 127.0.0.1:8180/do180.html

* + Use the podman diff command to examine the differences in the container between the image and the new layer created by the container.

$ sudo podman diff official-httpd

* + Often, web server container images label the /var/www/html directory as a

volume. In these cases, any files added to this directory are not considered part

of the container file system, and would not show in the output of the git diff

command.

* + The quay.io/redhattraining/httpd-parent container image does not label

the /var/www/html directory as a volume. As a result, the change to the /var/

www/html/do180.html file is considered a change to the underlying container file

system.

* + Create a new image with the changes created by the running container.
    - Stop the official-httpd container.

$ sudo podman stop official-httpd

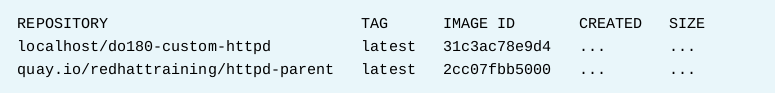
* + - Commit the changes to a new container image with a new name. Use your name as the author of the changes.

sudo podman commit \

> -a 'Your Name' official-httpd do180-custom-httpd

* + - List the available container images.

$ sudo podman images



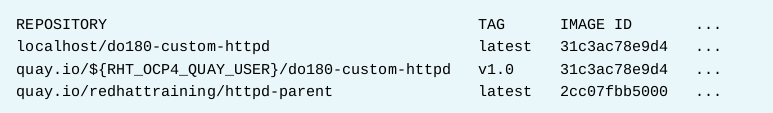
* + - To tag the image with the registry host name and tag, run the following command

sudo podman tag do180-custom-httpd \

> quay.io/${RHT\_OCP4\_QUAY\_USER}/do180-custom-httpd:v1.0

* + - Run the podman images command to ensure that the new name has been added to the cache.

$ sudo podman images



* + - Publish the image to your Quay.io registry.

$ sudo podman push \

> quay.io/${RHT\_OCP4\_QUAY\_USER}/do180-custom-httpd:v1.0

**Creating Custom Container Images**

* **Reusing Existing Dockerfiles**
  + One method of creating container images has been covered so far: create a container, modify it to meet the requirements of the application to run in it, and then commit the changes to an image. This option, although straightforward, is only suitable for using or testing very specific changes. It does not follow best software practices, like maintainability, automation of building, and repeatability.
  + Dockerfiles are another option for creating container images, addressing these limitations. Dockerfiles are easy to share, version control, reuse, and extend.
  + Dockerfiles also make it easy to extend one image, called a child image, from another image, called a parent image. A child image incorporates everything in the parent image and all changes and additions made to create it.
  + To share and reuse images, many popular applications, languages, and frameworks are already
  + available in public image registries such as Quay.io. It is not trivial to customize an application configuration to follow recommended practices for containers, and so starting from a proven parent image usually saves a lot of work.
  + Using a high-quality parent image enhances maintainability, especially if the parent image is kept updated by its author to account for bug fixes and security issues.
  + Typical scenarios in creating a Dockerfile for building a child image from an existing container image include:
    - Add new runtime libraries, such as database connectors.
    - Include organization-wide customization such as SSL certificates and authentication providers.
    - Add internal libraries to be shared as a single image layer by multiple container images for different applications.
  + Changing an existing Dockerfile to create a new image can also be a sensible approach in other scenarios. For example:
    - Trim the container image by removing unused material (such as man pages, or documentation found in /usr/share/doc).
    - Lock either the parent image or some included software package to a specific release to lower risk related to future software updates.
    - Two sources of container images to use either as parent images or for changing their Dockerfiles are Docker Hub and the Red Hat Software Collections Library (RHSCL).
* **Working with the Red Hat Software Collections Library**
  + Red Hat Software Collections Library (RHSCL), or simply Software Collections, is Red Hat's solution for developers who need to use the latest development tools that usually do not fit the standard RHEL release schedule.
  + Red Hat Enterprise Linux (RHEL) provides a stable environment for enterprise applications. This requires RHEL to keep the major releases of upstream packages at the same level to prevent API and configuration file format changes. Security and performance fixes are backported from later upstream releases, but new features that would break backward-compatibility are not backported.
  + RHSCL allows software developers to use the latest version without impacting RHEL, because RHSCL packages do not replace or conflict with default RHEL packages. Default RHEL packages and RHSCL packages are installed side-by-side.
  + All RHEL subscribers have access to the RHSCL. To enable a particular software

collection for a specific user or application environment (for example, MySQL 5.7,

which is named rh-mysql57), enable the RHSCL software Yum repositories and

follow a few simple steps.

* **Finding Dockerfiles from the red Hat Software Collections Library**
  + RHSCL is the source of most container images provided by the Red Hat image registry for use by RHEL Atomic Host and OpenShift Container Platform customers.
  + Red Hat provides RHSCL Dockerfiles and related sources in the rhscl-dockerfiles package available from the RHSCL repository. Community users can get Dockerfiles for CentOS-based equivalent container images from https://github.com/sclorg?q=-container.
* **Container Images in Red Hat Container Catalog (RHCC)**
  + Mission-critical applications require trusted containers. The Red Hat Container Catalog is a repository of reliable, tested, certified, and curated collection of container images built on versions of Red Hat Enterprise Linux (RHEL) and related systems.
  + Container images available through
  + RHCC has undergone a quality-assurance process. All components have been rebuilt by Red Hat to avoid known security vulnerabilities. They are upgraded on a regular basis so that they contain the required version of software even when a new image is not yet available. Using RHCC, you can browse and search for images, and you can access information about each image, such as its

version, contents, and usage.

* **Searching for Images Using Quay.io**
  + Quay.io is an advanced container repository from CoreOS optimized for team collaboration. You can search for container images using https://quay.io/search.
  + Clicking on an image's name provides access to the image information page, including access to all existing tags for the image, and the command to pull the image.
* **Finding Dockerfiles on Docker Hub**
  + Anyone can create a Docker Hub account and publish container images there. There are no general assurances about quality and security; images on Docker Hub range from professionally supported to one-time experiments. Each image has to be evaluated individually.
  + After searching for an image, the documentation page might provide a link to its Dockerfile. For example, the first result when searching for mysql is the documentation page for the MySQL official image at https://hub.docker.com/\_/mysql.
  + On that page, the link for the 5.6/Dockerfile image points to the docker-library GitHub project, which hosts Dockerfiles for images built by the Docker community automatic build system.
  + The direct URL for the Docker Hub MySQL 5.6 Dockerfile tree is https://github.com/docker-library/mysql/blob/master/5.6.
* **Describing How to use the OpenShift Source-to-Image Tool**
  + Source-to-Image (S2I) provides an alternative to using Dockerfiles to create new container images and can be used either as a feature from OpenShift or as the standalone s2i utility. S2I allows developers to work using their usual tools, instead of learning Dockerfile syntax and using operating system commands such as yum, and usually creates slimmer images, with fewer layers.
  + S2I uses the following process to build a custom container image for an application:
    - Start a container from a base container image called the builder image, which includes a programming language runtime and essential development tools such as compilers and package managers.
    - Fetch the application source code, usually from a Git server, and send it to the container.
    - Build the application binary files inside the container.
    - Save the container, after some clean up, as a new container image, which includes the programming language runtime and the application binaries.
  + The builder image is a regular container image following a standard directory structure and providing scripts that are called during the S2I process. Most of these builder images can also be used as base images for Dockerfiles, outside the S2I process.
  + The s2i command is used to run the S2I process outside of OpenShift, in a Docker-only environment. It can be installed on a RHEL system from the source-to-image RPM package, and on other platforms, including Windows and Mac OS, from the installers available in the S2I project on GitHub.
* **Building Custom Container Images with Dockerfile**
  + **Building Base Containers**
    - A Dockerfile is a mechanism to automate the building of container images. Building an image from a Dockerfile is a three-step process:
      * Create a working directory
      * Write the Dockerfile
      * Build the image with Podman
    - **Create a Working Directory**
      * The working directory is the directory containing all files needed to build the image.
      * Creating an empty working directory is good practice to avoid incorporating unnecessary files into the image.
      * For security reasons, the root directory, /, should never be used as a working directory for image builds.
    - **Write the Dockerfile Specification**
      * A Dockerfile is a text file that must exist in the working directory. This file contains the instructions needed to build the image. The basic syntax of a Dockerfile follows:

# Comment

INSTRUCTION arguments

* + - * Lines that begin with a hash, or pound, symbol (#) are comments. INSTRUCTION states for any Dockerfile instruction keyword. Instructions are not case-sensitive, but the convention is to make instructions all uppercase to improve visibility.
      * The first non-comment instruction must be a FROM instruction to specify the base image.
      * Dockerfile instructions are executed into a new container using this image and then committed to a new image. The next instruction (if any) executes into that new image. The execution order of instructions is the order of their appearance in the Dockerfile.
      * Each Dockerfile instruction runs in an independent container using an intermediate image built from every previous command. This means each instruction is independent from other instructions in the Dockerfile.
      * The following is an example Dockerfile for building a simple Apache web server container:



* + - * 1: Lines that begin with a hash, or pound, sign (#) are comments.
      * 2: The FROM instruction declares that the new container image extends ubi7/ubi:7.7 container base image. Dockerfiles can use any other container image as a base image, not only images from operating system distributions. Red Hat provides a set of container images that are certified and tested and highly recommends using these container images as a base.
      * 3: The LABEL is responsible for adding generic metadata to an image. A LABEL is a simple key-value pair.
      * 4: MAINTAINER indicates the Author field of the generated container image's metadata. You can use the podman inspect command to view image metadata.
      * 5: RUN executes commands in a new layer on top of the current image. The shell that is used to execute commands is /bin/sh.
      * 6: EXPOSE indicates that the container listens on the specified network port at runtime. The EXPOSE instruction defines metadata only; it does not make ports accessible from the host. The -p option in the podman run command exposes container ports from the host.
      * 7: ENV is responsible for defining environment variables that are available in the container. You can declare multiple ENV instructions within the Dockerfile. You can use the env command inside the container to view each of the environment variables.
      * 8: ADD instruction copies files or folders from a local or remote source and adds them to the container's file system. If used to copy local files, those must be in the working directory. ADD

instruction unpacks local .tar files to the destination image directory.

* + - * 9: COPY copies files from the working directory and adds them to the container's file system. It is not possible to copy a remote file using its URL with this Dockerfile instruction.
      * 10: USER specifies the username or the UID to use when running the container image for the RUN, CMD, and ENTRYPOINT instructions. It is a good practice to define a different user other

than root for security reasons.

* + - * 11: ENTRYPOINT specifies the default command to execute when the image runs in a container.If omitted, the default ENTRYPOINT is /bin/sh -c.
      * 12: CMD provides the default arguments for the ENTRYPOINT instruction. If the default
      * ENTRYPOINT applies (/bin/sh -c), then CMD forms an executable command and parameters that run at container start.
  + **CMD and ENTRYPOINT**
    - ENTRYPOINT and CMD instructions have two formats:
      * Exec form (using a JSON array):

ENTRYPOINT ["command", "param1", "param2"]

CMD ["param1","param2"]

* + - * Shell form:

ENTRYPOINT command param1 param2

CMD param1 param2

* + - Exec form is the preferred form. Shell form wraps the commands in a /bin/sh -c shell, creating a sometimes unnecessary shell process. Also, some combinations are not allowed, or may not work as expected.
    - For example, if ENTRYPOINT is ["ping"] (exec form) and CMD is localhost (shell form), then the expected executed command is ping localhost, but the container tries ping /bin/sh -c localhost, which is a malformed command.
    - The Dockerfile should contain at most one ENTRYPOINT and one CMD instruction. If more than one of each is present, then only the last instruction takes effect. CMD can be present without specifying an ENTRYPOINT. In this case, the base image's ENTRYPOINT applies, or the default ENTRYPOINT if none is defined
    - Podman can override the CMD instruction when starting a container. If present, all parameters for the podman run command after the image name form the CMD instruction. For example, the following instruction causes the running container to display the current time.

ENTRYPOINT ["/bin/date", "+%H:%M"]

* + - The ENTRYPOINT defines both the command to be executed and the parameters. So the CMD instruction cannot be used. The following example provides the same functionality, with the added benefit of the CMD instruction being overwritable when a container starts.

ENTRYPOINT ["/bin/date"]

CMD ["+%H:%M"

* + - In both cases, when a container starts without providing a parameter, the current time is displayed.

$ sudo podman run -it do180/rhel

11:41

* + - In the second case, if a parameter appears after the image name in the podman run command, it overwrites the CMD instruction. The following command displays the current day of the week instead of the time.

$ sudo podman run -it do180/rhel +%A

Tuesday

* + - Another approach is using the default ENTRYPOINT and the CMD instruction to define the initial command. The following instruction displays the current time, with the added benefit of being able to be overridden at runtime.

CMD ["date", "+%H:%M"]

* + **ADD and COPY**
    - The ADD and COPY instructions have two forms.
      * The Shell form

ADD <source>... <destination>

COPY <source>... <destination>

* + - * The Exec form

ADD ["<source>",... "<destination>"]

COPY ["<source>",... "<destination>"]

* + - If the source is a file system path, it must be inside the working directory.
    - The ADD instruction also allows you to specify a resource using a URL

ADD http://someserver.com/filename.pdf /var/www/html

* + - If the source is a compressed file, then the ADD instruction decompresses the file to the destination folder. The COPY instruction does not have this functionality.
    - Both the ADD and COPY instructions copy the files, retaining permissions, with root as the owner, even if the USER instruction is specified. Red Hat recommends using a RUN instruction after the copy to change the owner and avoid “permission denied” errors.
  + **Layering Image**
    - Each instruction in a Dockerfile creates a new image layer. Having too many instructions in a Dockerfile causes too many layers, resulting in large images. For example, consider the following RUN instructions in a Dockerfile:

RUN yum --disablerepo=\* --enablerepo="rhel-7-server-rpms"

RUN yum update -y

RUN yum install -y httpd

* + - The previous example is not a good practice when creating container images. It creates three layers (one for each RUN instruction) while only the last is meaningful. Red Hat recommends minimizing the number of layers. You can achieve the same objective while creating a single layer by using the && conjunction:

RUN yum --disablerepo=\* --enablerepo="rhel-7-server-rpms" && yum update -y && yum install -y httpd

* + - The problem with this approach is that the readability of the Dockerfile decays. Use the \ escape code to insert line breaks and improve readability. You can also indent lines to align the commands.

RUN yum --disablerepo=\* --enablerepo="rhel-7-server-rpms" && \

yum update -y && \

yum install -y httpd

* + - This example creates only one layer, and the readability improves. RUN, COPY, and ADD instructions create new image layers, but RUN can be improved this way.
    - Red Hat recommends applying similar formatting rules to other instructions accepting multiple parameters, such as LABEL and ENV:

LABEL version="2.0" \

description="This is an example container image" \

creationDate="01-09-2017"

ENV MYSQL\_ROOT\_PASSWORD="my\_password" \

MYSQL\_DATABASE "my\_database"

* + **Building Images with Podman**
    - The podman build command processes the Dockerfile and builds a new image based on the instructions it contains. The syntax for this command is as follows.

$ podman build -t NAME:TAG DIR

* + - DIR is the path to the working directory, which must include the Dockerfile. It can be the current directory as designated by a dot (.) if the working directory is the current directory. NAME:TAG is a name with a tag given to the new image. If TAG is not specified, then the image is automatically tagged as latest.
* **Example: Creating a Basic Apache Container Image**
  + Create the Apache Dockerfile
    - Open a terminal on the workstation. Use your preferred editor and create a new Dockerfile.

$ vim /home/student/DO180/labs/dockerfile-create/Dockerfile

* + - Docker FIle

FROM ubi7/ubi:7.7

MAINTAINER Your Name <youremail>

LABEL description="A custom Apache container based on UBI 7"

RUN yum install -y httpd && \

yum clean all

RUN echo "Hello from Dockerfile" > /var/www/html/index.html

EXPOSE 80

CMD ["httpd", "-D", "FOREGROUND"]

* + Build and verify the Apache container image.

$ cd /home/student/DO180/labs/dockerfile-create

$ sudo podman build --layers=false \

> -t do180/apache .

* + After the build process has finished, run podman images to see the new image in the image repository

$ sudo podman images

* + Run the Apache container

sudo podman run --name lab-apache \

> -d -p 10080:80 do180/apache

* + Run the podman ps command to see the running container

sudo podman ps

* + Use the curl command to verify that the server is running

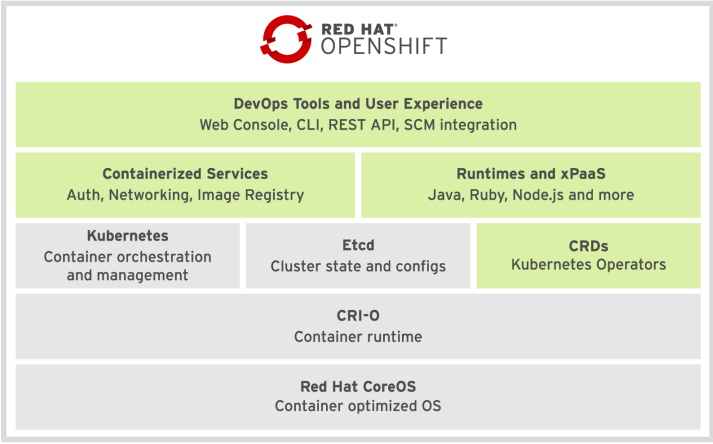
curl 127.0.0.1:10080

**Deploying Containerized Application on OpenShift**

* **Describing Kubernetes and OpenShiftArchitecture**
* **Kubernetes and OpenShift**
  + Kubernetes is an orchestration service that simplifies the deployment, management, and scaling of containerized applications.
  + One of the main advantages of using Kubernetes is that it uses several nodes to ensure the resiliency and scalability of its managed applications.
  + Kubernetes forms a cluster of node servers that run containers and are

centrally managed by a set of master servers.

