ORACLE University

Integrated Cloud Applications & Platform Services

Build Container-Native Apps using Oracle Container Pipelines

Activity Guide

D107085GC10 | D107606

Learn more from Oracle University at **education.oracle.com**

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.

1002172020

Table of Contents

Practices for Lesson 3: Introducing Container Architecture

Practices for Lesson 3: Overview

Overview

In this lesson, you will learn how to create Docker Images and test them as containers. Further, the images will be pushed into Oracle Cloud Registry (OCIR). Verify the image and observe the properties.

Note:

The lab files required for the course are part of ekit downloadable.

Assumptions:

The following are required for this practice:

- ➔ OCI Cloud Account and access to OCI Registry
- ➔ Linux Machine

Step 1: Create and Test Docker Image

1. Connect to the Linux machine assigned to you. Details will be shared by your instructor. Open a terminal window and switch to root user

   ```
   su – root
   ```

   If prompted, change the password.
2. Enable Docker as a service.

   ```
   systemctl enable docker
   ```

3. Create directory to store Dockerfiles using the `mkdir` command.

   ```
   mkdir Dockerfiles
   ```

4. Switch to the Dockerfiles directory.

   ```
   cd Dockerfiles
   ```

5. Create a Docker file to build a Docker image

   ```
   vi Dockerfile
   ```

File - Dockerfile

```
from python
add password.py /code/password.py
run chmod +x /code/password.py
entrypoint ["/usr/bin/python","/code/password.py"]
cmd ["$122455ab"]
```

The application is to enable a Python program, which will validate a password entered through command line.

Assumption is the python program file `(password.py)` is already available for deployment in labs folder `(/home/oracle/LabFiles)`.

Copy the file to `Dockerfiles` directory

Command - `cp /home/oracle/LabFiles/password.py .`

For your reference, the python program `password.py` is as given below -

File – password.py

```
import sys
password=sys.argv[1]
print ("password ="+password)
invpass=[]
import re
flag = 0
cnt=0
while (cnt<len(password)):
    if (len(password)<8):
        if 1 not in invpass:
            invpass.append(1)
        flag = -1

    if not re.search("[a-z]", password):
        if 2 not in invpass:
         invpass.append(2)
        flag = -1

    if not re.search("[A-Z]", password):
        if 3 not in invpass:
         invpass.append(3)
        flag = -1

    if not re.search("[0-9]", password):
        if 4 not in invpass:
         invpass.append(4)
        flag = -1
    if not re.search("[_@$]", password):
        if 5 not in invpass:
         invpass.append(5)
        flag = -1
    if re.search("\s", password):
```

```
            if 6 not in invpass:
             invpass.append(6)
            flag = -1
        cnt+=1
    if flag==0:
        print ("Valid Password...")
    else:
       print ("Not Valid Password - Below rules aren\'t met")
    for j in invpass:
        if (j==1):
            print ("Not Enough Length")
        elif (j==2):
            print ("No Lower Case Characters")
        elif (j==3):
            print ("No Upper Case")
        elif (j==4):
            print ("No Digits Characters")
        elif (j==5):
            print("No special characters")
        else:
            print ("No Spaces Allowed")
```

6. Build a successful image using the `docker build` command.

```
docker build -t pythonpass -f Dockerfile .
```

- -t : Name and optionally a tag in the ‘name:tag’ format
- -f : Name of the Dockerfile

Note - There is a dot at the end of the build command indicating that the Dockerfile is in the current directory

```
[root@edvmr1p0 Dockerfiles]# docker build -t pythonpass -f Dockerfile .
Sending build context to Docker daemon  4.096kB
Step 1/5 : from python
 ---> 02d2bb146b3b
Step 2/5 : add password.py /code/password.py
 ---> Using cache
 ---> e7c27ade7f67
Step 3/5 : run chmod +x /code/password.py
 ---> Running in a920a3159b2b
Removing intermediate container a920a3159b2b
 ---> 4e8da1995bc1
Step 4/5 : entrypoint ["/usr/bin/python","/code/password.py"]
 ---> Running in 72213ea2e4d6
Removing intermediate container 72213ea2e4d6
 ---> 588b3e126ed7
Step 5/5 : cmd ["$122455ab"]
 ---> Running in 85fdffd83f74
Removing intermediate container 85fdffd83f74
 ---> eb2ea36263b7
Successfully built eb2ea36263b7
Successfully tagged pythonpass:latest
```

7. Once the image is successfully built, observe Repo Tags and Entrypoint as defined in Dockerfile.

```
docker inspect pythonpass
```

```
[root@edvmr1p0 Dockerfiles]# docker inspect pythonpass
[
    {
        "Id": "sha256:eb2ea36263b7cbb15600c663d7ea4c4d3a8cf4b288d2ff6832d6878e305250a0",
        "RepoTags": [
            "pythonpass:latest"
        ],
        "RepoDigests": [],
        "Parent": "sha256:588b3e126ed730f533f614042522cfca68fec8a0c006e6d44548933a420c1df8",
        "Comment": "",
        "Created": "2019-10-09T11:11:12.402674946Z",
        "Container": "85fdffd83f745e778f4d3674454558ac5eb9b515a1b1f2c871944709072f32e7",
        "ContainerConfig": {
            "Hostname": "85fdffd83f74",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [
                "PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
                "LANG=C.UTF-8",
                "GPG_KEY=0D96DF4D4110E5C43FBFB17F2D347EA6AA65421D",
                "PYTHON_VERSION=3.7.4",
                "PYTHON_PIP_VERSION=19.2.3",
                "PYTHON_GET_PIP_URL=https://github.com/pypa/get-pip/raw/309a56c5fd94bd1134053a541cb4657a4e47e09d/get-pip.py",
                "PYTHON_GET_PIP_SHA256=57e3643ff19f018f8a00dfaa6b7e4620e3c1a7a2171fd218425366ec006b3bfe"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"$122455ab\"]"
            ],
            "ArgsEscaped": true,
            "Image": "sha256:588b3e126ed730f533f614042522cfca68fec8a0c006e6d44548933a420c1df8",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": [
                "/usr/bin/python",
                "/code/password.py"
            ],
```

8. Now let's observe the image created along with its layers.

```
docker history pythonpass
```

```
[root@edvmr1p0 Dockerfiles]# docker history pythonpass
IMAGE               CREATED             CREATED BY                                      SIZE
          COMMENT
eb2ea36263b7        10 minutes ago      /bin/sh -c #(nop)  CMD ["$122455ab"]            0B

588b3e126ed7        10 minutes ago      /bin/sh -c #(nop)  ENTRYPOINT ["/usr/bin/pyt…   0B

4e8da1995bc1        10 minutes ago      |0 /bin/sh -c chmod +x /code/password.py       1.23kB

e7c27ade7f67        13 minutes ago      /bin/sh -c #(nop) ADD file:bdde075d851f4f6db…   1.23kB
```

9. Validate the image by creating the instance of container.

```
docker run -d --name pythontest pythonpass
```

- -d : Run container in background
- -name : Assign a name to the container

```
[root@edvmr1p0 Dockerfiles]# docker run -d --name pythontest pythonpass
7225baa510bf67bd3d085b1f4d48b971a1161d57c12d5ae5edc79374357c8f68
```

10. Check all the docker containers being processed with name as `pythontest`

```
docker ps -a | grep pythontest
```

```
[root@edvmr1p0 Dockerfiles]# docker ps -a|grep pythontest
7225baa510bf        pythonpass          "/usr/bin/python /co…"   About a minute ago   Exited (1)
 About a minute ago                       pythontest
```

11. Let us test the logs of container.

```
docker logs pythontest
```

```
password =$122455ab
Not Valid Password - Below rules aren't met
No Upper Case
```

If you have time, redo with other password combinations (in Dockerfile) and check the logs.

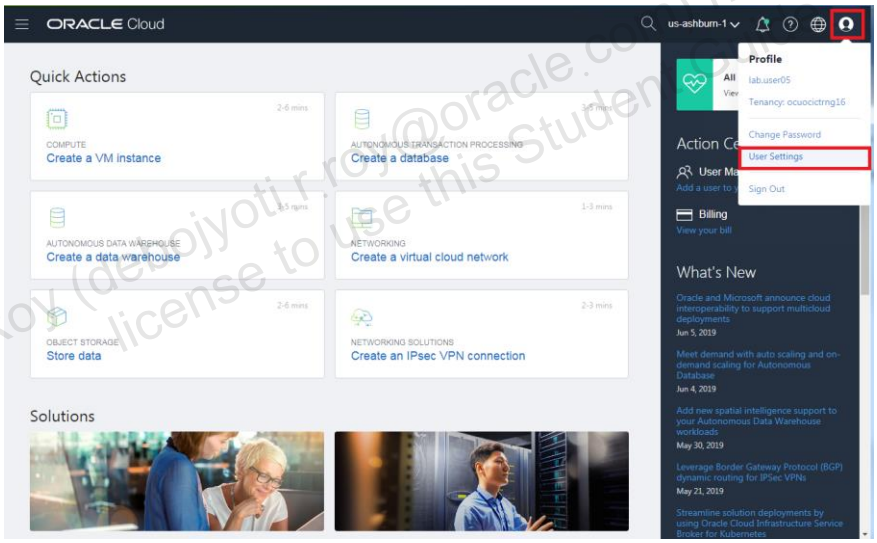
Finally exit as `root` user to switch back to the default `oracle` user.

Step 2: Push the image to OCI Registry

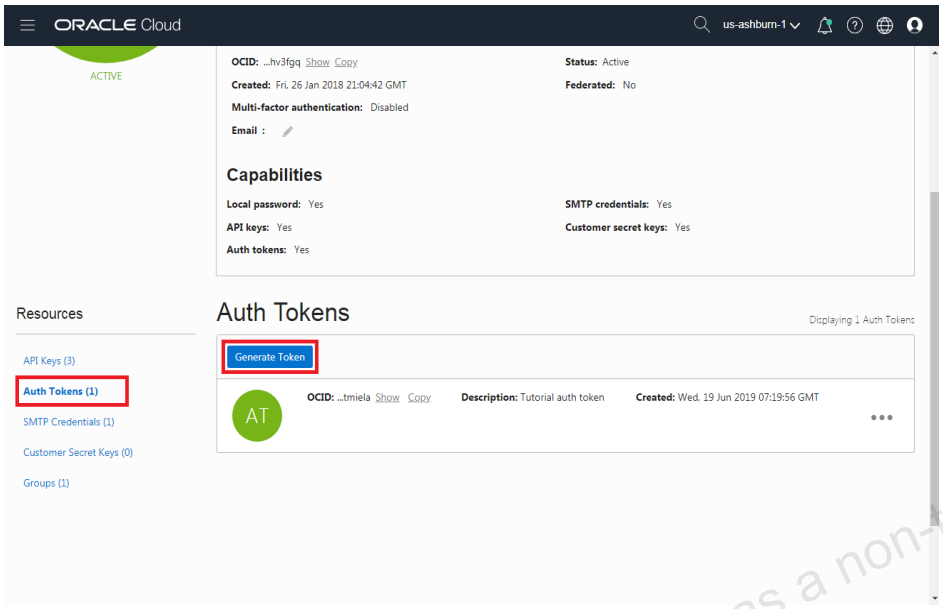
Now let us create an auth token, log in to OCI Registry from the Docker CLI, tag the image to push it to OCI registry and verify the image pushed to OCI Registry.

1. Sign in to **Oracle Cloud Infrastructure**.
    a. In a browser on your local machine, enter the **URL** provided to you (for example, `https://console.us-ashburn-1.oraclecloud.com/`).
    b. Specify a tenant that is assigned to you. Details are provided by the instructor.
    c. Enter username and password that is assigned to you. Details are provided by the instructor.
    d. You will now gain access to OCI Console.

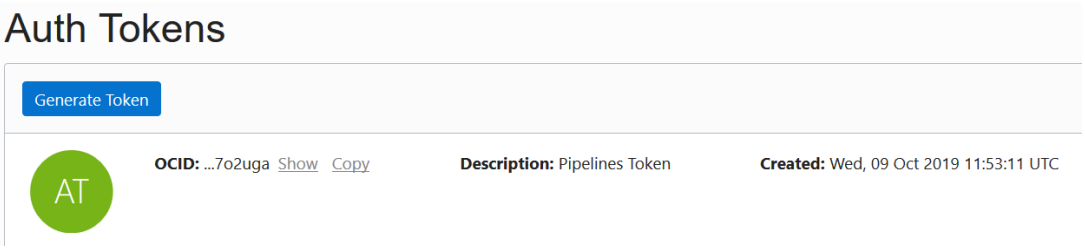
2. Generate an **Auth Token** from Oracle Cloud Infrastructure.
    a. On the top-right corner of the console, you will see the **User menu**; navigate to **User Settings**.



b. On the **User Settings** page, navigate to **Auth Tokens** and then click **Generate Token**.



c. Enter a description for the auth token and click **Generate Token**. The new auth token will be displayed.



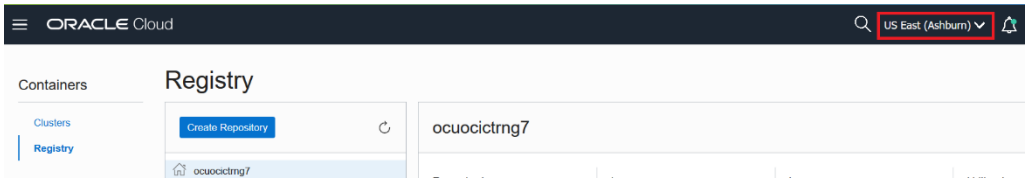
d. Copy the auth token immediately to a secure location from where you can retrieve it later, because later the auth token won't be displayed.

e. Close the **Generate Token** dialog box.

3. Confirm that you can access Oracle Cloud Infrastructure Registry.

a. In the Console, under **Solutions and Platform** in **Navigation** menu, go to **Developer Services** and click **Registry**.

b. Choose the region of the tenancy allocated to you (for example, us-ashburn-1).



4. Log in to OCI Registry from the Docker CLI.
   a. In a terminal window on your Linux machine, log in to Oracle Cloud Infrastructure Registry by entering.

   Notes: `docker login <region-code>.ocir.io`

   - Where <region-code> is the code for the OCI Registry region.
   - **iad** is the region code of **us-ashburn-1**.
   - **Note** : Based on the tenant, you can identify the region code, See the Availability by Region Name and Region Code topic in the Oracle Cloud Infrastructure Registry documentation for the list of region codes.

   b. When prompted, enter your username in the format **<tenancy_name>/<username>.**
   c. When prompted for password, enter the **Auth Token** you copied earlier as the password.

```
[oracle@edvmr1p0 ~]$ docker login iad.ocir.io
Username (ocuocictrng7/lab.user13): ocuocictrng7/lab.user08
Password:
WARNING! Your password will be stored unencrypted in /home/oracle/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

5. Create a reference for the image.
   a. In a terminal window, give a tag to the image that you're going to push to OCI Registry by entering the `docker tag` command.

   Notes: `docker tag <<imagename>>`

   `<region-code>.ocir.io/<tenancy-name>/<repo-name>/<image-name>:<tag>`**`<userID>`**

   `Example:` **`docker tag pythonpass iad.ocir.io/ocuocictrng16/newrepo/pythonpass:latestora001`**

   Where:

   - `<region-code>`: It is the code for the OCI Registry region you are using. For example, for **`us-ashburn-1`** it's **`iad.`**
   - `ocir.io`: It is the OCI Registry name.
   - `<tenancy-name>`: The name of the tenancy to which the image has been pushed.
   - `<repo-name>`: (optional) The name of the repository to which you want to push the image. If this is not specified, then the name of the image is used as the repository name in OCI Registry.

- `<image-name>`: The name you want to give the image in OCI Registry.
- `<tag>`: It is the image tag you want to give the image in OCI Registry.

