Building and testing Java with Gradle

In this article

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

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 Gradle build system. 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 Gradle. For a list of software and the pre-installed versions for JDK and Gradle, 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 Gradle framework. For more information, see the Gradle User Manual.

Using self-hosted runners on GitHub Enterprise Server

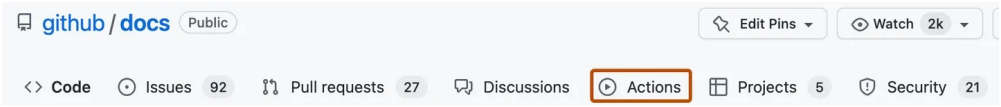
When using setup actions (such as `actions/setup-LANGUAGE`) on GitHub Enterprise Server with self-hosted runners, you might need to set up the tools cache on runners that do not have internet access. For more information, see "Setting up the tool cache on self-hosted runners without internet access."

Using a Gradle starter workflow

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

GitHub provides a starter workflow for Gradle that should work for most Java with Gradle 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 Gradle".
5. On the "Java with Gradle" workflow, click **Configure**.

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

YAML

```yaml
name: Java CI with Gradle

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

permissions:
  contents: read

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v4
    - name: Set up JDK 11
      uses: actions/setup-java@v3
      with:
        java-version: '11'
        distribution: 'temurin'
    - name: Build with Gradle
      uses: gradle/gradle-build-
action@bd5760595778326ba7f1441bcf7e88b49de61a25 # v2.6.0
      with:
        arguments: build
```

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

> **Notes**:

- This starter workflow contains an action that is not certified by GitHub. Actions provided by third parties are governed by separate terms of service, privacy policy, and support documentation.
- If you use actions from third parties you should use a version specified by a commit SHA. If the action is revised and you want to use the newer version, you will need to update the SHA. You can specify a version by referencing a tag or a branch, however the action may change without warning. For more information, see "Security hardening for GitHub Actions."

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 `build` task by default. In the default Gradle 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 task, you can specify those. For example, you may want to run the `package` task that's configured in your *ci.gradle* file.

YAML

```yaml
steps:
  - uses: actions/checkout@v4
  - uses: actions/setup-java@v3
    with:
      java-version: '17'
      distribution: 'temurin'
  - name: Validate Gradle wrapper
    uses: gradle/wrapper-validation-
action@ccb4328a959376b642e027874838f60f8e596de3
  - name: Run the Gradle package task
    uses: gradle/gradle-build-action@749f47bda3e44aa060e82d7b3ef7e40d953bd629
```

```
    with:
      arguments: -b ci.gradle package
```

Caching dependencies

Your build dependencies can be cached to speed up your workflow runs. After a successful run, the `gradle/gradle-build-action` caches important parts of the Gradle user home directory. In future jobs, the cache will be restored so that build scripts won't need to be recompiled and dependencies won't need to be downloaded from remote package repositories.

Caching is enabled by default when using the `gradle/gradle-build-action` action. For more information, see `gradle/gradle-build-action`.

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."

Gradle will usually create output files like JARs, EARs, or WARs in the `build/libs` directory. 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'
  - name: Validate Gradle wrapper
    uses: gradle/wrapper-validation-
action@ccb4328a959376b642e027874838f60f8e596de3
  - name: Build with Gradle
    uses: gradle/gradle-build-action@749f47bda3e44aa060e82d7b3ef7e40d953bd629
    with:
      arguments: build
  - uses: actions/upload-artifact@v3
    with:
      name: Package
      path: build/libs
```