* + A server can act as both a server and a node, but those roles are usually segregated for increased stability.
  + **Kubernetes Terminology**
    - **Node**: A server that hosts applications in a Kubernetes cluster.
    - **Master Node:** A node server that manages the control plane in a Kubernetes cluster. Master nodes provide basic cluster services such as APIs or controllers.
    - **Worker Node**: Also named Compute Node, worker nodes execute workloads for the cluster. Application pods are scheduled onto worker nodes
    - **Resource:** Resources are any kind of component definition managed by Kubernetes. Resources contain the configuration of the managed component (for example, the role assigned to a node), and the current state of the component (for example, if the node is available).
    - **Controller**: A controller is a Kubernetes process that watches resources and makes changes attempting to move the current state towards the desired state.
    - **Label:** A key-value pair that can be assigned to any Kubernetes resource. Selectors use labels to filter eligible resources for scheduling and other operations.
    - **Namespace:** A scope for Kubernetes resources and processes, so that resources with the same name can be used in different boundaries.
  + The latest Kubernetes versions implement many controllers as Operators. Operators are Kubernetes plug-in components that can react to cluster events and control the state of resources.
  + **Red Hat OpenShift Container Platform is a set of modular components and services built on top of Red Hat CoreOS and Kubernetes.**
  + RHOCP adds PaaS capabilities such as remote management, increased security, monitoring and auditing, application life-cycle management, and self-service interfaces for developers.
  + An OpenShift cluster is a Kubernetes cluster that can be managed the same way, but using the management tools provided by OpenShift, such as the command-line interface or the web console. This allows for more productive workflows and makes common tasks much easier.
  + **OpenShift Terminology**
    - **Infra Node:** A node server containing infrastructure services like monitoring, logging, or external routing.
    - **Console:** A web UI provided by the RHOCP cluster that allows developers and administrators to interact with cluster resources.
    - **Project:** OpenShift's extension of Kubernetes' namespaces. Allows the definition of user access control (UAC) to resources.
  + The following schema illustrates the OpenShift Container Platform stack.



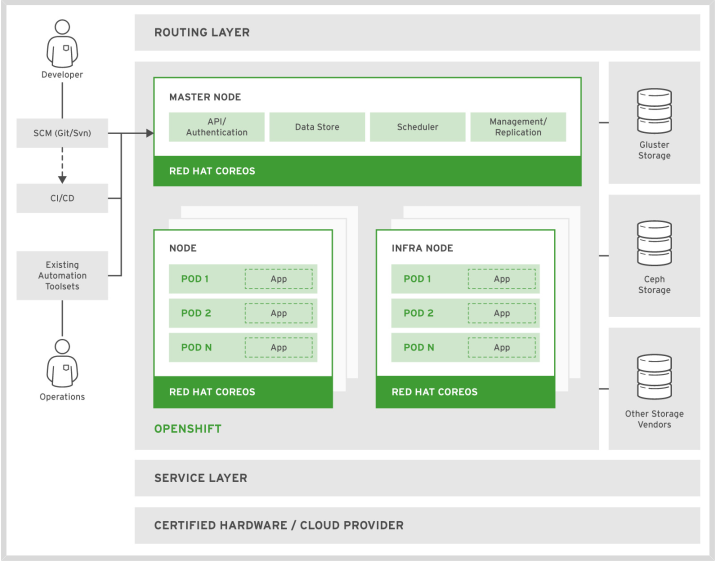
* + From bottom to top, and from left to right, this shows the basic container infrastructure, integrated and enhanced by Red Hat
    - The base OS is Red Hat CoreOS. Red Hat CoreOS is a Linux distribution focused on providing an immutable operating system for container execution.
    - CRI-O is an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using OCI (Open Container Initiative) compatible runtimes. CRI-O can use any container runtime that satisfies CRI: runc (used by the Docker service), libpod (used by Podman) or rkt (from CoreOS).
    - Kubernetes manages a cluster of hosts, physical or virtual, that run containers. It uses resources that describe multi container applications composed of multiple resources, and how they interconnect.
    - Etcd is a distributed key-value store, used by Kubernetes to store configuration and state information about the containers and other resources inside the Kubernetes cluster.
    - Custom Resource Definitions (CRDs) are resource types stored in Etcd and managed by Kubernetes. These resource types form the state and configuration of all resources managed by OpenShift.
    - Containerized services fulfill many PaaS infrastructure functions, such as networking and authorization. RHOCP uses the basic container infrastructure from Kubernetes and the underlying container runtime for most internal functions. That is, most RHOCP internal services

run as containers orchestrated by Kubern

* + - Runtimes and xPaaS are base container images ready for use by developers, each preconfigured with a particular runtime language or database. The xPaaS offering is a set of base images for Red Hat middleware products such as JBoss EAP and ActiveMQ. Red Hat OpenShift Application Runtimes (RHOAR) are a set of runtimes optimized for cloud native applications in OpenShift. The application runtimes available are Red Hat JBoss EAP, OpenJDK, Thorntail, Eclipse Vert.x,

Spring Boot, and Node.js.

* + - DevOps tools and user experience: RHOCP provides web UI and CLI management tools for managing user applications and RHOCP services. The OpenShift web UI and CLI tools are built from REST APIs which can be used by external tools such as IDEs and CI platforms.



* **New Features in RHOCP 4**
  + RHOCP 4 is a massive change from previous versions. As well as keeping backwards compatibility with previous releases, it includes new features, such as:
    - CoreOS as the mandatory operating system for all nodes, offering an immutable infrastructure optimized for containers.
    - A brand new cluster installer which guides the process of installation and update.
    - A self-managing platform, able to automatically apply cluster updates and recoveries without disruption.
    - A redesigned application life-cycle management.
    - An Operator SDK to build, test, and package Operators.
* **Describing Kubernetes Resource Types**
  + Kubernetes has six main resource types that can be created and configured using a YAML or a JSON file, or using OpenShift management tools:
  + **Pods (po):** Represent a collection of containers that share resources, such as IP addresses and persistent storage volumes. It is the basic unit of work for Kubernetes.
  + **Services (svc):** Define a single IP/port combination that provides access to a pool of pods. By default, services connect clients to pods in a round-robin fashion.
  + **Replication Controllers (rc):** A Kubernetes resource that defines how pods are replicated (horizontally scaled) into different nodes. Replication controllers are a basic Kubernetes service to provide high availability for pods and containers.
  + **Persistent Volumes (pv):** Define storage areas to be used by Kubernetes pods.
  + **Persistent Volume Claims (pvc):** Represent a request for storage by a pod. PVCs link a PV to a pod so its containers can make use of it, usually by mounting the storage into the container's file system.
  + **ConfigMaps (cm) and Secrets:** Contains a set of keys and values that can be used by other resources. ConfigMaps and Secrets are usually used to centralize configuration values used by several resources. Secrets differ from ConfigMaps maps in that Secrets' values are always encoded (not encrypted) and their access is restricted to fewer authorized users.
* **OpenShift Resource Types**
  + The main resource types added by OpenShift Container Platform to Kubernetes are as follows:
  + **Deployment config (dc):** Represents the set of containers included in a pod, and the deployment strategies to be used. A dc also provides a basic but extensible continuous delivery workflow.
  + **Build config (bc):** Defines a process to be executed in the OpenShift project. Used by the OpenShift Source-to-Image (S2I) feature to build a container image from application source code stored in a Git repository. A bc works together with a dc to provide a basic but extensible continuous integration and continuous delivery workflows.
  + **Routes:** Represent a DNS host name recognized by the OpenShift router as an ingress point for applications and microservices.
  + To obtain a list of all the resources available in a RHOCP cluster and their

abbreviations, use the oc api-resources or kubectl api-resources commands.

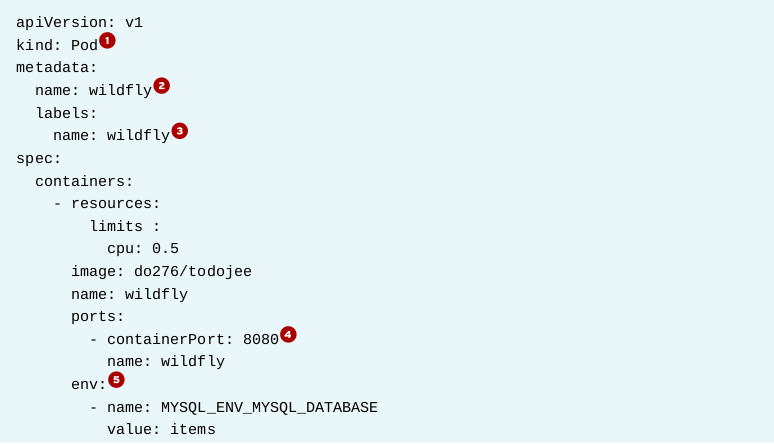
* **Networking**
  + Each container deployed in a Kubernetes cluster has an IP address assigned from an internal network that is accessible only from the node running the container. Because of the container's ephemeral nature, IP addresses are constantly assigned and released.
  + Kubernetes provides a software-defined network (SDN) that spawns the internal container networks from multiple nodes and allows containers from any pod, inside any host, to access pods from other hosts. Access to the SDN only works from inside the same Kubernetes cluster.
  + Containers inside Kubernetes pods should not connect to each other's dynamic IP address directly.
  + Services resolve this problem by linking more stable IP addresses from the SDN to the pods. If pods are restarted, replicated, or rescheduled to different nodes, services are updated, providing scalability and fault tolerance.
  + External access to containers is more complicated. Kubernetes services can specify a NodePort attribute, which is a network port redirected by all the cluster nodes to the SDN. Then, the containers in the node can redirect a port to the node's port. Unfortunately, none of these approaches scale well.
  + OpenShift makes external access to containers both scalable and simpler by defining route resources. A route defines external-facing DNS names and ports for a service. A router (ingress controller) forwards HTTP and TLS requests to the service addresses inside the Kubernetes SDN. The only requirement is that the desired DNS names are mapped to the IP addresses of the RHOCP router nodes.
* **Creating Kubernetes Resources**
  + **The Red Hat OpenShift Container Platform (RHOCP) Command-line Tool**
    - The main method of interacting with an RHOCP cluster is using the oc command. The basic usage of the command is through its subcommands in the following syntax:

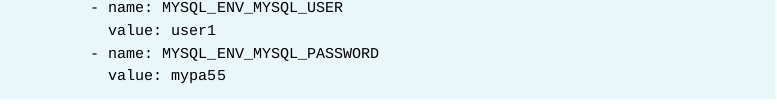
$> oc <command>

* + - Before interacting with a cluster, most operations require a logged-in user. The syntax to log in is shown below:

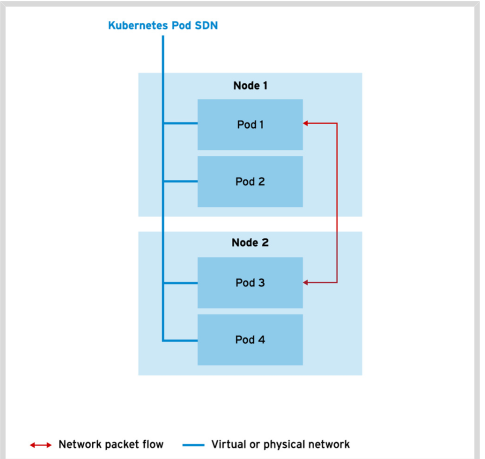
$> oc login <clusterUrl>

* + **Describing Pod Resource Definition Syntax**
    - RHOCP runs containers inside Kubernetes pods, and to create a pod from a container image, OpenShift needs a pod resource definition. This can be provided either as a JSON or YAML text file, or can be generated from defaults by the oc new-app command or the OpenShift web console.
    - A pod is a collection of containers and other resources. An example of a WildFly application server pod definition in YAML format is shown below:



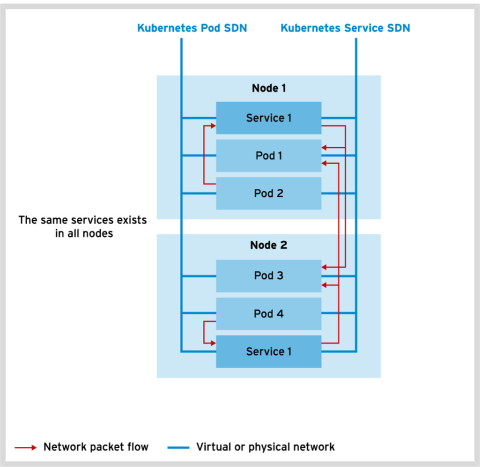


* + - 1. Declares a Kubernetes pod resource type.
    - 2. A unique name for a pod in Kubernetes that allows administrators to run commands on it.
    - 3. Creates a label with a key named name that other resources in Kubernetes, usually as a service, can use to find it.
    - 4. A container-dependent attribute identifying which port on the container is exposed.
    - 5. Defines a collection of environment variables.
    - Some pods may require environment variables that can be read by a container.
    - Kubernetes transforms all the name and value pairs to environment variables. For instance, the MYSQL\_ENV\_MYSQL\_USER variable is declared internally by the Kubernetes runtime with a value of user1, and is forwarded to the container image definition. Because the container uses the same variable name to get the user's login, the value is used by the WildFly container instance to set the username that accesses a MySQL database instance.
  + **Describing Service Resource Definition Syntax**
    - Kubernetes provides a virtual network that allows pods from different workers to connect. But, Kubernetes provides no easy way for a pod to discover the IP addresses of other pods.

****

* + - Services are essential resources to any OpenShift application. They allow containers in one pod to open network connections to containers in another pod.
    - A pod can be restarted for many reasons, and it gets a different internal IP address each time.
    - Instead of a pod having to discover the IP address of another pod after each restart, a service provides a stable IP address for other pods to

use, no matter what worker node runs the pod after each restart.



* + - Most real-world applications do not run as a single pod. They need to scale horizontally, so many pods run the same containers from the same pod resource definition to meet growing user demand.
    - A service is tied to a set of pods, providing a single IP address for the whole set, and a load-balancing client request among member pods.
    - The set of pods running behind a service is managed by a DeploymentConfig resource.
    - A DeploymentConfig resource embeds a ReplicationController that manages how many pod copies replicas) have to be created, and creates new ones if any of them fail.
    - The following example shows a minimal service definition in JSON syntax



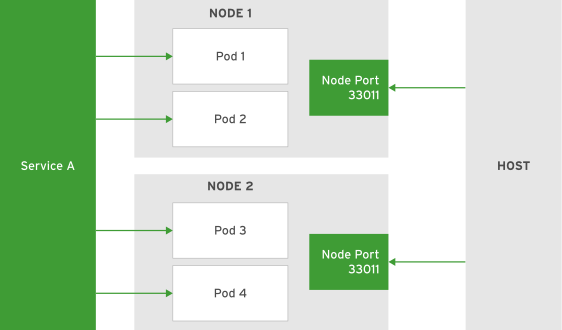
* + - 1. The kind of Kubernetes resource. In this case, a Service.
    - 2. A unique name for the service.
    - 3. ports is an array of objects that describes network ports exposed by the service. The targetPort attribute has to match a containerPort from a pod container definition, and the port attribute is the port that is exposed by the service. Clients connect to the service port and the service forwards packets to the pod targetPort.
    - 4. selector is how the service finds pods to forward packets to. The target pods need to have matching labels in their metadata attributes. If the service finds multiple pods with matching labels, it balances network connections between them.
    - Each service is assigned a unique IP address for clients to connect to. This IP address comes from another internal OpenShift SDN, distinct from the pods' internal network, but visible only to pods. Each pod matching the selector is added to the service resource as an endpoint.
  + **Discovering Services**
    - An application typically finds a service IP address and port by using environment variables. For each service inside an OpenShift project, the following environment variables are automatically defined and injected into containers for all pods inside the same project:
    - SVC\_NAME\_SERVICE\_HOST is the service IP address.
    - SVC\_NAME\_SERVICE\_PORT is the service TCP port.
    - The SVC\_NAME part of the variable is changed to comply with DNS naming restrictions: letters are capitalized and underscores (\_) are replaced by dashes (-)
    - Another way to discover a service from a pod is by using the OpenShift internal DNS server, which is visible only to pods. Each service is dynamically assigned an SRV record with an FQDN of the form:
    - SVC\_NAME.PROJECT\_NAME.svc.cluster.local
    - When discovering services using environment variables, a pod has to be created and started only after the service is created. If the application was written to discover services using DNS queries, however, it can find services created after the pod was started.
    - There are two ways for an application to access the service from outside an OpenShift cluster:
      * **NodePort type:** This is an older Kubernetes-based approach, where the service is exposed to external clients by binding to available ports on the worker node host, which then proxies

connections to the service IP address. Use the oc edit svc command to edit service attributes and specify NodePort as the value for type, and provide a port value for the nodePort attribute. OpenShift then proxies connections to the service via the public IP

address of the worker node host and the port value set in nodePort.

* + - * **OpenShift Routes:** This is the preferred approach in OpenShift to expose services using a unique URL. Use the oc expose command to expose a service for external access or expose

service from the OpenShift web console.



