**Github Action**

**1. Github Action:** compliance

GitHub Actions allows you to automate, customize, and execute your software development workflows directly in your GitHub repository. Here’s a breakdown of key concepts and how to set up workflows in GitHub Actions:

**Key Concepts**

1. **Workflow**: A workflow is an automated process defined by a YAML file stored in the .github/workflows directory of your repository. It can be triggered by events such as pushes, pull requests, or on a schedule.
2. **Job**: A workflow can contain one or more jobs, which are a set of steps executed in a specific order. Jobs run on virtual environments (runners) that GitHub provides.
3. **Step**: A step is a single task performed as part of a job. Each step can run commands, use actions (reusable code packages), or scripts.
4. **Action**: An action is a reusable piece of code that can be shared and used in workflows. You can create your own actions or use actions shared by the community.
5. **Event**: An event is a specific activity that triggers a workflow, like a push or pull\_request.

**About billing for GitHub Actions**

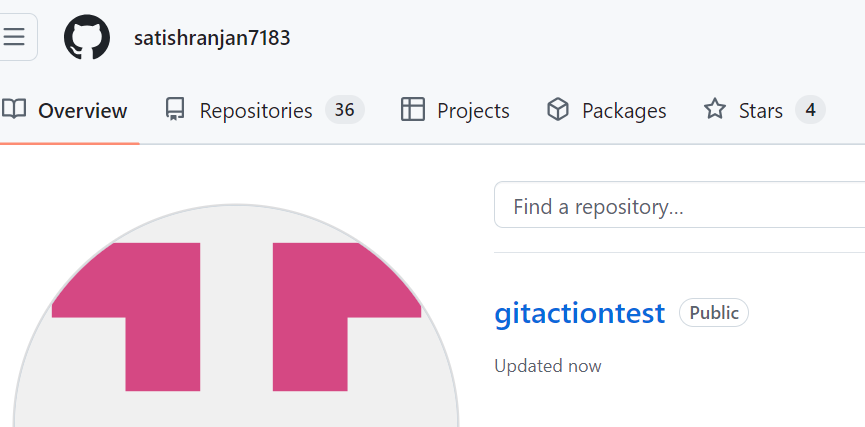
https://docs.github.com/en/billing/managing-billing-for-your-products/managing-billing-for-github-actions/about-billing-for-github-actions

[**Managing GitHub Actions settings for a repository - GitHub Docs**](https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/enabling-features-for-your-repository/managing-github-actions-settings-for-a-repository)**:**

<https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/enabling-features-for-your-repository/managing-github-actions-settings-for-a-repository>

Step to create First CICD pipeline using GIT Action

Step1: Need to create one Repository and put all code there in repository.



Step2: Go inside the repository, and then create Action

A screenshot of a computer

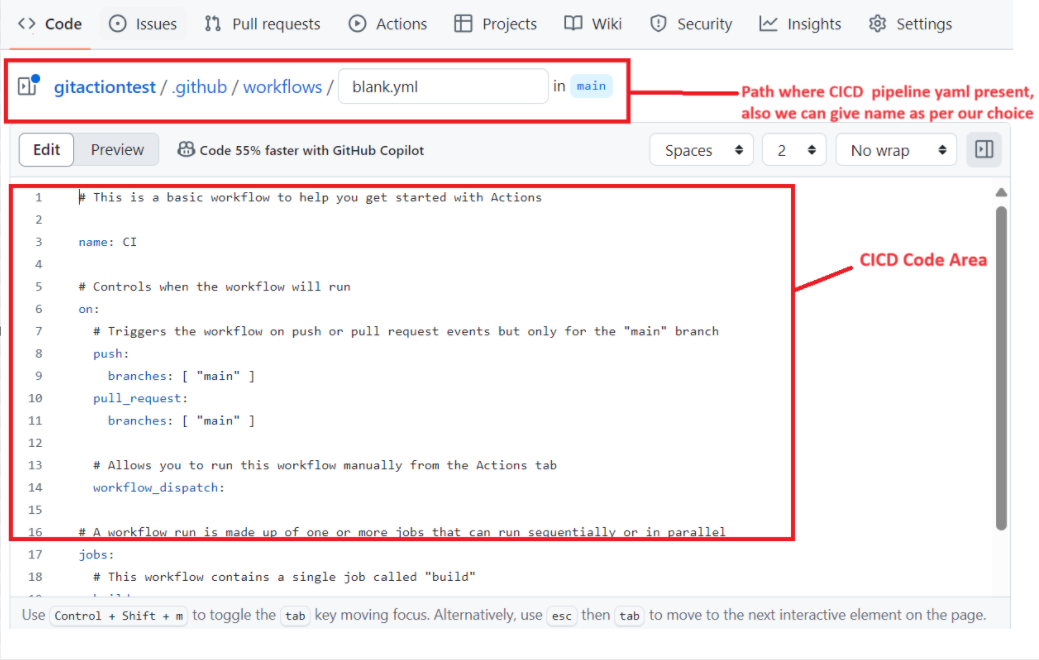
Description automatically generated

Step3: There are multiple deployment option available, but first time we work on simple workflow.

