It is a sample java project which uses maven a build tool to build war artifact and creates a docker image. Before deploying war artifact into GitHub packages and pushing docker image to docker hub, we run unit test and sonar check for quality and validation.

Also, there’s implementation of bumping pom version for feature changes and raise pull request after checks are done. Artifact and images deployment is done on pull request.

**build\_test.yaml**

The code represents a GitHub Actions workflow called "Build and Test". It is triggered whenever a push event occurs on branches that match the patterns 'feature\*' or 'feature/\*'. The workflow includes several jobs:

1. Test
2. Sonar
3. Updatepom
4. raisepr.

The **TEST** job is responsible for running unit tests. It runs on the latest version of Ubuntu and includes the following steps:

1. Checkout the repository using the actions/checkout@v3 action.
2. Setup JDK 11 using the actions/setup-java@v3 action.
3. Execute Maven package command to build the project and skip running tests.
4. Execute Maven verify command to perform additional verification and run tests.

The **SONAR** job is responsible for scanning the code using Sonar Cloud. It runs on the latest version of Ubuntu and includes the following steps:

1. Checkout the repository using the actions/checkout@v3 action.
2. Setup JDK 11 using the actions/setup-java@v3 action.
3. Execute Maven verify command with Sonar profile and specify the Sonar token from secrets to perform the scan.
4. Set the GITHUB\_TOKEN environment variable to access GitHub API during the scan.

The **UPDATEPOM** job is responsible for bumping the version number in the POM file. It runs on the latest version of Ubuntu and has a dependency on the test and sonar jobs. It includes the following steps:

1. Checkout the repository using the actions/checkout@v3 action.
2. Install Python dependencies.
3. Run a Python script to update the POM version.
4. Configure git user details.
5. Add changes, commit them with a message, and push them to the repository.

The **RAISEPR** job is responsible for creating a pull request to the main branch. It runs on the latest version of Ubuntu and has a dependency on the updatepom job. It includes the following steps:

1. Checkout the repository using the actions/checkout@v3 action.
2. Create a pull request using the GitHub CLI (gh) command. The pull request is created with the title "Merge feature into main branch" and a body indicating that it was created by a GitHub action.
3. Set the GITHUB\_TOKEN environment variable to access GitHub API during the pull request creation.

**deployment.yaml**

The code provided is a GitHub Actions workflow named "Deployment". It is triggered when a pull request is opened or reopened. The workflow consists of two jobs:

1. Artifact
2. Docker
3. deploy-EC2

The "**ARTIFACT**" job is responsible for publishing an artifact to GitHub Packages. It runs on the latest version of Ubuntu and includes the following steps:

1. Checkout the repository using the actions/checkout@v3 action.
2. Set up JDK 11 using the actions/setup-java@v3 action. The distribution is set to 'temurin' and the Java version is set to 11.
3. Publish the artifact on GitHub Packages using the Maven command mvn -B clean deploy -DskipTests. The environment variable GITHUB\_TOKEN is used for authentication.

The "**DOCKER**" job is responsible for publishing a Docker image to Docker Hub. It also runs on the latest version of Ubuntu and depends on the completion of the "artifact" job. It includes the following steps:

1. Checkout the repository using the actions/checkout@v1 action.
2. Login to Docker Hub using the docker login command with the Docker Hub username and password stored in the GitHub secrets.
3. Build the Docker image using the docker build command. The image is tagged with the latest version and the first 8 characters of the GitHub SHA.
4. Publish the Docker image using the docker push command. The repository to push to is stored in the DOCKER\_REPO secret.

The “**deploy-EC2”** job is responsible for deploying the application to an EC2 instance using Ansible. It runs on a self-hosted runner (runs on ec2 instance) and depends on the completion of the docker job. The steps in this job are:

1. Checking out the code using the actions/checkout@v3 action.
2. Setting an environment variable REPO\_NAME with the value of the repository name obtained from the GitHub event.
3. Running the Ansible playbook deploy\_app.yaml with the ansible-playbook command, passing the repo\_name as an extra variable

Configuring EC2 as runner for GitHub actions:

In your gut hub Repository go to Settings>>Actions>>Runners>>New self-hosted runner. Follow the instructions to setup as runner.

Also, ansible is configured on the runner to execute playbooks.

**Ansible playbook (deploy\_app.yaml):**

This code is an Ansible playbook that automates the creation and configuration of an Amazon EC2 instance.

1. Starts an EC2 instance with a public IP address using the Amazon AWS EC2 module. AMI is preconfigured with docker, maven, java and ssh connection to ec2 runner (ansible control node)
2. Registers the public IP address of the created instance as a variable named "ec2\_ip".
3. Displays the value of "ec2\_ip" using the debug module.
4. Adds the created EC2 instance to the "just\_created" group using the add\_host module.
5. Starts a new play targeting the "just\_created" group.
6. Waits for SSH to come up on the EC2 instance using the wait\_for\_connection module.
7. Copies a Docker compose file to the remote EC2 instance using the copy module.
8. Runs the "docker-compose up -d" command on the remote EC2 instance using the shell module.

**Migrating from Jenkins to GitHub actions:**

Follow the instructions from below link:

<https://docs.github.com/en/actions/migrating-to-github-actions/automated-migrations/migrating-from-jenkins-with-github-actions-importer>

Summarized steps:

1. Jenkins server
2. Jenkins API token
3. GitHub personal access token with workflow and repo permissions
4. Linux/ windows server with docker and GitHub cli installed:
   1. Instructions for GitHub cli

<https://github.com/cli/cli#installation>

1. Authorize your git for GitHub cli:

gh auth login

1. Install GitHub actions importer cli extension:

gh extension install github/gh-actions-importer

1. verify installation:

gh actions-importer -h

1. configuring creds:
   1. gh actions-importer configure:
      1. scroll using arrows.
      2. select Jenkins using space.
      3. click enter.
      4. update base URL for GitHub
      5. update GitHub token
      6. update Jenkins URL
      7. update Jenkins token
2. Run the GitHub Actions Importer update CLI command to connect to GitHub Packages Container registry and ensure that the container image is updated to the latest version:

gh actions-importer update

1. Perform audit for Jenkins:

gh actions-importer audit jenkins --output-dir tmp/audit

1. Above command
   1. Fetches all of the projects defined in a Jenkins server.
   2. Converts each pipeline to its equivalent GitHub Actions workflow.
   3. Generates a report that summarizes how complete and complex of a migration is possible with GitHub Actions Importer.

For manual migration:

<https://docs.github.com/en/actions/migrating-to-github-actions/manual-migrations/migrating-from-jenkins-to-github-actions>

**Migrating Source Code Repository from Gitlab to GitHub:**

* Using GitHub import repository tool over the UI

It requires:

1. URL of source repository (Gitlab)
2. credentials to read GitLab repository.

this doesn’t work when Gitlab is hosted on private network and importer tool doesn’t have access to code.

* Using CLI
  1. Create a new repository in GitHub with same name as GitLab
  2. Clone the repository of GitLab into local

git clone <GitLab URL>

cd <repo name>

* 1. Push your GitLab repository to newly create GitHub repository

git push –mirror <GitHub URL>