Workflow commands for GitHub Actions

In this article

You can use workflow commands when running shell commands in a workflow or in an action's code.

Bash PowerShell

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.

About workflow commands

Actions can communicate with the runner machine to set environment variables, output values used by other actions, add debug messages to the output logs, and other tasks.

Most workflow commands use the `echo` command in a specific format, while others are invoked by writing to a file. For more information, see "Environment files."

Example of a workflow command

Bash

```
echo "::workflow-command parameter1={data},parameter2={data}::{command value}"
```

```
Write-Output "::workflow-command parameter1={data},parameter2={data}::{command value
```

Note: Workflow command and parameter names are not case-sensitive.

Warning: If you are using Command Prompt, omit double quote characters (`"`) when using workflow commands.

Using workflow commands to access toolkit functions

The actions/toolkit includes a number of functions that can be executed as workflow commands. Use the `::` syntax to run the workflow commands within your YAML file; these commands are then sent to the runner over `stdout`. For example, instead of using code to set an output, as below:

JavaScript

```
core.setOutput('SELECTED_COLOR', 'green');
```

Example: Setting a value

You can use the `set-output` command in your workflow to set the same value:

YAML

```
      - name: Set selected color
        run: echo '::set-output name=SELECTED_COLOR::green'
        id: random-color-generator
      - name: Get color

        run: echo "The selected color is ${{ steps.random-color-generator.outputs.SE
```

YAML

```
      - name: Set selected color
        run: Write-Output "::set-output name=SELECTED_COLOR::green"
        id: random-color-generator
      - name: Get color

        run: Write-Output "The selected color is ${{ steps.random-color-generator.ou
```

The following table shows which toolkit functions are available within a workflow:

Toolkit function	Equivalent workflow command

`core.addPath`	Accessible using environment file `GITHUB_PATH`
`core.debug`	`debug`
`core.notice`	`notice`
`core.error`	`error`
`core.endGroup`	`endgroup`
`core.exportVariable`	Accessible using environment file `GITHUB_ENV`
`core.getInput`	Accessible using environment variable `INPUT_{NAME}`
`core.getState`	Accessible using environment variable `STATE_{NAME}`
`core.isDebug`	Accessible using environment variable `RUNNER_DEBUG`
`core.saveState`	`save-state`
`core.setCommandEcho`	`echo`
`core.setFailed`	Used as a shortcut for `::error` and `exit 1`
`core.setOutput`	`set-output`
`core.setSecret`	`add-mask`
`core.startGroup`	`group`
`core.warning`	`warning`

Setting an output parameter

Sets an action's output parameter.

Text

```
::set-output name={name}::{value}
```

Optionally, you can also declare output parameters in an action's metadata file. For more information, see "Metadata syntax for GitHub Actions."

You can escape multiline strings for setting an output parameter by creating an environment variable and using it in a workflow command. For more information, see "Setting an environment variable."

Example: Setting an output parameter

Bash

```
echo "::set-output name=action_fruit::strawberry"
```

```
Write-Output "::set-output name=action_fruit::strawberry"
```

Setting a debug message

Prints a debug message to the log. You must create a secret named `ACTIONS_STEP_DEBUG` with the value `true` to see the debug messages set by this command in the log. For more information, see "Enabling debug logging."

Text

```
::debug::{message}
```

Example: Setting a debug message

Bash

```
echo "::debug::Set the Octocat variable"
```

```
Write-Output "::debug::Set the Octocat variable"
```

Setting a notice message

Creates a notice message and prints the message to the log. This message will create an annotation, which can associate the message with a particular file in your repository. Optionally, your message can specify a position within the file.

Text

```
::notice file={name},line={line},endLine={endLine},title={title}::{message}
```

Parameter	Value
`title`	Custom title
`file`	Filename
`col`	Column number, starting at 1
`endColumn`	End column number
`line`	Line number, starting at 1
`endLine`	End line number

Example: Setting a notice message

Bash

```
echo "::notice file=app.js,line=1,col=5,endColumn=7::Missing semicolon"
```

```
Write-Output "::notice file=app.js,line=1,col=5,endColumn=7::Missing semicolon"
```

Setting a warning message

Creates a warning message and prints the message to the log. This message will create an annotation, which can associate the message with a particular file in your repository. Optionally, your message can specify a position within the file.