A screenshot of a computer

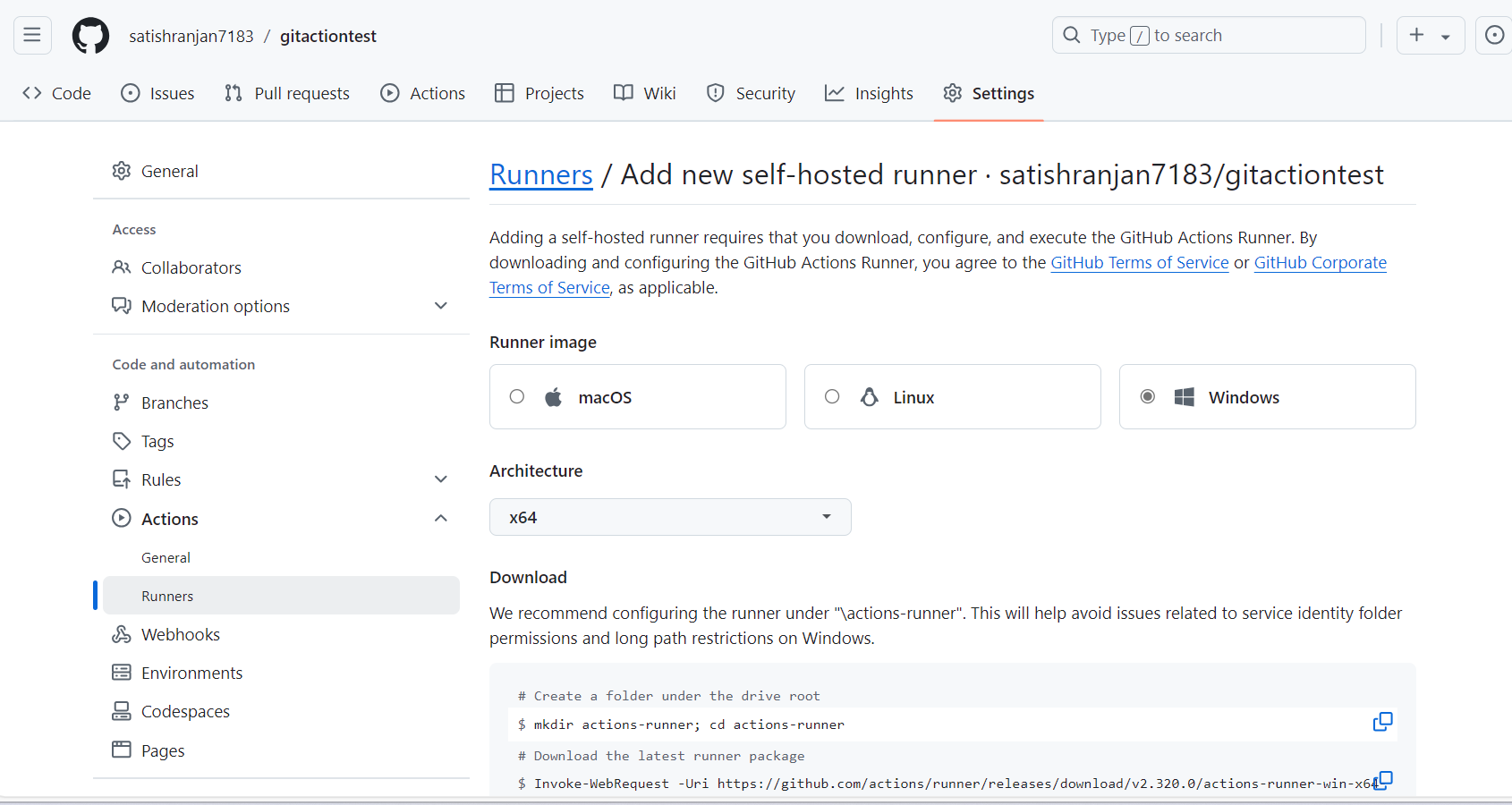
Description automatically generated

Step4: Path of cicd Yaml, also we can write cicd pipeline code.



Step5: Download Self-Agent host and configure in our laptop

Go to Setting-> Click on Runners 🡪 click on Windows (for window self agent host) 🡪 Then follow the instruction for self agent host.