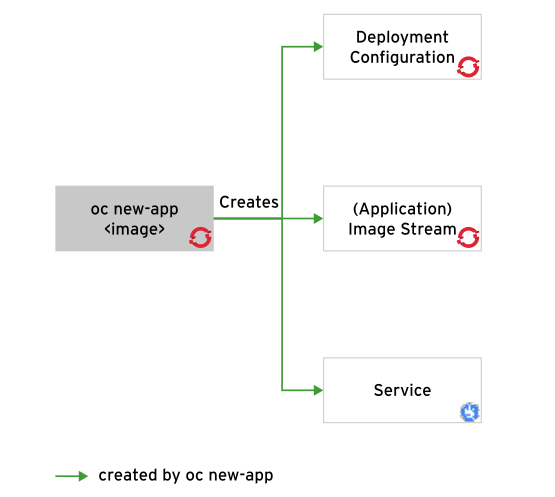
* + - * OpenShift provides the oc port-forward command for forwarding a local port to a pod port. This is different from having access to a pod through a service resource:
        + The port-forwarding mapping exists only on the workstation where the oc client runs, while a service maps a port for all network users.
        + A service load-balances connections to potentially multiple pods, whereas a port-forwarding mapping forwards connections to a single pod.
        + The following example demonstrates the use of the oc port-forward command:

$ oc port-forward mysql-openshift-1-glqrp 3306:3306

* + - * + The previous command forwards port 3306 from the developer machine to port 3306 on the db pod, where a MySQL server (inside a container) accepts network connections.
        + When running this command, make sure you leave the terminal window running. Closing the window or canceling the process stops the port mapping.
  + **Creating New Applications**
    - Simple applications, complex multi tier applications, and microservice applications can be described by a single resource definition file.
    - This single file would contain many pod definitions, service definitions to connect the pods, replication controllers or DeploymentConfigs to horizontally scale the application pods, PersistentVolumeClaims to persist application data, and anything else needed that can be managed by OpenShift.
    - In OpenShift 4.5 the oc new-app command now produces Deployment resources instead of DeploymentConfig resources by default.
    - To create DeploymentConfig resources, you can pass the --as-deployment-config flag when invoking oc new-app,
    - The oc new-app command can be used with the -o json or -o yaml option to create a skeleton resource definition file in JSON or YAML format, respectively.
    - This file can be customized and used to create an application using the oc create -f <filename> command, or merged with other resource definition files to create a composite application.
    - The oc new-app command can create application pods to run on OpenShift in many different ways. It can create pods from existing docker images, from Dockerfiles, and from raw source code using the Source-to-Image (S2I) process.
    - Run the oc new-app -h command to understand all the different options available for creating new applications on OpenShift.
    - The following command creates an application based on an image, mysql, from Docker Hub, with the label set to db=mysql:

$ oc new-app mysql --as-deployment-config \

> MYSQL\_USER=user MYSQL\_PASSWORD=pass MYSQL\_DATABASE=testdb -l db=mysql



* + - The following command creates an application based on an image from a private Docker image registry

oc new-app --docker-image=myregistry.com/mycompany/myapp --name=myapp --as-

deployment-config

* + - The following command creates an application based on source code stored in a Git repository:

oc new-app <https://github.com/openshift/ruby-hello-world> --name=ruby-hello --as-

deployment-config

* + **Managing OpenShift Resources at the Command Line**
    - Use the oc get command to retrieve information about resources in the cluster. Generally, this command outputs only the most important characteristics of the resources and omits more detailed information.
    - The oc get RESOURCE\_TYPE command displays a summary of all resources of the specified type.
    - **oc get all:**
      * Use the oc get all command to retrieve a summary of the most important components of a cluster. This command iterates through the major resource types for the current project and prints

out a summary of their information.

* + - **oc describe RESOURCE\_TYPE RESOURCE\_NAME**
      * If the summaries provided by oc get are insufficient, use the oc describe command to retrieve additional information.
    - **oc get RESOURCE\_TYPE RESOURCE\_NAME -o yaml**
      * This command can be used to export a resource definition. Typical use cases include creating a backup, or to aid in the modification of a definition. The -o yaml option prints out the object representation in YAML format, but this can be changed to JSON format by providing a -o json option.
    - **oc create**
      * This command creates resources from a resource definition. Typically, this is paired with the oc get RESOURCE\_TYPE RESOURCE\_NAME -o yaml command for editing definitions.
    - **oc edit**
      * This command allows the user to edit resources of a resource definition. By default, this command opens a vi buffer for editing the resource definition.
    - **oc delete RESOURCE\_TYPE name**
      * The oc delete command removes a resource from an OpenShift cluster.
    - **oc exec CONTAINER\_ID options command**
      * The oc exec command executes commands inside a container. You can use this command to run interactive and noninteractive batch commands as part of a script.
  + **Labelling resources**
    - When working with many resources in the same project, it is often useful to group those resources by application, environment, or some other criteria.
    - To establish these groups, you define labels for the resources in your project. Labels are part of the metadata section of a resource, and are

defined as key/value pairs, as shown in the following example:

****

* + - Many oc subcommands support a -l option to process resources from a label specification. For the oc get command, the -l option acts as a selector to only retrieve objects that have a matching label.

$ oc get svc,dc -l app=nexus

* + - When using templates to generate resources, labels are especially useful. A template resource has a labels section separated from the metadata.labels section. Labels defined in the labels section do not apply to the template itself, but are added to every resource generated by the

template.



* + - The previous example defines a template resource with a single label: maintainer: redhat. The template generates a service resource with three labels: app: nexus, template: nexus-persistent-template, and version: 1
  + **Example: Deploying a Database Server on OpenShift**
    - Log in to the OpenShift cluster.
      * oc login -u ${RHT\_OCP4\_DEV\_USER} -p \

> ${RHT\_OCP4\_DEV\_PASSWORD} ${RHT\_OCP4\_MASTER\_API}

* + - * RHT\_OCP4\_DEV\_USER , RHT\_OCP4\_DEV\_PASSWORD and RHT\_OCP4\_MASTER\_API is variable configure before running above command.
    - Create a new project that contains your RHOCP developer username

$ oc new-project ${RHT\_OCP4\_DEV\_USER}-mysql-openshift

* + - Create a new application from the rhscl/mysql-57-rhel7 container image using the oc new-app command. This image requires that you use the -e option to set the MYSQL\_USER, MYSQL\_PASSWORD, MYSQL\_DATABASE, and MYSQL\_ROOT\_PASSWORD environment variables.

$ oc new-app --as-deployment-config \

> --docker-image=registry.access.redhat.com/rhscl/mysql-57-rhel7:latest \

> --name=mysql-openshift \

> -e MYSQL\_USER=user1 -e MYSQL\_PASSWORD=mypa55 -e MYSQL\_DATABASE=testdb \

> -e MYSQL\_ROOT\_PASSWORD=r00tpa55

* + - Run the oc status command to view the status of the new application and verify that the deployment of the MySQL image was successful

$ oc status

* + - List the pods in this project to verify that the MySQL pod is ready and running

$ oc get pods

* + - Use the oc describe command to view more details about the pod

oc describe pod mysql-openshift-1-xg665

* + - List the services in this project and verify that the service to access the MySQL pod was created

$ oc get svc

* + - Retrieve the details of the mysql-openshift service using the oc describe

command and note that the service type is ClusterIP by default

$ oc describe service mysql-openshift

* + - View details about the deployment configuration (dc) for this application

$ oc describe dc mysql-openshift

* + - Expose the service creating a route with a default name and a fully qualified domain name (FQDN)

$ oc expose service mysql-openshift

route.route.openshift.io/mysql-openshift exposed

$ oc get routes

* + - Connect to the MySQL database server and verify that the database was created successfully.

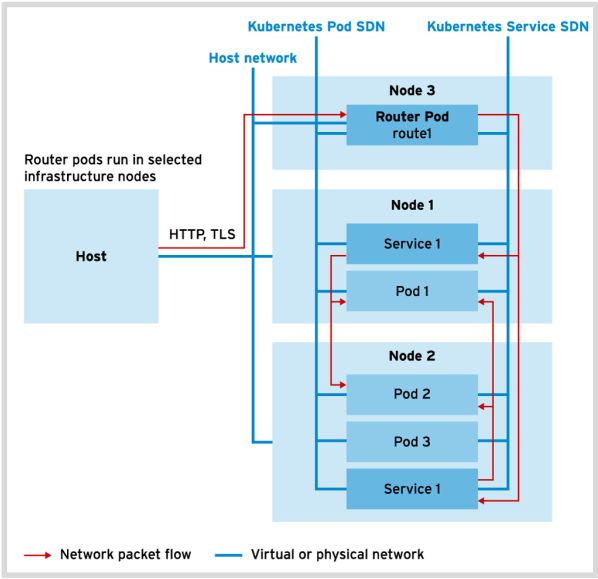
$ oc port-forward mysql-openshift-1-xg665 3306:3306

New Tab

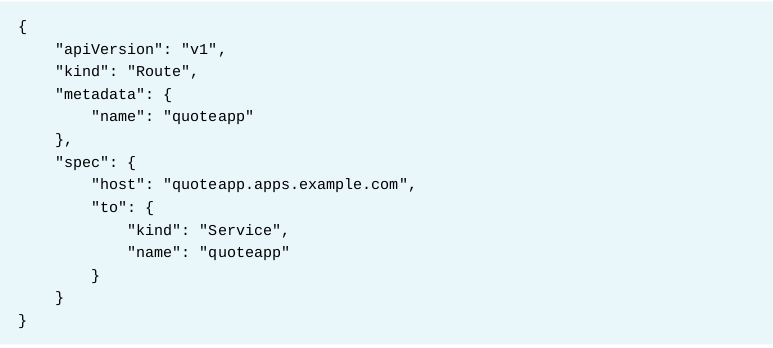
mysql -uuser1 -pmypa55 --protocol tcp -h localhost

MySQL [(none)]> show databases;

* **Creating Routes:**
  + **Working with Routes**
    - Services allow for network access between pods inside an OpenShift instance, and routes allow for network access to pods from users and applications outside the OpenShift instance.



* + - A route connects a public-facing IP address and DNS host name to an internal-facing service IP. It uses the service resource to find the endpoints; that is, the ports exposed by the service.
    - OpenShift routes are implemented by a cluster-wide router service, which runs as a containerized application in the OpenShift cluster. OpenShift scales and replicates router pods like any other OpenShift application.
    - The router service uses HAProxy as the default implementation.
    - An important consideration for OpenShift administrators is that the public DNS host names configured for routes need to point to the public-facing IP addresses of the nodes running the router. Router pods, unlike regular application pods, bind to their nodes' public IP addresses instead of to the internal pod SDN.
    - The following example shows a minimal route defined using JSON syntax:



* + - The apiVersion, kind, and metadata attributes follow standard Kubernetes resource definition rules. The Route value for kind shows that this is a route resource, and the metadata.name attribute gives this particular route the identifier quoteapp.
    - As with pods and services, the main part is the spec attribute, which is an object containing the following attributes:
      * host is a string containing the FQDN associated with the route. DNS must resolve this FQDN to the IP address of the OpenShift router. The details to modify DNS configuration are outside the

scope of this course

* + - * to is an object stating the resource this route points to. In this case, the route points to an OpenShift Service with the name set to quoteapp.
    - Names of different resource types do not collide. It is perfectly legal to have a route named quoteapp that points to a service also named quoteapp.
    - Unlike services, which use selectors to link to pod resources containing specific labels, a route links directly to the service resource name.
  + **Creating Routes**
    - oc create command to create route resources, just like any other OpenShift resource.
    - You must provide a JSON or YAML resource definition file, which defines the route, to the oc create command.
    - The oc new-app command does not create a route resource when building a pod from container images, Dockerfiles, or application source code. After all, oc new-app does not know if the pod is intended to be accessible from outside the OpenShift instance or not.
    - Another way to create a route is to use the oc expose service command, passing a service resource name as the input. The --name option can be used to control the name of the route resource. For example:

$ oc expose service quotedb --name quote

* + - By default, routes created by oc expose generate DNS names of the form:

route-name-project-name.default-domain

* + - route-name is the name assigned to the route. If no explicit name is set, OpenShift assigns the route the same name as the originating resource (for example, the service name).
    - project-name is the name of the project containing the resource.
    - default-domain is configured on the OpenShift master and corresponds to the wildcard DNS domain listed as a prerequisite for installing OpenShift.
    - For example, creating a route named quote in a project named test from an OpenShift instance where the wildcard domain is cloudapps.example.com results in the FQDN quote-test.cloudapps.example.com.
    - The DNS server that hosts the wildcard domain knows nothing about route host names. It merely resolves any name to the configured IP addresses. Only the OpenShift router knows about route host names, treating each one as an HTTP virtual host. The OpenShift router blocks invalid wildcard domain host names that do not correspond to any route and returns an HTTP 404 error.
  + **Leveraging the Default Routing Service**
    - The default routing service is implemented as an HAProxy pod. Router pods, containers, and their configuration can be inspected just like any other resource in an OpenShift cluster

$ oc get pod --all-namespaces -l app=router

* + - oc describe pod command to get the routing configuration details

$ oc describe pod router-default-746b5cfb65-f6sdm

* + **Example: Exposing a Service as a Route**
    - Log in to the OpenShift cluster.

$ oc login -u ${RHT\_OCP4\_DEV\_USER} -p \

> ${RHT\_OCP4\_DEV\_PASSWORD} ${RHT\_OCP4\_MASTER\_API}

* + - Create a new project that contains your RHOCP developer username for the resources

$ oc new-project ${RHT\_OCP4\_DEV\_USER}-route

* + - Create a new PHP application using Source-to-Image from the php-helloworld directory in the Git repository at http://github.com/${RHT\_OCP4\_GITHUB\_USER}/DO180-apps/

$ oc new-app --as-deployment-config \

> php:7.3~https://github.com/${RHT\_OCP4\_GITHUB\_USER}/DO180-apps \

> --context-dir php-helloworld --name php-helloworld

* + - Wait until the application finishes building and deploying by monitoring the progress with the oc get pods -w command

$ oc get pods -w

* + - Alternatively, monitor the build and deployment logs with the oc logs -f

bc/php-helloworld and oc logs -f dc/php-helloworld commands,

respectively. Press Ctrl+C to exit the command if necessary.

$ oc logs -f bc/php-helloworld

* + - Review the service for this application using the oc describe command

$ oc describe svc/php-helloworld

* + - Expose the service, which creates a route. Use the default name and fully qualified domain name (FQDN) for the route

$ oc expose svc/php-helloworld

$ oc describe route

* + - Access the service from a host external to the cluster to verify that the service and route are working

$ curl \

> php-helloworld-${RHT\_OCP4\_DEV\_USER}-route.${RHT\_OCP4\_WILDCARD\_DOMAIN}

* + - Replace this route with a route named xyz
      * Delete the current route.

$ oc delete route/php-helloworld

* + - * Create a route for the service with a name of ${RHT\_OCP4\_DEV\_USER}-xyz.

$ oc expose svc/php-helloworld \

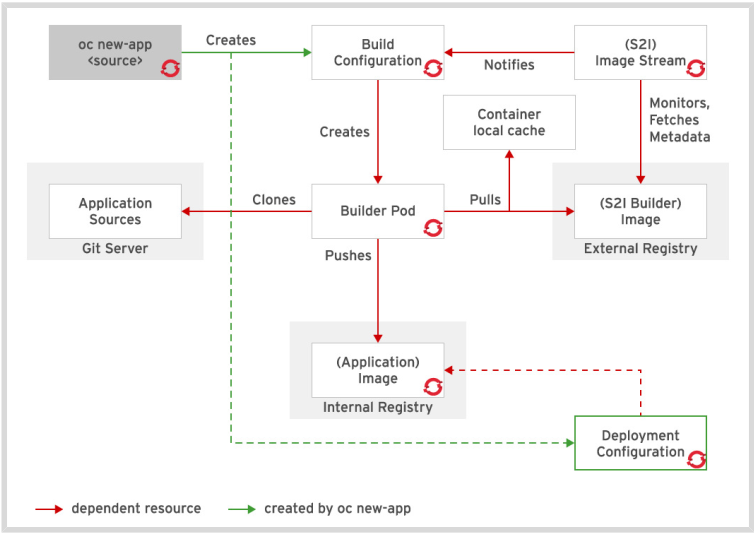
> --name=${RHT\_OCP4\_DEV\_USER}-xyz

* + - Make an HTTP request using the FQDN on port 80

$ curl \

> ${RHT\_OCP4\_DEV\_USER}-xyz-${RHT\_OCP4\_DEV\_USER}-route.${RHT\_OCP4\_WILDCARD\_DOMAIN}

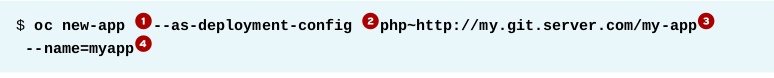
* **Creating Application with Source-to-Image**
  + Source-to-Image (S2I) is a tool that makes it easy to build container images from application source code. This tool takes an application's source code from a Git repository, injects the source code into a base container based on the language and framework desired, and produces a new container image that runs the assembled application.



* + S2I is the primary strategy used for building applications in the OpenShift Container Platform. The main reasons for using source builds are:
  + **User efficiency:** Developers do not need to understand Dockerfiles and operating system commands such as yum install. They work using their standard programming language tools.
  + **Patching:** S2I allows for rebuilding all the applications consistently if a base image needs a patch due to a security issue. For example, if a security issue is found in a PHP base image, then updating this image with security patches updates all applications that use this image as a base.
  + **Speed**: With S2I, the assembly process can perform a large number of complex operations without creating a new layer at each step, resulting in faster builds
  + **Ecosystem**: S2I encourages a shared ecosystem of images where base images and scripts can be customized and reused across multiple types of applications.
  + **Describing Image Streams**
    - OpenShift deploys new versions of user applications into pods quickly. To create a new application, in addition to the application source code, a base image (the S2I builder image) is required. If either of these two components gets updated, OpenShift creates a new container image. Pods created using the older container image are replaced by pods using the new image.
    - Even though it is evident that the container image needs to be updated when application code changes, it may not be evident that the deployed pods also need to be updated should the builder image change.
    - The image stream resource is a configuration that names specific container images associated with image stream tags, an alias for these container images. OpenShift builds applications against an image stream. The OpenShift installer populates several image streams by default during installation.