```
[oracle@edvmr1p0 ~]$ docker tag pythonpass iad.ocir.io/ocuocictrng7/newrepo/pythonpass:latestora001
[oracle@edvmr1p0 ~]$
```

6. Push the image to OCI Registry.
   a. In your terminal window, push the **Docker image** from the Docker CLI to OCI Registry.

      Notes: `docker push`

      `<region-code>.ocir.io/<tenancy-name>/<repo-name>/<image-name>:<tag>`**`<userid>`**

      `Example:` **`docker push iad.ocir.io/ocuocictrng16/newrepo/pythonpass:latestora001`**

      Where:

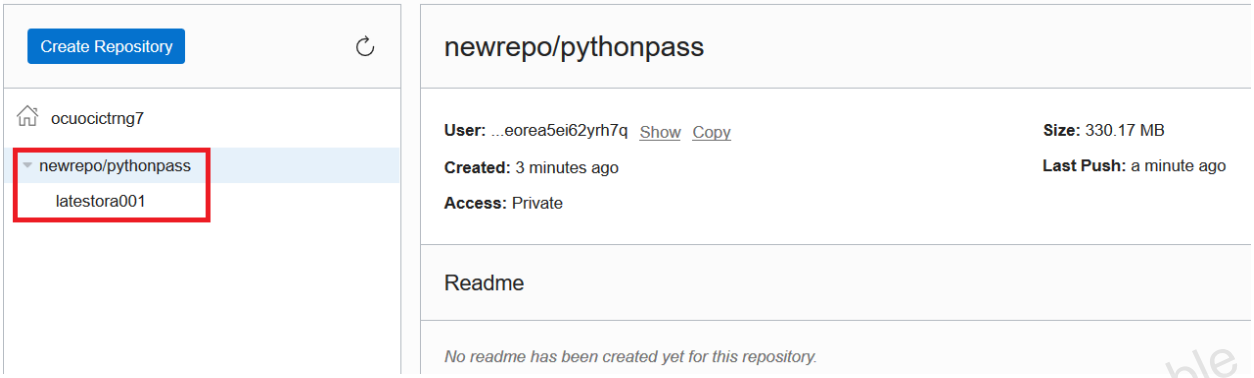
      - `<region-code>`: It is the code for the OCI Registry region you are using. For example, for `us-ashburn-1` it's **`iad`**.
      - `ocir.io`: It is the OCI Registry name.
      - `<tenancy-name>`: The name of the tenancy to which the image has been pushed.
      - `<repo-name>`: (optional) The name of the repository to which you want to push the image. If this is not specified, then the name of the image is used as the repository name in OCI Registry.
      - `<image-name>`: The name you want to give the image in OCI Registry.
      - `<tag>`: It is the tag you want to give the image in OCI Registry.

   The different layers of the image will be pushed successfully.

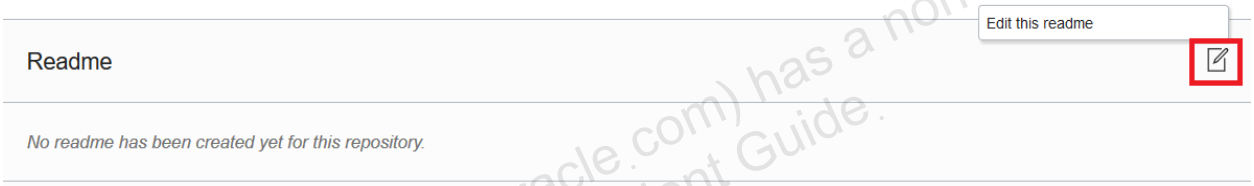
```
[oracle@edvmr1p0 ~]$ docker push iad.ocir.io/ocuocictrng7/newrepo/pythonpass:latestora001
The push refers to repository [iad.ocir.io/ocuocictrng7/newrepo/pythonpass]
19b7357e86f6: Pushed
6ae321ef967a: Pushed
46ed3d879948: Pushed
fbeeb71995b3: Pushed
bb9e1c111e49: Pushed
ac3ac7a153b5: Pushed
3bfeb766f97b: Pushed
ea1227feeccb: Pushed
9cae1895156d: Pushed
52dba9daa22c: Pushed
78c1b9419976: Pushed
latestora001: digest: sha256:4d2c34e4f0814f96d8c7c33c0917dd4bb0d7f058bd58bdf4dcd8be2f9fc807bd size: 2631
[oracle@edvmr1p0 ~]$
```

7. Verify that the image is pushed to OCIR
   a. Navigate to **OCI Registry**.
   b. Click the name of the **newrepo** repository that contains the image that you just pushed. You see:
      - The different images in the repository. In this case, there is only one image, with the tag latest.

- Details about the repository, including who created it and when, its size, and whether it's a public or a private repository
- The readme associated with the repository. In this case, there is no readme yet.

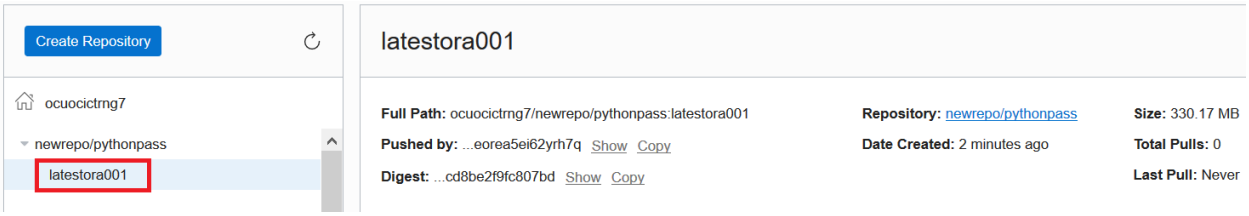


8. Provide a readme for the **newrepo** repository as follows:
   a. Click the **Edit** button in the **Readme** section.



   b. On the **Edit** tab of the Edit Readme dialog, select the **Markdown** option and copy and paste the following description of the **pythonpass** image into the **Content** field:
      a. *The First Docker Image pushed by oraXX with Dockerfile <<Filename>>. Contact <<your email>> for any documentation and queries. Limited Edition and not for Production.*
   c. Click the **Preview** tab to see how the readme will appear.
   d. Click **Save** to close the Edit Readme dialog.

   Note – Readme is associated with the repository and not the image. Hence the description updated by you could be overwritten by another participant.
9. Click the **latest** image tag. The details section shows you the size of the image, when it was pushed and by which user, and the number of times the image has been pulled.



Practices for Lesson 4: Exploring Web Interfaces

Practices for Lesson 4: Overview

Overview

In this lesson, you will learn how to utilize Container Pipeline using Wercker via Oracle Cloud. This pipeline will demonstrate CI/CD operations from building the image initiated from Dockerfile, setting the container, and running the application. It will also further lead to pushing the image to Docker registry.

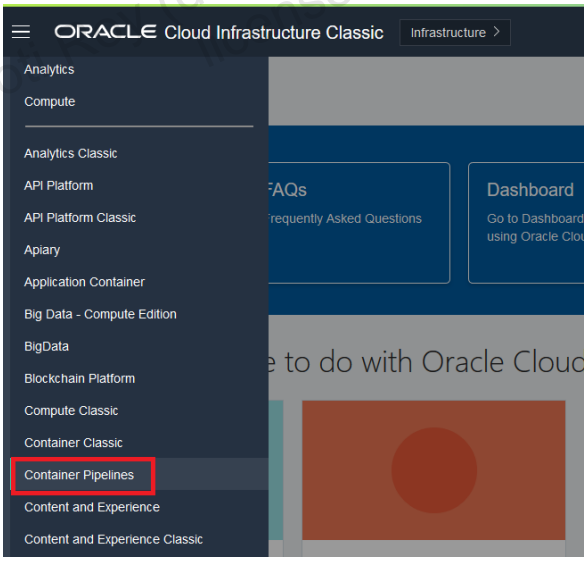
Assumptions:

The following are required for this practice:

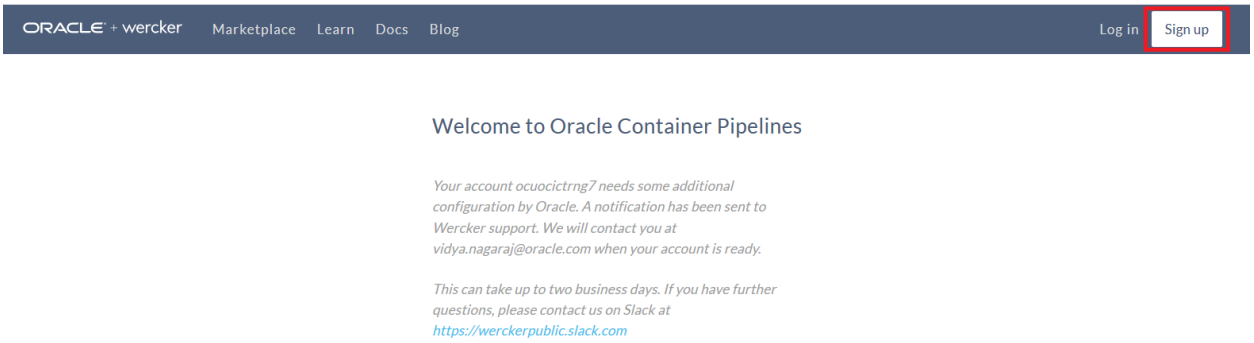
- Cloud account with access to Container Pipelines (app.wercker.com)
- GIT account with commit permissions
- Hub.docker.com (Cloud Docker Account)

Step 1: Access Container Pipelines

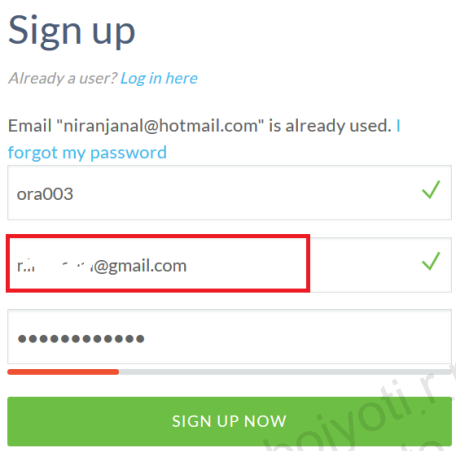
1. Sign in to IDCS Oracle Cloud (via URL - `https://myservices-ocuocictrng17.console.oraclecloud.com/mycloud/cloudportal/dashboard`) using the assigned credentials. For example -
   - `Username - ora001`
   - `Password - O44nS013y`
2. You will gain access to Oracle Cloud Dashboard. Click the hamburger menu in the left pane and choose **Container Pipelines** service.



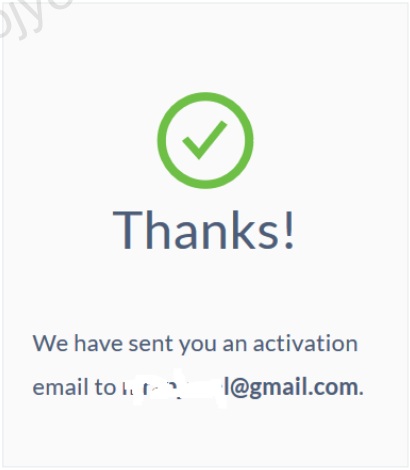
3. Sign up for an account using your desired credentials.



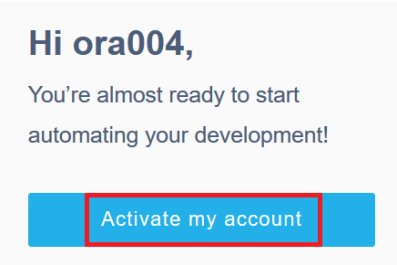
4. Along with username and password, specify an e-mail account during sign up.



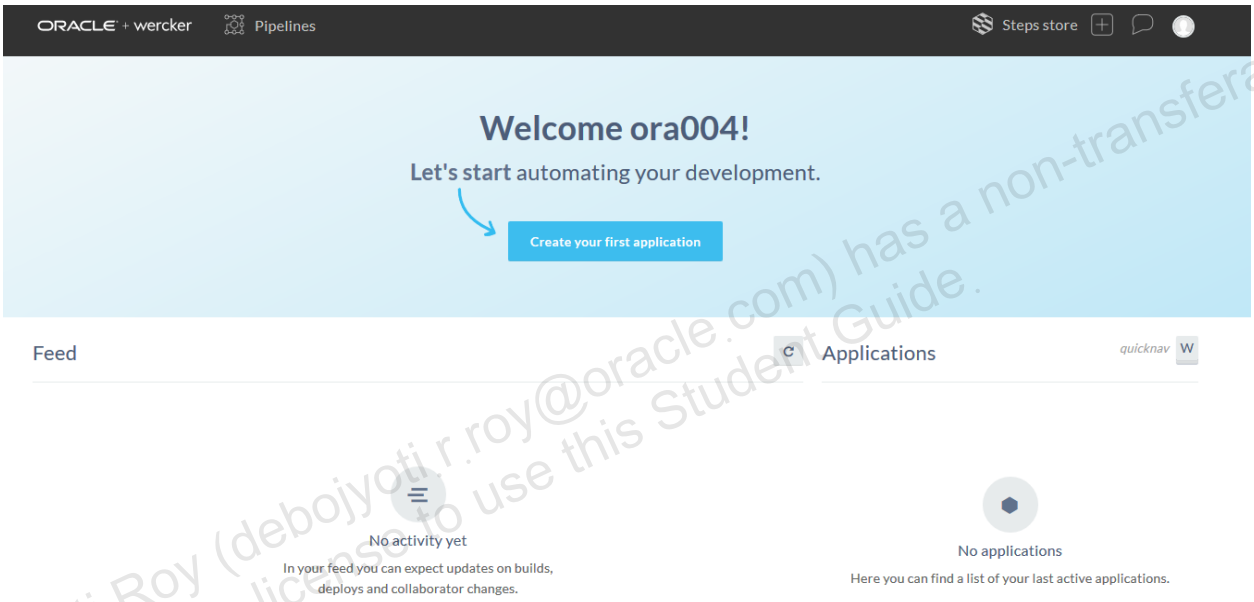
5. You will receive an activation e-mail to the account specified during sign up.



6. Now login to your mailbox and activate your wercker account. Look for an email with subject line as - *Welcome to wercker*



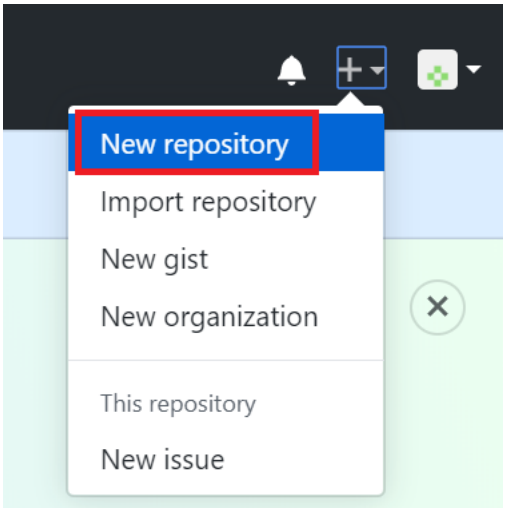
7. Post activation, you will gain access to the **Applications** page.



Step 2: Get the applications in place

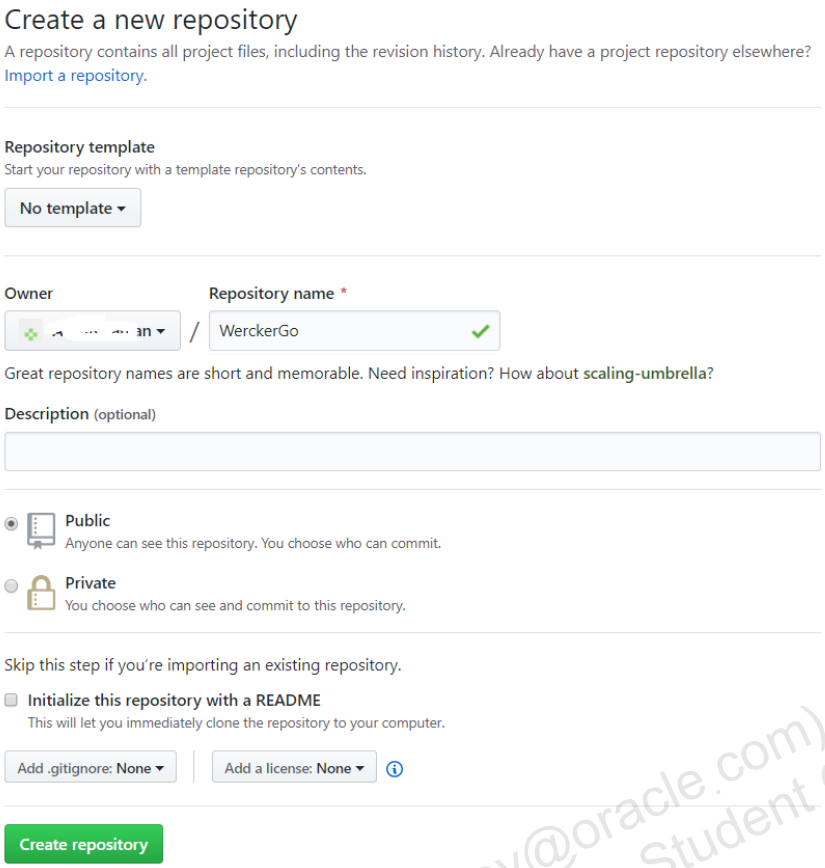
We are deploying a Go application that has a port server. We will be using the Github. This works with Direct Docker Daemon, which will implement a CI and CD Pipeline.

1. Open your GitHub Account, choose **New repository**.

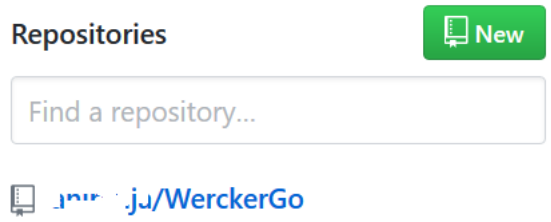


2. Create a repository by name `WerckerGo`.

Public repositories are accessed directly, while private repositories are accessed by SSH Keys internally. For this exercise, we will use a public repository.



3. Confirm that the repository is created



1. Create the file with file name: `main.go` with below snippet.

```
package main

import (
      "encoding/json"
      "log"
      "net/http"
```

```
)

type citiesResponse struct {
	Cities []string `json:"cities"` // Cities capitalised to export
it, otherwise json encoder will ignore it.
}

func cityHandler(res http.ResponseWriter, req *http.Request) {
	cities := citiesResponse{
		Cities: []string{"Amsterdam", "Berlin", "New York", "San
Francisco", "Tokyo"}}

	res.Header().Set("Content-Type", "application/json; charset=utf-
8")
	json.NewEncoder(res).Encode(cities)
}

func defaultHandler(res http.ResponseWriter, req *http.Request) {
	res.Header().Set("Content-Type", "text/plain; charset=utf-8")
	res.Write([]byte("Hello World!!"))
}

func main() {
	http.HandleFunc("/", defaultHandler)
	http.HandleFunc("/cities.json", cityHandler)
	err := http.ListenAndServe(":5000", nil)
	if err != nil {
		log.Fatal("Unable to listen on port 5000 : ", err)
	}
}
```

Commit the file once done.

2. Create a file named `main_test.go`.

```
package main
import (
	"net/http"
	"net/http/httptest"
```

```
      "testing"
)
func TestHandleIndexReturnsWithStatusOK(t *testing.T) {
      request, _ := http.NewRequest("GET", "/", nil)
      response := httptest.NewRecorder()

      cityHandler(response, request)
      if response.Code != http.StatusOK {
            t.Fatalf("Non-expected status code%v:\n\tbody: %v",
"200", response.Code)
      }
}
```

a.

b. **Commit** the file.

c.

3. Create a file named `Dockerfile`.

```
FROM golang
WORKDIR /work
ADD . .
RUN go test ./...
RUN go build -o /bin/myapp .
WORKDIR /
RUN rm -r /work
CMD ["/bin/myapp"]
```

a. **Commit** the file

4. Create a file named `wercker.yml`.

a. **Note:** Pay special attention to indentation while working with YAML.

b. You can verify your yaml file using YAML Validator - https://codebeautify.org/yaml-validator

c. For your reference, the yml file is posted in the Linux machine @ `/home/oracle/LabFiles/wercker_lab4.yml`