Text

```
::warning file={name},line={line},endLine={endLine},title={title}::{message}
```

Parameter	Value
`title`	Custom title
`file`	Filename
`col`	Column number, starting at 1
`endColumn`	End column number
`line`	Line number, starting at 1
`endLine`	End line number

Example: Setting a warning message

Bash

```
echo "::warning file=app.js,line=1,col=5,endColumn=7::Missing semicolon"
```

```
Write-Output "::warning file=app.js,line=1,col=5,endColumn=7::Missing semicolon"
```

Setting an error message

Creates an error message and prints the message to the log. This message will create an annotation, which can associate the message with a particular file in your repository. Optionally, your message can specify a position within the file.

Text

```
::error file={name},line={line},endLine={endLine},title={title}::{message}
```

Parameter	Value
`title`	Custom title
`file`	Filename
`col`	Column number, starting at 1
`endColumn`	End column number
`line`	Line number, starting at 1
`endLine`	End line number

Example: Setting an error message

Bash

```
echo "::error file=app.js,line=1,col=5,endColumn=7::Missing semicolon"
```

```
Write-Output "::error file=app.js,line=1,col=5,endColumn=7::Missing semicolon"
```

Grouping log lines

Creates an expandable group in the log. To create a group, use the `group` command and specify a `title`. Anything you print to the log between the `group` and `endgroup` commands is nested inside an expandable entry in the log.

Text

```
::group::{title}
::endgroup::
```

Example: Grouping log lines

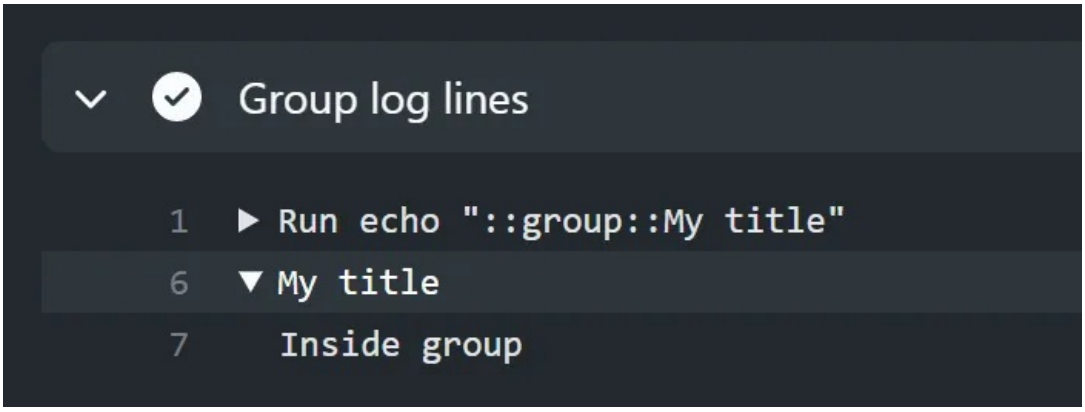
YAML

```
jobs:
  bash-example:
    runs-on: ubuntu-latest
    steps:
      - name: Group of log lines
        run: |
            echo "::group::My title"
            echo "Inside group"
            echo "::endgroup::"
```

YAML

```
jobs:
  powershell-example:
```

```
    runs-on: windows-latest
    steps:
      - name: Group of log lines
        run: |
            Write-Output "::group::My title"
            Write-Output "Inside group"
            Write-Output "::endgroup::"
```



Masking a value in a log

Text

```
::add-mask::{value}
```

Masking a value prevents a string or variable from being printed in the log. Each masked word separated by whitespace is replaced with the `*` character. You can use an environment variable or string for the mask's `value`. When you mask a value, it is treated as a secret and will be redacted on the runner. For example, after you mask a value, you won't be able to set that value as an output.

Example: Masking a string

When you print `"Mona The Octocat"` in the log, you'll see `"***"`.

Bash

```
echo "::add-mask::Mona The Octocat"
```

```
Write-Output "::add-mask::Mona The Octocat"
```

Warning: Make sure you register the secret with 'add-mask' before outputting it in the build logs or using it in any other workflow commands.

Example: Masking an environment variable

When you print the variable `MY_NAME` or the value `"Mona The Octocat"` in the log, you'll see `"***"` instead of `"Mona The Octocat"`.

YAML

```yaml
jobs:
  bash-example:
    runs-on: ubuntu-latest
    env:
      MY_NAME: "Mona The Octocat"
    steps:
      - name: bash-version
        run: echo "::add-mask::$MY_NAME"
```

YAML

```yaml
jobs:
  powershell-example:
    runs-on: windows-latest
    env:
      MY_NAME: "Mona The Octocat"
    steps:
      - name: powershell-version
        run: Write-Output "::add-mask::$env:MY_NAME"
```

Example: Masking a generated output within a single job

Note: You must use `add-mask` before you use `set-output`. Otherwise, the output will not be masked.

