**LAB 7 : JAVA WEB APP DEPLOYMENT USING AZURE CICD**

**Lab 8: Getting Started with YAML Pipelines in Azure DevOps**

**Lab Objective**

By the end of this lab, you will:  
✅ Understand the **concept and structure** of YAML pipelines  
✅ Know how to **create a YAML pipeline from scratch** in Azure DevOps  
✅ Be able to **run and monitor** your pipeline  
✅ Understand how YAML defines **stages, jobs, and steps**

**Prerequisites**

Before starting, ensure:

* You have an **Azure DevOps organization and project** created.
* You have a **Git repository** connected to the project.
* You have at least one **self-hosted or Microsoft-hosted agent** configured.  
  (We’ll assume you have test-pool as a self-hosted agent pool.)

**Step 1: Understanding What a YAML Pipeline Is**

**💡 What is YAML?**

YAML (YAML Ain’t Markup Language) is a **human-readable configuration language** used to define pipelines as code.

Instead of clicking buttons in the classic editor, we define everything in a .yml or .yaml file stored alongside your source code.

**🧩 Why YAML Pipelines?**

| **Benefit** | **Explanation** |
| --- | --- |
| **Pipeline as Code** | Pipeline definition is version-controlled along with the app. |
| **Reusable** | YAML templates and parameters make it modular. |
| **Portable** | Works across repos and environments. |
| **Automated CI/CD** | Defines CI/CD workflow end-to-end. |

**Step 2: YAML Pipeline Core Structure**

Every YAML pipeline has this structure:

trigger:

branches:

include:

- main

pool:

name: 'test-pool'

stages:

- stage: Build

jobs:

- job: BuildJob

steps:

- script: echo "Hello, YAML!"

displayName: "Run a sample script"

**🔍 Breakdown of the Structure**

| **Section** | **Description** |
| --- | --- |
| **trigger** | Defines when the pipeline runs (e.g., on commit to a branch). |
| **pool** | Specifies which agent pool executes the pipeline. |
| **stages** | Logical grouping of pipeline phases (e.g., Build, Test, Deploy). |
| **jobs** | Independent units of work that can run in parallel or sequence. |
| **steps** | Actual tasks or scripts to execute commands. |

**Step 3: Creating Your First YAML Pipeline**

**1️⃣ In Azure DevOps UI**

1. Go to **Pipelines → Create Pipeline**
2. Choose **“Azure Repos Git”**
3. Select your repository
4. Choose **“Starter Pipeline”**
5. Replace the contents with this YAML:

trigger:

branches:

include:

- main

pool:

name: 'test-pool' # Or use 'vmImage: ubuntu-latest' for Microsoft-hosted

stages:

- stage: Build

displayName: "Build Stage"

jobs:

- job: BuildJob

displayName: "Run Basic Build"

steps:

- script: echo "Welcome to YAML Pipelines!"

displayName: "Step 1 - Print Message"

- script: |

echo "Current directory:"

pwd

echo "List of files:"

ls -l

displayName: "Step 2 - List Repo Files"

- script: |

echo "Simulating build..."

mkdir output

echo "Build successful!" > output/build.txt

displayName: "Step 3 - Create a Build Output File"

- task: PublishBuildArtifacts@1

inputs:

PathtoPublish: 'output'

ArtifactName: 'sample-build-artifact'

publishLocation: 'Container'

displayName: "Step 4 - Publish Artifact"

**2️⃣ Save and Run**

* Save the pipeline as azure-pipelines.yml in your repo.
* Click **Run** → select the **main** branch → **Run Pipeline**.

Azure DevOps will:

1. Checkout your repo
2. Execute the steps
3. Publish the output/build.txt artifact

**Step 4: Exploring the Pipeline Run**

Once the pipeline completes:

1. Click on the **Run ID** → opens the run summary.
2. Explore:
   * **Logs** (per step)
   * **Artifacts** tab → contains sample-build-artifact
   * **Pipeline graph** showing stages/jobs flow.