```
d. build:
e.   box: alpine
f.   docker: true
```

```
    steps:
      - script:
          name: Install docker
          code: apk --no-cache add docker
      - script:
          name: Install Curl
          code: apk --no-cache add curl
      - script:
          name: Explore the docker daemon
          code: |
            echo DOCKER_NETWORK_NAME=$DOCKER_NETWORK_NAME
            echo DOCKER_HOST=$DOCKER_HOST
            echo Running docker version
            docker version
            echo running docker ps
            docker ps
            echo running docker images
            docker images
            echo listing the docker networks
            docker network list
      - script:
          name: create an image from the Dockerfile using docker
    build
          code: |
            docker build -t docker.io/$USERNAME/direct-docker-
    golang:latest .
            echo running docker images
            docker images
      - script:
          name: start the newly-created image using docker run
          code: |
            docker run --rm -d -p 5000:5000 --name testcontainer \
              --network=$DOCKER_NETWORK_NAME
    docker.io/$USERNAME/direct-docker-golang:latest
            echo running docker ps
            docker ps
```

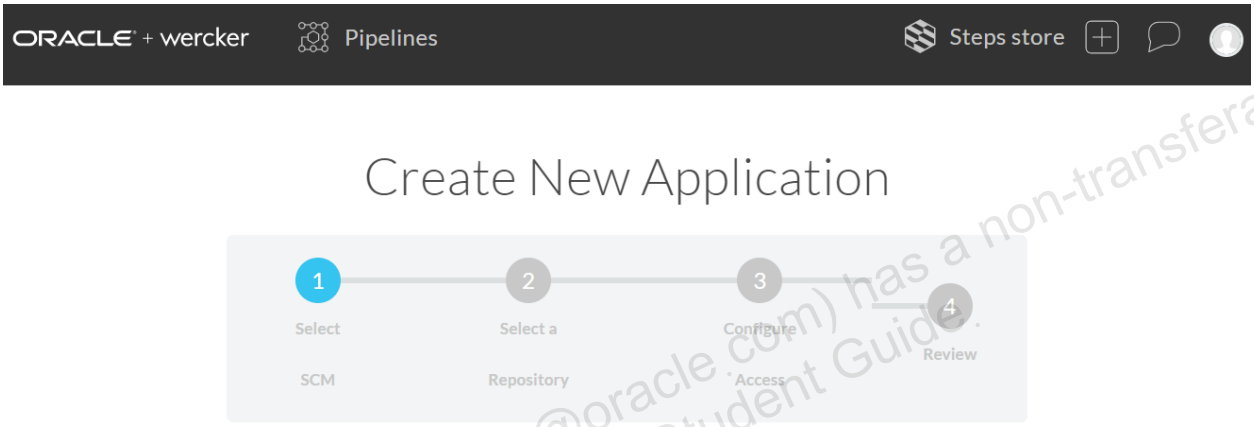
```
    - script:
        name: Wait for container to start
        code: |
          until $(curl --output /dev/null --silent --head --fail
  http://testcontainer:5000); do printf '.'; sleep 5; done
    - script:
        name: Test the container that we started using docker-
  run
        code: |
            if curlOutput=`curl -s testcontainer:5000`; then
                export expected="Hello World!!"
                if [ "$curlOutput" == "$expected" ]; then
                    echo "Test passed: container gave expected
  response: " $expected
                else
                    echo "Test failed: container gave unexpected
  response: " $curlOutput
                    echo "The expected response was: " $expected
                    exit 1
                fi
            else
                echo "Test failed: container did not respond"
                exit 1
            fi
    - script:
        name: kill the container using docker kill
        code: |
          docker kill testcontainer
          echo running docker ps
              docker ps
    - script:
        name: push the newly-create image to a repository
        code: |
          docker login -u $USERNAME -p $PASSWORD
          docker push docker.io/$USERNAME/direct-docker-
  golang:latest
```

ttt. **Commit** the file.

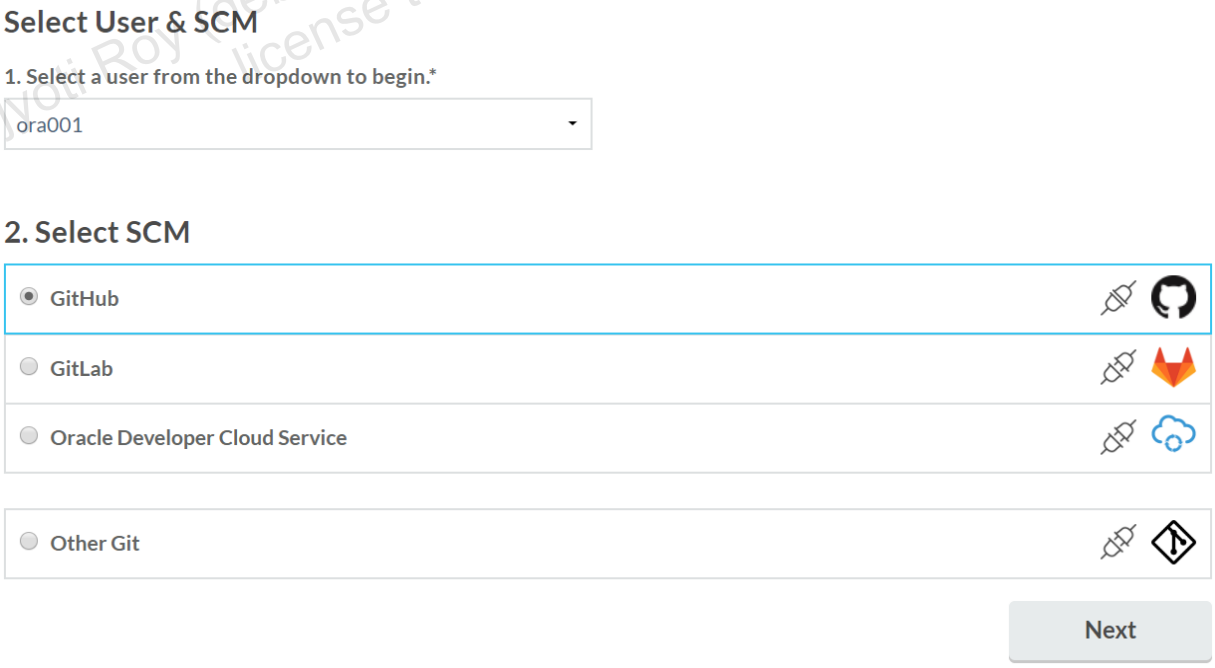
Step 3: Set up the CI/CD pipeline by creating an application

1. Access Container Pipelines application page, and click **Add application**

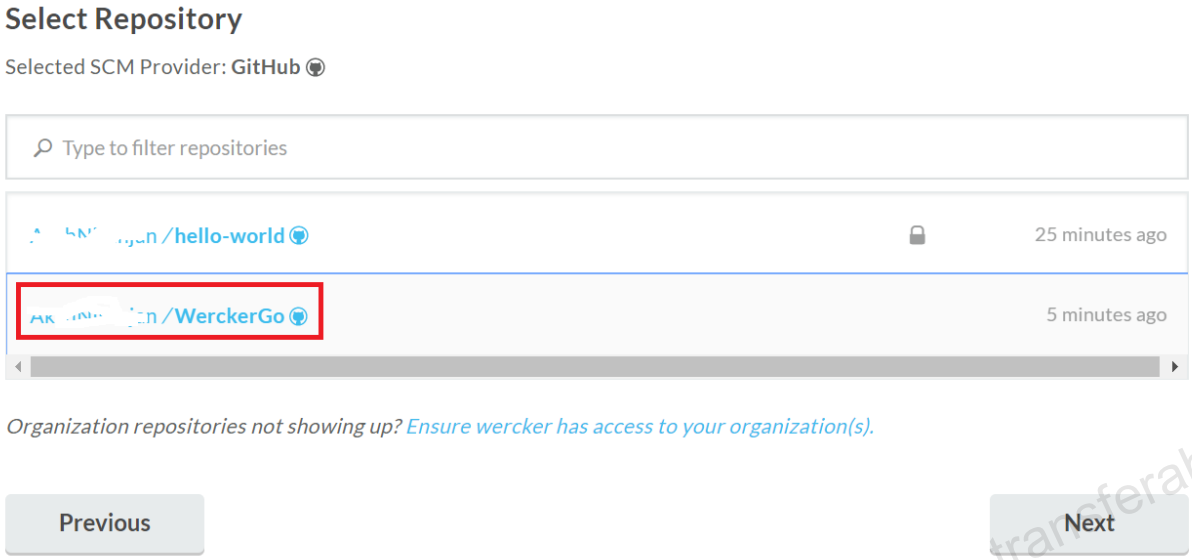
2. Complete the wizard to create application.



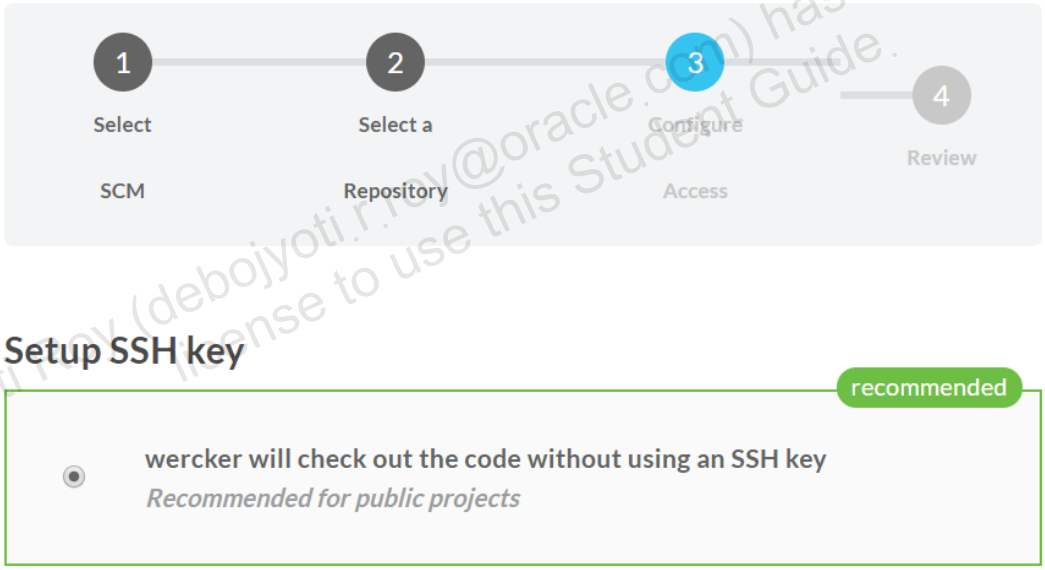
3. Authorize **GitHub** with username and password for Container Pipeline. Click **Next**.



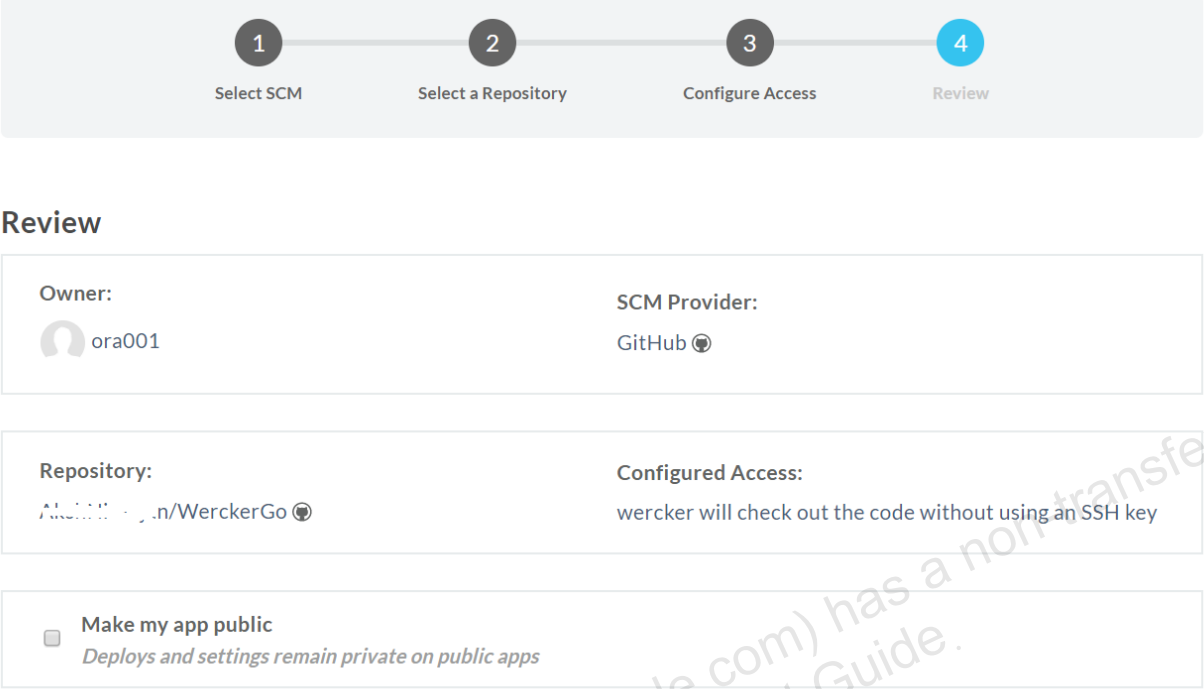
4. Choose the repository - `WerckerGo` (created in previous step)



5. Configure access without keys. Click **Next**.



6. Review and click **Create** to create application.

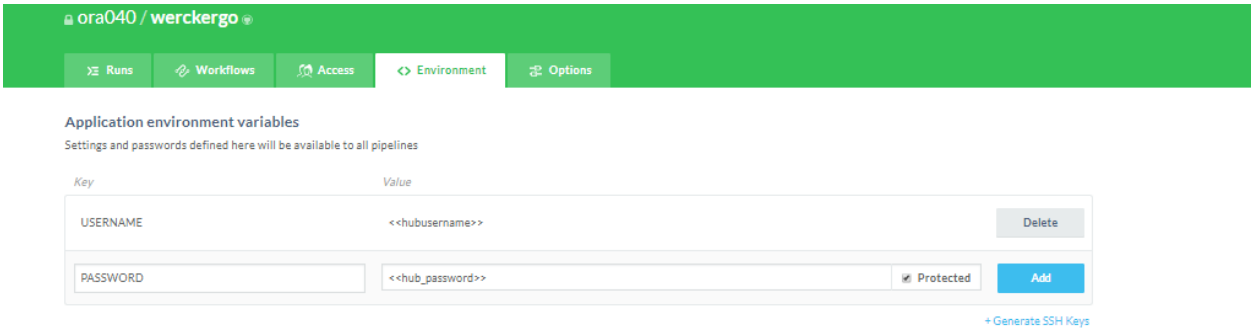


Before your build the application, let us understand the environments required for this application.

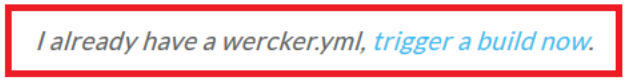
7. Click the **Environment** tab in this page below.

If you have observed the wercker.yml file, there are two environmental variables used - **USERNAME** and **PASSWORD**. These are your credentials for `hub.docker.com` for your default cloud registry in docker.io. Password should be protected as it is encrypted data while username can be generic.

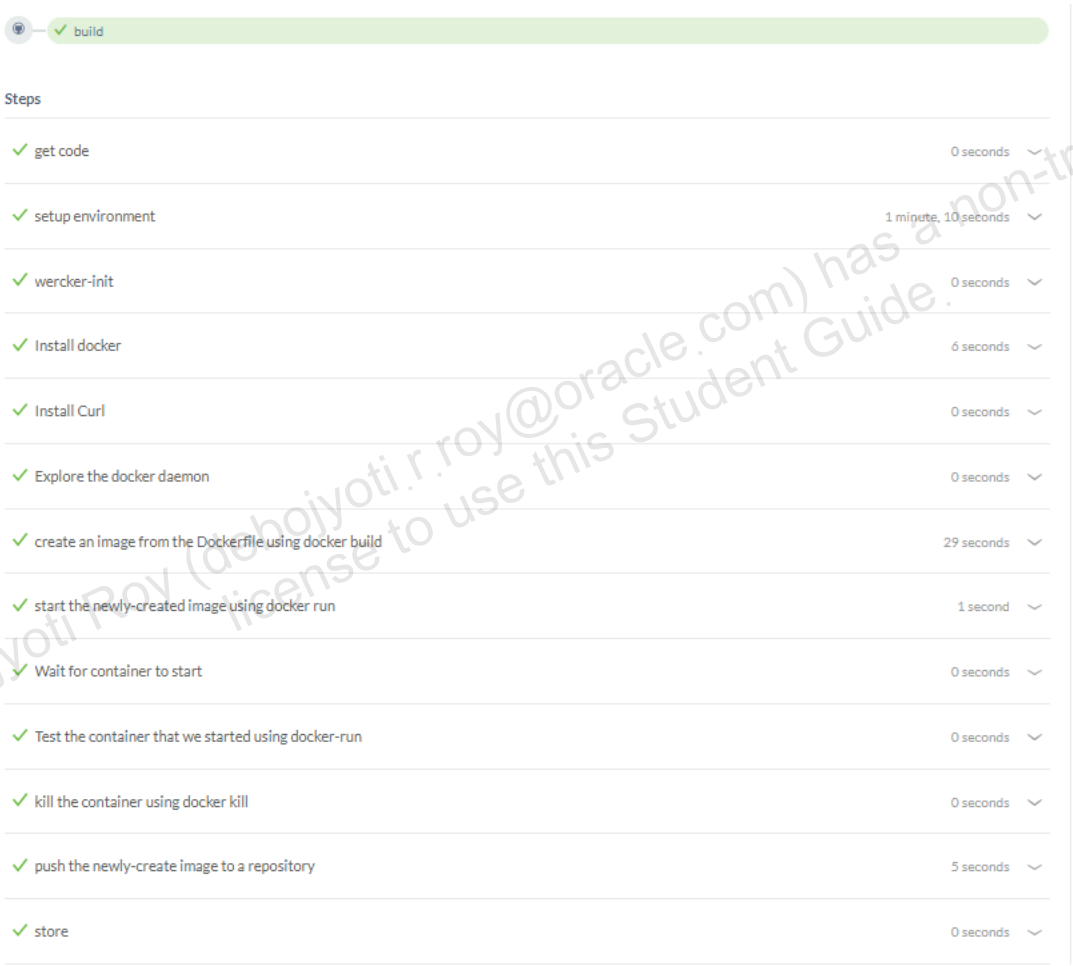
Create two environmental variables as below. Specify the username and password as you have created for `hub.docker.com`.



8. Click on the **Runs** tab and trigger the build. You already have a `wrecker.yml`.



9. Upon successful runs, the entire store is built as an automated pipeline. Now, let's observe the sections of install docker, creating image from Dockerfile using docker build, wait for the container to start, and observe the application.



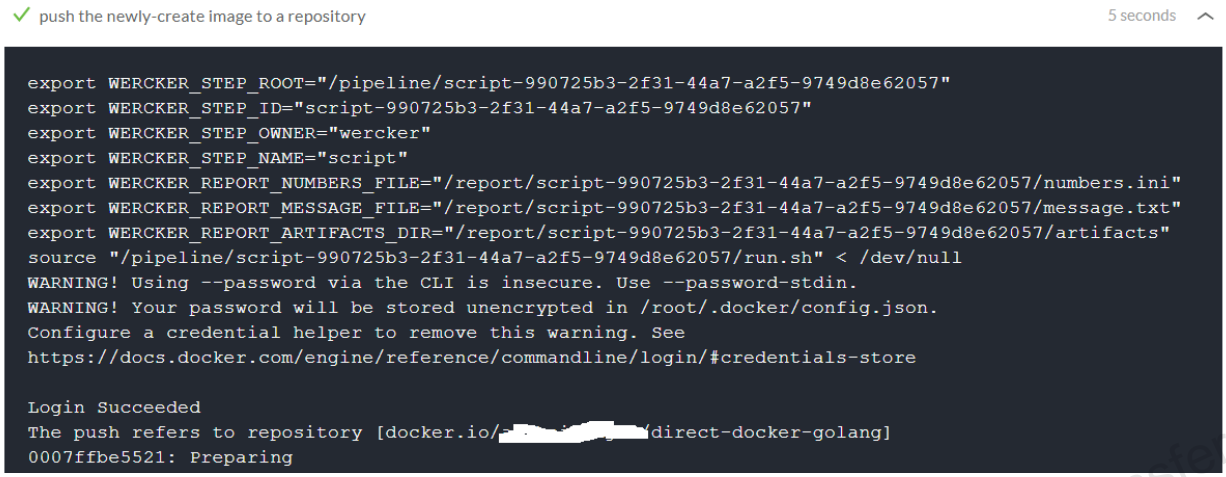
✓ Install docker 6 seconds

```
export WERCKER_STEP_ROOT="/pipeline/script-45a53b0f-f73a-45de-986f-ac7a266618fc"
export WERCKER_STEP_ID="script-45a53b0f-f73a-45de-986f-ac7a266618fc"
export WERCKER_STEP_OWNER="wercker"
export WERCKER_STEP_NAME="script"
export WERCKER_REPORT_NUMBERS_FILE="/report/script-45a53b0f-f73a-45de-986f-ac7a266618fc/numbers.ini"
export WERCKER_REPORT_MESSAGE_FILE="/report/script-45a53b0f-f73a-45de-986f-ac7a266618fc/message.txt"
export WERCKER_REPORT_ARTIFACTS_DIR="/report/script-45a53b0f-f73a-45de-986f-ac7a266618fc/artifacts"
source "/pipeline/script-45a53b0f-f73a-45de-986f-ac7a266618fc/run.sh" < /dev/null
fetch http://dl-cdn.alpinelinux.org/alpine/v3.10/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.10/community/x86_64/APKINDEX.tar.gz
(1/12) Installing ca-certificates (20190108-r0)
(2/12) Installing libseccomp (2.4.1-r0)
(3/12) Installing runc (1.0.0_rc8-r0)
(4/12) Installing containerd (1.2.9-r0)
(5/12) Installing libmnl (1.0.4-r0)
(6/12) Installing libnftnl-libs (1.1.3-r0)
(7/12) Installing iptables (1.8.3-r0)
(8/12) Installing tini-static (0.18.0-r0)
(9/12) Installing device-mapper-libs (2.02.184-r0)
(10/12) Installing docker-engine (18.09.8-r0)
(11/12) Installing docker-cli (18.09.8-r0)
(12/12) Installing docker (18.09.8-r0)
Executing docker-18.09.8-r0.pre-install
Executing busybox-1.30.1-r2.trigger
Executing ca-certificates-20190108-r0.trigger
OK: 278 MiB in 26 packages
```

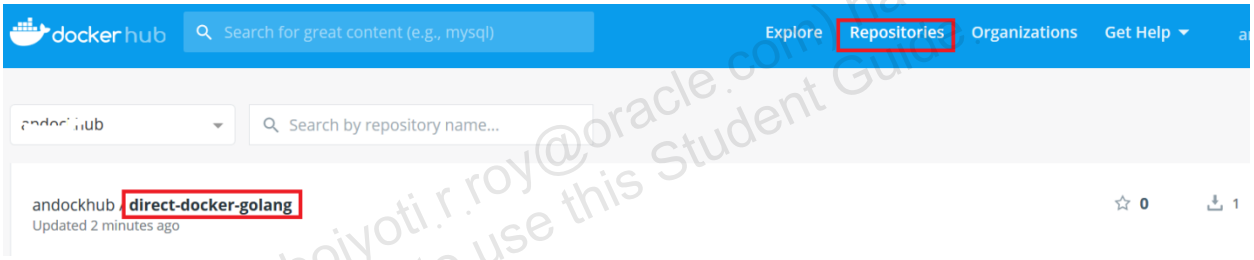
✓ Install Curl 0 seconds

```
export WERCKER_STEP_ROOT="/pipeline/script-f912fc32-fa81-48e7-9592-6ad230f3da0d"
export WERCKER_STEP_ID="script-f912fc32-fa81-48e7-9592-6ad230f3da0d"
export WERCKER_STEP_OWNER="wercker"
export WERCKER_STEP_NAME="script"
export WERCKER_REPORT_NUMBERS_FILE="/report/script-f912fc32-fa81-48e7-9592-6ad230f3da0d/numbers.ini"
export WERCKER_REPORT_MESSAGE_FILE="/report/script-f912fc32-fa81-48e7-9592-6ad230f3da0d/message.txt"
export WERCKER_REPORT_ARTIFACTS_DIR="/report/script-f912fc32-fa81-48e7-9592-6ad230f3da0d/artifacts"
source "/pipeline/script-f912fc32-fa81-48e7-9592-6ad230f3da0d/run.sh" < /dev/null
fetch http://dl-cdn.alpinelinux.org/alpine/v3.10/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.10/community/x86_64/APKINDEX.tar.gz
(1/3) Installing nghttp2-libs (1.39.2-r0)
(2/3) Installing libcurl (7.66.0-r0)
(3/3) Installing curl (7.66.0-r0)
Executing busybox-1.30.1-r2.trigger
OK: 279 MiB in 29 packages
```

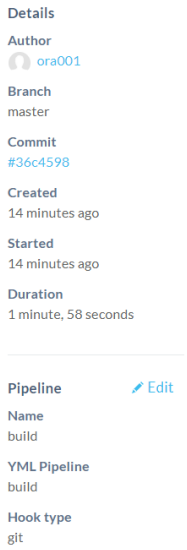
10. Upon successful build, you can see that the image is pushed to repository. You will be able to see the image is successfully pushed to `hub.docker.com` under your repository name.



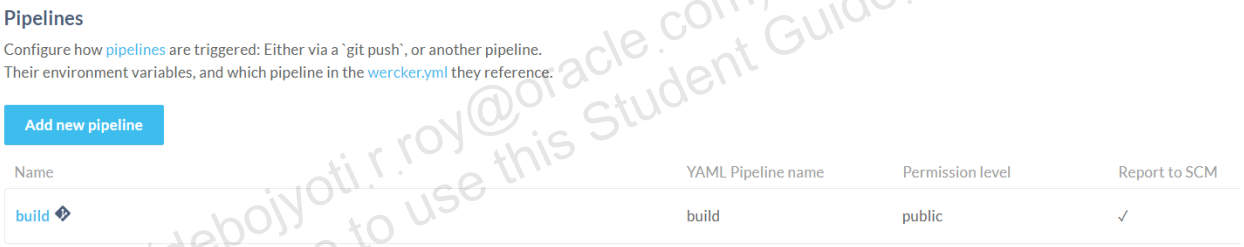
11. Now log in to your docker hub account. Check out the image under your repositories that are created by Wercker pipeline.



12. In Container Pipelines, switch to the Runs tab. Click on the successful build to view the duration of the run:



13. You will also be able to observe the build workflows on the **Workflows** tab in Container Pipelines.



Practices for Lesson 5: Development Workflow

Practices for Lesson 5: Overview

Overview

In this lesson, you will learn how to install Wercker CLI and leverage Container Pipelines as service.

Assumptions:

The following are required for this practice:

- Cloud account with access to Container Pipelines (app.wercker.com)
- GIT account with commit permissions
- Linux Machine

Step 1: Download and install the Wercker CLI

1. Connect to the Linux machine assigned to you. Details will be shared by your instructor. Open a terminal window and switch to root user

   ```
   su – root
   ```

2. Download the install Wercker CLI using the following command

   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.

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

```
[root@edvmr1p0 ~]# curl -L https://s3.amazonaws.com/downloads.wercker.com/cli/stable/lin
ux_amd64/wercker -o /usr/local/bin/wercker
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 36.3M  100 36.3M    0     0  3333k      0  0:00:11  0:00:11 --:--:-- 3563k
[root@edvmr1p0 ~]#
```

3. Set the correct permissions.

   ```
   chmod u+x /usr/local/bin/wercker
   ```

4. Run the version command to verify wercker was installed correctly.

   ```
   wercker --version
   ```

```
[root@edvmr1p0 bin]# wercker --version
wercker version 1.0.1555 (Compiled at: 2019-04-05T16:07:29Z, Git commit: 6dabecf
7daf1e1a050659b81365c5d994f7d5ad4)
[root@edvmr1p0 bin]#
```

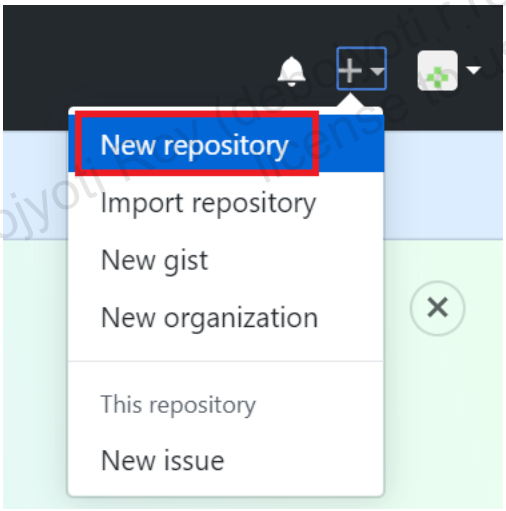
5. You can find out more about the utility by running wercker help command.

   ```
   wercker --help
   ```

```
[root@edvmr1p0 ~]# wercker --help
NAME:
   wercker - build and deploy from the command line

USAGE:
   wercker [global options] command [command options] [arguments...]

