Module 5: Build & Test Automation

1. Tasks and Templates in Azure Pipelines

Azure Pipelines is **task-driven**: each build or deployment is composed of a sequence of tasks. In YAML pipelines, tasks and templates help make configurations **modular, reusable, and maintainable**.

1.1 Tasks

A **task** is the smallest building block in a pipeline job. It represents a single action, such as:

- Restoring dependencies
- Building the solution
- Running tests
- Packaging artifacts
- Deploying to an environment

In YAML, a task is typically written as:

```
- task: TaskName@Version
  displayName: 'Human readable name'
  inputs:
    input1: 'value'
    input2: 'value'
```

Common Task Examples

- **.NET / NuGet**
    - NuGetCommand@2 – restore, pack, push packages
    - DotNetCoreCLI@2 – build, test, publish
- **Node.js / npm**
    - NodeTool@0 – select Node version
    - Npm@1 – npm install, test, publish
- **Java / Maven / Gradle**
    - Maven@4 – build, test, package
    - Gradle@2 – run Gradle tasks
- **Artifacts & Publishing**
    - PublishBuildArtifacts@1 – publish pipeline artifacts

    - PublishPipelineArtifact@1 – pipeline artifact (recommended in modern scenarios)

1.2 Templates

Templates are **reusable YAML fragments** that help you avoid duplication across pipelines. They support **DRY (Don't Repeat Yourself)** practices.

You can create templates for:

- Common build steps
- Standard test & coverage pipelines
- Shared deployment logic

Template Types

1. **Stage templates** – reuse full stages
2. **Job templates** – reuse jobs
3. **Step templates** – reuse common lists of steps

Example: Step Template

File: .azure-pipelines/templates/build-dotnet.yml

```
parameters:
  solution: ''
  buildConfiguration: 'Release'

steps:
- task: DotNetCoreCLI@2
  displayName: 'Restore'
  inputs:
    command: 'restore'
    projects: '${{ parameters.solution }}'

- task: DotNetCoreCLI@2
  displayName: 'Build'
  inputs:
    command: 'build'
    projects: '${{ parameters.solution }}'
    arguments: '--configuration ${{ parameters.buildConfiguration }}'
```

Usage in main pipeline:

```
steps:
- template: .azure-pipelines/templates/build-dotnet.yml
  parameters:
    solution: 'src/WebApp/WebApp.csproj'
    buildConfiguration: 'Release'
```

Benefits:

- Centralized changes (update once, reused everywhere)
- Standardization across teams and services

---

2. NuGet / NPM / Maven Package Restore

Modern applications typically rely on package managers. In CI pipelines, **dependency restore** is a fundamental step before build.

2.1 NuGet Restore (.NET)

You can restore NuGet packages using either **NuGet tasks** or **DotNet CLI**.

Using NuGetCommand@2:

```
- task: NuGetCommand@2
  displayName: 'NuGet restore'
  inputs:
    command: 'restore'
    restoreSolution: 'src/WebApp/WebApp.sln'
```

Using DotNetCoreCLI@2:

```
- task: DotNetCoreCLI@2
  displayName: 'dotnet restore'
  inputs:
    command: 'restore'
    projects: 'src/WebApp/WebApp.csproj'
```

2.2 npm Restore (Node.js)

```
- task: NodeTool@0
  displayName: 'Use Node 18.x'
  inputs:
    versionSpec: '18.x'


- task: Npm@1
  displayName: 'npm install'
  inputs:
    command: 'install'
    workingDir: 'src/webapp'
```

2.3 Maven Restore (Java)

Maven restore is part of the build lifecycle (mvn clean package also resolves dependencies):

```
- task: Maven@4
  displayName: 'Maven build and restore'
  inputs:
    mavenPomFile: 'src/webapp/pom.xml'
    goals: 'clean package'
```

---

3. Unit Testing and Test Results Publishing

One of the key DevOps goals is **shifting testing left**, integrating tests into every CI run.

3.1 Running Unit Tests

.NET Example

```
- task: DotNetCoreCLI@2
  displayName: 'Run unit tests'
  inputs:
    command: 'test'
    projects: 'tests/WebApp.Tests/WebApp.Tests.csproj'
    arguments: '--configuration Release --logger trx'
    publishTestResults: false
```

Note: --logger trx generates test result files (.trx) that can be published.

3.2 Publishing Test Results

Use PublishTestResults@2 to surface test results in Azure DevOps:

```
- task: PublishTestResults@2
  displayName: 'Publish test results'
  inputs:
    testResultsFormat: 'VSTest'
    testResultsFiles: '**/*.trx'
    failTaskOnFailedTests: true
```

Other Formats

- JUnit – common for Java / Node / Python
- NUnit / xUnit – .NET testing frameworks

Example for JUnit (Java/Node):

```
- task: PublishTestResults@2
  displayName: 'Publish JUnit test results'
  inputs:
    testResultsFormat: 'JUnit'
    testResultsFiles: '**/TEST-*.xml'
    failTaskOnFailedTests: true
```

---

4. Code Coverage & Lint Checks

Code coverage and linting help ensure **quality and maintainability**.

