ORACLE University

Integrated Cloud Applications & Platform Services

Build Container-Native Apps using Oracle Container Pipelines

Student Guide

D107085GC10 | D107605

ORACLE

Disclaimer

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS
The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

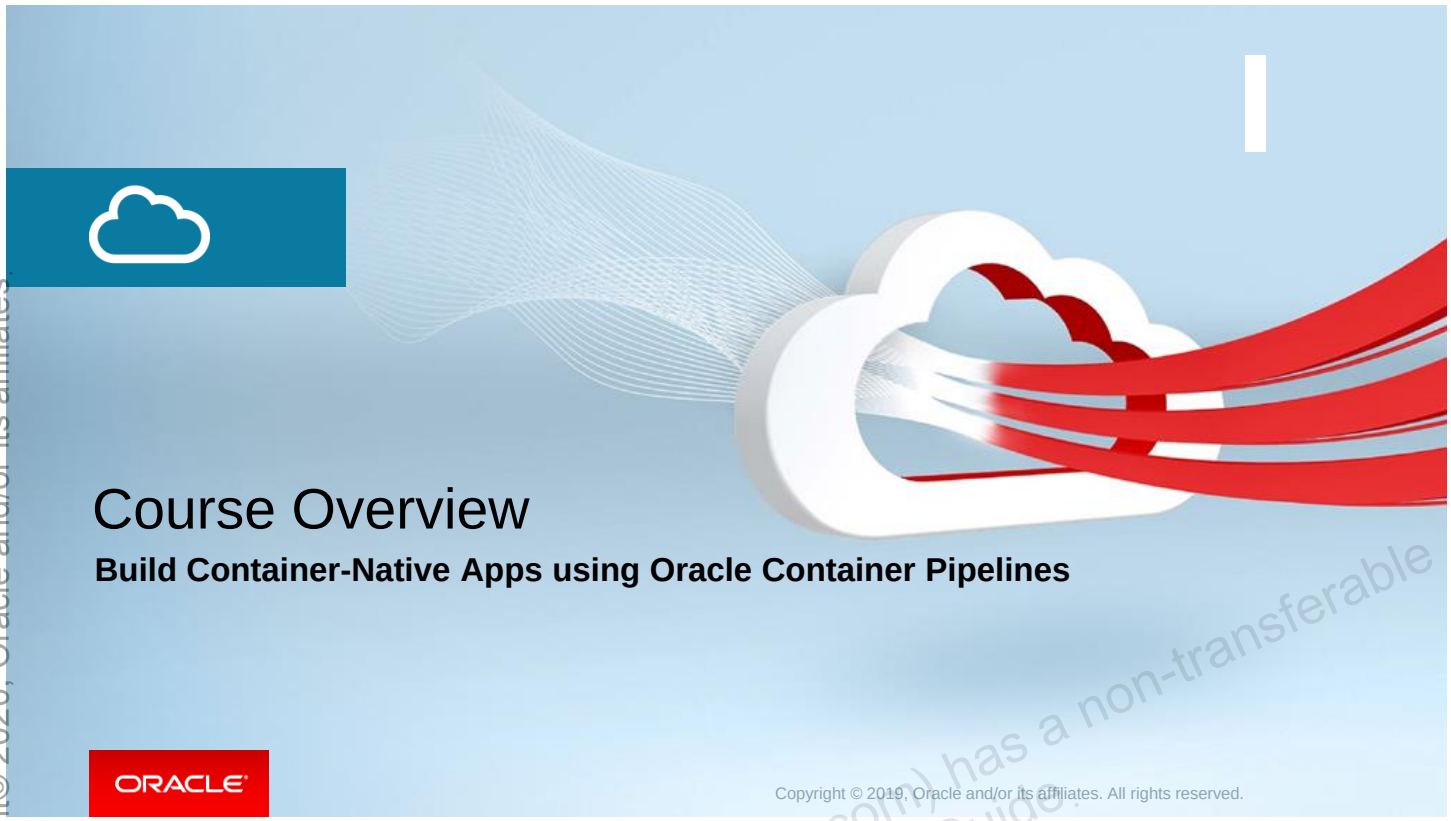
Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

1012262019

Contents

5 Development Workflow



Course Overview

Build Container-Native Apps using Oracle Container Pipelines

Course Objectives

After completing this course, you should be able to:

- Leverage the core principles and functionalities of Container Pipelines
- Create and manage workflows
- Set up deployment pipeline with Container pipeline
- Trigger deployment to OKE

Target Audience

- Cloud Application Developer
- Integration Developer
- Cloud Administrator
- Build /Release teams of Applications migrating to OCI

Prerequisites

- Basic understanding of Oracle Cloud and Cloud computing
- Awareness of Microservices or SOA architecture.
- Familiarity with Docker (Docker Architecture and Operations)
- Working Knowledge of Kubernetes including deployment of applications in Kubernetes.
- Awareness to Devops – CI/CD Pipeline

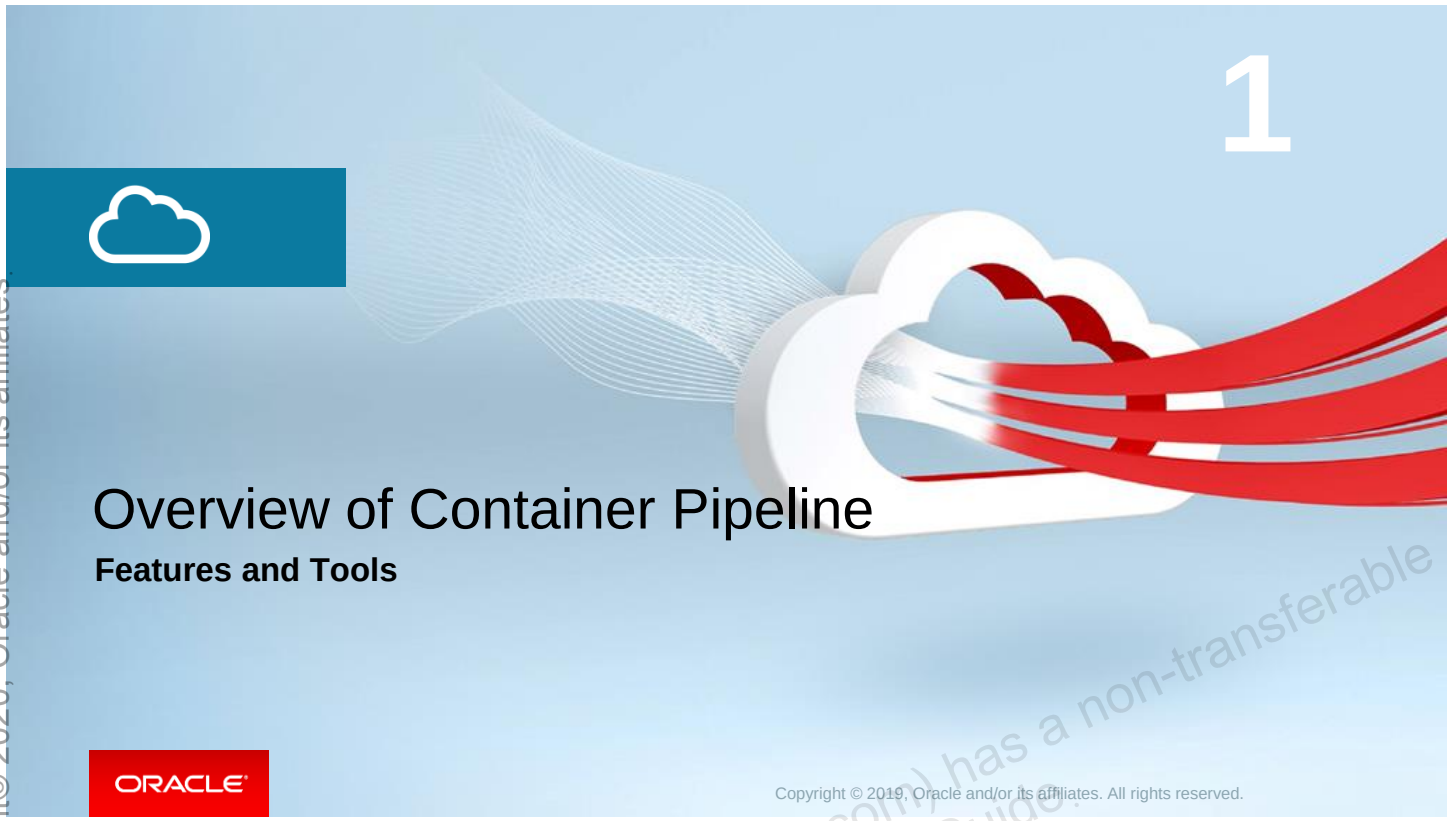
Schedule

Day	Lessons
Day 1	Lesson 1 – Overview of Container Pipeline Lesson 2 – Platform and Languages Support Lesson 3 – Introducing Container Architecture Lesson 4 – Exploring Web Interfaces
Day 2	Lesson 5 – Development Workflow Lesson 6 – Deployment with Container Pipeline Lesson 7 – Integrations

Summary

After completing this course, you should be able to:

- Appreciate the core principles and functionalities of Container Pipelines
- Manage workflows end to end
- Orchestrate deployment pipeline with Container pipeline
- Deploy to OKE



Overview of Container Pipeline

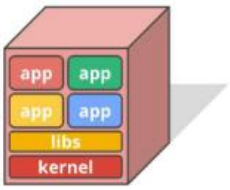
Features and Tools

Objectives

After completing this lesson, you should be able to:

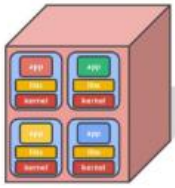
- Describe Oracle Container Pipeline
- Explain the features of Oracle Container Pipeline
- Describe the technologies and tools involved
- Explain the big picture
- Access the building blocks of Oracle Container Pipeline

Container: Overview



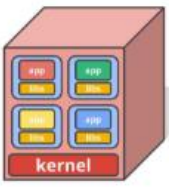
Physical Machine

- ✗ No isolation
- ✗ Common libs
- ✗ Highly coupled Apps & OS



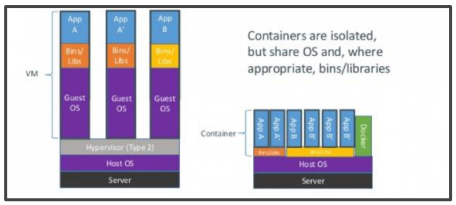
Virtual Machines

- ✓ Isolation
- ✓ No Common Libs
- ✗ Expensive and Inefficient
- ✗ Hard to manage



Containers

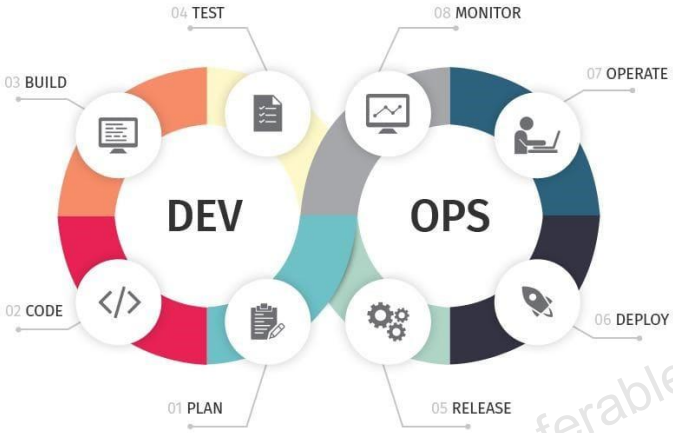
- ✓ Isolation
- ✓ No Common Libs
- ✓ Less overhead
- ✗ Less Dependency on Host OS



Container in cloud computing is basically an approach to operating system virtualization. By this, the user can work with a program and its dependencies using resource procedures that are isolated.

The concept relating to the container in cloud computing is beneficial to the enterprises. Then there is the virtual machine, which is low cost and offers internal as well as cloud portability. Some big organizations comprise both virtual machines and cloud containers. The enterprises have to evaluate the pros and cons of each of these and deploy the most apt one.

The Development Engine – “**evolved**”



Devops is a set of cultural practices related to software development. It emphasizes collaboration between all parts of the IT organization and the “continuous delivery” of software.

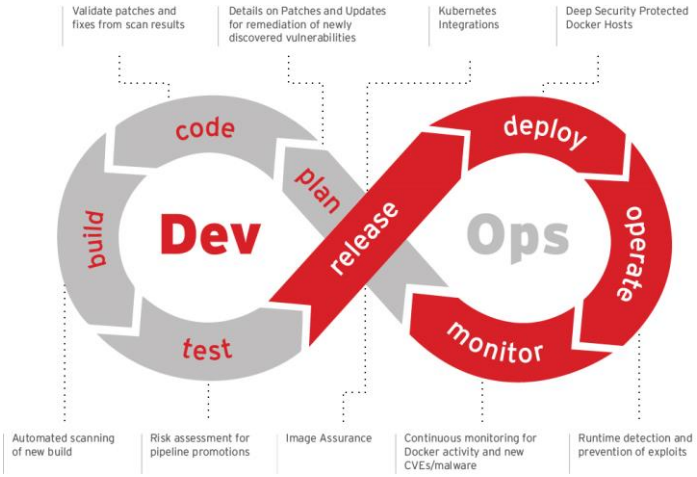
Containers and DevOps often are part of the same conversation. That’s not because they are the same thing. However, they are very distinct concepts.

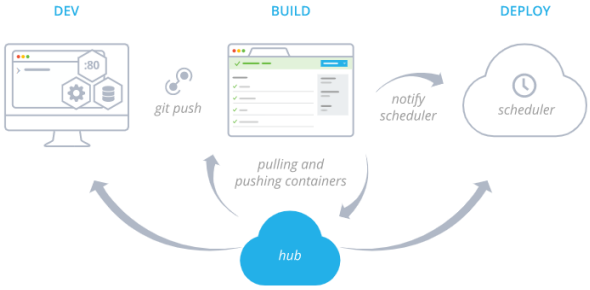
Containers are a type of technology that makes it easier to host applications inside portable environments. Containers have been around in the Linux world for well over a decade. But it was not until 2013, when Docker launched, that containers become popular.

DevOps, in contrast, is a set of cultural practices related to software development. It emphasizes collaboration between all parts of the IT organization and the “continuous delivery” of software. As you see in the slide – multiple teams have come forward to deliver a goal – Release.

An important thing to understand is DevOps is not tied to any particular type of technology. Adopting it does not mean that you simply use a particular set of tools or frameworks to build software. You can do DevOps with any type of tool.

What's New?





Wercker is a container-centric platform for developing, building, and delivering your applications in the modern world of microservices, containers, and the cloud.

Wercker provides you with a toolchain and cloud platform to speed up your development process. It allows you to build apps for the 21st century by containerizing your code and automating your development pipelines.

Why Wercker?

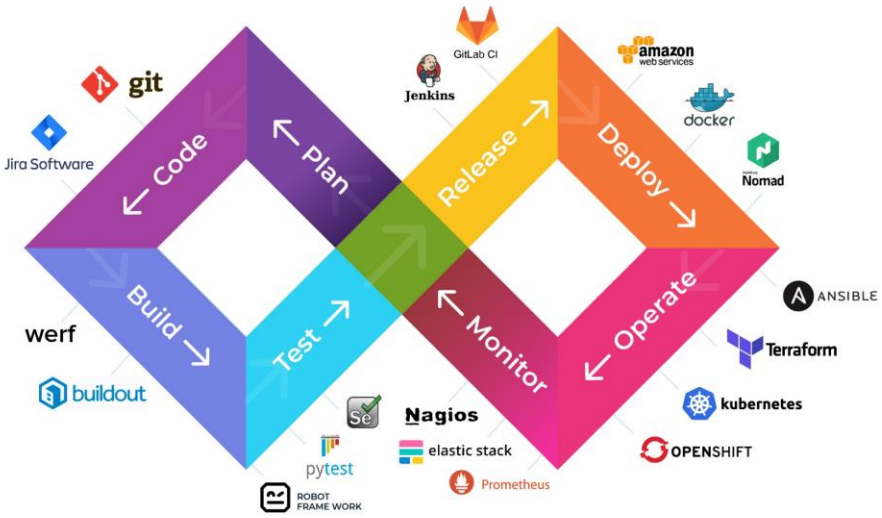
The landscape of production has changed: monolithic is out, loosely coupled microservices are in. Modern applications consist of multiple moving parts, but most of the existing developer tools we use were designed and built in the world of monolithic applications.

Working with microservices poses new challenges: your applications now consist of multiple processes, multiple configurations, multiple environments, and more than one codebase.

To fully leverage the advantages of microservices, they should be managed carefully.

Wercker lets you do just that. By setting up automated development pipelines, very much in the spirit of "release early, release often", you can build and deploy your services with just a git push. You can then let Wercker compile and run any other steps that are necessary to build your project before it gets deployed to a target of your choosing.

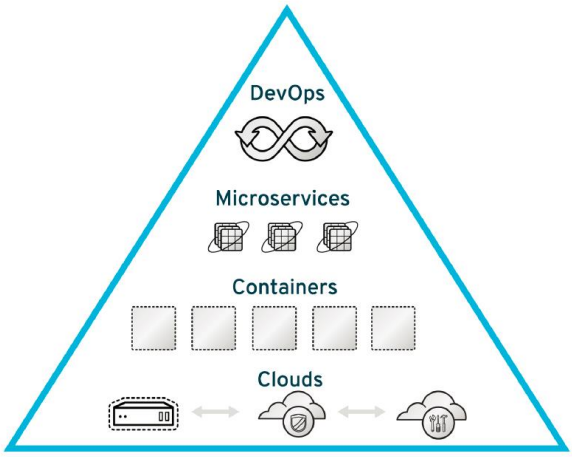
Big Picture: Devops

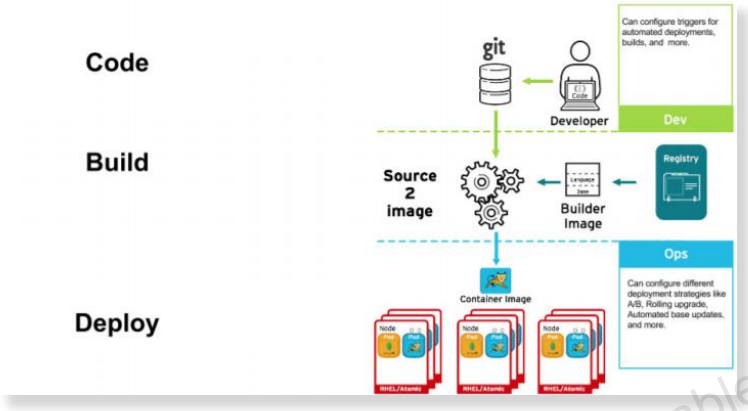


- **Containers** simplify the build/test/deploy pipelines in **DevOps**.
- With Docker **containers**, developers own what's in the **container** (application and service, and dependencies to frameworks and components) and how the **containers** and services behave together as an application.

ORACLE

Boils Down to Engineering As Microservices





Microservices is an approach to application development in which a large application is built as a suite of modular services.

A **container** is an isolated guest in container-based virtualization. Container-based virtualization uses a single kernel to run multiple instances of an operating system.

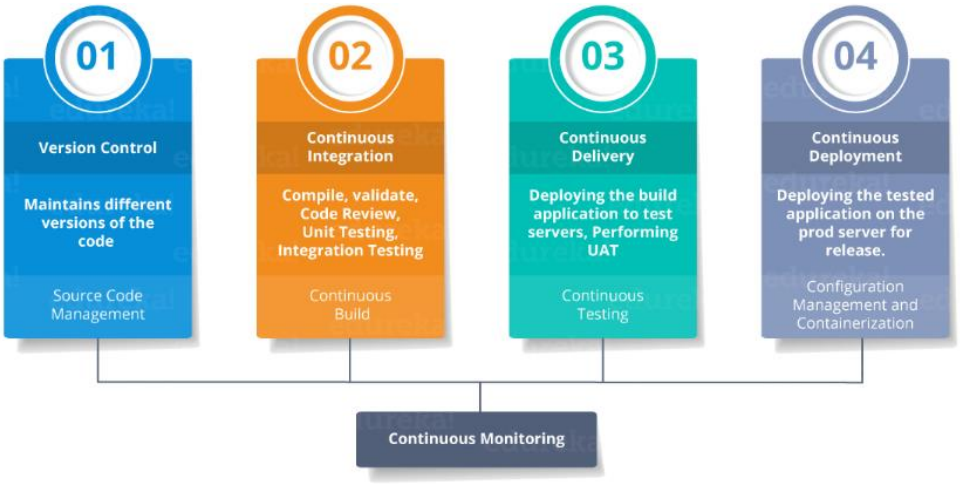
Microservices is an approach to application development in which a large application is built as a suite of modular services. Each module supports a specific business goal and uses a simple, well-defined interface to communication with other modules.

A **container** is an isolated guest in container-based virtualization. Container-based virtualization uses a single kernel to run multiple instances of an operating system. Each instance runs in a completely isolated environment, so there is no risk that one container can gain access to another's files.

Microservices and container technologies have become hot trends in software development over the last couple of years. Why? Because, as Luke Marsden explains for instance, microservice architectures used in combination with containers allow developers to "decouple software into smaller functional pieces," and containers "extend this decoupling, separating software from the underlying hardware."

Benefits include increased development speed and aptitude for change, along with the ability to deliver better and faster updates. This results in more resilient, scalable, and portable solutions. Moreover, when a single module fails, the larger application can remain largely unaffected.

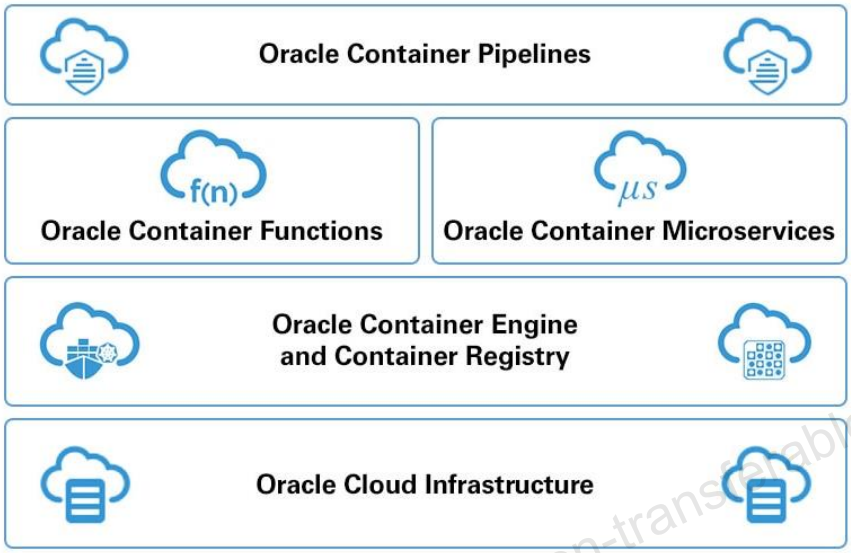
Pipeline



A **pipeline** in a software engineering term is a set of automated processes that allow developers and DevOps professionals to reliably and efficiently compile, build, and deploy their code to their production compute platforms.

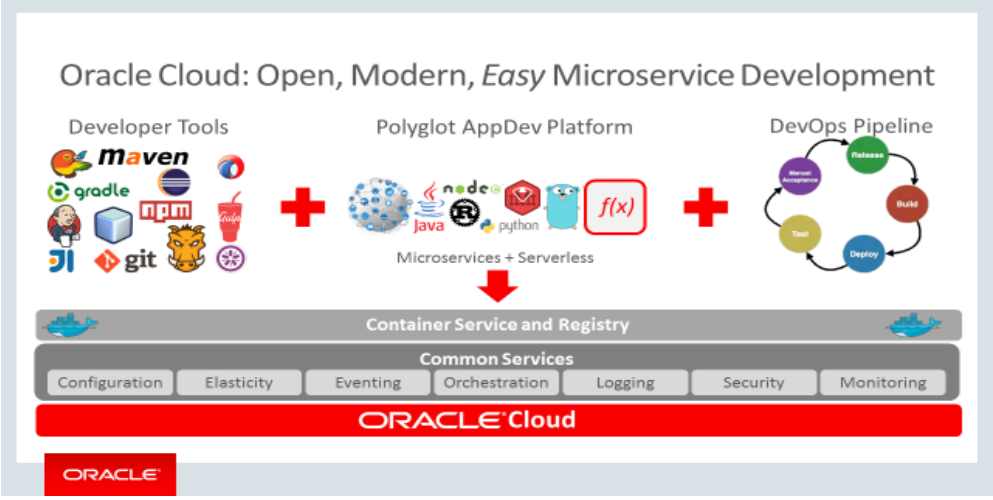
Oracle Container Pipeline

- The Oracle Cloud Platform is an open, modern, and easy way for organizations to implement proven ways to build modern applications.
- The platform provides a comprehensive AppDev suite, as well as an automated DevOps platform that supports CI/CD, mobile and API-first delivery, and robust monitoring in a simple dashboard.

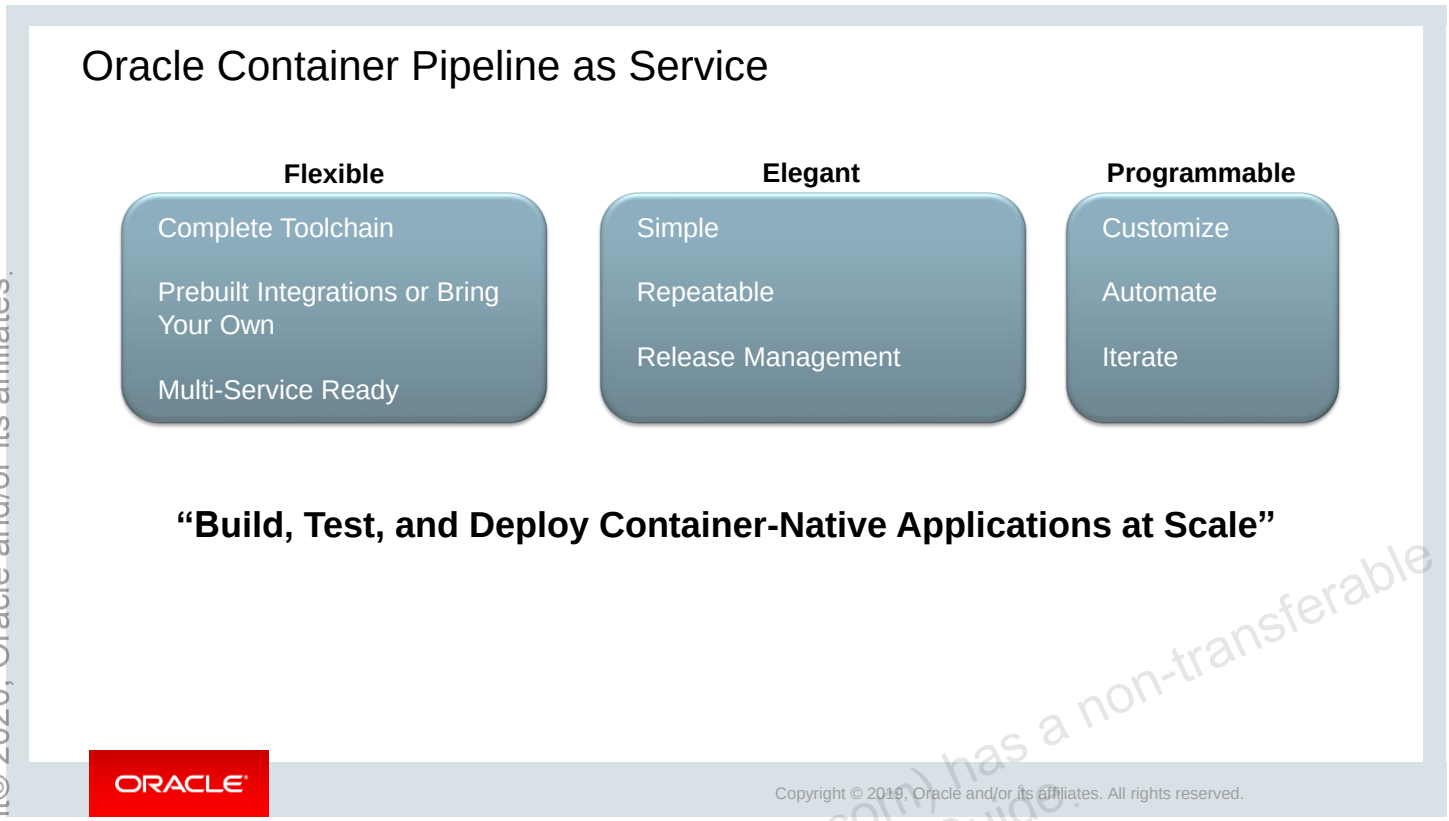


The Oracle Cloud Platform is an open, modern, and easy way for organizations to implement proven ways to build modern applications. The platform provides a comprehensive AppDev suite, as well as an automated DevOps platform that supports CICD, mobile and API-first delivery, and robust monitoring in a simple dashboard.