VERSION:
   1.0.1555 (Compiled at: 2019-04-05T16:07:29Z, Git commit: 6dabecf7daf1e1a050659b81365c5d994f7d5ad4)

AUTHOR:
  Team wercker - <pleasemailus@wercker.com>

COMMANDS:
   build, b     build a project
   dev  develop and run 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       logout from wercker
   pull, p      pull <build id>
   version, v   print versions
   doc  Generate usage documentation
   docker       docker <docker-command> <args>...
   step, s      manage steps
```

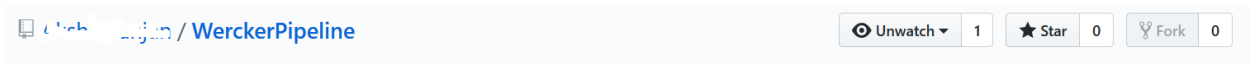
Step 2: Create a Container Application and Pipeline

1. Open your GitHub Account, choose **New repository**.



2. Create a repository by name `WerckerPipeline`.

Public repositories are accessed directly, while private repositories are accessed by SSH Keys internally. For this exercise, we will use a public repository.



3. Create the file with file name: `wercker.yml` with below snippet.

Note: Pay special attention to indentation while working with YAML.

You can verify your yaml file using YAML Validator - https://codebeautify.org/yaml-validator

For your reference, the yml file is posted in the Linux machine @
`/home/oracle/LabFiles/wercker_lab5.yml`

```
box: ubuntu
build:
  steps:
    - script:
      name: step one
      code: echo "Step One"
    - script:
      name: step two
      code: echo "Step Two"
    - script:
      name: step three
      code: echo "Step Three"
    - script:
      name: step four
      code: echo "Step Four"
    - script:
      name: step five
      code: echo "Step Five"
deploy:
  steps:
    - script:
      name: step one
      code: echo "Step One"
    - script:
      name: step two
      code: echo "Step Two"
    - script:
      name: step three
      code: echo "Step Three"
    - script:
      name: step four
      code: echo "Step Four"
    - script:
      name: step five
      code: echo "Step Five"
```

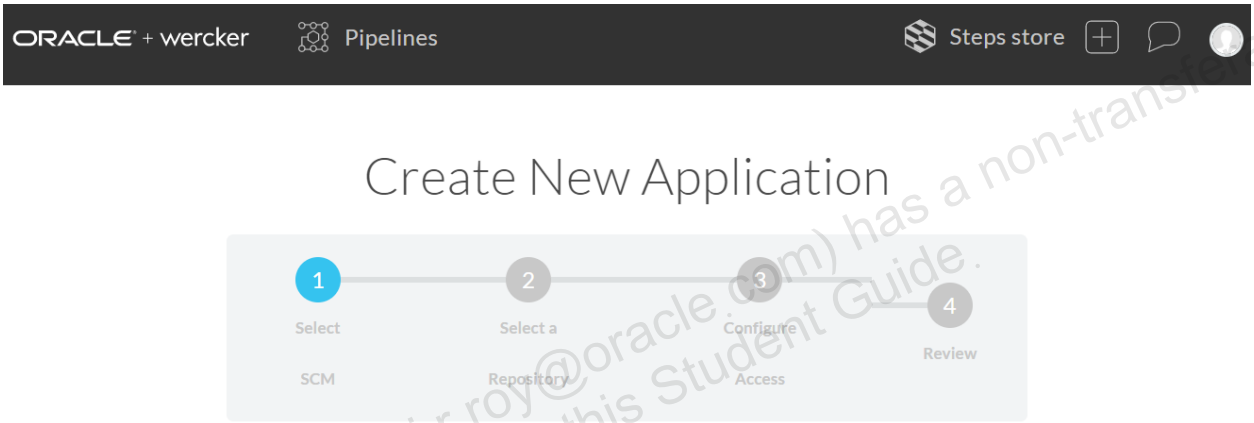
Note: The yml has 2 pipelines, `build` and `deploy`.

- `build` – is an automatic default pipeline that gets automatically executed.
- `deploy` – needs to be triggered explicitly.

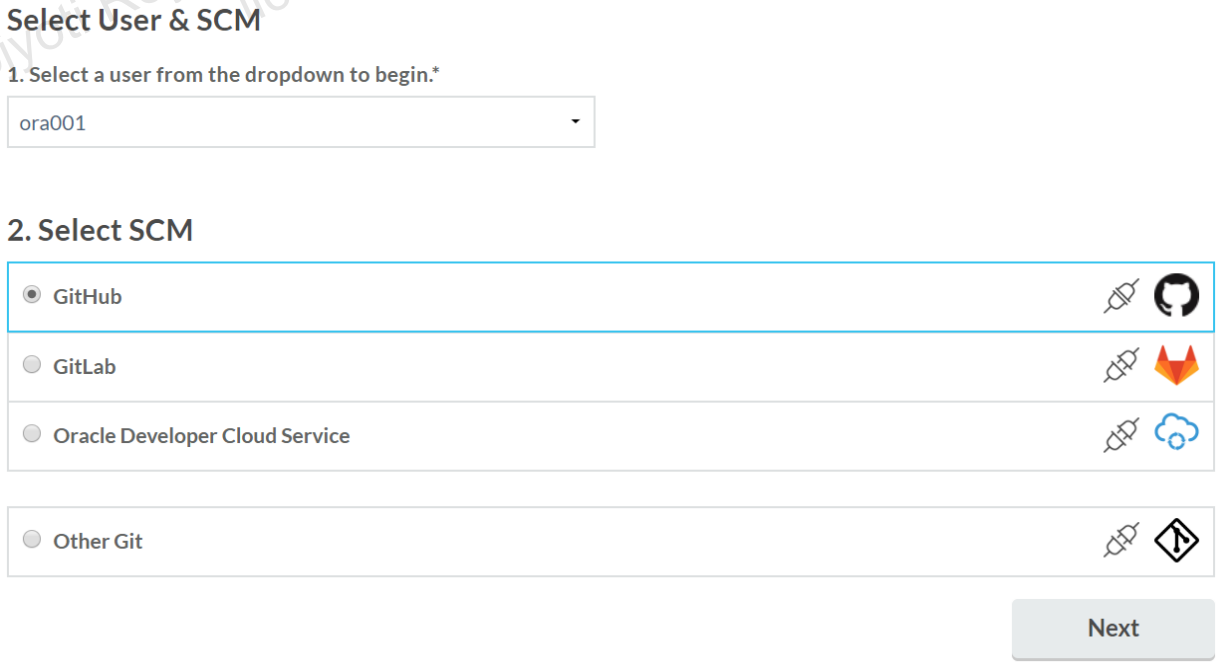
Commit the file once done.

4. Access Container Pipelines application page, and click **Add application**

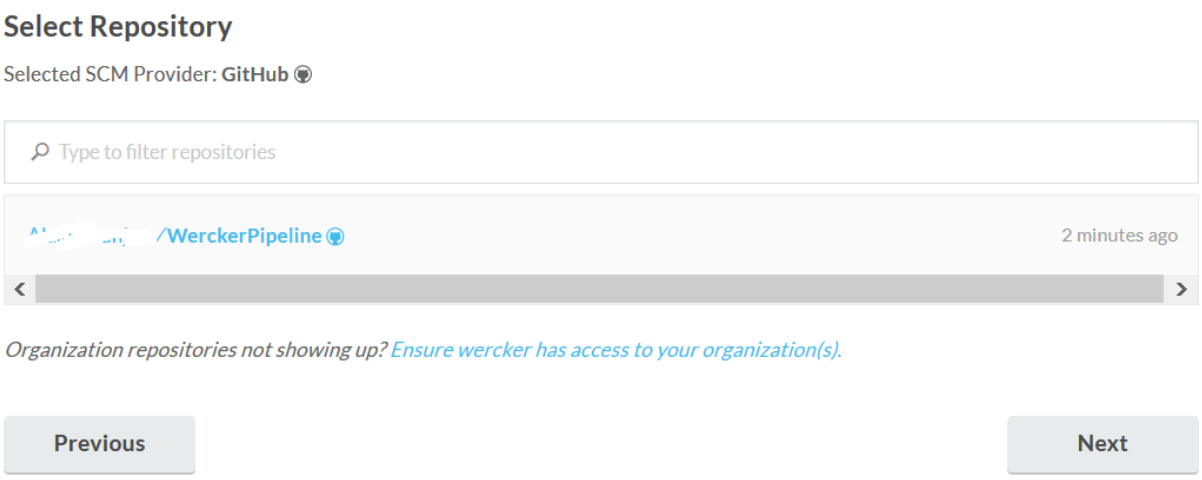
5. Complete the wizard to create application.



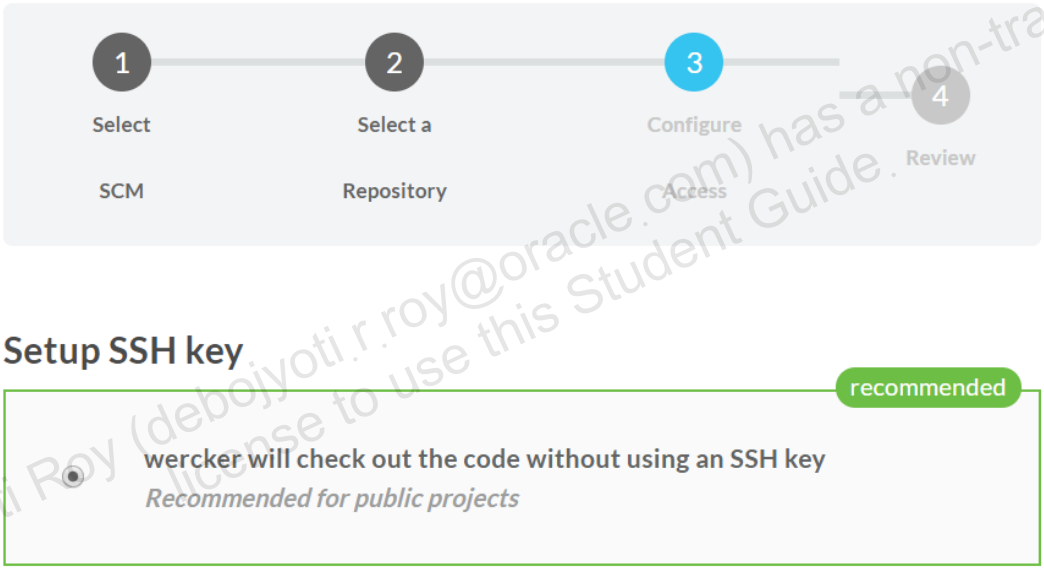
6. Authorize **GitHub** with username and password for Container Pipeline. Click **Next**.



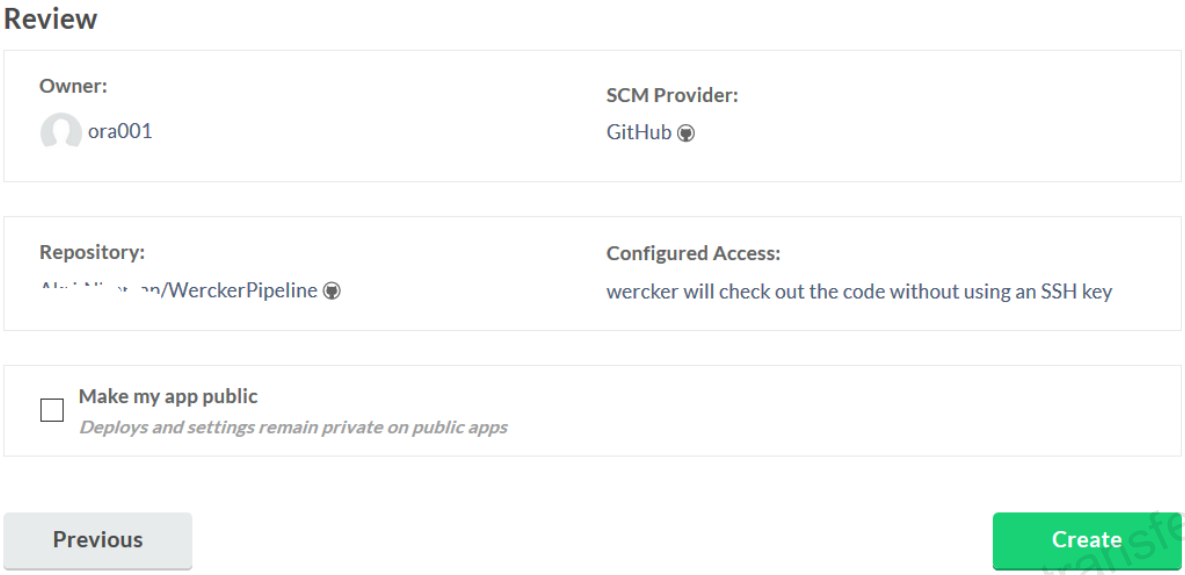
7. Choose the repository - `WerckerPipeline` (created in previous step)



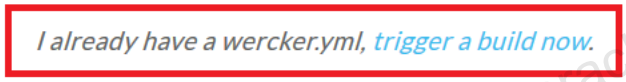
8. Configure access without keys. Click **Next**.



9. Review and click **Create** to create application.

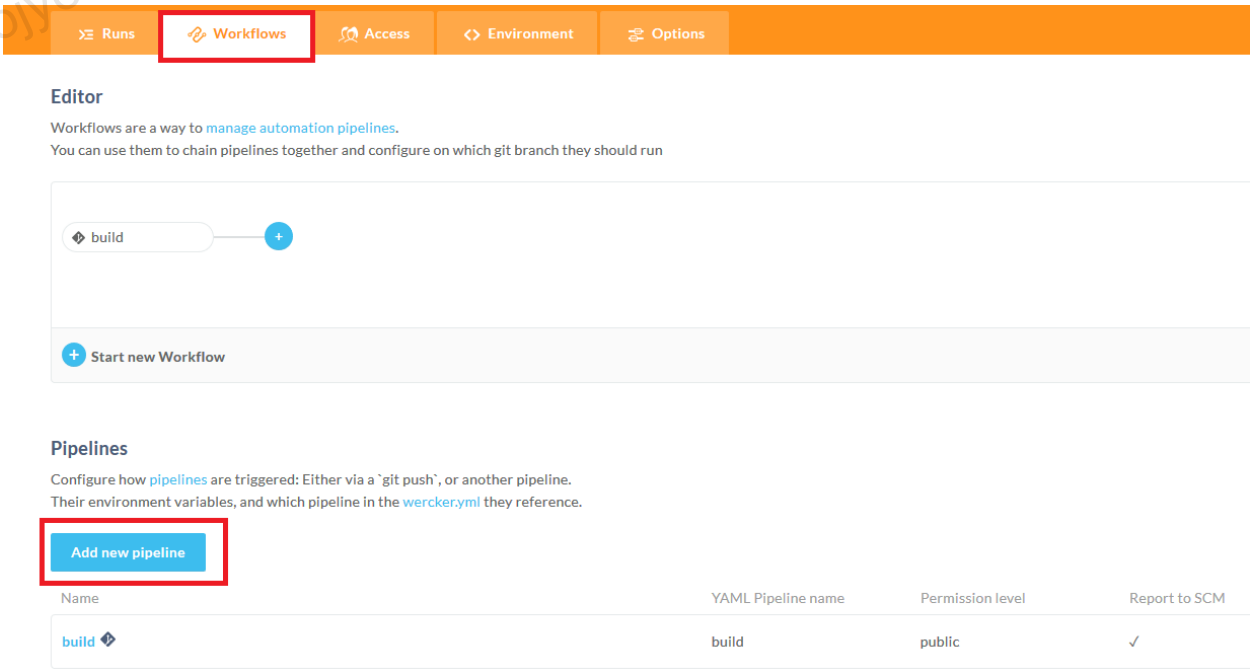


10. You are now ready to trigger the build. You already have a `wercker.yml`.

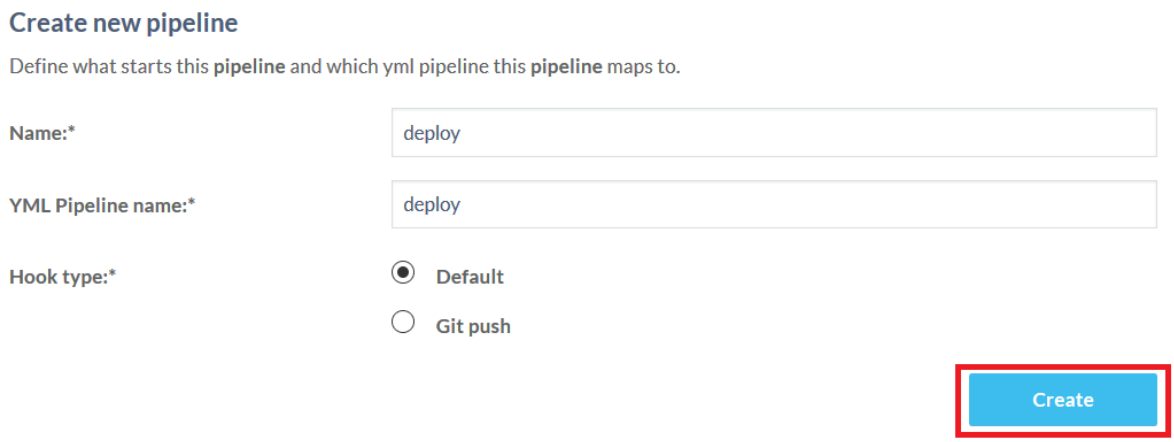


11. Once the build starts, the build should be complete in stages as given below. This is the default pipeline.

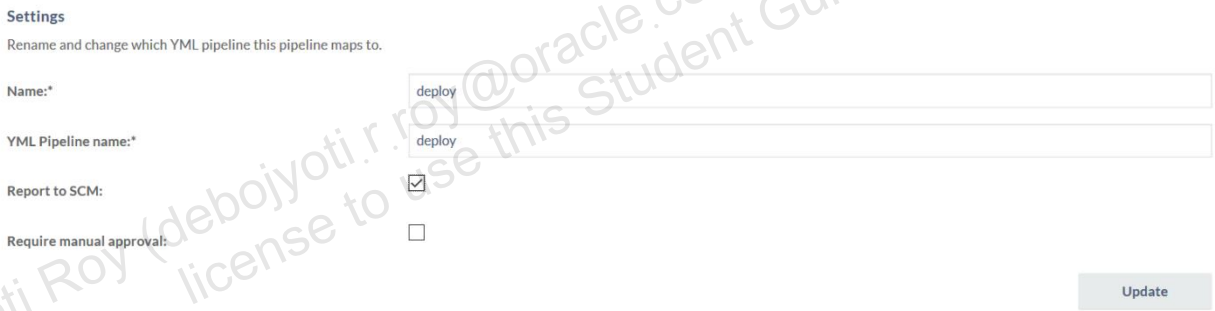
12. You can choose to add a new pipeline (in our case the *deploy* pipeline) to the workflow. Click on **Workflows** tab and choose **Add new pipeline**



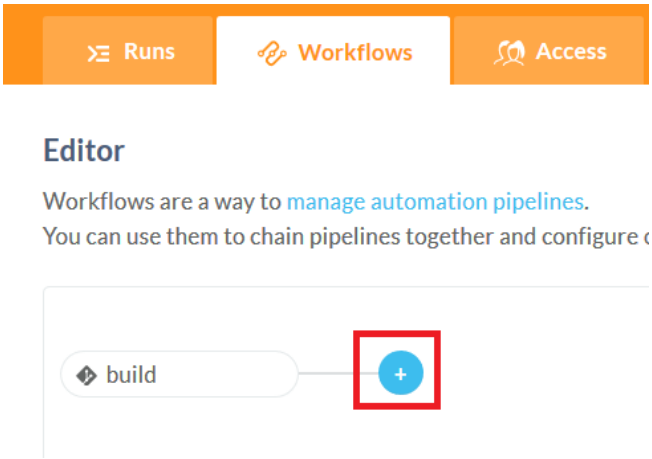
13. Enter the pipeline name - **deploy** and choose **default** as hook type. Click **Create**.

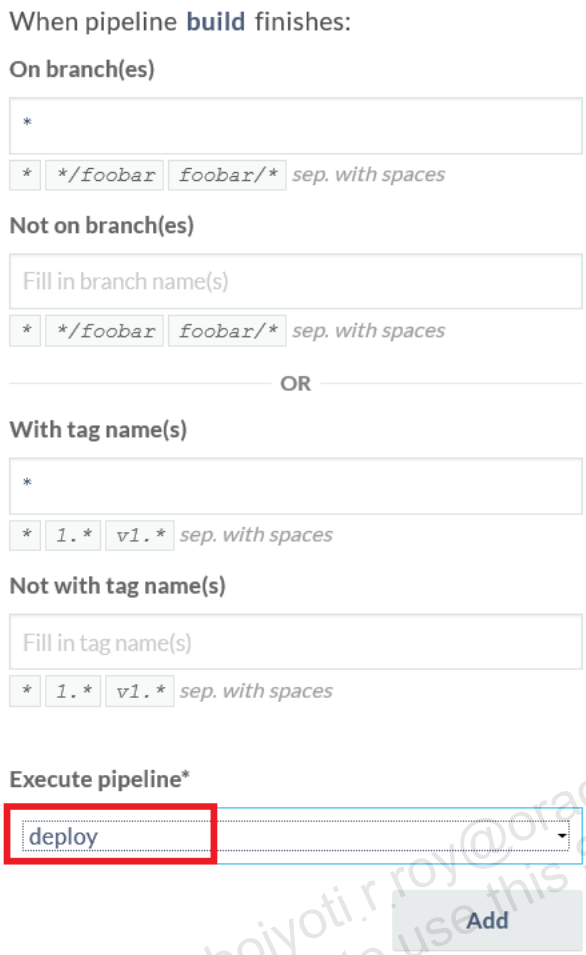


14. Upon creation, you will be in **Environment Variables** and **Settings** page.
    This pipeline does not need any environment variables to be set.
    Under **Settings**, you can choose to report to SCM (GIT) or choose manual approval. Click **Update**.

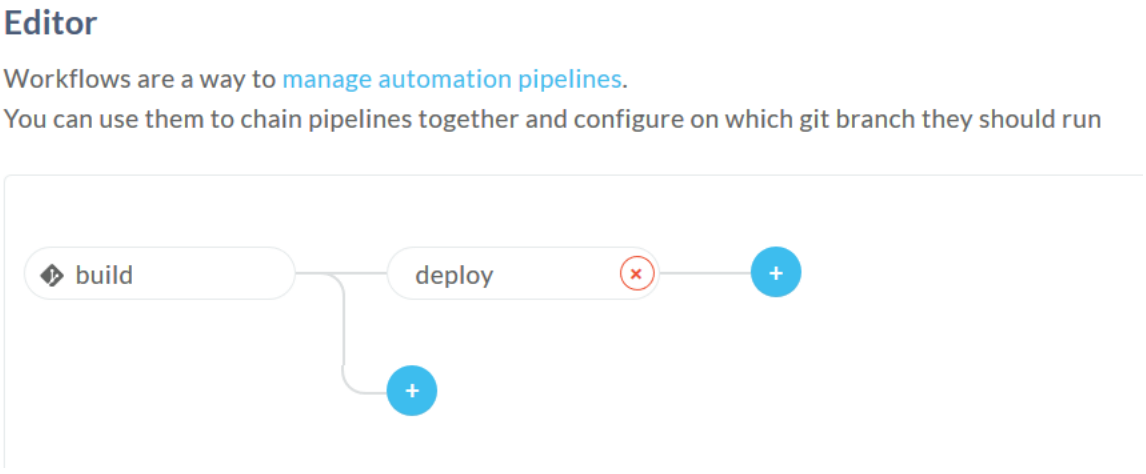


15. Now that the pipeline is added, we will associate the same with the default build pipeline. Click on **Workflows** tab to add the deploy pipeline *(You may need to refresh the Workflows tab to be able to perform this step)*

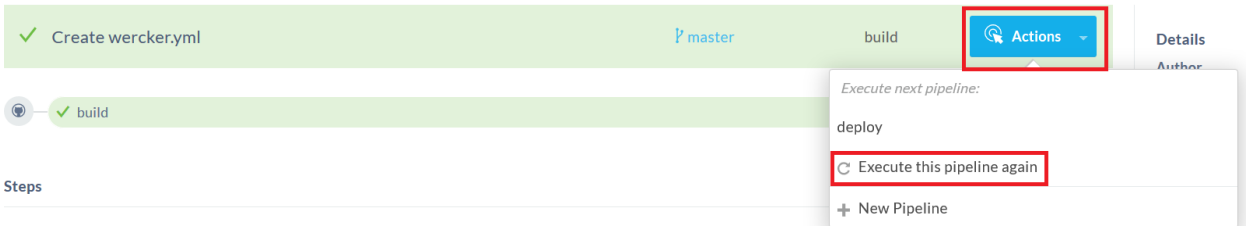




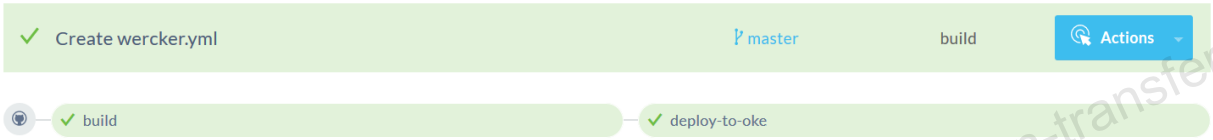
16. Deploy pipeline is added to the workflow. Once the build pipeline completes execution, deploy pipeline will start executing.



17. Now trigger the build again. Switch to the **Runs** tab, click on the successful build. Choose **Actions > Execute this pipeline again**.



18. You will notice that both `build` and `deploy` pipelines are successfully executed in sequence.



You were successful in adding a pipeline to the default and triggering the same.

Practices for Lesson 6: Deployment in Kubernetes

Practices for Lesson 6: Overview

Overview

In this lesson, you will learn how to deploy a container in Oracle Kubernetes Engine (OKE). You also learn how to leverage Wercker and YAML to implement this CD automation.

Assumptions

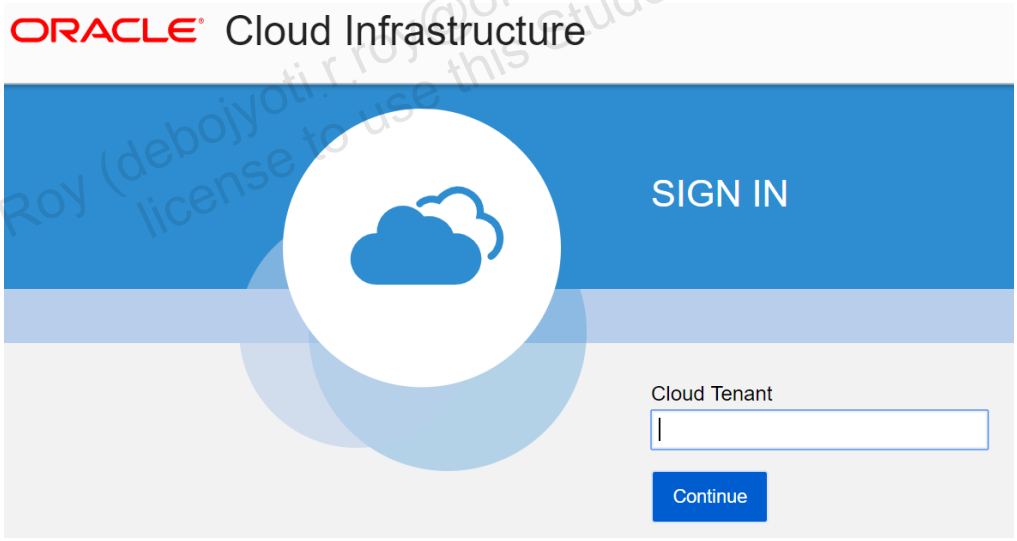
The following are required to set up your pipeline:

- ➔ Cloud account with access to Container Pipelines (app.wercker.com)
- ➔ OCI Cloud Account and access to OKE Cluster
- ➔ GIT account with commit permissions
- ➔ Linux Machine

Step 1 - Configuring Prerequisites of Oracle Container Service for Kubernetes

1. Sign in to **Oracle Cloud Infrastructure**.

In a browser on your local machine, enter the **URL** provided to you (for example, `https://console.us-ashburn-1.oraclecloud.com/`).



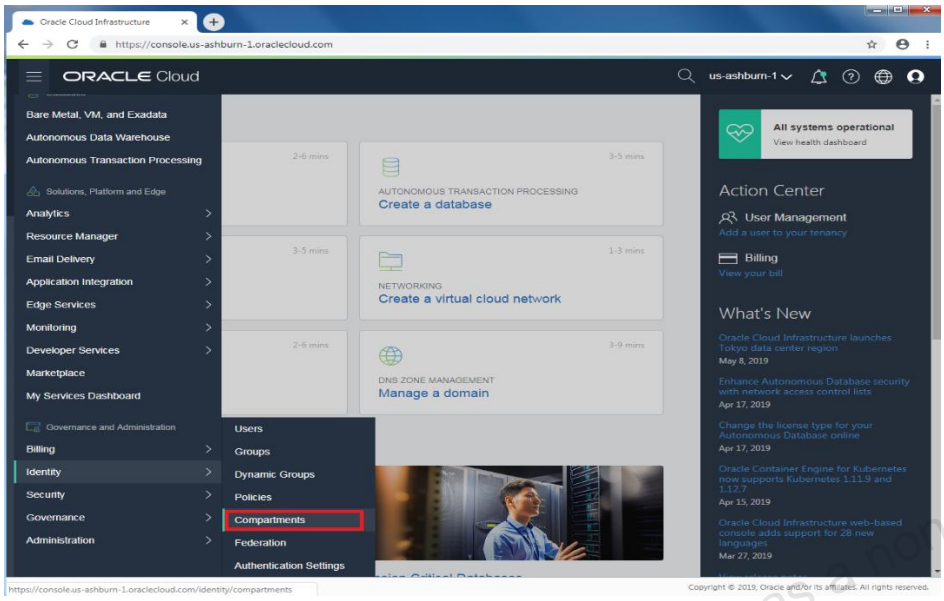
Specify a tenant in that is assigned to you. Details are provided by the instructor.

Enter username and password that is assigned to you. Details are provided by the instructor.

You will now gain access to OCI Console.

2. Within your tenancy, there must be a compartment to contain the resources.

On the left, you will see the **Navigation** menu; navigate to **Identity** and to **Compartments**.



Browse through the Compartments List to access your Compartment. Note that you will be given access to a specific compartment along with the required access privileges.

3. Configuring the Oracle Cloud Infrastructure **CLI.**

Copy public-private key pair provided by your instructor :

1) Copy the Public key provided to you by your instructor.
2) Connect to the Linux machine provided to you (details shared by your instructor)
3) Navigate to *Applications -> Accessories -> Text Editor.*
4) Paste the Public key and name the file "`oci_api_key_public.pem`".
5) Save the file in the location `/home/oracle/LabFiles`.
6) Similarly copy the Private Key provided by the instructor and name the file "`oci_api_key.pem`".
7) Save the file in the location `/home/oracle/LabFiles` folder.

Note: Ensure there are no leading and trailing spaces when you are copying the keys to the text editor.

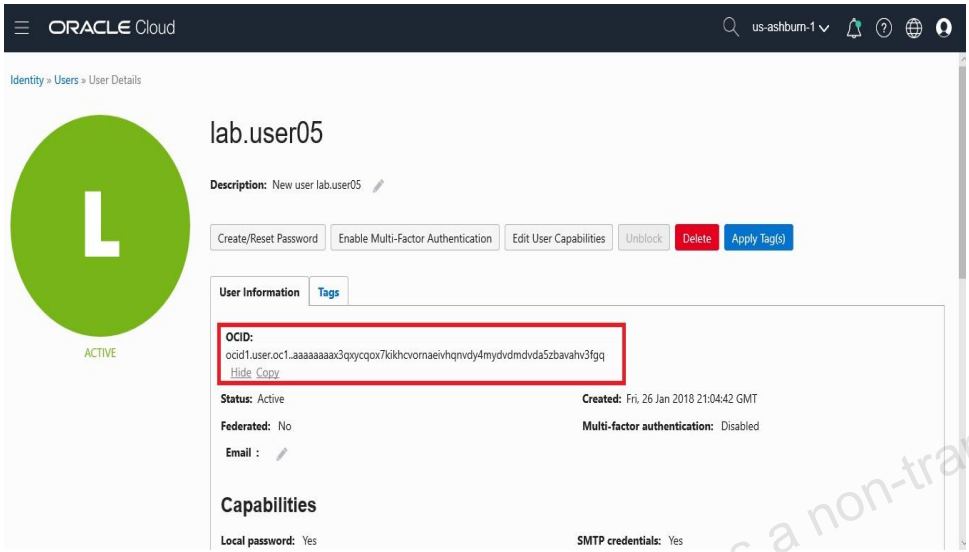
Open a terminal window and run the following command: oci setup config