$ oc get is -n openshift

* + - OpenShift detects when an image stream changes and takes action based on that change. If a security issue arises in the nodejs-010-rhel7 image, it can be updated in the image repository, and OpenShift can automatically trigger a new build of the application code.
    - It is likely that an organization chooses several supported base S2I images from Red Hat, but may also create their own base images.
  + **Building an Application with S2I and the CLI**
    - Building an application with S2I can be accomplished using the OpenShift CLI.
    - An application can be created using the S2I process with the oc new-app command from the CLI.



* + - 1 Create a DeploymentConfig instead of a Deployment
    - 2 The image stream used in the process appears to the left of the tilde (~).
    - 3 The URL after the tilde indicates the location of the source code's Git repository.
    - 4 Sets the application name.
  + Instead of using the tilde, you can set the image stream by using the -i option.

$ oc new-app --as-deployment-config -i php http://services.lab.example.com/app

--name=myapp

* + The oc new-app command allows creating applications using source code from a local or remote Git repository.f only a source repository is specified, oc new-app tries to identify the correct image stream to use for building the application. In addition to application code, S2I can also identify and process Dockerfiles to create a new image.

$ oc new-app --as-deployment-config .

* + When using a local Git repository, the repository must have a remote origin that

points to a URL accessible by the OpenShift instance.

* + It is also possible to create an application using a remote Git repository and a context subdirectory

$ oc new-app --as-deployment-config \

https://github.com/openshift/sti-ruby.git \

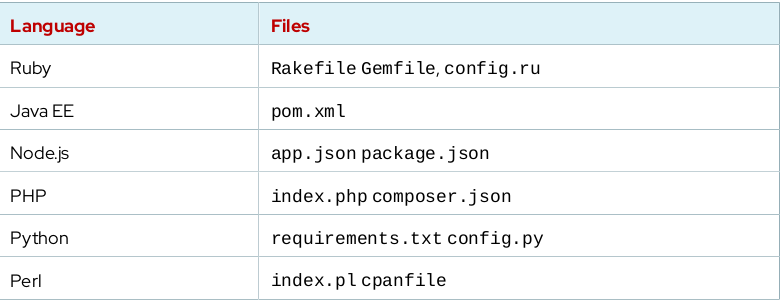
--context-dir=2.0/test/puma-test-app

* + Finally, it is possible to create an application using a remote Git repository with a specific branch reference

$ oc new-app --as-deployment-config \

https://github.com/openshift/ruby-hello-world.git#beta4

* + If an image stream is not specified in the command, new-app attempts to determine which language builder to use based on the presence of certain files in the root of the repository



* + After a language is detected, the new-app command searches for image stream tags that have support for the detected language, or an image stream that matches the name of the detected language.
  + Create a JSON resource definition file by using the -o json parameter and output redirection

$ oc -o json new-app --as-deployment-config \

> php~http://services.lab.example.com/app \

> --name=myapp > s2i.json

* + This JSON definition file creates a list of resources. The first resource is the image stream



* + 1 Define a resource type of image stream.
  + 2 Name the image stream myapp.
  + The build configuration (bc) is responsible for defining input parameters and triggers that are executed to transform the source code into a runnable image. The BuildConfig (BC) is the second resource





* + 1 Define a resource type of BuildConfig.
  + 2 Name the BuildConfig myapp.
  + 3 Define the address to the source code Git repository.
  + 4 Define the strategy to use S2I.
  + 5 Define the builder image as the php:7.3 image stream.
  + 6 Name the output image stream myapp:latest.
  + The third resource is the deployment configuration that is responsible for customizing the deployment process in OpenShift. It may include parameters and triggers that are necessary to create new container instances, and are translated into a replication controller from Kubernetes. Some of the features provided by DeploymentConfig objects are
    - User customizable strategies to transition from the existing deployments to new deployments.
    - Rollbacks to a previous deployment.
    - Manual replication scaling.







* + 1 Define a resource type of DeploymentConfig.
  + 2 Name the DeploymentConfig myapp.
  + 3 A configuration change trigger causes a new deployment to be created any time the replication controller template changes.
  + 4 An image change trigger causes the creation of a new deployment each time a new version of the myapp:latest image is available in the repository.
  + 5 Define the container image to deploy: myapp:latest.
  + 6 Specifies the container ports.
  + After creating a new application, the build process starts. Use the oc get builds command to see a list of application builds

$ oc get builds

* + OpenShift allows viewing the build logs. The following command shows the last few lines of the build log

$ oc logs build/myapp-1

* + If the build is not Running yet, or OpenShift has not deployed the s2i-build pod

Yet, the above command throws an error. Just wait a few moments and retry it.

* + Trigger a new build with the oc start-build build\_config\_name command

$ oc get buildconfig

$ oc start-build myapp

* + **Relationship Between Build and Deployment Configurations**
    - The BuildConfig pod is responsible for creating the images in OpenShift and pushing them to the internal container registry. Any source code or content update typically requires a new build to guarantee the image is updated
    - The DeploymentConfig pod is responsible for deploying pods to OpenShift.
    - The outcome of a DeploymentConfig pod execution is the creation of pods with the images deployed in the internal container registry. Any existing running pod may be destroyed, depending on how the

DeploymentConfig resource is set.

* + - The BuildConfig and DeploymentConfig resources do not interact directly. The BuildConfig resource creates or updates a container image. The DeploymentConfig reacts to this new image or updated image event and creates pods from the container image.
  + **Example: Creating a Containerized Application with Source-to-Image**
    - Inspect the PHP source code for the sample application and create and push a new branch named s2i.
      * Enter your local clone of the DO180-apps Git repository and checkout the master branch of the course's repository

$ cd ~/DO180-apps

git checkout master

* + - * Create a new branch to save any changes

$ git checkout -b s2i

$ git push -u origin s2i

* + - * Review the code
    - Log in to the OpenShift cluster.

$ oc login -u ${RHT\_OCP4\_DEV\_USER} -p \

> ${RHT\_OCP4\_DEV\_PASSWORD} ${RHT\_OCP4\_MASTER\_API}

* + - Create a new project that contains your RHOCP developer username

$ oc new-project ${RHT\_OCP4\_DEV\_USER}-s2i

* + - Create a new PHP application using Source-to-Image from the php-helloworld directory using the s2i branch
      * Use the oc new-app command to create the PHP application.

$ oc new-app --as-deployment-config php:7.3 \

> --name=php-helloworld \

> https://github.com/${RHT\_OCP4\_GITHUB\_USER}/DO180-apps#s2i \

> --context-dir php-helloworld

* + - * Verify that the build process starts with the oc get pods command

$ oc get pods

* + - * Examine the logs for this build. Use the build pod name for this build, php-helloworld-1-build

$ oc logs --all-containers \

> -f php-helloworld-1-build

* + - * Review the DeploymentConfig for this application

$ oc describe dc/php-helloworld

* + - * Add a route to test the application

$ oc expose service php-helloworld \

> --name ${RHT\_OCP4\_DEV\_USER}-helloworld

* + - * Find the URL associated with the new route

$ oc get route -o jsonpath='{..spec.host}{"\n"}'

${RHT\_OCP4\_DEV\_USER}-helloworld-${RHT\_OCP4\_DEV\_USER}-s2i.

${RHT\_OCP4\_WILDCARD\_DOMAIN}

* + - * Test the application by sending an HTTP GET request to the URL

$ curl -s \

> ${RHT\_OCP4\_DEV\_USER}-helloworld-${RHT\_OCP4\_DEV\_USER}-s2i.\

> ${RHT\_OCP4\_WILDCARD\_DOMAIN}

* + - Explore starting application builds by changing the application in its Git repository and executing the proper commands to start a new Source-to-Image build.
      * Change the file
      * Commit the changes and push the code back to the remote Git repository

$ git add .

$ git commit -m 'Changed index page contents.'

$ git push origin s2i

* + - * Start a new Source-to-Image build process and wait for it to build and deploy

$ oc start-build php-helloworld

* + - * Test that the application serves the new content

$ curl -s \

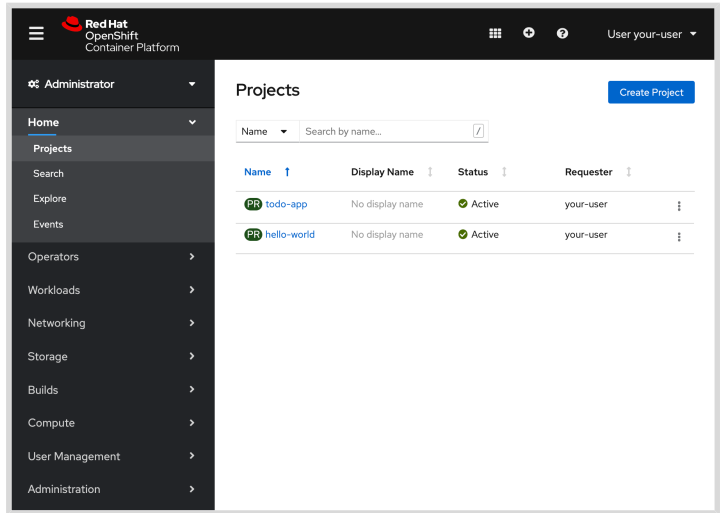
> ${RHT\_OCP4\_DEV\_USER}-helloworld-${RHT\_OCP4\_DEV\_USER}-s2i.\

> ${RHT\_OCP4\_WILDCARD\_DOMAIN}

* + **Creating Applications with the OpenShift Web Console**
    - **Accessing the OpenShift Web Console**
      * The OpenShift web console allows users to execute many of the same tasks as the OpenShift command-line client. You can create projects, add applications to projects, view application resources, and manipulate application configurations as needed. The OpenShift web console runs as one or more pods, each pod running on a master node.
      * The default URL is of the format https://console- openshift-console.{wildcard DNS domain for the RHOCP cluster}/
      * The web console uses a REST API to communicate with the OpenShift cluster. By default, the REST API endpoint is accessed with a different DNS name and self-signed certificate. You must

also trust this certificate for the REST API endpoint.

* + - * After you have trusted the two OpenShift certificates, the console requires authentication to proceed.
      * **Managing Projects**

****

* + - * + Upon successful login, the Home → Projects page displays a list of projects you can access. From

this page you can create, edit, or delete a project.

* + - * + If you have permission to view cluster metrics, then you are instead redirected to the Home →Overview page.
        + This page shows general information and metrics about the cluster. The Overview menu item is hidden from users without authority to view cluster metrics.
      * **Navigating the Web Console**
        + A navigation menu is located on the left side of the web console. The menu includes two perspectives: Administrator and Developer. Each item in the menu expands to provide access to a set of related management functions. When the Administrator perspective is selected, these items are as follows
        + **Home:**The home menu allows users to quickly access projects and resources. From this menu, you

can browse and manage projects, search or explore cluster resources, and inspect cluster

events.

* + - * + **Operators:**The OperatorHub is a catalog that allows you to discover, search and install operators in the

cluster. After installing an operator, you can use the Installed Operators option to manage the operator or search for other installed operators.

* + - * + **Workloads:**These options enable management of several types of Kubernetes and OpenShift resources,

such as pods and deployment configurations. Other advanced deployment options that are accessible from this menu, such as configuration maps, secrets, and cron jobs, are beyond the scope of the course.

* + - * + **Storage**: This menu contains options to configure persistent storage for project applications. In particular, persistent volumes and persistent volume claims for a project are managed from the Storage menu.
        + **Builds:**The Build Configs option displays a list of project build configurations. Click a build

configuration link in this view to access an overview page for the specified build configuration. From this page, you can view and edit the application's build configuration.

The Builds option provides a list of recent build processes for application container images in the project. Click the link for a particular build to access the build logs for that particular build process.

The Image Streams option provides a list of image streams defined in the project. Click an image stream entry in this list to access an overview page to view and manage that image stream.

* + - * + **Compute**

Provides options to access and manage the compute nodes of the OpenShift cluster. From this page, you can also configure node health checks and autoscaling.

* + - * + **User management:** From this option, you can configure authentication and authorization using users, groups,

roles, role bindings and service accounts.

* + - * + **Administration:** Provides options to manage cluster and project settings, such as resource quotas and role-based access controls. Functions in the Administration section are outside the scope of this

course.

* + - * **Creating New Applications**
        + From the Developer perspective, use +Add to select a way to create a new application in an OpenShift project.
        + You can add an application from the Developer Catalog, which offers a selection of Source-to-Image (S2I) templates, builder images and Helm charts to create

technology-specific applications. Select a desired template, and provide the necessary information to deploy the new application.

* + - * + You are not limited to deploying an application from the catalog. You can also deploy an application using:

A container image hosted on a remote container registry.

A YAML file that specifies the Kubernetes and OpenShift resources to create.

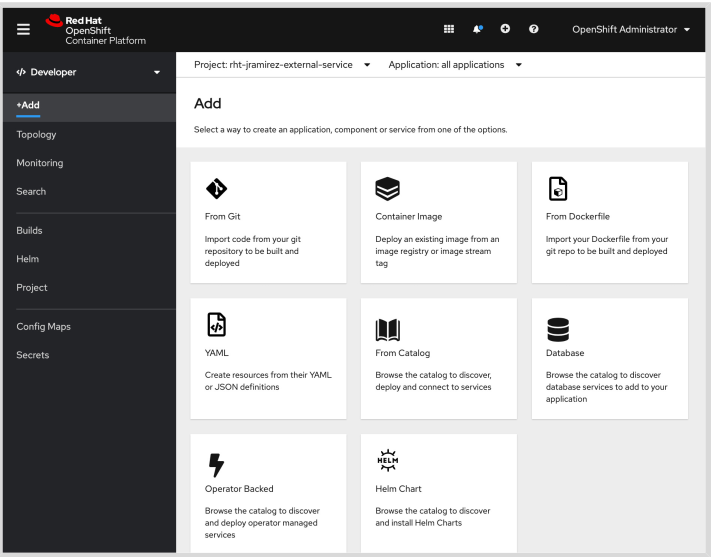
A builder image using the source code from your own git repository.

Dockerfile.

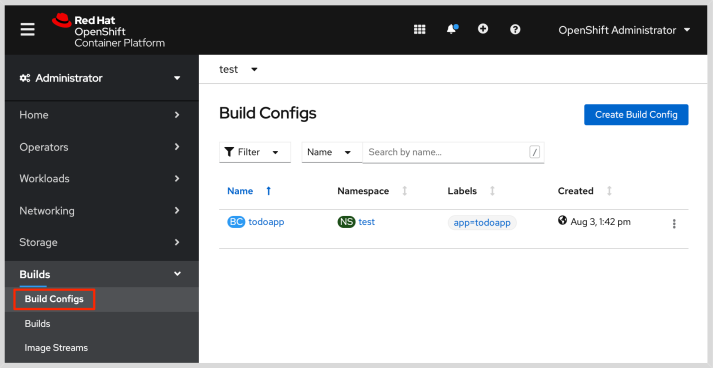
* + - * + To create an application with one of these methods, select the appropriate option in the Add page.
        + Use the Container Image option to deploy an existing container image.
        + Use the YAML option to create the resources specified in a YAML file. Use the From Git option to deploy your source

code using a builder image.

* + - * + Use the From Dockerfile option to specify and build the image from a specific Dockerfile.
        + The Databases, Operator Backed and Helm Chart options are shortcuts to the catalog.



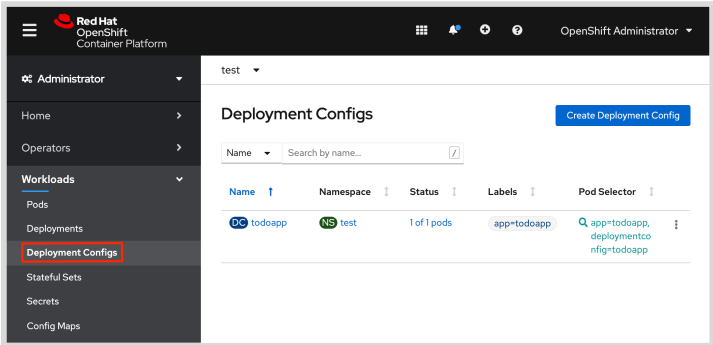
* + - * **Managing Application Builds**
        + From the Administrator perspective, click the Build Configs option of the Builds menu after you add a Source-to-Image application to a project. The new build configuration is accessible from this view

****

* + - * + Click a build configuration in the list to view an overview page for the selected build configuration.

From the overview page, you can

* + - * + View the build configuration parameters, such as the URL for the source code's Git repository.
        + View and edit the environment variables that are set in the builder container, during an application build process.
        + View a list of recent application builds, and click a selected build to access logs from the build process.
      * **Managing Deployed Applications**
        + The Workloads menu provides access to deployment configurations in the project.