Oracle Container Pipeline as Service (Big Picture)



To deploy and operate containers, Oracle offers Container Engine for Kubernetes (OKE), which is certified by CNCF. We chose unmodified Kubernetes for container orchestration because of the openness and interoperability it can provide. You can quickly spin up DevTest environments in the cloud and run production on premises, or run everything in Oracle Cloud. You can also keep your on-premises environment for DevTest.



Complete Toolchain

- Seamlessly build, test, and deploy your container-based applications using the Container Pipelines CI/CD, fully integrated Docker v2 compatible private Container Registry and Container Engine.

Prebuilt Integrations or Bring Your Own

- Use a library of prebuilt integrations with popular deployment targets such as Kubernetes, or bring your own.

Multi-Service Ready

- Spin up multiple concurrent microservice pipeline Docker containers for easy management and debugging.

Simple

- Chain and trigger pipelines to create complex multi-stage, multi-branch CI/CD workflows with ease.

Repeatable

- Reuse patterns defined in steps, pipelines, and workflows to ensure consistency. Use the CLI on your developer laptop.

Release Management

- Use the dashboard to keep up-to-date with build and deployment operations within your team. Container Pipelines also supports communication and collaboration tools, so everyone knows when containers are updated.

Customize

- Use workflows to capture your patterns and best practices for development and deployment.

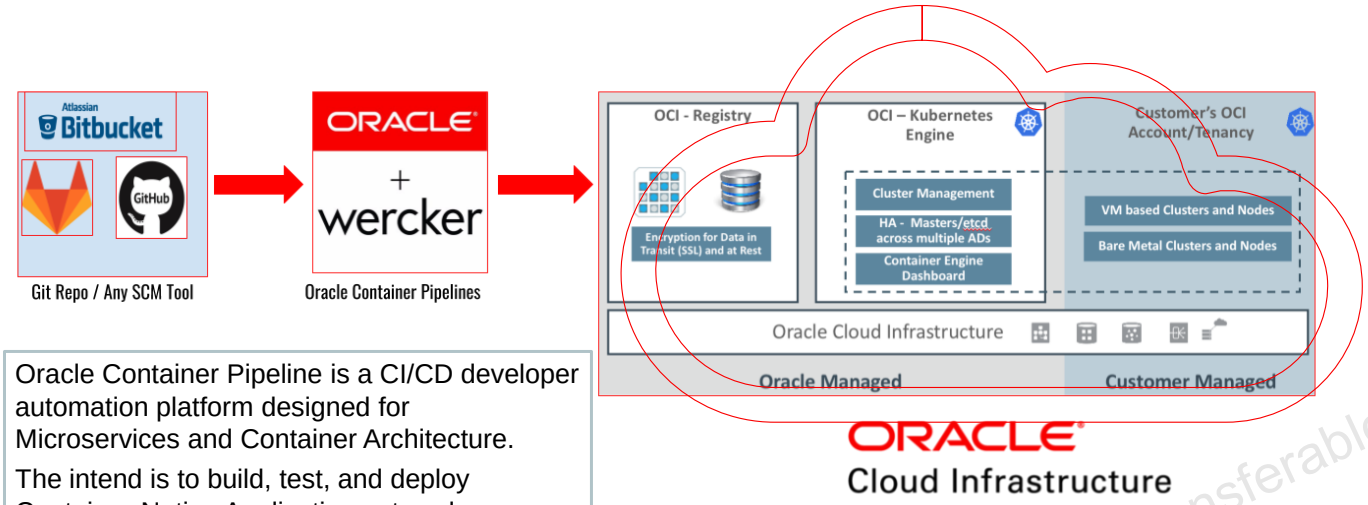
Automate

- Use steps to accomplish specific automation tasks. Steps can be created from scratch or consumed from our steps marketplace.

Iterate

- Leverage the open source CLI tool to rapidly build, test, and deploy without leaving your local development environment.

Tools: Oracle Container Pipeline



Oracle Container Pipeline is a CI/CD developer automation platform designed for Microservices and Container Architecture.

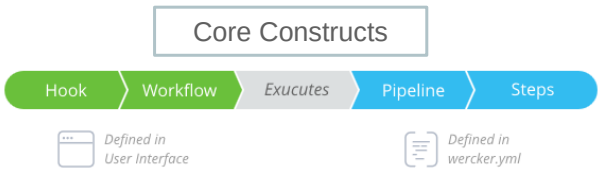
The intend is to build, test, and deploy Container-Native Applications at scale.

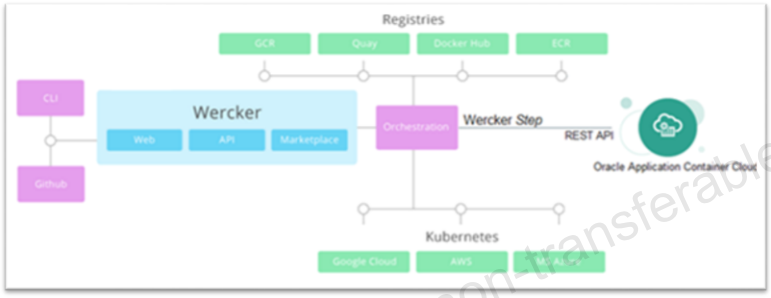
ORACLE Cloud Infrastructure

ORACLE

You can code faster, fix bugs more easily, create complex automated pipelines for builds, set up notifications and alerts, and get more stuff done. Oracle is a Platinum member of the Cloud Native Computing Foundation and a member of the Open Container Initiative.

Building Blocks – Oracle Container Pipeline





Steps are self-contained Bash scripts or compiled binaries for accomplishing specific automation tasks, defined in the `wercker.yml` file of your application. Steps can be written manually or borrowed from the community via the Steps Registry.

- Use internal steps for interaction with your Docker Container.
- Use script steps to automate shell commands.
- Create custom steps to address specific challenges.

Pipelines are the heart and soul of Wercker. They're where you define the actions (steps) and environment for your tasks. They are also often where you define your tests, builds, and deploys. They are an aggregate of steps and will pass or fail based on the steps within. A great advantage of Wercker's pipelines is that the builds can be run concurrently, allowing for parallelized testing to save valuable development time.

Workflows provide you with a mechanism for managing your pipelines. With Workflows, you can connect your pipelines in series and parallel to form complex, end-to-end CI/CD flows that can take your applications from source to production.

When using Wercker, think of running in the order of steps, pipelines, and workflows. You need steps and an environment for your pipelines, and you need pipelines for your workflow to connect them and form the flows you need to go from development to production.

Terminology

Keyword	Abstract
Application	Your project with all settings, git information and pipelines
Box	Abstract definition for container that runs pipeline
Build	Single execution of build pipeline
Container	Lightweight VM
Deploy	Execution of deploy pipeline
Deploy pipeline	A target of deploy that needs to be executed.

Terminology

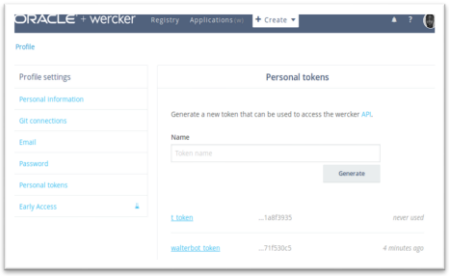
Keyword	Abstract
Service	Container spin up process
Stack	Underlying architecture that runs the pipeline
Step	Individual units of work
Pipeline	Sequential execution of steps
Registry	Marketplace that contains all steps developed by community
Wercker.yml	Defines the configuration of your automation pipelines with a set of tasks (steps) that you want to execute

Interfaces in Container Pipeline

Wercker CLI

Wercker GUI

Wercker API



Wercker Web

The Wercker Web interface is a graphical, web-based front end for adding applications, configuring environment variables, and monitoring your workflows.

Wercker CLI

The Wercker CLI provides a command-line interface that you can run on your local development station to build and test applications. It supports Linux and macOS.

Wercker API

Using the Wercker API, you can control tasks in Wercker using HTTP endpoints.

Summary

- Got an overview of Oracle Container Pipeline
- Got an understanding of Devops, Microservices and Automations in Container Pipelines
- Understood Container Pipelines Tools, and Technologies

Platform and Languages Support

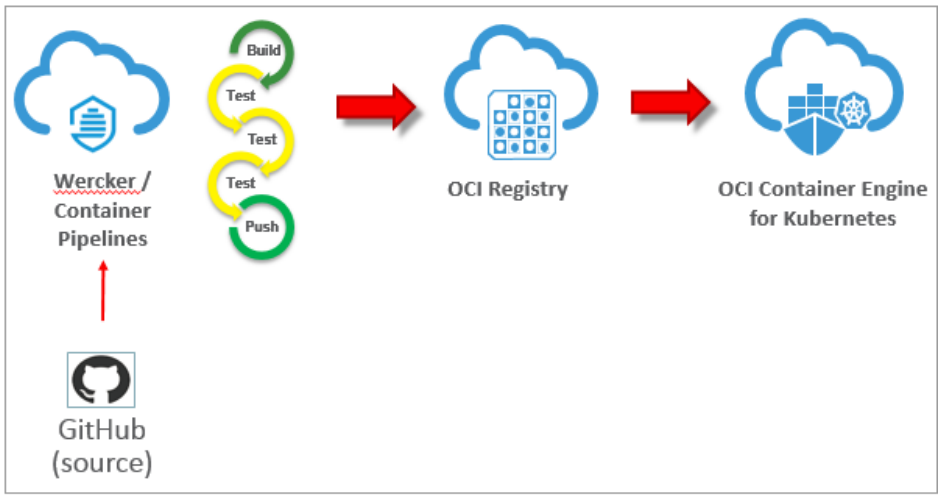
The Versatility of Container Pipeline

Objectives

After completing this lesson, you should be able to identify the:

- Supported languages in Oracle Container Pipeline
- Supported platforms in Oracle Container Pipeline

Context of Container Pipeline

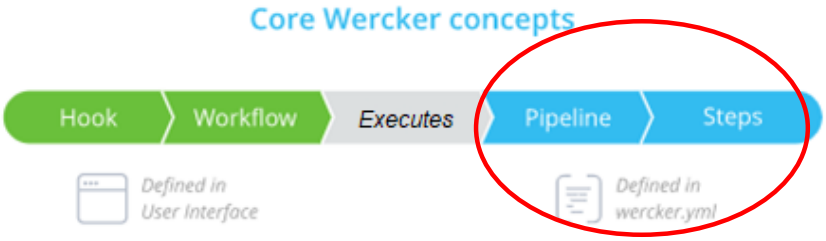


Wercker runs a pipeline and builds the image, pushes the image to OCIR, and then deploys the container to an instance of OKE, replacing the running containers/pods, thus updating the application.

As part of the development process, developers write new code, and merge their code back into the master (source code). In the slide, the repository of this source code is GitHub. Wercker is integrated into GitHub so that, when there is a commit (new code, or changes are made to a branch or the master), Wercker can automatically build a container image.

In the case of a commit to master, Wercker runs a pipeline and builds the image, pushes the image to OCIR and then deploys the container to an instance of OKE, replacing the running containers/pods, and thus updating the application.

Languages Support: YAML and Wercker.yml



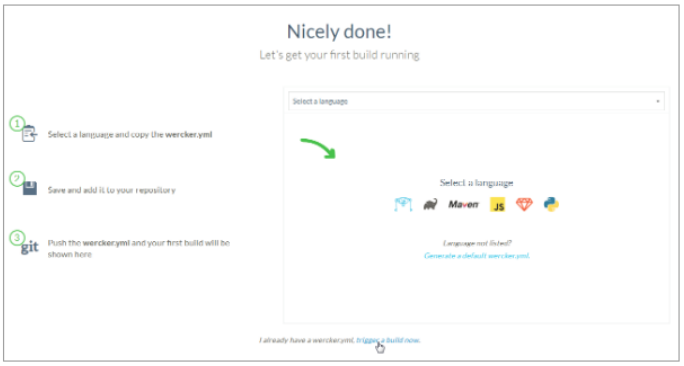
YAML is a human-readable data-serialization language. It is commonly used for configuration files, but could be used in many applications where data is being stored or transmitted.

`wercker.yml` is the only config file you need for using Wercker. In this config file, you will define all the steps required to successfully develop, build, and deploy your application.

Obviously, your code can be written in a number of languages. We need an Integrator. YAML targets many of the same communications applications as XML but has a minimal syntax, which intentionally differs from SGML. Continuous development describes a process for iterative software development and is an umbrella over several other processes including continuous integration, continuous testing, continuous delivery, and continuous deployment.

Connector: `wercker.yml`

The `wercker.yml` configuration file manages the automation pipelines through as many steps as needed. You can think of steps as calls to action processes and pipelines as collections of one or more steps.



Wercker is a software automation tool that aims to improve the Continuous-Integration and Continuous-Delivery (CI/CD) process for developers and organizations. It enables the creation of automated workflows, or pipelines, which specify a series of tasks or commands that are run on your code whenever a change is pushed to the source repository.

Language: Java

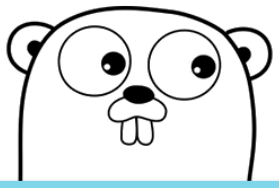
Java is an object-oriented, class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology. Java is a programming language and a platform. Java is a high level, robust, object-oriented and secure programming language.

```
# The container definition we want to use for developing our app
box:
  id: openjdk
  ports:
    - "8080"
```


In the YAML, `id:` is the specification for openjdk as the application that connects it as Glue.

Language: Go

Go is an open source programming language that makes it easy to build **simple**, **reliable**, and **efficient** software.



Download Go

/path/to/getting-started-golang/wercker.yml

```
box:
id: golang
ports:
  - 3000

dev:
steps:
- internal/watch:
    code: |
      go build ./...
      ./source
    reload: true

# Build definition
build:
    # The steps that will be executed on build
    steps:

    # golint step
        - wercker/golint

    # Build the project
```

ORACLE

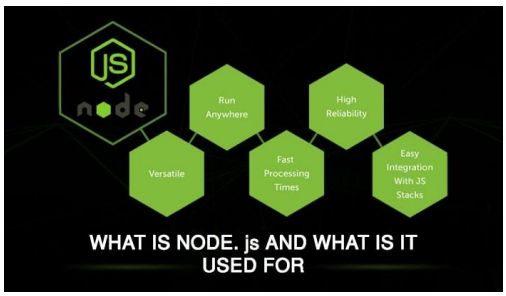
Here in YAML, `id:` is the specification for Go Lang as the application that connects it as Glue.

Language: Python


```
# use a small python 2.x container
box: python:2-slim

# Build definition
build:
  # The steps that will be executed on build
  steps:
    # A step that executes `pip install` command
    - pip-install

    # A custom script step, name value is used in the UI
    # and the code value contains the command that get executed
    - script:
        name: python unit test
        code: |
          python app_test.py
```


The id property states which container image you want to use for your project. We're developing in Python; therefore, for this sample, we've chosen the python:2.7-slim image.

Language: JavaScript




```
# The container definition we want to use for developing our app
box:
  id: node:latest
  ports:
    - "8080"
```

Note the id property of the YAML file. It points to node as the development is in JavaScript.

Note that the id property states which container image you want to use for your project. Because we're developing with NodeJS, in this sample, we're using the node:latest image.

Language: Ruby

A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.

In `wercker.yml`, you can observe the box ID mentioning ruby 22.

```
box: phusion/passenger-ruby22
build:
    steps:
        - bundle-install
        - script:
            name: rspec
            code: bundle exec rspec
```

Ruby is a language of careful balance. Its creator, Yukihiro “Matz” Matsumoto, blended parts of his favorite languages (Perl, Smalltalk, Eiffel, Ada, and Lisp) to form a new language that balanced functional programming with imperative programming.

He has often said that he is “trying to make Ruby natural, not simple,” in a way that mirrors life.

Building on this, he adds: Ruby is simple in appearance, but is very complex inside, just like our human body.

Language: PHP

PHP (recursive acronym for PHP: *Hypertext Preprocessor*) is a widely-used open source general-purpose scripting language that is especially suited for web development and can be embedded into HTML.

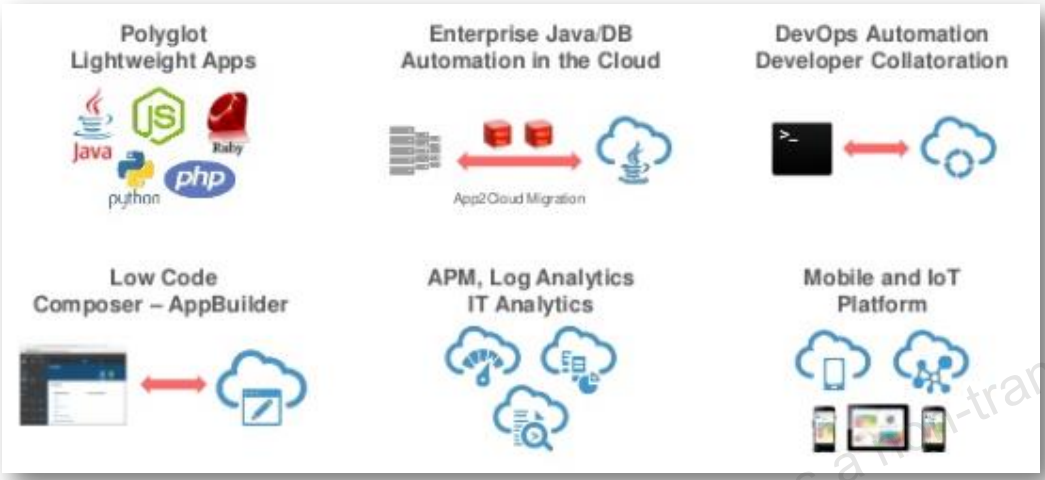

```
# The container definition we want to use for developing our app
box:
  id: php
  ports:
    - "8080"
```


What distinguishes PHP from something like client-side JavaScript is that the code is executed on the server, generating HTML, which is then sent to the client. The client would receive the results of running that script, but would not know what the underlying code was. You can even configure your web server to process all your HTML files with PHP, and then there's really no way that users can tell what you have up your sleeve.

The best things in using PHP are that it is extremely simple for a newcomer, but offers many advanced features for a professional programmer.

Supporting Platform

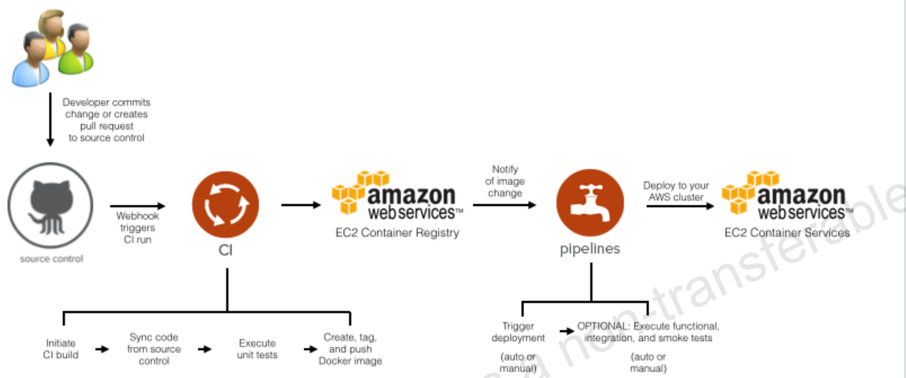
It is all about platforms integrating to Container Pipeline services. The versatility of the platform lies in inner core – Continuous Development.



You can look at Polygot approach of mixing up technologies at the same time looking at Enterprise Applications with Fast delivery with an awesome log analytics background supporting all kinds of devices.

Supporting Platform: Amazon ECS and ECR

Amazon Elastic Container Registry (Amazon ECR) is a managed AWS Docker registry service that is secure, scalable, and reliable. Amazon ECR supports private Docker repositories with resource-based permissions using AWS IAM so that specific users or Amazon EC2 instances can access repositories and images.



ORACLE

Amazon Elastic Container Service (Amazon ECS) is a highly scalable, fast, container management service that makes it easy to run, stop, and manage Docker containers on a cluster. You can host your cluster on a serverless infrastructure that is managed by Amazon ECS by launching your services or tasks using the Fargate launch type. Amazon ECS lets you launch and stop container-based applications with simple API calls, allows you to get the state of your cluster from a centralized service, and gives you access to many familiar Amazon EC2 features.

You can use Amazon ECS to schedule the placement of containers across your cluster based on your resource needs, isolation policies, and availability requirements. Amazon ECS eliminates the need for you to operate your own cluster management and configuration management systems or worry about scaling your management infrastructure.

Supporting Platforms: Heroku

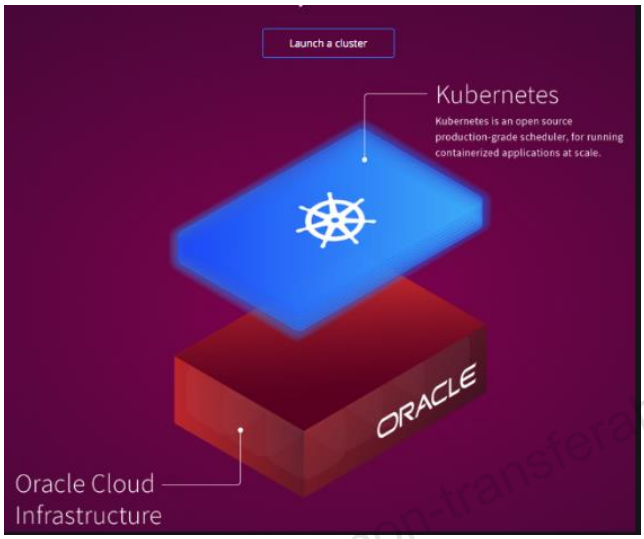
Heroku is a platform as a service based on a managed container system, with integrated data services and a powerful ecosystem, for deploying and running modern apps. The Heroku developer experience is an app-centric approach for software delivery, integrated with today's most popular developer tools and workflows.

Heroku uses Git as the main way to deploy your application, so you don't need to spend time learning tools you are not already using. When Heroku receives your git push, it works out what language and framework you have used and configures, builds, and deploys your application automatically. If your language or framework has a commonly used build tool, then Heroku installs that as well, so it builds your application in the same way you do locally.

Supporting Platforms: Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

It was originally designed by Google, and is now maintained by the Cloud Native Computing Foundation.



Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

The name Kubernetes originates from Greek, meaning helmsman or pilot.

Summary

In this lesson, you should have learned about:

- Platforms and Languages supported in Container Pipelines

.

Introducing Container Architecture

Running Containers with Speed But with Infra as Machines

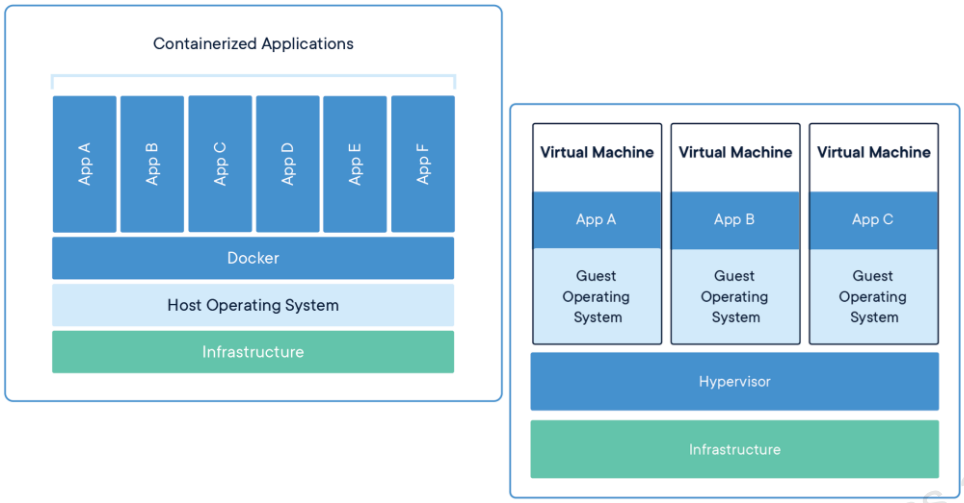
Objectives

After completing this lesson, you should be able to:

- Demonstrate working of containers
- Building container images
- Pushing and running images
- Accessing OCIR (OCI Registry)

What Is a Container?

A **container** is a standard unit of software that packages code and all its dependencies so that the application runs quickly and reliably from one computing environment to another.



Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications, and require fewer VMs and operating systems.

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries, and libraries—taking up tens of GBs. VMs can also be slow to boot.

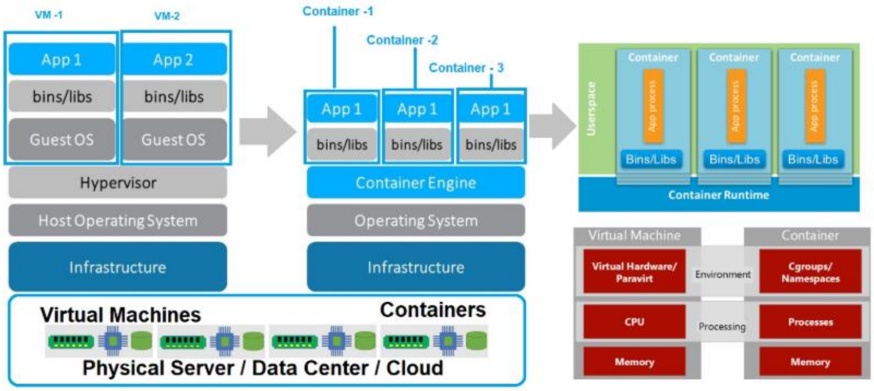
A container is a standard unit of software that packages code and all its dependencies so that the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, stand-alone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings.