Step6: Follow below step

A screenshot of a computer error

Description automatically generated

A screenshot of a computer

Description automatically generated

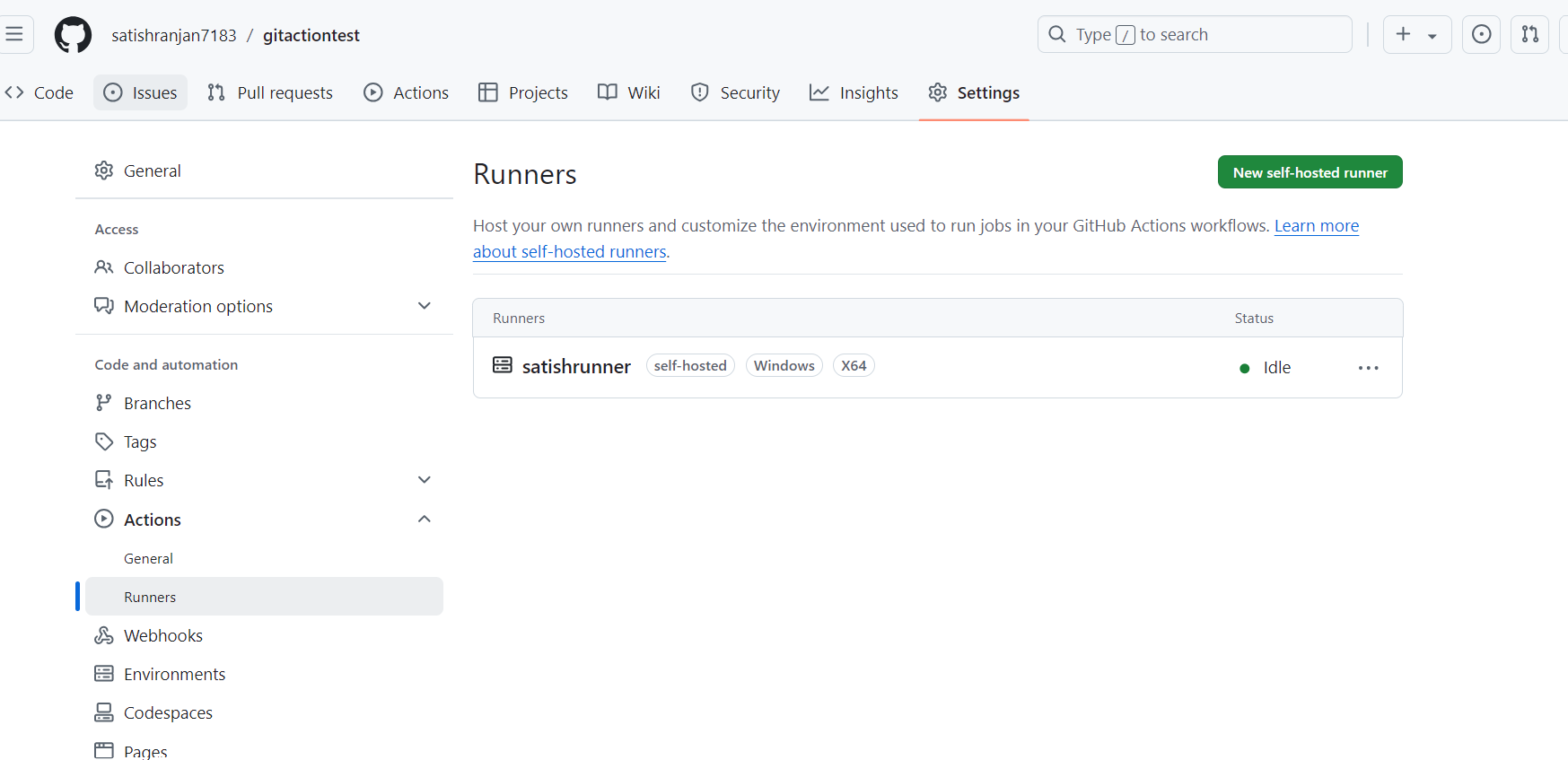
A screenshot of a computer program

Description automatically generated

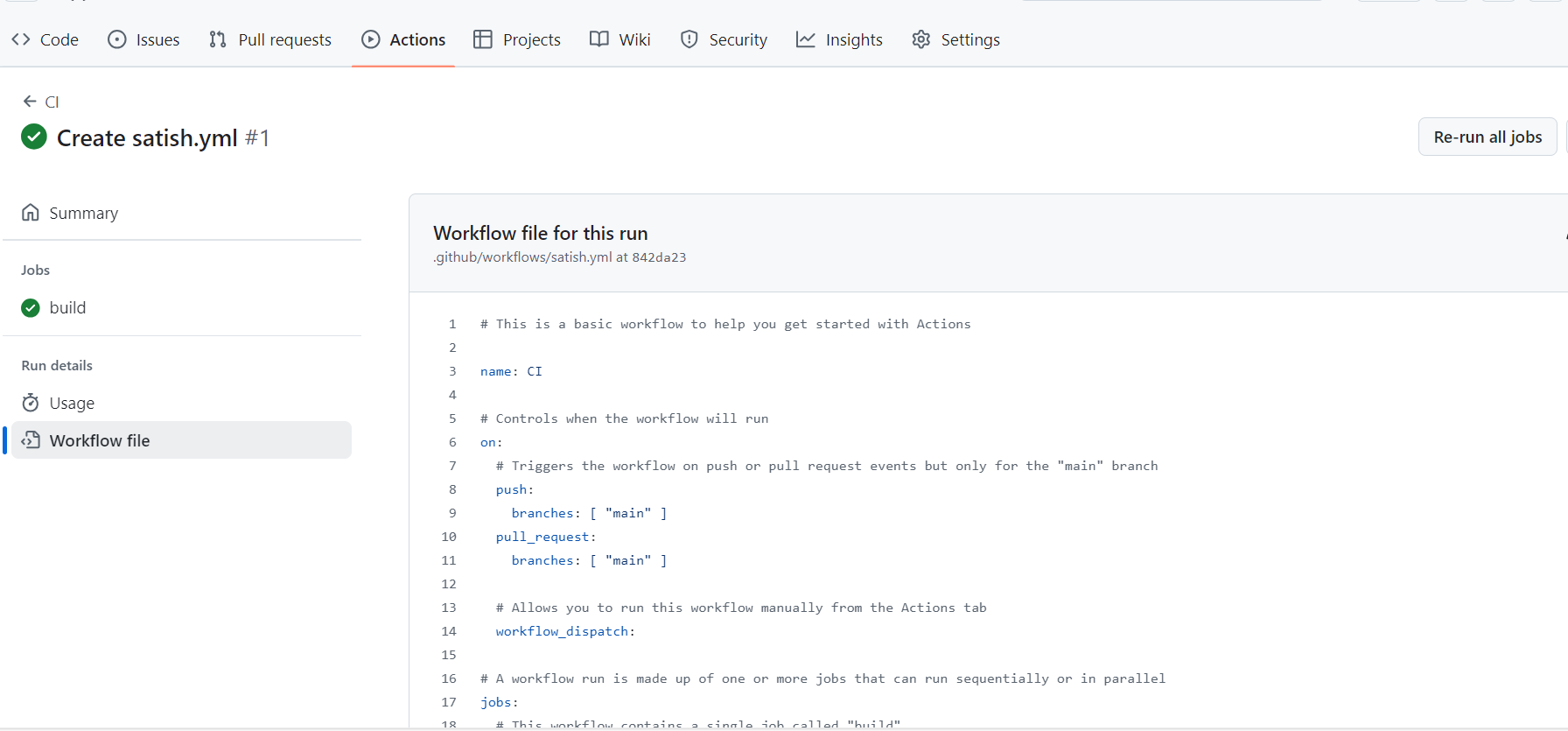
A black screen with white text

Description automatically generated

Step7 : Now we can check our runner is online. We can check in setting -> action -> runner.



Step8: Now we can run our pipeline.



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

When working with GitHub Actions for CI/CD pipelines, it's helpful to understand some key terms. Here’s a list of important concepts:

**1. Workflow**

A workflow is a set of automated processes defined in a YAML file (usually located in the .github/workflows directory). It specifies what actions to run and when.

**2. Trigger**

Triggers are events that initiate a workflow. Common triggers include:

* **push**: When code is pushed to a branch.
* **pull\_request**: When a pull request is opened or updated.
* **schedule**: At specific times (like a cron job).

**3. Job**

A job is a collection of steps that run in a specific environment. Jobs can run sequentially or in parallel. Each job runs on its own runner.

**4. Step**

A step is an individual task within a job. Steps can run commands or use actions. Each step is executed in the order they are defined.

**5. Action**

Actions are reusable units of code that perform specific tasks (like checking out code, installing dependencies, or deploying). You can use existing actions from the GitHub Marketplace or create your own.

**6. Runner**

A runner is a server that runs your workflows. GitHub provides hosted runners, or you can set up self-hosted runners on your own machines.

**7. Environment**

An environment is a specific deployment context. You can define different environments (like staging, production) and configure deployment rules and protection settings.

**8. Secret**

Secrets are sensitive information (like API keys or passwords) that you can store securely in GitHub. They are accessed in workflows using the secrets context.

**9. Matrix Builds**

Matrix builds allow you to run multiple jobs in parallel with different configurations (like different Node.js versions or operating systems) using a single job definition.

**10. Artifacts**

Artifacts are files produced by your workflow (like build outputs or test reports) that you can save and access after the workflow completes.

**11. Context**

Contexts are variables available in workflows that provide information about the workflow run, repository, event, and more. Common contexts include github, env, and secrets.

**12. Cache**

Caching helps speed up workflow runs by storing dependencies (like package files) between workflow executions. You can use the actions/cache action for this purpose.

**Summary**

These terms are foundational for understanding how to set up and manage CI/CD pipelines using GitHub Actions. Familiarity with these concepts will help you create efficient and effective automation workflows.

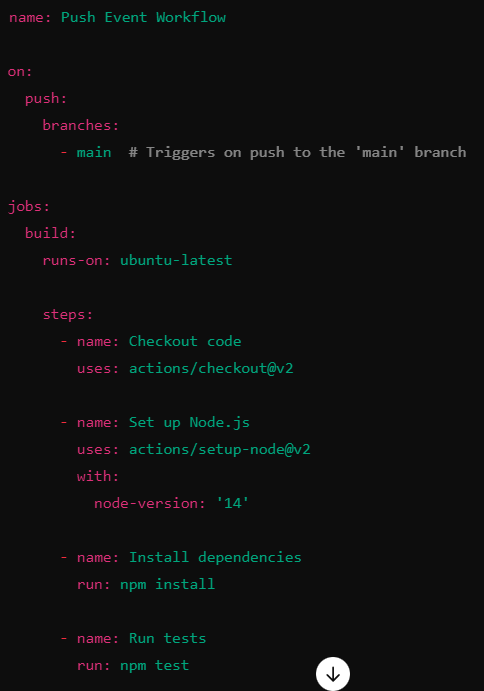
[**Managing GitHub Actions terms**](https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/enabling-features-for-your-repository/managing-github-actions-settings-for-a-repository)**:**

**The Push Event** : The **push event** in GitHub Actions triggers workflows whenever a commit is pushed to a repository. This is a common event used for continuous integration (CI) processes, allowing you to automatically run tests, build applications, or deploy code whenever new changes are added.

**Breakdown of the Workflow**

* **name**: The name of the workflow.
* **on**: This specifies the event that triggers the workflow. Here, it's set to push, and only on the main branch.
* **jobs**: This section defines what jobs will be executed. In this case, there's one job called build.
* **runs-on**: Specifies the environment where the job will run (e.g., ubuntu-latest).
* **steps**: A series of steps to execute, including:
  + Checking out the code.
  + Setting up Node.js.
  + Installing dependencies.
  + Running tests.