```
[oracle@edvmr1p0 ~]$ oci setup config
    This command provides a walkthrough of creating a valid CLI config file.
```

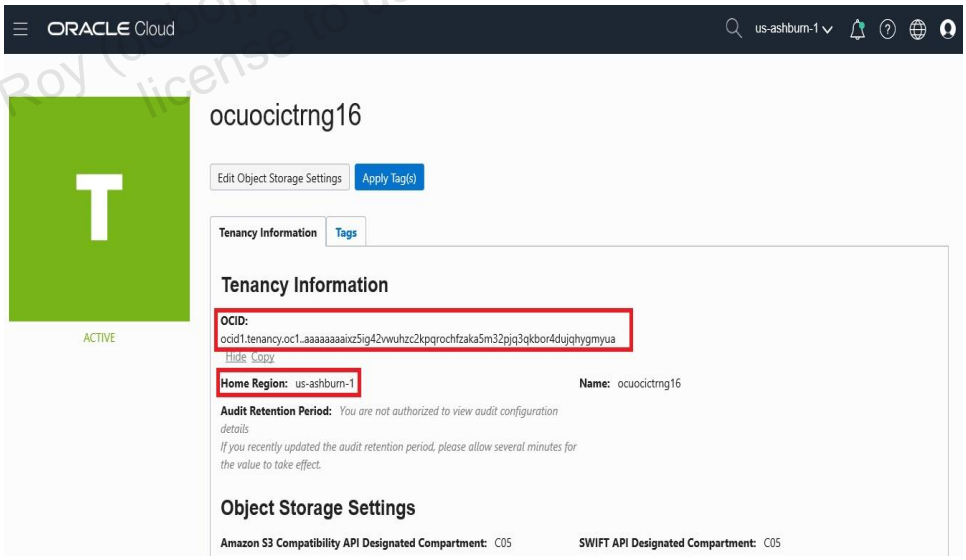
Accept the default location for config file by pressing Enter. It prompts to enter User OCID, Tenancy OCID, and Region name.

Navigate to the OCI Console:

1) Get the User OCID from **User Details** page (Identity -> Users -> Click on your User Name)



2) Get the Tenancy OCID and Region from **Tenancy Details** page by navigating through the Navigation menu, Administration > Tenancy Details, respectively, as shown below.



Enter the User OCID, Tenancy OCID, and Region that you copied in the previous step, in the OCI setup console screen as shown below:

```
    Region:

        https://docs.cloud.oracle.com/Content/General/Concepts/regions.htm

    General config documentation:

        https://docs.cloud.oracle.com/Content/API/Concepts/sdkconfig.htm


Enter a location for your config [/home/oracle/.oci/config]:
Enter a user OCID: ocid1.user.oc1..aaaaaaaa7s433f4nocuajmbns4v5noqhxb7wxeo5wa4ng
f7iffgv5wkrrjja
Enter a tenancy OCID:
```

```
Enter a location for your config [/home/oracle/.oci/config]:
Enter a user OCID: ocid1.user.oc1..aaaaaaaa7s433f4nocuajmbns4v5noqhxb7wxeo5wa4ng
f7iffgv5wkrrjja
Enter a tenancy OCID: ocid1.tenancy.oc1..aaaaaaaa4jlkvlylxuwrnpcgybni5e34gqxzf6h
ywqtjcrmwdjy7coihqcga
Enter a region (e.g. ap-seoul-1, ap-tokyo-1, ca-toronto-1, eu-frankfurt-1, uk-lo
ndon-1, us-ashburn-1, us-gov-ashburn-1, us-gov-chicago-1, us-gov-phoenix-1, us-l
angley-1, us-luke-1, us-phoenix-1): us-ashburn-1
```

It prompts to ask whether to generate the RSA key (API Signing key pair):

We are not generating a new key here as we already have one existing. You can specify **n** and give the path as `/home/oracle/LabFiles/oci_api_key.pem`.

```
Do you want to generate a new RSA key pair? (If you decline you will be asked to
 supply the path to an existing key.) [Y/n]: n
Enter the location of your private key file: /home/oracle/LabFiles/oci_api_key.p
em
Fingerprint: 9e:42:46:e9:92:1e:b0:53:60:01:cd:de:bd:88:31:56
Config written to /home/oracle/.oci/config


    If you haven't already uploaded your public key through the console,
    follow the instructions on the page linked below in the section 'How to
    upload the public key':

        https://docs.cloud.oracle.com/Content/API/Concepts/apisigningkey.htm#How
2
```

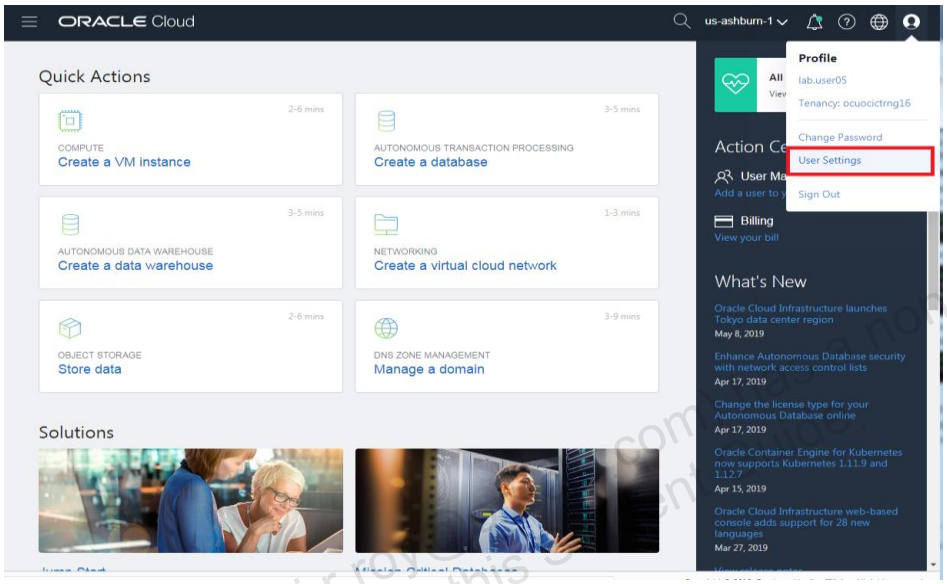
The config is now written to `/home/oracle/.oci/config`. The Public Key now needs to be uploaded to OCI Console.

4. Upload the Public Key of the API signing key pair.

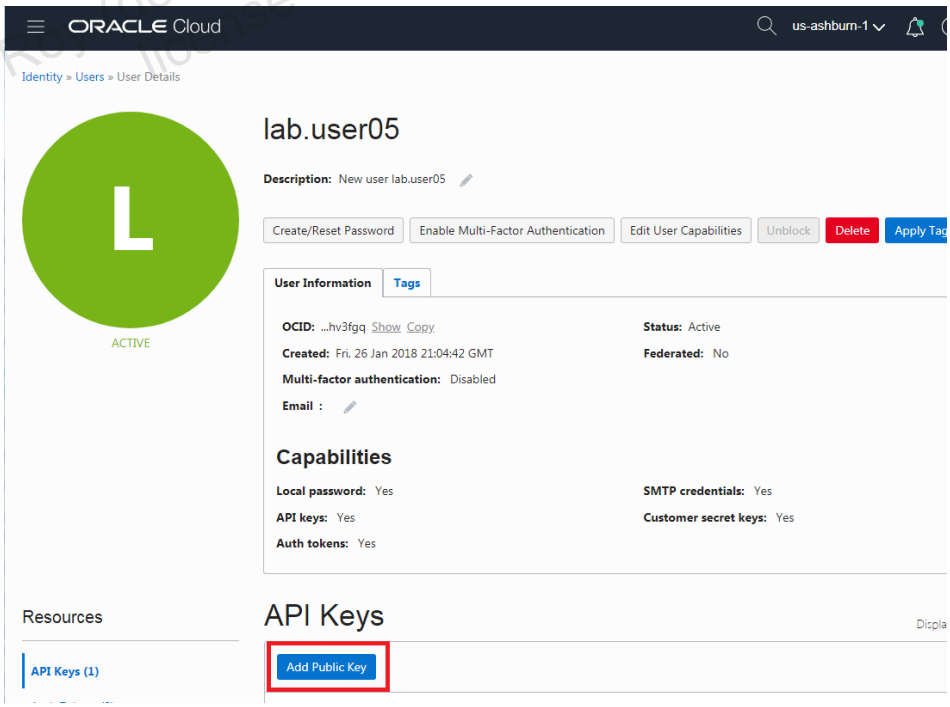
Use the below command and copy the contents of the Public key to the clipboard using **cat** (you need to paste the value into the console in the next step).