Container images become containers at runtime, and in the case of Docker containers, images become containers when they run on Docker Engine. Available for both Linux- and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Docker containers that run on Docker Engine are:

- **Standard:** Docker created the industry standard for containers so they could be portable anywhere
- **Lightweight:** Containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs
- **Secure:** Applications are safer in containers, and Docker provides the strongest default isolation capabilities in the industry

The Container Architecture



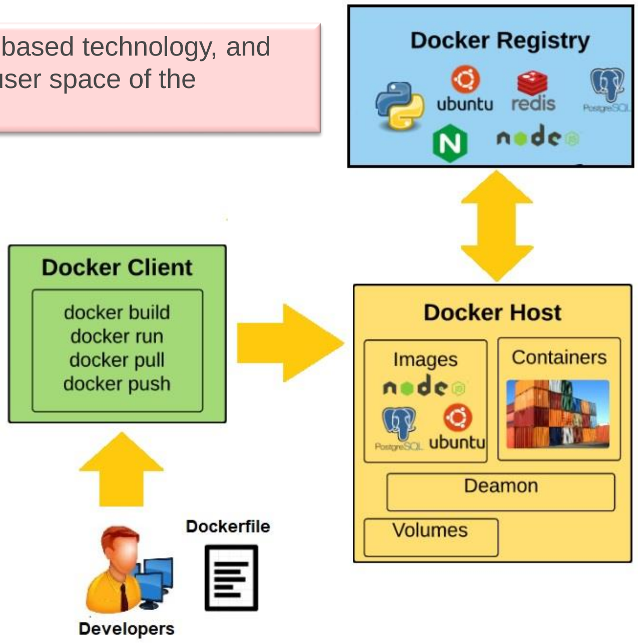
VMs and Containers are aimed at improving server utilization and reducing physical server sprawl and total cost of ownership for the IT department. However, if we can see the growth in terms of adoption in the last few years, there is an exponential growth in container-based cloud deployment.

ORACLE

Containers are a way of packaging software, mainly all of the application's code, libraries, and dependencies. They provide a lightweight virtual environment that groups and isolates a set of processes and resources such as memory, CPU, disk from the host and any other containers. The isolation guarantees that any processes inside the container cannot see any processes or resources outside the container. A container provides operating system-level virtualization by abstracting the user space. Containers also provide most of the features provided by virtual machines like IP addresses, volume mounting, resource management (CPU, memory, disk), SSH (exec), OS images, and container images, but containers do not provide an init system as containers are designed to run a single process. Generally, containers make use of Linux kernel features like namespaces (ipc, uts, mount, pid, network, and user) and cgroups for providing an abstraction layer on top of an existing kernel instance for creating isolated environments similar to virtual machines.

About Docker

Docker is container-based technology, and containers are just user space of the operating system.



ORACLE

Docker is container-based technology, and containers are just user space of the operating system. At the low level, a container is just a set of processes that are isolated from the rest of the system, running from a distinct image that provides all files necessary to support the processes. It is built for running applications. In Docker, the containers running share the host OS kernel. Docker is an open-source project based on Linux containers. It uses Linux Kernel features like namespaces and control groups to create containers on top of an operating system. Docker encloses an application's software into an invisible box with everything it needs to run like OS, application code, run-time, system tools, and libraries. Docker containers are built off Docker images and the images are read only. So Docker adds a read-write file system over the read-only file system of the image to create a container.

Docker Client: This is the CLI (command-line interface) tool used to configure and interact with Docker. Whenever developers or users use the commands like docker run, the client sends these commands to Docker daemon (dockerd), which carries them out. The Docker command uses the Docker API, and the Docker client can communicate with more than one daemon.

Docker Daemon: This is the Docker server that runs as the daemon. This daemon listens to API requests and manages Docker objects (images, containers, networks, and volumes). A daemon can also communicate with other daemons to manage Docker services.

Images: Images are the read-only template/snapshot used to create a Docker container, and these images can be pushed to and pulled from public or private repositories. This is the build component of Docker. Images are lightweight, small, and fast as compared to Virtual Machine Images.

Docker file: Used for building images

Containers: Applications run using containers, and containers are the running instance of an image. We can create, run, stop, move, or delete a container using the CLI and also can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

Docker Registries: This is the distribution component of Docker or called as central repository of Docker images, which stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. When we use the Docker pull or Docker run commands, then the required images are pulled from our configured registry. When we use the Docker push command, our image is pushed to our configured registry. We can upgrade the application by pulling the new version of the image and redeploying the containers.

Docker Engine: Combination of Docker daemon, Rest API, and CLI tool

What Is a Docker Image?

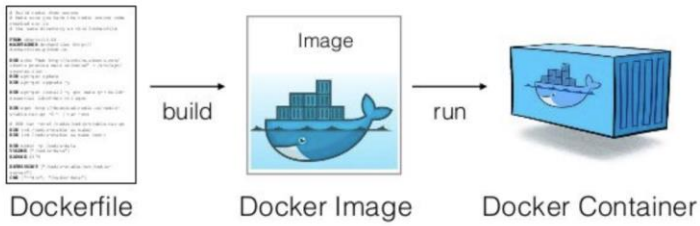
Read-only template with instructions for creating a container

- Built over another image with customization
- Published in registry
- Based on layered architecture
- Is compressed and encrypted – SHA256

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image that is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers that have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Custom Build Docker Image



- Dockerfile is a text file encompassed with keywords.
- Building a Dockerfile will get a Docker image with layered architecture.
- One can push Docker image to a custom-build registry
- Running an image will yield a container.

Dockerfile

- Dockerfile is a text file
- Naming notation is Dockerfile (default file) with no extension
- Docker keywords are case neutral
- Comments are marked by #

Keyword	Context
FROM	The source base kernel
RUN	Core skeleton of the image Binary,source code.
ENTRYPOINT	Boot Strapping – executes when a container is instantiated
CMD	-- args parameter of the boot strapping
MAINTAINER	Metadata of the Docker image

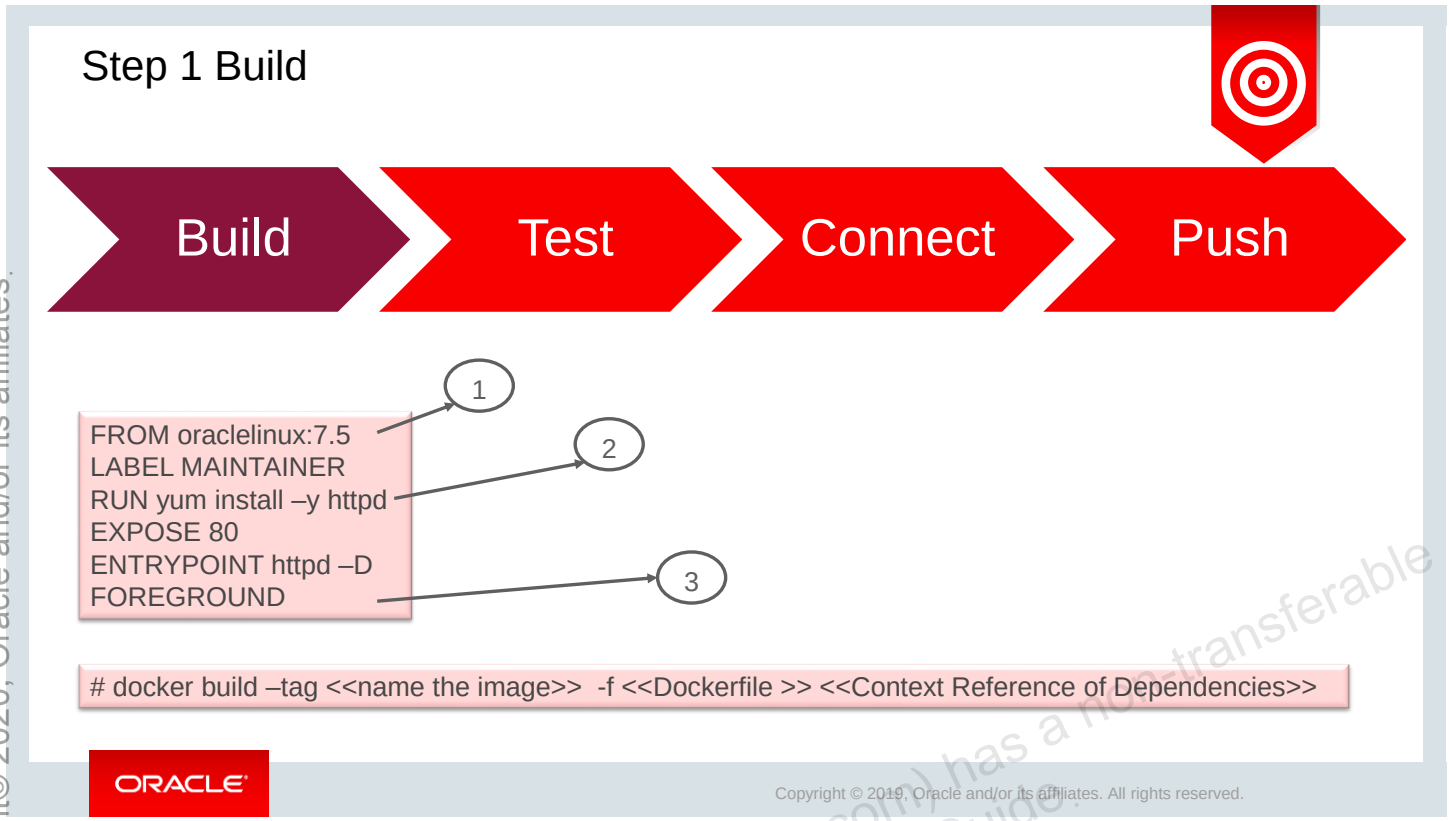
A Dockerfile is a text configuration file written using a special syntax.

It describes step-by-step instructions of all the commands you need to run to assemble a Docker image.

How Do We Host the Image?

The steps to build the image are the following:

- Build the image from Dockerfile
- Test the image built
- Log in to the OCIR and tag the image
- Push the image to OCIR



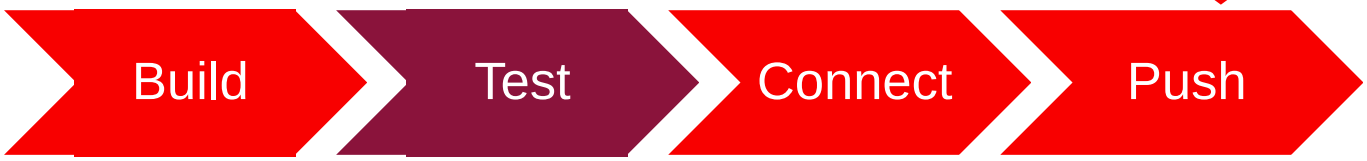
There are two types of image build – delivering image from a container via Commit and an automation of Commit via Dockerfile.

The intent of docker build is recursive commit of the lines in Dockerfile. Each statement will be executed in the background and will be committed as a layer. Once committed, the intermediate container created will be pruned. As the build progresses, the finer image is built with layers and with definition specification of Docker image.

1 – Base image from where this image is created
2 – Installing binary for the image

3 – Service to be instantiated when the image is instantiated

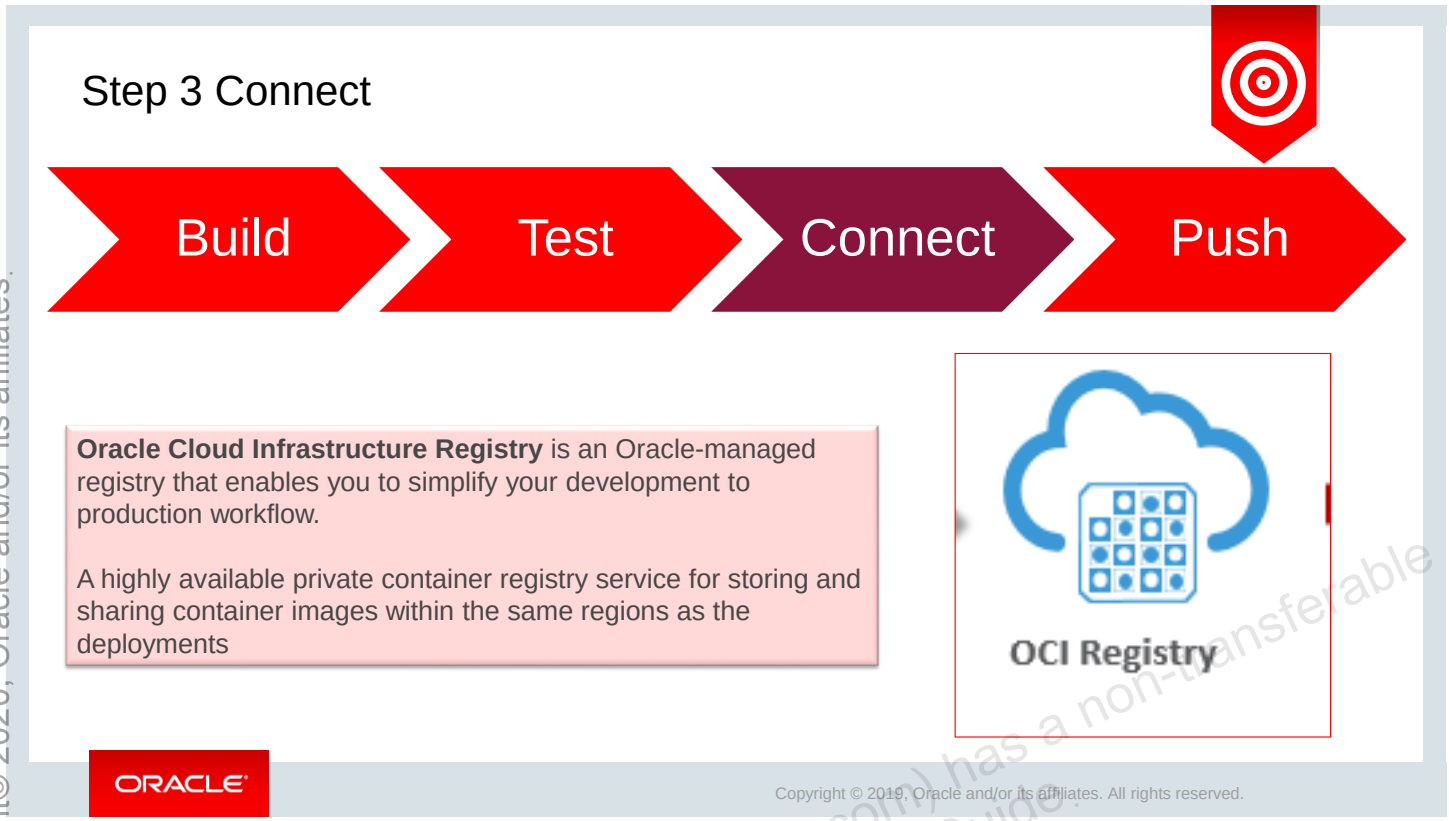
Step 2 Test



The logical reason for testing an image is to create a container.

docker run –d –name testhttp –P ol:httpd

-d detached mode
-P Exposing port to Dynamic Port
--name <<name of the Container>

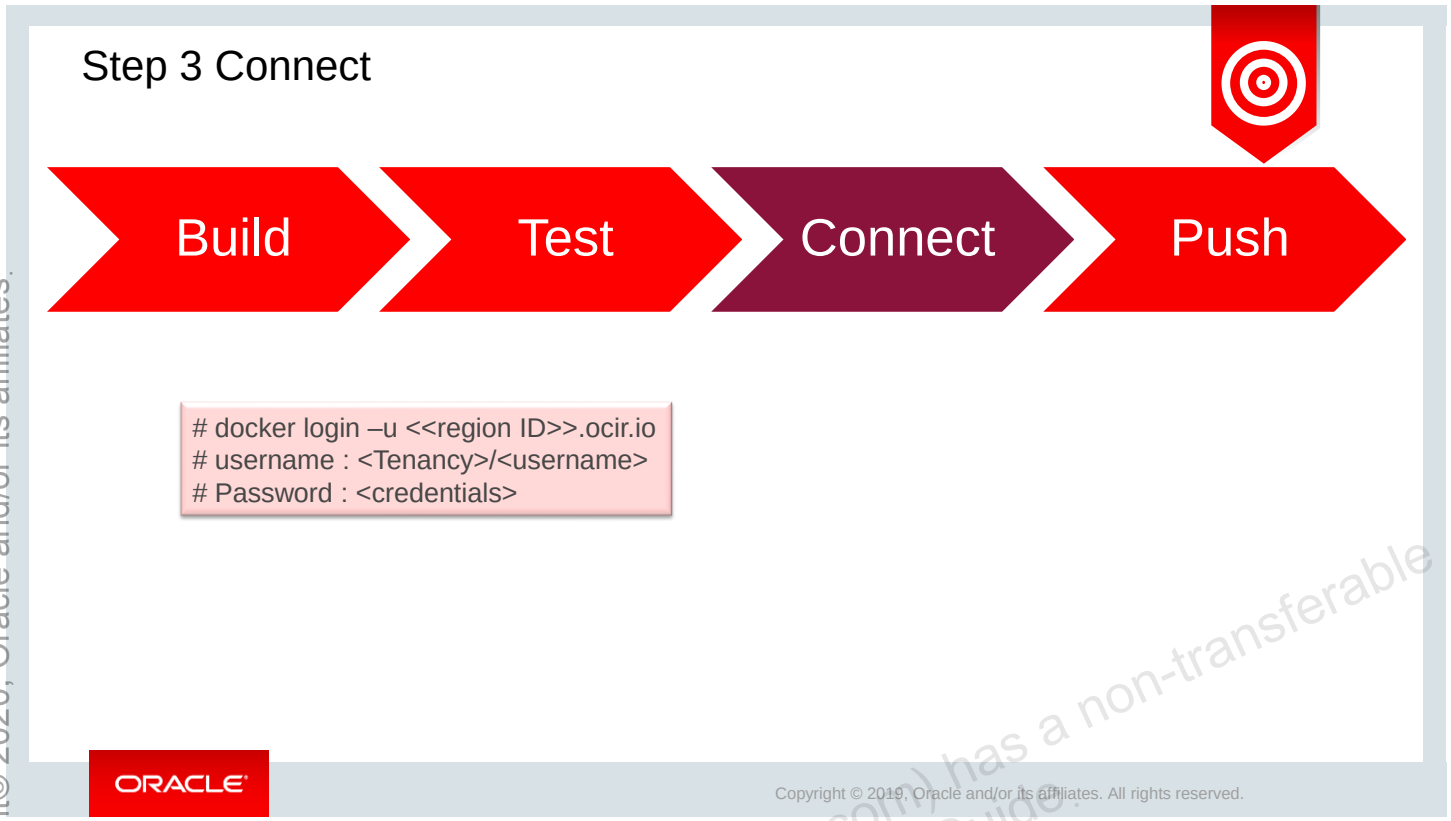


OCIR makes it easy for you as a developer to store, share, and manage development artifacts like Docker images.

The highly available and scalable architecture of Oracle Cloud Infrastructure ensures you can reliably deploy your applications, so you don't have to worry about operational issues or scaling the underlying infrastructure.

You can use Oracle Cloud Infrastructure Registry as a private Docker registry for internal use, pushing and pulling Docker images to and from the Registry using the Docker V2 API and the standard Docker command-line interface (CLI). You can also use Oracle Cloud Infrastructure Registry as a public Docker registry, enabling any user with Internet access and knowledge of the appropriate URL to pull images from public repositories in Oracle Cloud Infrastructure Registry.

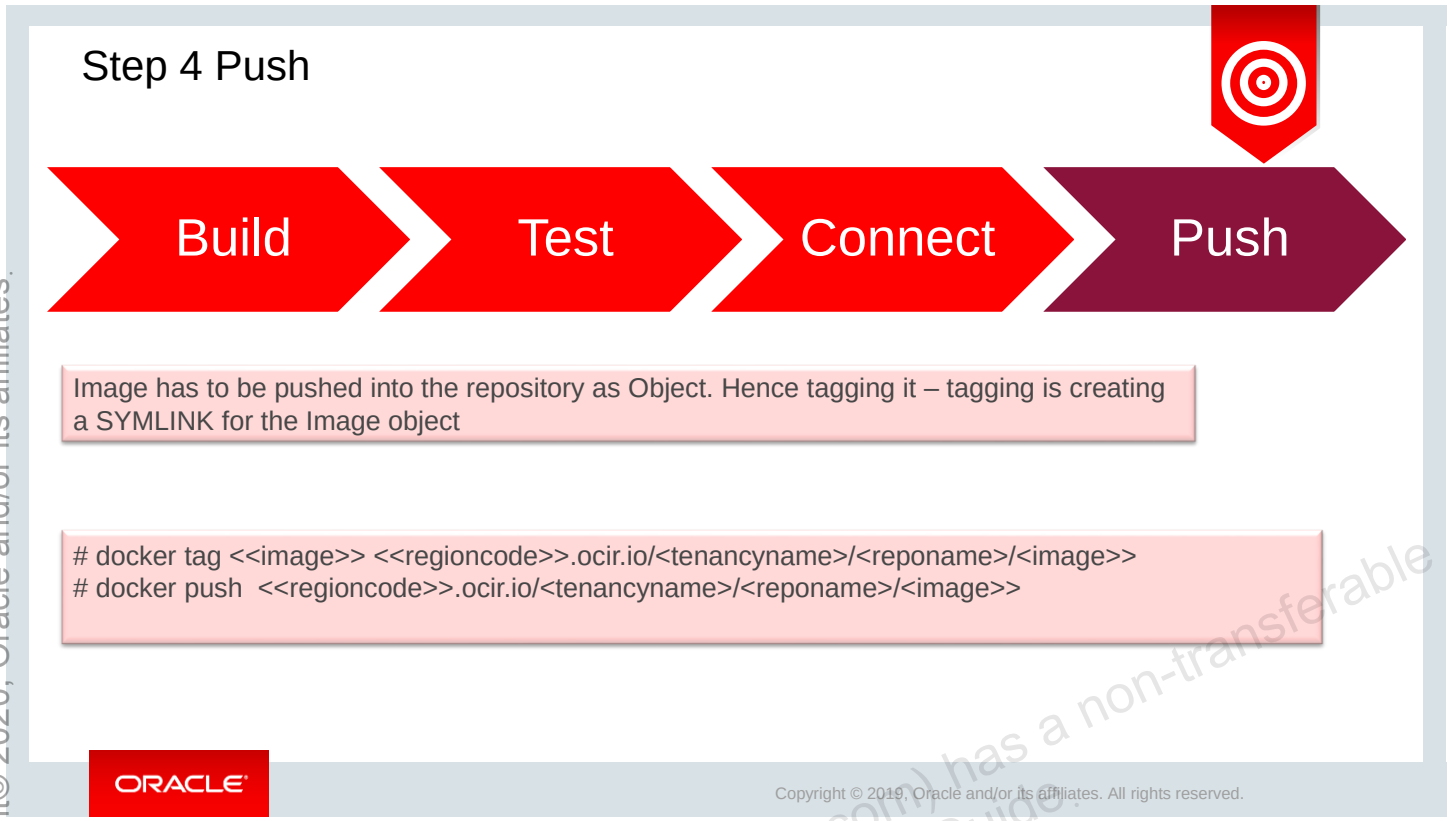
Oracle Cloud Infrastructure Registry supports private access from other Oracle Cloud Infrastructure resources in a virtual cloud network (VCN) in the same region through a service gateway. Setting up and using a service gateway on a VCN lets resources (such as worker nodes in clusters managed by Container Engine for Kubernetes) access Oracle Cloud Infrastructure services such as Oracle Cloud Infrastructure Registry without exposing them to the public Internet. No Internet gateway is required, and resources can be in a private subnet and use only private IP addresses. Oracle Cloud Infrastructure Registry is integrated with IAM, which provides easy authentication with native Oracle Cloud Infrastructure identity.



In a terminal window on the client machine running Docker, log in to Oracle Cloud Infrastructure Registry by entering:docker login <region-code>.ocir.io, where <region-code> is the code for the Oracle Cloud Infrastructure Registry region you're using. For example, phx. See the Availability by Region Name and Region Code topic in the Oracle Cloud Infrastructure Registry documentation for the list of region codes.When prompted, enter your username in the format <tenancy-namespace>/<username>.

For example, ansh81vru1zp/jdoe@acme.com. If your tenancy is federated with Oracle Identity Cloud Service, use the format <tenancy-namespace>/oracleidentitycloudservice/<username>.

When prompted, enter the auth token you copied earlier as the password.



The intent is to push the image to your repository in OCIR.

In a terminal window on the client machine running Docker, give a tag to the image that you're going to push to Oracle Cloud Infrastructure Registry by entering:

docker tag karthequian/helloworld:latest <region-code>.ocir.io/<tenancy-namespace>/<repo-name>/<image-name>:<tag>where:<region-code> is the code for the Oracle Cloud Infrastructure Registry region you're using.

- ocir.io is the Oracle Cloud Infrastructure Registry name.
- <tenancy-namespace> is the auto-generated Object Storage namespace string of the tenancy (as shown on the **Tenancy Information** page) to which you want to push the image. For example, the namespace of the acme-dev tenancy might be ansh81vru1zp. Note that your user must have access to the tenancy.
- <repo-name> (if specified) is the name of a repository to which you want to push the image (for example, project01). Note that specifying a repository is optional. If you don't specify a repository name, the name of the image is used as the repository name in Oracle Cloud Infrastructure Registry.
- <image-name> is the name you want to give the image in Oracle Cloud Infrastructure Registry (for example, helloworld).
- <tag> is an image tag you want to give the image in Oracle Cloud Infrastructure Registry (for example, latest).