Using the push event in GitHub Actions is an effective way to automate processes in response to code changes. This setup helps maintain code quality and facilitates continuous integration.



**The Action in workflow :** An action is like a small task or a step that helps you automate your work in GitHub. You can think of it as a piece of code that performs a specific function, like checking out your code, running tests, or deploying an application.

**How Do Actions Work in a Workflow?**

1. **Combine Steps**: In a workflow, you combine multiple actions to create a sequence of tasks that run automatically when something happens (like a push to your repository).
2. **Use Pre-Built Actions**: You can use actions that others have already created, which you can find in the GitHub Marketplace. For example, there are actions for testing code, building software, or deploying to different services.
3. **Custom Actions**: You can also create your own actions tailored to your needs.

A screenshot of a computer

Description automatically generated

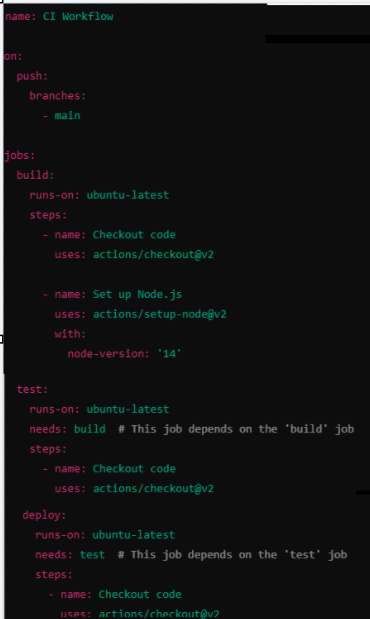
**Breakdown of the Example**

1. **Checkout Code**: The action actions/checkout@v2 pulls your code from GitHub so the workflow can use it.
2. **Set Up Node.js**: The action actions/setup-node@v2 installs Node.js, which your project needs to run.
3. **Install Dependencies**: This step uses a command to install the necessary packages for your project.
4. **Run Tests**: This step runs tests to ensure your code works as expected.

[**Adding**](https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/enabling-features-for-your-repository/managing-github-actions-settings-for-a-repository) **Multiple Jobs:** We can define multiple jobs in a GitHub Actions workflow to perform different tasks in parallel or sequentially. Each job can run independently, allowing you to organize your workflows more efficiently.

Breakdown of the Example

1. Jobs Section:
   * build Job:
     + Runs on ubuntu-latest.
     + Checks out the code, sets up Node.js, installs dependencies, and runs a build command.
   * test Job:
     + Also runs on ubuntu-latest.
     + Depends on the build job (it won’t run until the build job is complete).
     + Checks out the code and runs tests.
   * deploy Job:
     + Runs on ubuntu-latest.
     + Depends on the test job (it only runs after tests are complete).
     + Runs a simple deployment command (you would replace this with your actual deployment script).

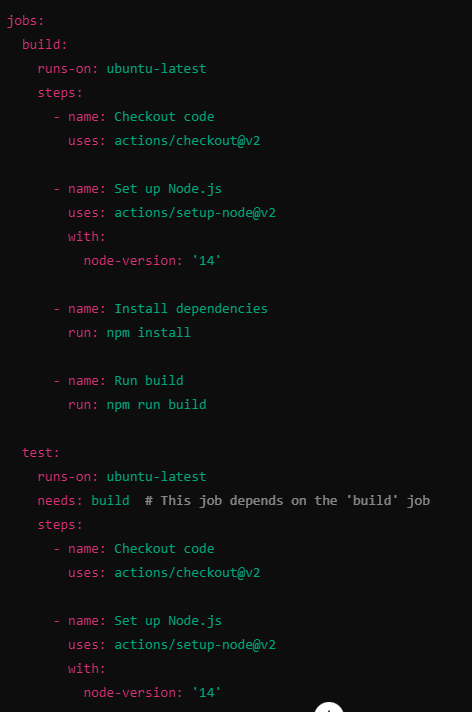


**Important Points**

* **Job Dependencies**: Use the needs keyword to define dependencies between jobs. This ensures that certain jobs run only after specified jobs have completed successfully.
* **Parallel Execution**: By default, jobs run in parallel unless dependencies are defined. This can speed up your workflows significantly.
* **Separate Environments**: Each job runs in its own virtual environment, allowing for separate configurations or dependencies without interference.

**Conclusion :** Using multiple jobs in GitHub Actions allows you to break down your workflows into manageable tasks that can be executed in parallel or in a specific order. This structure is beneficial for CI/CD processes, making them more efficient and organized.

**need :** In GitHub Actions, the needs keyword is used to define job dependencies, ensuring that one job runs only after the completion of one or more specified jobs. This is useful for orchestrating workflows where certain tasks must complete successfully before subsequent tasks begin.

****

**Using Multiple Triggers (Events) :** Using multiple triggers in GitHub Actions allows you to run a workflow in response to different events happening in your repository. Here’s a simple explanation of how it works:

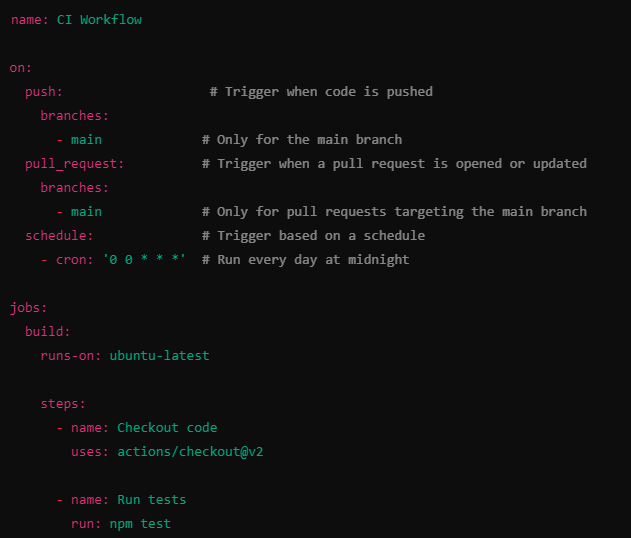
**What Are Triggers?**

Triggers (or events) are actions that start your workflow. Common triggers include:

* **push**: When code is pushed to a branch.
* **pull\_request**: When a pull request is opened, updated, or merged.
* **schedule**: At specific times, similar to a cron job.

**How to Use Multiple Triggers**

You can set up a workflow to respond to more than one event. This means that the same workflow can run when different actions happen.



**Multiple Triggers**:

* The workflow will run when:
  + Code is pushed to the main branch.
  + A pull request is created or updated that targets the main branch.
  + At midnight every day.

**Job Definition**:

* There’s a job named build that runs on the latest version of Ubuntu.

**Steps**:

* The first step checks out the code from the repository.
* The second step runs tests using the npm test command.

