



# **Workflow commands for GitHub Actions**

#### In this article

About workflow commands

Using workflow commands to access toolkit functions

Setting a debug message

Setting a notice message

Setting a warning message

Setting an error message

Grouping log lines

Masking a value in a log

Stopping and starting workflow commands

Sending values to the pre and post actions

**Environment files** 

Setting an environment variable

Setting an output parameter

Adding a job summary

Adding a system path

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 <u>GitHub public roadmap</u>.

#### 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** $\mathscr O$



**Note:** Workflow command and parameter names are case insensitive.

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

# Using workflow commands to access toolkit functions *∂*

The <u>actions/toolkit</u> 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 <a href="stdout">stdout</a>. For example, instead of using code to create an error annotation, as below:



#### **Example: Creating an annotation for an error** $\mathscr O$

You can use the error command in your workflow to create the same error annotation:



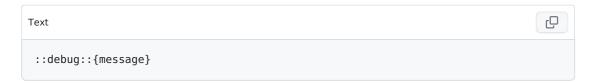
The following table shows which toolkit functions are available within a workflow:

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.summary	Accessible using environment file GITHUB_STEP_SUMMARY
core.saveState	Accessible using environment file GITHUB_STATE
core.setCommandEcho	echo
core.setFailed	Used as a shortcut for ::error and exit 1
core.setOutput	Accessible using environment file GITHUB_OUTPUT
core.setSecret	add-mask
core.startGroup	group
core.warning	warning

# 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."



# **Example: Setting a debug message** $\mathscr O$



# 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.



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** $\mathscr O$



# 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.



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 $\mathscr {O}$



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

# **Setting an error message** *P*

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.



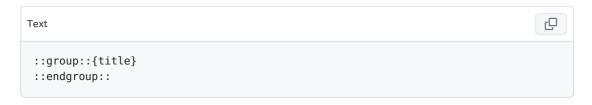
Parameter	Value
title	Custom title
file	Filename
col	Column number, starting at 1
endColumn	End column number
line	Line number, starting at 1
endLine	End line number

# **Example: Setting an error message** $\mathscr E$



# **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.



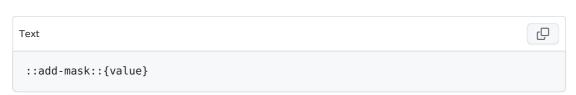
# **Example: Grouping log lines** $\mathscr P$

VAMI

```
jobs:
  bash-example:
    runs-on: ubuntu-latest
      - name: Group of log lines
        run:
           echo "::group::My title"
           echo "Inside group"
           echo "::endgroup::"
                                                                          Q
ΥΔΜΙ
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::"
               Group log lines
               ▶ Run echo "::group::My title"
               ▼ My title
                  Inside group
```

2

# Masking a value in a log ₽



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** $\mathscr O$

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



```
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
                                                                                 Q
jobs:
  bash-example:
    runs-on: ubuntu-latest
    env:
      MY_NAME: "Mona The Octocat"
    steps:
      - name: bash-version
         run: echo "::add-mask::$MY NAME"
YAML
                                                                                 Q
jobs:
  powershell-example:
    runs-on: windows-latest
    env:
      MY NAME: "Mona The Octocat"
    steps:
      - name: powershell-version
         run: Write-Output "::add-mask::$env:MY NAME"
```

# Example: Masking a generated output within a single job @

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 GITHUB\_OUTPUT to make the secret available to other steps within the job.

```
echo "the secret number is ${{ steps.sets-a-secret.outputs.secret-
number }}"
```

```
ф
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::$TheSecret"
          "secret-number=$TheSecret" >> $env:GITHUB OUTPUT
       - name: Use that secret output (protected by a mask)
        shell: pwsh
         run:
          Write-Output "the secret number is ${{ steps.sets-a-
secret.outputs.secret-number }}"
```

# Example: Masking and passing a secret between jobs or workflows $\mathscr O$

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.
- Qenerate 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 "<u>Using secrets in GitHub</u> Actions."

#### 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.

```
on: push

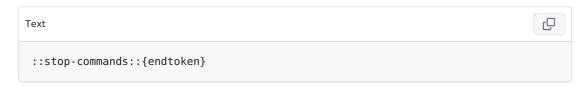
jobs:
    secret-generator:
    runs-on: ubuntu-latest
    outputs:
        handle: ${{ steps.generate-secret.outputs.handle }}
    steps:
    - uses: some/secret-store@v1
        with:
        credentials: ${{ secrets.SECRET_STORE_CREDENTIALS }}
```

```
instance: ${{ secrets.SECRET_STORE_INSTANCE }}
  - name: generate secret
    id: generate-secret
    shell: bash
    run:
      GENERATED SECRET=$((RANDOM))
      echo "::add-mask::$GENERATED SECRET"
      SECRET HANDLE=$(secret-store store-secret "$GENERATED SECRET")
      echo "handle=$SECRET_HANDLE" >> "$GITHUB_OUTPUT"
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
                                                                                 Q
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
 "$Generated Secret")
        "handle=$Secret Handle" >> $env:GITHUB OUTPUT
  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.handle }}"
        Set-Variable -Name Retrieved Secret -Value (Retrieve-Secret
 "$Secret Handle")
        echo "::add-mask::$Retrieved Secret"
        echo "We retrieved our masked secret: $Retrieved_Secret"
```