Practice 3: Overview

- You will learn to create Docker image and test them as containers
- You will learn to push images into Oracle Cloud Registry (OCIR)

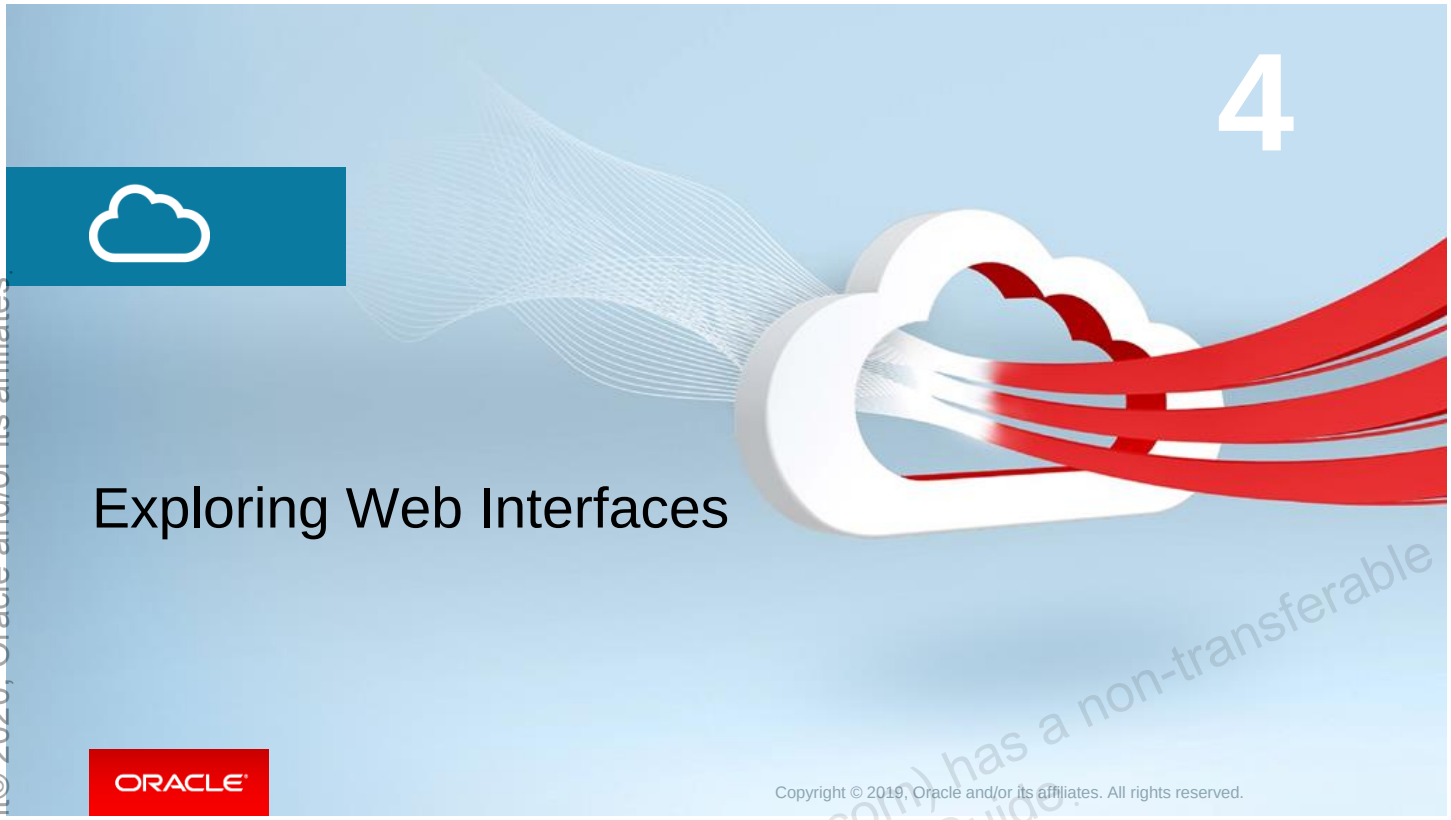
Summary

In this lesson, you should have learned how to:

- Demonstrate working of containers
- Build container images
- Push and run images
- Access OCIR (OCI Registry)

4

Exploring Web Interfaces



Objectives

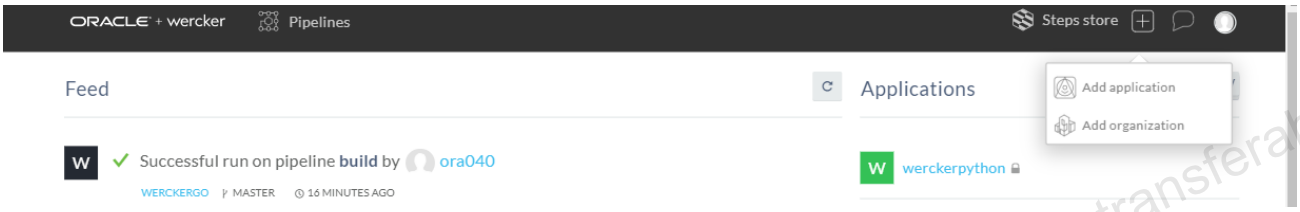
After completing this lesson, you should be able to:

- Explore web interfaces of Container Pipelines
- Add application and pipelines
- Add steps and provide access to users
- Utilize organizations in Container Pipeline

Web Interfaces in Container Pipeline

One can interface with Container Pipeline with the following methods:

- WERCKER CLI
- WERCKER API
- WERCKER GUI



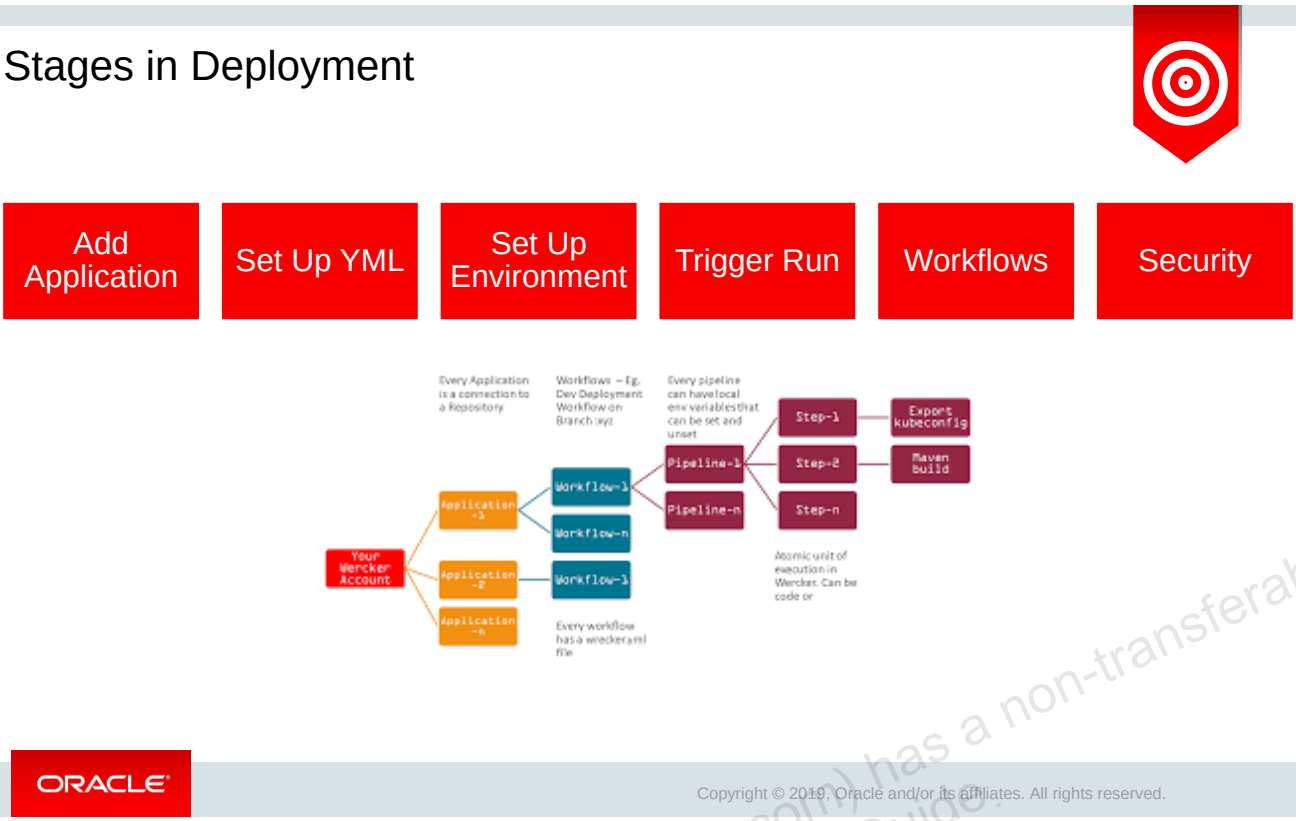
The Wercker GUI is the Wercker WEB Interface. Wercker CLI is deprecated.

When you choose Add an Application, choose the owner of the new application. You can either specify your own user account or choose from a list of organizations where you're a member of the organization's owners team.

If you select an organization, note that by default your Git provider credentials will be used for API operations. If you want a different user's credentials to be used, you can go back and change that later (by selecting an alternative user from the **API user** list on the **Options** page).

Choose the Git provider containing the repository from which the application will be created.

Stages in Deployment



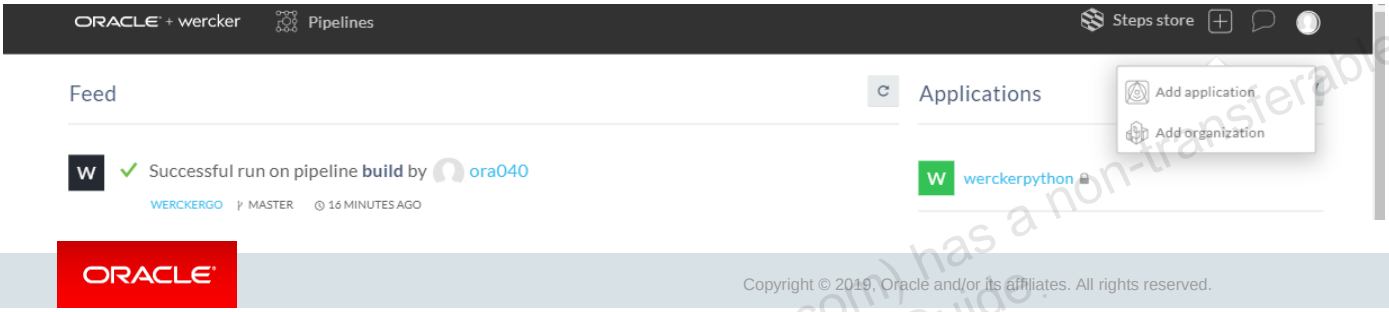
The stages in deploying for the CI/CD pipeline are divided into six stages as depicted above in the slide. Each stage has a focus area.

Stage 1: Add an Application

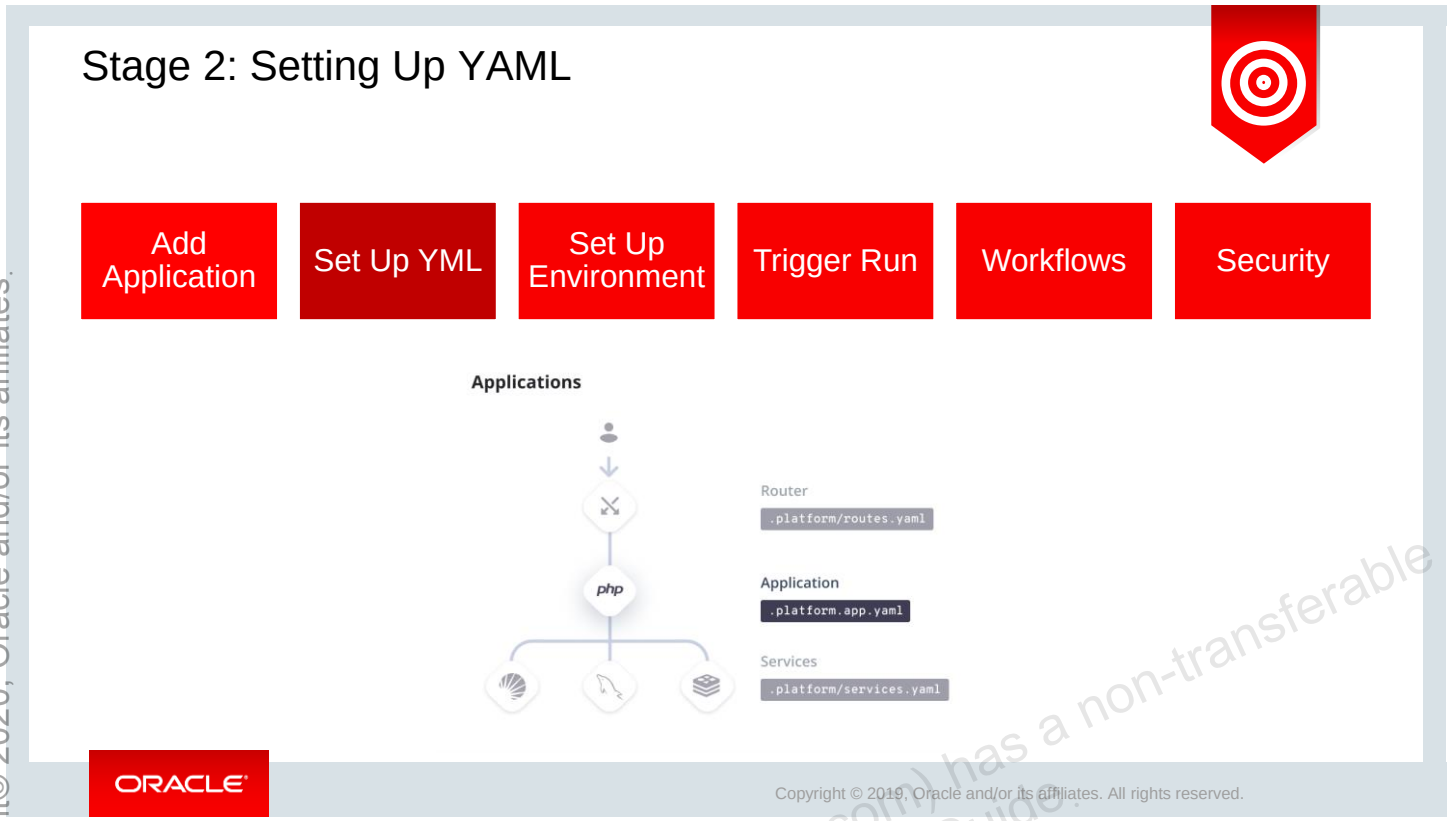
Add Application | Set Up YML | Set Up Environment | Trigger Run | Workflows | Security

You create an application in Wercker from a repository in your Git provider (GitHub, Bitbucket, or GitLab).

You add new applications to Wercker using a simple wizard. The wizard takes you through a couple of guided steps to get the best setup from the start.



Add an Application represents adding a Source code into your repository. All relevant code will be available here in this REPO. You can then build a Docker image.



YAML—short for “YAML Ain’t Markup Language”- is a human-readable, data-oriented markup language used to parse wercker.yml files. While there are other popular data formats that can be used (like XML and JSON), YAML is a perfect choice for configuration files because it is easy for humans to read and write, and nearly every popular programming language offers libraries for programmatically working with YAML.

Before we get into the various components and data structures available within YAML, let’s get some basic rules out of the way:

Indentation

YAML is a whitespace-indented language, meaning that indentation is used to denote structure. Items of the same indentation are considered to be siblings, while more or less indentation denotes child and parent relationships, respectively.

Because they are universally supported, spaces are the whitespace character of choice within YAML files, and tabs are never allowed. Indenting with tabs instead of spaces within a YAML file is a common mistake for beginners and can be very difficult to diagnose, so take special care to use the proper indentation method whenever working with YAML. To help mitigate this issue, it is highly recommended to display whitespace characters in your editor, otherwise, it could be difficult to tell if the following code block is valid or not:

While some languages - like PHP and MySQL - are not case-sensitive, YAML is not so.

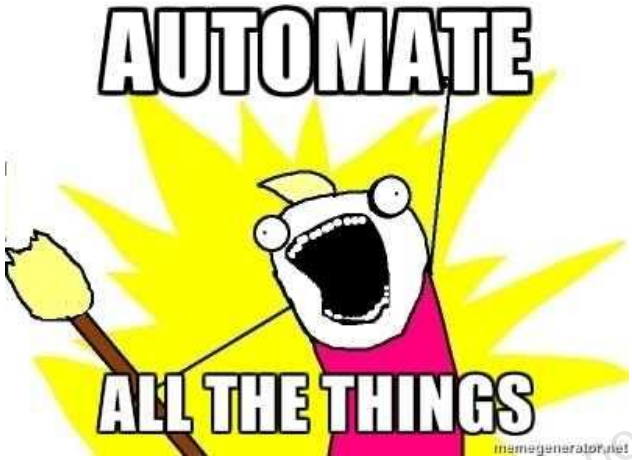
Once you get the basic rules out of the way, YAML is a relatively straightforward language consisting of only a handful of standard components, the most common of which are scalars, sequences, and mappings.

Scalars integers, floats, booleans, strings…these are scalars. While YAML has strict rules around indentation and case sensitivity, it is relatively flexible when it comes to scalars. Each type has its validation rules, allowing you to define values in a way that is appropriate for the configuration file:

YAML Lint - http://www.yamllint.com/- A basic YAML syntax validator

Set Up YAML: wercker.yml (1)

The `wercker.yml` defines the configuration of your automation pipelines with a collection of steps that you wish to execute.



In your wercker.yml, you can specify any pipeline you like. There is one special pipeline called dev, which will only be executed when running it with the CLI using the wercker dev command. Examples of pipeline names: build-base-container, build, push-to-registry, deploy-Kubernetes

Upon creating a project in Wercker, we automatically create a build Workflow with a Git hook that executes a buildpipeline.

A pipeline can have its own base box. For example, it might use a Docker container as a starting point. You can use different base boxes for each pipeline.

Each pipeline can also specify its own Services. For example, a testing pipeline may require access to a database server, whereas a deploy pipeline probably would not.

Set Up YAML: wercker.yml (2)

```
box: nodesource/trusty
# Build definition
build:
  # The steps that will be executed on build
  steps:
    # A step that executes `npm install` command
    - npm-install
    # A step that executes `npm test` command
    - npm-test

    # A custom script step, name value is used in the UI
    # and the code value contains the command that get executed
    - script:
        name: echo nodejs information
        code: |
          echo "node version $(node -v) running"
          echo "npm version $(npm -v) running"
```


A wercker.yml defines the steps required to execute automation tasks for your application, along with the pipelines that group them.

Here is a working example for a Node.js application, built from the nodesource trusty Docker base image, which would be automatically pulled at run time.

This wercker.yml defines a build pipeline (Step 1) that executes the npm-install and npm-tests, followed by a script step (Step 3) that executes some inline code for displaying the node and npm versions.

Set Up YAML: Adding Steps (3)

Steps are self-contained bash scripts or compiled binaries for accomplishing specific automation tasks, defined in the wercker.yml file of your application. Steps can be written manually or borrowed from the community via the Steps Registry.

```
build:
    steps:
      - npm-install@1.0.5:
          package: jshint
          strict-ssl: false
      - npm-test
```

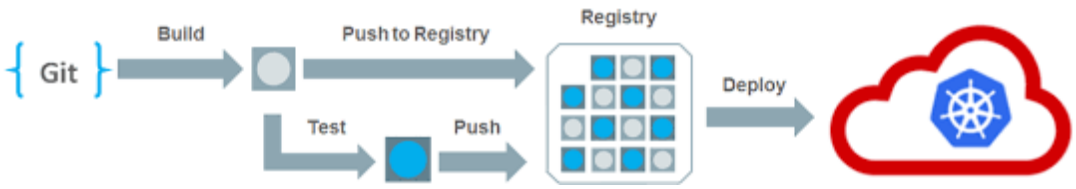

In the above slide, this will pass two parameters to the npm install step: package and strict-ssl.

Apart from predefined steps, there are also custom steps, also known as inline steps. Custom steps allow you to run bash scripts directly within your pipelines.

Wercker also has the notion of after-steps, which are ideally suited for notifications.

Set Up YAML: Internal Steps (4)

Internal steps are developed by Wercker and are baked into the Wercker CLI. Internal steps interact with the Docker API that is external from the container. From a technical perspective, it is not possible to interact with the Docker daemon from within a container.



This is why we created these internal steps.

The **internal/docker-push** step can be used in two ways.

You can use it to push an image that has been built using the internal/docker-build step to a Docker registry. You must set the image-name property to identify the image you wish to push, as shown in the following example. This example uses Docker Hub, but you can use any image registry.

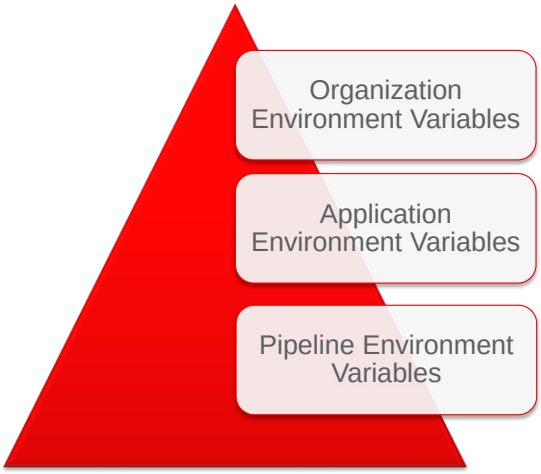
The **internal/docker-push step** supports the following properties:

- image-name: This property is only used when pushing an image created previously using a docker-build step. It indicates the name of the image to be pushed. If you choose to set this property, then specify the same name that you specified while running the internal/docker-build step. However, if you omit this property, then the currently running pipeline container is committed and pushed. The internal/docker-push step supports the following properties:
- username: Indicates the user name that is used to authenticate the registry
- password: Indicates the password that is used to authenticate the registry
- email: Indicates the email address that is used to authenticate the registry
- repository: Indicates the name of the repository. If you are using a repository other than Docker Hub, prefix the value with the hostname of the private repository.
- tag: A space- or comma-separated list of Docker tags that are applied to the built container. Wercker automatically applies a tag with the build number, and if a tag is left blank, Wercker uses the docker default of latest.
- registry: Indicates the end point of the registry. Leave this property empty if you're pushing the image to Docker hub. However, if you're pushing the image to other registries, then begin the registry name with https:// and make sure that it has the same prefix as that of the repository. Append /v2 to the URL to push to registries that support V2 of the registry API.

The internal/watch step gives you a long-running step that can be configured to reload on file changes. A very common use case for this step is front-end development.

Stage 3: Set Up Environment

What if you have an API key you need during a pipeline run? This is information that is unique for each server you want to deploy to. It may also be too sensitive to store in the repository.



Wercker supports a number of ways to store and expose this data using environment variables.

Organization-level environment variables

When you have an organization, you can create environment variables on that organization from the organization settings. These environment variables will be available to all projects and pipelines in that organization. These can be used to set some global settings that should apply to everything under an organization, such as an SSH key to fetch private GitHub repositories. Keep in mind that public projects under the organization will also have access to these variables.

Application-level environment variables

The second set of variables are specific to the application. All runs under this application will receive these environment variables. This can be useful for setting AWS secrets, Docker credentials, or anything else that several pipelines in the application might use.

In the case of an identical name, an environment variable defined on an application will override an environment variable defined on the organization.

Pipeline-level environment variables

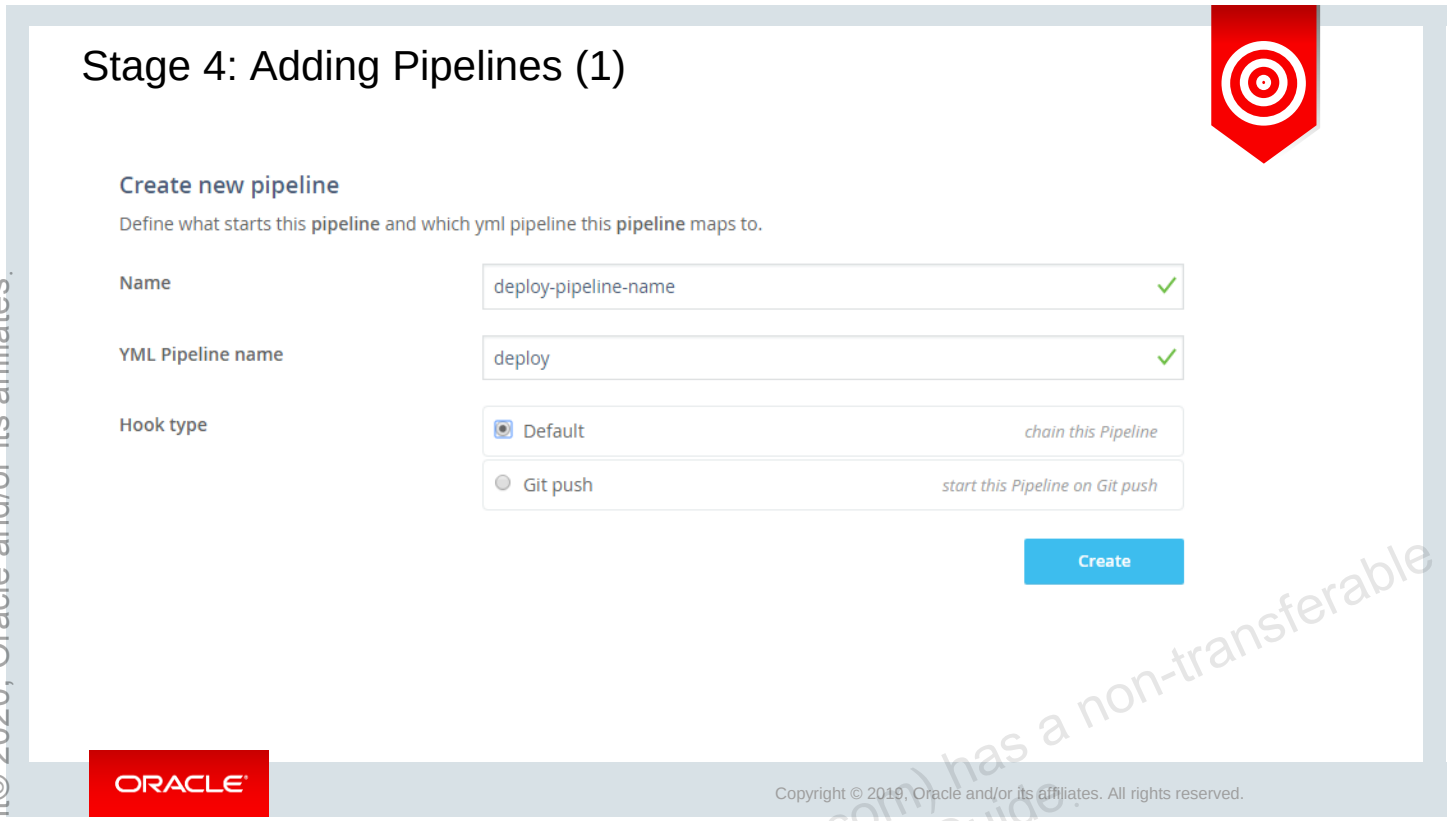
The third set of variables are specific to pipelines in a workflow. These are made available to each pipeline run. Typically, these are used to store information such as hostnames, SSH keys, passwords, and more.

The pipeline environment variable will override environment variables with the same name on the organization or the application.

Available environment variables

You can also use a script step and use the env command to see the full list of all variables at that moment during the pipeline execution. This is convenient since there are reasons the environment variables step does not show all environment variables available during the pipeline run.

Stage 4: Adding Pipelines (1)



While all Wercker applications ship with the build pipeline out of the box, this isn't always enough. You may also need to add custom pipelines to come into play. With pipelines, automating a build process in stages or when code changes is a nearly painless process and takes only minutes to set up.

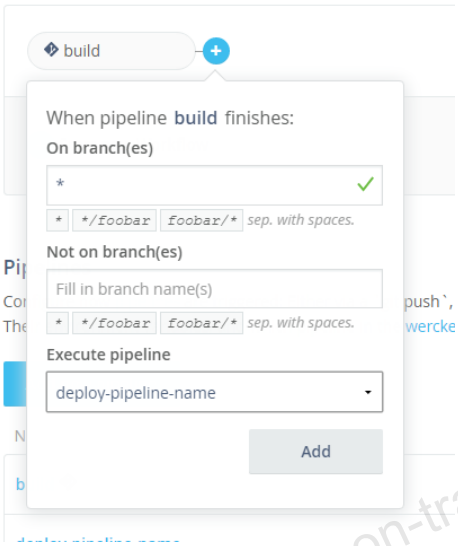
Wercker pipelines are just isolated build scripts that can be run independently of one another. This means that however you design your build process (the box you run, environment variables you define, services you spin up, etc.), the same level of configuration can be achieved for any additional pipeline.

Once Wercker automatically creates the build pipeline when you import your application, half of the work is done, but what about the other half? To run the deploy pipeline, we first must create a new pipeline within our application's Wercker dashboard.

This can be accomplished by clicking on "Add new pipeline" under the "Workflows" tab. While the Name and YML Pipeline Name inputs should be self-explanatory, the Hook Type might not be as clear. This value is used to decide when to run the pipeline: Default runs whenever you tell it to run, and can be automated using workflows, and Git Push runs whenever you push new code up to designated branches.