If you do not need to pass your secret from one job to another job, you can:

1. Generate the secret (without outputting it).
2. Mask it with `add-mask`.
3. Use `set-output` to make the secret available to other steps within the job.

YAML

```yaml
on: push
jobs:
  generate-a-secret-output:
    runs-on: ubuntu-latest
    steps:
      - id: sets-a-secret
        name: Generate, mask, and output a secret
        run: |
          the_secret=$((RANDOM))
          echo "::add-mask::$the_secret"
          echo "::set-output name=secret-number::$the_secret"
      - name: Use that secret output (protected by a mask)
        run: |
          echo "the secret number is ${{ steps.sets-a-secret.outputs.secret-number }
```

YAML

```yaml
on: push
jobs:
  generate-a-secret-output:
    runs-on: ubuntu-latest
    steps:
      - id: sets-a-secret
```

```
        name: Generate, mask, and output a secret
        shell: pwsh
        run: |
          Set-Variable -Name TheSecret -Value (Get-Random)
          Write-Output "::add-mask::$TheSecret"
          Write-Output "::set-output name=secret-number::$TheSecret"
      - name: Use that secret output (protected by a mask)
        shell: pwsh
        run: |
          Write-Output "the secret number is ${{ steps.sets-a-secret.outputs.secret
```

Example: Masking and passing a secret between jobs or workflows

If you want to pass a masked secret between jobs or workflows, you should store the secret in a store and then retrieve it in the subsequent job or workflow.

Setup

1. Set up a secret store to store the secret that you will generate during your workflow. For example, Vault.
2. Generate a key for reading and writing to that secret store. Store the key as a repository secret. In the following example workflow, the secret name is `SECRET_STORE_CREDENTIALS`. For more information, see "Encrypted secrets."

Workflow

Note: This workflow uses an imaginary secret store, `secret-store`, which has imaginary commands `store-secret` and `retrieve-secret`. `some/secret-store@27b31702a0e7fc50959f5ad993c78deac1bdfc29` is an imaginary action that installs the `secret-store` application and configures it to connect to an `instance` with `credentials`.

YAML

```yaml
on: push

jobs:
  secret-generator:
    runs-on: ubuntu-latest
    steps:
    - uses: some/secret-store@v1
      with:
        credentials: ${{ secrets.SECRET_STORE_CREDENTIALS }}
        instance: ${{ secrets.SECRET_STORE_INSTANCE }}
    - name: generate secret
      shell: bash
      run: |
        GENERATED_SECRET=$((RANDOM))
        echo "::add-mask::$GENERATED_SECRET"
        SECRET_HANDLE=$(secret-store store-secret "$GENERATED_SECRET")
        echo "::set-output name=handle::$secret_handle"
  secret-consumer:
    runs-on: macos-latest
    needs: secret-generator
    steps:
    - uses: some/secret-store@v1
      with:
        credentials: ${{ secrets.SECRET_STORE_CREDENTIALS }}
        instance: ${{ secrets.SECRET_STORE_INSTANCE }}
```

```
    - name: use secret
      shell: bash
      run: |
        SECRET_HANDLE="${{ needs.secret-generator.outputs.handle }}"
        RETRIEVED_SECRET=$(secret-store retrieve-secret "$SECRET_HANDLE")
        echo "::add-mask::$RETRIEVED_SECRET"
        echo "We retrieved our masked secret: $RETRIEVED_SECRET"
```

YAML

```yaml
on: push

jobs:
  secret-generator:
    runs-on: ubuntu-latest
    steps:
    - uses: some/secret-store@v1
      with:
        credentials: ${{ secrets.SECRET_STORE_CREDENTIALS }}
        instance: ${{ secrets.SECRET_STORE_INSTANCE }}
    - name: generate secret
      shell: pwsh
      run: |
        Set-Variable -Name Generated_Secret -Value (Get-Random)
        Write-Output "::add-mask::$Generated_Secret"
        Set-Variable -Name Secret_Handle -Value (Store-Secret "$Generated_Secret")
        Write-Output "::set-output name=handle::$Secret_Handle"
  secret-consumer:
    runs-on: macos-latest
    needs: secret-generator
    steps:
    - uses: some/secret-store@v1
      with:
        credentials: ${{ secrets.SECRET_STORE_CREDENTIALS }}
        instance: ${{ secrets.SECRET_STORE_INSTANCE }}
    - name: use secret
      shell: pwsh
      run: |
        Set-Variable -Name Secret_Handle -Value "${{ needs.secret-generator.outputs
        Set-Variable -Name Retrieved_Secret -Value (Retrieve-Secret "$Secret_Handle'
        echo "::add-mask::$Retrieved_Secret"
        echo "We retrieved our masked secret: $Retrieved_Secret"
```

Stopping and starting workflow commands

Stops processing any workflow commands. This special command allows you to log anything without accidentally running a workflow command. For example, you could stop logging to output an entire script that has comments.

