JANUARY 16, 2025 / #GITHUB

Learn to Use GitHub Actions: a Step-by-Step Guide

Rajdeep Singh

GitHub. Actions help you automate, build, test, and deploy your app from your GitHub. They also help you perform code reviews and tests, manage branches, triage issues, and more.

In simple terms, the GitHub workflow creates an environment (virtual machine-based on the **runner**) to test, build, and deploy your code into the cloud based on the action that you describe in the GitHub Action file.

This tutorial teaches you how to add a GitHub Action, providing an example and step-by-step guidance. It is suitable for both beginners and intermediate developers.

Table of Contents:

knowledge of how to use GitHub and YAML. If you don't know GitHub fundamentals, check out this in-depth tutorial on Git and GitHub. And here's an introduction to YAML.

You'll also need to understand the main concepts behind **events**, **workflows**, **jobs**, and **runners** and why they're important when creating a GitHub Action.

These are the key ingredients of GitHub actions, so we'll go through them one by one before diving into the primary part of the tutorial.

Key GitHub Actions Concepts

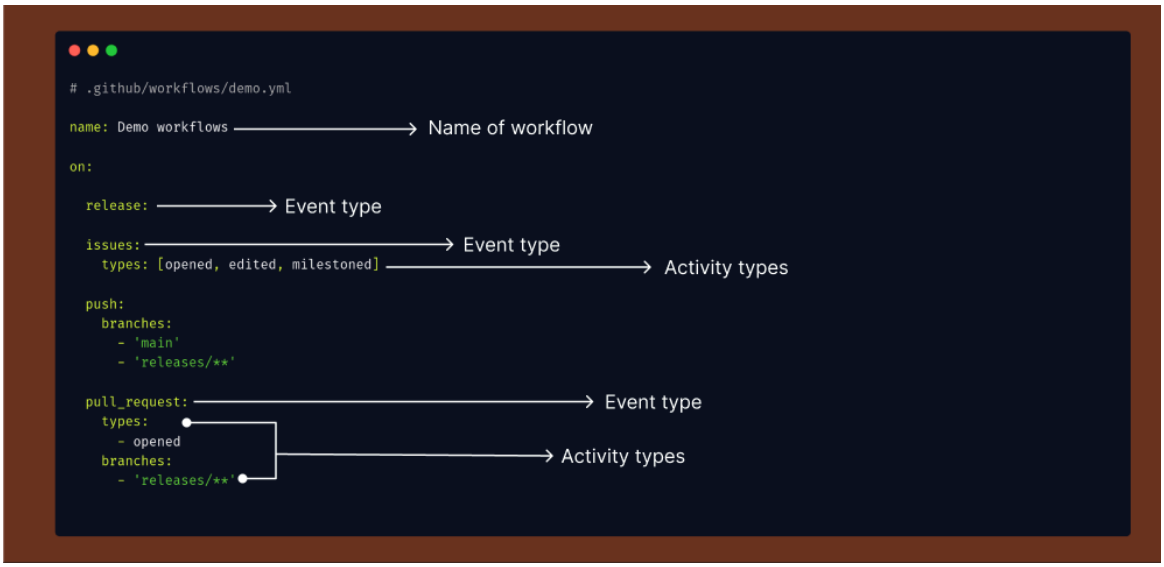
Workflows

A workflow is a configurable automated process that runs one or more jobs. It is created with a YAML file in your repository and runs when an event triggers it. Workflows can also be triggered manually or on a defined schedule.

Workflows are defined in the `.github/workflows` directory in a repository. In the repository, you can create multiple workflows that perform different tasks, such as:

1. Building and testing pull requests
2. Deploying your application on the cloud
3. Running a test on every pull request

workflow in your GitHub repository. For example, when you push code to the repository, it triggers the `push` event. The same happens when you create a new issue – it triggers the `issues` event. And when somebody makes a pull request in your repository, it triggers the `pull_request` event.



These are some common GitHub Action events:

1. Push
2. pull_request
3. release
4. label
5. issues
6. milestone
7. label

documentation.