```
cat /home/oracle/LabFiles/oci_api_key_public.pem
```

In the top-right corner of the console, open the **User** menu and then click **User Settings**.



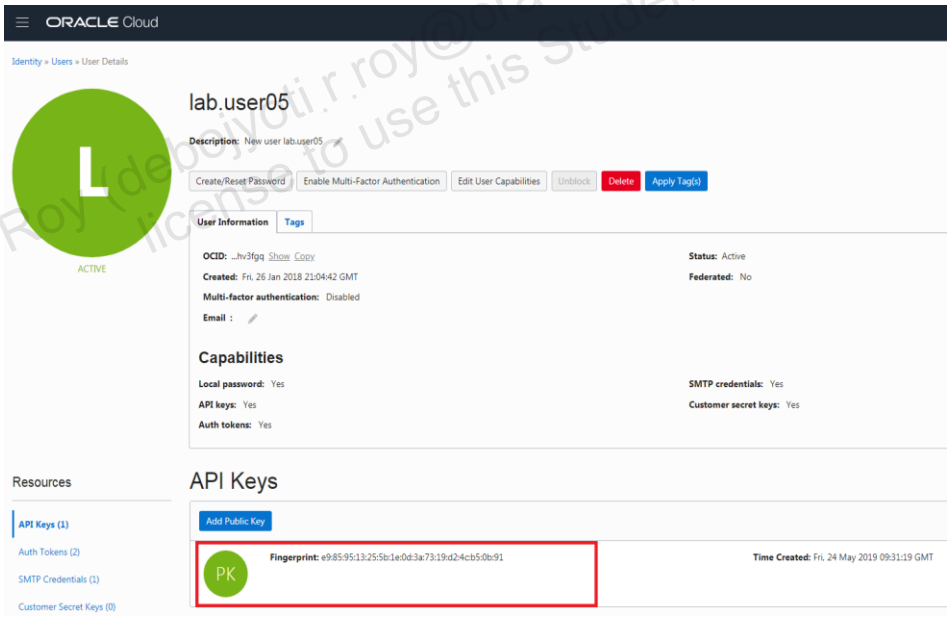
Click **Add Public Key**.



Paste the **Public keys** value into the window and click **Add**.

Note: Make sure there are no leading or trailing spaces after you paste the Public key.

When the key is uploaded, its **Fingerprint** is displayed as shown below.



5. Verify the OCI CLI Connectivity.

Check the OCI CLI connectivity: `oci os ns get`

It displays your configured **Tenancy Name.** This ensures that CLI has been configured successfully. Ignore the warnings if any.

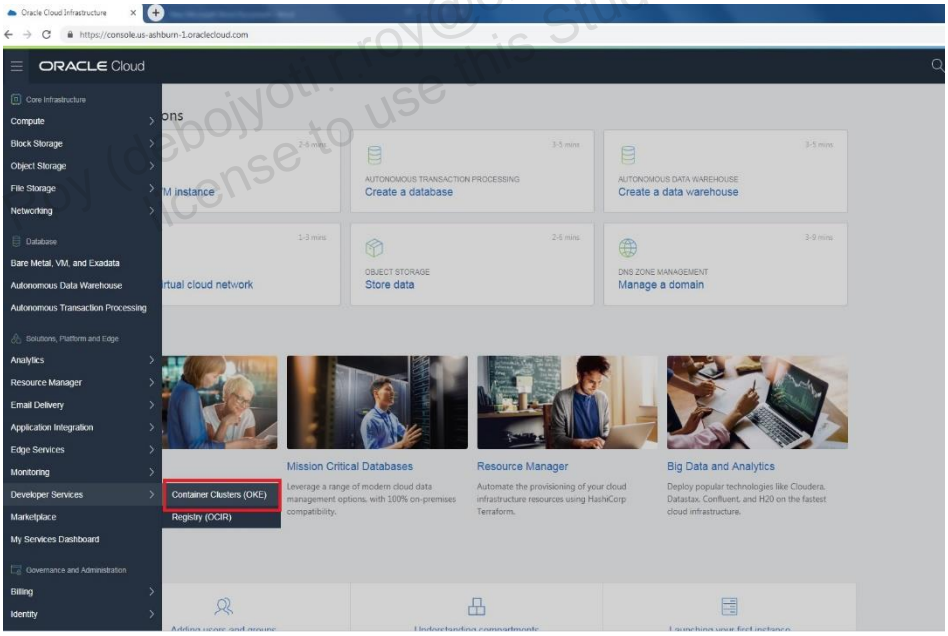
```
[oracle@edvmr1p0 ~]$ oci os ns get
WARNING: Permissions on /home/oracle/LabFiles/oci_api_key.pem are too open.
To fix this please try executing the following command:
oci setup repair-file-permissions --file /home/oracle/LabFiles/oci_api_key.pem
Alternatively to hide this warning, you may set the environment variable, OCI_CLI_SUPPRESS_FILE
_PERMISSIONS_WARNING:
export OCI_CLI_SUPPRESS_FILE_PERMISSIONS_WARNING=True

{
  "data": "ocuocictrng7"
}
```

6. Close the terminal.

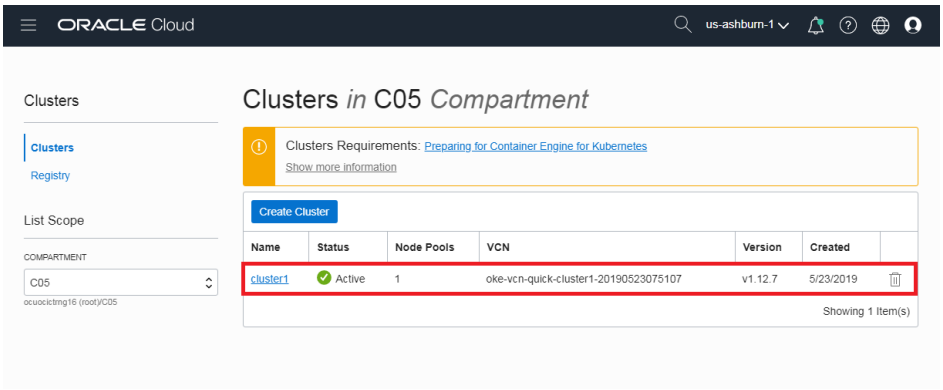
Step 2: Installing and Configuring kubeconfig on Oracle Cloud Infrastructure

1. Download the `kubeconfig` file.

Navigate to the OCI console. On the left, you will see the **Navigation** menu; navigate to **Developer Services** and click **Container Clusters(OKE).**

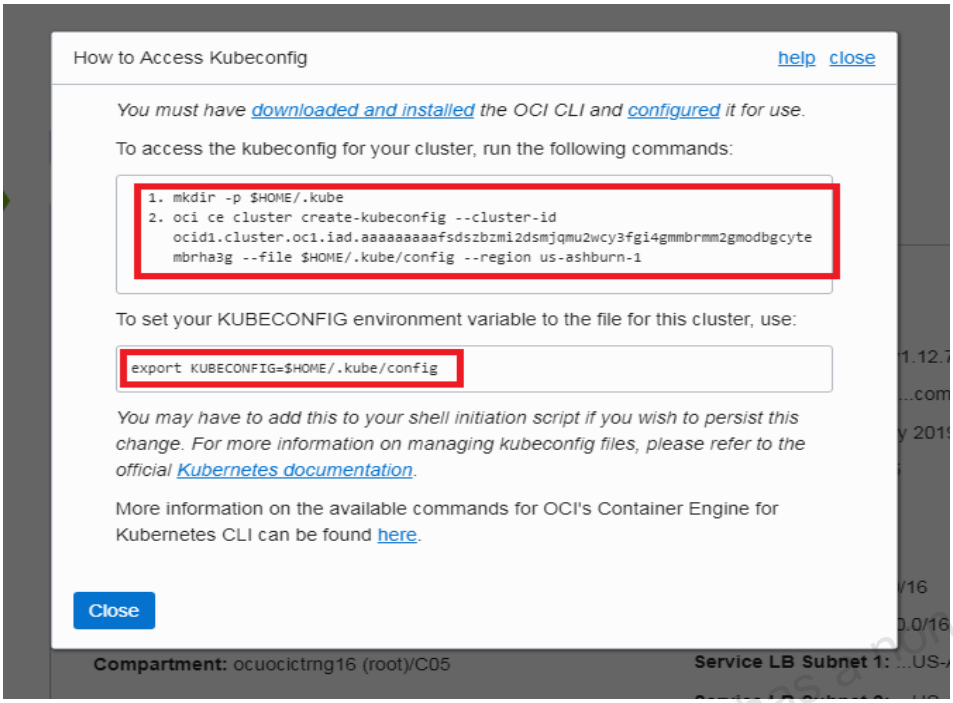


On the Cluster List page, click the name of the cluster in the root compartment.



The Cluster page shows the details of the cluster. Click **Access Kubeconfig** dialog box.

It provides the commands to access the `kubeconfig` for the respective cluster as shown below. Copy the second command.



On your Linux machine, open a terminal window, create a directory to contain the `kubeconfig` file,

```
mkdir -p $HOME/.kube
```

Run the Oracle Cloud Infrastructure CLI command that you copied in the previous step.

Note: OCID of the cluster differs from cluster to cluster. For convenience, the command in the How to Access Kubeconfig dialog box already includes the respective cluster's OCID.

```
[oracle@edvmr1p0 ~]$ mkdir -p $HOME/.kube
[oracle@edvmr1p0 ~]$ oci ce cluster create-kubeconfig --cluster-id ocid1.cluster
.oc1.iad.aaaaaaaaaeytenztmeytiolcmnqtkyjzg4ygcztcheygkmjugc3gmnlcmvsw --file $HO
ME/.kube/config --region us-ashburn-1
WARNING: Permissions on /home/oracle/LabFiles/oci_api_key.pem are too open.
To fix this please try executing the following command:
oci setup repair-file-permissions --file /home/oracle/LabFiles/oci_api_key.pem
Alternatively to hide this warning, you may set the environment variable, OCI_CL
I_SUPPRESS_FILE_PERMISSIONS_WARNING:
export OCI_CLI_SUPPRESS_FILE_PERMISSIONS_WARNING=True

New config written to the Kubeconfig file /home/oracle/.kube/config
```

Set the value of the KUBECONFIG environment variable to the name and location of the **`kubeconfig`** file

```
[oracle@edvmr1p0 ~]$ export KUBECONFIG=$HOME/.kube/config
[oracle@edvmr1p0 ~]$
```

Step 3: Creating Service Account for deployment

1. Confirm that **kubectl** is installed by running command.

   ```
   kubectl version
   ```

   ```
   Client Version: version.Info{Major:"1", Minor:"14", GitVersion:"v1.14.3", GitCommit:"5e53fd6bc17c0dec8434817e
   69b04a25d8ae0ff0", GitTreeState:"clean", BuildDate:"2019-06-06T01:44:30Z", GoVersion:"go1.12.5", Compiler:"gc
   ", Platform:"linux/amd64"}
   Server Version: version.Info{Major:"1", Minor:"12+", GitVersion:"v1.12.7-3+1a4d39a9763d23", GitCommit:"1a4d39
   a9763d23548d16e262d6dbe0fb2198a8ce", GitTreeState:"clean", BuildDate:"2019-03-27T01:47:38Z", GoVersion:"go1.1
   0.8", Compiler:"gc", Platform:"linux/amd64"}
   ```

2. Verify that you can use **kubectl** to connect to the new cluster that has been created. In a terminal window, enter the following command.

   ```
   kubectl get nodes
   ```

3. You can see the details of the nodes running in the cluster, as shown below.

   ```
   NAME       STATUS   ROLES    AGE   VERSION
   10.0.0.2   Ready    node     24d   v1.12.7
   10.0.1.2   Ready    node     24d   v1.12.7
   10.0.2.2   Ready    node_    24d   v1.12.7
   ```

4. Inorder to proceed, we need a **Service Account** with full privileges for the wercker CD to create the pipeline and be able to deploy to Kubernetes cluster.

   Here is the yml file which will help create a service account, named **oke-admin** in **kube-system** namespace with **ClusterRoleBinding.**

   In the Linux machine – use vi editor and create a file in current directory called **sa.yml** as per the below specification.

   Note - Pay special attention to indentation while working with YAML.

   You can verify your yaml file using YAML Validator - https://codebeautify.org/yaml-validator

   For your reference, the yml file is posted in the Linux machine @ `/home/oracle/LabFiles/sa.yml`

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: oke-admin
  namespace: kube-system
---
```

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: oke-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: oke-admin
    namespace: kube-system
```

5. Now execute the yml file using the below command. This will create the service account.
   `kubectl create –f sa.yml`
6. Use the below command to verify if the desired service account is created in the `kube-system` namespace -
   `kubectl get sa –n kube-system`
   (Observe `oke-admin` account being created with all the necessary privileges for cluster deployment).

```
[oracle@edvmr1p0 ~]$ kubectl get sa -n kube-system
WARNING: Permissions on /home/oracle/LabFiles/oci_api_key.pem are too open.
To fix this please try executing the following command:
oci setup repair-file-permissions --file /home/oracle/LabFiles/oci_api_key.pem
Alternatively to hide this warning, you may set the environment variable, OCI_CL
I_SUPPRESS_FILE_PERMISSIONS_WARNING:
export OCI_CLI_SUPPRESS_FILE_PERMISSIONS_WARNING=True

NAME                   SECRETS   AGE
default                1         8d
flannel                1         8d
kube-dns               1         8d
kube-proxy             1         8d
kubernetes-dashboard   1         8d
oke-admin              1         20h
tiller                 1         8d
[oracle@edvmr1p0 ~]$
```

7. Use the `describe` command to fetch the token name of the `oke-admin` account

```
kubectl describe sa oke-admin -n kube-system
```

```
Name:                oke-admin
Namespace:           kube-system
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   oke-admin-token-k2cdf
Tokens:              oke-admin-token-k2cdf
Events:              <none>
```

Token is an internal encrypted config parameter through which you can communicate with the cluster. Make a note of the token name, which is required for us to get into the Kubernetes cluster.

8. Use the `describe secret` command to fetch the token value. Store the highlighted token value in an editor as KUBERNETES_TOKEN

```
kubectl describe secret oke-admin-token-k2cdf -n kube-system
```

Here `oke-admin-token-k2cdf` is the name of your token, fetched from the previous step.

```
Name:         oke-admin-token-k2cdf
Namespace:    kube-system
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: oke-admin
              kubernetes.io/service-account.uid: e49d9f7e-2240-11ea-9c34-0a580aedd269

Type:  kubernetes.io/service-account-token

Data
====
namespace:  11 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby
9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJrdWJlLXN5c3RlbSIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5hbWUiO
iJva2UtYWRtaW4tdG9rZW4tazJjZGYiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoib2tlLWFk
bWluIiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZXJ2aWNlLWFjY291bnQudWlkIjoiZTQ5ZDlmN2UtMjI0MC0xMWVhLTljMzQtMGE
1ODBhZWRkMjY5Iiwic3ViIjoic3lzdGVtOnNlcnZpY2VhY2NvdW50Omt1YmUtc3lzdGVtOm9rZS1hZG1pbiJ9.UaFPg7mJIWrv9mcJSSunet7Ph
plNr2J7g2GVswFKdU9S9I9L0HDN6xjTtXbECK1nq1nA3tq9TvM3gpTmDY_X2s9xZciRaqXVmRM8dKhDy6yYdhYJHlhVSdil2jtwhL3wcQfBSrh2
LbvuEdPk-MMk_Ry-ZvAwHlKUYF0OrjUmC5wJUgtCyibXULf7p4jcji4GdwAATLLCqSB0dogCg1LN7xXNLIl5IrhIvv5ZUa07APv6qQpXmsrmad1
0-2VsrLH7EWP4qFZhV7lr6feyAKR9XGsN4PMpqcbHjsH00-Fy-fpMYyQjbu-oeTlJ5RZRnDmlA5UIkfyF1laHq_4Z4Op_Ww
ca.crt:     1289 bytes
[oracle@edvmr1p0 ~]$
```

9. Next, we need the master machine for the token. Use the below command to fetch the server or the master machine details of the token.

```
kubectl config view
```

```
[oracle@edvmr1p0 ~]$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://c3dmn3bmy2g.us-ashburn-1.clusters.oci.oraclecloud.com:6443
  name: cluster-c3dmn3bmy2g
contexts:
- context:
    cluster: cluster-c3dmn3bmy2g
    user: user-c3dmn3bmy2g
  name: context-c3dmn3bmy2g
current-context: context-c3dmn3bmy2g
kind: Config
preferences: {}
```

Store the highlighted value of server in an editor as KUBERNETES_MASTER.

We would need both KUBERNETES_TOKEN and KUBERNETES_MASTER for our deployment.

Step 3: Get the GIT Ready

1. Create a repository by name `wercker-k8s`.
2. Create the file with file name: `ingress.yml.template` with below snippet.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: rest-simpleapp-ing
  annotations:
    kubernetes.io/ingress.class: 'nginx'
    ingress.kubernetes.io/add-base-url: 'true'
    ingress.kubernetes.io/rewrite-target: /
spec:
  tls:
  - secretName: tls-secret
  rules:
  - http:
      paths:
      - path: /$WERCKER_APPLICATION_OWNER_NAME/app/
        backend:
          serviceName: rest-simpleapp-svc
          servicePort: 8088
```

Commit the file once done.

10. Create a file `kubernetes-deployment.yml.template` with below snippet, **Commit** the file.

```
apiVersion: extensions/v1beta1
kind: Deployment

metadata:
  name: rest-simpleapp

  labels:
    run: rest-simpleapp
spec:
  replicas: 1
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        run: rest-simpleapp
    spec:
      containers:
      - image: nginx
        imagePullPolicy: Always
        name: rest-simpleapp
        ports:
        - containerPort: 3000
          protocol: TCP
      restartPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
  name: rest-simpleapp-svc

spec:
```

```
  selector:
    run: rest-simpleapp
  ports:
    - port: 8088
      targetPort: 3000
type: NodePort
```

11. Create a file `package.json` with below snippet, **Commit** the file.

```
{
        "name": "simple-app",
        "version": "0.0.1",
        "private": true,
        "scripts": {
          "start": "node app.js"
        },
        "dependencies": {
          "body-parser": "^1.15.0",
          "express": ">=3.11.0"
        }
}
```

12. Create a file `app.js` with below snippet, **Commit** the file.

```
var express = require('express')
          , http = require('http')
          , bodyParser = require('body-parser')


     var app = express();



     app.set('port', process.env.PORT || 3000);
     app.use(bodyParser.json()); // for parsing
application/json



     http.createServer(app).listen(app.get('port'), function
() {
```

```
          console.log('Express server listening on port ' +
app.get('port'));
      });



      app.post('/simple-app', function (req, res) {
          console.log("Received Post request - will handle it
now ");
          handlePost(req, res);
      });



      app.get('/about', function (req, res) {
          res.writeHead(200, { 'Content-Type': 'text/html' });
          res.write("About Simple App REST API V3.3:");
          res.write("/simple-app - POST  ");
          res.write("incoming headers" +
JSON.stringify(req.headers));
          res.end();
      });



      app.get('/stuff', function (req, res) {
          res.writeHead(200, { 'Content-Type':
'application/json' });
          // from https://developer.mozilla.org/en-
US/docs/Learn/JavaScript/Objects/JSON
          var stuff = {
              "squadName": "Raymond Job en Frank",
              "homeTown": "Metro City",
              "formed": 2016,
              "secretBase": "Super tower",
              "active": true,
              "members": [
                {
                  "name": "Molecule Man",
                  "age": 29,
                  "secretIdentity": "Dan Jukes",
                  "powers": [
```

```
                    "Radiation resistance",
                    "Turning tiny",
                    "Radiation blast"
                  ]
                },
                {
                  "name": "Madame Uppercut",
                  "age": 39,
                  "secretIdentity": "Jane Wilson",
                  "powers": [
                    "Million tonne punch",
                    "Damage resistance",
                    "Superhuman reflexes"
                  ]
                },
                {
                  "name": "Eternal Flame",
                  "age": 1000000,
                  "secretIdentity": "Unknown",
                  "powers": [
                    "Immortality",
                    "Heat Immunity",
                    "Inferno",
                    "Teleportation",
                    "Interdimensional travel"
                  ]
                }
              ]
            }
        res.end(JSON.stringify(stuff));
    });

    handlePost =
        function (req, res) {
            var input = req.body;
            var result = input;
            result.summary = "result from simple-app -
changed version";
```

```
            res.writeHead(200, { 'Content-Type':
'application/json' });
            res.end(JSON.stringify(result));
        }
```

13. Finally create the core file - `wercker.yml` and **commit** the file.

Note - Pay special attention to indentation while working with YAML.

You can verify your yaml file using YAML Validator - https://codebeautify.org/yaml-validator

For your reference, the yml file is posted in the Linux machine @ `/home/oracle/LabFiles/wercker_lab6.yml`

```
box: "node:6.10"
build:
  steps:
    -
      script:
        code: "npm install"
        name: "A step that executes `npm install` command"
    -
      script:
        code: "openssl req -x509 -nodes -days 365 -newkey rsa:2048
-keyout tls.key -out tls.crt -subj \"/CN=nginxsvc/O=nginxsvc\""
        name: "Create TLS key and cert"
deploy-to-oke:
  box:
    cmd: /bin/sh
    id: alpine
  steps:
    - bash-template
    -
      script:
        code: "cat kubernetes-deployment.yml"
        name: "Visualise Kubernetes config"
    -
      kubectl:
        command: "create namespace $WERCKER_APPLICATION_OWNER_NAME"
        insecure-skip-tls-verify: true
        name: "create namespace"
```

```
        server: $KUBERNETES_MASTER
        token: $KUBERNETES_TOKEN
    -
      kubectl:
        command: "create secret docker-registry wrelease --docker-
server=wcr.io --docker-username=$WERCKER_APPLICATION_OWNER_NAME --
docker-password=$KUBERNETES_TOKEN --docker-
email=${WERCKER_APPLICATION_OWNER_NAME}@mail.com --
namespace=$WERCKER_APPLICATION_OWNER_NAME"
        insecure-skip-tls-verify: true
        name: "create OCR secret"
        server: $KUBERNETES_MASTER
        token: $KUBERNETES_TOKEN
    -
      kubectl:
        command: "create secret tls tls-secret --key
$WERCKER_ROOT/tls.key --cert $WERCKER_ROOT/tls.crt --
namespace=$WERCKER_APPLICATION_OWNER_NAME"
        insecure-skip-tls-verify: true
        name: "create TLS secret"
        server: $KUBERNETES_MASTER
        token: $KUBERNETES_TOKEN
    -
      kubectl:
        command: "create -f $WERCKER_ROOT/kubernetes-deployment.yml
--namespace=$WERCKER_APPLICATION_OWNER_NAME"
        insecure-skip-tls-verify: true
        name: "create deployment"
        server: $KUBERNETES_MASTER
        token: $KUBERNETES_TOKEN
    -
      script:
        code: "cat ingress.yml"
        name: "Visualise Ingress config"
    -
      kubectl:
        command: "create -f $WERCKER_ROOT/ingress.yml --
namespace=$WERCKER_APPLICATION_OWNER_NAME"
        insecure-skip-tls-verify: true
```

```
        name: "create Ingress configuration"
        server: $KUBERNETES_MASTER
        token: $KUBERNETES_TOKEN
    -
      script:
        code: "cat ingress.yml"
        name: "Visualise Ingress config"
functional-test:
  steps:
    -
      script:
        code: |-
            mkdir -p "/pipeline"
            node $WERCKER_ROOT/app.js &
            sleep 5
            SIMPLE_APP_URL=http://localhost:3000/simple-app
            echo 'Microservice URL=' $SIMPLE_APP_URL
            if curl -X POST -H "Content-Type: application/json" -X
POST -d '{"simple-param":"John","param2":"Doe"}' $SIMPLE_APP_URL |
grep "simple"
            then
                # if the keyword is in the conent
                echo "Test passed"
            else
                echo "Test failed"
                exit -1
            fi
            sleep 1
```

