Building and testing .NET

In this article

You can create a continuous integration (CI) workflow to build and test your .NET project.

Introduction

This guide shows you how to build, test, and publish a .NET package.

GitHub-hosted runners have a tools cache with preinstalled software, which includes the .NET Core SDK. For a full list of up-to-date software and the preinstalled versions of .NET Core SDK, see software installed on GitHub-hosted runners.

Prerequisites

You should already be familiar with YAML syntax and how it's used with GitHub Actions. For more information, see "Workflow syntax for GitHub Actions."

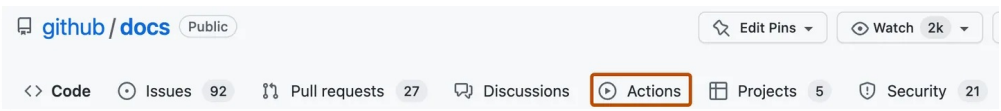
We recommend that you have a basic understanding of the .NET Core SDK. For more information, see Getting started with .NET.

Using a .NET starter workflow

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

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

1. On GitHub.com, 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 "dotnet".
5. On the ".NET" workflow, click **Configure**.
6. Edit the workflow as required. For example, change the .NET version.
7. Click **Commit changes**.

   The `dotnet.yml` workflow file is added to the `.github/workflows` directory of your repository.

Specifying a .NET version

To use a preinstalled version of the .NET Core SDK on a GitHub-hosted runner, use the `setup-dotnet` action. This action finds a specific version of .NET from the tools cache on each runner, and adds the necessary binaries to `PATH`. These changes will persist for the remainder of the job.

The `setup-dotnet` action is the recommended way of using .NET with GitHub Actions, because it ensures consistent behavior across different runners and different versions of .NET. If you are using a self-hosted runner, you must install .NET and add it to `PATH`. For more information, see the `setup-dotnet` action.

Using multiple .NET versions

```yaml
name: dotnet package

on: [push]

jobs:
  build:

    runs-on: ubuntu-latest
    strategy:
      matrix:
        dotnet-version: [ '3.1.x', '6.0.x' ]

    steps:
      - uses: actions/checkout@v4
      - name: Setup dotnet ${{ matrix.dotnet-version }}
        uses: actions/setup-dotnet@v3
        with:
          dotnet-version: ${{ matrix.dotnet-version }}
      # You can test your matrix by printing the current dotnet version
      - name: Display dotnet version
        run: dotnet --version
```

Using a specific .NET version

You can configure your job to use a specific version of .NET, such as `6.0.22`. Alternatively, you can use semantic version syntax to get the latest minor release. This example uses the latest minor release of .NET 6.

```yaml
    - name: Setup .NET 6.x
      uses: actions/setup-dotnet@v3
      with:
```

```
        # Semantic version range syntax or exact version of a dotnet version
        dotnet-version: '6.x'
```

Installing dependencies

GitHub-hosted runners have the NuGet package manager installed. You can use the dotnet CLI to install dependencies from the NuGet package registry before building and testing your code. For example, the YAML below installs the `Newtonsoft` package.

```
steps:
- uses: actions/checkout@v4
- name: Setup dotnet
  uses: actions/setup-dotnet@v3
  with:
    dotnet-version: '6.0.x'
- name: Install dependencies
  run: dotnet add package Newtonsoft.Json --version 12.0.1
```

Caching dependencies

You can cache NuGet dependencies using a unique key, which allows you to restore the dependencies for future workflows with the `cache` action. For example, the YAML below installs the `Newtonsoft` package.

For more information, see "Caching dependencies to speed up workflows."

```
steps:
- uses: actions/checkout@v4
- name: Setup dotnet
  uses: actions/setup-dotnet@v3
  with:
    dotnet-version: '6.0.x'
- uses: actions/cache@v3
  with:
    path: ~/.nuget/packages
    # Look to see if there is a cache hit for the corresponding requirements file
    key: ${{ runner.os }}-nuget-${{ hashFiles('**/*.csproj') }}
    restore-keys: |
      ${{ runner.os }}-nuget
- name: Install dependencies
  run: dotnet add package Newtonsoft.Json --version 12.0.1
```

Note: Depending on the number of dependencies, it may be faster to use the dependency cache. Projects with many large dependencies should see a performance increase as it cuts down the time required for downloading. Projects with fewer dependencies may not see a significant performance increase and may even see a slight decrease due to how NuGet installs cached dependencies. The performance varies from project to project.

Building and testing your code

You can use the same commands that you use locally to build and test your code. This example demonstrates how to use `dotnet build` and `dotnet test` in a job:

```
steps:
- uses: actions/checkout@v4
- name: Setup dotnet
  uses: actions/setup-dotnet@v3
  with:
```

```
    dotnet-version: '6.0.x'
- name: Install dependencies
  run: dotnet restore
- name: Build
  run: dotnet build
- name: Test with the dotnet CLI
  run: dotnet test
```

Packaging workflow data as artifacts

After a workflow completes, you can upload the resulting artifacts for analysis. For example, you may need to save log files, core dumps, test results, or screenshots. The following example demonstrates how you can use the `upload-artifact` action to upload test results.

For more information, see "Storing workflow data as artifacts."

```
name: dotnet package

on: [push]

jobs:
  build:

    runs-on: ubuntu-latest
    strategy:
      matrix:
        dotnet-version: [ '3.1.x', '6.0.x' ]

      steps:
        - uses: actions/checkout@v4
        - name: Setup dotnet
          uses: actions/setup-dotnet@v3
          with:
            dotnet-version: ${{ matrix.dotnet-version }}
        - name: Install dependencies
          run: dotnet restore
        - name: Test with dotnet
          run: dotnet test --logger trx --results-directory "TestResults-${{
matrix.dotnet-version }}"
        - name: Upload dotnet test results
          uses: actions/upload-artifact@v3
          with:
            name: dotnet-results-${{ matrix.dotnet-version }}
            path: TestResults-${{ matrix.dotnet-version }}
          # Use always() to always run this step to publish test results when
there are test failures
          if: ${{ always() }}
```

Publishing to package registries

You can configure your workflow to publish your .NET package to a package registry when your CI tests pass. You can use repository secrets to store any tokens or credentials needed to publish your binary. The following example creates and publishes a package to GitHub Packages using `dotnet core cli`.

```
name: Upload dotnet package

on:
  release:
    types: [created]

jobs:
```

```yaml
  deploy:
    runs-on: ubuntu-latest
    permissions:
      packages: write
      contents: read
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-dotnet@v3
        with:
          dotnet-version: '6.0.x' # SDK Version to use.
          source-url: https://nuget.pkg.github.com/<owner>/index.json
        env:
          NUGET_AUTH_TOKEN: ${{secrets.GITHUB_TOKEN}}
      - run: dotnet build --configuration Release <my project>
      - name: Create the package
        run: dotnet pack --configuration Release <my project>
      - name: Publish the package to GPR
        run: dotnet nuget push <my project>/bin/Release/*.nupkg
```