

Lab 3.6

Artifact Promotion and Traceability

Toepasbare Cloud Security voor IT-Professionals

Contents

1. Artifact Promotion and Traceability	3
1.1. Context	3
1.2. Learning goals	3
1.3. Estimated time	3
1.4. Before you start	3
2. Review the Build Stage	5
3. Review Artifact Publishing	6
4. Download and Inspect the Artifact	7
5. Trace Artifact Use in DeployDev	9
6. Create the DeployTest Stage from Scratch	11
6.1. Check Required Test Values	11
6.2. Add Test Variables	11
6.3. Add the New DeployTest Stage	12
6.4. Commit and Run the Pipeline	13
6.5. Verify Test Promotion	14
7. Review Environment-Specific Configuration	15
8. Optional: Add an Approval Gate	17
9. Optional: Rollback Discussion	18
10. Reflection Questions	19
10.1. Question 1	19
10.2. Question 2	19
10.3. Question 3	19
10.4. Question 4	20
11. Summary	21
12. Key Takeaway	22

1. Artifact Promotion and Traceability

1.1. Context

By now the pipeline can provision infrastructure, validate code, scan security risks, build the backend, deploy to dev, and run post-deployment checks.

This lab focuses on one important DevSecOps principle:

“Build once. Test once. Promote the same artifact. Do not rebuild per environment.”

You will first prove that DeployDev uses the artifact produced by the Build stage. Then you will create a new DeployTest stage that promotes that same artifact to a test environment.

Your local workstation is Windows with PowerShell. The Azure DevOps pipeline still runs on Ubuntu agents where the YAML says ubuntu-latest.

1.2. Learning goals

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

- Explain why rebuilding per environment is risky.
- Find the published build artifact in Azure DevOps.
- Verify artifact metadata such as commit SHA and build ID.
- Confirm that deployment stages download the artifact instead of rebuilding.
- Create or provision test infrastructure before deploying to the test environment.
- Create a DeployTest stage that promotes the same artifact used in dev.
- Explain how environment-specific configuration is injected at deployment time.

1.3. Estimated time

75 minutes

Suggested timing:

Time	Activity
10 minutes	Review Build stage artifact creation
10 minutes	Download and inspect the artifact
10 minutes	Trace artifact use in DeployDev
15 minutes	Create or verify test infrastructure
20 minutes	Create and verify the DeployTest stage
10 minutes	Reflection

1.4. Before you start

Make sure you have:

- ☐ A successful pipeline run
- ☐ Build stage completed
- ☐ Dev deployment completed

- [] Access to Azure DevOps pipeline logs and artifacts
- [] Terraform pipeline stage for dev infrastructure available
- [] A Terraform configuration that can be reused for another environment

Important

The `DeployTest` stage does not exist yet. The test infrastructure may also not exist yet. You will first add a `ProvisionTestInfra` stage and then create `DeployTest` to promote the already-built artifact to that test environment.

2. Review the Build Stage

Open `azure-pipelines.yml`.

Find the Build stage.

What stage does Build depend on?

Your answer:

What condition controls whether Build runs?

Your answer:

Find the step that creates metadata files.

Common metadata files include:

File	Purpose
BUILD_VERSION	Version or build version used by the pipeline
BUILD_COMMIT	Git commit SHA that triggered the build
BUILD_NUMBER	Azure DevOps build number or build ID
BUILD_DATE	When the artifact was created
BUILD_BRANCH	Branch that triggered the pipeline

Which metadata files does your pipeline create?

Your answer:

Where are these files written before the artifact is published?

Your answer:

3. Review Artifact Publishing

Find the PublishPipelineArtifact task in the Build stage.

What artifact name is used?

Your answer, for example backend-app:

What path is published?

Your answer:

Does the published artifact include the metadata files? How can you tell?

Your answer:

4. Download and Inspect the Artifact

In Azure DevOps:

1. Open the pipeline run.
2. Go to the run summary.
3. Find the Artifacts section.
4. Download the backend artifact.
5. Extract it locally.

Use PowerShell to list the extracted files:

```
Get-ChildItem -Recurse .\backend-app | Select-Object FullName
```

Open the metadata files:

```
Get-Content .\backend-app\BUILD_COMMIT  
Get-Content .\backend-app\BUILD_VERSION  
Get-Content .\backend-app\BUILD_NUMBER
```

Adjust the path if your artifact extracts to a different folder.

BUILD_COMMIT:

Your answer:

BUILD_VERSION:

Your answer:

BUILD_NUMBER:

Your answer:

Go to Azure DevOps Repos and find the commit SHA from BUILD_COMMIT.

Commit message:

Your answer:

Does this match the commit you expected?

Your answer:

5. Trace Artifact Use in DeployDev

Open the DeployDev stage in `azure-pipelines.yml`.

Find the step that downloads the artifact.

It may use one of these styles:

```
- download: current  
  artifact: backend-app
```

or:

```
- task: DownloadPipelineArtifact@2  
  inputs:  
    artifactName: 'backend-app'
```

What artifact does DeployDev download?

Your answer:

Does DeployDev run `npm install`, `npm run build`, or another build command?

Your answer:

Search inside DeployDev for build-related commands:

```
npm install  
npm ci  
npm run build  
npm run compile  
tsc  
vite build
```

These commands should belong in validation/build stages, not in deployment or promotion stages.

Where does the deployed application code come from?

Your answer:

Open the DeployDev logs in Azure DevOps.

What does the artifact download log say?

Your answer:

6. Create the DeployTest Stage from Scratch

DeployTest is not present in the starter pipeline yet. In this section you create it yourself.