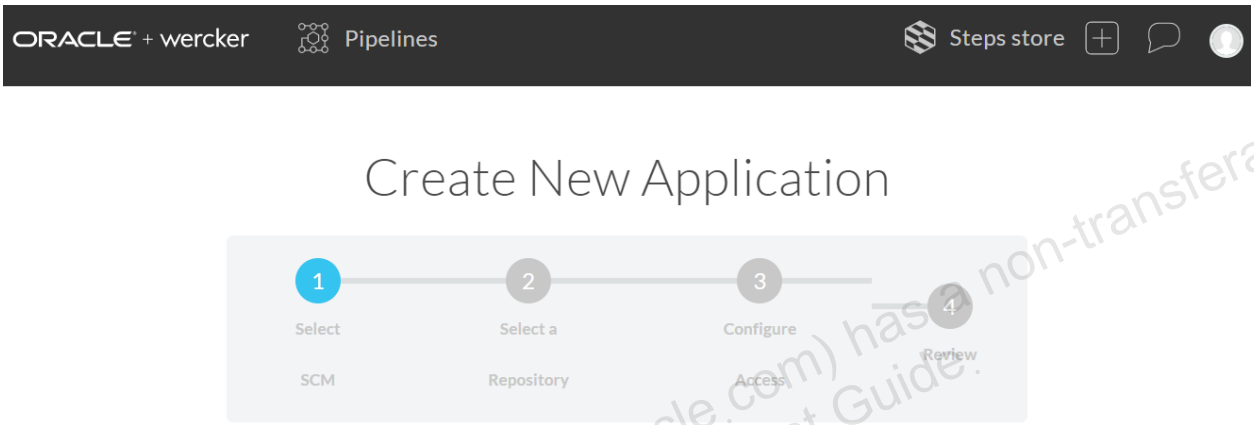
14. Finally , your repository should have the following files -

app.js	Create app.js
ingress.yml.template	Create ingress.yml.template
kubernetes-deployment.yml.template	Create kubernetes-deployment.yml.template
package.json	Create package.json
wercker.yml	Create wercker.yml

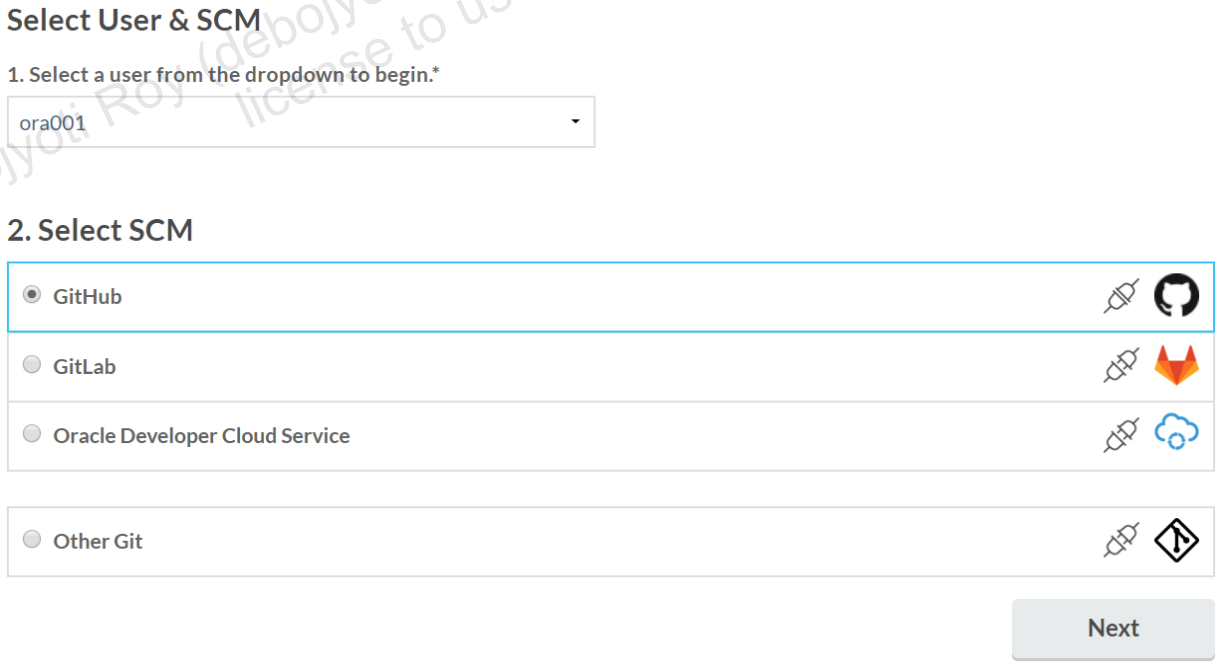
Step 4: Set up the CI/CD pipeline by creating an application

1. Access Container Pipelines application page, and click **Add application**

2. Complete the wizard to create application.

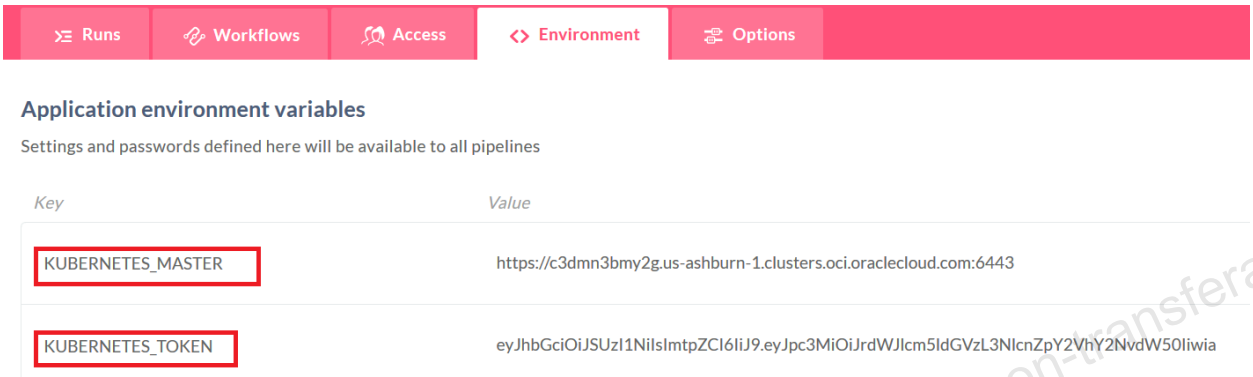


3. Authorize **GitHub** with username and password for Container Pipeline. Click **Next**.

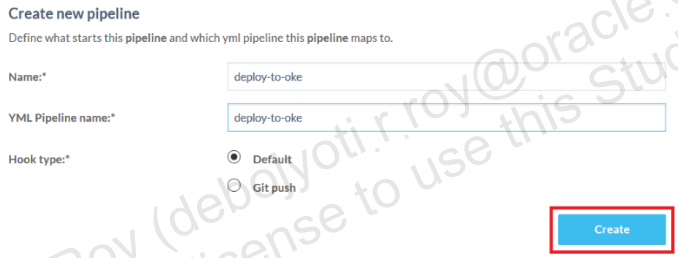


4. Choose the repository - `wercker-k8s` (created in previous step)
5. Configure access without keys. Click **Next**.
6. Review and click **Create** to create application.

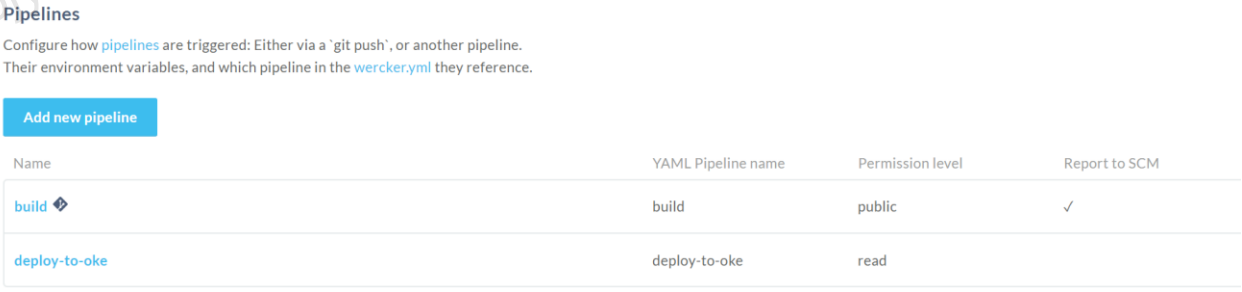
7. Before your build the application, create the environment variables required for this application. Click the **Environment** tab in this page below.
8. If you have observed the `wercker.yml` file, there are two environmental variables used - KUBERNETES_MASTER and KUBERNETES_TOKEN. Create two environmental variables as below. Remember you had fetched the values for these variables in the earlier step.



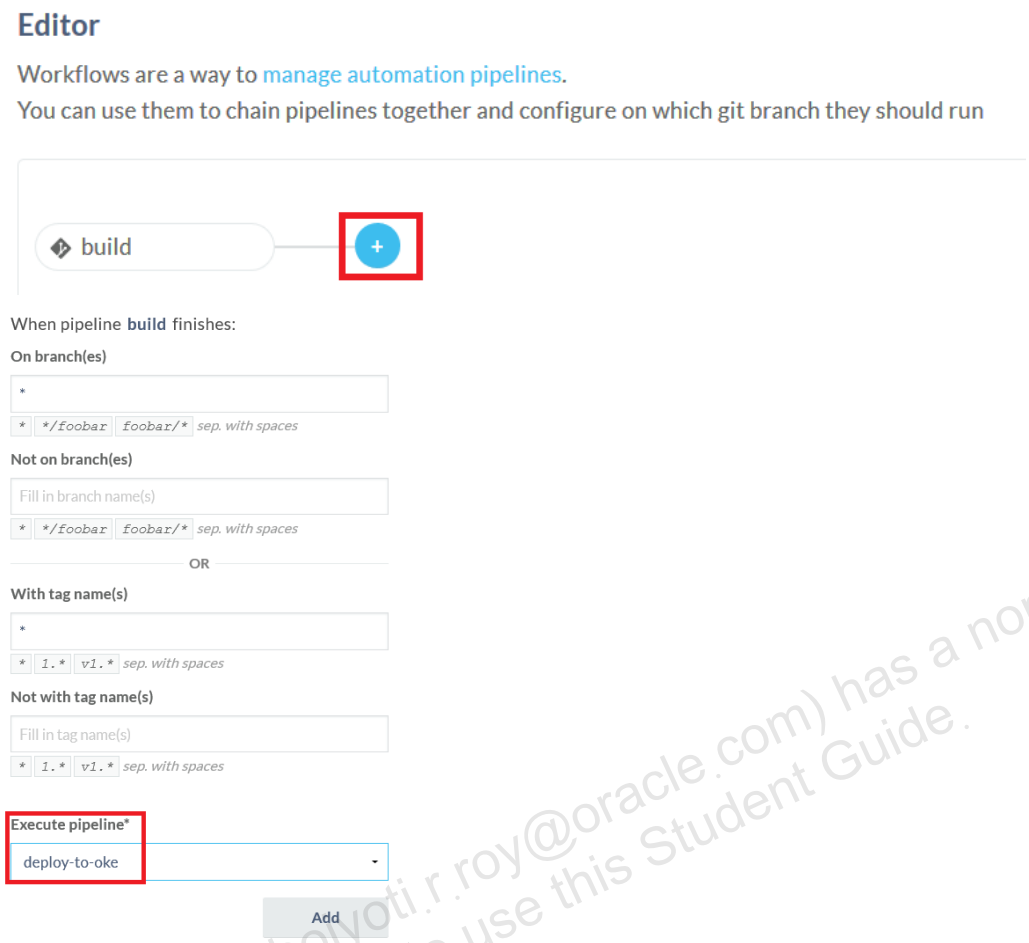
9. Now add a new pipeline (in our case the *deploy-to-oke* pipeline) to the workflow. Click on **Workflows** tab and choose **Add new pipeline**
10. Enter the pipeline name – **deploy-to-oke** and choose **default** as hook type. Click **Create**.



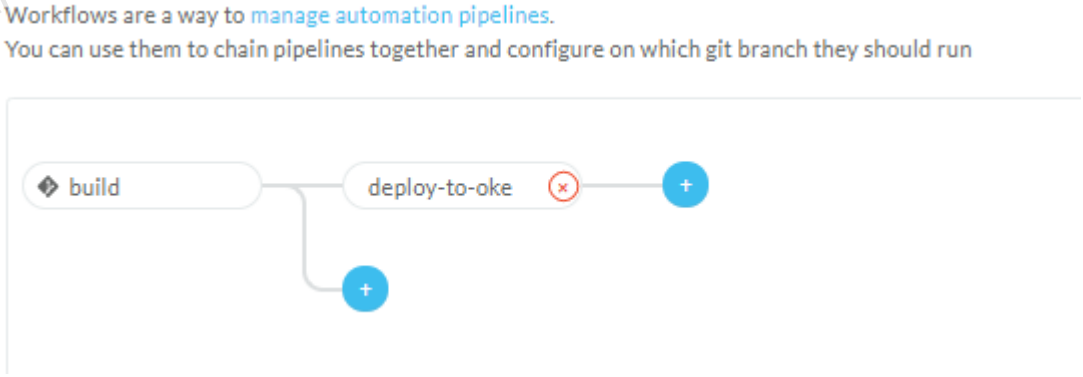
11. Confirm that the pipeline is created.



12. Now that the pipeline is added, we will associate the same with the default build pipeline. Click on **Workflows** tab to add the deploy-to-oke pipeline *(You may need to refresh the Workflows tab to be able to perform this step)*

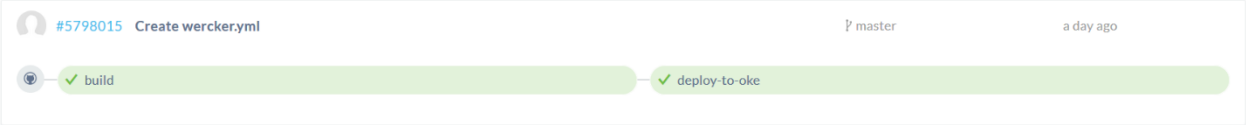


13. Verify that the workflow is created as below.



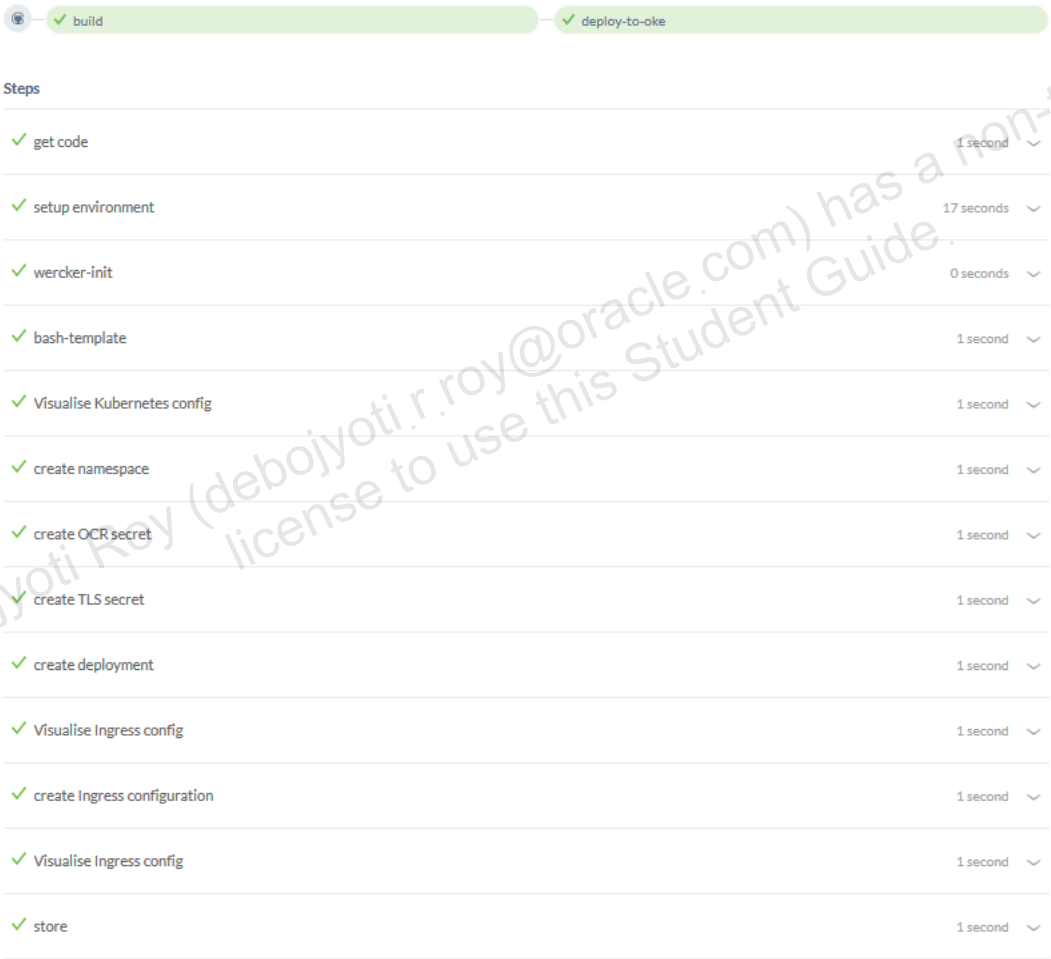
14. You are now ready to trigger the build. You already have a `wercker.yml`.

15. Once the build starts, the build should be completed in stages and both the pipelines gets executed in sequence.



Upon execution of deployment pipeline, the objects are deployed in OKE cluster under a specific node.

Upon successful completion of the pipeline, it should trigger the activities as per the following script steps in `wercker.yml`:



16. To verify the deployment, switch to your Linux machine. First verify if the namespace is created using the below command -

```
kubectl get ns
```

```
NAME          STATUS   AGE
default       Active   8d
kube-public   Active   8d
kube-system   Active   8d
ora005        Active   21h
[oracle@edvmr1p0 ~]$
```

You will see that your namespace (here `ora005`) is created.

17. Fetch the pods in your namespace –

```
kubectl get pods -n ora005
```

```
NAME                                READY   STATUS    RESTARTS   AGE
rest-simpleapp-56665dd964-2phfb   1/1     Running   0          66m
```

You will see that the `rest-simple` pod is running

18. Verify if the rest service is created.

```
kubectl get services -n ora011
```

```
NAME                TYPE       CLUSTER-IP    EXTERNAL-IP   PORT(S)
rest-simpleapp-svc  NodePort   10.96.23.74   <none>        8088:31567/TCP
```

Congratulations!! You were successfully with your deployment.

From this point on, it is very simple to further evolve the application. Modify the code, test locally, commit, and push to Git. The changed application is automatically built and deployed to the managed Kubernetes cluster via container pipeline with OKE.

Practices for Lesson 7: Integrations

Practices for Lesson 7: Overview

Overview

In this lesson, you will learn how to connect to Slack by using Oracle Container Pipeline.

Assumptions

The following accounts are required for this practice:

- Cloud account with access to Container Pipelines (app.wercker.com)
- GIT account with commit permissions
- Slack account

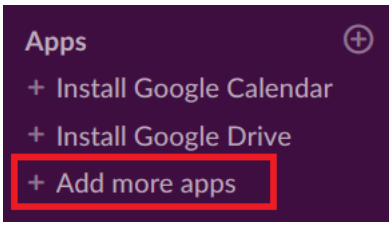
1. Step 1: Prepare for SLACK

- Log in to your Slack account.
- If you do not have a Slack account, create one.
- Go to `https://slack.com/create#email`.
- Enter your email address, and then click **Next**. Check your email for a six-digit confirmation code.
- Enter your code, name your workspace, and click **Next**.
- Create a new channel for your workspace. Channels organize conversations about any topic, such as a project your team is working on.
- Add coworkers' email addresses if you're ready to invite others. Otherwise, select **skip for now**.
- Click **See your channel in Slack** to visit your workspace.
- Select **Finish Signing Up** at the top of the screen. Enter your name and password, and then click **Next**.
- Review your workspace name and URL, and then click **Finish** to save.

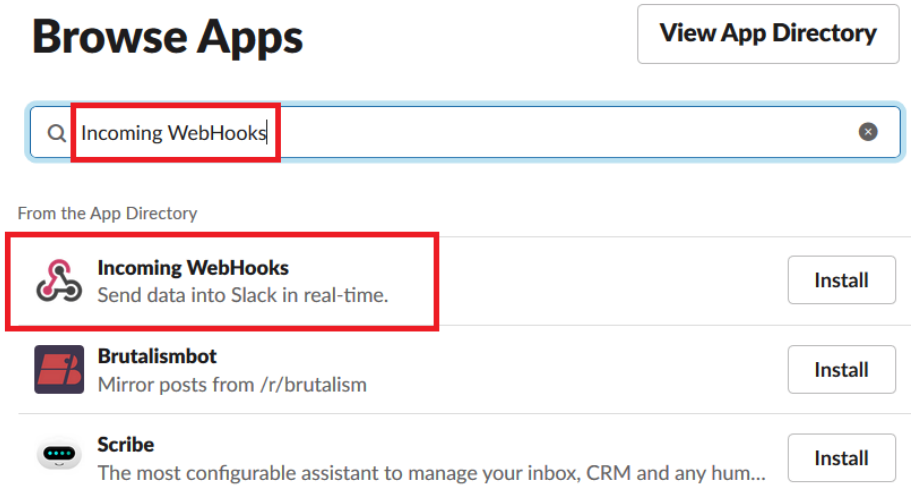
Welcome to Slack!!