Text

```
::stop-commands::{endtoken}
```

To stop the processing of workflow commands, pass a unique token to `stop-commands`. To resume processing workflow commands, pass the same token that you used to stop workflow commands.

Warning: Make sure the token you're using is randomly generated and unique for each run.

Text

```
::{endtoken}::
```

Example: Stopping and starting workflow commands

YAML

```yaml
jobs:
  workflow-command-job:
    runs-on: ubuntu-latest
    steps:
      - name: Disable workflow commands
        run: |
          echo '::warning:: This is a warning message, to demonstrate that commands
          stopMarker=$(uuidgen)
          echo "::stop-commands::$stopMarker"
          echo '::warning:: This will NOT be rendered as a warning, because stop-cor
          echo "::$stopMarker::"
          echo '::warning:: This is a warning again, because stop-commands has been
```

YAML

```yaml
jobs:
  workflow-command-job:
    runs-on: windows-latest
    steps:
      - name: Disable workflow commands
        run: |
          Write-Output '::warning:: This is a warning message, to demonstrate that
          $stopMarker = New-Guid
          Write-Output "::stop-commands::$stopMarker"
          Write-Output '::warning:: This will NOT be rendered as a warning, because
          Write-Output "::$stopMarker::"
          Write-Output '::warning:: This is a warning again, because stop-commands
```

Echoing command outputs

Enables or disables echoing of workflow commands. For example, if you use the `set-output` command in a workflow, it sets an output parameter but the workflow run's log does not show the command itself. If you enable command echoing, then the log shows the command, such as `::set-output name={name}::{value}`.

Text

```
::echo::on
::echo::off
```

Command echoing is disabled by default. However, a workflow command is echoed if there are any errors processing the command.

The `add-mask`, `debug`, `warning`, and `error` commands do not support echoing because their outputs are already echoed to the log.

You can also enable command echoing globally by turning on step debug logging using the `ACTIONS_STEP_DEBUG` secret. For more information, see "Enabling debug logging". In contrast, the `echo` workflow command lets you enable command echoing at a more granular level, rather than enabling it for every workflow in a repository.

Example: Toggling command echoing

YAML

```yaml
jobs:
  workflow-command-job:
    runs-on: ubuntu-latest
    steps:
      - name: toggle workflow command echoing
        run: |
          echo '::set-output name=action_echo::disabled'
          echo '::echo::on'
          echo '::set-output name=action_echo::enabled'
          echo '::echo::off'
          echo '::set-output name=action_echo::disabled'
```

YAML

```yaml
jobs:
  workflow-command-job:
    runs-on: windows-latest
    steps:
      - name: toggle workflow command echoing
        run: |
          write-output "::set-output name=action_echo::disabled"
          write-output "::echo::on"
          write-output "::set-output name=action_echo::enabled"
          write-output "::echo::off"
          write-output "::set-output name=action_echo::disabled"
```

The example above prints the following lines to the log:

Text

```
::set-output name=action_echo::enabled
::echo::off
```

Only the second `set-output` and `echo` workflow commands are included in the log because command echoing was only enabled when they were run. Even though it is not always echoed, the output parameter is set in all cases.

Sending values to the pre and post actions

You can use the `save-state` command to create environment variables for sharing with your workflow's `pre:` or `post:` actions. For example, you can create a file with the `pre:` action, pass the file location to the `main:` action, and then use the `post:` action to delete the file. Alternatively, you could create a file with the `main:` action, pass the file location to the `post:` action, and also use the `post:` action to delete the file.

If you have multiple `pre:` or `post:` actions, you can only access the saved value in the action where `save-state` was used. For more information on the `post:` action, see "Metadata syntax for GitHub Actions."

The `save-state` command can only be run within an action, and is not available to YAML files. The saved value is stored as an environment value with the `STATE_` prefix.

This example uses JavaScript to run the `save-state` command. The resulting environment variable is named `STATE_processID` with the value of `12345`:

JavaScript

```
console.log('::save-state name=processID::12345')
```

The `STATE_processID` variable is then exclusively available to the cleanup script running under the `main` action. This example runs in `main` and uses JavaScript to display the value assigned to the `STATE_processID` environment variable:

JavaScript

```
console.log("The running PID from the main action is: " +  process.env.STATE_process
```

Environment files

During the execution of a workflow, the runner generates temporary files that can be used to perform certain actions. The path to these files are exposed via environment variables. You will need to use UTF-8 encoding when writing to these files to ensure proper processing of the commands. Multiple commands can be written to the same file, separated by newlines.