**Benefits of Using Multiple Triggers**

* **Flexibility**: You can automate different tasks based on various events without creating separate workflows.
* **Efficiency**: The same workflow can handle multiple scenarios, making it easier to manage.

Using multiple triggers in GitHub Actions lets you create a single workflow that responds to various events, like code pushes, pull requests, or scheduled times. This helps automate your processes more effectively and keeps your workflows organized.

**Expressions & Context Objects :** In GitHub Actions, expressions and context objects are powerful tools that help you access information and control the flow of your workflows. Here’s a simple explanation of both concepts:

**Expressions**

Expressions are snippets of code that let you access and manipulate data in your workflows. They are enclosed in ${{ }} and can include variables, functions, and more.

Example of Expressions

**Accessing Secrets:** You can access secrets stored in your repository:



**Using Conditions**: You can use expressions to run steps conditionally:



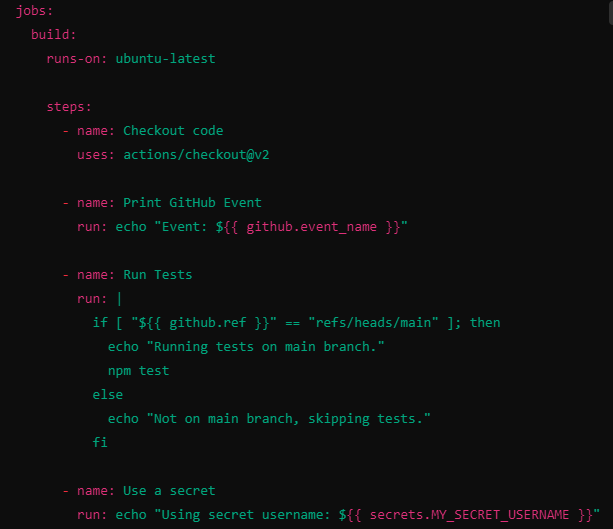
**Context Objects**

Context objects are predefined variables that contain information about the workflow run, the repository, the event that triggered the workflow, and more. They help you access useful data without hardcoding values.

**Common Context Objects**

1. **github**: Contains information about the GitHub event and repository.
   * Example: ${{ github.event\_name }} gives the name of the event that triggered the workflow.
2. **env**: Contains environment variables you’ve defined.
   * Example: ${{ env.MY\_ENV\_VAR }} accesses an environment variable.
3. **secrets**: Contains secrets that you’ve set up in your repository.
   * Example: ${{ secrets.MY\_SECRET }} accesses a secret.
4. **job**: Contains information about the current job.
   * Example: ${{ job.status }} gives the status of the current job (success or failure).
5. **runner**: Contains details about the runner executing the job.
   * Example: ${{ runner.os }} tells you the operating system of the runner (e.g., Linux, Windows).

Example Using Expressions and Context Objects



**Breakdown of the Example**

1. **Printing the Event**: The step prints the event that triggered the workflow using ${{ github.event\_name }}.
2. **Conditional Test Running**: The tests run only if the branch is main. The condition checks ${{ github.ref }} to decide.
3. **Using a Secret**: The workflow accesses a secret username with ${{ secrets.MY\_SECRET\_USERNAME }}.

**Summary:** Expressions and context objects in GitHub Actions allow you to make your workflows dynamic and responsive to different situations. They help you access valuable information about the workflow, control the flow of execution, and securely handle sensitive data.

**Activity Type and Event Filters:**

In GitHub Actions, **Activity Types** and **Event Filters** are used to control when workflows are triggered based on specific conditions or activities within a repository. Understanding these concepts can help you create efficient and tailored automation for your CI/CD processes.

**Activity Types :**

**Activity Types** refer to the different kinds of actions that can trigger workflows. These actions are defined under specific events, such as push, pull\_request, and others. Each event can have different activity types that can be used to refine the triggering conditions.

**Common Activity Types**

Here are some common activity types associated with popular GitHub events:

1. **push**:
   * Triggers on pushes to the repository.
   * Can filter by branches and paths.
2. **pull\_request**:
   * Triggers on pull request actions.
   * Common activity types include opened, edited, synchronize, closed, reopened, labeled, and unlabeled.
3. **issues**:
   * Triggers on actions related to issues.
   * Common activity types include opened, edited, closed, labeled, and unlabeled.
4. **release**:
   * Triggers when a release is created, published, or deleted.
5. **workflow\_dispatch**:
   * Allows manual triggering from the GitHub UI.

**Event Filters**

**Event Filters** allow you to further refine the conditions under which a workflow runs by specifying criteria related to branches, tags, or paths.

**Common Filters**

1. **Branches**:
   * You can specify which branches should trigger the workflow.

A black screen with white text

Description automatically generated

1. **Tags**:
   * You can filter events based on tags.
   * A black screen with white text

     Description automatically generated
2. **Paths**:
   * You can specify which files or directories must be changed to trigger the workflow.
   * A black screen with white text

     Description automatically generated

**Example Workflow with Activity Types and Filters**

Here’s an example of a GitHub Actions workflow that combines activity types and filters:

A screenshot of a computer program

Description automatically generated

**Explanation of the Example**

* **on:**: Specifies the events that will trigger the workflow.
* **push event**:
  + Triggers only on pushes to the main branch.
  + Only triggers if files in the src directory are modified.
* **pull\_request event**:
  + Triggers on specific actions (opened, edited, labeled) for pull requests targeting the main branch.
* **jobs:**: Defines the jobs to run when the specified events occur.

**Conclusion**

Using **Activity Types** and **Event Filters** in GitHub Actions allows you to create precise workflows that respond only to relevant changes and actions in your repository. This targeted approach enhances the efficiency of your CI/CD processes. If you have further questions or need specific examples, feel free to ask!

**Forks and Pull request event**

When a user forks a repository, they create a copy of the original repository under their own GitHub account. They can then make changes in their fork and submit a pull request to propose those changes to the original repository.

Considerations for Workflows in Forked Repositories

1. Access to Secrets: Workflows triggered by pull requests from forks do not have access to secrets stored in the original repository. This is a security measure to prevent exposure of sensitive data.
2. Workflow Runs: If a workflow runs on a pull request coming from a fork, it will only execute the steps that do not require access to secrets.
3. Branch Protection Rules: You can set up branch protection rules to require that all pull requests pass specific checks (like tests) before merging, which can include workflows that run on pull requests.

A screenshot of a computer program

Description automatically generated

**Key Points**

* **Forking** allows users to contribute to projects without direct access to the original repository.
* **Pull request workflows** help maintain code quality by automating checks on contributions.
* Be mindful of security restrictions regarding secrets when working with forks.

Using the pull request event in GitHub Actions is essential for managing contributions, especially from forks. By configuring your workflows appropriately, you can automate testing and ensure code quality before merging changes.