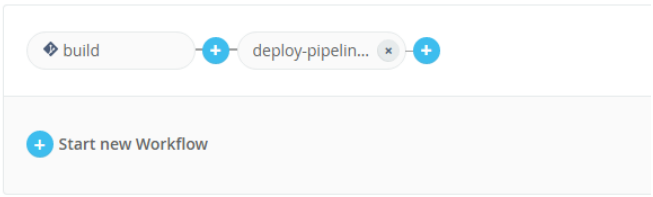
Stage 4: Adding Pipelines (2)

This step is only necessary if the Default hook type was chosen.



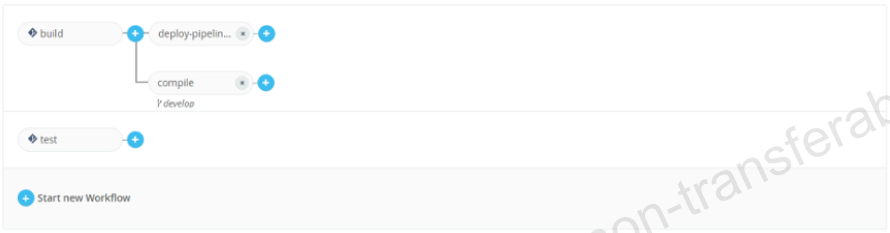
Once we've created our pipeline, we then need to add it to a workflow. To do this, you can either start a new workflow or (in the case of a standard build -> deploy process), you can configure it to run after an existing step.

Stage 4: Adding Pipeline (3)



1

2



1 → As you can see from the slide, our demo deploy pipeline has been configured to run when the build pipeline finishes. It is important to note that with workflows, your configured pipelines can be executed serially or in parallel under any number of conditions, allowing for incredibly complex build processes with minimal setup.

2→ That's it!

You've successfully created your first Wercker pipelines. On your next build, you should see your new pipeline run immediately after the build step. We can make the pipeline powerful when it can be for automating multiple builds, deploying code and assets to different servers, and running branch-specific scripts.

Providing Access

Currently there are three roles defined within wercker:

- execute pipelines
- execute and manage pipelines
- admin

Execute Pipelines

A user with Execute Pipelines permissions can see pipeline runs, view deploy logs, and can follow/unfollow applications.

Execute & Manage Pipelines

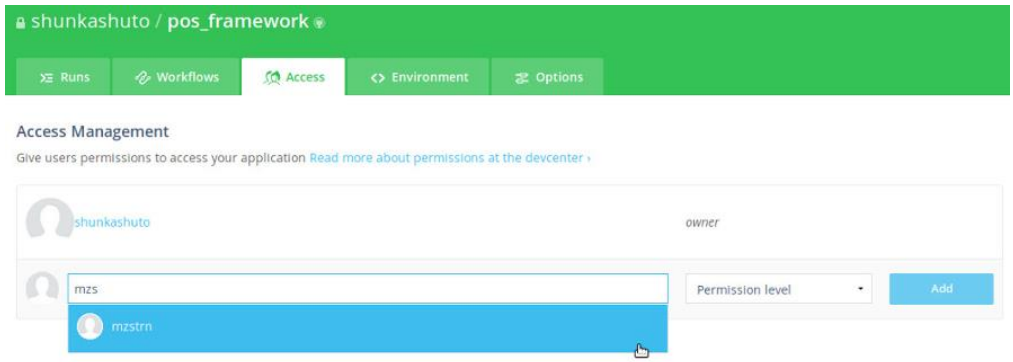
A user with Execute & Manage Pipelines permissions can do all things a user with Execute Pipelines permissions can, but can also edit and join Pipelines in to Workflows

Admin

Like the owner of an application, users with admin permissions can do the same as with Execute & Manage Pipelines permissions and also:

- change permissions for collaborators
- manage the SSH Keys
- change public /private setting
- change the repository access
- change the infrastructure stack
- change the API user
- change the ownership of a application
- delete the application
- activate/deactivate support for the application from Options tab

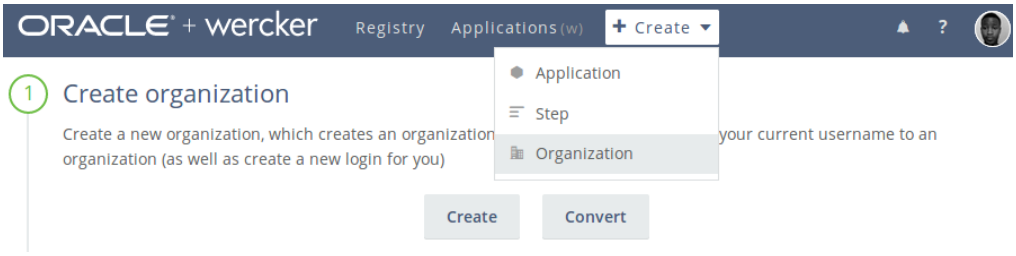
Providing Access



To add a user, enter the name or address, select the permission level (Execute Pipelines, Execute & Manage Pipelines, or Admin), and click Add. The new user will be added to the list. To access an application, click Applications in the Wercker toolbar. This will bring up the Applications page, listing all of the applications to which you have access.

Using Organizations

Through organizations, you can create and organize teams to make access management on applications way easier and more efficient.



You can start to create a new ‘Organization’ from the create drop-down list or from your profile settings page.

As you can see, you can choose between two options on how to create a new organization.

Create

Creating a new organization only requires you to fill in an organization name and contact email address. This email address is also used to create your organizations avatar with Gravatar.

Convert

With convert, you can use the username that you are logged in with, to convert your user to a new organization with the name of the user who is currently logged in.

Please note that converting a user to an organization is a permanent action.

Fill in a contact email address. This email address will also be used to create your organization’s avatar with Gravatar.

Pick a new username

Because your current username will be used as the organization’s name, you will need to enter a new username for the current user with which you can log in.

Once you have completed converting your user, you will be logged out and in, in order to finalize the process.

Finished

When you have created your organization, a new owners team is added as the first team. The owners team is responsible for the applications, people, teams, and access management of an organization.

Practice 4: Overview

You will learn how to utilize Container Pipeline using Wercker via Oracle Cloud and also demonstrate CI/CD operations.

Summary

In this lesson, you should have learned how to:

- Explore web interfaces of Container Pipelines
- Add application and pipelines
- Add steps and provide access to users
- Utilize organizations in Container Pipeline

5

Development Workflow

Objectives

After completing this lesson, you should be able to:

- Explore API/end points of container pipelines
- Experience CLI usage
- Create steps and explore steps
- Manage pipeline and workflows
- Integrate YAML in API

Wercker API

The Wercker platform consists of four main tools:
- Wercker Web
- Wercker Command-Line Interface (CLI)
- API
- Steps Marketplace (Steps Registry)

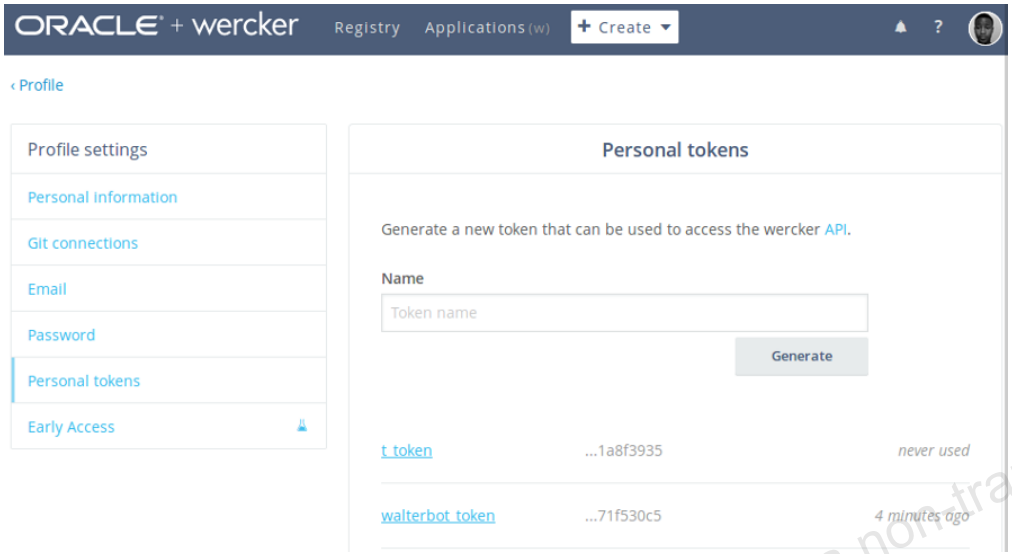
Using the Wercker API, you can control tasks in Wercker using HTTP end points. The API is currently in alpha.

All end points in the documentation use relative paths. The host and the prefix path need to be the following:

```
https://app.wercker.com/api/v3/
```

The slide shows the list of HTTP Verbs and Errors that will be used for HTTP end points. REST API versions and errors are based on REST versions implemented.

End Points: Authentication



For some end points, you do not need to authenticate. For example, you're able to get the details from a public project without authenticating. Most end points, however, do require authentication.

Currently Wercker doesn't support a full OAuth 2.0 implementation. But you can create and manage your tokens for your account on the personal profile page under personal tokens. You can have multiple tokens active at the same time.

Make sure to keep these tokens secret because they are the equivalent of your username and password. We will store a hashed version of the token while sending the token to you. Make sure, therefore, that you store the token right after creating it, because you won't be able to retrieve it again.

Authorization Token

The recommended way to use a token is to use the Authorization header. This is the header format that is used:

```
Authorization: Bearer <TOKEN>
```

```
curl -H 'Authorization: Bearer <TOKEN>' https://app.wercker.com/api/v3/applications/wercker/docs
```


It is also possible to add the token as a query string parameter. This, however, is not recommended and should only be used if it is impossible to use the header method. You should use the token query string key, and the value should be the token. The example shown is with curl command as depicted in the slide.

Client Library: Wercker API

This example will be using “request” library of nodejs connecting to Wercker API.

Below is a small code sample to authenticate and retrieve data about an application on wercker in Node.js using the request library.

Client Library: Wercker API

```
var request = require('request');

var options = {
  url: 'https://app.wercker.com/api/v3/applications/wercker/docs',
  headers: {
    'Authorization': 'Bearer <TOKEN>'
  }
};

function callback(error, response, body) {
  if (!error && response.statusCode == 200) {
    var result = JSON.parse(body);
    console.log(result)
  }
  else {
    console.log(error)
  }
}

request(options, callback);
```


Let's look into the code snippet here of nodeJS.

1. → is the url of REST API bearing the bearer token of authorization
2. → Using the request library of node js for managing HTTP request
3. → Invoking the request function

Client Library: Wercker

This example implements using Python. The code uses Requests of Python library

Client Library: Wercker

Here in this example of Python file, you can observe url and headers having REST API end points from wercker having the bearer authorization.

```
import requests
import json

url = 'https://app.wercker.com/api/v3/applications/wercker/docs'
headers = {'Authorization':'Bearer <TOKEN>'}
r = requests.get(url, headers=headers, verify=True)
print r.json()
```

End Point: List User, Get Details of Applications

List all applications owned by the user or organization. The result will only contain applications that the authenticated user has access to. If the call is made without a token, only public applications will be returned.

```
GET /api/v3/applications/:username
```

Get the details of a single application.

```
GET /api/v3/applications/:username/:application
```

Querystring values

Name	Description
stack	Stack used by application. Currently supported: 1 (classic), 6 (Docker enabled)
limit	Limit the results that will get returned. Default: 20. Min: 1. Max: 100.
skip	Skip a certain amount of builds.
sort	Sort builds using this key. Default: nameAsc. Possible values: nameAsc, nameDesc, createdAtAsc, createdAtDesc, updatedAtAsc, updatedAtDesc.

End Point: Updates Applications

Update a single application. Currently, it is only possible to change the ignored branches for the application. The updated application will be returned.
`PATCH /api/v3/applications/:username/:application`

Retrieve all builds of an application.
`GET /api/v3/applications/:username/:application/builds`

Querystring values

Name	Description
branch	Branch of the commit that triggered a build.
commit	Commit of the build that triggered the build. 40 characters.
result	Result of the build. Possible values: aborted, unknown, passed, failed.
stack	Stack used to run a build. Currently supported: 1 (classic), 6 (Docker enabled)
status	Status of the build. Possible values: notstarted, started, finished, running.
limit	Limit the results that will get returned. Default: 10. Min: 1. Max: 20.
skip	Skip a certain amount of builds.
sort	Sort builds using this key. Default: creationDateDesc. Possible values: creationDateAsc, creationDateDesc.

End Point: Runs Details of Pipeline

Get the last 20 runs for a given pipeline or application.
Returns an array of run objects.
`GET /api/v3/runs`

Get the steps for a run.
Returns an array of step objects.
`GET /api/v3/runs/:runId/steps`

Trigger a new run for an application.
Returns a run object.
It is possible to add environment variables, which will be added to the run. The order of the array will be maintained, which makes it possible to use environment variables that were defined earlier. Any environment variables defined as part of the application or workflow will be overwritten, if using the same key.
`POST /api/v3/runs/`

End Point: Workflows

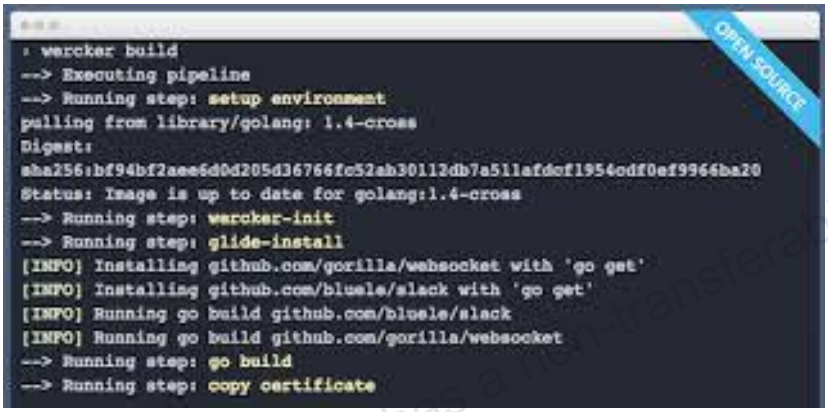
Get the last 10 workflows.
Returns an array of workflow objects.
`GET /api/v3/workflows`

URL querystring paramaters

Name	Type	Description
applicationId	String	Required The ID of the application.
limit	Integer	Optional Specify how many workflow objects should be returned. Max: 20, default: 10
skip	Integer	Optional Skip the first X runs. Min: 0, default: 0
sort	String	Optional Valid values: creationDateAsc or creationDateDesc. Default creationDateDesc

Wercker CLI

The Wercker Command Line Interface (CLI) is an application that replicates the behaviour of Wercker in your local development environment, allowing you to build and test your applications and CI/CD flows locally.



Wercker CLI : Installation

In order to make use of all the features within the CLI, you will need a working Docker environment. You can install Docker and Docker-machine using Docker Toolbox.

1

After installing Docker and Docker-machine, you'll need to create a new virtual machine that will run Docker:

```
docker-machine create --driver virtualbox dev
```

Then, once you've created your VM, you will need to export some variables to your environment:

```
eval "$(docker-machine env dev)"
```

2

```
curl -L
https://s3.amazonaws.com/downloads.wercker.com/cli/stable/darwin_amd64/wercker -o /usr/local/bin/wercker
```

3

1. Docker installation required for Wercker CLI
2. Create a VM (Docker Machine)
3. Install Wercker CLI on the DM

Wercker CLI

```
COMMANDS:
   build, b     build a project
   dev          build a local project
   check-config check the project's yaml
   deploy, d    deploy a project
   detect, de   detect the type of project
   login, l     log into wercker
   logout, l    logout from wercker
   pull, p      pull a build result
   version, v   display version information
   help, h      Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --environment "ENVIRONMENT"   specify additional environment variables in a file
   --verbose                     print more information
   --no-colors                   wercker output will not use colors (does not apply to step output)
   --debug                       print additional debug information
   --journal                     Send logs to systemd-journald. Suppresses stdout logging.
   --auth-token                  authentication token to use
   --help, -h                    show help
   --version, -v                 print the version
```


Detect

The detect command introspects your projects and generates wercker.yml files for your projects.

Currently go, python, nodejs, and ruby are supported in the detect command.

Build and Dev

The build and dev commands execute these pipelines locally. They are explained in detail here: build and dev.

Note: the deploy command is currently not supported.

Pulling builds

Using the pull command, you can download a container from the wercker platform after which you can use Docker commands to debug this container locally. Note that you have to add the internal/store to your wercker.yml. You can read more about pulling builds here.

Logging in

You can log in to wercker with your username and password as follows:

wercker login. This will save a token in your $HOME/.wercker folder, so you don't have to log in the next time.

Note that if you've signed up with GitHub, you will need a password for the CLI, which you can create on your profile page on Wercker.

Update

You can check if you're running the latest version of the CLI by running:

> wercker version : **1.0.54** Git commit: dabc15876b877209047fa926774f97f001afbf43 **No** new version available. When your Wercker version is not up to date, the CLI will ask you to download a newer version of the CLI.

Note: when upgrading to a new version, the binary will download into the current working directory. Be sure to replace the current binary.

Wercker CLI : Exploring Steps

The wercker dev command executes the dev pipeline designed for live development. Whereas with the Wercker build command a copy of your code is built inside a container in order to mitigate side effects, the dev command directly mounts your local directory inside the container.

```
box: nodesource/trusty
dev:
  steps:
    - npm-install
    - internal/watch:
        code: node app.js
        reload: true
```

ORACLE

During local development, you often times want to reload your environment on code changes. The internal/watch step is a long-running step that does exactly that and goes hand in hand with the wercker dev command. In order to use wercker dev and internal/watch, you create a dev section in your wercker.yml file:

box: nodesource/trusty dev: steps: **-** npm-install **-** internal/watch: code: node app.js reload: true.

In this example, we set up a dev pipeline in which one initial step. We run npm-install to prepare the container's environment and install the necessary dependencies.

Next, we use the internal/watch step to launch the application and through reload: true we reload the environment on file changes.

Wercker CLI: Build

Using the Wercker CLI, you can build projects locally on your machine.
`wercker build`

Mounting your working directory
You can mount your local project folder directly to the container's pipeline path by running:
`wercker build --direct-mount`

Committing containers locally
By default, running wercker build will not save the container. In order to save the container, you must commit it first:
`wercker build --commit <image_name>`

internal/docker-push
The container will only be committed to the Docker host, after which you can run it. Alternatively, if using the internal/docker-push together with the docker-local flag, the build container will also be committed to the local host.

Using the Wercker CLI, you can build projects locally on your machine.

Wercker CLI : Inspect

Often you want to have a closer look at what happened within your pipelines.
Builds that you have pulled or built locally using the CLI can be inspected.

This is useful for debugging purposes and having a closer look at what happened during your build.

As wercker currently supports Docker containers, introspection is done via the Docker command.

Assuming you have built a container locally or have pulled a build artifact as container, you can run the following command to inspect your container:

docker run -it –name c1 build-<BUILD-ID> /bin/bash.

This gives you a prompt inside the container. Now you can jump into the pipelinedirectory and have a look at the contents of your build pipeline.

After you've run the container, you can retrieve its container-id by running docker ps -a. You can then stop the container by using docker stop <container id>.

Wercker CLI : Environment

When running wercker build locally, you can pass environment variables defined on the host machine to the container.
All environment variables that start with X_ will be made available in the container, with the X_ prefix removed from the key in the container.

It is also possible to create an ENVIRONMENT file that contains multiple environment variables. Wercker will then automatically synchronize the variables defined in this file with your containers. If you want to use multiple environment files, you can specify the file to be used by adding the --environment flag while running wercker build command:

wercker --environment wercker.env build**.**

The keys in the environment file also need to start with X_.

To create an environment variable whose value comes from a file and contains multiple lines (an SSH key, for example), a command like this one may be used:

export X_BITBUCKET_PRIVATE=**"$(awk 'BEGIN {} {file=file$0"\n"} END {print file}' ~/.ssh/id_rsa | sed -e 's/\n$//')"**

Steps

Steps are self-contained bash scripts or compiled binaries for accomplishing specific automation tasks, defined in the wercker.yml file of your application.
Steps can be written manually or borrowed from the community via the Steps Registry.
An example of a step with parameters:

```
build:
    steps:
      - npm-install@1.0.5:
          package: jshint
          strict-ssl: false
      - npm-test
```


This will pass two parameters to the npm install step: package and strict-ssl.

Apart from predefined steps, there are also custom steps, also known as inline steps. Custom steps allow you to run bash scripts directly within your pipelines:

Exploring Steps: After-Steps

Wercker also has the notion of after-steps, which are ideally suited for notifications.

```
deploy:
    steps:
        - script:
            name: fabric deploy
            code: |
              fab deploy
    after-steps:
        - hipchat-notify:
            token: $HIPCHAT_TOKEN
            room_id: id
            from-name: name
```

Pipelines executions can contain after-steps. After-steps are steps that are executed after a pipeline run has either failed or passed. A good use-case for after-steps is notifications to an IRC channel or HipChat Room, which close an issue on a project management system or send out a newsletter after a successful deploy.

Here's an example that leverages the hipchat-notify step to send a notification. Note the use of the after-steps element to signify that the message has to be sent after the steps of the deploy. One major difference between steps and after-steps is that failing after-steps do not change the result of the pipeline run.

Exploring Steps: Internal Steps

Internal steps interact with the Docker API that is external from the container. From a technical perspective, it is not possible to interact with the Docker daemon from within a container. This is why we created these internal steps.

```
- internal/docker-push:
    # specify the image to be pushed - this is the one we
created earlier
    image-name: my-new-image
    username: $USERNAME # Registry username
    password: $PASSWORD # Registry password
    registry: https://hub.docker.com
    repository: $USERNAME/docker-build-golang
```


Internal steps are developed by Wercker and are baked into the Wercker CLI. Internal steps interact with the Docker API that is external from the container. From a technical perspective, it is not possible to interact with the Docker daemon from within a container. This is why we created these internal steps.

Exploring Steps:

The script step allows you to execute a single or more shell commands. It has one required property: code, and (as always) you can use the name property to give the step a clear name.

```
build:
    steps:
        - script:
            name: identify distribution
            code: cat /etc/lsb-release
        - script:
            name: starting xvfb
            code: |
              # Start xvfb which gives the context an virtual display
              # which is required for tests that require an GUI
              export DISPLAY=:99.0
              start-stop-daemon --start --quiet --pidfile /tmp/xvfb_99.pid --make-pidfile --background --
exec /usr/bin/Xvfb -- :99 -screen 0 1024x768x24 -ac +extension GLX +render -noreset
              # Give xvfb time to start. 3 seconds is the default for all xvfb-run commands.
              sleep 3
```

Exploring Steps: Create Steps

To be able to publish a step, you first need a Wercker application containing the following files:

- step manifest file called step.yml
- run.sh file that is the entry-point of your step
- wercker.yml file with an internal/publish-step step

If you cannot find a step that solves a challenge you are facing in the Steps Store, you can create your own step and publish it. Steps are similar to any other projects on Wercker, but instead of deploying to a cloud provider, you deploy to the Steps Store in order to make the step available.

Although a step can technically be written in any language, we recommended writing them in bash or in golang. The former is usually installed on most Docker containers, whereas the latter compiles to a single binary. Both programming languages do not require any dependencies or libraries to be present that might not be installed in various Docker containers people use, making them quite portable. If you do create a step in a different programming language, and require certain run-time components, make sure to document this in the step README.

Exploring Steps: Step Manifest File

The step manifest file defines the configuration of your step. Below is an example of a Wercker step manifest, which we'll go over:

```
# step.yml
name: slack-notifier
version: 1.4.0
summary: Posts Wercker build/deploy status to a Slack channel.
tags:
 - notification
 - webhook
 - slack
properties:
- name: url
 type: string
 required: true
- name: channel
 type: string
 required: false
- name: username
 type: string
 required: false
```


The summary field is the description for the step. We recommend using a small single line or single paragraph description. All other documentation should be in the README.

Only name, version, and summary are required. All the following properties are optional, though we encourage people to use them.

The tags field contains an array of strings with keywords of this step. We recommend a few tags (at most 5) to describe the step.

The keywords field contains an array of strings with keywords of this step. We recommend a few tags (at most 5) to describe the step.

The properties field contains metadata describing the parameters that are available for the step. This is a map; the key is the name of the step, and the value is a object with the following properties:

- name - the name of the property
- type - the type of the data of the parameter. Currently supported: string
- required - boolean indicating if the parameter is required or not (currently not enforced)
- default - value that gets used, if no parameter was provided through the wercker.yml

Exploring Steps: run.sh file

The run.sh file contains the entrypoint, your step logic. This should be bash code in the run.sh file itself. If you want to create a more complex application in a different language, then call this from within the run.sh file.

For each property you specified in your step.yml, Wercker sets a corresponding environment variable. For example, the value of the url property would be made available in the $WERCKER_SLACK_NOTIFIER_URL environment variable. The name of the environment variable will be WERCKER_ followed by the name of the step (from the step.yml) converted to uppercase and with hyphens changed to underscores, followed by another underscore and then the name of the property converted to uppercase.

Notice that any hyphens you use in your parameter names will be transformed to underscores.

Exploring Steps : Wercker.yml

In order to publish a step to the Steps Store, the step's wercker.yml file needs to contain the publish-step step, usually at the end. Below is an example of the wercker.yml file for the aforementioned slack-notifier step:

```
# wercker.yml
box: debian:stable-slim
build:
 steps:
 - shellcheck:
 files: run.sh

 - script:
 name: prepare output
 code: rm -rf $WERCKER_ROOT/.git

publish:
 steps:
 - install-packages:
 packages: ca-certificates

 - internal/publish-step:
 owner: wercker
```


The publish-step step has the following optional parameters:

- auth_token: contains the publishing user's personal token
- owner: can be used to specify the owner of the step (for example, an organization). NOTE: Be sure to change the owner to your user id or the organization you want to own the step.
- private: used to publish private steps that are only accessible by the owner or all members in an organization, if the private step is owned by an organization

Having a separate pipeline publish (or deploy) that contains the publish-step is not required.

Pipelines

The pipeline is the heart and soul of Wercker.
It's where you define the actions (steps) and environment for your tasks.
It's also often where you define your tests, builds, and deploys. They are an aggregate of steps and will pass or fail based on the steps within.

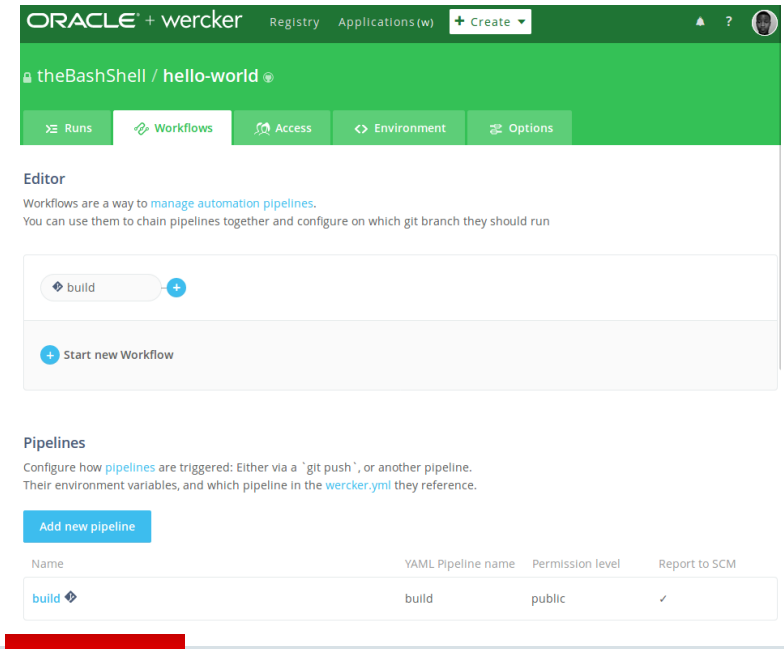
Each pipeline also comes with an environment. Some environments are set by default by the Wercker tool. When run on wercker.com, they are defined by the settings you've entered on wercker.com.

