# Building and testing Java with Maven

**In this article**



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 cache files and 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 "Using 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

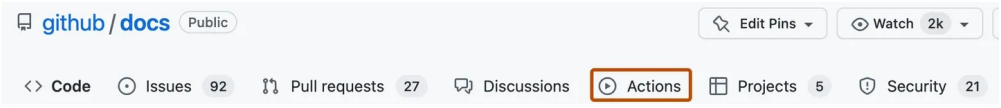
When using setup actions (such as `actions/setup-LANGUAGE`) on GitHub Enterprise Server with self-hosted runners, you might need to set up the tools cache on runners that do not have internet access. For more information, see "Setting up the tool cache on self-hosted runners without internet access."

## Using a Maven starter workflow

To get started quickly, add a starter workflow to the `.github/workflows` directory of your repository.

GitHub provides a starter workflow for Maven that should work for most Java with Maven projects. The subsequent sections of this guide give examples of how you can customize this starter workflow.

1. On your GitHub Enterprise Server instance, navigate to the main page of the repository.
2. Under your repository name, click **Actions**.




3. If you already have a workflow in your repository, click **New workflow**.
4. The "Choose a workflow" page shows a selection of recommended starter workflows. Search for "Java with Maven".
5. On the "Java with Maven" workflow, click **Configure**.

   If you don't find the "Java with Maven" starter workflow, copy the following workflow code to a new file called `maven.yml` in the `.github/workflows` directory of your repository.

YAML

```yaml
name: Java CI with Maven

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - name: Set up JDK 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'
        cache: maven
    - name: Build with Maven
      run: mvn -B package --file pom.xml

    # Optional: Uploads the full dependency graph to GitHub to improve the
quality of Dependabot alerts this repository can receive
    - name: Update dependency graph
      uses: advanced-security/maven-dependency-submission-
action@571e99aab1055c2e71a1e2309b9691de18d6b7d6
```

6. Edit the workflow as required. For example, change the Java version.

7. Click **Commit changes**.

### Specifying the Java 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'`, `'temurin'` and `x64`.

YAML

```yaml
steps:
  - uses: actions/checkout@v4
  - name: Set up JDK 11 for x64
    uses: actions/setup-java@v3
    with:
      java-version: '11'
      distribution: 'temurin'
      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@v4
  - uses: actions/setup-java@v3
    with:
      java-version: '17'
      distribution: 'temurin'
  - name: Run the Maven verify phase
    run: mvn --batch-mode --update-snapshots verify
```

## Caching dependencies

You can cache your dependencies to speed up your workflow runs. After a successful run, your local Maven repository will be stored in a cache. In future workflow runs, the cache will be restored so that dependencies don't need to be downloaded from remote Maven repositories. You can cache dependencies simply using the `setup-java` action or can use `cache` action for custom and more advanced configuration.

YAML

```yaml
steps:
  - uses: actions/checkout@v4
  - name: Set up JDK 11
    uses: actions/setup-java@v3
    with:
      java-version: '17'
      distribution: 'temurin'
      cache: maven
  - name: Build with Maven
    run: mvn --batch-mode --update-snapshots verify
```

This workflow will save the contents of your local Maven repository, located in the `.m2` directory of the runner's home directory. The cache key will be the hashed contents of *pom.xml*, so changes to *pom.xml* will invalidate the cache.

## 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@v4
  - uses: actions/setup-java@v3
    with:
      java-version: '17'
      distribution: 'temurin'
  - run: mvn --batch-mode --update-snapshots verify
  - run: mkdir staging && cp target/*.jar staging
  - uses: actions/upload-artifact@v3
    with:
      name: Package
      path: staging
```