It's a good idea to specify the event type in a GitHub Action. For example, specifying the `pull_request` event will trigger the action whenever any user creates a pull request in the GitHub repository.

```
# .github/workflows/demo.yml

on:
  issues:
    types: [opened, edited, milestoned]

  pull_request:
    types:
      - opened
    branches:
      - 'releases/**'
```

This is helpful because, if you don't declare a specific event activity type in your event type, it can lead to unnecessary resources getting used. The GitHub Action will be triggered with every new pull request – so it's best to define which type of event you're using.

Jobs

GitHub Action jobs run in parallel by default. A GitHub Action workflow runs one or more jobs, each containing a set of steps that execute commands or actions. Here's an example:

```
# .github/workflows/demo.yml
```

```
# A workflow run is made up of one or more jobs that can run sequentially
jobs:

jobs:
```

You can set one job to depend on (an)other job(s). If jobs don't have dependencies, they'll run in parallel. When one job depends on another, it will wait for that job to finish before it starts.

```
# .github/workflows/demo.yml

jobs:
  build:
    name: Build
    needs: [ Development ]
    steps:
      - name: Build and deploy on Cloud
  dev:
    name: Development
    steps:
      - name: Run the developer

  Test:
    needs: [ build, dev ]
    name: Testing
    steps:
      - name: Testing the application
```

Runners

Ubuntu Linux, Microsoft Windows, and macOS to run your workflows.

```yaml
# .github/workflows/demo.yml

name: Demo workflows

on:
  # Triggers the workflow on push or pull request events but only for the
   push:
    branches: [ "main" ]

# A workflow run is made up of one or more jobs that can run sequentially
jobs:
  # This workflow contains a single job called "build"
  build:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest
```

To define the runners, specify the runner value in the `runs-on` option. You can provide it as a **single string** or an **array of strings**.

```yaml
# .github/workflows/demo.yml

# String
runs-on: ubuntu-latest
# Array of string
runs-on: [ ubuntu-latest, windows-latest, macos-latest ]
```

Now that you're familiar with the key elements of GitHub Actions and how they work, let's see how to use Actions in practice.

Your Repository

You can create a GitHub Action in GitHub very easily. There are two ways to do it:

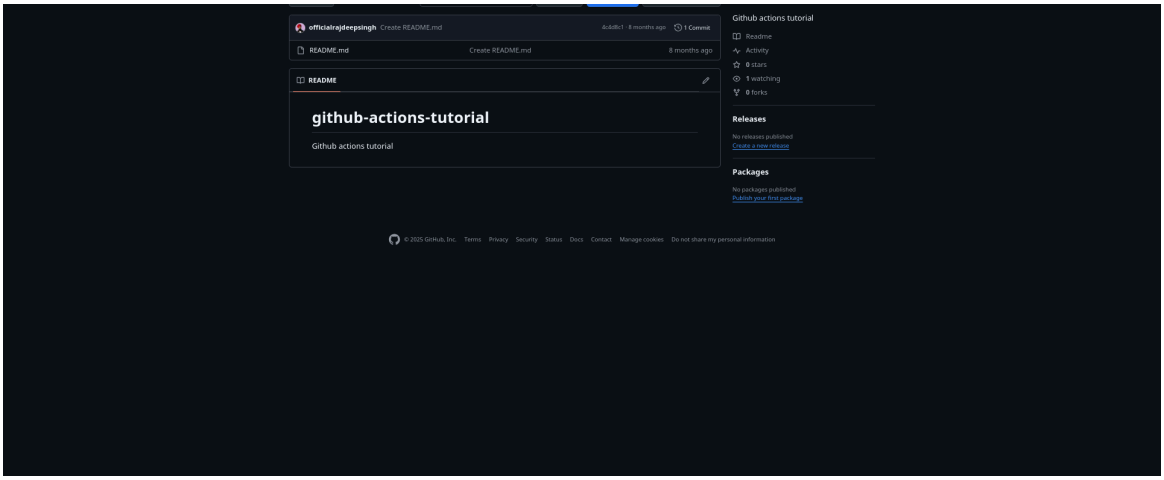
1. Using the Github UI
2. Locally with your IDE

Many developers use the GitHub UI to create an Action. This is a common way to create an Action. You don't need to create a `.github/workflow` folder when you use the GitHub UI. GitHub automatically creates this folder for you. On the other hand, for complex Github actions, you'll typically use your IDE.