4.1 Code Coverage (.NET Example)

Use **Coverlet** with dotnet test and then publish coverage.

```
- task: DotNetCoreCLI@2
  displayName: 'Run tests with coverage'
  inputs:
    command: 'test'
    projects: 'tests/WebApp.Tests/WebApp.Tests.csproj'
    arguments: >
      --configuration Release
      /p:CollectCoverage=true
      /p:CoverletOutput=$(Build.SourcesDirectory)/TestResults/coverage.json
      /p:CoverletOutputFormat=cobertura

- task: PublishCodeCoverageResults@2
  displayName: 'Publish code coverage'
  inputs:
    codeCoverageTool: 'Cobertura'
    summaryFileLocation: '$(Build.SourcesDirectory)/TestResults/coverage.cobertura.xml'
    reportDirectory: '$(Build.SourcesDirectory)/TestResults'
    failIfCoverageEmpty: true
```

You can use tools like **ReportGenerator** to convert coverage outputs if needed.

4.2 Lint Checks (Static Code Quality)

Node.js Example (ESLint):

```
- task: Npm@1
  displayName: 'Run lint'
  inputs:
    command: 'custom'
    workingDir: 'src/webapp'
    customCommand: 'run lint'
```

Assumption: package.json has:

```
"scripts": {
  "lint": "eslint ."
}
```

Java Example (Checkstyle/SpotBugs via Maven):

```
- task: Maven@4
  displayName: 'Run Maven verify with quality plugins'
  inputs:
    mavenPomFile: 'src/webapp/pom.xml'
    goals: 'clean verify'
```

Lint/fail rules can be configured in respective tools.

---

5. Lab: Automated Build + Tests + Artifact Publish

This lab combines all critical aspects:

- Dependency restore
- Build
- Unit tests
- Test results
- (Optional) Code coverage
- Artifact publish

We'll use a **.NET application** as an example, but the structure is easily portable to Java/Node.

5.1 Lab Scenario

You have a solution:

```
src/
  WebApp/
    WebApp.csproj
tests/
  WebApp.Tests/
    WebApp.Tests.csproj
```

Goal:
Set up an **end-to-end CI pipeline** that:

1. Restores NuGet packages
2. Builds the application
3. Runs unit tests
4. Publishes test results
5. Publishes build artifacts for later deployment

5.2 Step 1 – Create azure-pipelines.yml

Place this file at the root of the repository:

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

pr:
  branches:
    include:
      - main
      - develop

pool:
  vmImage: 'windows-latest'

variables:
  buildConfiguration: 'Release'

stages:
- stage: BuildAndTest
  displayName: 'Build, Test & Publish Artifacts'
  jobs:
  - job: BuildJob
    displayName: 'Build and Test Job'
    steps:
    # 1. Restore dependencies
    - task: DotNetCoreCLI@2
      displayName: 'Restore NuGet packages'
      inputs:
```

```
    command: 'restore'
    projects: 'src/WebApp/WebApp.csproj'

# 2. Build solution
- task: DotNetCoreCLI@2
  displayName: 'Build solution'
  inputs:
    command: 'build'
    projects: 'src/WebApp/WebApp.csproj'
    arguments: '--configuration $(buildConfiguration)'
    publishTestResults: false

# 3. Run unit tests
- task: DotNetCoreCLI@2
  displayName: 'Run unit tests'
  inputs:
    command: 'test'
    projects: 'tests/WebApp.Tests/WebApp.Tests.csproj'
    arguments: '--configuration $(buildConfiguration) --logger trx'
    publishTestResults: false

# 4. Publish test results
- task: PublishTestResults@2
  displayName: 'Publish test results'
  inputs:
    testResultsFormat: 'VSTest'
    testResultsFiles: '**/*.trx'
    failTaskOnFailedTests: true

# 5. Publish build artifacts
- task: PublishBuildArtifacts@1
```

```
  displayName: 'Publish build artifacts'
  inputs:
    PathtoPublish: '$(Build.SourcesDirectory)/src/WebApp/bin/$(buildConfiguration)'
    ArtifactName: 'drop'
    publishLocation: 'Container'
```

5.3 Step 2 – Create and Run Pipeline

1. Go to **Pipelines → Pipelines** in Azure DevOps
2. Click **New Pipeline**
3. Select **Azure Repos Git** and your repo
4. Choose **Existing Azure Pipelines YAML file**
5. Pick /azure-pipelines.yml
6. Save and run

5.4 Step 3 – Validate Results

- Check pipeline stages → BuildAndTest
- Confirm:
    - Restore, build, and test tasks are green
    - Test results are visible under **Tests** in pipeline view
    - Artifact drop is created under **Artifacts**

5.5 Optional Extension – Add Code Coverage Step

Add after the test task:

```
- task: DotNetCoreCLI@2
  displayName: 'Run tests with coverage'
  inputs:
    command: 'test'
    projects: 'tests/WebApp.Tests/WebApp.Tests.csproj'
    arguments: >
      --configuration $(buildConfiguration)
      /p:CollectCoverage=true
      /p:CoverletOutput=$(Build.SourcesDirectory)/TestResults/coverage.json
      /p:CoverletOutputFormat=cobertura
    publishTestResults: false

- task: PublishCodeCoverageResults@2
  displayName: 'Publish code coverage'
  inputs:
    codeCoverageTool: 'Cobertura'
    summaryFileLocation: '$(Build.SourcesDirectory)/TestResults/coverage.cobertura.xml'
    reportDirectory: '$(Build.SourcesDirectory)/TestResults'
    failIfCoverageEmpty: true
```