Module 9 - Security & Monitoring

1. Secrets & Service Connections – Best Practices

In a modern CI/CD setup, **secrets** (passwords, keys, tokens, certificates) and **service connections** (links to external services like Azure, GitHub, container registries) are at the heart of **security**.

Treat your pipeline like production code: **never hard-code secrets** and always apply **least privilege**.

1.1 Where to Store Secrets in Azure DevOps

Recommended options:

1. **Variable Groups (Library)**
    - Can mark individual variables as **secret**
    - Can be linked to specific pipelines
    - Good for:
        - Connection strings
        - API keys
        - Non-rotating or rarely rotating secrets
2. **Azure Key Vault Integration (Best for Enterprise)**
    - Secrets are stored in **Azure Key Vault**, not in DevOps itself
    - Azure DevOps uses a service connection to read them at runtime
    - Good for:
        - Highly sensitive secrets
        - Centralized secret management across many apps
        - Automatic rotation scenarios

Avoid:
Storing secrets in code (YAML, .csproj, .config) or checking them into Git.
Echoing secrets in logs (echo $(password)).

1.2 Secrets in YAML – How to Use Them

Variables from Library or Key Vault show up as $(VAR_NAME).

Example:

variables:

- group: vg-app-secrets # contains DB_PASSWORD as a secret

```
steps:
- script: |
    echo "Starting app migration..."
  displayName: 'Migrate database'
  env:
    DB_PASSWORD: $(DB_PASSWORD)   # used as environment variable only
```

Note: Azure DevOps **masks** secret values in logs (shows as ***).

1.3 Service Connections – Best Practices

Service connections are used to authenticate pipelines to:

- Azure Resource Manager
- Docker registries
- GitHub, Bitbucket
- SonarCloud, Snyk, etc.

General Guidelines:

- Use **Service Principal-based Azure RM connections**, not user accounts.
- Scope them at **resource group** level when possible, not whole subscription.
- Assign the **least privilege role** (Contributor/Reader/custom role) required.
- Use **Managed Identity** where applicable (self-hosted agents in Azure).
- Use clear naming conventions, e.g.:
    - sc-azure-dev-rg-app
    - sc-acr-prod
    - sc-sonarcloud-org-xyz

1.4 Secure Usage Patterns

- **Separate service connections per environment**:
  sc-azure-dev, sc-azure-qa, sc-azure-prod
  → easier to enforce tighter access for production.
- **Lock down who can modify service connections**:
  Only DevOps admins or platform team, not every developer.

- **Review service connection usage regularly**:
  Which pipelines use which connections, especially for production.

---

2. Integrating SonarCloud / Snyk for Code Analysis

Security and quality are **non-negotiable** in modern pipelines. SonarCloud and Snyk are popular SaaS tools for:

- **SonarCloud** → Code quality & static analysis (bugs, smells, coverage)
- **Snyk** → Dependency and container vulnerability scanning

2.1 SonarCloud Integration – Overview

SonarCloud provides:

- Code smells, bug detection
- Code coverage visualization
- Quality gates (pass/fail pipeline based on conditions)

Integration Steps (high-level):

1. Create account + organization in **SonarCloud**
2. Install **SonarCloud extension** in Azure DevOps
3. Create a **SonarCloud service connection** in Azure DevOps
4. Add Sonar tasks to pipeline:
    - **Prepare Analysis Configuration**
    - **Run Build & Tests**
    - **Run Code Analysis**

Typical YAML Snippet (for .NET):

```
steps:
- task: SonarCloudPrepare@1
  displayName: 'Prepare SonarCloud analysis'
  inputs:
    SonarCloud: 'sc-sonarcloud-org'
    organization: 'my-org'
    scannerMode: 'MSBuild'
    projectKey: 'my-org_my-project'
    projectName: 'My Project'
```

```
- task: DotNetCoreCLI@2
  displayName: 'Build & Test'
  inputs:
    command: 'test'
    projects: '**/*Tests.csproj'
    arguments: '--configuration Release /p:CollectCoverage=true'

- task: SonarCloudAnalyze@1
  displayName: 'Run SonarCloud analysis'

- task: SonarCloudPublish@1
  displayName: 'Publish SonarCloud quality gate'
  inputs:
    pollingTimeoutSec: '300'
```

You can **fail the build** if the quality gate fails, enforcing minimum standards.