Most commands in the following examples use double quotes for echoing strings, which will attempt to interpolate characters like `$` for shell variable names. To always use literal values in quoted strings, you can use single quotes instead.

Note: PowerShell versions 5.1 and below (`shell: powershell`) do not use UTF-8 by default, so you must specify the UTF-8 encoding. For example:

YAML

```yaml
jobs:
legacy-powershell-example:
  runs-on: windows-latest
  steps:
    - shell: powershell
      run: |
        "mypath" | Out-File -FilePath $env:GITHUB_PATH -Encoding utf8 -Append
```

PowerShell Core versions 6 and higher (`shell: pwsh`) use UTF-8 by default. For example:

YAML

```yaml
jobs:
powershell-core-example:
  runs-on: windows-latest
  steps:
    - shell: pwsh
      run: |
        "mypath" >> $env:GITHUB_PATH
```

Setting an environment variable

Bash

```
echo "{environment_variable_name}={value}" >> "$GITHUB_ENV"
```

- Using PowerShell version 6 and higher:

```
"{environment_variable_name}={value}" >> $env:GITHUB_ENV
```

- Using PowerShell version 5.1 and below:

```
"{environment_variable_name}={value}" | Out-File -FilePath $env:GITHUB_ENV -Enc
```

You can make an environment variable available to any subsequent steps in a workflow job by defining or updating the environment variable and writing this to the `GITHUB_ENV` environment file. The step that creates or updates the environment variable does not have access to the new value, but all subsequent steps in a job will have access. The names of environment variables are case-sensitive, and you can include punctuation. For more information, see "Variables."

Example of writing an environment variable to `GITHUB_ENV`

YAML

```yaml
steps:
  - name: Set the value
    id: step_one
    run: |
      echo "action_state=yellow" >> "$GITHUB_ENV"
  - name: Use the value
    id: step_two
    run: |

      echo "${{ env.action_state }}" # This will output 'yellow'
```

YAML

```yaml
steps:
  - name: Set the value
    id: step_one
    run: |
      "action_state=yellow" >> $env:GITHUB_ENV
  - name: Use the value
    id: step_two
    run: |

      Write-Output "${{ env.action_state }}" # This will output 'yellow'
```

Multiline strings

For multiline strings, you may use a delimiter with the following syntax.

Text

```
{name}<<{delimiter}
{value}
{delimiter}
```

Warning: Make sure the delimiter you're using is randomly generated and unique for each run. For more information, see "Security hardening for GitHub Actions".

Example of a multiline string

This example selects a random value for `$EOF` as a delimiter, and sets the `JSON_RESPONSE` environment variable to the value of the `curl` response.

YAML

```yaml
steps:
  - name: Set the value in bash
    id: step_one
    run: |
      EOF=$(dd if=/dev/urandom bs=15 count=1 status=none | base64)
      echo "JSON_RESPONSE<<$EOF" >> "$GITHUB_ENV"
      curl https://example.com >> "$GITHUB_ENV"
      echo "$EOF" >> "$GITHUB_ENV"
```

YAML

```yaml
steps:
  - name: Set the value in pwsh
    id: step_one
    run: |
      -join (1..15 | ForEach {[char]((48..57)+(65..90)+(97..122) | Get-Random)}) | s
      "JSON_RESPONSE<<$EOF" >> $env:GITHUB_ENV
      (Invoke-WebRequest -Uri "https://example.com").Content >> $env:GITHUB_ENV
      "$EOF" >> $env:GITHUB_ENV
    shell: pwsh
```

Adding a system path

Prepends a directory to the system `PATH` variable and automatically makes it available to all subsequent actions in the current job; the currently running action cannot access the updated path variable. To see the currently defined paths for your job, you can use `echo "$PATH"` in a step or an action.

Bash

```
echo "{path}" >> $GITHUB_PATH
```

```
"{path}" >> $env:GITHUB_PATH
```

Example of adding a system path

This example demonstrates how to add the user `$HOME/.local/bin` directory to `PATH`:

Bash

```
echo "$HOME/.local/bin" >> $GITHUB_PATH
```

This example demonstrates how to add the user `$env:HOMEPATH/.local/bin` directory to `PATH`:

```
"$env:HOMEPATH/.local/bin" >> $env:GITHUB_PATH
```