**Step 5: Expanding the Pipeline**

Let’s make it a bit more realistic — simulate Build, Test, and Publish.

trigger:

branches:

include:

- main

pool:

name: 'test-pool'

stages:

- stage: Build

displayName: "Build Stage"

jobs:

- job: BuildJob

displayName: "Build Application"

steps:

- script: echo "Installing dependencies..."

displayName: "Install Dependencies"

- script: echo "Building app..."

displayName: "Build Step"

- script: echo "Build completed successfully!"

displayName: "Confirm Build"

- stage: Test

displayName: "Test Stage"

dependsOn: Build

jobs:

- job: TestJob

displayName: "Run Tests"

steps:

- script: echo "Running unit tests..."

displayName: "Execute Tests"

- script: echo "All tests passed!"

displayName: "Confirm Test"

- stage: Publish

displayName: "Publish Artifacts"

dependsOn: Test

jobs:

- job: PublishJob

displayName: "Publish Outputs"

steps:

- script: |

mkdir -p $(Build.ArtifactStagingDirectory)/output

echo "Final output for deployment" > $(Build.ArtifactStagingDirectory)/output/result.txt

displayName: "Prepare Output"

- task: PublishBuildArtifacts@1

inputs:

PathtoPublish: '$(Build.ArtifactStagingDirectory)/output'

ArtifactName: 'final-artifact'

publishLocation: 'Container'

**🔍 Explanation**

| **Section** | **Description** |
| --- | --- |
| **dependsOn** | Ensures stages execute in order: Build → Test → Publish |
| **$(Build.ArtifactStagingDirectory)** | Predefined variable for storing artifacts |
| **PublishBuildArtifacts@1** | Uploads your output files to Azure DevOps artifact storage |

**Step 6: Key Azure DevOps Variables**

YAML pipelines have many predefined variables.  
Some useful ones:

| **Variable** | **Meaning** |
| --- | --- |
| $(Build.SourcesDirectory) | Folder where your source code is checked out |
| $(Build.ArtifactStagingDirectory) | Temporary directory for staging build outputs |
| $(Build.BuildNumber) | Unique number for each pipeline run |
| $(Agent.OS) | OS type of the build agent |
| $(System.DefaultWorkingDirectory) | Root working directory for pipeline steps |

More: Microsoft Docs – Predefined Variables

**Step 7: Tips for Writing YAML Pipelines**

✅ **Indentation matters!** YAML is space-sensitive (use 2 spaces, not tabs).  
✅ Always give each step a **displayName** (helps with logs).  
✅ Use **variables** to make paths portable.  
✅ Use **dependsOn** to control stage order.  
✅ Validate your YAML with Azure DevOps’ **“Validate”** button before running.

**Lab 9 —Pipeline with Azure DevOps YAML**

**Overview**

This lab walks through creating a **CI pipeline using YAML** in Azure DevOps for a **Python Flask application** and publishing build artifacts so they can be consumed in a **CD pipeline** later.

We will cover:

1. Flask app setup
2. Azure DevOps project creation
3. Repository setup
4. YAML pipeline creation
5. Publishing build artifacts
6. Running and verifying builds

**Pre-requisites**

* Azure DevOps account (free tier is fine)
* Access to Azure DevOps Project
* Azure CLI installed (optional but useful)
* Git installed
* Basic knowledge of Python and CI/CD
* Familiarity with YAML is helpful but not required

**Step 1 — Create Sample Flask Application**

We will use a simple Python Flask app.

**Directory structure:**

flask-app/

├── app.py

├── requirements.txt

├── tests/

│ └── test\_app.py

└── README.md

**app.py**

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

return "Hello, Azure DevOps CI Pipeline!"

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", port=5000)

**requirements.txt**

Flask==2.2.5

Werkzeug==2.2.3

**tests/test\_app.py**

import unittest

from app import app

class BasicTestCase(unittest.TestCase):

def setUp(self):

self.app = app.test\_client()

self.app.testing = True

def test\_home(self):

rv = self.app.get('/')

self.assertEqual(rv.status\_code, 200)

self.assertEqual(rv.data, b"Hello, Azure DevOps CI Pipeline!")

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**README.md**

# Flask App

Simple Flask application for CI Pipeline Demo in Azure DevOps.

**Step 2 — Push Code to Azure Repos**

1. **Create a repository**:
   * Name: flask-app.
2. **Clone repository locally**:

git clone https://dev.azure.com/<your-org>/<project-name>/\_git/flask-app

cd flask-app

1. Copy Flask app code into repo folder.
2. Add files:

git add .

git commit -m "Initial Flask app with tests"

git push origin main

**Step 3 — Understanding YAML Pipelines**

Azure Pipelines YAML is a **text file (azure-pipelines.yml) that defines CI/CD workflows**.

Advantages:

* Version-controlled alongside application code.
* Reusable pipelines.
* Easily customizable.

Key YAML pipeline concepts:

* **trigger** → specifies when the pipeline runs.
* **pool** → specifies agent pool to run pipeline.
* **steps** → tasks to be executed sequentially.
* **variables** → reusable configuration parameters.

