Module 4: Pipelines Basics

1. Classic vs YAML Pipelines

Azure Pipelines offers **two primary approaches** to defining CI/CD workflows:

- **Classic (GUI-based) Pipelines**
- **YAML (Pipeline-as-Code) Pipelines**

Both use the same underlying engine but differ in **how** the pipeline is defined, stored, and maintained.

1.1 Classic Pipelines

Classic pipelines are created and managed using the **graphical UI** in the Azure DevOps portal.

Key Characteristics

- Defined through a **visual editor** (tasks, phases, agents configured via UI)
- Stored in **Azure DevOps**, *not* in the source repository
- Very useful for teams:
    - New to DevOps and CI/CD
    - Preferring click-and-configure over writing YAML
    - Migrating from legacy tools

Common Usage

- GUI-based build definitions
- Classic release pipelines with environments and approvals
- Quick POCs and demos

Pros

- Easier for beginners, less intimidating than YAML
- Visual representation of tasks, flows, and environments
- No need to modify repo to change pipeline

Cons

- Configuration is **not version-controlled** with the code
- Harder to review changes (no PR-based evolution)
- Less suitable for "Everything as Code" practices

1.2 YAML Pipelines

YAML pipelines are defined using a **.yml file inside the code repository**.

Key Characteristics

- Pipeline definition is stored alongside the application code
- Changes to pipeline go through PRs and code reviews
- Supports **multi-stage** CI/CD in a single YAML file
- Highly reusable via templates and variable groups

Pros

- Pipeline-as-code → version control, history, rollback
- Supports reusability (templates) across repos/services
- Better suited for microservices and large-scale DevOps

Cons

- Learning curve for YAML syntax and structure
- Initial setup may take longer than Classic

1.3 Classic vs YAML – Summary Table

Aspect	**Classic Pipeline**	**YAML Pipeline**
Definition Style	UI-based (visual designer)	File-based (azure-pipelines.yml)
Storage	In Azure DevOps	In source repo
Version Control	Limited (UI history)	Full Git history & PRs
Multi-Stage Support	Separate build & release	Single multi-stage YAML
Reusability	Limited	Templates, variables, libraries
Ideal Audience	Beginners / legacy users	Modern DevOps / “Everything as Code”

---

2. Build Agents (Hosted vs Self-hosted)

Pipelines run on **agents**, which are machines that execute the jobs/tasks defined in a pipeline.

2.1 Microsoft-Hosted Agents

Agents fully managed by Microsoft.

Characteristics

- Run on Microsoft-managed VMs
- Pre-installed tools (SDKs, CLI tools, build tools, etc.)
- Automatically provisioned and cleaned up per job
- Billed based on parallel jobs and usage minutes (or free tier for small use)

Pros

- Zero maintenance – no patching or upgrading
- Fast onboarding
- Good for most standard tech stacks (.NET, Java, Node, Python, etc.)

Cons

- Limited customization (you can't preinstall everything you want permanently)
- No direct access to private on-prem resources unless configured with networking (self-hosted or service endpoints)

2.2 Self-hosted Agents

Agents you provision and manage on your own infrastructure (on-prem VM, cloud VM, container, etc.).

Characteristics

- You install the **Azure Pipelines agent** on your machine
- Can run on Windows, Linux, or macOS
- You control installed tools, network access, and performance

Pros

- Full control over software, tools, versions
- Can access internal networks/resources (on-prem databases, file servers, internal APIs)
- Potentially lower cost at scale (using existing infrastructure)

Cons

- You manage OS patching, agent updates, scaling, and security
- Requires infra and operational ownership

2.3 Hosted vs Self-Hosted – When to Use What

- **Use Microsoft-hosted agents when**:
  - Standard tech stacks
  - Public cloud-only apps

    - Minimal infra management desired
- **Use Self-hosted agents when**:
    - Needs access to internal networks/systems
    - Highly customized toolchain
    - Very specific OS configurations or hardware needs

---

3. Triggers in YAML Pipelines (Branch, Path, PR)

Triggers define **when** a pipeline runs.

3.1 Branch Triggers

Run the pipeline when changes are pushed to specific branches.

```
trigger:
  branches:
    include:
      - main
      - develop
```

- Triggered whenever a commit is pushed to main or develop.

3.2 Path Filters

Control triggers based on **changed file paths**.

```
trigger:
  branches:
    include:
      - main
  paths:
    include:
      - src/**
    exclude:
      - docs/**
```

- Pipeline runs if files under src/ change
- Ignores changes under docs/

3.3 Pull Request (PR) Triggers

Run validation when a pull request is opened/updated.

```
pr:
  branches:
    include:
      - main
      - develop
```

- Used for **PR validation**: build + tests must pass before merging.

You can combine trigger and pr in the same file to control both push and PR behavior.

3.4 Scheduled Triggers (Optional)

Run pipelines on a **schedule**, like nightly builds.

```
schedules:
- cron: "0 1 * * *"   # Every day at 1 AM UTC
  displayName: Nightly Build
  branches:
    include:
      - main
  always: true
```

---

4. Variables & Secrets Management

Variables make pipelines **configurable** and **DRY (Don't Repeat Yourself)**.