Let’s look at each approach now.

Create a GitHub Action Using the GitHub UI

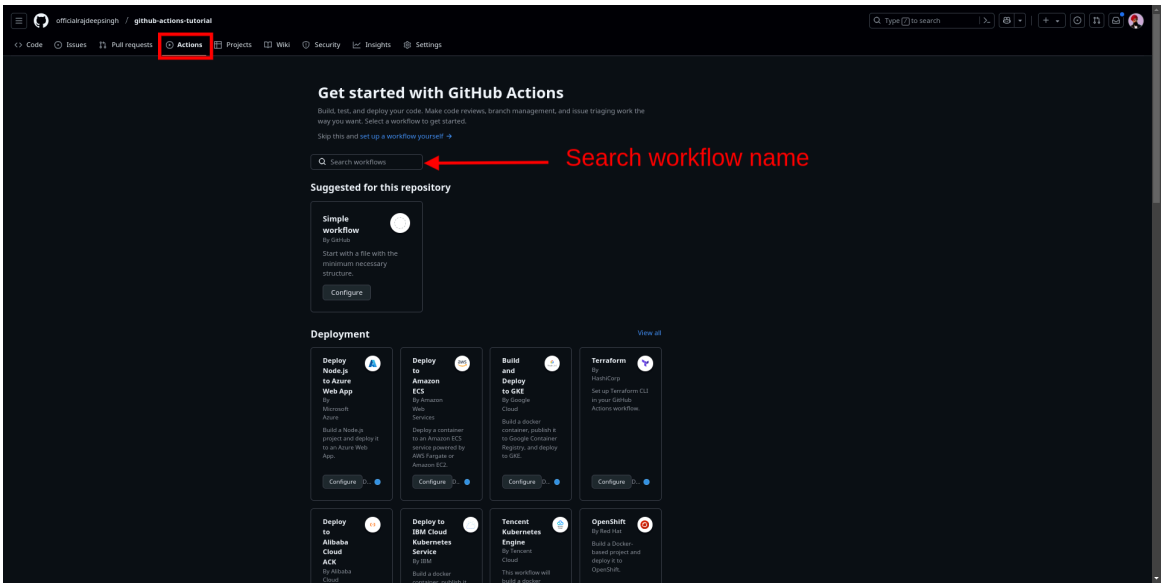
First, go to the GitHub repository where you want to create your GitHub Action.



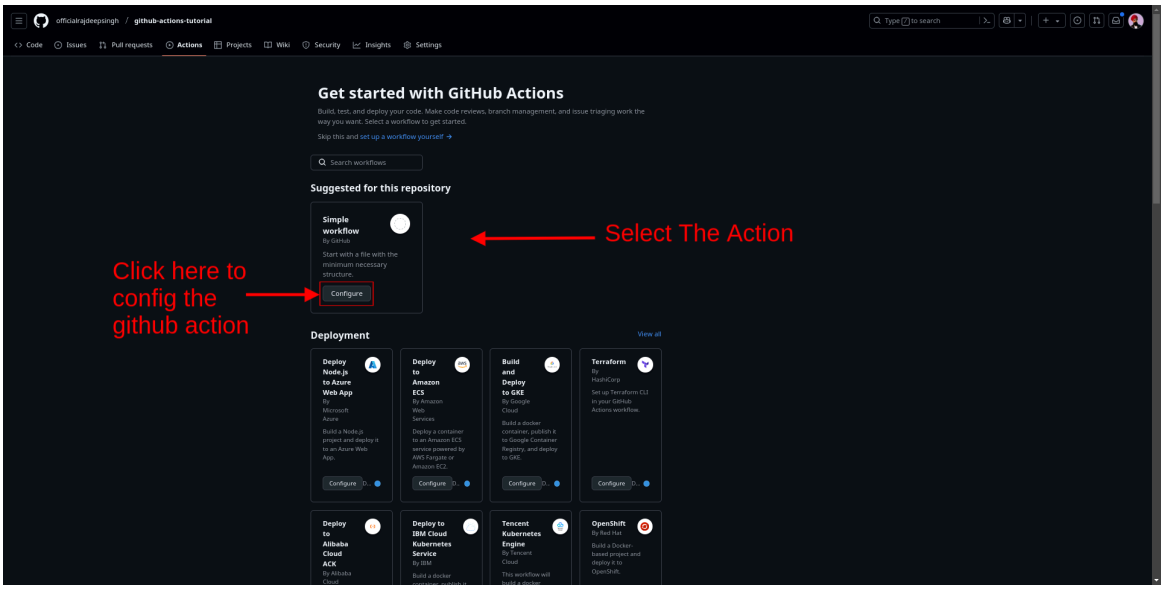
To create the action, follow these steps:

1. Click on the Action Tab

Click on the Action tab to create a GitHub Action. You'll see the following page:

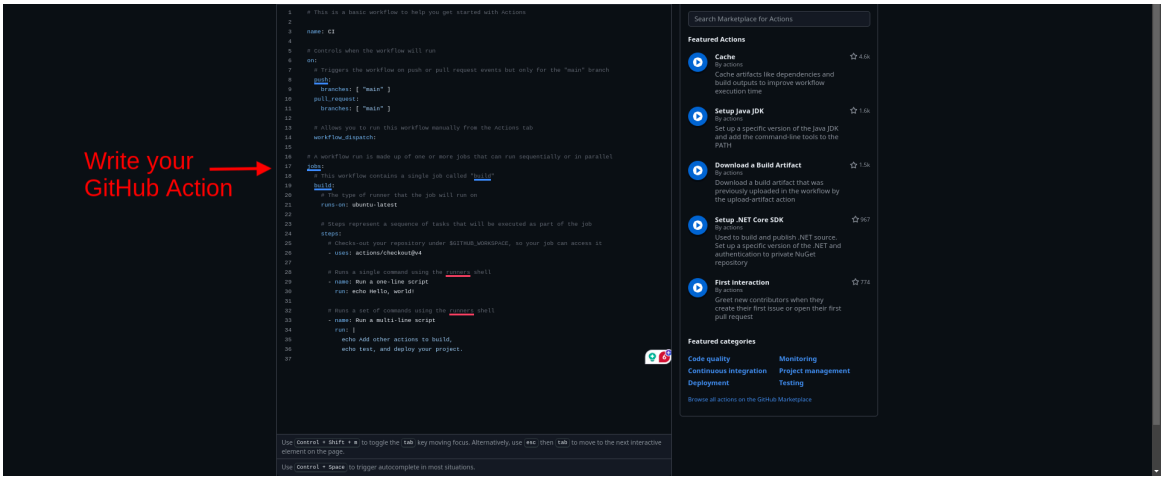


your project. Select the GitHub workflow and click on the configure button to create your action.



3. Create the GitHub workflow

You'll see the following page where you can edit and create your action. Click on the commit change button to save the action.



And that's it – you've created your GitHub Action.