--

Install python :

sudo apt update

sudo apt install -y software-properties-common

sudo add-apt-repository ppa:deadsnakes/ppa

sudo apt update

sudo apt install -y python3.10 python3.10-venv python3.10-dev

python3.10 –version

* + Install pip on local machine agent :

sudo apt update

sudo apt install -y python3.10-distutils wget

wget https://bootstrap.pypa.io/get-pip.py

sudo python3.10 get-pip.py

**Step 4 — Create YAML CI Pipeline with Artifact Publishing**

Create azure-pipelines.yml in the repo root:

trigger:

branches:

include:

- main

stages:

# ===== STAGE 1: Build =====

- stage: Build

displayName: "Build Stage"

jobs:

- job: BuildJob

displayName: "Build Flask App"

pool:

name: 'test-pool'

steps:

- script: |

python3.10 --version

displayName: "Check Python Version"

- script: |

python3.10 -m pip install --upgrade pip

python3.10 -m pip install -r requirements.txt

displayName: "Install Dependencies"

# ===== STAGE 2: Test =====

- stage: Test

displayName: "Test Stage"

dependsOn: Build

jobs:

- job: TestJob

displayName: "Run Unit Tests"

pool:

name: 'test-pool'

steps:

- script: |

python3.10 -m unittest discover -s tests

displayName: "Run Unit Tests"

# ===== STAGE 3: Run App (Sanity Check) =====

- stage: RunApp

displayName: "Run Flask App"

dependsOn: Test

jobs:

- job: RunJob

displayName: "Run Flask App for Sanity Check"

pool:

name: 'test-pool'

steps:

# Step 1: Run app in background

- script: |

nohup python3.10 app.py &

sleep 5

displayName: "Start Flask App"

# Step 2: Check app is running

- script: |

curl http://127.0.0.1:5000 || exit 1

displayName: "Sanity Check - HTTP Request"

# Step 3: Stop app

- script: |

pkill -f app.py

displayName: "Stop Flask App"

# ===== STAGE 3: Publish =====

- stage: Publish

displayName: "Publish Artifacts Stage"

dependsOn: RunApp

jobs:

- job: PublishJob

displayName: "Publish Flask App Artifact"

pool:

name: 'test-pool'

steps:

# Step 1: Prepare artifact folder

- script: |

mkdir -p $(Build.ArtifactStagingDirectory)/flask-app

cp -r \* $(Build.ArtifactStagingDirectory)/flask-app

displayName: "Prepare Artifact Files"

# Step 2: Publish artifact

- task: PublishBuildArtifacts@1

inputs:

PathtoPublish: '$(Build.ArtifactStagingDirectory)/flask-app'

ArtifactName: 'flask-app-artifact'

publishLocation: 'Container'

**Top-level structure**

trigger:

branches:

include:

- main

* **Trigger** → This pipeline runs automatically when a change is pushed to the main branch.

**Stage 1 — Build**

- stage: Build

**Purpose:**  
Install dependencies and prepare the app for testing.

**Job:**

- job: BuildJob

pool:

name: 'test-pool'

* Runs the build job on your **self-hosted agent pool** named test-pool.

**Steps:**

1. **Check Python version**
2. python3.10 --version

→ Ensures your self-hosted agent has Python 3.10 installed.

1. **Install dependencies**
2. python3.10 -m pip install --upgrade pip
3. python3.10 -m pip install -r requirements.txt

→

* + python3.10 -m pip install --upgrade pip: Upgrades pip to the latest version for Python 3.10.
  + pip install -r requirements.txt: Installs all required Python libraries listed in requirements.txt (including Flask, Werkzeug, etc.).

**Stage 2 — Test**

- stage: Test

**Purpose:**  
Run unit tests to verify the app works correctly before moving further.

**Job:**

- job: TestJob

Runs unit tests on the Flask app.

**Steps:**

python3.10 -m unittest discover -s tests

* Uses Python’s built-in unittest framework.
* discover automatically finds all tests in the tests folder.
* If any test fails → the pipeline stops.

**Stage 3 — Run App (Sanity Check)**

- stage: RunApp

**Purpose:**  
Run the Flask app briefly to check that it works before publishing artifacts.

**Job:**

- job: RunJob

**Steps:**

1. **Start the Flask app**
2. nohup python3.10 app.py &
3. sleep 5
   * nohup: Runs the app in the background even if the terminal closes.
   * &: Runs in background.
   * sleep 5: Waits 5 seconds to give the app time to start.
4. **Sanity check with HTTP request**
5. curl http://127.0.0.1:5000 || exit 1
   * Uses curl to send a request to the app.
   * If the request fails → exits with error code 1, which stops the pipeline.
6. **Stop the Flask app**
7. pkill -f app.py
   * Kills the Flask app process started earlier.