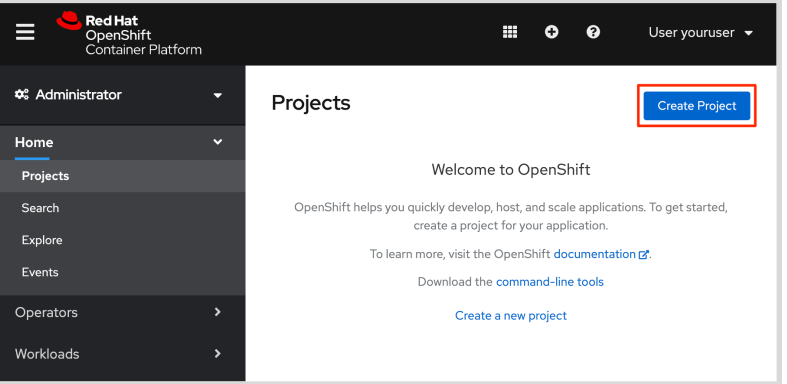


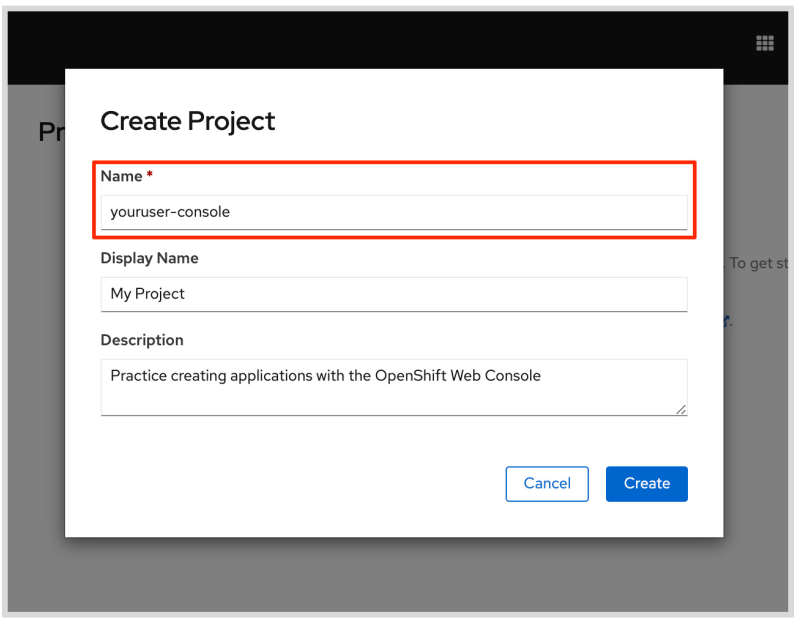
* + - * + Click a deployment configuration entry in the list to view an overview page for the selection. From the overview page, you can
        + View the deployment configuration parameters, such as the specifications of an application container image.
        + Change the desired number of application pods to manually scale the application.
        + View and edit the environment variables that are set in the deployed application container.
        + View a list of application pods, and click a selected pod to access logs for that pod.
      * **Other Web Console Features**
        + Manage resources, such as project quotas, user membership, secrets, and other advanced resources.
        + Create persistent volume claims.
        + Monitor builds, deployments, pods, and system events.
        + Create continuous integration and deployment pipelines with Jenkins.
  + **Example: Creating an Application with the Web Console**
    - Inspect the PHP source code for the sample application and create and push a new branch named console
    - $ cd ~/DO180-apps
    - $ git checkout master
    - Create a new branch to save any changes
    - $ git checkout -b console
    - $ git push -u origin console
    - Review the PHP source code of the application, inside the the php-helloworld folder.
    - Open a web browser and navigate to https://console-openshift-console.

${RHT\_OCP4\_WILDCARD\_DOMAIN} to access the OpenShift web console. Log in and create a new project named youruser-console.

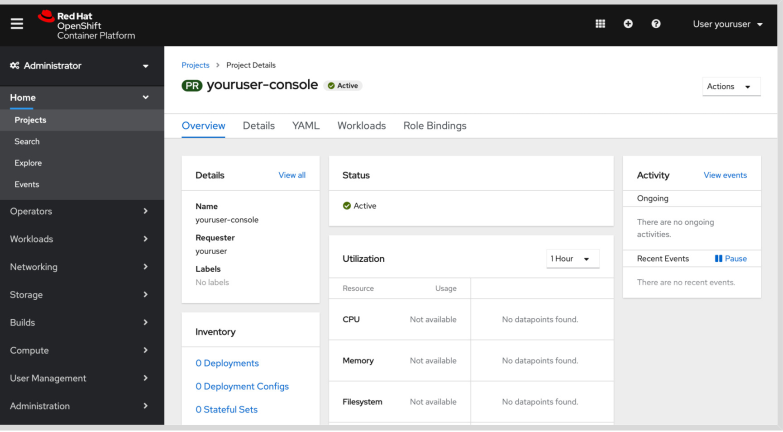
* + - $ echo $RHT\_OCP4\_WILDCARD\_DOMAIN

apps.cluster.lab.example.com

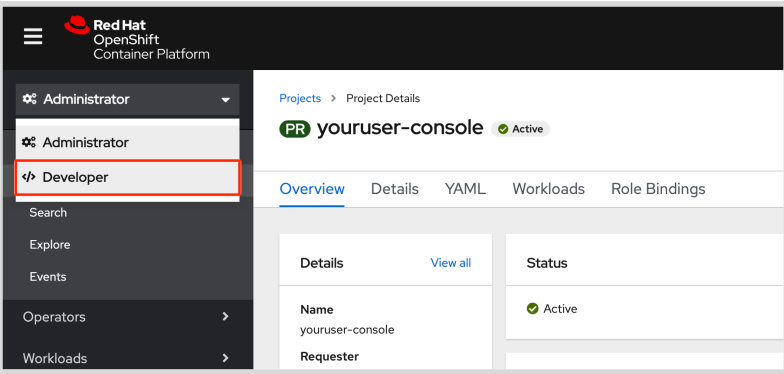




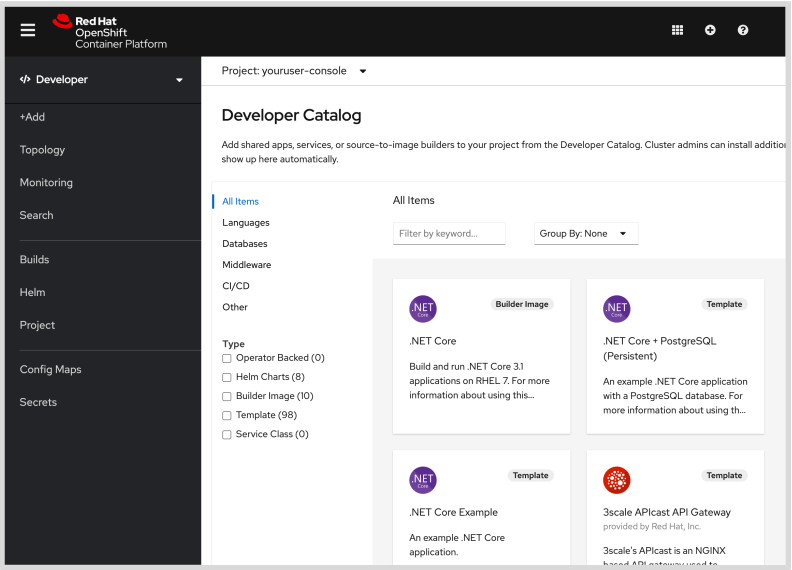
* + - After you have completed the required fields, click Create in the Create Project dialog box to go to the Project Status page for the youruser-console project



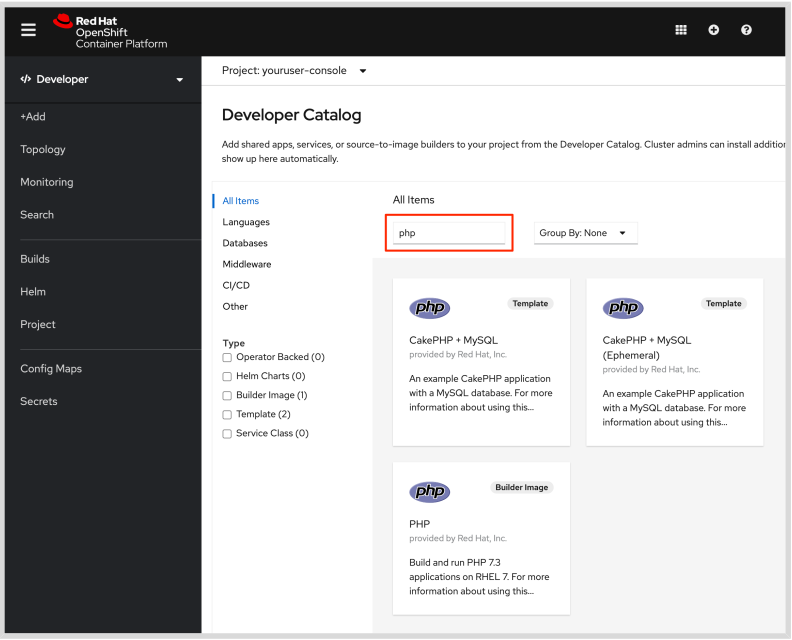
* + - Create the new php-helloworld application with a PHP template
      * Switch to the Developer perspective using the drop-down at the top of the left-hand menu



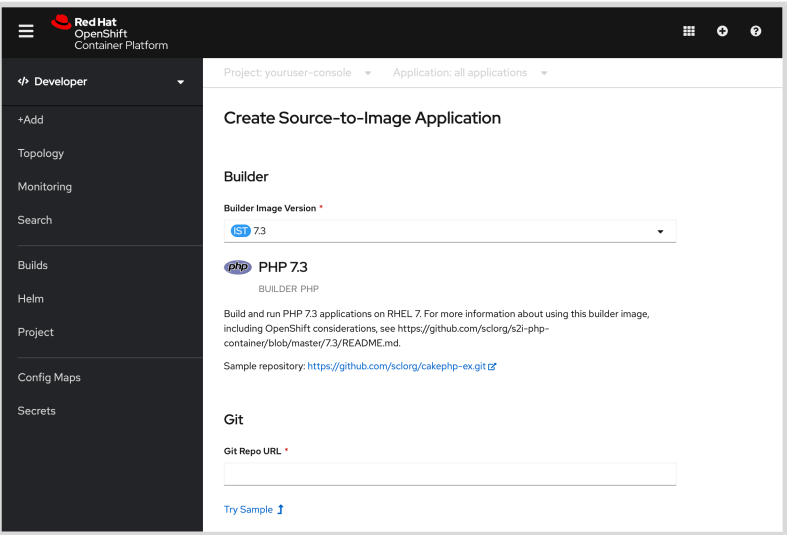
* + - * Click From Catalog to display a list of technology templates. Uncheck the Operator Backed checkbox to see all of the templates.



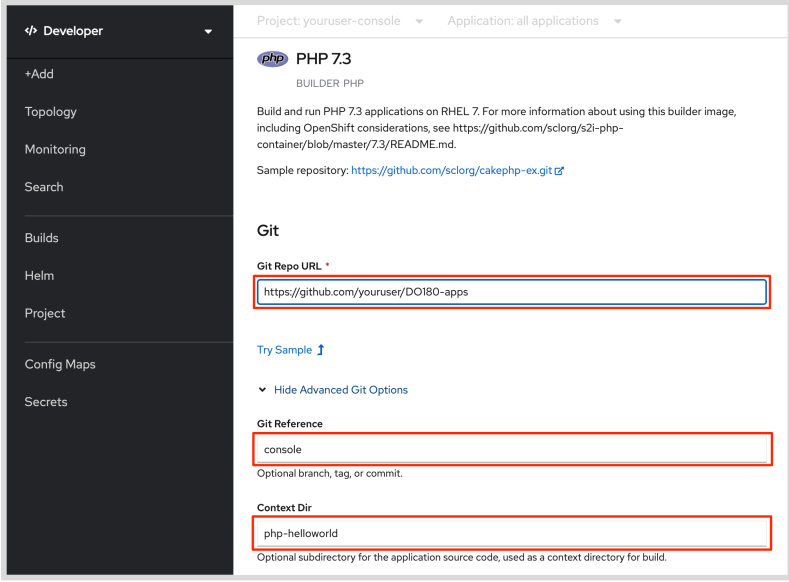
* + - * Enter php in the Filter by keyword field.



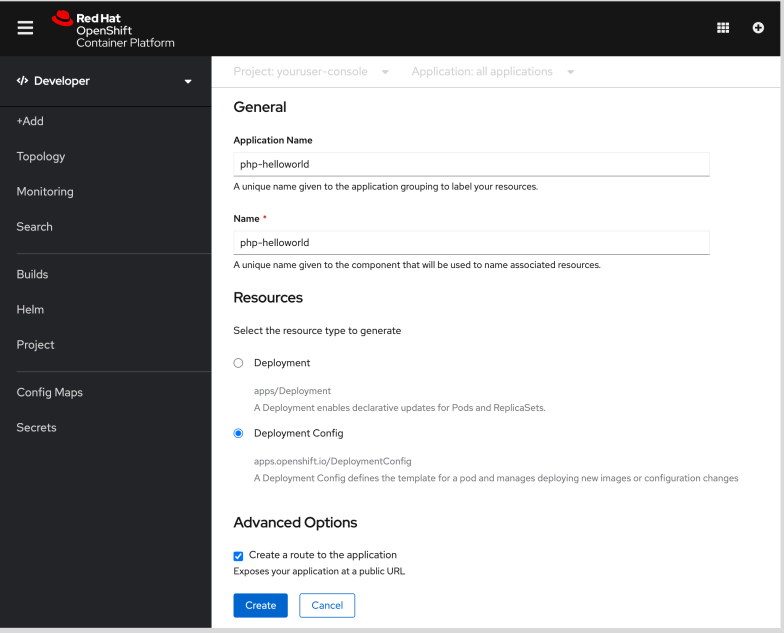
* + - * After filtering, click the PHP builder image to display the PHP dialog box. Click Create Application to display the Create Source-to-Image Application page.



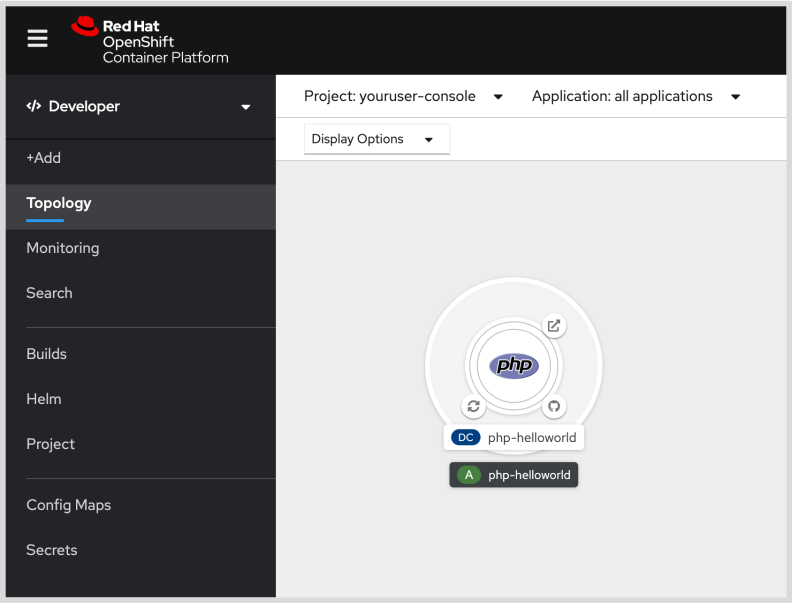
* + - * Change the Builder Image Version to PHP version 7.3.
      * Specify the location of the source code git repository: https://
      * github.com/yourgituser/DO180-apps.
      * Use the Advanced Git Options to set the context directory to php-helloworld and branch console



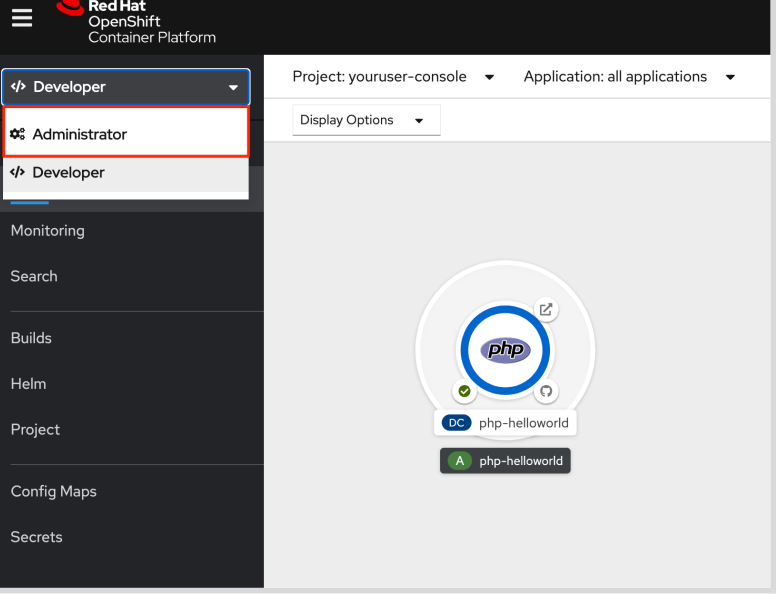
* + - * Enter php-helloworld for both the application name and the name used for associated resources. Select Deployment Config as the resource type.