4.1 Simple Variables

```
variables:
  buildConfiguration: 'Release'
  dotnetVersion: '8.0.x'
```

Usage:

```
- script: dotnet build --configuration $(buildConfiguration)
```

4.2 Variable Groups (Library)

- Defined in **Pipelines → Library → Variable groups**
- Can be reused across multiple pipelines
- Good for environment-specific values (e.g., DEV_URL, TEST_URL)

```
variables:
- group: Common-App-Settings
```

4.3 Secrets Management

Secrets should **never** be hard-coded. Use:

1. **Secret Variables** in Variable Groups
2. **Azure Key Vault integration**

4.3.1 Secret Variables

- Mark variables as **secret** in the UI.
- They are masked in logs.

```
variables:
  - name: dbPassword
    value: $(dbPassword)   # Already secret in the library
```

4.3.2 Azure Key Vault Integration

- Create a Key Vault in Azure
- Store secrets there
- Link Key Vault to Azure DevOps variable group

Example usage in YAML (after linking via UI):

```
variables:
- group: KeyVault-Secrets
```

Then use as:

```
- script: echo "Connecting to database..."
  env:
    DB_PASSWORD: $(dbPassword)
```

Secrets are **never printed** in logs (they appear as ***).

---

5. Lab: Create a YAML Build Pipeline for .NET / Java App

This lab can be used in training/workshops.

5.1 Pre-requisites

- Azure DevOps organization and project
- A Git repository with:
    - Either a **.NET** app (e.g., ASP.NET Core)
    - Or a **Java** app (e.g., Maven or Gradle project)

Directory example:

.NET Example

```
src/
  WebApp/
    WebApp.csproj
```

Java Example (Maven)

```
src/
  webapp/
    pom.xml
```

5.2 Step 1: Create azure-pipelines.yml in the Repo

Option A: .NET Sample YAML

```
trigger:
  branches:
    include:
      - main
      - develop


pr:
  branches:
    include:
      - main
      - develop
```

```yaml
pool:
  vmImage: 'windows-latest'

variables:
  buildConfiguration: 'Release'

stages:
- stage: Build
  displayName: Build and Test .NET App
  jobs:
  - job: BuildJob
    displayName: Build Job
    steps:
    - task: UseDotNet@2
      displayName: 'Install .NET SDK'
      inputs:
        packageType: 'sdk'
        version: '8.0.x'

    - script: |
        dotnet restore ./src/WebApp/WebApp.csproj
      displayName: 'Restore dependencies'

    - script: |
        dotnet build ./src/WebApp/WebApp.csproj --configuration $(buildConfiguration) --no-restore
      displayName: 'Build project'

    - script: |
        dotnet test ./tests/WebApp.Tests/WebApp.Tests.csproj --configuration $(buildConfiguration) --
no-build --logger trx
      displayName: 'Run tests'
    - task: PublishBuildArtifacts@1
      displayName: 'Publish build artifacts'
      inputs:
        PathtoPublish: '$(Build.SourcesDirectory)/src/WebApp/bin/$(buildConfiguration)/net8.0'
        ArtifactName: 'drop'
        publishLocation: 'Container'
```

Option B: Java (Maven) Sample YAML

```yaml
trigger:
  branches:
    include:
      - main
      - develop

pr:
  branches:
    include:
      - main
      - develop

pool:
  vmImage: 'ubuntu-latest'

variables:
  mavenOptions: '-Xmx1024m'
  mavenPomFile: 'src/webapp/pom.xml'
  mavenGoals: 'clean package'

stages:
- stage: Build
```

```
displayName: Build and Test Java App
jobs:
- job: BuildJob
  steps:
  - task: Maven@4
    displayName: 'Maven build'
    inputs:
      mavenPomFile: '$(mavenPomFile)'
      options: '$(mavenOptions)'
      goals: '$(mavenGoals)'
      publishJUnitResults: true
      testResultsFiles: '**/surefire-reports/TEST-*.xml'
      javaHomeOption: 'JDKVersion'
      jdkVersionOption: '1.11'
      mavenVersionOption: 'Default'
      mavenAuthenticateFeed: false
      effectivePomSkip: true

  - task: PublishBuildArtifacts@1
    displayName: 'Publish build artifacts'
    inputs:
      PathtoPublish: '$(Build.SourcesDirectory)/src/webapp/target'
      ArtifactName: 'drop'
      publishLocation: 'Container'
```

5.3 Step 2: Create Pipeline in Azure DevOps

1. Go to **Pipelines → Pipelines**
2. Click **New Pipeline**
3. Choose **Azure Repos Git** (or GitHub, if applicable)
4. Select your repository
5. When prompted:

    - Choose **Existing Azure Pipelines YAML file**
    - Select /azure-pipelines.yml

6. Save and run

5.4 Step 3: Validate the Pipeline

- Check that:
    - Agent is allocated correctly (hosted windows-latest/ubuntu-latest)
    - Build restores dependencies
    - Build and tests complete successfully
    - Artifacts (drop) are published

5.5 Step 4: Add a Secret (Optional Lab Extension)

1. Go to **Pipelines → Library → Variable groups**
2. Create App-Secrets
3. Add:
    - connectionString (mark as **secret**)
4. Link variable group in YAML:

```
variables:
- group: App-Secrets
```

5. Use in script (only as env variable, don’t echo it):

```
- script: echo "Using connection string in application startup"
  env:
    ConnectionString: $(connectionString)
```