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 PowerShell

In this article

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

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 use PowerShell for CI. It describes how to use Pester, install dependencies, test your module, and publish to the PowerShell Gallery.

GitHub-hosted runners have a tools cache with pre-installed software, which includes PowerShell and Pester.

For a full list of up-to-date software and the pre-installed versions of PowerShell and Pester, see "About GitHub-hosted runners".

Prerequisites

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

We recommend that you have a basic understanding of PowerShell and Pester. For more information, see:

- Getting started with PowerShell
- Pester

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."

Adding a workflow for Pester

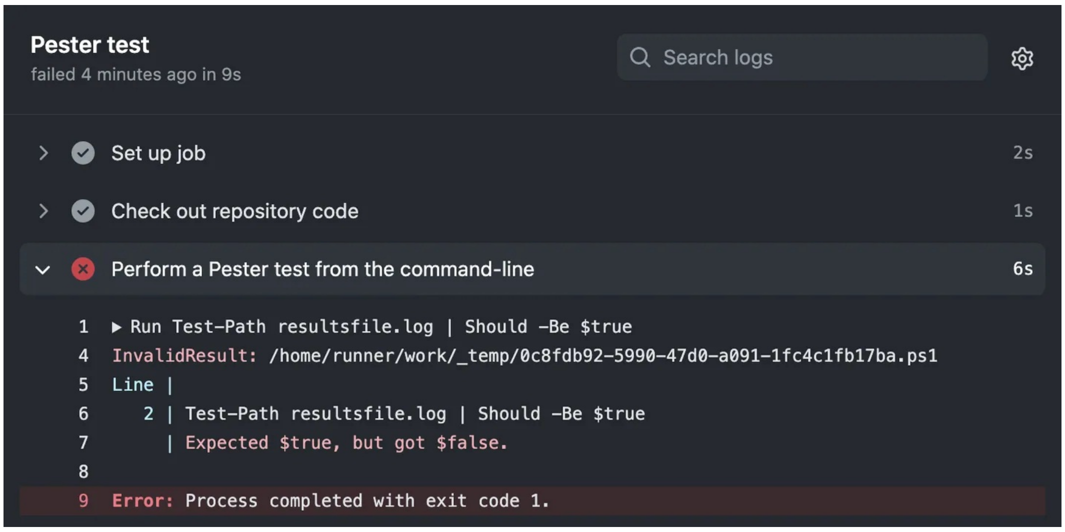
To automate your testing with PowerShell and Pester, you can add a workflow that runs every time a change is pushed to your repository. In the following example, `Test-Path` is used to check that a file called `resultsfile.log` is present.

This example workflow file must be added to your repository's `.github/workflows/` directory:

```
name: Test PowerShell on Ubuntu
on: push

jobs:
  pester-test:
    name: Pester test
    runs-on: ubuntu-latest
    steps:
      - name: Check out repository code
        uses: actions/checkout@v2
      - name: Perform a Pester test from the command-line
        shell: pwsh
        run: Test-Path resultsfile.log | Should -Be $true
      - name: Perform a Pester test from the Tests.ps1 file
        shell: pwsh
        run: |
          Invoke-Pester Unit.Tests.ps1 -Passthru
```

- `shell: pwsh` - Configures the job to use PowerShell when running the `run` commands.
- `run: Test-Path resultsfile.log` - Check whether a file called `resultsfile.log` is present in the repository's root directory.
- `Should -Be $true` - Uses Pester to define an expected result. If the result is unexpected, then GitHub Actions flags this as a failed test. For example:


Pester test
failed 4 minutes ago in 9s
Search logs
Set up job 2s
Check out repository code 1s
Perform a Pester test from the command-line 6s

```
1  ▶ Run Test-Path resultsfile.log | Should -Be $true
4  InvalidResult: /home/runner/work/_temp/0c8fdb92-5990-47d0-a091-1fc4c1fb17ba.ps1
5  Line |
6     2 | Test-Path resultsfile.log | Should -Be $true
7       | Expected $true, but got $false.
8
9  Error: Process completed with exit code 1.
```

- `Invoke-Pester Unit.Tests.ps1 -Passthru` - Uses Pester to execute tests defined in a file called `Unit.Tests.ps1`. For example, to perform the same test described above, the `Unit.Tests.ps1` will contain the following:

```
Describe "Check results file is present" {
    It "Check results file is present" {
        Test-Path resultsfile.log | Should -Be $true
    }
}
```

PowerShell module locations

The table below describes the locations for various PowerShell modules in each GitHub-hosted runner.

