This version of GitHub Enterprise was discontinued on 2023-03-15. No patch releases will be made, even for critical security issues. For better performance, improved security, and new features, upgrade to the latest version of GitHub Enterprise. For help with the upgrade, contact GitHub Enterprise support.

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 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 "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 Gradle framework. For more information, see Getting Started in the Gradle 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

Using the Gradle starter workflow 🔗

GitHub provides a Gradle starter workflow that will work for most Gradle-based Java projects. For more information, see the [Gradle starter workflow](#).

To get started quickly, you can choose the preconfigured Gradle 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
# This workflow uses actions that are not certified by GitHub.
# They are provided by a third-party and are governed by
# separate terms of service, privacy policy, and support
# documentation.

# GitHub recommends pinning actions to a commit SHA.
# To get a newer version, you will need to update the SHA.
# You can also reference a tag or branch, but the action may change without warning

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: Validate Gradle wrapper
        uses: gradle/wrapper-validation-action@ccb4328a959376b642e027874838f60f8e59(
      - name: Build with Gradle
        uses: gradle/gradle-build-action@749f47bda3e44aa060e82d7b3ef7e40d953bd629
        with:
          arguments: build
```

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 "Validate Gradle wrapper" step validates the checksums of Gradle Wrapper JAR files present in the source tree.

4. The "Build with Gradle" step does a build using the `gradle/gradle-build-action` action provided by the Gradle organization on GitHub. The action takes care of invoking Gradle, collecting results, and caching state between jobs. For more information see [gradle/gradle-build-action](#).

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.

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

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

```
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
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 `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
steps:
```

```yaml
    - uses: actions/checkout@v2
    - uses: actions/setup-java@v2
      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
```

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@v2
  - uses: actions/setup-java@v2
    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@v2
    with:
      name: Package
      path: build/libs
```