Create a GitHub Action Locally with Your IDE

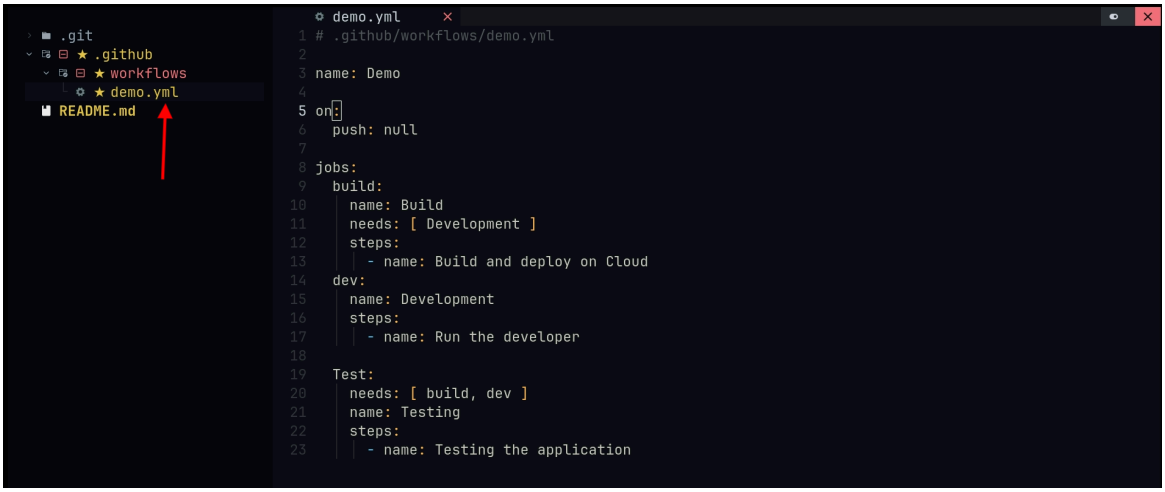
First, open your project in your current IDE, such as VS Code, Neovim, Vim, or whatever. Then, create a `.github/workflow/name-of-workflow.yml` file in your project. Copy and paste the following code and save and push your local code into the GitHub repository.

Following the GitHub workflow action example code is printed a hello world message.

```
# .github/workflows/demo.yml

name: CI

# Controls when the workflow will run
on:
  # Triggers the workflow on push or pull request events but only for the
  push:
```

```
# This workflow contains a single job called "build"
build:
  # The type of runner that the job will run on
  runs-on: ubuntu-latest

  # Steps represent a sequence of tasks that will be executed as part c
  steps:
    # Checks out your repository under $GITHUB_WORKSPACE, so your job c
    - uses: actions/checkout@v4

    # Runs a single command using the runners shell
    - name: Run a one-line script
      run: echo Hello, world!
```

I'm using the Neovim IDE to create a `.github/workflow/demo.yml` file. It looks like this.



GitHub Actions Syntax

syntax you'll use to create your Actions.

We'll work with this example Action and go through it part by part below:

```yaml
# .github/workflows/demo.yml

name: Github Action Template

on:

  pull_request:
    branches: [ "main" ]

  schedule:
    - cron:  '30 5,17 * * *'

  workflow_call:
    inputs:
      username:
        description: 'A username passed from the caller workflow'
        default: 'john-doe'
        required: false
        type: string

  permissions:
    actions: read|write|none

  # permissions : read|write|none

# A workflow run is made up of one or more jobs that can run sequentially
jobs:

  # This workflow contains a single job called "build"

  build:

    runs-on: ubuntu-latest
```

```
# Checks-out your repository under $GITHUB_WORKSPACE, so your job c
- uses: actions/checkout@v4
  if: ${{ github.event_name == 'pull_request' && github.event.actic
  shell: zsh
  name: NPM Install Package
  run: npm install
  with:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    first_name: Github
    last_name: Action
    args: The ${{ github.event_name }} event triggered this step.
    entrypoint: /bin/echo
```

Now, let's understand each option that you can see in this GitHub Action example workflow:

1. `name` : The name describes the workflow name.
2. `pull_request` : The pull request is part of the event type. It means somebody added a pull request in your repository and the following workflow was run.
3. `schedule` : With a schedule, you can define the time schedule in your workflows. You can schedule a workflow to run at certain tasks on specific UTC times or based on intervals after five minutes, and so on.
4. `workflow_call` : This defines the inputs and outputs for a reusable workflow.
5. `permissions` : In GitHub, certain tasks need special permissions when working with the GitHub app and GitHub API. For example, for issues, `write` permission permits a user