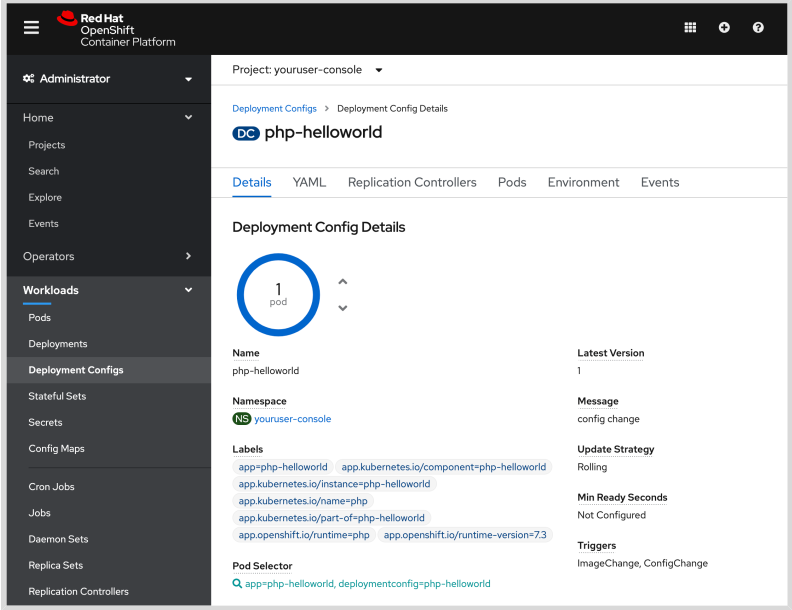
* + - * Scroll to the bottom of the page, and select Create a route to the application. Click Create to create the required OpenShift and Kubernetes resources for the application.
      * You are redirected to the Topology page



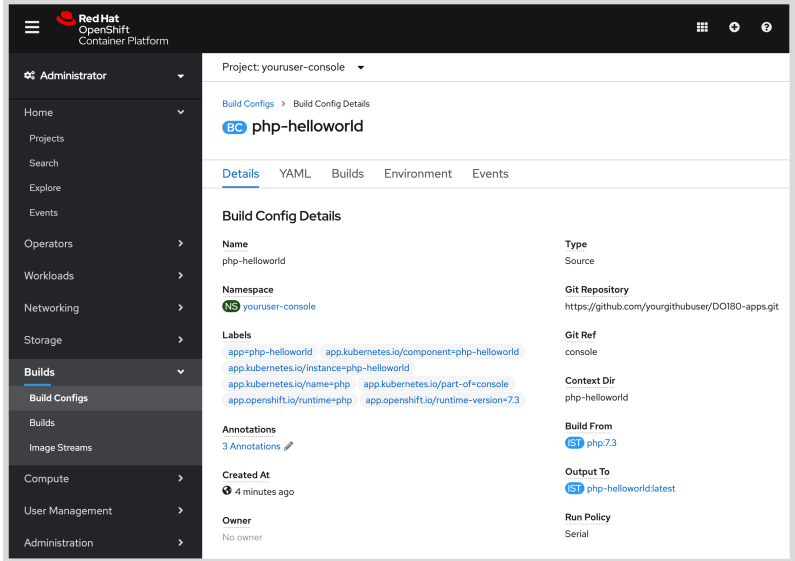
* + - * This page indicates that the php-helloworld application is created. The DC annotation to the left of the php-helloworld link is an acronym for Deployment Config. This link redirects to a page containing information about the application's deployment configuration.
      * Switch back to the Administrator perspective



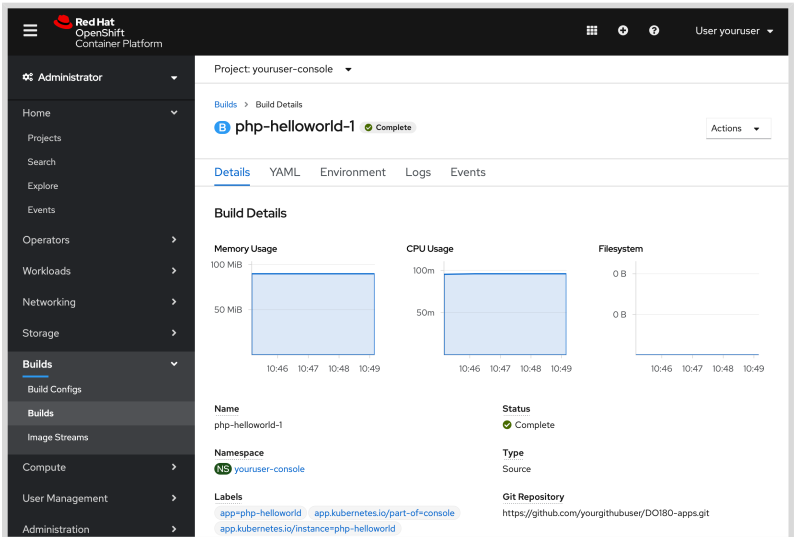
* + - Use the navigation bar on the left side of the OpenShift web console to locate information for the application's OpenShift and Kubernetes resources
      * DeploymentConfig
      * BuildConfig
      * Build Logs
      * Service
      * Route
      * Examine the deployment configuration. In the navigation bar, click Workloads to reveal more menu choices. Click Deployment Configs to display a list of deployment configurations for the youruser-console project. Click the php-helloworld link to display deployment configuration information.



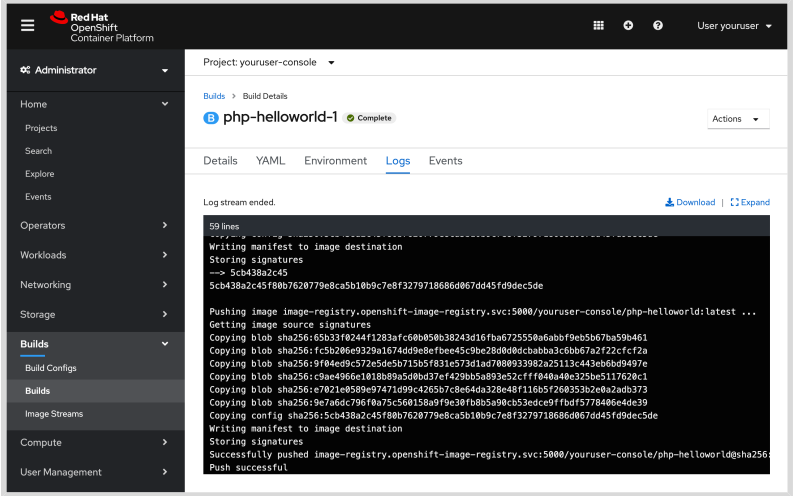
* + - * Explore the available information from the Details tab. The build may still be running when you reach this page, so the DC might not have a value of 1 pod, yet.
      * If you click the up and down arrow icons next to the doughnut chart that indicates the number of pods, you can scale the application up and down horizontally.
      * Examine the build configuration. In the navigation bar, click Builds to reveal more menu choices. Click Build Configs to display a list of build configurations for the youruser-console project. Click the php-helloworld link to display the build configuration for the application.



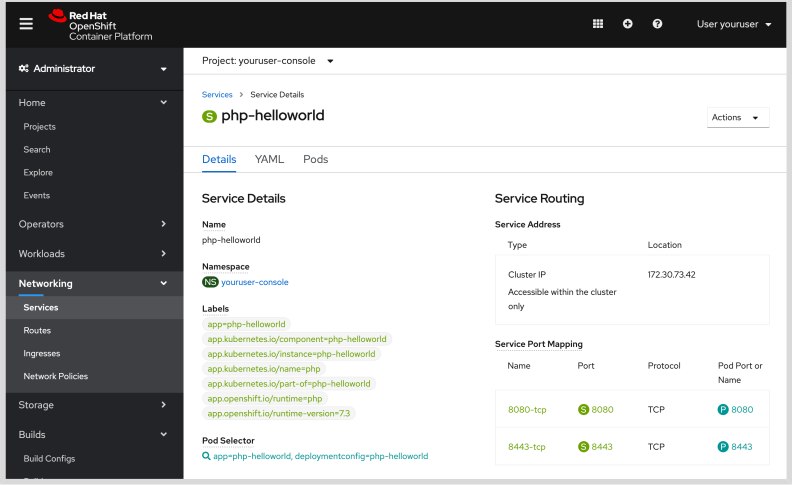
* + - * Explore the available information from the Details tab. The YAML tab allows you to view and edit the build configuration as a YAML file. The Builds tab provides an historical list of builds, along with a link to more information for each build. The Environment tab allows you to view and edit environment variables for the application's build environment. The Events tab displays a list of build related events and metadata.
      * Examine the logs for the Source-to-Image build of the application. In the Builds menu, click Builds to display a list of recent builds for the youruser-console project.
      * Click the php-helloworld-1 link to access information for the first build of the php-helloworld application



* + - * Explore the available information from the Details tab. Next, click the Logs tab. A scrollable text box contains output from the build process

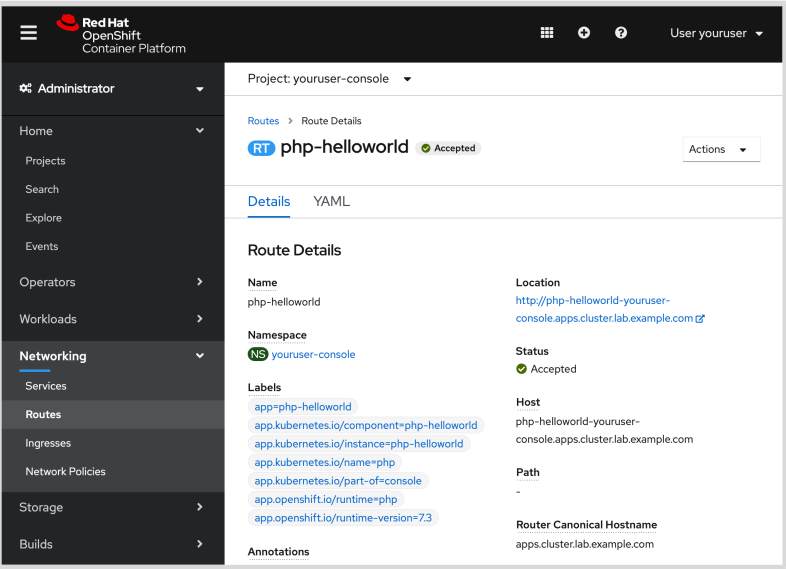


* + - * When Podman builds a container image, similar output is observed compared with the output shown in the browser.
      * Locate information for the php-helloworld application's service. In the navigation bar, click Networking to reveal more menu choices. Click Services to display a list of services for the youruser-console project. Click the php-helloworld link to display the information associated with the application's service

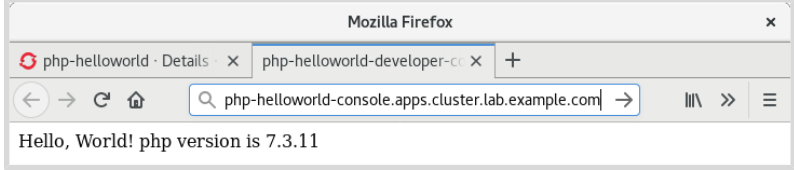


* + - * Explore the available information from the Details tab. The YAML tab allows you to view and edit the service configuration, as a YAML file. The Pods tab displays the current list of pods that provide the application service.
      * Locate external route information for the application. On the navigation bar, click Networking → Routes to display a list of configured routes for the youruser- console project. Click the php-helloworld link to display information associated

with the application's route

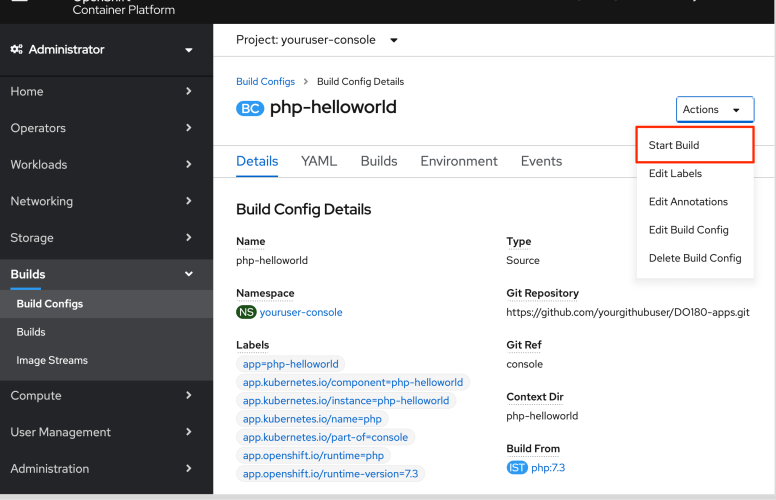


* + - * Explore the available information from the Details tab. The Location field provides a link to the external route for the application; [http://php-helloworld-](about:blank) ${RHT\_OCP4\_DEV\_USER}-console.${RHT\_OCP4\_WILDCARD\_DOMAIN}. Click the link to access the application in a new tab



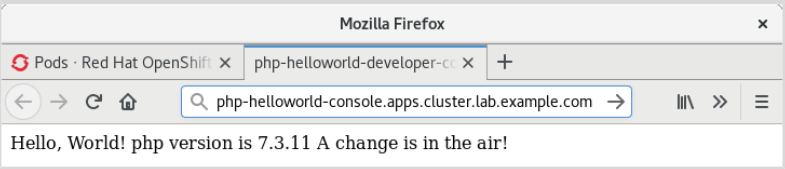
* + - * Modify the application code, commit the change, push the code to the remote Git repository, and trigger a new application build
      * Trigger an application build manually from the web console.

On the navigation bar, click Builds → Build Configs and then click the php-helloworld link to access the Build Config Details page. From the Actions menu in the upper right of the screen, click Start Build

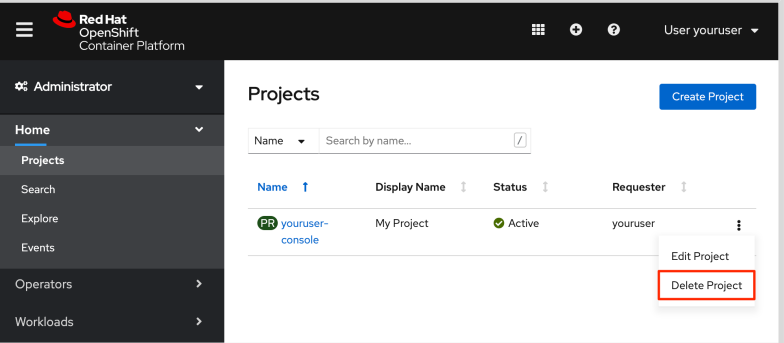


* + - * You are redirected to a Build Details page for the new build. Click the Logs tab to monitor progress of the build. The last line of a successful build contains Push successful.
      * When the build completes, the deployment starts. Go to the Workloads → Pods section, and wait for the new pod to be deployed and running.
      * Reload the http://php-helloworld-${RHT\_OCP4\_DEV\_USER}-console.

${RHT\_OCP4\_WILDCARD\_DOMAIN} URL in the browser. The application response corresponds to the updated source code

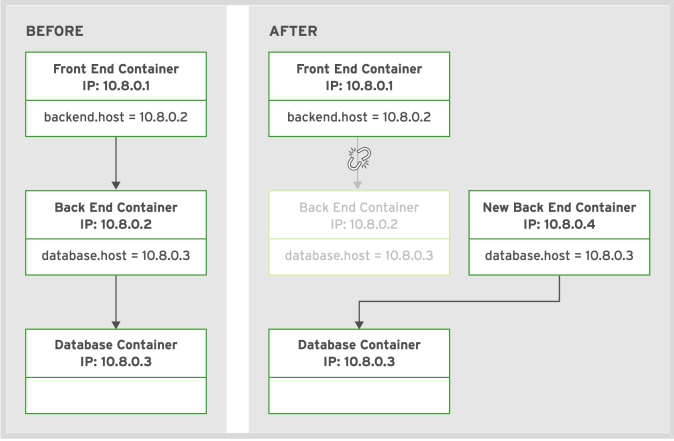


* + - Delete the project. On the navigation bar, click Home → Projects. Click the icon at the right side of the row containing an entry for the youruser-console project. Click Delete Project from the menu that appears.



**Deploying Multi-Container Application.**

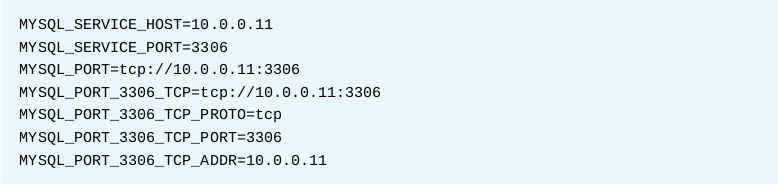
* **Considerations for Multi-Container Applications**
  + **Leveraging Multi-Container Applications**
    - A more complex application, however, can get the benefits of deploying different components into different containers.
    - Consider an application composed of a front-end web application, a REST back end, and a database server. Those components may have different dependencies, requirements and life cycles.
    - Although it is possible to orchestrate multi-container applications' containers manually, Kubernetes and OpenShift provide tools to facilitate orchestration. Attempting to manually manage dozens or hundreds of containers quickly becomes complicated.
  + **Discovering Services in a Multi-Container Application**
    - Podman uses Container Network Interface (CNI) to create a software-defined network (SDN) between all containers in the host. Unless stated otherwise, CNI assigns a new IP address to a container when it starts.
    - Each container exposes all ports to other containers in the same SDN. As such, services are readily accessible within the same network. The containers expose ports to external networks only by explicit configuration.
    - Due to the dynamic nature of container IP addresses, applications cannot rely on either fixed IP addresses or fixed DNS host names to communicate with middleware services and other application services.
    - Containers with dynamic IP addresses can become a problem when working with multi-container applications because each container must be able to communicate with others to use services upon which it depends.
    - For example, consider an application composed of a front-end container, a back-end container, and a database. The front-end container needs to retrieve the IP address of the back-end container. Similarly, the back-end container needs to retrieve the IP address of the database container. Additionally, the IP address could change if a container restarts, so a process is needed to ensure any change in IP triggers an update to existing containers.