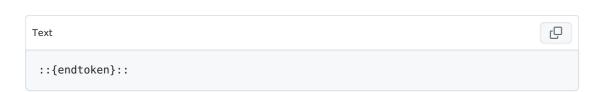
# Stopping and starting workflow commands &

anything without accidentally running a workflow command. For example, you could stop logging to output an entire script that has comments.



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.



#### **Example: Stopping and starting workflow commands** $\mathscr {P}$

```
Q
YAML
jobs:
  workflow-command-job:
    runs-on: ubuntu-latest
    steps:
       - name: Disable workflow commands
          echo '::warning:: This is a warning message, to demonstrate that
commands are being processed.'
          stopMarker=$(uuidgen)
          echo "::stop-commands::$stopMarker"
          echo '::warning:: This will NOT be rendered as a warning, because stop-
commands has been invoked.'
          echo "::$stopMarker::"
          echo '::warning:: This is a warning again, because stop-commands has
been turned off.'
```

```
YAML
                                                                                 ſΩ
jobs:
  workflow-command-job:
    runs-on: windows-latest
    steps:
      - name: Disable workflow commands
          Write-Output '::warning:: This is a warning message, to demonstrate
that commands are being processed.'
           $stopMarker = New-Guid
          Write-Output "::stop-commands::$stopMarker"
          Write-Output '::warning:: This will NOT be rendered as a warning,
because stop-commands has been invoked.'
          Write-Output "::$stopMarker::"
          Write-Output '::warning:: This is a warning again, because stop-
commands has been turned off.'
```

# Sending values to the pre and post actions ${\mathscr O}$

You can create environment variables for sharing with your workflow's pre: or post: actions by writing to the file located at GITHUB\_STATE. 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 it was written to GITHUB\_STATE. For more information on the post: action, see "Metadata syntax for GitHub Actions."

The GITHUB\_STATE file is only available within an action. The saved value is stored as an environment value with the STATE\_ prefix.

This example uses JavaScript to write to the GITHUB\_STATE file. The resulting environment variable is named STATE processID with the value of 12345:

```
import * as fs from 'fs'
import * as os from 'os'

fs.appendFileSync(process.env.GITHUB_STATE, `processID=12345${os.EOL}`, {
   encoding: 'utf8'
})
```

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_processID);
```

#### **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.

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: Q 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: Q YAML jobs: powershell-core-example:

```
runs-on: windows-latest
steps:
    - shell: pwsh
    run: |
        "mypath" | Out-File -FilePath $env:GITHUB_PATH -Append
```

# Setting an environment variable &

**Note**: To avoid issues, it's good practice to treat environment variables as case sensitive, irrespective of the behavior of the operating system and shell you are using.

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

• Using PowerShell version 6 and higher:

```
PowerShell

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

Using PowerShell version 5.1 and below:

YAML

```
PowerShell

"{environment_variable_name}={value}" | Out-File -FilePath $env:GITHUB_ENV -
Encoding utf8 -Append
```

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.

You can't overwrite the value of the default environment variables named <code>GITHUB\_\*</code> and <code>RUNNER\_\*</code>. Currently you can overwrite the value of the <code>CI</code> variable. However, it's not guaranteed that this will always be possible. For more information about the default environment variables, see "Variables."

# Example of writing an environment variable to GITHUB\_ENV &

```
steps:
    name: Set the value
    id: step_one
    run: |
        echo "action_state=yellow" >> "$GITHUB_ENV"
    name: Use the value
    id: step_two
    run: |
        printf '%s\n' "$action_state" # This will output 'yellow'
```

Q

#### 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 won't occur on a line of its own within the value. If the value is completely arbitrary then you shouldn't use this format. Write the value to a file instead.

#### Example of a multiline string &

This example uses EOF as the delimiter, and sets the JSON\_RESPONSE environment variable to the value of the curl response.

```
steps:
    - name: Set the value in bash
    id: step_one
    run: |
     {
        echo 'JSON_RESPONSE<<EOF'
        curl https://example.com
        echo EOF
     } >> "$GITHUB_ENV"
```

# Setting an output parameter *∂*

Sets a step's output parameter. Note that the step will need an id to be defined to later retrieve the output value. You can set multi-line output values with the same technique used in the "Multiline strings" section to define multi-line environment variables.



