Complete CI Tutorial: GitHub Actions DevSecOps Pipeline (Java + Docker)

This tutorial is based on: https://github.com/vilasvarghesescaler/docker-k8s/blob/master/.github/workflows/ci.yml

This tutorial explains **how to design, implement, and understand** a real-world **CI pipeline using GitHub Actions**, based on the workflow you provided.

By the end, you will understand:

- How CI works in GitHub Actions
- Why each stage exists
- How security is integrated (DevSecOps)
- How Docker images are built, tested, scanned, and pushed

What are we building?

We are building a **production-style Continuous Integration pipeline** that automatically:

- Pulls source code
- Builds a Java application
- Runs linting and unit tests
- Performs code security scanning (SAST)
- Performs dependency scanning (SCA)
- Builds a Docker image
- Scans the container for vulnerabilities
- Runs the container and tests it
- Pushes a verified image to DockerHub

CI Architecture Overview

```
Developer Push → GitHub Actions Runner →
Checkout → Build → Test → Security Scan →
Docker Build → Image Scan → Container Test → Push to Registry
```

This pipeline follows **DevSecOps principles** — security is applied **before** the software is shipped.

Prerequisites

Before using this pipeline, you need:

Application

- Java Maven project

- Working Dockerfile
- App running on port 8080

DockerHub Account

Create:

- DockerHub username
- DockerHub access token

GitHub Secrets (Mandatory)

Go to:

`GitHub Repo → Settings → Secrets and variables → Actions → New repository secret`

Create:

Secret Name	Value
DOCKERHUB_USERNAME	your_dockerhub_username
DOCKERHUB_TOKEN	dockerhub_access_token

These secrets securely authenticate your pipeline.

Pipeline Trigger

```
on:
  push:
    branches:
      - master
  workflow_dispatch:
```

- What this does

- Runs automatically on every push to `master`
- Can also be run manually from GitHub UI

This enables **continuous integration**.

Job Configuration

```
jobs:
  ci-pipeline:
    runs-on: ubuntu-latest
```

Your entire CI runs on a **fresh Linux virtual machine** provided by GitHub.

Permissions

```
permissions:
  contents: read
  security-events: write
```

This allows the pipeline to:

- Read source code
- Upload vulnerability reports to GitHub Security tab

Pipeline Stages Explained

- 1. Checkout Source Code

```
- uses: actions/checkout@v4
```

Downloads your repository into the runner.

Without this step → nothing to build.

---

2. Setup Java

```
- uses: actions/setup-java@v3
```

Installs:

- Java 11
- Maven
- Dependency caching

Speeds up builds and ensures version consistency.

---

3. Linting (Code Quality)

```
mvn checkstyle:check
```

Purpose:

- Enforces coding standards
- Detects bad practices early

`continue-on-error: true` means:

- Pipeline continues
- But violations are visible

Used to control technical debt.

---

4. SAST – CodeQL

```
github/codeql-action
```

Detects:

- SQL injection
- Command injection
- Insecure deserialization
- OWASP Top 10 issues

This scans **source code itself**.

Prevents insecure code from entering production.

---

5. SCA – OWASP Dependency Check

```
dependency-check/Dependency-Check_Action
```

Finds vulnerabilities in:

- Maven libraries
- Open-source dependencies

Protects against **supply chain attacks**.

---

6. Unit Testing

```
mvn test
```

Validates:

- Business logic
- Functional correctness

Pipeline fails if tests fail.

📌 Prevents broken builds.

---

7. Build Application

```
mvn clean package -DskipTests
```

Creates:

- Compiled JAR/WAR
- Ready for Docker packaging

Separates testing from packaging.

---

8. Docker Image Build

```
docker build -t username/test:latest .
```

Creates immutable application image.

This ensures **environment consistency**.

---

9. Trivy Image Scan

```
aquasecurity/trivy-action
```

Detects vulnerabilities in:

- Linux OS packages
- Java libraries
- CVEs

`exit-code: 1` → pipeline fails if critical/high vulnerabilities exist.

Prevents insecure containers from being shipped.

Upload Scan Results

```
github/codeql-action/upload-sarif
```

Uploads Trivy findings to:

```
GitHub → Security → Code scanning alerts
```

Enables centralized vulnerability tracking.

---

10. Container Runtime Testing

```
docker run …
curl http://localhost:8080
```

Verifies:

- Container boots
- App responds
- No runtime crash

This is a **smoke test**.

---

11. DockerHub Login

```
docker/login-action@v3
```

Uses secrets to authenticate securely.

Prevents hard-coding credentials.

---

12. Push Docker Image

```
docker push username/test:latest
```

Publishes trusted image.

Enables deployment pipelines (CD).

What happens when code is pushed?

1. Developer pushes code
2. GitHub Actions spins up VM
3. Code is built & tested
4. Security scans execute
5. Docker image is built
6. Image is scanned
7. Container tested
8. Image pushed if all checks pass

DevOps & Security Concepts Demonstrated

- Continuous Integration
- Shift-left security
- DevSecOps
- Supply chain protection
- Immutable artifacts
- Quality gates
- Infrastructure-as-code
- Zero-trust credentials

What makes this pipeline “industry-grade”?

✔ Multi-layered security
✔ Code + dependency + container scanning
✔ Failing on critical vulnerabilities
✔ Runtime verification
✔ Artifact promotion
✔ Secure secrets handling
✔ GitHub Security integration

Recommended Extensions

- Add SonarQube quality gates
- Add SBOM generation

- Push to AWS ECR
- Add CD to EKS/ECS
- Slack alerts
- Artifact versioning
- Infrastructure pipeline

Final takeaway

This pipeline is not “just CI.”

It is a **DevSecOps quality gate system** that ensures:

> Only tested, scanned, verified, and trusted software is allowed to move forward.