	Ubuntu	macOS	Windows
PowerShell system modules	`/opt/microsoft/powershell/7/Modules/*`	`/usr/local/microsoft/powershell/7/Modules/*`	`C:\program files\powershell\7\Modules\*`
PowerShell add-on modules	`/usr/local/share/powershell/Modules/*`	`/usr/local/share/powershell/Modules/*`	`C:\Modules\*`
User-installed modules	`/home/runner/.local/share/powershell/Modules/*`	`/Users/runner/.local/share/powershell/Modules/*`	`C:\Users\runneradmin\Documents\PowerShell\Modules\*`

Installing dependencies

GitHub-hosted runners have PowerShell 7 and Pester installed. You can use `Install-Module` to install additional dependencies from the PowerShell Gallery before building and testing your code.

Note: The pre-installed packages (such as Pester) used by GitHub-hosted runners are regularly updated, and can introduce significant changes. As a result, it is recommended that you always specify the required package versions by using `Install-Module` with `-MaximumVersion`.

For example, the following job installs the `SqlServer` and `PSScriptAnalyzer` modules:

```
jobs:
  install-dependencies:
    name: Install dependencies
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Install from PSGallery
        shell: pwsh
        run: |
          Set-PSRepository PSGallery -InstallationPolicy Trusted
          Install-Module SqlServer, PSScriptAnalyzer
```

Note: By default, no repositories are trusted by PowerShell. When installing modules from the PowerShell Gallery, you must explicitly set the installation policy for `PSGallery` to `Trusted`.

Testing your code

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

Using PSScriptAnalyzer to lint code

The following example installs `PSScriptAnalyzer` and uses it to lint all `ps1` files in the repository. For more information, see PSScriptAnalyzer on GitHub.

```
  lint-with-PSScriptAnalyzer:
    name: Install and run PSScriptAnalyzer
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Install PSScriptAnalyzer module
        shell: pwsh
        run: |
          Set-PSRepository PSGallery -InstallationPolicy Trusted
          Install-Module PSScriptAnalyzer -ErrorAction Stop
      - name: Lint with PSScriptAnalyzer
        shell: pwsh
        run: |
          Invoke-ScriptAnalyzer -Path *.ps1 -Recurse -Outvariable issues
          $errors   = $issues.Where({$_.Severity -eq 'Error'})
          $warnings = $issues.Where({$_.Severity -eq 'Warning'})
          if ($errors) {
              Write-Error "There were $($errors.Count) errors and $($warnings.Count)
          } else {
              Write-Output "There were $($errors.Count) errors and $($warnings.Coun
          }
```

Packaging workflow data as artifacts

You can upload artifacts to view after a workflow completes. For example, you may need to save log files, core dumps, test results, or screenshots. For more information, see "Storing workflow data as artifacts."

The following example demonstrates how you can use the `upload-artifact` action to archive the test results received from `Invoke-Pester`. For more information, see the `upload-artifact` action.

```
name: Upload artifact from Ubuntu

on: [push]

jobs:
  upload-pester-results:
    name: Run Pester and upload results
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Test with Pester
        shell: pwsh
        run: Invoke-Pester Unit.Tests.ps1 -Passthru | Export-CliXml -Path Unit.Tests
      - name: Upload test results
        uses: actions/upload-artifact@v2
        with:
          name: ubuntu-Unit-Tests
          path: Unit.Tests.xml
    if: ${{ always() }}
```

The `always()` function configures the job to continue processing even if there are test failures. For more information, see "Contexts."

Publishing to PowerShell Gallery

You can configure your workflow to publish your PowerShell module to the PowerShell

Gallery when your CI tests pass. You can use secrets to store any tokens or credentials needed to publish your package. For more information, see "Encrypted secrets."

The following example creates a package and uses `Publish-Module` to publish it to the PowerShell Gallery:

```
name: Publish PowerShell Module

on:
  release:
    types: [created]

jobs:
  publish-to-gallery:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Build and publish
        env:
          NUGET_KEY: ${{ secrets.NUGET_KEY }}
        shell: pwsh
        run: |
          ./build.ps1 -Path /tmp/samplemodule
          Publish-Module -Path /tmp/samplemodule -NuGetApiKey $env:NUGET_KEY -Verbos
```