Steps are the actions performed within pipelines.

The configuration file and format for your pipelines is the wercker.yml file.

Every artifact from a build pipeline is both a Docker container as well as a tarball of the source code.

Pipeline



Note - When creating an application on Wercker Web, a build pipeline, which contains a Git hook and executes a pipeline called build, will be automatically created.

This section describes how pipelines can be defined and updated on the hosted Wercker platform.

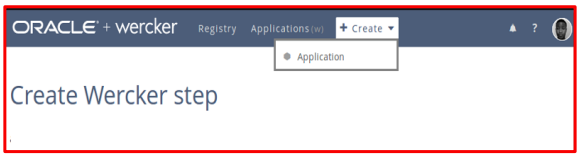
Creating pipelines

To create a new pipeline so that it can be used in Workflows, you must:

1. Give it a name, e.g.: “deploy-to-dockerhub”.
2. Decide how this pipeline gets executed. This can be a Git source or another pipeline.
3. Specify which pipeline you would like to execute. The pipeline name corresponds to the pipeline name as you defined it in your wercker.yml.

Publishing Steps: Wercker Store

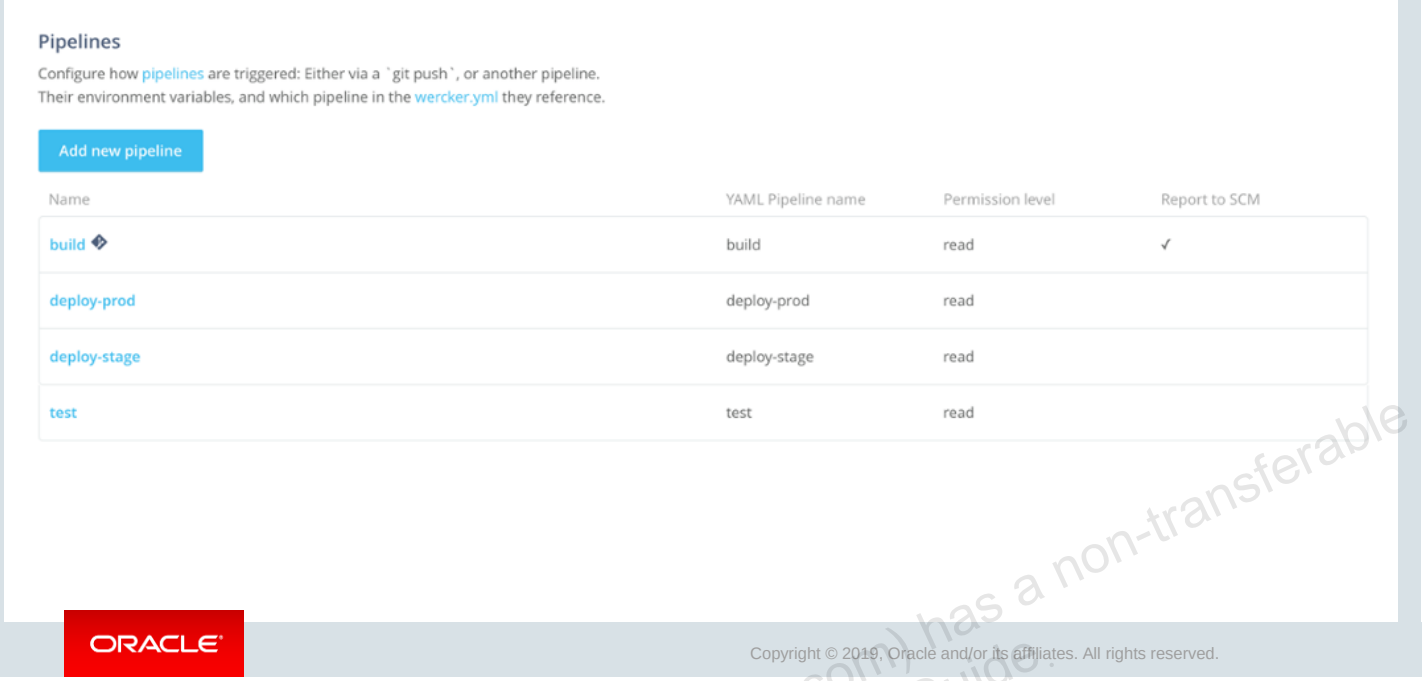
Publishing steps is done by deploying your step to the Steps Store. To do so, create a project on Wercker for your step by going to Create and selecting Application.




```
box: Debian
build:
 steps:
  # use the slack notifier step described above
  - owner/slack-notifier:
  URL: https://someplace.slack.com
  channel: my-channel
```


The build process will run in the same way as any other application on Wercker. After the application has built, and the build pipeline has run, you can then publish the step by selecting publish (or whichever pipeline you decided to contain the publish-step) from the **Actions** drop-down list of any completed (green) build.

Pipelines: Updating



An overview of all your pipelines is available in your application's Workflows tab. Selecting a pipeline allows you to edit its settings.

Manually triggering a pipeline

You can configure a pipeline to run only when you manually trigger it. To do this:

- Select the pipeline that you want to run manually on the **Workflows** tab.
- Under the **Settings** section, check the **Require manual approval** check box and click **Update**.
- Now when you run this pipeline from the **Runs** tab, you can see an "approve run" button (like a play button) in front of your pipeline.

WERCKER Cache

A cache directory is shared between pipelines in your Workflows. The path to this directory is stored in the environment variable $WERCKER_CACHE_DIR. A step can leverage this cache to share assets between builds.

```
if [ -f "$WERCKER_CACHE_DIR/mystep/a-dependency.bin" ]; then
    debug "a-dependency.bin found in cache"
else
    debug "a-dependency.bin not found in cache, will download"
    curl -o "$WERCKER_CACHE_DIR/mystep/a-dependency.bin" "http://example.com/a-dependency.bin"
fi
```

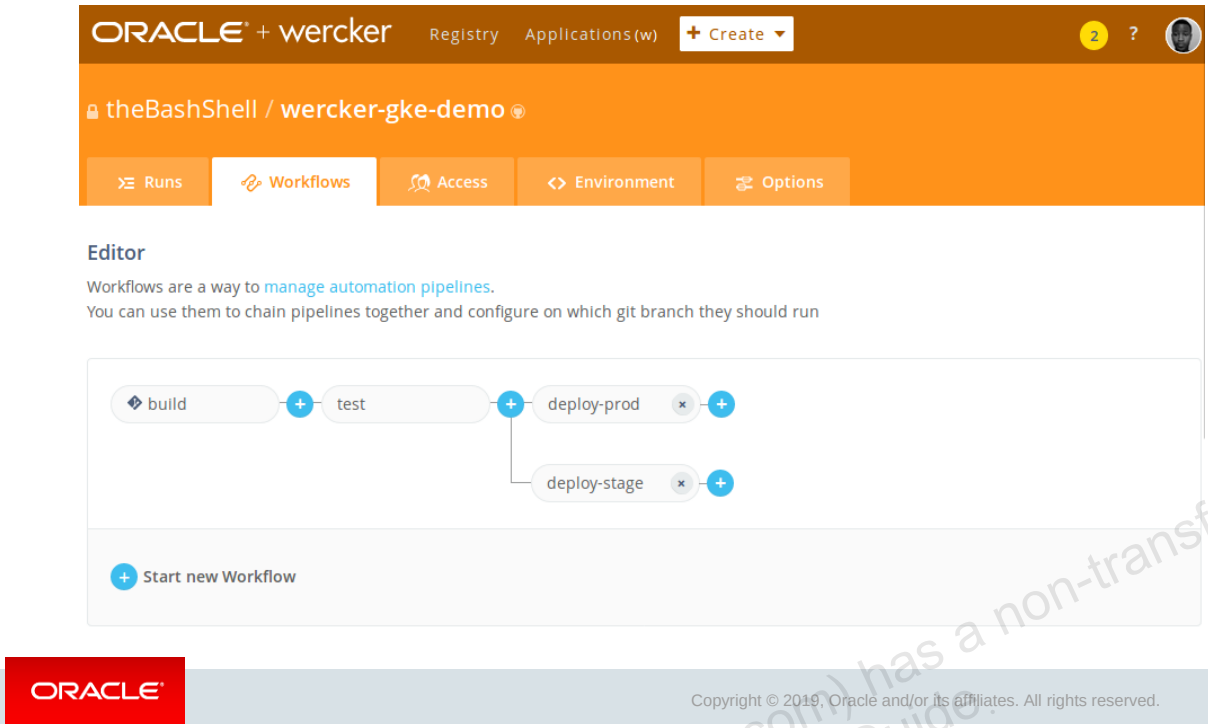

Out of the box, we've enabled caching by default for three dependency installers:

- npm-install
- bundle-install
- pip-install combined with virtualenv

These steps leverage the cache to shorten the installation time of dependencies. This works by storing the end result of the downloading and compiling of dependent packages.

Future pipeline executions can use this as a starting point, and only new dependencies that were not cached are downloaded. At the start of every pipeline run, the cache directory is filled with the cached content from the last successful build, if not older than 14 days (and is < 1GB).

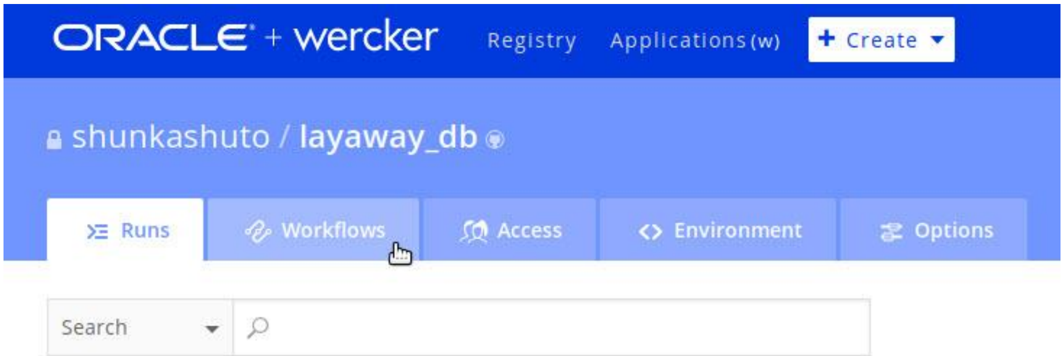
Workflows



To create a new workflow, go to the “Workflows” tab of your application Wercker. From here, you can specify the pipelines you have created your application’s wercker.yml file.

Once Wercker is aware of your application’s pipelines, you can join them together visually in the editor to form Workflows.

Workflows: Hooks



Hooks

Hooks define how pipelines get triggered. By default, pipelines will listen for other pipelines to finish. Alternatively, you can assign a Git hook to the pipeline to make it trigger on Git changes.

Git Hooks

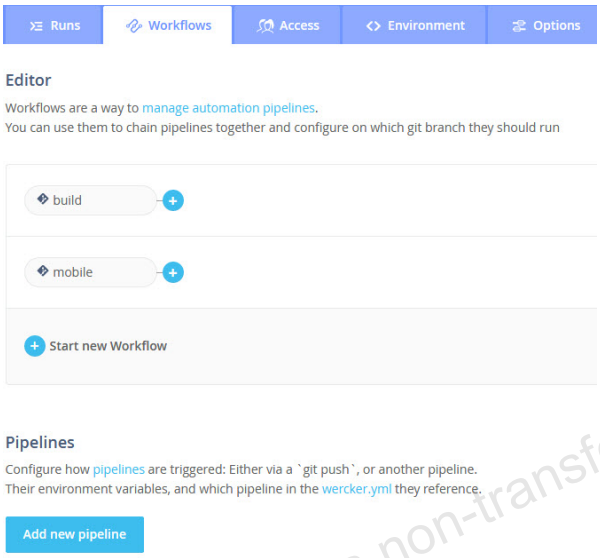
Git hooks work just like a build pipeline. When a change is detected for a Git source, Wercker will trigger the pipeline that is attached to it. Git hooks allow you to specify which branches should be ignored in the Hooks section of your Workflow.

Pipeline Hooks (default)

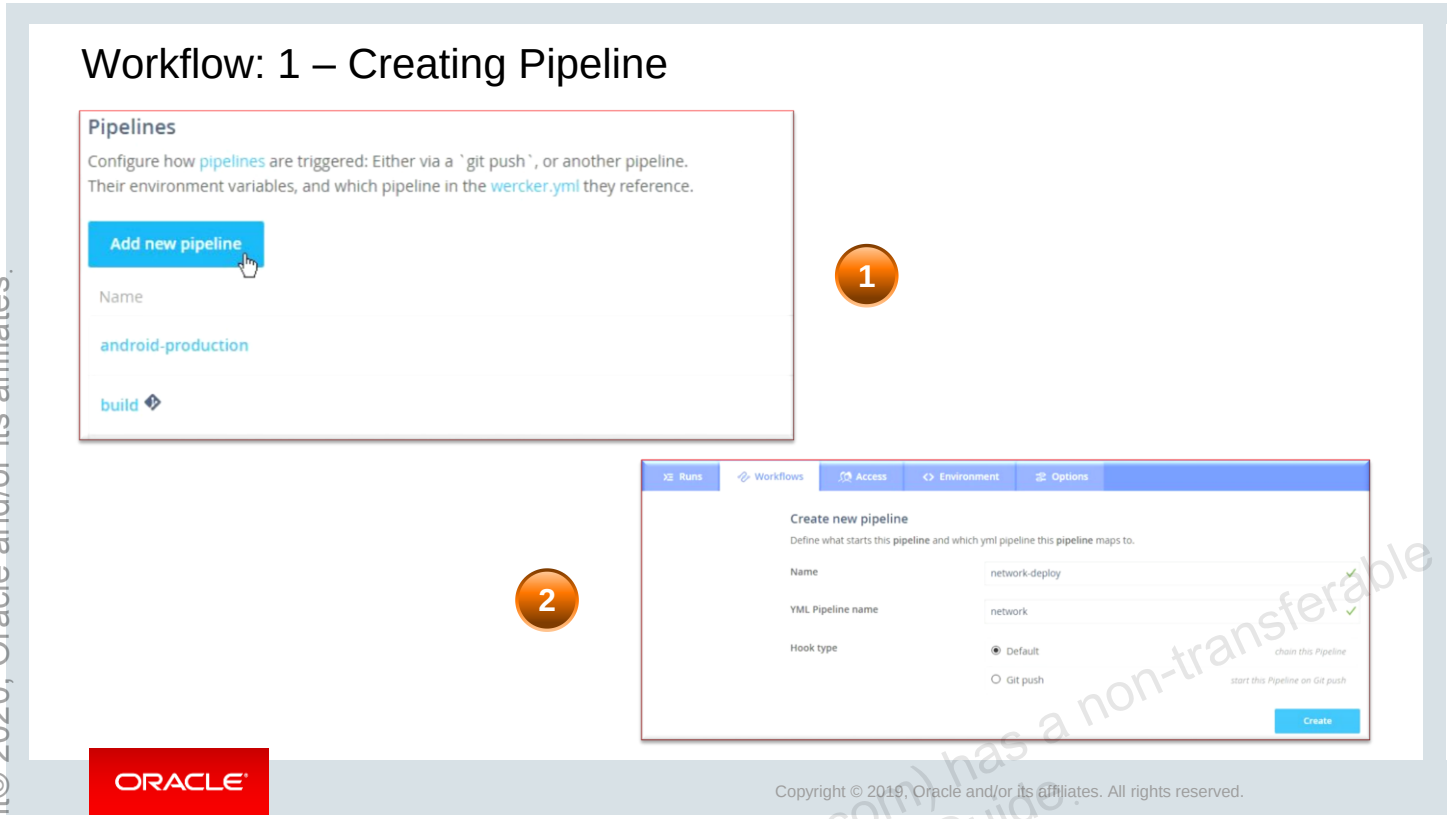
The default hook enables you to start a new pipeline based on the result of a previous pipeline execution. This allows you to chain pipelines together.

Workflow: Creation

The Workflow tab is divided into two sections.
At the top is the editor, where you create and edit workflows.
Below is the Pipelines section, where you create and edit workflow pipelines.



Workflow pipelines represent the pipelines in your wercker.yml document, but they include some added properties that are used within the workflow. It is for this reason that you must create and edit pipelines in the Workflows tab. Since a workflow is made up of pipelines, you need to create the pipelines that it will use before you create the workflow.



Step 1

Before you create workflow pipelines, you should create the corresponding pipelines in your project's wercker.yml document, since each workflow pipeline will reference a wercker.yml pipeline.

To create a workflow pipeline, click the Add new pipeline button in the Workflows tab. This will bring up the Create new pipeline screen.

Step 2

In the Create new pipeline screen, enter the following information:

Name

This is the name that will appear in the Pipelines list and in the workflow diagram. It does not need to be identical to the name of the wercker.yml pipeline that it references, but it should clearly indicate the pipeline's function.

YML Pipeline Name

This is the name of the pipeline in your project's wercker.yml document. It needs to be exactly the same as the name in wercker.yml. When the workflow pipeline is triggered, it will cause the wercker.yml pipeline to execute. Note that a wercker.yml pipeline can be referenced by more than one workflow pipeline.

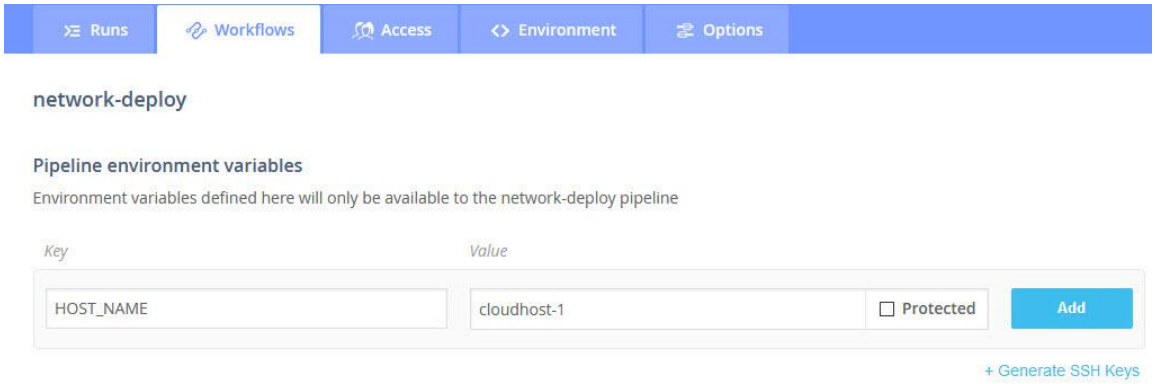
Hook Type

The hook is the action that triggers the workflow pipeline.

The default option sets the pipeline to be triggered by another pipeline in the workflow. You can place a pipeline with the default hook anywhere in the workflow.

The Git push hook sets the pipeline to be triggered by a push from Git. This means that the pipeline will be the first one in a workflow. When you create a pipeline with a Git push hook, it is automatically listed as the start of a new workflow in the Editor.

Pipelines: More

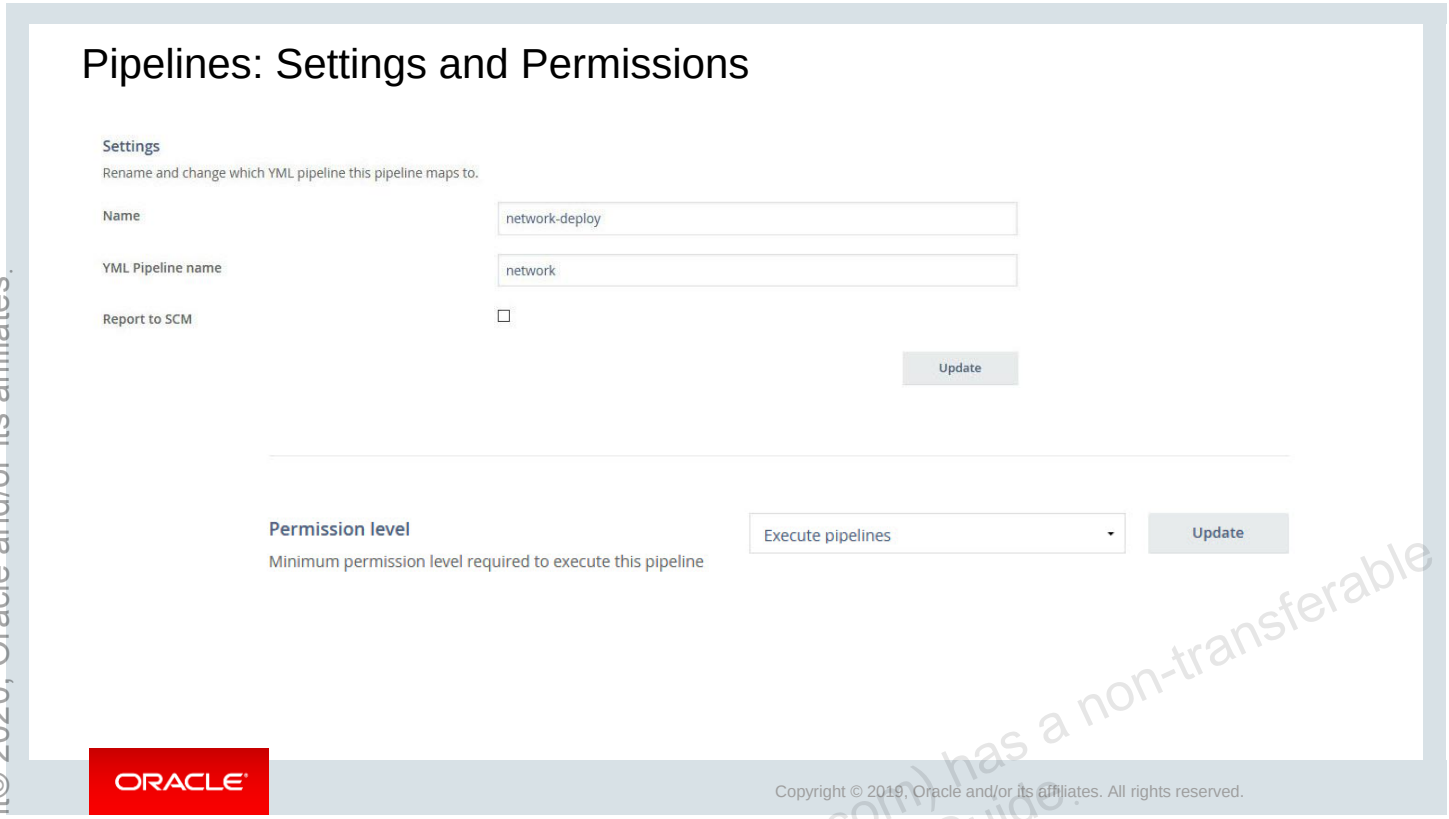


The pipeline details screen allows you to enter or edit detailed information regarding the pipeline.

Pipeline Environment Variables

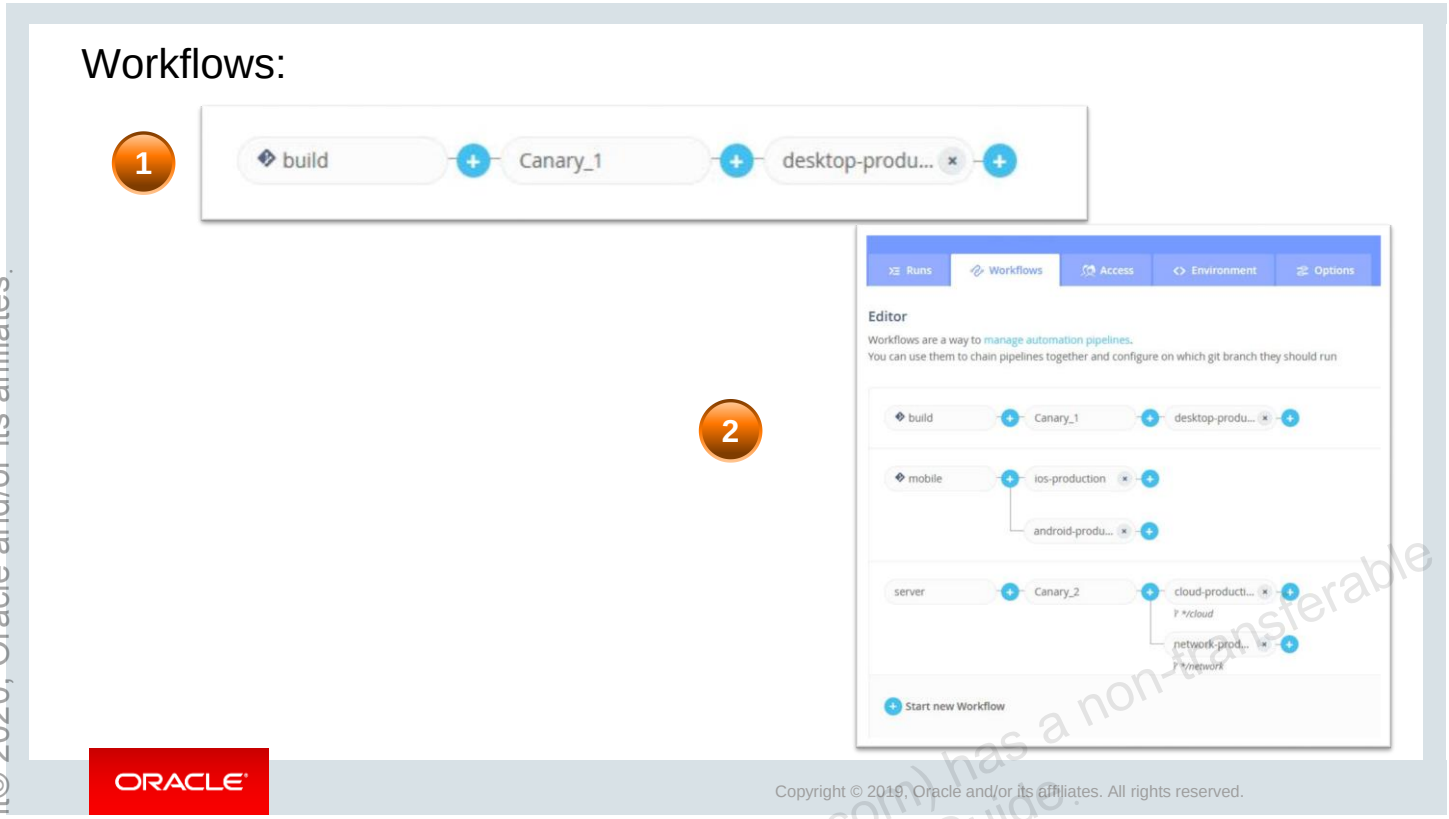
These are optional environment variables that will be available only within the scope of the workflow pipeline. They can be used to represent such things as device addresses, passwords, and SSH keys, as well as any other environment variable that is relevant to the pipeline. To create an environment variable, you must enter the name of the variable in the Key field and the value in the Value field. Each variable must have both a Key and a Value.

Pipelines: Settings and Permissions



The Settings section of the pipeline details page allows you to change the name and YML pipeline name for the pipeline. You can also click the Report to SCM check box if you want the Software Configuration Management role to be notified when the pipeline is triggered.