2. Step 2: Incorporate the Incoming Hook to Slack

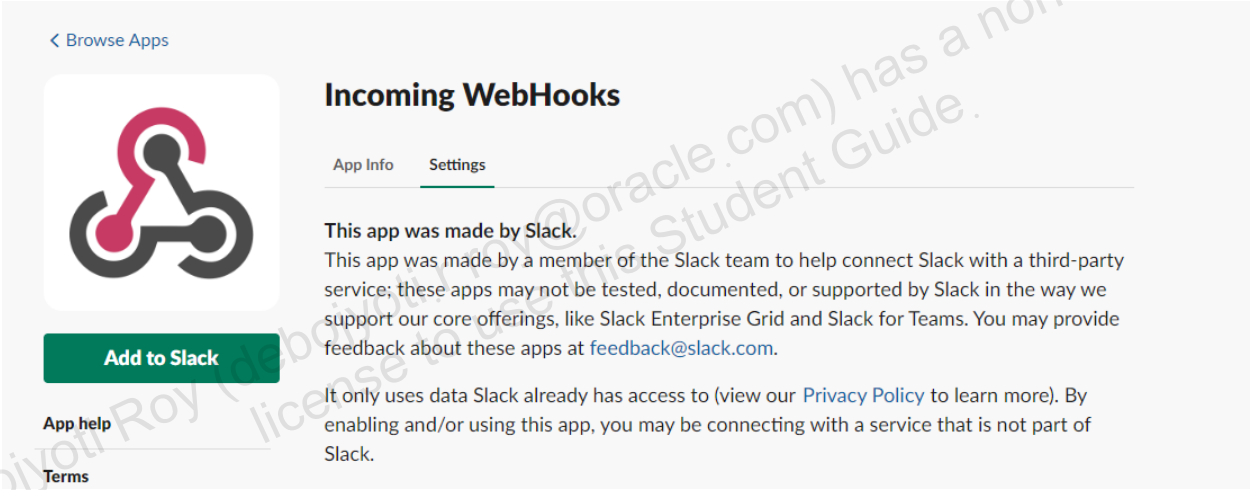
1. On the Slack home page, select **Apps > + Add more apps**



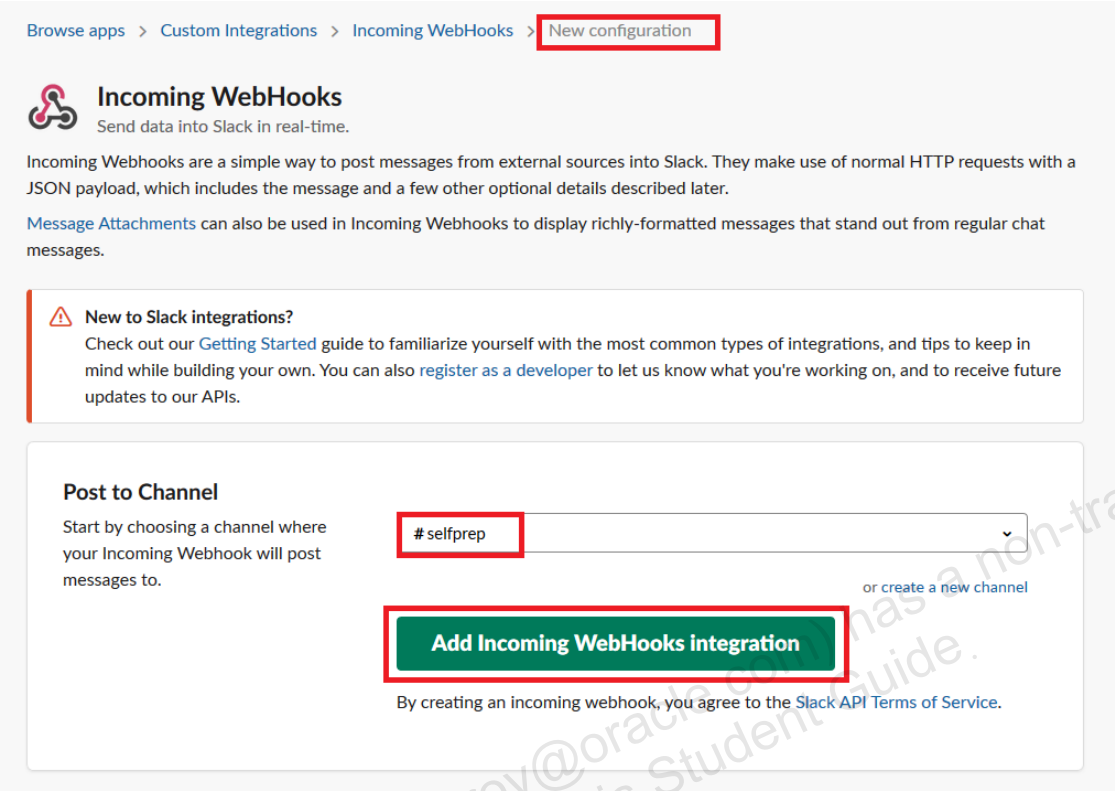
2. Browse the apps and search for **Incoming WebHooks**.



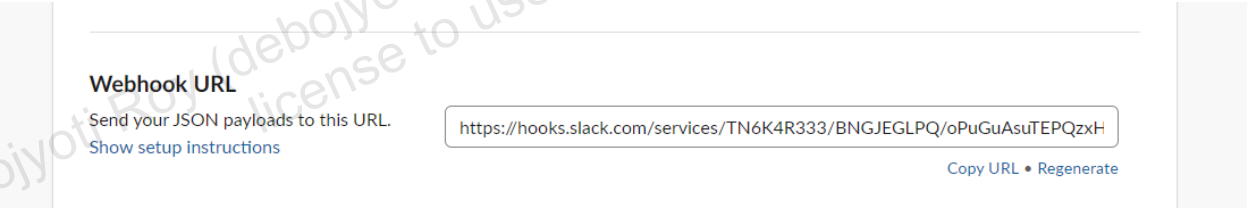
3. Choose **Add to Slack** to include WebHooks.



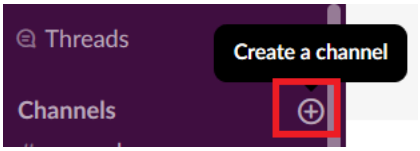
4. On the **configuration** page, choose your **channel**, where you want webhook to post messages. Click **Add Incoming WebHooks integration**.



5. Now note the **Webhook URL**. Click **Save Settings**.



6. On the Slack home page (still open in the previous tab of your browser), choose **Channels > Create a channel**. Create a channel named **notifications.**



The setup for Slack hook is complete.

2. Step 3: Create wercker.yml to enable Integration with the SLACK App

The wercker build is to integrate with SLACK application services and with the werckerBOT application. This application will send out a notification to a specific channel along with the status of the wercker build.

1. Create a new repository under your GIT named `testslack`.
2. Create a new file, `wercker.yml`, with the following script:

a. **Note:** Pay special attention to indentation while working with YAML.

b. You can verify your yaml file using YAML Validator - https://codebeautify.org/yaml-validator

c. For your reference, the yml file is posted in the Linux machine @ `/home/oracle/LabFiles/wercker_lab7.yml`

3.

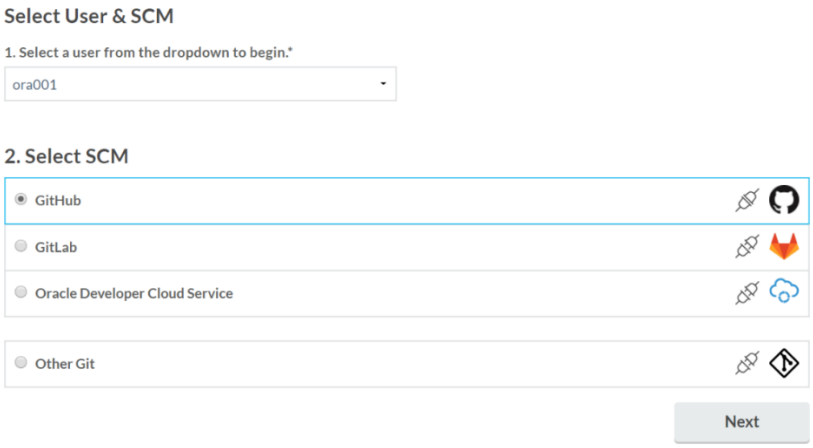
```
box: debian:stable-slim
build:
    steps:
    - install-packages:
        packages: curl
    - script:
        name: install composer
        code: apt install -y curl
    - slack-notifier:
        url: $SLACK_URL
        channel: notifications
        custom_message: Deploying $WERCKER_GIT_BRANCH to
$WERCKER_DEPLOYTARGET_NAME ($WERCKER_DEPLOY_URL)
        custom_color: #439fe0
```

3. **Commit** the file in GIT Repository.
4. Observe the slack notifier in the YML file.

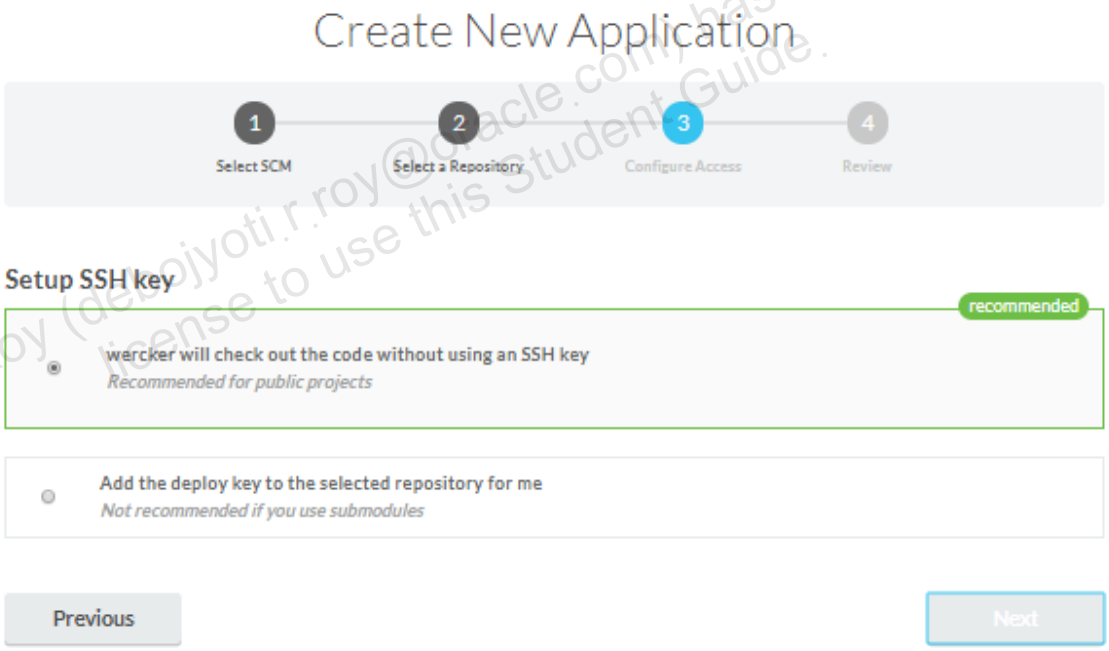
17. Step 4: Set Up the CI/CD Pipeline in Container Pipelines

1. Create a new application by choosing Add application.

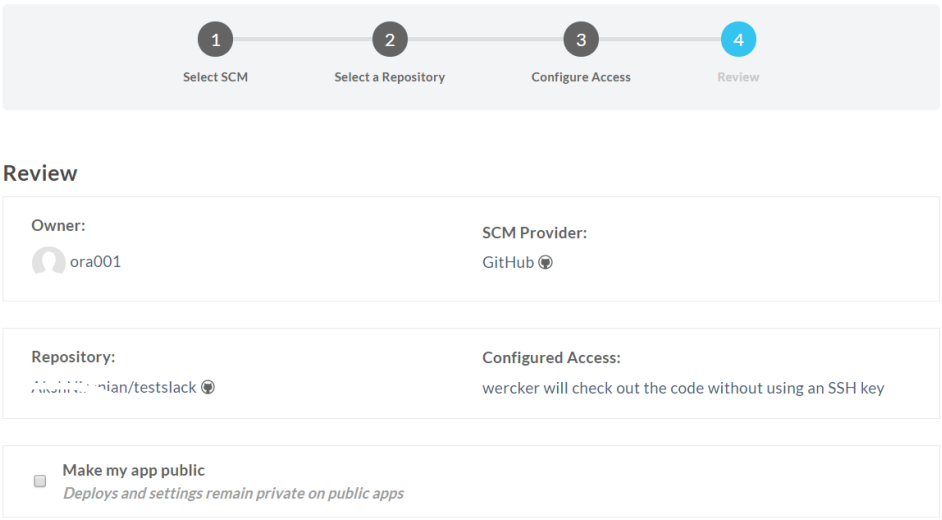
2. Select the **user** and then select **GitHub** as the repository.



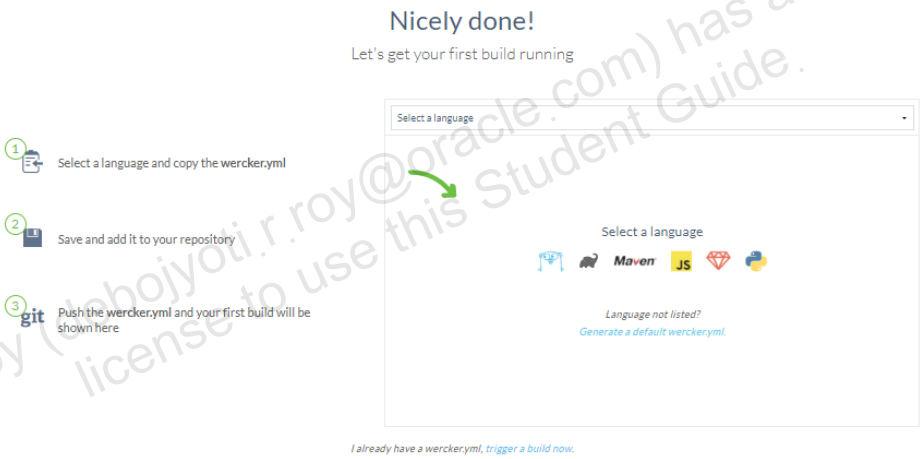
3. Now choose the repository, **testslack**.
4. In **Setup SSH Key**, choose **"wercker will check out the code without using an SSH key"** because the repository is public. Click **Next.**



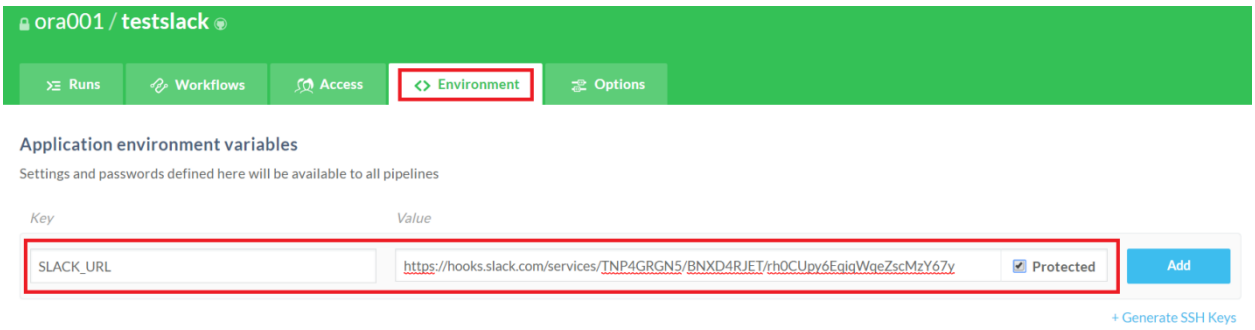
5. Click **Create** after confirming the SCM details.



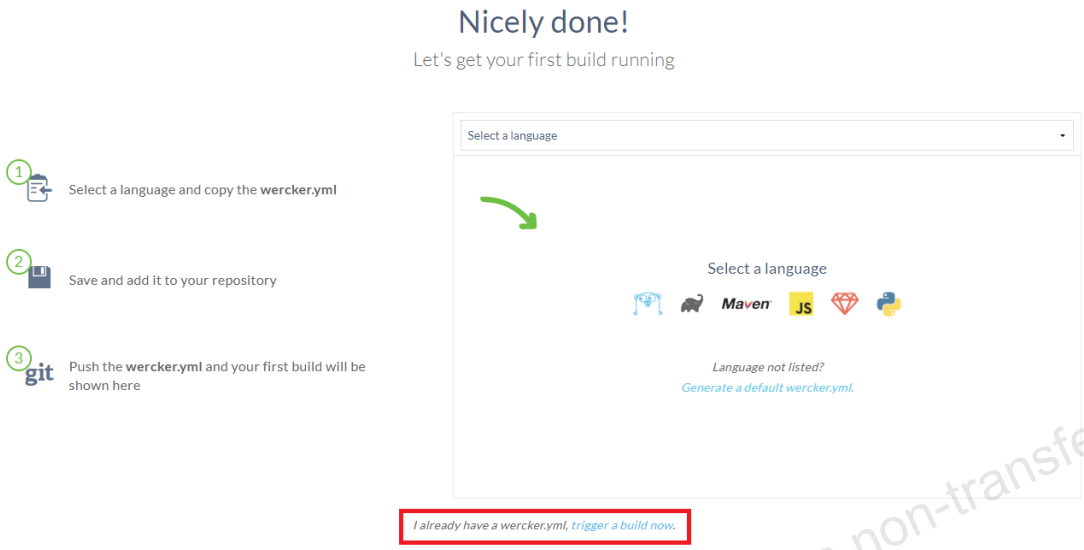
6. Application is successfully built.



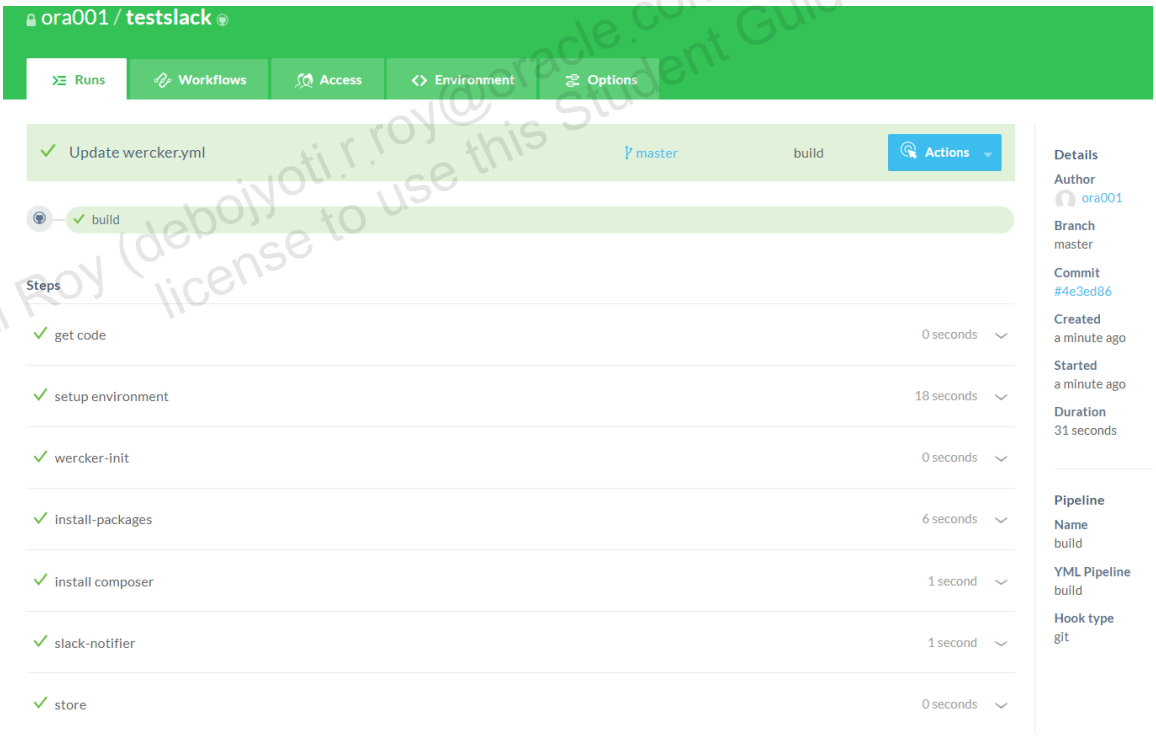
7. Before you trigger the build URL, you need to set the environment variables.
    2. Choose the **Environment** tab section.
3. Create the **SLACK_URL** environment variable. The value for this variable is the WEBHOOK URL that you fetched earlier. This is an important token; therefore, tag it as protected.



8. You are all set. Switch to the **Runs** pane and trigger the build.

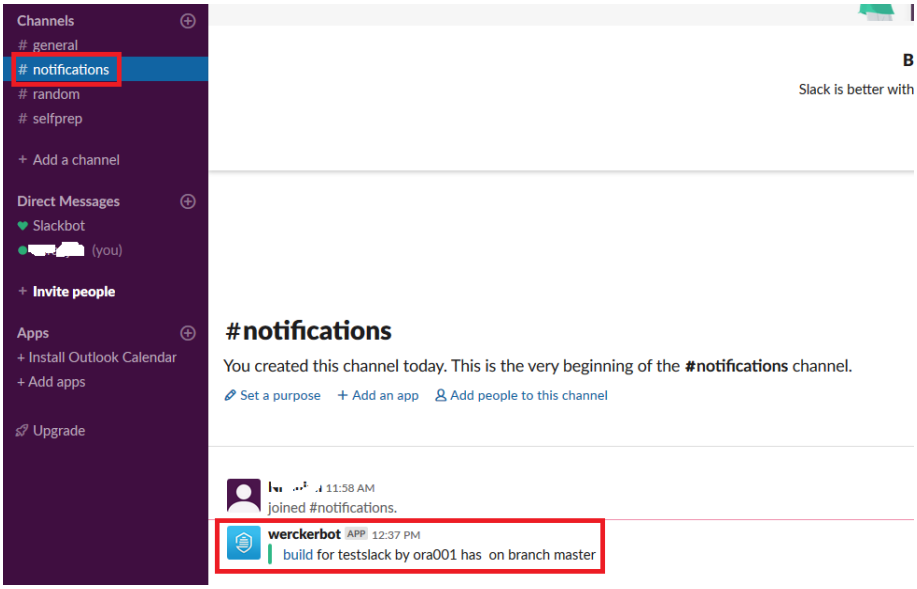


4. The build process will complete successfully. If you encounter any error, debug and rerun the build.



5. Step 5: Test the Slack Notification

1. Open your S**lack** page.
2. Open your **channel - notifications** page.
3. You can see **werckerbot** coming from the `ora` user.



Congratulations!! You have implemented and integrated Slack notifications with Oracle Container Pipeline.