The new stage must deploy the **same** artifact that was already built and deployed to dev. It must not run `npm install`, `npm run build`, or any other rebuild step. Only environment-specific configuration may change.

6.1. Check Required Test Values

You need the test resource group and test App Service name.

These values can come from Terraform outputs, pipeline variables.

Test resource group:

Your answer:

Test App Service name:

Your answer:

Where did these values come from?

Examples: Terraform output, Azure Portal, pipeline variable, instructor handout.

Resource group usage

The `AzureWebApp@1` task deploys to the App Service by name through the configured Azure service connection. The test resource group may not appear directly in the deployment task, but it is still useful when verifying the deployment in the Azure Portal or with Azure CLI.

6.2. Add Test Variables

At the top of `azure-pipelines.yml`, add variables for the test environment if they are not already present.

Example:

```
variables:  
  artifactName: 'backend-app'
```

```
testResourceGroup: '<TEST_RESOURCE_GROUP_NAME>'
testAppServiceName: '<TEST_APP_SERVICE_NAME>'
```

If your pipeline already exposes test values from Terraform outputs, you may use those instead of hardcoding variables.

Artifact name

Use the exact artifact name from the Build stage. If your Build stage publishes `backend-app`, use `backend-app`. If the pipeline already has an `artifactName` variable, reuse that variable instead of creating a second name.

Which variables did you add or reuse?

Your answer:

6.3. Add the New DeployTest Stage

Add a **new** stage after PostDeploymentTests. Do not edit DeployDev into DeployTest; keep both stages so you can compare them. You might want to make a copy of DeployDev.

Figure out what else is needed to deploy to the test stage

Think about the resource group and permissions required. These will need to be added pipeline as well as the Azure environment. Keep it simple for now, you can use the same service connection and permissions as the dev deployment if your test environment is in the same subscription. The key point is that the artifact should be the same, only the environment configuration should change. Also check the global resource names you notice issues?

What stage does DeployTest depend on, and why?

Your answer:

Which artifact does DeployTest download?

Your answer:

Does DeployTest contain any build command?

Your answer:

Search inside DeployTest for build-related commands:

```
npm install
npm ci
npm run build
npm run compile
tsc
vite build
```

The promotion stage should download and deploy the existing artifact. It should not rebuild the app.

6.4. Commit and Run the Pipeline

Commit and push the pipeline change:

```
git add azure-pipelines.yml
git commit -m "feat: add test artifact promotion stage"
git push
```

Open the new pipeline run in Azure DevOps.

Does the newly created DeployTest stage appear?

Your answer:

Does DeployTest start only after PostDeploymentTests completed?

Your answer:

Did DeployTest complete successfully? If not, what failed?

Your answer:

6.5. Verify Test Promotion

Open the DeployTest logs for the stage you just created.

Find the artifact download step.

What artifact and build run were downloaded?

Your answer:

Compare this with the DeployDev artifact download log.

Is the artifact the same as the one deployed to dev?

Your answer:

How can you prove this from the logs or metadata?

Your answer:

7. Review Environment-Specific Configuration

The artifact should contain versioned application code and metadata, not environment-specific secrets or hardcoded environment configuration.

Find the deployment task, usually AzureWebApp@1.

Look for appSettings or environment variables in both DeployDev and DeployTest.

Which settings are injected during the dev deployment?

Paste or summarize the relevant settings:

Which settings are injected during the test deployment?

Paste or summarize the relevant settings:

Where do these values come from?

Examples: pipeline variables, Terraform outputs, Azure DevOps variable groups, or Key Vault references.

Why should these values not be baked into the artifact during build?

Your answer:

8. Optional: Add an Approval Gate

If your instructor asks for it, configure an approval on the Azure DevOps test environment.

In Azure DevOps:

1. Go to Pipelines -> Environments.
2. Open or create the environment named test.
3. Add an approval/check.
4. Add yourself or the instructor as approver.
5. Run the pipeline again and observe where it pauses.

Did the pipeline pause before DeployTest?

Your answer:

Who approved the deployment?

Your answer:

9. Optional: Rollback Discussion

You do not need to implement rollback in this lab, but answer the scenario.

A production issue is discovered after deploying Build #456.

A previous Build #455 is known to be stable.

What would a safe rollback do?

Your answer:

Why is redeploying a previous artifact safer than rebuilding old source code?

Your answer:

10. Reflection Questions

10.1. Question 1

Why is rebuilding separately for dev, test, and production risky?

Your answer:

10.2. Question 2

How does BUILD_COMMIT help during incident response?

Your answer:

10.3. Question 3

Dev and test deploy the same artifact, but the app behaves differently. What could explain the difference?

Your answer:

10.4. Question 4

A hotfix is needed. Should you patch production manually or create a new artifact through the pipeline?

Your answer:

11. Summary

In this lab, you:

- Reviewed how the Build stage creates and publishes an artifact.
- Downloaded the artifact and inspected metadata.
- Verified that DeployDev downloads the artifact instead of rebuilding.
- Created a ProvisionTestInfra stage for the test environment.
- Created a DeployTest stage that promotes the same artifact.
- Compared Dev and Test artifact usage through logs and metadata.
- Reviewed how configuration is injected per environment.

12. Key Takeaway

Commit -> Build -> Tests and scans -> Artifact -> Dev -> Post-deployment tests -> Provision Test -> Test -> Prod

The artifact should stay the same.
Only the environment configuration changes.

Note

This completes the Day 3 DevSecOps lab flow: quality gates, security scans, post-deployment testing, artifact promotion, and traceability.