**Cancelling workflows and skipping workflows:**

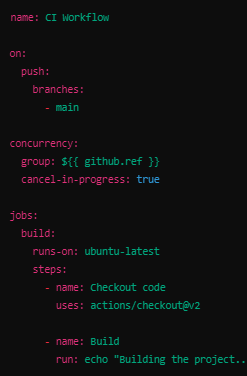
In GitHub Actions, you may sometimes want to cancel workflows that are currently running or skip workflows based on certain conditions. Here’s how to manage both scenarios effectively.

**Canceling Workflows**

You can cancel in-progress workflows when new commits are pushed to the same branch or when certain conditions are met. This is particularly useful in scenarios where a commit may render previous workflows unnecessary (e.g., rapid development cycles).

Automatic Cancellation

You can enable automatic cancellation for workflows triggered by the push event by using the concurrency key. Here’s how it works:



**Explanation of Automatic Cancellation**

* **concurrency**: This key allows you to group workflows by reference (like a branch). If a new workflow is triggered for the same reference while a previous one is still running, the previous workflow will be canceled automatically.
* **group: ${{ github.ref }}**: Groups workflows by branch or tag.
* **cancel-in-progress: true**: Automatically cancels any in-progress workflows in the same group.

**skipping workflows**

You might want to skip a workflow under certain conditions, such as when specific files are modified or based on environment variables. You can achieve this by using conditional statement

A screenshot of a computer program

Description automatically generated

**Explanation of Skipping Workflows**

* **if:**: The if condition checks the commit message. If it matches "skip ci," the step will be skipped.
* You can similarly use conditions to skip entire jobs based on various criteria.

**Conclusion :** By managing workflow cancellations and skips effectively, you can optimize your CI/CD processes in GitHub Actions. Using concurrency allows for efficient handling of rapid commits, while if conditions enable you to control when specific tasks run based on your needs

**Artifact, Output, Caching**

In GitHub Actions, artifacts, outputs, and caching are powerful features that help you manage and share data between jobs and workflows effectively. Here's a detailed overview of each:

**Artifacts**

Artifacts are files or data produced by a job that you want to store for later use. This can include build outputs, test results, logs, or any files you need to retain after a workflow completes.

**Uploading Artifacts**

You can upload artifacts using the actions/upload-artifact action:

A screen shot of a computer program

Description automatically generated

**Downloading Artifacts**

You can download artifacts in another job or workflow using the actions/download-artifact action:

A screen shot of a computer

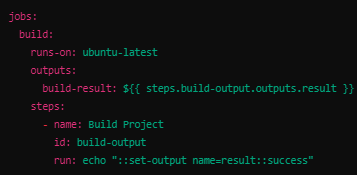
Description automatically generated

**Outputs**

**Outputs** allow you to pass data between jobs within the same workflow. This is useful when you want to use the result of one job in subsequent jobs.

**Setting Outputs**

You can set an output in a job using the echo command:



**Using Outputs in Other Jobs**

You can reference outputs from one job in another:

A computer screen with text

Description automatically generated

**Caching**

**Caching** allows you to store dependencies or files between workflow runs to speed up your CI/CD processes. This is particularly useful for dependencies in languages like Node.js, Python, or Ruby.

**Using the Cache Action**

You can use the actions/cache action to cache dependencies:

A screenshot of a computer program

Description automatically generated

**Explanation of Caching Example**

* **path**: Specifies the path to cache (e.g., node\_modules).
* **key**: A unique identifier for the cache, which can change if your dependencies change (using the hash of the package-lock.json).
* **restore-keys**: A fallback key to restore a cache if an exact match is not found.

**Conclusion**

* **Artifacts** help you store and retrieve files produced during a workflow.
* **Outputs** enable you to share data between jobs within a workflow.
* **Caching** speeds up your workflows by storing and reusing dependencies or files.

These features can greatly enhance the efficiency and usability of your GitHub Actions workflows. If you have further questions or need specific examples, feel free to ask!

**Environment variable and secret in git action:**

**Environment Variables**

**What They Are:** These are key-value pairs that can be set for your workflow. They can store configuration settings or any data your scripts might need.

**How to Use Them:** You can define them directly in your workflow file. For example:

* **A screenshot of a computer program

  Description automatically generated**

**Secrets**

**What They Are:** Secrets are sensitive information, like passwords or API keys. GitHub encrypts these, so they're safe from unauthorized access.

**How to Use Them:** You store them in your repository settings, and then you can reference them in your workflow. For example:

* Go to your repository on GitHub.
* Click on "Settings" > "Secrets and variables" > "Actions".
* Add a new secret (e.g., MY\_SECRET).

Then, in your workflow, you can use it like this:

**A screen shot of a computer

Description automatically generated**

Summary

* Use environment variables for general data needed during your workflows.
* Use secrets for sensitive data that needs to be kept private.

**Utilizing Repository Environments**

Using **Repository Environments** in GitHub Actions helps you manage deployment and workflow settings more effectively, especially when dealing with different stages (like development, staging, and production). Here’s a simple breakdown of how to utilize them:

**What Are Repository Environments?**

* **Environments** allow you to define specific settings, such as environment variables and secrets, for different stages of your workflow.
* They also provide protection rules, like requiring approvals before deploying to certain environments.

**How to Set Up Repository Environments**

1. **Create an Environment:**
   * Go to your GitHub repository.
   * Click on "Settings" > "Environments".
   * Click on "New environment" and give it a name (e.g., production, staging).
2. **Add Secrets:**
   * Within the environment settings, you can add secrets specific to that environment (like database credentials, API keys, etc.).
3. **Set Protection Rules (optional):**
   * You can set rules to require approvals or to restrict who can deploy to the environment.

**Using Environments in Your Workflow**

Here’s a simple example of how to use an environment in a GitHub Actions workflow:



**Key Points**

* **Access Environment Variables and Secrets:** You can access environment variables and secrets defined in the environment using ${{ secrets.SECRET\_NAME }}.
* **Conditional Steps:** You can create workflows that only run certain steps based on the environment.
* **Audit and Control:** Using environments allows for better audit trails and controls over deployments.

Utilizing repository environments in GitHub Actions helps you manage different settings and secrets for various stages of your application, ensuring a smoother and safer deployment process.

**Controlling workflow and job execution:**

**If condition**

**Controlling execution via if**

**Working with special condion function**

**Condiotion jobs**

**Matrix Strategy**

**Include key and exclude key**

**Saving time and code with reusable workflows**

**Adding inputs to reusable workflows**

**Reusable workflows and secret**

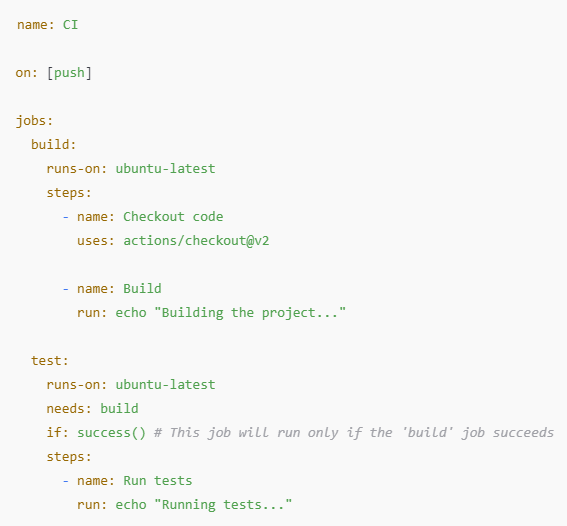
**Reusable workflows outputs**

**Conditional job and step**

In GitHub Actions, you can use conditionals to control whether a job or step runs based on the results of previous jobs or steps, or based on the event that triggered the workflow. Here’s a quick guide on how to set up conditional jobs and steps.