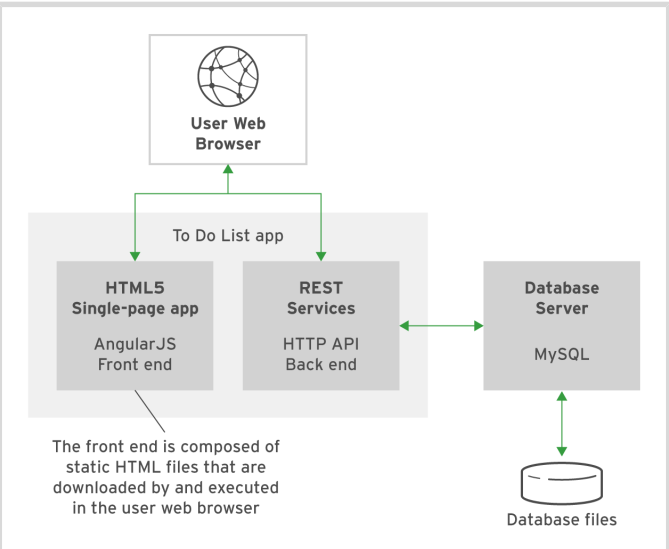
* + - Both Kubernetes and OpenShift provide potential solutions to the issue of service discoverability and the dynamic nature of container networking.
  + **Comparing Podman and Kubernetes**
    - Using environment variables allows you to share information between containers with Podman. However, there are still some limitations and some manual work involved in ensuring that all environment variables stay in sync, especially when working with many containers.
    - Kubernetes provides an approach to solve this problem by creating services for your containers
    - Pods are attached to a Kubernetes namespace, which OpenShift calls a project.
    - When a pod starts, Kubernetes automatically adds a set of environment variables for each service defined on the same namespace.
    - Any service defined on Kubernetes generates environment variables for the IP address and port number where the service is available. Kubernetes automatically injects these environment variables into the containers from pods in the same namespace.
    - These environment variables usually follow a convention:
      * Uppercase: All environment variables are set using uppercase names.
      * Snake Case: Any environment variable created by a service is usually composed of multiple words separated with an underscore (\_).
      * Service name first: The first word for an environment variable created by a service is the service name.
      * Protocol type: Most network environment variables include the protocol type (TCP or UDP).
    - These are the environment variables generated by Kubernetes for a service:
      * **<SERVICE\_NAME>\_SERVICE\_HOST**: Represents the IP address enabled by a service to access a pod.
      * **<SERVICE\_NAME>\_SERVICE\_PORT:** Represents the port where the server port is listed.
      * **<SERVICE\_NAME>\_PORT:** Represents the address, port, and protocol provided by the service for external access.
      * **<SERVICE\_NAME>\_PORT\_<PORT\_NUMBER>\_<PROTOCOL>:** Defines an alias for the <SERVICE\_NAME>\_PORT.
      * **<SERVICE\_NAME>\_PORT\_<PORT\_NUMBER>\_<PROTOCOL>\_PROTO:** Identifies the protocol type (TCP or UDP).
      * **<SERVICE\_NAME>\_PORT\_<PORT\_NUMBER>\_<PROTOCOL>\_PORT:** Defines an alias for <SERVICE\_NAME>\_SERVICE\_PORT.
      * **<SERVICE\_NAME>\_PORT\_<PORT\_NUMBER>\_<PROTOCOL>\_ADDR:** Defines an alias for <SERVICE\_NAME>\_SERVICE\_HOST.
    - For instance, given the following service:



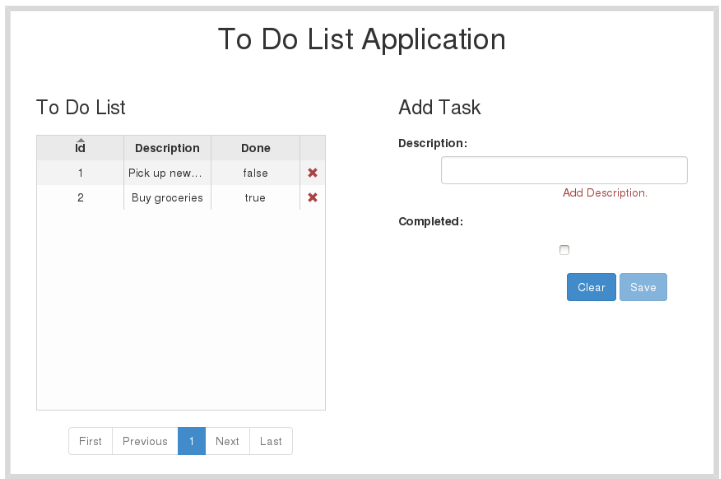
* + - The following environment variables are available for each pod created after the service, on the same namespace



* + - Other relevant <SERVICE\_NAME>\_PORT\_\* environment variable names are set on the basis of the protocol. IP address and port number are set in the <SERVICE\_NAME>\_PORT environment variable. For example, MYSQL\_PORT=tcp://10.0.0.11:3306 entry leads to the creation of environment variables with names such as MYSQL\_PORT\_3306\_TCP, MYSQL\_PORT\_3306\_TCP\_PROTO, MYSQL\_PORT\_3306\_TCP\_PORT, and MYSQL\_PORT\_3306\_TCP\_ADDR. If the protocol component of an environment variable is undefined, Kubernetes uses the TCP protocol and assigns the variable names accordingly
  + **Describing the To Do List Application**
    - To Do List Application Logical Architecture

****

* + - The presentation tier is built as a single-page HTML5 front-end using AngularJS**.**
    - The business tier is composed of an HTTP API back-end, with Node.js.
    - The persistence tier based on a MySQL database server



* + - On the left is a table with items to complete, and on the right is a form to add a new item.
    - Application Version:
      * Nodejs: Represents the way a typical developer would create the application as a single unit, without caring to break it into tiers or services.
      * nodejs\_api: Shows the changes needed to break the application into presentation and business tiers. Each tier corresponds to an isolated container image.
    - The sources of both of these application versions are available from the todoapp/nodejs folder in the Git repository at: <https://github.com/RedHatTraining/DO180-apps.git>
    - **Example: Deploying the Web Application and MySQL Containers**
    - In this example we are using DO180 git repo
    - <https://github.com/MohitBis/DO180-apps>
    - Build the MySQL image.

$ cd ~/DO180/labs/multicontainer-design/images/mysql

$ sudo podman build -t do180/mysql-57-rhel7 \

> --layers=false .

$ sudo podman images

* + - Build the Node.js parent image using the provided Dockerfile.

cd ~/DO180/labs/multicontainer-design/images/nodejs

sudo podman build -t do180/nodejs \

> --layers=false .

$ sudo podman images \

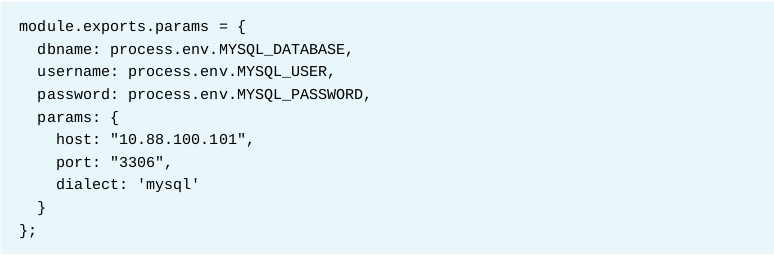
> --format "table {{.ID}} {{.Repository}} {{.Tag}}"

* + - Build the To Do List application child image using the provided Dockerfile.

/home/student/DO180/labs/multicontainer-design/deploy/nodejs/

Dockerfile

* + - /home/student/DO180/labs/multicontainer-design/deploy/nodejs/nodejs-source/models/db.js



* + - Notice the environment variables used by the REST API. These variables are exposed to the container using -e options with the podman run command
    - The host and port details of the MySQL container are embedded with the REST API application. The host, as shown above in the db.js file, is the IP address of the mysql container.
    - Build the child image.

$ cd ~/DO180/labs/multicontainer-design/deploy/nodejs

$ ./build.sh

$ sudo podman images \

> --format "table {{.ID}} {{.Repository}} {{.Tag}}"

* + - Modify the existing script to create containers with the appropriate IP, as defined in the previous step. In this script, the order of commands is given such that it starts the mysql container and then starts the todoapi container before connecting it to the mysql container. After invoking every container, there is a wait time of 9 seconds, so each container has time to start.
    - Edit the run.sh file located at /home/student/DO180/labs/multicontainer-

design/deploy/nodejs/networked to insert the podman run command at the appropriate line for invoking mysql container.

sudo podman run -d --name mysql -e MYSQL\_DATABASE=items -e MYSQL\_USER=user1 \

-e MYSQL\_PASSWORD=mypa55 -e MYSQL\_ROOT\_PASSWORD=r00tpa55 \

-v $PWD/work/data:/var/lib/mysql/data \

-v $PWD/work/init:/var/lib/mysql/init -p 30306:3306 \

--ip 10.88.100.101 do180/mysql-57-rhel7

sudo podman run -d --name todoapi -e MYSQL\_DATABASE=items -e MYSQL\_USER=user1 \

-e MYSQL\_PASSWORD=mypa55 -p 30080:30080 \

do180/todonodejs

* + - Run the containers.

$ cd \

/home/student/DO180/labs/multicontainer-design/deploy/nodejs/networked

$ ./run.sh

* + - Test the application.

$ curl -w "\n" \

> <http://127.0.0.1:30080/todo/api/items/1>

* **Deploying a Multi-Container Application on OpenShift**
  + **Examining the Skeleton of a Template**
    - OpenShift templates provide a way to simplify the creation of resources that an application requires.
    - A template defines a set of related resources to be created together, as well as a set of application parameters. The attributes of template resources are typically defined in terms of the template parameters, such as a resource's name attribute.
    - For example, an application might consist of a front-end web application and a database server. Each consists of a service resource and a deployment configuration resource. They share a set of credentials (parameters) for the front end to authenticate to the back end. The template can be processed by specifying parameters or by allowing them to be automatically generated (for example, for a unique database password) in order to instantiate the list of resources in the

template as a cohesive application.

* + - The OpenShift installer creates several templates by default in the openshift namespace. Run the oc get templates command with the -n openshift option to list these pre installed templates

$ oc get templates -n openshift

* + - The following shows a YAML template definition

$ oc get template mysql-persistent -n openshift -o yaml

****

* + - 1 Defines a list of arbitrary tags to associate with this template. Enter any of these tags in the UI to find this template.
    - 2 Define the template name.
    - 3 The objects section defines the list of OpenShift resources for this template. This template creates four resources: a Secret, a Service, a PersistentVolumeClaim, and a DeploymentConfig.
    - 4 All four resource objects have their names set to the value of the

DATABASE\_SERVICE\_NAME parameter.

* + - 5 6 The parameters section contains a list of nine parameters. Template resources often define their attributes using the values of these parameters, as demonstrated with the DATABASE\_SERVICE\_NAME parameter.
    - 7 If you do not specify a value for the MYSQL\_PASSWORD parameter when you create an application with this template, OpenShift generates a password that matches this regular expression.
    - You can publish a new template to the OpenShift cluster so that other developers can build an application from the template.
    - Assume you have an task list application named todo that requires an OpenShift DeploymentConfig, Service, and Route object for deployment.
    - You create a YAML template definition file that defines attributes for these OpenShift resources, along with definitions for any required parameters.
    - Assuming the template is defined in the todo-template.yaml file, use the

oc create command to publish the application template

$ oc create -f todo-template.yaml

* + - By default, the template is created under the current project unless you specify a different one using the -n option

$ oc create -f todo-template.yaml \

> -n openshift

* + - Any template created under the openshift namespace (OpenShift project) is available in the web console under the dialog box accessible in the Catalog → Developer Catalog menu item. Moreover, any template created under the current project is accessible from that project
    - **Parameters**
      * Templates define a set of parameters, which are assigned values.
    - OpenShift resources defined in the template can get their configuration values by referencing named parameters.
    - Parameters in a template can have default values, but they are optional.

Any default value can be replaced when processing the template.

* + - Each parameter value can be set either explicitly by using the oc process command, or generated by OpenShift according to the parameter configuration
    - There are two ways to list available parameters from a template. The first one is using the oc describe command

$ oc describe template mysql-persistent -n openshift

$ oc process --parameters mysql-persistent -n openshift

* + **Processing a Template Using the CLI**
    - When you process a template, you generate a list of resources to create a new application. To process a template, use the oc process command

$ oc process -f <filename>

* + - The previous command processes a template file, in either JSON or YAML format, and returns the list of resources to standard output. The format of the output resource list is JSON. To output the

resource list in YAML format, use the -o yaml with the oc process command

$ oc process -o yaml -f <filename>

* + - Another option is to process a template from the current project or the openshift project

$ oc process <uploaded-template-name>

* + - The oc process command returns a list of resources to standard output. This output can be redirected to a file

$ oc process -o yaml -f filename

> myapp.yaml

* + - Templates often generate resources with configurable attributes that are based on the template parameters. To override a parameter, use the -p option followed by a <name>=<value> pair.

$ oc process -o yaml -f mysql.yaml \

> -p MYSQL\_USER=dev -p MYSQL\_PASSWORD=$P4SSD -p MYSQL\_DATABASE=bank \

> -p VOLUME\_CAPACITY=10Gi > mysqlProcessed.yaml

* + - To create the application, use the generated YAML resource definition file

$ oc create -f mysqlProcessed.yaml

* + - Alternatively, it is possible to process the template and create the application without saving a resource definition file by using a UNIX pipe

$ oc process -f mysql.yaml -p MYSQL\_USER=dev \

> -p MYSQL\_PASSWORD=$P4SSD -p MYSQL\_DATABASE=bank \

-p VOLUME\_CAPACITY=10Gi | oc create -f -

* + - To use a template in the openshift project to create an application in your project, first export the template

$ oc get template mysql-persistent -o yaml \

> -n openshift > mysql-persistent-template.yaml

* + - You can also use two slashes (//) to provide the namespace as part of the template name

$ oc process openshift//mysql-persistent \

> -p MYSQL\_USER=dev -p MYSQL\_PASSWORD=$P4SSD -p MYSQL\_DATABASE=bank \

> -p VOLUME\_CAPACITY=10Gi | oc create -f -

* + - Alternatively, it is possible to create an application using the oc new-app command passing the template name as the --template option argument

$ oc new-app --template=mysql-persistent \

> -p MYSQL\_USER=dev -p MYSQL\_PASSWORD=$P4SSD -p MYSQL\_DATABASE=bank \

> -p VOLUME\_CAPACITY=10Gi \

> --as-deployment-config

* + **Configuring Persistent Storage for OpenShift Applications**
    - OpenShift Container Platform manages persistent storage as a set of pooled, cluster-wide resources.
    - To add a storage resource to the cluster, an OpenShift administrator creates a PersistentVolume object that defines the necessary metadata for the storage resource.
    - The metadata describes how the cluster accesses the storage, as well as other storage attributes such as capacity or throughput.
    - To list the PersistentVolume objects in a cluster, use the oc get pv command

$ oc get pv

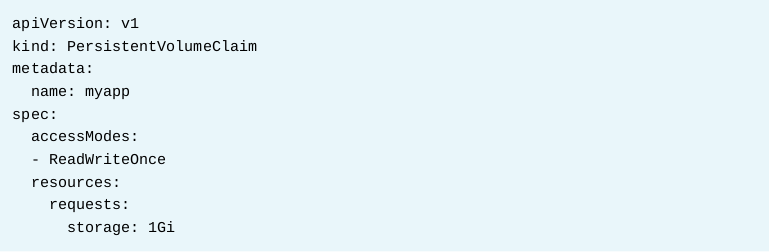
* + - To see the YAML definition for a given PersistentVolume, use the oc get command with the -o yaml option

$ oc get pv pv0001 -o yaml

* + - To add more PersistentVolume objects to a cluster, use the oc create command

$ oc create -f pv1001.yaml

* + - **Requesting Persistent Volumes**
      * When an application requires storage, you create a PersistentVolumeClaim (PVC) object to request a dedicated storage resource from the cluster pool. The following content from a file named pvc.yaml is an example definition for a PVC



* + - * The PVC defines storage requirements for the application, such as capacity or throughput. To create the PVC, use the oc create command

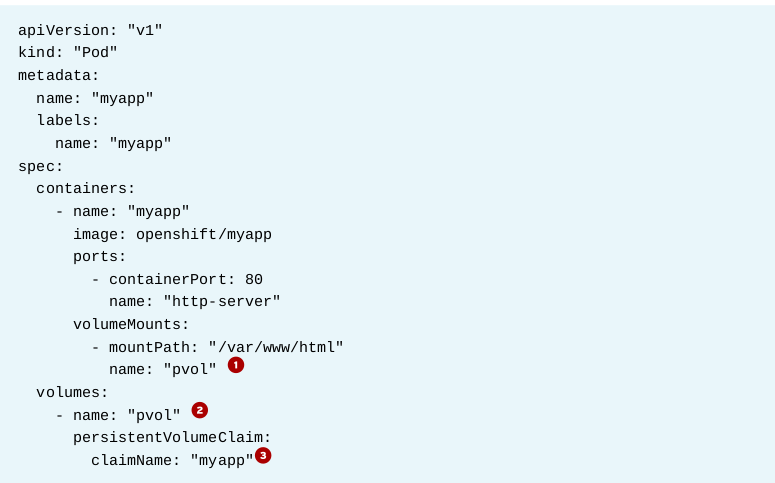
$ oc create -f pvc.yaml

* + - * After you create a PVC, OpenShift attempts to find an available PersistentVolume resource that satisfies the PVC's requirements. If OpenShift finds a match, it binds the PersistentVolume

object to the PersistentVolumeClaim object. To list the PVCs in a project, use the oc get pvc command

$ oc get pvc

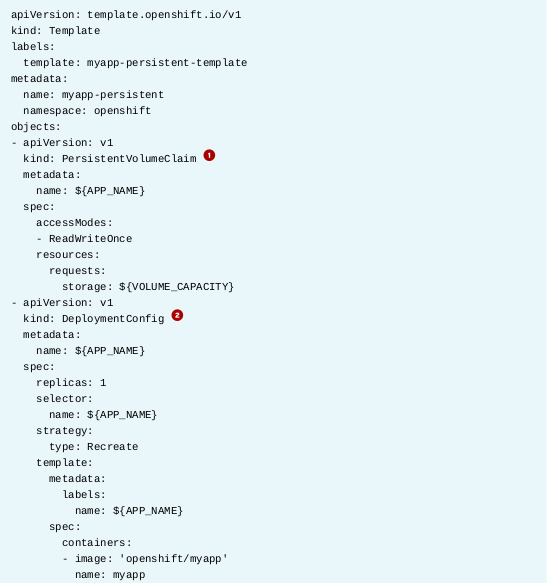
* + - * The output indicates whether a persistent volume is bound to the PVC, along with attributes of the PVC (such as capacity)
      * To use the persistent volume in an application pod, define a volume mount for a container that references the PersistentVolumeClaim object. The application pod definition below references a PersistentVolumeClaim object to define a volume mount for the application

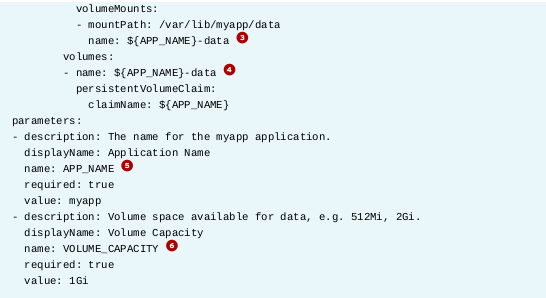


* + - * 1 This section declares that the pvol volume mounts at /var/www/html in the container file system
      * 2 This section defines the volume.
      * 3 The pvol volume references the myapp PVC. If OpenShift associates an available persistent volume to the myapp PVC, then the pvol volume refers to this associated volume.
    - **Configuring Persistent Storage with Templates**
      * Templates are often used to simplify the creation of applications requiring persistent storage. Many of these templates have a suffix of -persistent

$ oc get templates -n openshift | grep persistent

* + - * The following example template defines a PersistentVolumeClaim object along with a DeploymentConfig object





* + - * 1 2 The template defines a PersistentVolumeClaim and DeploymentConfig object. Both objects have names matching the value of the APP\_NAME parameter. The persistent volume

claim defines a capacity corresponding the value of the VOLUME\_CAPCITY parameter

* + - * 3 4 The DeploymentConfig object defines a volume mount referencing the PersistentVolumeClaim created by the template.
      * 5 6 The template defines two parameters: APP\_NAME and VOLUME\_CAPACITY. The template uses these parameters to specify the value of attributes for the PersistentVolumeClaim

and DeploymentConfig objects.