6. `jobs` : The `jobs` option runs one or more jobs in your GitHub Action, each containing a set of steps that execute commands or actions.
7. `runs-on` : The `runs-on` option defines the type of machine to run the job on.
8. `steps` : The jobs contain a sequence of tasks called `steps` . Steps can run commands, set tasks, or an action in your repository.
9. `name` : The name option is used to set the name of the job, which is displayed in the GitHub UI.
10. `if` : the `if` option works similarly to an if conditional. It prevents a step from running unless a condition is met.
11. `shell` : The `shell` option allows you to define a custom shell.
12. `run` : The `run` option helps run commands in the operating system's shell. For example, `run : ls` , `run : pwd` , and so on.
13. `uses` : With the `uses` option, you can run reusable units of code or other packages. You usually use it to run a GitHub package published by another developer on the GitHub marketplace. Most package developers use JavaScript or Docker container files.
14. `with` : the `with` option accepts a value as a map with a key/value pair. It has two sub-options: `args` and an `entrypoint` . The entrypoint is used to define the entry file for Dockerfile. The args option will be passed to the container's entrypoint.

For advanced GitHub Action syntax, you can check out the Github documentation.

GitHub Actions Examples

To better understand how GitHub Actions work, let's build four examples of a GitHub Action workflow. These are common examples that many developers use and will teach you how GitHub Actions work.

Node Setup

In the following GitHub Action, we'll set up a Node.js environment for our application. Once you've done that, you can test and deploy your Node.js application.

```yaml
# .github/workflows/nodejs.yml

name: Setup Node.js Env

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v4
      with:
```

```
      - run: npm test
```

For our example, we're running our action on an Ubuntu machine. The GitHub action is triggered whenever you (or someone) push code into the repository. The `actions/checkout@v4` extension sets the `$GITHUB_WORKSPACE` environment variable to your working directory.

The `actions/setup-node@v4` extension sets up the Node.js environment, and the GitHub `run` option executes the Linux command.

Deno Setup

In the following GitHub Action, we'll set up a Deno environment for our application. You can test and analyse (using deno lint) the code for errors, stylistic issues, and so on.

```
name: Deno

on:
  push:
    branches: ["main"]

permissions:
  contents: read

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - name: Setup repo
```

```
      with:
        deno-version: v2.1.5

    - name: Run linter
      run: deno lint

    - name: Run tests
      run: deno test -A
```

For this example, we're running our action on an Ubuntu machine. The GitHub action is triggered whenever you (or someone) push code into the repository. The `actions/checkout@v4` extension sets the `$GITHUB_WORKSPACE` environment variable to your working directory.

The `denoland/setup-deno@v2` extension sets up the Deno environment and the GitHub `run` option executes the Linux command.

Zip Files

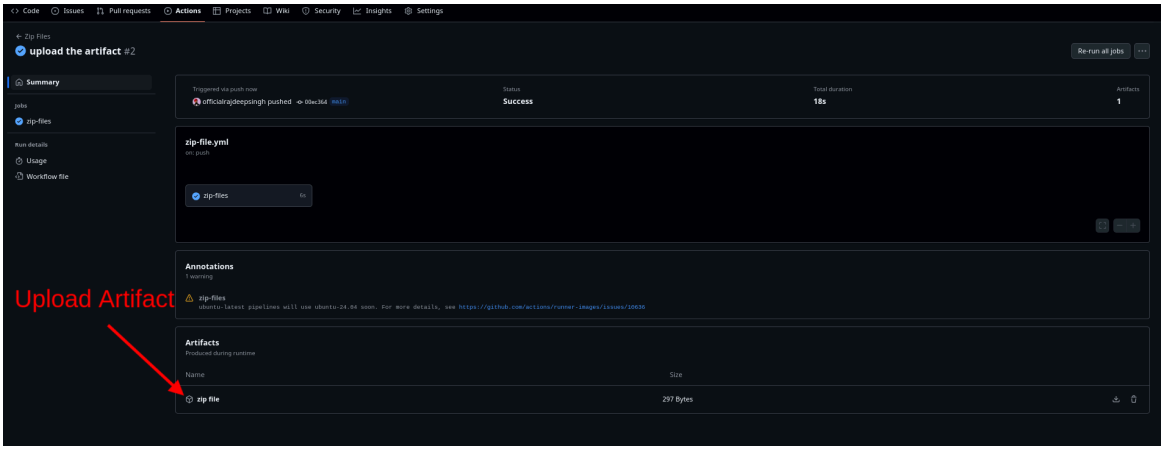
In the following example, we'll combine the `dist` folder and the `manifest.json` file into a zip archive. Then we'll save the zipped file as an artifact for later use or download:

```
name: Zip Files

on:
  release:
    types: [published]

jobs:
  zip-files:
```

```
- uses: vimtor/action-zip@v1.2
  with:
    files: dist/ manifest.json
    dest: build.zip

 - uses: actions/upload-artifact@v4
   with:
     name: zip file
     path: ${{ github.workspace }}/build.zip
```