### Example of setting an output parameter $\mathscr O$

This example demonstrates how to set the SELECTED\_COLOR output parameter and later retrieve it:

```
- name: Set color
id: color-selector
run: echo "SELECTED_COLOR=green" >> "$GITHUB_OUTPUT"
- name: Get color
env:
    SELECTED_COLOR: ${{ steps.color-selector.outputs.SELECTED_COLOR }}
run: echo "The selected color is $SELECTED_COLOR"
```

This example demonstrates how to set the SELECTED\_COLOR output parameter and later retrieve it:

# Adding a job summary &



You can set some custom Markdown for each job so that it will be displayed on the summary page of a workflow run. You can use job summaries to display and group

unique content, such as test result summaries, so that someone viewing the result of a workflow run doesn't need to go into the logs to see important information related to the run, such as failures.

Job summaries support <u>GitHub flavored Markdown</u>, and you can add your Markdown content for a step to the <u>GITHUB\_STEP\_SUMMARY</u> environment file. <u>GITHUB\_STEP\_SUMMARY</u> is unique for each step in a job. For more information about the per-step file that <u>GITHUB\_STEP\_SUMMARY</u> references, see "<u>Environment files</u>."

When a job finishes, the summaries for all steps in a job are grouped together into a single job summary and are shown on the workflow run summary page. If multiple jobs generate summaries, the job summaries are ordered by job completion time.

#### **Example of adding a job summary** @



#### Multiline Markdown content &

For multiline Markdown content, you can use >> to continuously append content for the current step. With every append operation, a newline character is automatically added.

#### Example of multiline Markdown content &

```
- name: Generate list using Markdown
  run: I
    echo "This is the lead in sentence for the list" >> $GITHUB STEP SUMMARY
    echo "" >> $GITHUB STEP SUMMARY # this is a blank line
    echo "- Lets add a bullet point" >> $GITHUB STEP SUMMARY
    echo "- Lets add a second bullet point" >> $GITHUB STEP SUMMARY
    echo "- How about a third one?" >> $GITHUB STEP SUMMARY
- name: Generate list using Markdown
  run:
    "This is the lead in sentence for the list" | Out-File -FilePath
$env:GITHUB STEP SUMMARY -Append
    "" | Out-File -FilePath $env:GITHUB STEP SUMMARY -Append # this is a blank
    "- Lets add a bullet point" | Out-File -FilePath $env:GITHUB STEP SUMMARY -
Append
    "- Lets add a second bullet point" | Out-File -FilePath
$env:GITHUB STEP SUMMARY -Append
    "- How about a third one?" | Out-File -FilePath $env:GITHUB_STEP_SUMMARY -
Append
```

#### Overwriting job summaries &

To clear all content for the current step, you can use > to overwrite any previously added content in Bash, or remove -Append in PowerShell

#### Example of overwriting job summaries $\mathscr{O}$

```
- name: Overwrite Markdown
run: |
   echo "Adding some Markdown content" >> $GITHUB_STEP_SUMMARY
   echo "There was an error, we need to clear the previous Markdown with some
new content." > $GITHUB_STEP_SUMMARY
```

#### Removing job summaries &

To completely remove a summary for the current step, the file that <code>GITHUB\_STEP\_SUMMARY</code> references can be deleted.

#### **Example of removing job summaries** $\mathscr P$

```
- name: Delete all summary content
run: |
   echo "Adding Markdown content that we want to remove before the step ends" >>
$GITHUB_STEP_SUMMARY
   rm $GITHUB_STEP_SUMMARY
```

```
    name: Delete all summary content
    run: |
        "Adding Markdown content that we want to remove before the step ends" | Out-
    File -FilePath $env:GITHUB_STEP_SUMMARY -Append
    Remove-Item $env:GITHUB_STEP_SUMMARY
```

After a step has completed, job summaries are uploaded and subsequent steps cannot modify previously uploaded Markdown content. Summaries automatically mask any secrets that might have been added accidentally. If a job summary contains sensitive information that must be deleted, you can delete the entire workflow run to remove all its job summaries. For more information see "Deleting a workflow run."

# Step isolation and limits &

Job summaries are isolated between steps and each step is restricted to a maximum size of 1MiB. Isolation is enforced between steps so that potentially malformed Markdown from a single step cannot break Markdown rendering for subsequent steps. If more than 1MiB of content is added for a step, then the upload for the step will fail and an error annotation will be created. Upload failures for job summaries do not affect the overall status of a step or a job. A maximum of 20 job summaries from steps are displayed per job.

# 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.



#### Example of adding a system path &

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



#### Legal

© 2023 GitHub, Inc. <u>Terms</u> <u>Privacy</u> <u>Status</u> <u>Pricing</u> <u>Expert services</u> <u>Blog</u>

"\$env:HOMEPATH/.local/bin" | Out-File -FilePath \$env:GITHUB\_PATH -Append