**Why this stage is useful:**  
It prevents publishing an artifact for a broken app.

**Stage 4 — Publish**

- stage: Publish

**Purpose:**  
Package the application and publish it as a build artifact so it can be deployed later.

**Job:**

- job: PublishJob

**Steps:**

1. **Prepare artifact folder**

mkdir -p $(Build.ArtifactStagingDirectory)/flask-app

cp app.py requirements.txt $(Build.ArtifactStagingDirectory)/flask-app/

cp -r templates static $(Build.ArtifactStagingDirectory)/flask-app/

* + $(Build.ArtifactStagingDirectory) → predefined Azure DevOps variable that is a staging area for artifacts.
  + Creates a clean folder (flask-app) containing:
    - app.py
    - requirements.txt
    - templates folder (HTML templates)
    - static folder (CSS, JS, images)
  + Excludes unnecessary files like tests, .git, logs, etc.

1. **Publish the artifact**

- task: PublishBuildArtifacts@1

inputs:

PathtoPublish: '$(Build.ArtifactStagingDirectory)/flask-app'

ArtifactName: 'flask-app-artifact'

publishLocation: 'Container'

* + PathtoPublish: Path to the folder with the build output.
  + ArtifactName: The name of the artifact.
  + publishLocation: 'Container': Publishes artifact to Azure DevOps’ artifact storage.

**Pipeline Flow Summary**

1. **Build stage** → installs dependencies
2. **Test stage** → runs unit tests
3. **RunApp stage** → sanity-checks the app runs
4. **Publish stage** → packages the app and publishes artifact

**Lab 10 : Creating Continuous Integration (CI) Pipelines with Python**

**Lab Objectives**

By the end of this lab, you will be able to:

1. Create a **YAML-based CI pipeline** for a Python application.
2. Configure build tasks: **restore dependencies → run tests → publish artifacts**.
3. Compare **Microsoft-hosted** and **self-hosted** agents.
4. Trigger builds automatically on code commits to branches.

**Prerequisites**

* Completed **Lab 6 (Branching & Merge Conflicts)**.
* Azure DevOps Project: DevOpsTraining.
* Repository: python-ci-demo (can be created afresh).
* Installed tools (on local system if doing self-hosted agent part):
  + Git
  + Visual Studio Code (optional)
  + Python 3.9+
  + pip and virtualenv

**Part A – Preparing a Python Project**

Let’s first create a simple Python application that can be built and tested automatically.

**Step 1: Clone Repository Locally**

git clone https://dev.azure.com/<organization>/<project>/\_git/python-ci-demo

cd python-ci-demo

**Step 2: Create Application Structure**

Inside the repository, create this folder layout:

python-ci-demo/

│

├── src/

│ ├── \_\_init\_\_.py

│ └── app.py

│

├── tests/

│ ├── \_\_init\_\_.py

│ └── test\_app.py

│

├── requirements.txt

└── README.md

**Step 3: Add Python Code**

**src/app.py**

def add(a, b):

return a + b

def subtract(a, b):

return a - b

if \_\_name\_\_ == "\_\_main\_\_":

print("Add 10 + 5 =", add(10, 5))

print("Subtract 10 - 5 =", subtract(10, 5))

**tests/test\_app.py**

from src.app import add, subtract

def test\_add():

assert add(3, 2) == 5

def test\_subtract():

assert subtract(3, 2) == 1

**requirements.txt**

pytest

**README.md**

# Python CI Demo

A simple Python project to demonstrate Azure DevOps Continuous Integration (CI) pipelines.

**Step 4: Commit & Push Code**

git add .

git commit -m "Initial Python project for CI demo"

git push origin dev

**Part B – Creating the YAML CI Pipeline**

**Step 1: Navigate to Azure DevOps Pipelines**

1. In Azure DevOps portal → open your project → **Pipelines → Pipelines**.
2. Click **New pipeline** → choose **Azure Repos Git (YAML)**.
3. Select repository → python-ci-demo.
4. Choose **Starter pipeline**.