In the Permission level section, you can set the minimum permission level required to execute the pipeline. The permission levels are public, Execute pipelines, Execute & Manage pipelines, and admin. Public is the most permissive level; the other three levels correspond to the Permission levels that you can grant in the Wercker Access tab.

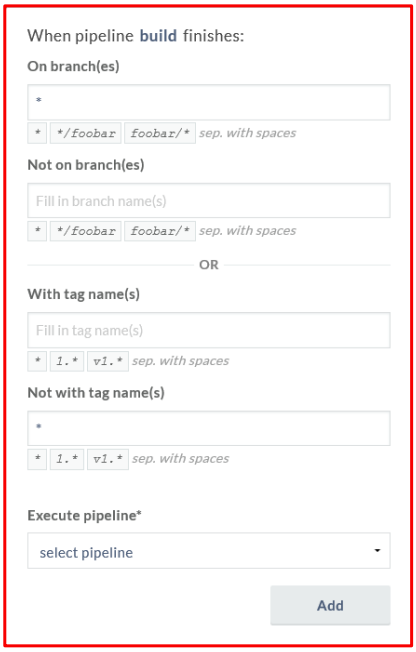


Step 1

By default, each project will have a Build workflow (based on the Build pipeline in wercker.yml), consisting only of the build pipeline itself. If you have created any other workflow pipelines that use the Git push hook, they will also be listed as separate workflows.

Step 2

A workflow typically consists of a set of pipelines chained together so that they execute in sequence, with each pipeline triggering the next. Workflows can fork, allowing parallel pipelines to execute, and they can include conditional execution, based on the branches involved in the build.



To add a pipeline to a workflow, click on any of the available nodes in the workflow (indicated by a white plus sign in a blue circle). The pipeline will execute at the node which you select; the node represents the completion of the previous pipeline. You can create a strictly sequential workflow, or one that forks, depending on the nodes that you select for connecting new pipelines.

Note: You cannot currently add more than ten sequential pipelines to a single workflow.

When you click a node, you will bring up the Add Pipeline dialog box:

Add Pipeline

The Add Pipeline dialog box enables you to select a pipeline that will execute after the current pipeline.

Branch filtering

When you add a pipeline, you can include and/or exclude the branches of your Git project to trigger the pipeline. Branch filtering is applied when you commit your code to the SCM repository.

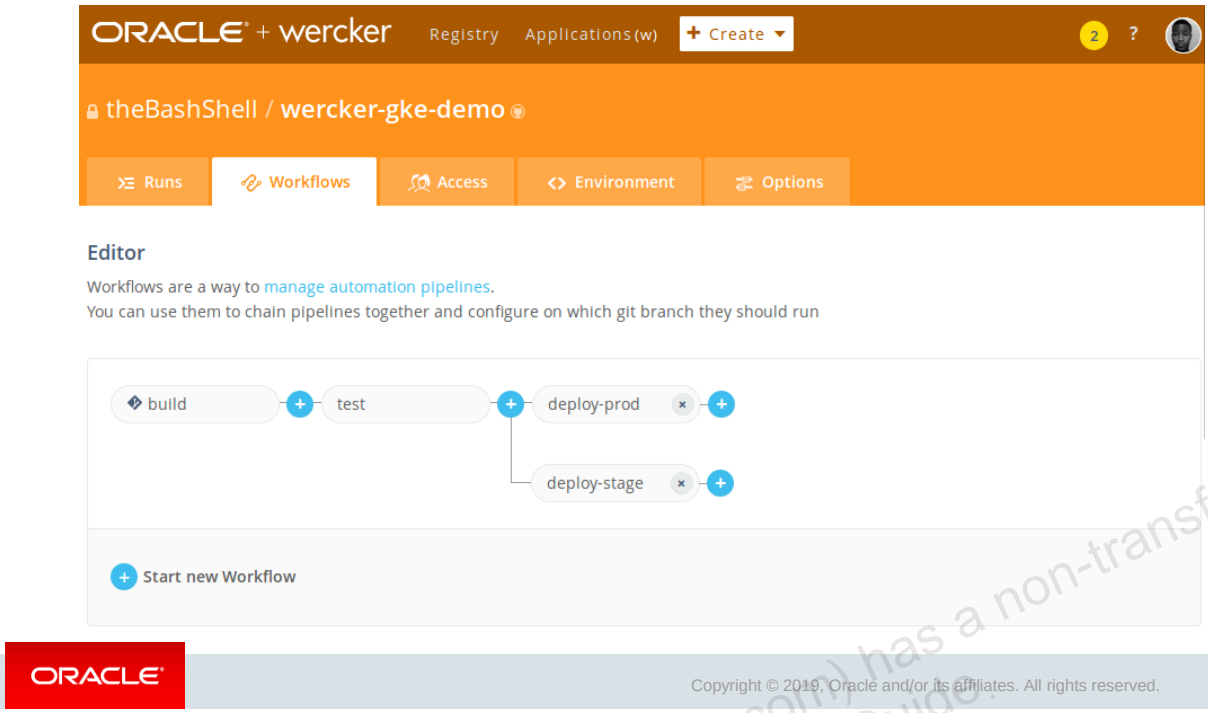
Tag filtering

You can also use tag filtering to run workflows. Unlike branch filtering, you must explicitly specify the tag filters that you want to apply to a pipeline. Tag filtering is applied when you push a tag to a repository.

Note: Tag filtering is available only for GitHub.

By default, the filter type (branch or tag) that you specify for the first pipeline applies to other pipelines in the workflow. However, you can override the filters for the subsequent pipelines, if required.

Workflow: Joined Pipelines



A major benefit introduced by Workflows is that you can chain as many pipelines as you want and have them run in series or parallel.

Workflows can consist of up to 10 joined pipelines.

YAML: Wercker.yml

A `wercker.yml` defines the steps required to execute automation tasks for your application, along with the pipelines that group them.

The wercker.yml defines the configuration of your automation pipelines with a collection of steps that you wish to execute.

In your wercker.yml, you can specify any pipeline you like. There is one special pipeline called dev, which will only be executed when running it with the CLI using the wercker dev command.

Examples of pipeline names: build-base-container, build, push-to-registry, deploy-kubernetes.

Upon creating a project in Wercker, we automatically create a build Workflow with a Git hook that executes a build pipeline.

A pipeline can have its own base box. For example, it might use a Docker container as a starting point. You can use different base boxes for each pipeline.

Each pipeline can also specify its own services. For example, a testing pipeline may require access to a database server, whereas a deploy pipeline probably would not.

YAML

```
- john:
    children:
      - james:
          children:
            - jim
            - jane
      - jenny:
          children: []
- jeremy:
    children: []
```

YAML is a whitespace-indented language, meaning that indentation is used to denote structure. Items of the same indentation are considered to be siblings, while more or less indentation denotes child and parent relationships, respectively.

YAML—short for “YAML Ain’t Markup Language”—is a human-readable, data-oriented markup language used to parse wercker.yml files. While there are other popular data formats that can be used (like XML and JSON), YAML is a perfect choice for configuration files because it is easy for humans to read and write, and nearly every popular programming language offers libraries for programmatically working with YAML.

Once you get the basic rules out of the way, YAML is a relatively straightforward language consisting of only a handful of standard components, the most common of which are scalars, sequences, and mappings.

Scalars Integers, floats, booleans, strings…these are scalars. While YAML has strict rules around indentation and case-sensitivity, it is relatively flexible when it comes to scalars. Each type has its validation rules, allowing you to define values in a way that is appropriate for the configuration file:

YAML: More

SEQUENCE

```
- Cat
- Dog
- Bird
- "Water Buffalo"
```

1

MAPPING

```
box: nodesource/trusty
# Build definition
build:
  # The steps that will be executed on build
  steps:
    # A step that executes `npm install` command
    - npm-install
    # A step that executes `npm test` command
    - npm-test

    # A custom script step, name value is used in the
UI
    # and the code value contains the command that
get executed
    - script:
```

2

- A sequence (or array, or list, as it's more often called) is exactly what it sounds like: a list of data. Each item is identified by a dash followed by a space and then the item.
- In its simplest form, a mapping—also known as a hash, dictionary, or associative array in other programming languages—is represented as a key: value pair (take note of the space following the colon, key: value is valid, key:value is not). This basic structure can be expanded using a combination of sequences and scalars, allowing for us to define complex, readable control structures.

Wercker Offline Tool

A helper tool that downloads the workflows, environment, and steps of a wercker application and generates the files necessary to run pipelines and workflows offline

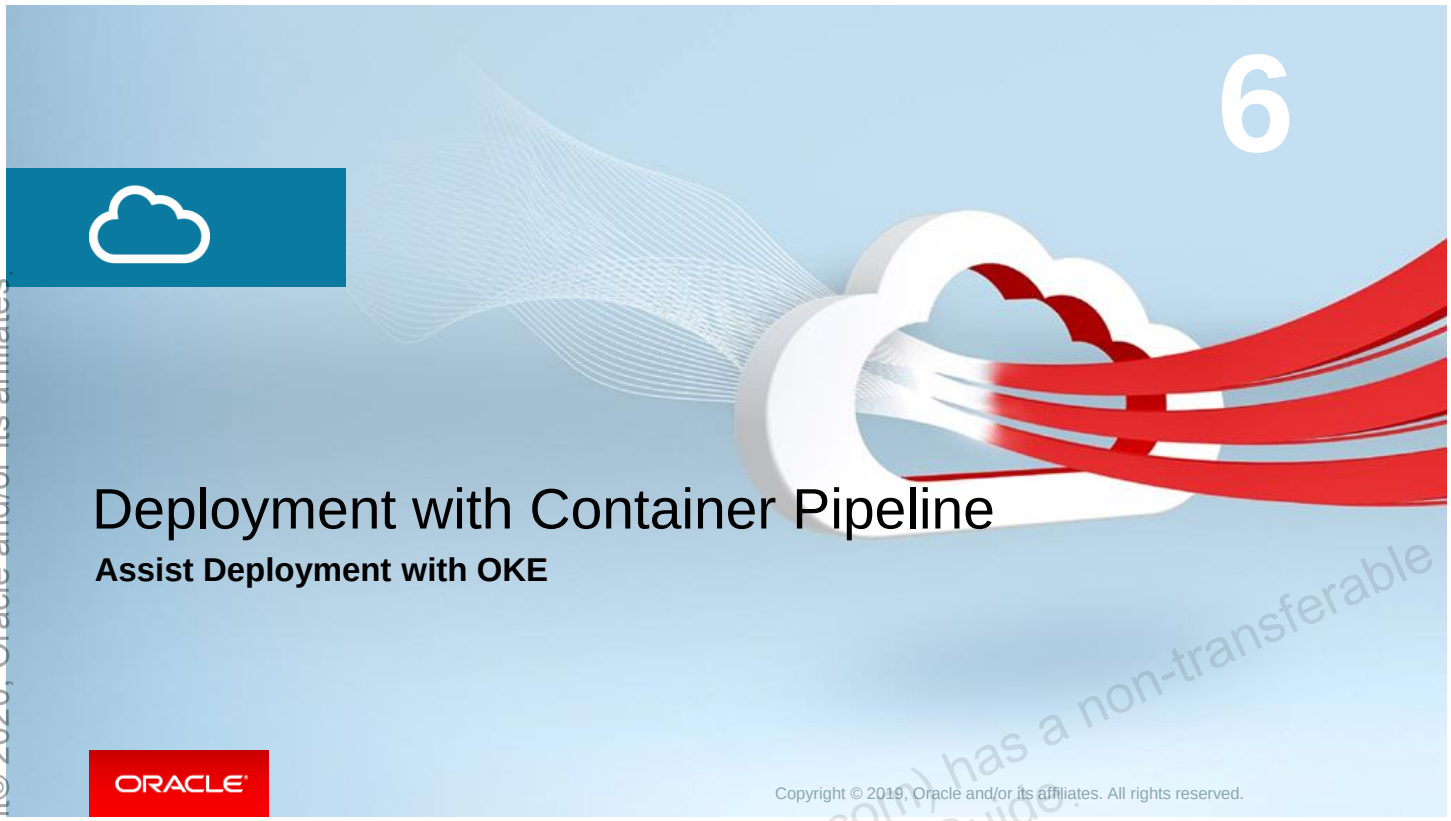
Practice 5: Overview

You will learn how to leverage Container Pipeline as service.

Summary

In this lesson, you should have learned how to:

- Explore API/end points of container pipelines
- Experience and integrate CLI commands
- Experience steps, pipeline, and workflows
- Integrate YAML in API



Deployment with Container Pipeline

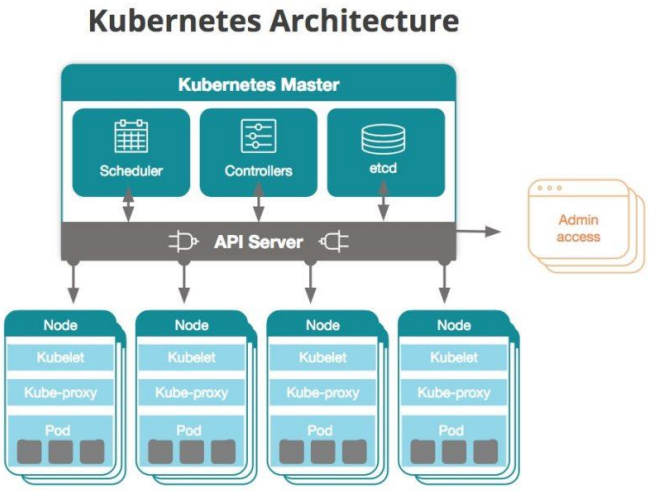
Assist Deployment with OKE

Objectives

After completing this lesson, you should be able to:

- Set up Deployment Pipeline in Container pipeline
- Trigger deployment in OKE

Kubernetes: Context



- The Kubernetes master is the access point (or the control plane) from which administrators and other users interact with the cluster to manage the scheduling and deployment of containers.
- A cluster will always have at least one master, but may have more depending on the cluster's replication pattern.

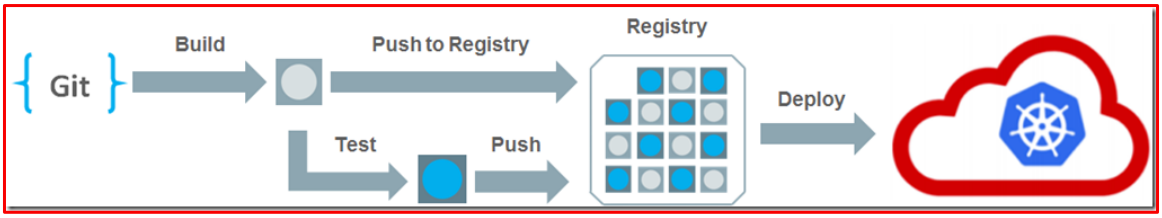
ORACLE

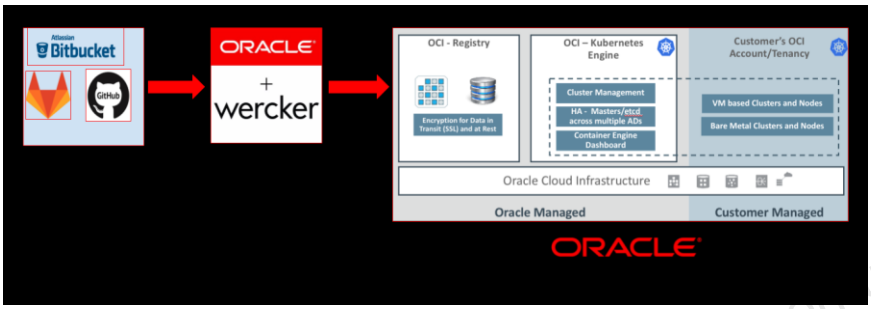
Kubernetes gives you the orchestration and management capabilities required to deploy containers at scale for these workloads. Kubernetes orchestration enables you to build application services that span multiple containers, schedule those containers across a cluster, scale those containers, and manage the health of those containers over time.

The central component of Kubernetes is the **cluster**. A cluster is made up of many virtual or physical machines that each serve a specialized function either as a master or as a node. Each node hosts groups of one or more containers (which contain your applications), and the master communicates with nodes about when to create or destroy containers. At the same time, it tells nodes how to reroute traffic based on new container alignments.

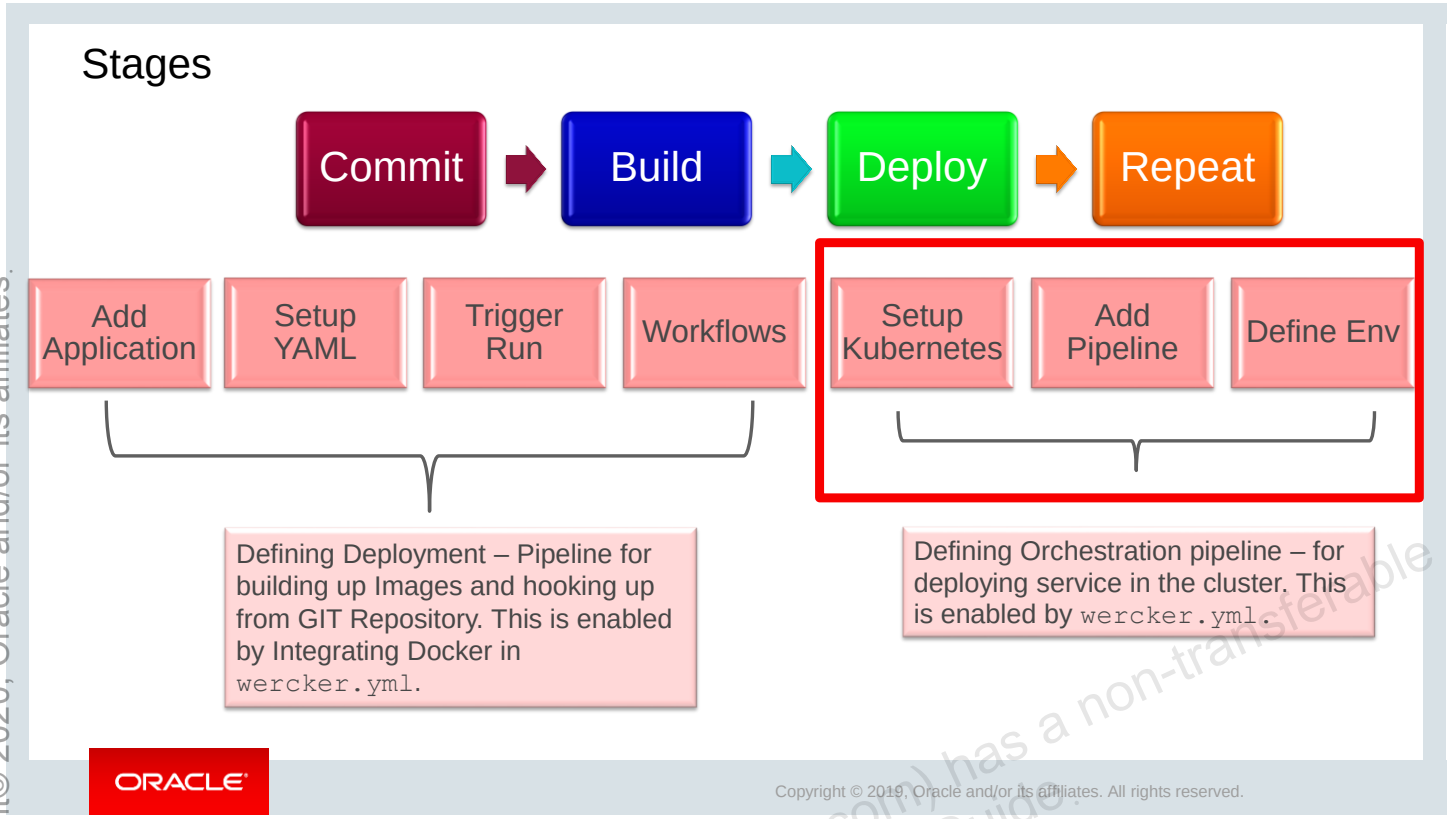
The diagram in the slide depicts a general outline of a Kubernetes **cluster.**

Oracle Container Pipeline: Deployment





You can think of steps, pipelines, and workflows like lego blocks to allow reusability at various levels for same or different applications.



To automate build, testing, and deployment configuration, we will be using Wercker.

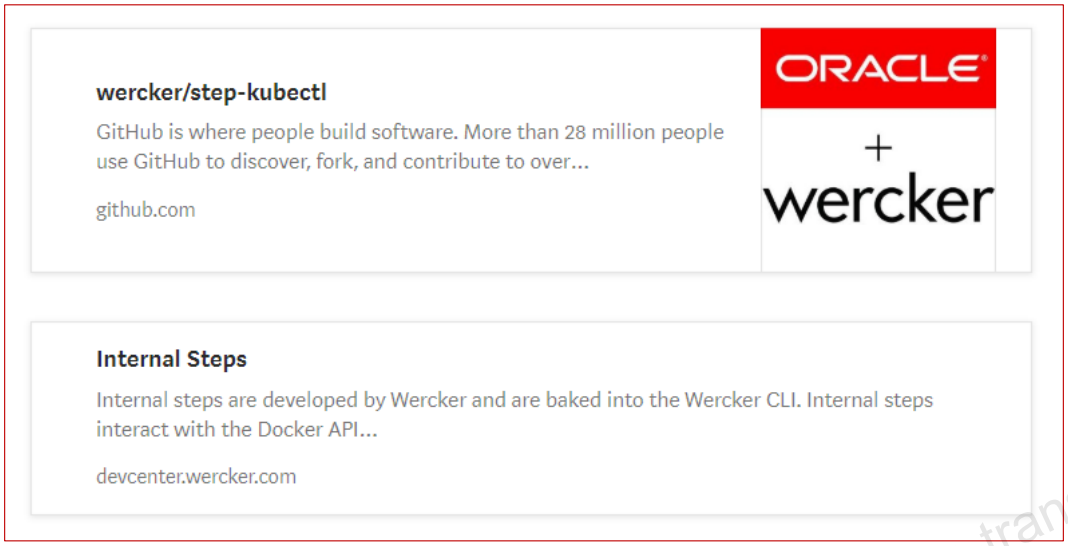
We need a platform that supports fault tolerance, scalability, and automation; therefore, we use Kubernetes.

Applications need to be dockerized and packaged as a pod for them to run on Kubernetes. Therefore, the application selected is built as a container.

We need a registry to store and act as a container repository and, therefore, we use Oracle managed container registry.

For ease of setting up and managing Kubernetes on a scalable and highly available cloud infrastructure, we use Oracle Kubernetes Engine.

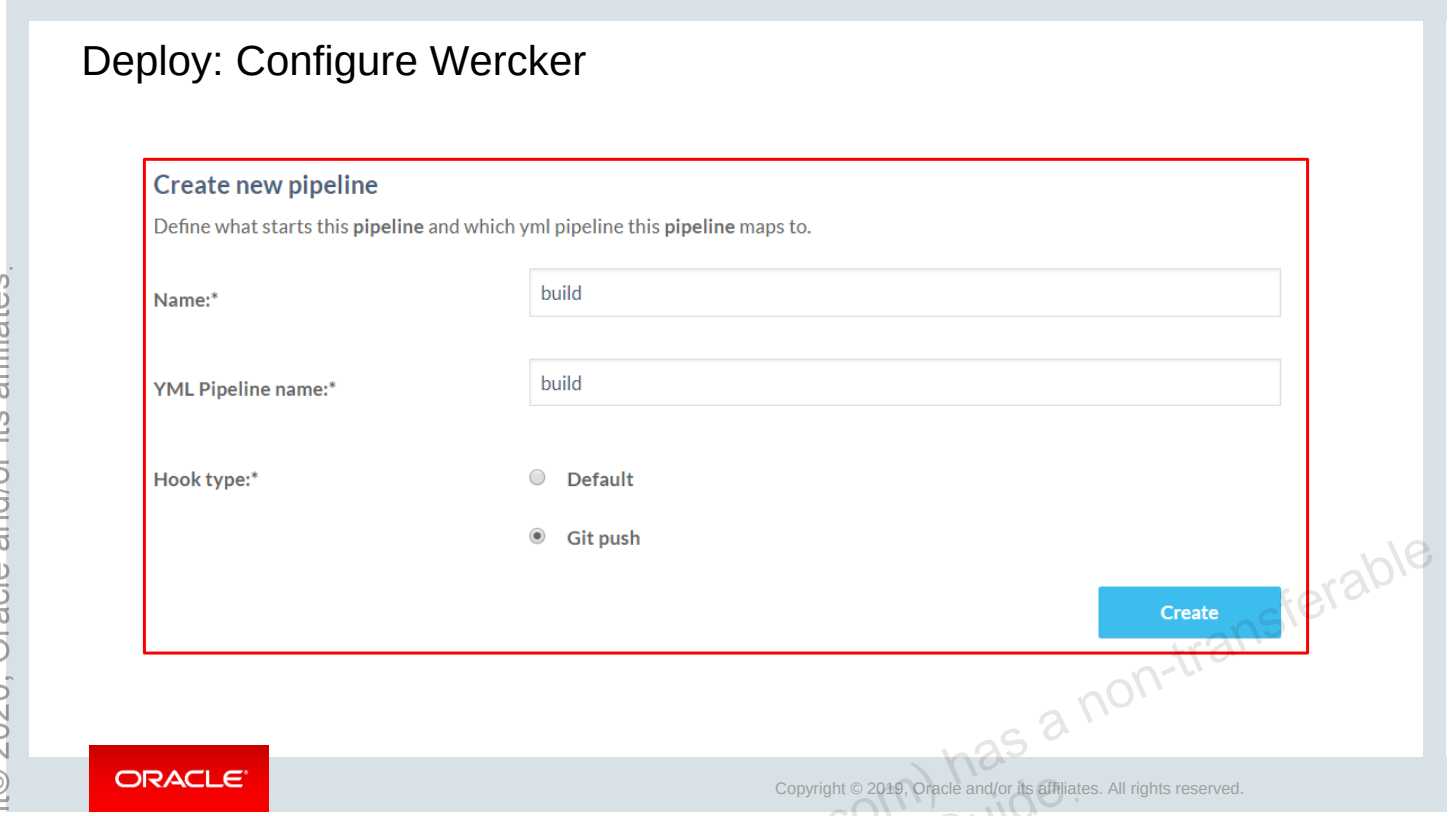
Deploy: Pipelines – Customizing Pipeline



The pipeline can be customized using the kind of steps you would like to execute. It is worthy to visit the step store to see predefined steps that can be reused in your `wercker.yml` file.

It is also noteworthy to understand that wercker supports docker and kubectl commands out of the box.

Deploy: Configure Wercker

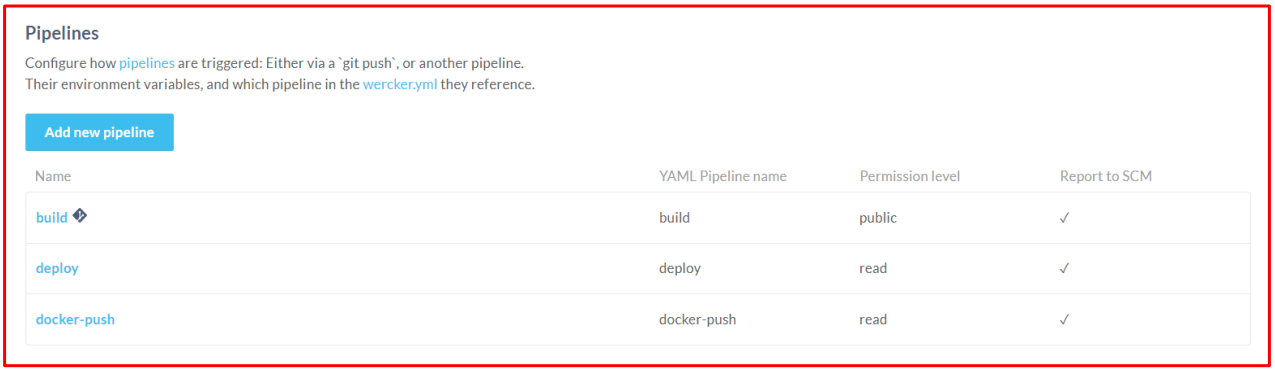


We want the build pipeline to execute on commit to the master repository. Therefore, select **Git Push** in the hook type. The output of the build pipeline is a container.

