Building and testing Java with Maven

In this article

Introduction

Prerequisites

Using the Maven starter workflow

Building and testing your code

Packaging workflow data as artifacts

You can create a continuous integration (CI) workflow in GitHub Actions to build and test your Java project with Maven.

Note: GitHub-hosted runners are not currently supported on GitHub Enterprise Server. You can see more information about planned future support on the GitHub public roadmap.

Introduction

This guide shows you how to create a workflow that performs continuous integration (CI) for your Java project using the Maven software project management tool. The workflow you create will allow you to see when commits to a pull request cause build or test failures against your default branch; this approach can help ensure that your code is always healthy. You can extend your CI workflow to upload artifacts from a workflow run.

GitHub-hosted runners have a tools cache with pre-installed software, which includes Java Development Kits (JDKs) and Maven. For a list of software and the pre-installed versions for JDK and Maven, see "About GitHub-hosted runners".

Prerequisites

You should be familiar with YAML and the syntax for GitHub Actions. For more information, see:

- "Workflow syntax for GitHub Actions"
- "Learn GitHub Actions"

We recommend that you have a basic understanding of Java and the Maven framework. For more information, see the Maven Getting Started Guide in the Maven documentation.

Using self-hosted runners on GitHub Enterprise Server

When using setup actions (such as `actions/setup-LANGUAGE`) on GitHub Enterprise Server with self-hosted runners, you might need to set up the tools cache on runners that do not have internet access. For more information, see "Setting up the tool cache on self-hosted

runners without internet access."

Using the Maven starter workflow

GitHub provides a Maven starter workflow that will work for most Maven-based Java projects. For more information, see the Maven starter workflow.

To get started quickly, you can choose the preconfigured Maven starter workflow when you create a new workflow. For more information, see the "Quickstart for GitHub Actions."

You can also add this workflow manually by creating a new file in the `.github/workflows` directory of your repository.

YAML

```yaml
name: Java CI

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2
      - name: Set up JDK 17
        uses: actions/setup-java@v2
        with:
          java-version: '17'
          distribution: 'temurin'
      - name: Build with Maven
        run: mvn --batch-mode --update-snapshots package
```

This workflow performs the following steps:

1. The `checkout` step downloads a copy of your repository on the runner.
2. The `setup-java` step configures the Eclipse Temurin (Java) 17 JDK by Eclipse Adoptium.
3. The "Build with Maven" step runs the Maven `package` target in non-interactive mode to ensure that your code builds, tests pass, and a package can be created.

The default starter workflows are excellent starting points when creating your build and test workflow, and you can customize the starter workflow to suit your project's needs.

Running on a different operating system

The starter workflow configures jobs to run on Linux, using the GitHub-hosted `ubuntu-latest` runners. You can change the `runs-on` key to run your jobs on a different operating system. For example, you can use the GitHub-hosted Windows runners.

```yaml
runs-on: windows-latest
```

Or, you can run on the GitHub-hosted macOS runners.

```yaml
runs-on: macos-latest
```

You can also run jobs in Docker containers, or you can provide a self-hosted runner that runs on your own infrastructure. For more information, see "Workflow syntax for GitHub Actions."

Specifying the JVM version and architecture

The starter workflow sets up the `PATH` to contain OpenJDK 8 for the x64 platform. If you want to use a different version of Java, or target a different architecture (`x64` or `x86`), you can use the `setup-java` action to choose a different Java runtime environment.

For example, to use version 11 of the JDK provided by Adoptium for the x64 platform, you can use the `setup-java` action and configure the `java-version`, `distribution` and `architecture` parameters to `'11'`, `'adopt'` and `x64`.

YAML

```yaml
steps:
  - uses: actions/checkout@v2
  - name: Set up JDK 11 for x64
    uses: actions/setup-java@v2
    with:
      java-version: '11'
      distribution: 'adopt'
      architecture: x64
```

For more information, see the `setup-java` action.

Building and testing your code

You can use the same commands that you use locally to build and test your code.

The starter workflow will run the `package` target by default. In the default Maven configuration, this command will download dependencies, build classes, run tests, and package classes into their distributable format, for example, a JAR file.

If you use different commands to build your project, or you want to use a different target, you can specify those. For example, you may want to run the `verify` target that's configured in a *pom-ci.xml* file.

YAML

```yaml
steps:
  - uses: actions/checkout@v2
  - uses: actions/setup-java@v2
    with:
      java-version: '17'
      distribution: 'temurin'
  - name: Run the Maven verify phase
    run: mvn --batch-mode --update-snapshots verify
```

Packaging workflow data as artifacts

After your build has succeeded and your tests have passed, you may want to upload the resulting Java packages as a build artifact. This will store the built packages as part of the workflow run, and allow you to download them. Artifacts can help you test and debug pull requests in your local environment before they're merged. For more information, see "Storing workflow data as artifacts."

Maven will usually create output files like JARs, EARs, or WARs in the `target` directory. To upload those as artifacts, you can copy them into a new directory that contains artifacts to upload. For example, you can create a directory called `staging`. Then you can upload the contents of that directory using the `upload-artifact` action.

YAML

```yaml
steps:
  - uses: actions/checkout@v2
  - uses: actions/setup-java@v2
    with:
      java-version: '17'
      distribution: 'temurin'
  - run: mvn --batch-mode --update-snapshots verify
  - run: mkdir staging && cp target/*.jar staging
  - uses: actions/upload-artifact@v2
    with:
      name: Package
      path: staging
```