**Step 2: Replace YAML Content**

Replace everything with the following:

trigger:

  branches:

    include:

      - dev

      - main

pool:

  vmImage: 'ubuntu-latest'

variables:

  pythonVersion: '3.9'

  artifactPath: '$(Build.ArtifactStagingDirectory)/drop'

steps:

- checkout: self

- task: UsePythonVersion@0

  displayName: 'Set up Python $(pythonVersion)'

  inputs:

    versionSpec: '$(pythonVersion)'

    addToPath: true

- script: |

    python -m pip install --upgrade pip

    pip install -r requirements.txt

  displayName: 'Install dependencies'

- script: |

    pytest -v --junitxml=$(Build.SourcesDirectory)/pytest-report.xml

  displayName: 'Run unit tests'

- task: PublishTestResults@2

  displayName: 'Publish test results'

  inputs:

    testResultsFormat: 'JUnit'

    testResultsFiles: '$(Build.SourcesDirectory)/pytest-report.xml'

    failTaskOnFailedTests: true

- task: ArchiveFiles@2

  displayName: 'Archive source code'

  inputs:

    rootFolderOrFile: '$(Build.SourcesDirectory)'

    includeRootFolder: false

    archiveType: 'zip'

    archiveFile: '$(artifactPath)/python-app.zip'

    replaceExistingArchive: true

- task: PublishBuildArtifacts@1

  displayName: 'Publish build artifact'

  inputs:

    PathtoPublish: '$(artifactPath)/python-app.zip'

    ArtifactName: 'drop'

    publishLocation: 'Container'

**Step 3: Save and Run**

1. Click **Save and Run**.
2. Commit to the dev branch as .azure-pipelines.yml.
3. Observe that the build automatically starts.

You’ll see the following sequence of logs:

* Setting up environment (ubuntu-latest)
* Installing dependencies
* Running pytest
* Publishing results and artifacts

✅ **Expected Outcome:**  
Pipeline completes successfully with 1 artifact (python-app.zip) and test results in the summary tab.

**Part C – Understanding the Build Process**

| **Stage** | **Purpose** | **Notes** |
| --- | --- | --- |
| **Setup Python** | Installs Python runtime on agent | Uses UsePythonVersion@0 task |
| **Install dependencies** | Installs required packages | From requirements.txt |
| **Run tests** | Executes unit tests | pytest generates XML report |
| **Publish results** | Uploads test report to Azure DevOps | Displays under "Tests" tab |
| **Publish artifacts** | Archives and stores code output | Used in later CD labs |

**Part D – Configuring Build Triggers**

1. The YAML file includes:
2. trigger:
3. branches:
4. include:
5. - dev
6. - main
7. This means any **commit or pull request** to dev or main triggers a new build.
8. Test it:
   * Modify README.md.
   * Commit and push:
   * git commit -am "Updated readme for CI trigger test"
   * git push origin dev
   * Go to **Pipelines → Runs** — new build should start automatically.

✅ **Trigger Verified!**

**Part E – Exploring Agents**

**1. Microsoft-hosted Agents**

* Ephemeral Linux VMs provided by Microsoft.
* Clean environment for every build.
* Defined via:
* pool:
* vmImage: 'ubuntu-latest'
* Advantages: No setup, auto-updates, multi-OS support.

**2. Self-hosted Agents (Optional Advanced Exercise)**

**When to use:**

* Need custom Python libraries, cached dependencies, or internal network access.

**Setup Steps:**

1. In **Project Settings → Agent Pools**, click **Add pool** (SelfHostedPool).
2. Select **New agent** → choose **Linux** or **Windows**.
3. Download the agent package to your local machine/VM.
4. Configure agent:
5. ./config.sh --url https://dev.azure.com/<org> --auth pat --token <YourPAT> --pool SelfHostedPool
6. ./run.sh
7. Update YAML to use it:
8. pool:
9. name: 'SelfHostedPool'
10. Commit and rerun pipeline.

✅ You’ll now see your pipeline executing on the self-hosted machine.

**Part F – Verifying Pipeline Artifacts and Results**

1. Go to **Pipelines → Runs → Latest Run**.
2. Check:
   * ✅ All tasks completed successfully.
   * 🧪 Test results tab shows 2 tests passed.
   * 📦 Artifact drop → contains python-app.zip.

**Verification Summary**

| **Step** | **Action** | **Verification** |
| --- | --- | --- |
| A | Python project created | Local repo structure ready |
| B | YAML pipeline created | Pipeline defined and committed |
| C | Build executed | Logs show successful run |
| D | Trigger tested | New commit triggered build |
| E | Artifact published | ZIP file available in artifacts tab |

**Expected End Result**

✅ A fully working **Python-based Continuous Integration (CI) pipeline** that:

* Installs dependencies and runs tests automatically.
* Publishes artifacts to Azure DevOps.
* Runs on Microsoft-hosted or self-hosted agents.
* Triggers automatically upon commits to dev or main.