Conditional Jobs

You can use the if keyword at the job level to specify conditions for running a job. Here’s an example:

****

Conditional Steps

You can also use the if keyword at the step level. Here’s how to do it:

**A screenshot of a computer program

Description automatically generated**

Common Conditionals

* success(): True if all previous jobs and steps were successful.
* failure(): True if any previous jobs or steps failed.
* always(): Always true, regardless of previous job results.
* github.ref: Use this to check the branch or tag that triggered the workflow (e.g., github.ref == 'refs/heads/main').

Example of Conditional Logic

* Here’s a more complex example that uses conditions based on different events:

**A screenshot of a computer program

Description automatically generated**

In this example, the deploy job will only run on push events to the main branch.

**Matrix job**

In GitHub Actions, a matrix job allows you to run the same set of steps multiple times with different configurations, such as different versions of a language, operating systems, or any other parameters. This is particularly useful for testing your code across different environments.

Basic Matrix Job Example

Here’s a simple example of a matrix job that runs tests across multiple versions of Node.js:

****

Explanation

* Matrix Configuration: The strategy section defines the matrix, which contains the different configurations (in this case, Node.js versions).
* Using Matrix Variables: In the steps, you can reference the current matrix variable using ${{ matrix.node-version }}.
* Runs on: Each combination of the matrix will run on a new instance of the specified runner (e.g., ubuntu-latest).

Advanced Matrix Example

* You can also create more complex matrices with multiple dimensions. For example, if you want to test both different Node.js versions and operating systems:

**A screenshot of a computer program

Description automatically generated**

Matrix Limitations

* Combinations: Each combination of matrix values will create a separate job, which can lead to many concurrent jobs if you have multiple dimensions and values.
* Job Dependencies: You can use the needs keyword to set dependencies between jobs, but keep in mind that jobs in a matrix run independently.

Conditional Execution with Matrix

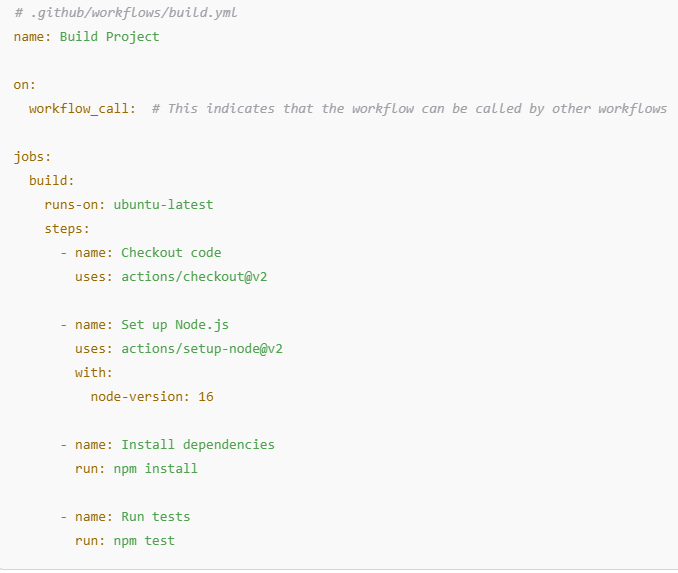
* You can also use conditions with matrix jobs:
* 
* In this example, the tests will not run for Node.js version 18.
* Using matrix jobs can significantly enhance the coverage of your tests and ensure your application works across various environments!

**Reusable wokflows**

Reusable workflows in GitHub Actions allow you to define a workflow once and then call it from other workflows, promoting reusability and reducing duplication. This can be particularly helpful for tasks like CI/CD pipelines, testing, or deployment that are common across multiple repositories or projects.

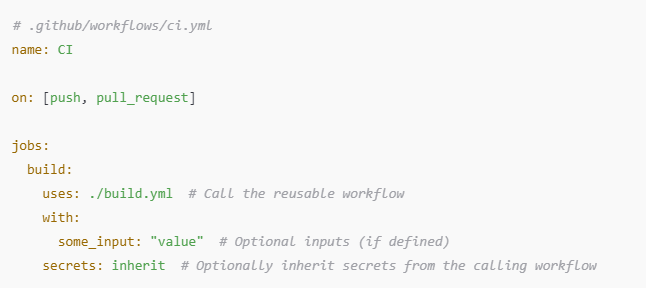
**Defining a Reusable Workflow**

To define a reusable workflow, you create a YAML file in the .github/workflows directory of your repository. Here’s an example of a reusable workflow for building a project:



**Calling a Reusable Workflow**

You can call the reusable workflow from another workflow using the uses keyword. Here’s how to do it:



**Passing Inputs**

You can define inputs in your reusable workflow and pass values when calling it:

**A screenshot of a computer program

Description automatically generated**

**Calling with Inputs**

You can then specify the input when calling the reusable workflow:

**A screenshot of a computer program

Description automatically generated**

**Inheriting Secrets**

If your reusable workflow requires access to secrets, you can choose to inherit secrets from the calling workflow using:

****

**Advantages of Reusable Workflows**

1. **DRY Principle**: Reduce duplication of code and logic across multiple workflows.
2. **Consistency**: Ensure consistency in the way tasks are executed across different workflows.
3. **Modularity**: Promote modular design in your CI/CD processes, making it easier to maintain and update workflows.

**Example Scenario**

Imagine you have multiple microservices that require a similar build and test process. Instead of duplicating the build logic in each service's workflow, you can create a reusable workflow and call it in each service’s CI pipeline.

**Conclusion**

Reusable workflows are a powerful feature in GitHub Actions that can significantly streamline your CI/CD processes. By defining workflows once and reusing them, you save time, reduce errors, and promote better practices in your development lifecycle.

**Containers for job**

In GitHub Actions, you can use **containers** to run your jobs in isolated environments. This is particularly useful when you need specific software versions or dependencies that are not available in the default runner environment.

**How to Use Containers in GitHub Actions**

1. **Specify a Container Image:** You can specify a Docker container image for your job using the container keyword. This image can be pulled from Docker Hub or any other container registry.
2. **Define the Job:** Here’s a simple example of how to set up a job that runs in a container:



**Key Points**

* **Isolation:** Each job runs in its own container, ensuring that dependencies and environment settings do not conflict with others.
* **Custom Images:** You can use public images from Docker Hub or create your own custom images for more specific needs.
* **Access to Host System:** By default, you have limited access to the host system. You can install packages within the container but not on the host directly.