2.2 Snyk Integration – Overview

Snyk focuses on:

- Vulnerabilities in open-source dependencies
- Container image scans
- IaC (Terraform, ARM, Kubernetes) security misconfigurations

Integration Concepts:

- Snyk uses **API token** stored as secret in Azure DevOps
- Pipeline runs snyk test or snyk monitor against:
    - package.json, pom.xml, requirements.txt, etc.
    - Docker images
    - Terraform/ARM manifests

Typical YAML Example (Node.js):

```
steps:
- task: NodeTool@0
```

```
  inputs:
    versionSpec: '18.x'

- script: |
    npm install
  displayName: 'Install dependencies'

- script: |
    npx snyk test --severity-threshold=medium
  displayName: 'Snyk test'
  env:
    SNYK_TOKEN: $(SNYK_TOKEN)
```

You can choose to:

- **Fail the build** on high severity vulnerabilities
- **Warn only** for lower severities

2.3 Best Practices for Code Analysis Integration

- Treat analysis as **part of CI**, not a separate job someone "might run".
- Set **clear quality policies**:
    - No new critical vulnerabilities
    - Minimum coverage (e.g., 80% for new code)
    - No code smells of certain severity
- Combine:
    - **SonarCloud** for code quality + coverage
    - **Snyk** for dependency and container vulnerabilities

---

3. Build & Release Auditing / Approvals

Azure DevOps provides rich auditing and approval capabilities to answer:

Who deployed **what**, **where**, and **when**?

3.1 Approvals in Pipelines

Approvals are configured at:

- **Environments** (recommended)
- **Classic Release Approvals** (legacy model)

For multi-stage YAML:

- Add environments: Dev, QA, Prod
- Attach approvals only to sensitive ones (Prod, maybe UAT)

Environment Example:

```
- stage: Deploy_Prod
  jobs:
  - deployment: DeployProd
    environment: 'Prod'   # Environment has manual approval configured
    strategy:
      runOnce:
        deploy:
          steps:
          - task: AzureWebApp@1
```

In DevOps UI, configure:

Environment → Approvals and Checks → Add Approver(s)

3.2 Auditing – What's Captured?

Azure DevOps provides logs of:

- Pipeline runs (who triggered, what branch, what commits)
- Approvals (who approved, at what time, comments)
- Changes to:
    - Pipelines (YAML history via Git)
    - Service connections
    - Permissions

For organizations with compliance needs (BFSI, healthcare, government), this audit trail is critical.

3.3 Governance Best Practices

- **Branch Protection + PR Policies** for main/master branch

- **Mandatory PR reviews** before merging to release branches
- **Build validation** as branch policy
- **Environment approvals** for Prod deployments
- **Separate roles**:
    - Developers: create PRs, pipelines
    - Leads/Managers: approve prod deployments
    - DevOps/Platform team: manage service connections, infra

---

4. Monitoring Pipeline Runs & Troubleshooting

Monitoring pipeline runs is about answering:

- Are builds and releases **healthy**?
- Where are **bottlenecks**?
- Why did a pipeline **fail**?

4.1 Monitoring Day-to-Day health

Key places in Azure DevOps:

1. **Pipelines → Runs**
    - See list of recent builds/releases
    - Filter by status (failed, succeeded, cancelled)
2. **Pipeline Analytics**
    - Average duration
    - Failure rate
    - Stage-level timings
3. **Dashboards**
    - Add widgets:
        - Build history
        - Release status
        - Failed runs chart

4.2 Common Failure Categories

1. **Code Issues**

- Compilation failures
- Failing unit tests
- Quality gate failures (Sonar, Snyk, etc.)

2. **Configuration Issues**
   - Wrong Azure subscription/resource name
   - Invalid paths or script errors
   - Missing variables or secrets
3. **Service Issues**
   - Expired PAT / Service principal secret
   - Revoked RBAC access
   - Network / firewall blocks
4. **Agent Issues**
   - Hosted agent image changes
   - Missing tools on self-hosted agent
   - Disk space issues

4.3 Troubleshooting Steps – A Practical Approach

When a pipeline fails:

1. **Look at the failing stage & task**
   - Azure DevOps highlights the exact step and error.
2. **Expand logs with "View raw logs"**
   - Get full error stack traces for script/CLI commands.
3. **Rerun failed jobs or stages**
   - Use "Rerun failed jobs" for transient issues.
4. **Test commands locally**
   - Copy the failing script/command
   - Run locally on dev machine or same OS as the agent.
5. **Validate access / credentials**
   - For Azure: az account show
   - For service connections: verify permissions, scope, expiration.
6. **Check recent changes**

    - Was YAML or variable group modified?
    - Any new service connection or secret rotation?

4.4 Observability for Pipelines

- Export pipeline logs/metrics to:
    - **Application Insights**
    - **Log Analytics**
    - **Splunk/ELK / Dynatrace**
- Track:
    - Build frequency
    - Deployment frequency
    - Mean Time To Recovery (MTTR)
    - Change failure rate

This aligns with **DORA metrics** for DevOps.