The wercker.yml pipeline name and the name defined here should match.

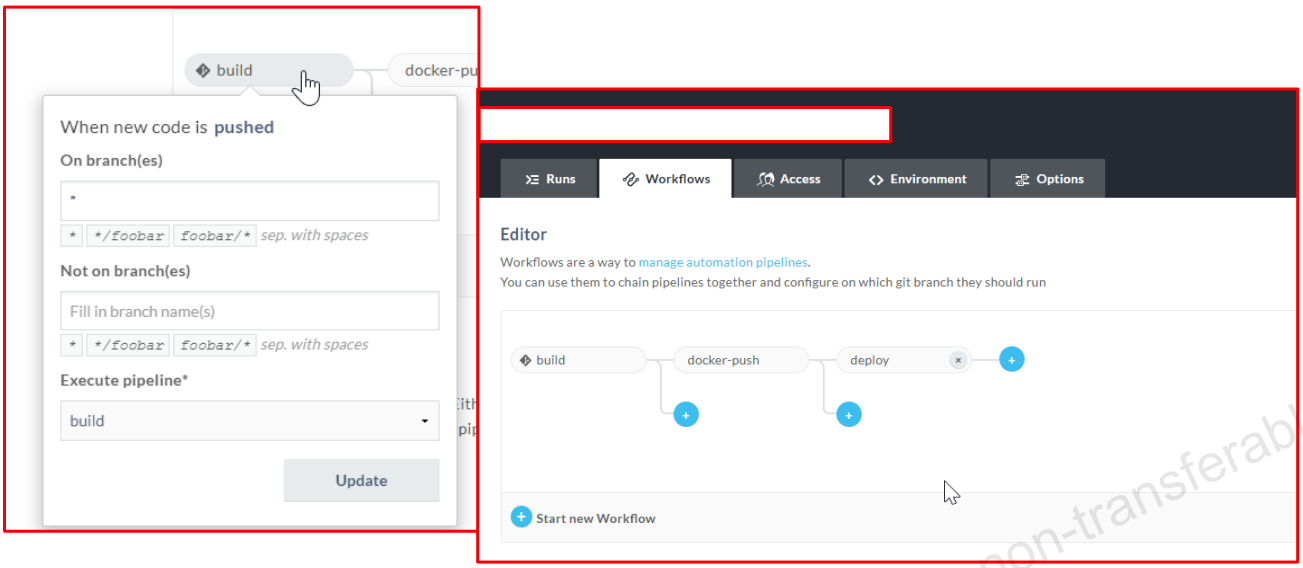
Push the build container to a private Docker repo and deploy the docket container onto a Kubernetes cluster, but these need to be triggered sequentially. Therefore, select Default in the rest of the pipelines for Hook type.

Deploy: Add Pipelines



To add pipelines to a workflow, select the **"+"** icon on the workflow, select your pipeline and add them sequentially/parallelly depending on your requirements.

Deploy: Add Pipelines



Here we are adding a workflow to every build.

Set Up Environments

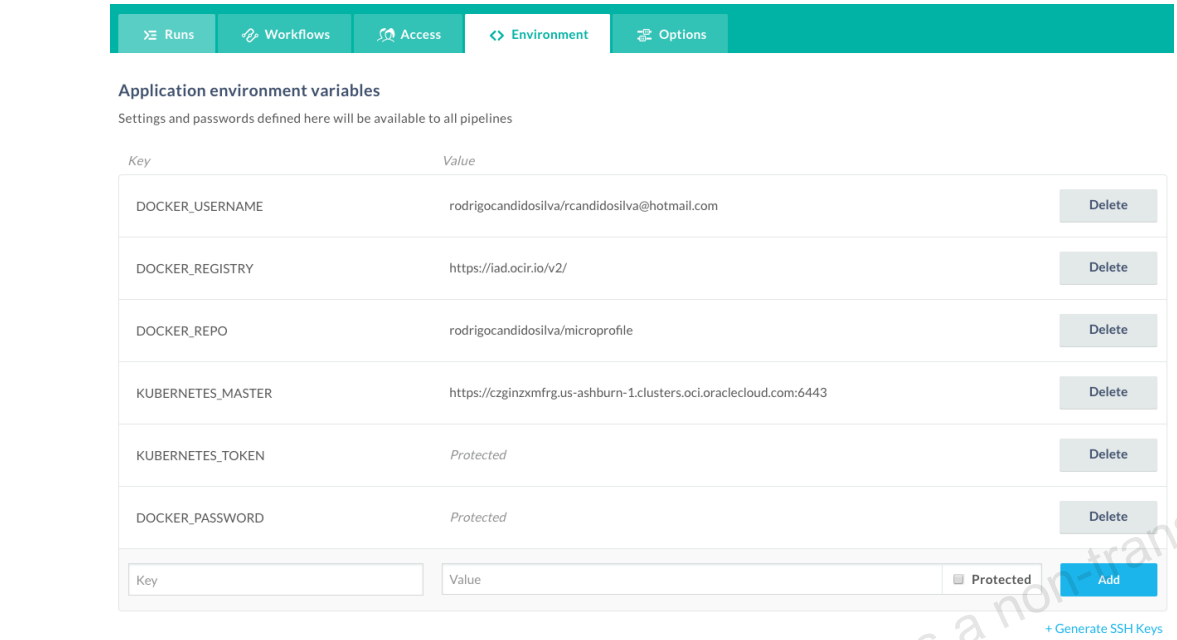
Name of Env Variable	Explanation
CONTAINER_REGISTRY_USERNAME	User name for Registry
API_USER_TOKEN	Authentication for User
TENANCY_NAME	Name of Tenancy where OKE is hosted
PRIVATE_REGISTRY_PATH	Endpoint of Private Registry
REPO_NAME	Name of Repository within registry
APP_NAME	Name of Application
SLACK_URL	Url of slack webhook for publication
SLACK_CHANNEL	Channel ID to post notifications
SLACK_USERNAME	Slack User name for auth
SLACK_TOKEN	Slack Token for Auth
KUBE_ENDPOINT	Cluster End Point - refr kubeconfig
KUBE_USER_TOKEN	Token provided in Kubeconfig

- The first three env variables are declared specific to Oracle Cloud Container Registry and Managed OKE.
- Modify the placeholders for the env variables in `wercker.yml` and the corresponding Kubernetes templates and the env variables in your wercker account.

Please define the following env variables in the workflow of your application in wercker.

The env variables provide placeholders for information such as Container Registry details for private image push and pull.

Set Up Environments



Finally, a few environmental variables were defined to complete the pipelines execution. You must set up the values for these variables through the application environment configuration in the project defined at Oracle Wercker.

DOCKER_USERNAME: User with access to the OCIR repository

DOCKER_REGISTRY: Register address of the OCIR repository

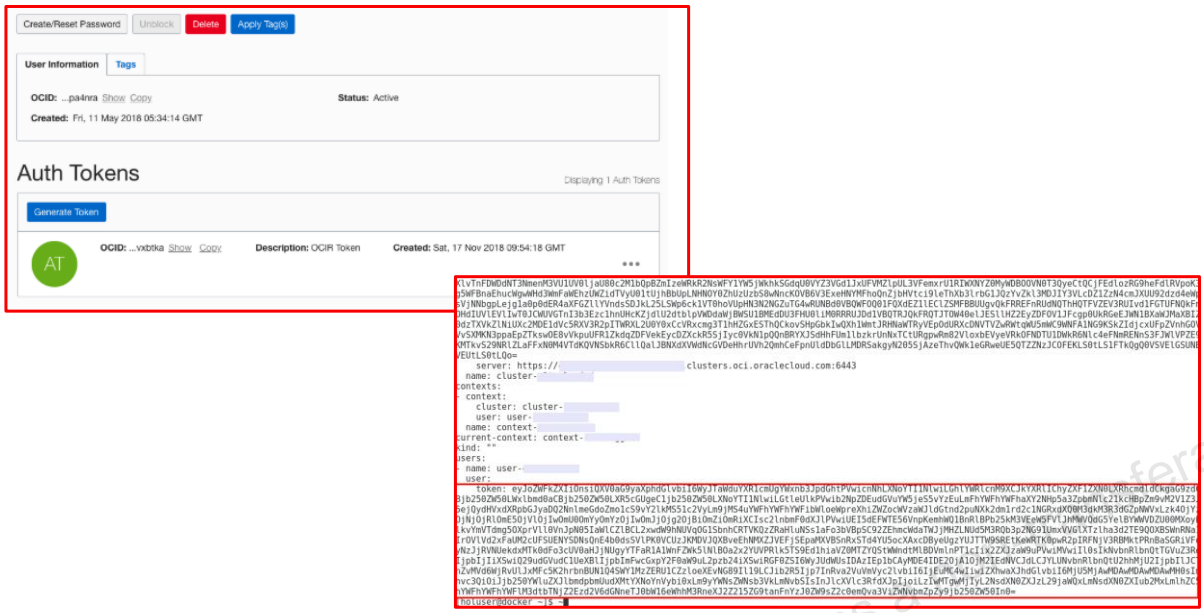
DOCKER_REPO: Name of the project image repository defined in OCIR

DOCKER_PASSWORD: Authentication token for OCIR repository access

KUBERNETES_MASTER: OKE cluster access address

KUBERNETES_TOKEN: OKE cluster access token

Set Up Environments: Kubernetes Token



To generate the OCIR repository access authentication token, you must request a new token for an existing user defined in the Oracle Cloud identity database.

You can check the Kubernetes cluster access token definition from the *kubeconfig* configuration that you performed earlier. Access the `~/.kube/config` file located on your local host and search for the user token reference.

Final: wercker.yml

```
box: golang
dev:
  steps:
    - internal/watch:
  after-steps:
    - install-packages:
        packages: ruby
deploy:
  box:
    id: golang
  steps:
    - kubectl:
        server: $KUBE_ENDPOINT
        token: $KUBE_USER_TOKEN
        cwd: $WERCKER_ROOT/kubernetes/deployment
        insecure-skip-tls-verify: true
        command: apply -f .
```

Note: In the steps, you can observe kubectl library with the endpoints of server and token.

KUBE_ENDPOINT	Cluster End Point - refr kubeconfig
KUBE_USER_TOKEN	Token provided in Kubeconfig


```
box: golang
dev:
  steps:
    - internal/watch:
        code: |
          go get github.com/Pallinder/go-randomdata
          go build -ldflags "-X cityHandler.minversion=`date -u +.%Y%m%d.%H%M%S`" main.go
          ./source
        reload: true
  after-steps:
    - install-packages:
        packages: ruby
    - pasonatquila/pretty-slack-notify@0.3.6:
        webhook_url: $SLACK_URL
        channel: $SLACK_CHANNEL
```

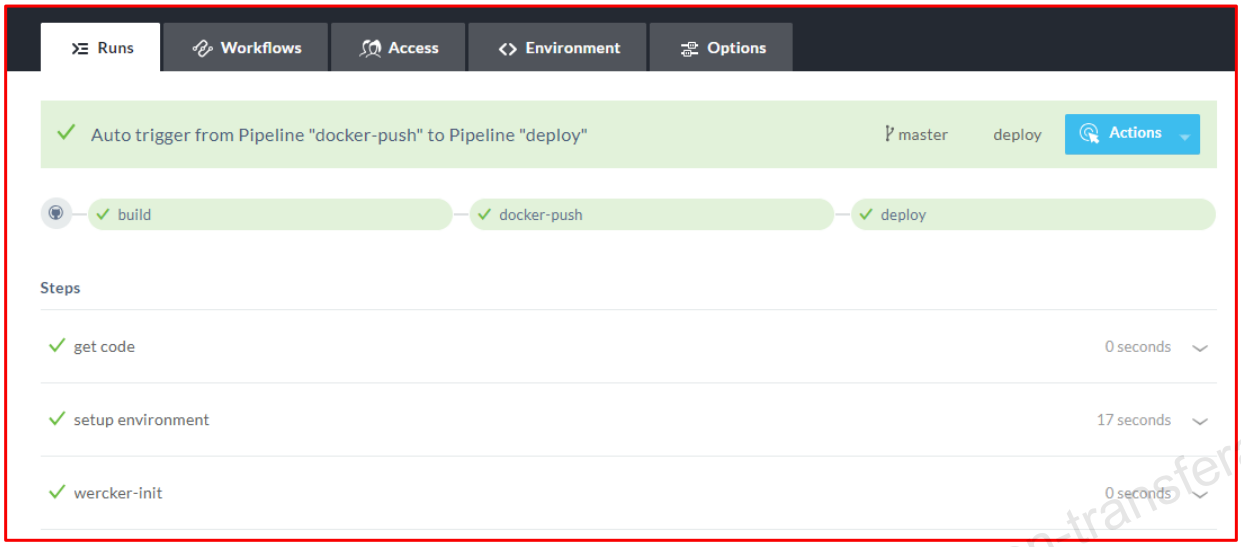
```
# Build definition
build:
  # The steps that will be executed on build
  steps:
    # golint step!
    - wercker/golint

    - script:
        name: debug
        code: |
          ls -l
    # Build the project
    - script:
        name: go build
        code: |
          go get github.com/Pallinder/go-randomdata
          go build -ldflags "-X cityHandler.minversion=`date -u +.%Y%m%d.%H%M%S`" main.go
          go build ./...
    # Test the project
    - script:
        name: go test
        code: |
          go test ./...
    - script:
        name: copy artifacts to output dir
        code: |
          # artifacts
          ls -l
          cp ${WERCKER_ROOT}/source "$WERCKER_OUTPUT_DIR"
          cp -a ${WERCKER_ROOT}/kubernetes "$WERCKER_OUTPUT_DIR"
  after-steps:
    - install-packages:
        packages: ruby
    - pasonatquila/pretty-slack-notify@0.3.6:
        webhook_url: $SLACK_URL
        channel: $SLACK_CHANNEL
```

```
docker-push:

  box:
    id: golang

  steps:
    - script:
        name: copy binaries to /opt/app
        code: |
          mkdir -p /opt
          cp ${WERCKER_ROOT}/source /opt/source
    # Pushing to a quay.io repository
    # For other docker repository refer to this doc:
http://devcenter.wercker.com/docs/containers/pushing-containers
    - internal/docker-push:
        username: $CONTAINER_REGISTRY_USERNAME
        password: $API_USER_TOKEN
        repository: $TENANCY_NAME/$REPO_NAME/$APP_NAME
        registry: $PRIVATE_REGISTRY_PATH
        tag: ${WERCKER_GIT_BRANCH}-${WERCKER_GIT_COMMIT},latest
        entrypoint: /opt/source
        port: 5000/tcp
  after-steps:
    - install-packages:
        packages: ruby
    - pasonatquila/pretty-slack-notify@0.3.6:
        webhook_url: $SLACK_URL
        channel: $SLACK_CHANNEL

deploy:

  box:
    id: golang
```

```
steps:

    - script:
        name: debug
        code: |
          echo $WERCKER_OUTPUT_DIR
          ls -l $WERCKER_OUTPUT_DIR
          echo $WERCKER_ROOT
          ls -l $WERCKER_ROOT
    - bash-template:
        cwd: $WERCKER_ROOT/kubernetes/deployment

    - script:
        name: Remove template files
        cwd: $WERCKER_ROOT/kubernetes/deployment
        code: |
          rm *.template.yaml
    - script:
        name: debug
        cwd: $WERCKER_ROOT/kubernetes/deployment
        code: |
          ls -l
    - script:
        name: echo tags
        code: |
          cat $WERCKER_ROOT/kubernetes/deployment/deployment.yaml
          echo "Docker image and tag:"
          echo "${WERCKER_GIT_BRANCH}-${WERCKER_GIT_COMMIT}"

    - kubectl:
        server: $KUBE_ENDPOINT
        token: $KUBE_USER_TOKEN
        cwd: $WERCKER_ROOT/kubernetes/deployment
        insecure-skip-tls-verify: true
        command: apply -f .
```

```
after-steps:
  - install-packages:
      packages: ruby
  - pasonatquila/pretty-slack-notify@0.3.6:
      webhook_url: $SLACK_URL
      channel: $SLACK_CHANNEL
```

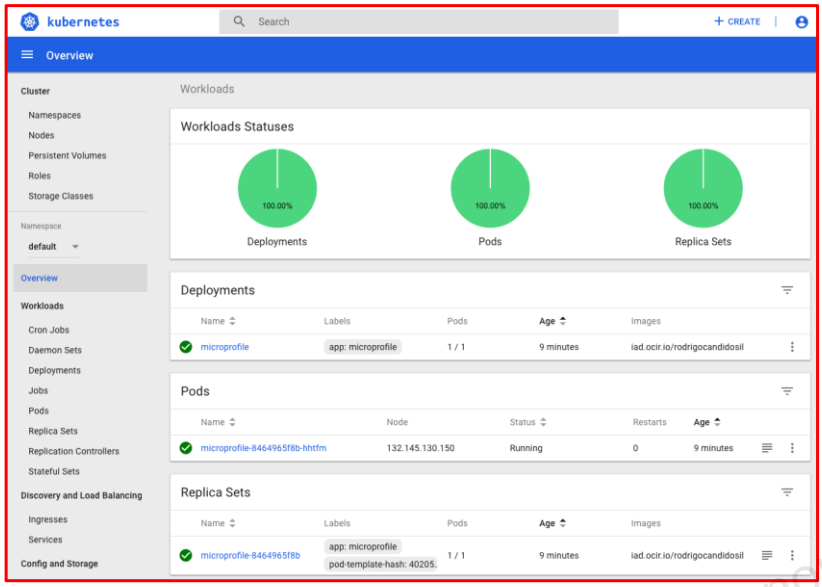
Runs Tab



The Runs tab provides an interface to check the terminal output and verify the status of each step. You can add a stepwise or a pipelinewise slack notifier and also copy app artifacts, such as logs and screenshots, and define it in the `wercker.yml` file.

To finish your configuration, you need to define the pipelines via the workflow editor in Oracle Wercker. For each pipeline defined in the `wercker.yml` file previously, you must add a new pipeline record.

Test



After the successful completion of the pipelines, you will be able to see the results by checking the new Docker image(s) persisted in the OCIR repository.

Practice 6: Overview

- You will learn how to deploy a Container in Oracle Kubernetes Engine (OKE)
- You will learn how to leverage Wercker and YAML to implement automation

Summary

In this lesson, you should have learned how to:

- Set up Deployment Pipeline in Container pipeline
- Trigger deployment in OKE

Integrations

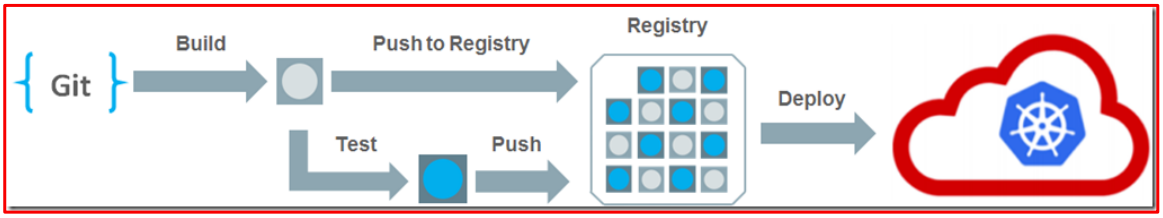
GIT and Notifications Support

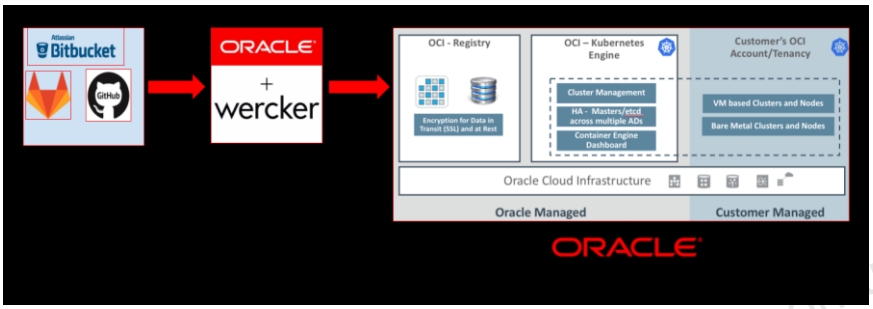
Objectives

After completing this lesson, you should be able to:

- Describe the use case of GIT
- Implement support for notifications

Oracle Container Pipeline: Deployment





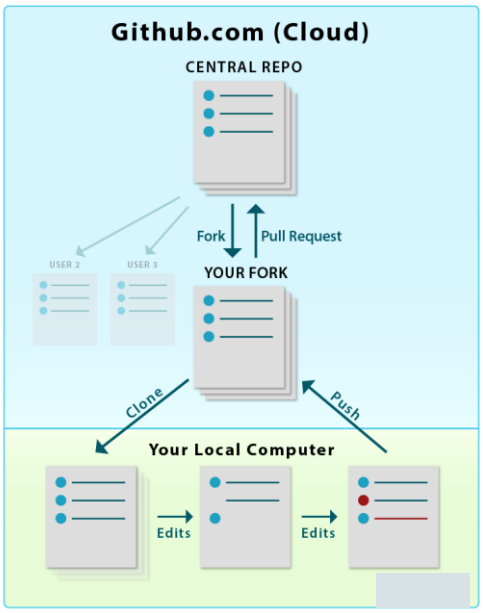
Context: GIT

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, nonlinear workflows

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.

Use Cases for GIT – How Wercker Pulls Request



Pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.

Wercker builds against the HEAD of the pull request branch and integrates with the build status API of GitHub.

Changes in the target branch are, therefore, not applicable to the builds. If, however, the pull request initiator adds a new commit to the PR (or rebases on the target branch), then this commit will be built again on Wercker.

Finally, any merged pull request will result in a new commit and thus in a new build.

GIT: Submodules

```
- script:
      name: install git
      code: |
         apt-get update
         apt-get install git -y
   - add-ssh-key:
      keyname: KEY_NAME
      host: github.com
   - add-to-known_hosts:
      hostname: github.com
      fingerprint:
16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48
   - script:
      name: initialize git submodules
      code: |
         git submodule update --init --recursive
```


You will have to fetch Git submodules manually. The slide shows a snippet of a wercker.yml step that does exactly that.

Use Cases: Notification Systems

In information technology, a notification system is a combination of software and hardware that provides a means of delivering a message to a set of recipients. It commonly shows activity related to an account. Such systems constitute an important aspect of modern web applications.

For example, a notification system can send an e-mail announcing when a computer network will be down for a scheduled maintenance.

The complexity of the notification system may vary. Complicated notification systems are used by businesses to reach critical employees. Emergency notification systems may take advantage of modern information technologies. Governments use them to inform people of upcoming danger.

Notification 1: HipChat

```
after-steps:
  - hipchat-notify:
      token: $HIPCHAT_TOKEN
      room-id: id
```

Name	Options	Description
token	(required)	Your HipChat token (retrieve yours from https://hipchat.com/account/api)
room-id	(required)	The ID of the HipChat room (retrieve yours from https://www.hipchat.com/rooms/ids).
passed-message	(optional)	The message which will be shown on a passed build or deploy
failed-message	(optional)	The message which will be shown on a failed build or deploy

Wercker can send notifications on build or deploy events to your HipChat channel.

For notifications, we leverage after-steps that are executed when a build or deploy finishes.

Various notifications for HipChat exist in the step registry, but here we showcase how to use one that Wercker has created.

You define after-steps in your wercker.yml either in your build or deploy pipeline (or both).

Tokens: To be able to post notifications, you need a V2 OAuth token. You can get it on the HipChat settings page of a user. This username will be used when posting notifications.

Notification 2: Slack

```
after-steps:
    - slack-notifier:
        url: $SLACK_URL
        channel: notifications
        username: werckerbot
        notify_on: "failed"
```

The url parameter is the Slack webhook that Wercker should post to. You can create an incoming webhook on your slack integration page. This url is then exposed as an environment variable that you create through the wercker web interface as a deploy pipeline variable.

Wercker can send notifications on build or deploy events to your Slack channel.

For notifications we leverage after-steps that are executed when a build or deploy finishes.

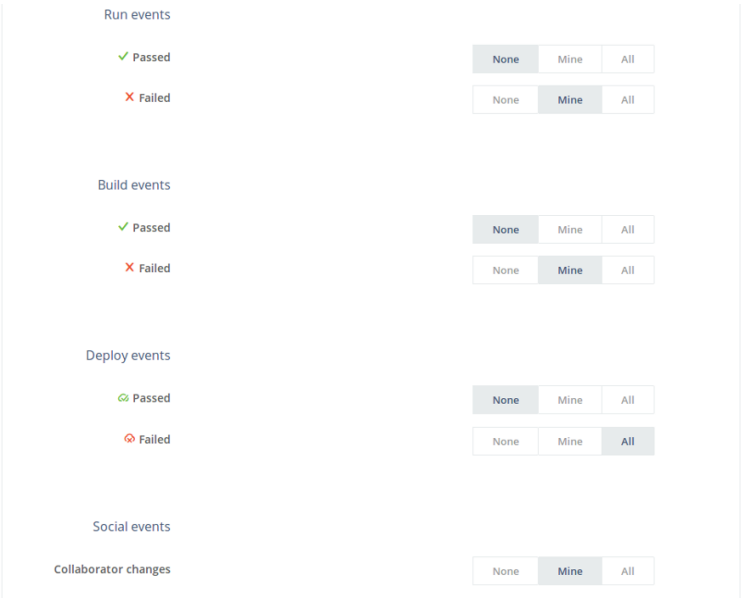
Various notifications for Slack exist in the step registry, but here we showcase how to use one that wercker has created.

You define after-steps in your wercker.yml either in your build or deploy pipeline (or both).

We encourage you to create a separate room for your notifications

The channel is the Slack chat room to which you want to post the notifications. The username defines the name under which notifications should be posted. Finally, the notify_on field specifies if you want to get notifications only on failed builds or deploys.

Notification: Email



Wercker has built-in support for email notifications. This means you do not have to define a custom after-step to get emails on failing or passing builds/deploys. However, built-in email notifications are systemwide and, therefore, cannot be used on application-specific level.

The same events that trigger the system notifications can also trigger emails. The emails get sent to the email address you signed up with.

If you want to change that email address and switch the notifications on or off, you can go to your profile settings page and then click the email link in the side menu.

When you choose to receive email notifications, we give the option to fine-tune this to your own liking. For example, let's say you are a programmer and hard at work on a new feature. You can choose to only receive emails when your own builds fail. Or if you are a manager and want to be notified when a new feature is released, you can choose to receive mails for all successful deploys.

Practice 7: Overview

You will learn how to connect to Slack via Oracle Container Pipeline

Summary

In this lesson, you should have learned how to:

- Describe the use case of GIT
- Implement support for notifications