* + - * With this template, you only need to specify the APP\_NAME and VOLUME\_CAPACITY parameters to deploy the myapp application with persistent storage

$ oc create myapp-template.yaml

template.template.openshift.io/myapp-persistent created

[student@workstation ~]$ oc process myapp-persistent \

> -p APP\_NAME=myapp-dev -p VOLUME\_CAPACITY=1Gi \

> | oc create -f -

* + **Example: Creating an Application with a Template**
    - Use the Dockerfile in the images/mysql subdirectory to build the database container. Publish the container image to quay.io with a tag of do180-mysql-57-rhel7

$ cd ~/DO180/labs/multicontainer-openshift/images/mysql

$ sudo podman build -t do180-mysql-57-rhel7 .

$ sudo podman login quay.io -u ${RHT\_OCP4\_QUAY\_USER}

$ sudo podman tag \

> do180-mysql-57-rhel7 quay.io/${RHT\_OCP4\_QUAY\_USER}/do180-mysql-57-rhel7

* + - Build the base image for the To Do List application using the Node.js Dockerfile, located in the exercise subdirectory images/nodejs. Tag the image as do180-nodejs

$ cd ~/DO180/labs/multicontainer-openshift

$ cd images/nodejs

$ sudo podman build --layers=false -t do180-nodejs .

* + - Use the build.sh script in the deploy/nodejs subdirectory to build the To Do List application. Publish the application image to quay.io with an image tag of do180-todonodejs

$ cd ~/DO180/labs/multicontainer-openshift

$ cd deploy/nodejs\

$ ./build.sh

$ sudo podman tag do180/todonodejs \

0/todonodejs \

> quay.io/${RHT\_OCP4\_QUAY\_USER}/do180-todonodejs

$ sudo podman push \

> quay.io/${RHT\_OCP4\_QUAY\_USER}/do180-todonodejs

* + - Create the To Do List application from the provided JSON template.

$ oc login -u ${RHT\_OCP4\_DEV\_USER} \

> -p ${RHT\_OCP4\_DEV\_PASSWORD} ${RHT\_OCP4\_MASTER\_API}

$ oc new-project ${RHT\_OCP4\_DEV\_USER}-template

$ cd /home/student/DO180/labs/multicontainer-openshift

$ oc process \

> -f todo-template.json -p RHT\_OCP4\_QUAY\_USER=${RHT\_OCP4\_QUAY\_USER} \

> | oc create -f -

$ oc get pods -w

* + - Expose the Service

$ oc expose service todoapi

* + - Test the application.

$ oc status | grep -o "http:.\*com"

* + - Use curl to test the REST API for the To Do List application.

$ curl -w "\n" \

> $(oc status | grep -o "http:.\*com")/todo/api/items/1

**Troubleshooting S21 Builds and Deployments**

* The Source-to-Image (S2I) process is a simple way to automatically create images based on the programming language of the application source code in OpenShift.
* While this process is often a convenient way to quickly deploy applications, problems can arise during the S2I image creation process, either by the programming language characteristics or the runtime environment that

require both developers and administrators to work together.

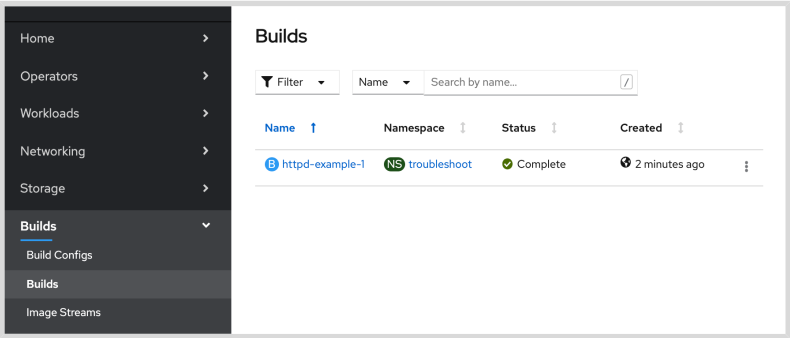
* It is important to understand the basic workflow for most of the programming languages

supported by OpenShift. The S2I image creation process is composed of two major steps:

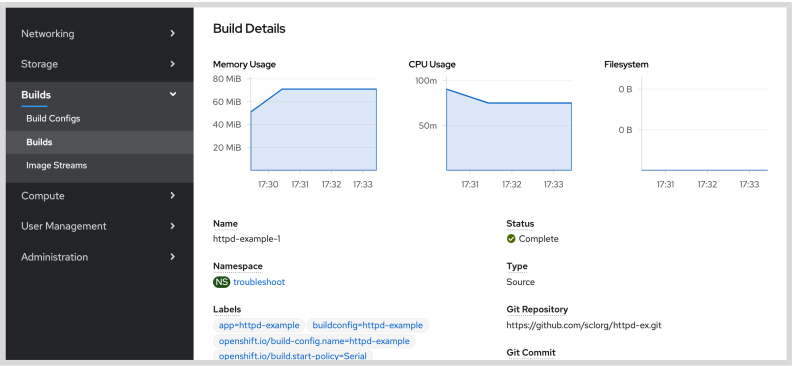
* + **Build step:** Responsible for compiling source code, downloading library dependencies, and packaging the application as a container image. Furthermore, the build step pushes the image to the OpenShift registry for the deployment step. The BuildConfig (BC) OpenShift resources drive the build step.
  + **Deployment step:** Responsible for starting a pod and making the application available for OpenShift. This step executes after the build step, but only if the build step succeeded. The DeploymentConfig (DC) OpenShift resources drive the deployment step.
* For the S2I process, each application uses its own BuildConfig and DeploymentConfig

objects, the name of which matches the application name. The deployment process aborts if the build fails.

* The S2I process starts each step in a separate pod. The build process creates a pod named <application-name>-build-<number>-<string>.
* For each build attempt, the entire build step executes and saves a log. Upon a successful build, the application starts on a separate pod named as <application-name>-<string>.
* The OpenShift web console can be used to access the details for each step. To identify any build issues, the logs for a build can be evaluated and analyzed by clicking the Builds link from the left panel, depicted as follows.



* For each build attempt, a history of the build, tagged with a number, is provided for evaluation. Clicking on the build name leads to the details page of the build



* The Logs tab of the build details page shows the output generated by the build execution. Those logs are handy to identify build issues.
* Use the Deployment Configs link under the Workloads section from the left panel to identify issues during the deployment step.
* After selecting the appropriate deployment configuration, details show in the Details section.
* The oc command-line interface has several subcommands for managing the logs. Likewise in the web interface, it has a set of commands which provides information about each step. For example, to retrieve the logs from a build configuration, run the following command.

$ oc logs bc/<application-name>

* If a build fails, after finding and fixing the issues, run the following command to request a new build

$ oc start-build <application-name>

* By issuing that command, OpenShift automatically spawns a new pod with the build process.

$ oc logs dc/<application-name>

* If the deployment is running or has failed, the command returns the logs of the process

deployment process. Otherwise, the command returns the logs from the application's pod

* **Describing Common Problems**
  + Sometimes, the source code requires some customization that may not be available in containerized environments, such as database credentials, file system access, or message queue information. Those values usually take the form of internal environment variables. Developers using the S2I process may need to access this information.
  + The oc logs command provides important information about the build, deploy, and run processes of an application during the execution of a pod. The logs may indicate missing values or options that must be enabled, incorrect parameters or flags, or environment incompatibilities.
  + Application logs must be clearly labelled to identify problems quickly without the need to learn the container internals.
  + **Troubleshooting Permission Issues**
    - OpenShift runs S2I containers using Red Hat Enterprise Linux as the base image, and any runtime difference may cause the S2I process to fail.
    - Sometimes, the developer runs into permission issues, such as access denied due to the wrong permissions, or incorrect environment permissions set by administrators.
    - S2I images enforce the use of a different user than the root user to access file systems and external resources. Also, Red Hat Enterprise Linux 7 enforces SELinux policies that restrict access to some file system resources, network ports, or processes.
    - Some containers may require a specific user ID, whereas S2I is designed to run containers using a random user as per the default OpenShift security policy.
* **Troubleshooting Containerized Applications**
  + **Forwarding Ports for Troubleshooting**
    - Occasionally developers and system administrators need special network access to a container that would not be needed by application users.
    - For example, developers may need to use the

administration console for a database or messaging service, or system administrators may make use of SSH access to a container to restart a terminated service. Such network access, in the form

network ports are usually not exposed by the default container configurations, and tend to require specialized clients used by developers and system administrators.

* + - Podman provides port forwarding features by using the -p option along with the run subcommand. In this case, there is no distinction between network access for regular application access and for troubleshooting.

$ sudo podman run --name db -p 30306:3306 mysql

* + - The previous command maps the host port 30306 to the port 3306 on the db container. This container is created from the mysql image, which starts a MySQL server that listens on port 3306.
    - OpenShift provides the oc port-forward command for forwarding a local port to a pod port. This is different than having access to a pod through a service resource:
      * The port-forwarding mapping exists only in the workstation where the oc client runs, while a service maps a port for all network users.
      * A service load-balances connections to potentially multiple pods, whereas a port-forwarding mapping forwards connections to a single pod.

$ oc port-forward db 30306 3306

* + - The previous command forwards port 30306 from the developer machine to port 3306 on the db pod, where a MySQL server (inside a container) accepts network connections.
    - When running this command, be sure to leave the terminal window running. Closing the window or canceling the process stops the port mapping.
    - While the podman run -p method of mapping (port-forwarding) can only be configured when the container is started, the mapping with the oc port-forward command can be created and destroyed at any time after a pod was created.
    - Creating a service of NodePort type for a database pod would be similar to running podman run -p. However, Red Hat discourages the usage of the NodePort approach to avoid exposing the service to direct connections. Mapping with port-forwarding in OpenShift is considered a more secure alternative.
  + **Enabling Remote Debugging with Port Forwarding**
    - Another use for the port forwarding feature is enabling remote debugging. Many integrated development environments (IDEs) provide the capability to remotely debug an application.
    - For example, JBoss Developer Studio (JBDS) allows users to utilize the Java Debug Wire Protocol (JDWP) to communicate between a debugger (JBDS) and the Java Virtual Machine. When enabled, developers can step through each line of code as it is being executed in real time.
    - For JDWP to work, the Java Virtual Machine (JVM) where the application runs must be started with options enabling remote debugging. For example, WildFly and JBoss EAP users must configure these options on application server startup. The following line in the standalone.conf file enables remote debugging by opening the JDWP TCP port 8787, for a

WildFly or EAP instance running in standalone mode:

JAVA\_OPTS="$JAVA\_OPTS \

> -agentlib:jdwp=transport=dt\_socket,address=8787,server=y,suspend=n"

* + - When the server starts with the debugger listening on port 8787, a port forwarding mapping needs to be created to forward connections from a local unused TCP port to port 8787 in the EAP pod.
    - If the developer workstation has no local JVM running with remote debugging enabled, the local port can also be 8787.
    - The following command assumes a WildFly pod named jappserver running a container from an image previously configured to enable remote debugging:

$ oc port-forward jappserver 8787:8787

* + - Once the debugger is enabled and the port forwarding mapping is created, users can set breakpoints in their IDE of choice and run the debugger by pointing to the application's host name and debug port (in this instance, 8787).
  + **Accessing Container Logs**
    - Podman and OpenShift provide the ability to view logs in running containers and pods to facilitate troubleshooting. But neither of them is aware of application specific logs. Both expect the

application to be configured to send all logging output to the standard output.

* + - A container is simply a process tree from the host OS perspective. When Podman starts a container either directly or on the RHOCP cluster, it redirects the container standard output and standard error, saving them on disk as part of the container's ephemeral storage. This way, the container logs can be viewed using podman and oc commands, even after the container was stopped, but not removed.
    - To retrieve the output of a running container, use the following podman command.

$ podman logs <containerName>

* + - In OpenShift, the following command returns the output for a container within a pod:

$ oc logs <podName> [-c <containerName>]

* + **OpenShift Events**
    - Some developers consider Podman and OpenShift logs to be too low-level, making troubleshooting difficult. Fortunately, OpenShift provides a high-level logging and auditing facility called events
    - OpenShift events signal significant actions like starting a container or destroying a pod.
    - To read OpenShift events, use the get subcommand with the events resource type for the oc command, as follows.

$ oc get events

* + - Events listed by the oc command this way are not filtered and span the whole RHOCP cluster.
    - Using a pipe to standard UNIX filters such as grep can help, but OpenShift offers an alternative in order to consult cluster events. The approach is provided by the describe subcommand.
    - For example, to only retrieve the events that relate to a mysql pod, refer Events field from the output of oc describe pod mysql command.

$ oc describe pod mysql

* + **Overriding Container Binaries**
    - Many container images do not contain all of the troubleshooting commands users expect to find in regular OS installations, such as telnet, netcat, ip, or traceroute. Stripping the image from basic utilities or binaries allows the image to remain slim, thus, running many containers per host.
    - One way to temporarily access some of these missing commands is mounting the host binaries folders, such as /bin, /sbin, and /lib, as volumes inside the container. This is possible because

the -v option from podman run command does not require matching VOLUME instructions to be present in the Dockerfile of the container image.

* + - To access these commands in OpenShift, you need to change the pod resource definition in order to define volumeMounts and volumeClaims objects. You also need to create a hostPath persistent volume.
    - The following command starts a container, and overrides the image's /bin folder with the one from the host. It also starts an interactive shell inside the container.

$ sudo podman run -it -v /bin:/bin image /bin/bash

* + - The directory of binaries to override depends on the base OS image. For example, some commands require shared libraries from the /lib directory. Some Linux distributions have different contents in /bin, /usr/bin, /lib, or /usr/lib, which would require using the -v option for each directory.
    - As an alternative, you can include these utilities in the base image. To do so, add instructions in a Dockerfile build definition. For example, examine the following excerpt from a Dockerfile definition, which is a child of the rhel7.5 image used throughout this course. The RUN instruction

installs the tools that are commonly used for network troubleshooting.

FROM ubi7/ubi:7.7

RUN yum install -y \

less \

dig \

ping \

iputils && \

yum clean all

* + **Transferring Files To and Out of Containers**
    - When troubleshooting or managing an application, you may need to retrieve or transfer files to and from running containers, such as configuration files or log files. There are several ways to move

files into and out of containers, as described in the following list.

* + - Volume mounts
      * Another option for copying files from the host to a container is the usage of volume mounts. You can mount a local directory to copy data into a container. For example, the following command sets /conf host directory as the volume to use for the Apache configuration directory in the container. This provides a convenient way to manage the Apache server without having to rebuild the container image.

$ sudo podman run -v /conf:/etc/httpd/conf -d do180/apache

* + - podman cp
      * The cp subcommand allows users to copy files both into and out of a running container. To copy a file into a container named todoapi, run the following command.

$ sudo podman cp standalone.conf

* + - * To copy a file from the container to the host, flip the order of the previous command.

$ sudo podman cp todoapi:/opt/jboss/standalone/conf/standalone.conf .

* + - * The podman cp command has the advantage of working with containers that were already started, while the following alternative (volume mounts) requires changes to the command

used to start a container.

* + - podman exec
      * For containers that are already running, the podman exec command can be piped to pass files both into and out of the running container by appending commands that are executed

in the container. The following example shows how to pass in and execute a SQL file inside a

MySQL container:

$ sudo podman exec -i <container> mysql -uroot -proot < /path/on/host/db.sql <

db.sql

* + - * Using the same concept, it is possible to retrieve data from a running container and place it in the host machine. A useful example of this is the usage of the mysqldump utility, which

creates a backup of MySQL database from the container and places it on the host.

* + - * sudo podman exec -it <containerName> sh \

-c 'exec mysqldump -h"$MYSQL\_PORT\_3306\_TCP\_ADDR" \

-P"$MYSQL\_PORT\_3306\_TCP\_PORT" \

-uroot -p"$MYSQL\_ENV\_MYSQL\_ROOT\_PASSWORD" items'

db\_dump.sql

* + - * The previous command uses the container environment variables to connect to the MySQL server to execute the mysqldump command and redirects the output to a file on the host

machine. It assumes that the container image provides the mysqldump utility, so there is no need to install the MySQL administration tools on the host.

* + - * The oc rsync command provides functionality similar to podman cp for containers running under OpenShift pods