About workflows

In this article

Get a high-level overview of GitHub Actions workflows, including triggers, syntax, and advanced features.

About workflows

A workflow is a configurable automated process that will run one or more jobs. Workflows are defined by a YAML file checked in to your repository and will run when triggered by an event in your repository, or they can be triggered manually, or at a defined schedule.

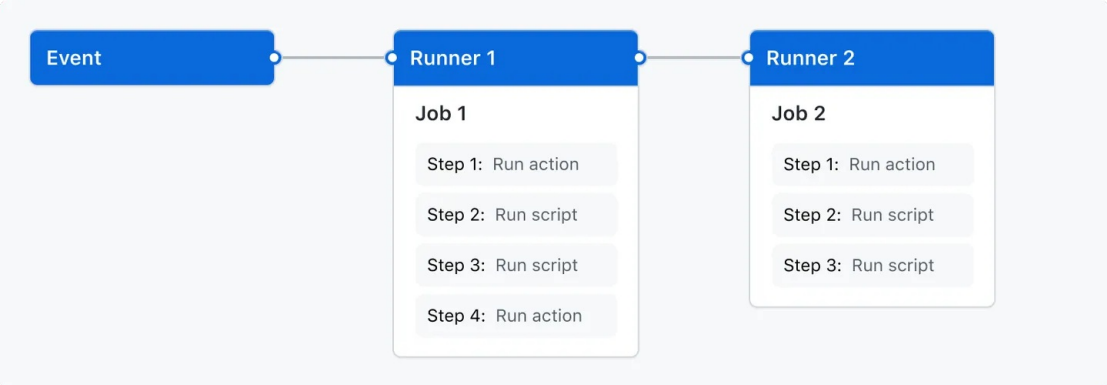
Workflows are defined in the `.github/workflows` directory in a repository, and a repository can have multiple workflows, each of which can perform a different set of tasks. For example, you can have one workflow to build and test pull requests, another workflow to deploy your application every time a release is created, and still another workflow that adds a label every time someone opens a new issue.

Workflow basics

A workflow must contain the following basic components:

1. One or more *events* that will trigger the workflow.
2. One or more *jobs*, each of which will execute on a *runner* machine and run a series of one or more *steps*.
3. Each step can either run a script that you define or run an action, which is a reusable extension that can simplify your workflow.

For more information on these basic components, see "Understanding GitHub Actions."



Triggering a workflow

Workflow triggers are events that cause a workflow to run. These events can be:

- Events that occur in your workflow's repository
- Events that occur outside of GitHub Enterprise Server and trigger a `repository_dispatch` event on GitHub Enterprise Server
- Scheduled times
- Manual

For example, you can configure your workflow to run when a push is made to the default branch of your repository, when a release is created, or when an issue is opened.

For more information, see "Triggering a workflow", and for a full list of events, see "Events that trigger workflows."

Workflow syntax

Workflow are defined using YAML. For the full reference of the YAML syntax for authoring workflows, see "Workflow syntax for GitHub Actions."

Create an example workflow

GitHub Actions uses YAML syntax to define the workflow. Each workflow is stored as a separate YAML file in your code repository, in a directory named `.github/workflows`.

You can create an example workflow in your repository that automatically triggers a series of commands whenever code is pushed. In this workflow, GitHub Actions checks out the pushed code, installs the bats testing framework, and runs a basic command to output the bats version: `bats -v`.

1. In your repository, create the `.github/workflows/` directory to store your workflow files.
2. In the `.github/workflows/` directory, create a new file called `learn-github-actions.yml` and add the following code.

YAML

```yaml
name: learn-github-actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
```

```
      - uses: actions/setup-node@v2
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

3. Commit these changes and push them to your GitHub repository.

Your new GitHub Actions workflow file is now installed in your repository and will run automatically each time someone pushes a change to the repository. To see the details about a workflow's execution history, see "Viewing the activity for a workflow run."

Understanding the workflow file

To help you understand how YAML syntax is used to create a workflow file, this section explains each line of the introduction's example:

Code	Explanation
`name: learn-github-actions`	*Optional* - The name of the workflow as it will appear in the "Actions" tab of the GitHub repository.
`on: [push]`	Specifies the trigger for this workflow. This example uses the `push` event, so a workflow run is triggered every time someone pushes a change to the repository or merges a pull request. This is triggered by a push to every branch; for examples of syntax that runs only on pushes to specific branches, paths, or tags, see "Workflow syntax for GitHub Actions."
`jobs:`	Groups together all the jobs that run in the `learn-github-actions` workflow.
`check-bats-version:`	Defines a job named `check-bats-version`. The child keys will define properties of the job.
`  runs-on: ubuntu-latest`	Configures the job to run on the latest version of an Ubuntu Linux runner. This means that the job will execute on a fresh virtual machine hosted by GitHub. For syntax examples using other runners, see "Workflow syntax for GitHub Actions."
`  steps:`	Groups together all the steps that run in the `check-bats-version` job. Each item nested under this section is a separate action or shell script.
`    - uses: actions/checkout@v2`	The `uses` keyword specifies that this step will run `v3` of the `actions/checkout` action. This is an action that checks out your repository onto the runner, allowing you to run scripts or other actions against your code (such as build and test tools). You should use the checkout action any time your workflow will run against the repository's code.
	This step uses the `actions/setup-node@v2` action

```
- uses: actions/setup-node@v2
  with:
    node-version: '14'
```

to install the specified version of the Node.js (this example uses v14). This puts both the `node` and `npm` commands in your `PATH`.