For this example, we're running our action on an Ubuntu machine. The GitHub Action is triggered whenever someone pushes code into the repository. The `actions/checkout@v4` extension sets the `$GITHUB_WORKSPACE` environment variable to your working directory.

The `vimtor/action-zip@v1.2` extension or package converts files into a zip folder. The `actions/upload-artifact@v4` package uploads artifacts during a workflow run.



Deploy a Static Website

```yaml
# Simple workflow for deploying static content to GitHub Pages
name: Deploy static content to Pages

on:
  # Runs on pushes targeting the default branch
  push:
    branches: ["main"]

# Sets permissions of the GITHUB_TOKEN to allow deployment to GitHub Page
permissions:
  contents: read
  pages: write
  id-token: write

# Allow only one concurrent deployment, skipping runs queued between the
# However, do NOT cancel in-progress runs as we want to allow these produ
concurrency:
  group: "pages"
  cancel-in-progress: false

jobs:

  # Single deploy job since we're just deploying
  deploy:
    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}
    runs-on: ubuntu-latest
    steps:

      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Pages
        uses: actions/configure-pages@v5

      - name: Upload artifact
        uses: actions/upload-pages-artifact@v3
        with:
          # Upload entire repository
```

```
        uses: actions/deploy-pages@v4
```

For this example, we're running our action on an Ubuntu machine. The GitHub action is triggered whenever you push code to the repository. The `actions/checkout@v4` extension sets the `$GITHUB_WORKSPACE` environment variable to your working directory.

The `actions/configure-pages@v5` package helps you configure GitHub Pages and allows you to gather metadata about your website. For more detail, refer to the configure-pages action documentation.

The `actions/upload-pages-artifact@v3` package helps you to package and upload artifacts that can be deployed to GitHub Pages.

The `actions/deploy-pages@v4` package is used to deploy your website to GitHub Pages.

Conclusion

Github Actions is a large topic. To more fully understand them, you can start with a basic Action example and then move on to more advanced Actions.

When you're using Github Actions, the biggest problem is waiting for the results. For example, creating and updating the date on which the GitHub Action file pushes code into GitHub and then waiting for the GitHub Action result. It can be a time-consuming task, so you can use

I have published an in-depth article on freeCodeCamp on how to use the Act CLI tool if you want to read more about that.

Thanks for reading!

Rajdeep Singh

JavaScript || Reactjs || Nextjs || Rust || Biotechnology || Bioinformatic || Front-end Developer || Author || https://ko-fi.com/officialrajdeepsingh

---

Learn to code — free 3,000-hour curriculum

Trending Books and Handbooks

REST APIs
JavaScript
GraphQL APIs
REST API Design
Linux
Docker
Node.js
Front-End Libraries
Clustering in Python
Coding Career Preparation
Clean Code
AI Chatbots
CSS Transforms
PHP
React
Golang
Todo APIs
Express and Node.js
Software Architecture
Full-Stack Developer Guide
TypeScript
Command Line
Access Control
Java
CI/CD
Python
JavaScript Classes
Python Code Examples
Programming Fundamentals
Python for JavaScript Devs

Mobile App

Our Charity