**Purpose**

This lab demonstrates [CircleCI](https://circleci.com/), a continuous integration/continuous deployment (CI/CD) system.

In this lab we will be deploying a pipeline to deploy one application to different environments. The lab will lead you through the anatomy of how CircleCI solves this problem. You can apply the same pipeline structure to other CI tools like Concourse or Jenkins.

For the purpose of this lab there are three environments:

* Local environment (i.e. your workstation)
* Review environment
* Production environment

This is a small but realistic example of a production pipeline. In your actual experience there will likely be more environments such as a QA, staging, pre-production, etc.

When code is pushed to GitHub and tests pass, CircleCI will deploy to the review environment automatically. When the business decides to release a new version of the app, CircleCI will deploy to production at the push of a button.

**Discussion points**

* Importance of CI
* Deploying to different environments
* Favor environment variables over [.NET Environments](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments)
* Benefits/drawbacks of CD
* Walk through CircleCI UI and local files.

**Get started**

Before starting the lab, pull in the CI pipeline definition and the build task:

**cd** ~/workspace/pal-tracker

git cherry-pick pipeline-start

This will create a .circleci directory and a scripts directory. We will explore this directory more deeply during the rest of this lab.

**Configure build**

1. Sign in to [CircleCI](https://circleci.com/login) with your GitHub account.
2. Go to the Add Projects tab on the left side and click Set Up Project for your pal-tracker repository.
3. Enable the pipeline by clicking Start Building. The initial build will fail because we have not yet configured the pipeline.
4. Configure the following environment variables from the [project configuration page](https://circleci.com/docs/2.0/env-vars/#setting-an-environment-variable-in-a-project). These are used in the build pipeline definition:
   * CF\_API
   * CF\_PASSWORD
   * CF\_USERNAME
   * CF\_ORG

**Build files**

CircleCI treats your build pipeline as an artifact of your software, also known as *pipelines as code*.

The .circleci directory contains the pipeline definition. The job of this file is to orchestrate the steps of your build pipeline.

[Hide config.yml](https://courses.education.pivotal.io/c/349802946/cloud-native-developer/dotnet-core-developer/pipelines/index.html" \l "pal-tracker27a96606-d98a-47eb-b08f-39db8be70e85)

pal-tracker/.circleci/config.yml

version: 2

jobs:

build:

docker:

- image: 'microsoft/dotnet:2.1-sdk'

environment:

DEBIAN\_FRONTEND: "noninteractive"

general:

branches:

only:

- my-work *# list of branches to build*

steps:

- checkout

- run:

name: Test and build

command: |

bash ./scripts/build.sh $CIRCLE\_SHA1

- store\_artifacts:

path: artifacts

- persist\_to\_workspace:

root: artifacts

paths:

- pal-tracker-\*.tgz

deployToReview:

docker:

- image: 'microsoft/dotnet:2.1-sdk'

environment:

DEBIAN\_FRONTEND: "noninteractive"

EXTRACT\_DEST: /tmp/extracted-archive

ENVIRONMENT: review

steps:

- attach\_workspace:

at: artifacts

- run:

name: Install CF CLI

command: |

apt-get update

apt-get -y install apt-transport-https ca-certificates

wget -q -O - https://packages.cloudfoundry.org/debian/cli.cloudfoundry.org.key | apt-key add -

echo "deb https://packages.cloudfoundry.org/debian stable main" | tee /etc/apt/sources.list.d/cloudfoundry-cli.list

apt-get update

apt-get install cf-cli

- run:

name: Unzip artifacts

command: |

mkdir -p $EXTRACT\_DEST

tar -xvzf artifacts/pal-tracker-$CIRCLE\_SHA1.tgz -C $EXTRACT\_DEST

- run:

name: Deploy

command: |

cd $EXTRACT\_DEST

cf login -a $CF\_API -u $CF\_USERNAME -p $CF\_PASSWORD -o $CF\_ORG -s $ENVIRONMENT

cf push -f manifest-$ENVIRONMENT.yml

deployToProd:

docker:

- image: 'microsoft/dotnet:2.1-sdk'

environment:

DEBIAN\_FRONTEND: "noninteractive"

EXTRACT\_DEST: /tmp/extracted-archive

ENVIRONMENT: production

steps:

- attach\_workspace:

at: artifacts

- run:

name: Install CF CLI

command: |

apt-get update

apt-get -y install apt-transport-https ca-certificates

wget -q -O - https://packages.cloudfoundry.org/debian/cli.cloudfoundry.org.key | apt-key add -

echo "deb https://packages.cloudfoundry.org/debian stable main" | tee /etc/apt/sources.list.d/cloudfoundry-cli.list

apt-get update

apt-get install cf-cli

- run:

name: Unzip artifacts

command: |

mkdir -p $EXTRACT\_DEST

tar -xvzf artifacts/pal-tracker-$CIRCLE\_SHA1.tgz -C $EXTRACT\_DEST

- run:

name: Deploy

command: |

cd $EXTRACT\_DEST

cf login -a $CF\_API -u $CF\_USERNAME -p $CF\_PASSWORD -o $CF\_ORG -s $ENVIRONMENT

cf push -f manifest-$ENVIRONMENT.yml

workflows:

version: 2

deployment-pipeline:

jobs:

- build

- deployToReview:

requires:

- build

- canDeployToProduction:

type: approval

requires:

- deployToReview

- deployToProd:

requires:

- canDeployToProduction

1. The *jobs* section defines distinct units of work that your pipeline will perform. In this case, there are three jobs: build,deployToReview, and deployToProduction.
2. Each job is run inside of Docker container. In our case, Linux images that have the .Net Core runtime installed on them.
3. Each job has multiple steps. These steps are Bash commands or scripts that are run at the Linux command line.
4. The build job stores away artifacts and persists the state of the workspace for use by subsequent jobs in the pipeline.
5. Finally, a workflow ties all of the steps together. There is a manual approval step required to deploy to production.

The scripts directory contains files that the pipeline uses to build and deploy your software.

scripts/build.sh runs the tests against the software and then builds a TAR file of the necessary artifacts that go into a specific version of the software.

**Configuring applications for multiple environments**

We are configuring two environments (review and production) for our application so we need two manifest files, one for each environment.

We also need a separate route for each application.

1. Move manifest.yml to manifest-review.yml
2. Copy manifest-review.yml to manifest-production.yml.
3. Remove the random-route setting and replace it with statically defined route for each environment. Routes are global to a foundation. We will differentiate your route from others in the Cloud Foundry instance by following [this guide](https://courses.education.pivotal.io/c/349802946/course-instructions/route-naming/index.html).

To find out the correct domain name for apps in your Cloud Foundry foundation, use the cf domains command.

1. Change the environment variables in the manifests to include the environment.

Here are *example* manifests for a *fictional* student named *Theodore Sampsonite*:

[Hide manifest-review.yml](https://courses.education.pivotal.io/c/349802946/cloud-native-developer/dotnet-core-developer/pipelines/index.html" \l "pal-tracker09508699-29a4-4f0e-8c19-575737684eff)

pal-tracker/manifest-review.yml

**---**

applications:

- name: pal-tracker

path: src/PalTracker/bin/Release/netcoreapp2.1/publish/

routes:

- route: msg-pal-tracker-review.apps.pikes.pal.pivotal.io

env:

WELCOME\_MESSAGE: Hello from the review environment

[Hide manifest-production.yml](https://courses.education.pivotal.io/c/349802946/cloud-native-developer/dotnet-core-developer/pipelines/index.html" \l "pal-tracker646c7ee0-babb-4423-9f97-3f78e0afdaca)

pal-tracker/manifest-production.yml

**---**

applications:

- name: pal-tracker

path: src/PalTracker/bin/Release/netcoreapp2.1/publish/

routes:

- route: msg-pal-tracker.apps.pikes.pal.pivotal.io

env:

WELCOME\_MESSAGE: Hello from the production environment

**Cloud Foundry changes for multiple environments**

Now that we have multiple environments to deploy to we need to configure Cloud Foundry with those environments.

1. Create a *review* space and a *production* space in your Cloud Foundry org with:
2. cf create-space review
3. cf create-space production

**Triggering your pipeline**

1. Commit and push your changes. This will trigger CircleCI to build your application.

**View deployed applications**

1. After the pipeline runs, check the deployed app in your review environment and verify that the welcome message is correct.
2. Navigate to your pipeline from the Workflows tab in CircleCI. Click the *canDeployToProduction* box to approve the deployment to production.
3. Confirm that the application deployed successfully to the production environment and displays the correct production greeting.

**Assignment**

Submit the assignment using the cloudNativeDeveloperPipelines gradle task. It requires you to provide the URLs of your review and production applications.

For example:

**cd** ~/workspace/assignment-submission

./gradlew cloudNativeDeveloperPipelines -PreviewUrl=https://${REVIEW\_APP\_URL} -PproductionUrl=https://${PRODUCTION\_APP\_URL}

**Wrap up**

Blue/green deploys