```
- run: npm install -g bats
```

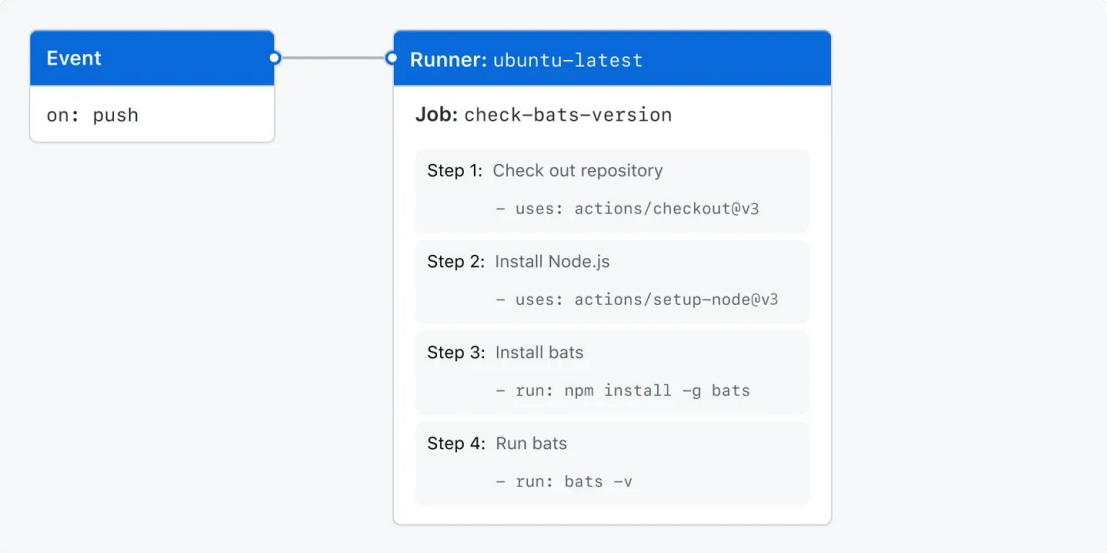
The `run` keyword tells the job to execute a command on the runner. In this case, you are using `npm` to install the `bats` software testing package.

```
- run: bats -v
```

Finally, you'll run the `bats` command with a parameter that outputs the software version.

Visualizing the workflow file

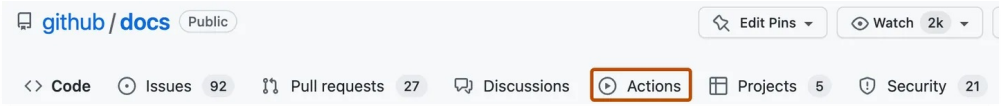
In this diagram, you can see the workflow file you just created and how the GitHub Actions components are organized in a hierarchy. Each step executes a single action or shell script. Steps 1 and 2 run actions, while steps 3 and 4 run shell scripts. To find more prebuilt actions for your workflows, see "Finding and customizing actions."



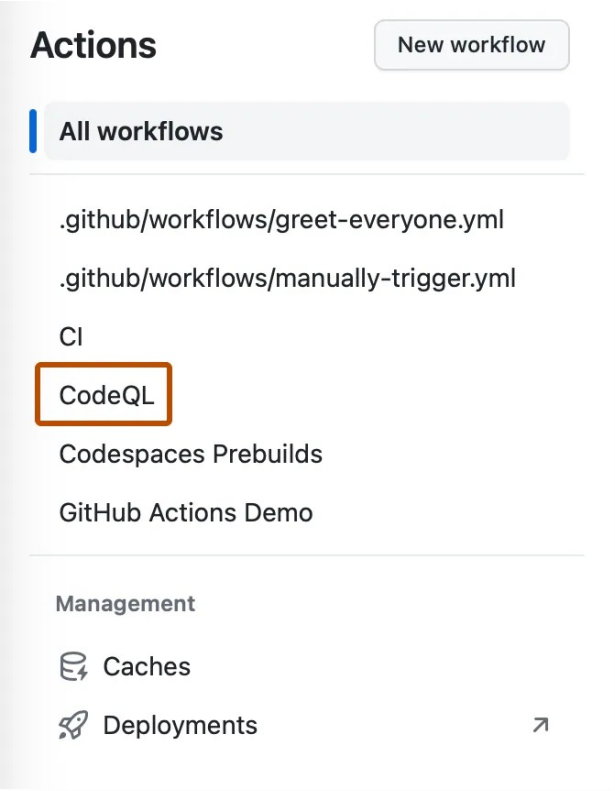
Viewing the activity for a workflow run

When your workflow is triggered, a *workflow run* is created that executes the workflow. After a workflow run has started, you can see a visualization graph of the run's progress and view each step's activity on GitHub.

1. On your GitHub Enterprise Server instance, navigate to the main page of the repository.
2. Under your repository name, click **Actions**.



3. In the left sidebar, click the workflow you want to see.



4. From the list of workflow runs, click the name of the run to see the workflow run summary.
5. In the left sidebar or in the visualization graph, click the job you want to see.
6. To view the results of a step, click the step.

For more on managing workflow runs, such as re-running, cancelling, or deleting a workflow run, see "Managing workflow runs."

Using starter workflows

GitHub provides preconfigured starter workflows that you can customize to create your own continuous integration workflow. GitHub Enterprise Server analyzes your code and shows you CI starter workflows that might be useful for your repository. For example, if your repository contains Node.js code, you'll see suggestions for Node.js projects. You can use starter workflows as a starting place to build your custom workflow or use them as-is.

You can browse the full list of starter workflows in the `actions/starter-workflows` repository on your GitHub Enterprise Server instance.

For more information on using and creating starter workflows, see "Using starter workflows" and "Creating starter workflows for your organization."

Advanced workflow features

This section briefly describes some of the advanced features of GitHub Actions that help you create more complex workflows.

Storing secrets

If your workflows use sensitive data, such as passwords or certificates, you can save

these in GitHub as *secrets* and then use them in your workflows as environment variables. This means that you will be able to create and share workflows without having to embed sensitive values directly in the workflow's YAML source.

This example job demonstrates how to reference an existing secret as an environment variable, and send it as a parameter to an example command.

```yaml
jobs:
  example-job:
    runs-on: ubuntu-latest
    steps:
      - name: Retrieve secret
        env:
          super_secret: ${{ secrets.SUPERSECRET }}
        run: |
          example-command "$super_secret"
```

For more information, see "Encrypted secrets."

Creating dependent jobs

By default, the jobs in your workflow all run in parallel at the same time. If you have a job that must only run after another job has completed, you can use the `needs` keyword to create this dependency. If one of the jobs fails, all dependent jobs are skipped; however, if you need the jobs to continue, you can define this using the `if` conditional statement.

In this example, the `setup`, `build`, and `test` jobs run in series, with `build` and `test` being dependent on the successful completion of the job that precedes them:

```yaml
jobs:
  setup:
    runs-on: ubuntu-latest
    steps:
      - run: ./setup_server.sh
  build:
    needs: setup
    runs-on: ubuntu-latest
    steps:
      - run: ./build_server.sh
  test:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - run: ./test_server.sh
```

For more information, see "Using jobs in a workflow."

Using a matrix

A matrix strategy lets you use variables in a single job definition to automatically create multiple job runs that are based on the combinations of the variables. For example, you can use a matrix strategy to test your code in multiple versions of a language or on multiple operating systems. The matrix is created using the `strategy` keyword, which receives the build options as an array. For example, this matrix will run the job multiple times, using different versions of Node.js:

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node: [12, 14, 16]
```

```
    steps:
      - uses: actions/setup-node@v2
        with:
          node-version: ${{ matrix.node }}
```

For more information, see "Using a matrix for your jobs."

Using databases and service containers

If your job requires a database or cache service, you can use the `services` keyword to create an ephemeral container to host the service; the resulting container is then available to all steps in that job and is removed when the job has completed. This example demonstrates how a job can use `services` to create a `postgres` container, and then use `node` to connect to the service.

```
jobs:
  container-job:
    runs-on: ubuntu-latest
    container: node:10.18-jessie
    services:
      postgres:
        image: postgres
    steps:
      - name: Check out repository code
        uses: actions/checkout@v2
      - name: Install dependencies
        run: npm ci
      - name: Connect to PostgreSQL
        run: node client.js
        env:
          POSTGRES_HOST: postgres
          POSTGRES_PORT: 5432
```

For more information, see "Using containerized services."

Using labels to route workflows

If you want to be sure that a particular type of runner will process your job, you can use labels to control where jobs are executed. You can assign labels to a self-hosted runner in addition to their default label of `self-hosted`. Then, you can refer to these labels in your YAML workflow, ensuring that the job is routed in a predictable way. GitHub-hosted runners have predefined labels assigned.

This example shows how a workflow can use labels to specify the required runner:

```
jobs:
  example-job:
    runs-on: [self-hosted, linux, x64, gpu]
```

A workflow will only run on a runner that has all the labels in the `runs-on` array. The job will preferentially go to an idle self-hosted runner with the specified labels.

To learn more about self-hosted runner labels, see "Using labels with self-hosted runners."

Reusing workflows

You can share workflows with your organization, publicly or privately, by calling one workflow from within another workflow. This allows you to reuse workflows, avoiding duplication and making your workflows easier to maintain. For more information, see "Reusing workflows."

Using environments

You can configure environments with protection rules and secrets to control the execution of jobs in a workflow. Each job in a workflow can reference a single environment. Any protection rules configured for the environment must pass before a job referencing the environment is sent to a runner. For more information, see "Using environments for deployment."