**Benefits of Using Containers**

* **Consistency:** Ensures that your jobs run in a consistent environment, reducing "it works on my machine" issues.
* **Dependency Management:** You can easily manage dependencies without affecting the base runner environment.
* **Flexibility:** Choose any environment that suits your project needs (Python, Ruby, Go, etc.).

**Summary**

Using containers in GitHub Actions allows you to define specific environments for your jobs, ensuring consistency and isolation. This is especially useful for projects that require specific software versions or dependencies.

**Service Container**

In GitHub Actions, **service containers** allow you to run additional containers alongside your main job container. This is especially useful for services like databases or caching servers that your application might need during testing or deployment.

**How to Use Service Containers**

1. **Define Service Containers:** You can specify service containers in your workflow file. Here’s an example where a job uses a PostgreSQL database as a service container:

A screenshot of a computer program

Description automatically generated

**Key Points**

* **Service Definition:** The services key is used to define one or more service containers. Each service is specified with an image and can have environment variables, ports, and health checks.
* **Networking:** The service containers are accessible by the main job container using localhost and the specified port. For example, you can connect to the PostgreSQL service at localhost:5432.
* **Health Checks:** You can define health checks for your services to ensure they are ready before your job runs. This helps prevent issues where your application tries to connect to a service that isn’t fully started yet.

**Benefits of Using Service Containers**

* **Integrated Testing:** Easily run tests that require external services without needing to set them up manually.
* **Isolation:** Each service runs in its own container, keeping your testing environment clean and isolated.
* **Flexibility:** You can use different services as needed (e.g., Redis, MySQL, MongoDB) simply by adding them to your workflow.

**Summary**

Service containers in GitHub Actions allow you to spin up additional containers that your main job can interact with, such as databases or other services. This enhances your testing capabilities and provides a more comprehensive CI/CD setup.

4o mini

**Adding services**

Adding services in GitHub Actions allows you to run additional containers alongside your main job, which is useful for testing or deploying applications that depend on external services like databases, caching systems, or message brokers.

**How to Add Services in GitHub Actions**

Here’s a step-by-step guide with an example:

1. **Define the Service:** Use the services keyword in your job to define the containers you need.
2. **Configure the Service:** Specify the Docker image, environment variables, ports, and any other configuration needed.

**Example Workflow with a Database Service**

Here’s an example that sets up a PostgreSQL database as a service while running a Node.js application:



**Key Components**

* **services:** This key is where you define the service containers your job needs.
* **image:** Specifies the Docker image for the service (e.g., postgres:13).
* **ports:** Maps the service’s ports so they can be accessed by the main job.
* **env:** Sets environment variables needed for the service (like database credentials).
* **options:** Allows you to specify additional Docker options, such as health checks to ensure the service is ready before your tests run.
* **DATABASE\_URL:** An environment variable for your main job, typically used to connect to the service.

**Benefits of Adding Services**

* **Integrated Testing:** You can test your application against real services, ensuring compatibility and functionality.
* **Isolation:** Each service runs in its own container, preventing conflicts between different services.
* **Flexibility:** Easily switch out services or modify configurations as needed.

**Summary**

Adding services in GitHub Actions is straightforward and allows you to create a more robust testing environment by running necessary services like databases alongside your application. This enhances the reliability and effectiveness of your CI/CD workflows.

**Communication between jobs and service container**

In GitHub Actions, jobs can communicate with service containers defined in the workflow. This is especially useful for testing applications that rely on external services like databases or message queues. Here’s how communication works and how to set it up effectively.

**How Communication Works**

1. **Service Containers**: Each service container runs in an isolated environment but is accessible to the jobs in the same workflow. You can communicate with a service container using localhost and the port you've defined.
2. **Environment Variables**: You often need to set up environment variables in your job to provide the necessary connection details (like username, password, and database name) to your application.

**Example Setup**

Here’s an example demonstrating how to set up communication between a job and a PostgreSQL service container:

A screenshot of a computer program

Description automatically generated

**Key Points**

* **Accessing the Service**: You connect to the service using localhost and the port you specified (e.g., localhost:5432 for PostgreSQL).
* **Environment Variables**: Set environment variables in the job to provide your application with the necessary connection details.
* **Health Checks**: It's important to ensure that your service is ready before running any tests. Use the options field to define health checks, which ensure the service is available before the job proceeds.

**Communication Patterns**

1. **Direct Database Connections**: Your application code can connect directly to the service container using the provided connection string.
2. **APIs**: If you're testing an API that relies on a service, the service can be accessed in the same way, allowing your tests to validate functionality against a real instance of that service.

**Summary**

Communication between jobs and service containers in GitHub Actions is straightforward. By using localhost with the appropriate port and setting up necessary environment variables, your jobs can effectively interact with service containers, enabling comprehensive testing of your applications in a CI/CD pipeline.

**Build Custom Action**

Building a custom action in GitHub Actions allows you to automate workflows in your software development projects. Here’s a step-by-step guide to creating a custom GitHub Action:

**Step 1: Set Up Your Repository**

1. **Create a new repository** or navigate to an existing one on GitHub.
2. **Create a directory for your action**. You can name it something like .github/actions/my-action.

**Step 2: Create Action Metadata File**

1. Inside your action directory, create a file named action.yml.
2. Define the action's metadata in this file. Here’s an example:

A screenshot of a computer code

Description automatically generated

**Step 3: Write the Action Logic**

1. **Create your main script file** in the same directory (e.g., index.js).
2. Write the logic for your action. Here’s a simple example using Node.js:
3. A screen shot of a computer program

   Description automatically generated

**Step 4: Add Dependencies**

If your action requires any dependencies, create a package.json file and install them. For example:

A screenshot of a computer code

Description automatically generated

Then run:

npm install

**Step 5: Create a Workflow File**

1. In your repository, create a directory called .github/workflows.
2. Inside this directory, create a file (e.g., main.yml) to define the workflow that uses your action:
3. A screenshot of a computer program

   Description automatically generated

**Step 6: Test Your Action**

1. Commit your changes and push them to GitHub.
2. Trigger the workflow (e.g., by pushing to the main branch) and check the Actions tab in your repository to see your action in action!

**Step 7: Publish (Optional)**

If you want to share your action, consider publishing it to the GitHub Marketplace by creating a README.md file, following GitHub's guidelines, and tagging your repository.

**Conclusion**

Creating a custom GitHub Action involves defining its metadata, writing the action logic, and integrating it into your workflows. This allows for greater automation and customization tailored to your project's needs!

**Why use custom action**

Using custom actions in GitHub Actions provides several key advantages that can enhance your development workflows:

1. Tailored Automation

* Custom actions allow you to automate tasks that are specific to your project’s needs, making it easier to implement workflows that suit your unique processes.

2. Reusability

* Once developed, custom actions can be reused across different workflows and repositories, promoting consistency and reducing the need to duplicate code.

3. Simplified Workflows

* By encapsulating complex logic in a single action, you can simplify your workflow files, making them easier to read, maintain, and understand.

4. Improved Efficiency

* Automating repetitive tasks, such as testing, building, or deploying, saves time and reduces the risk of human error, leading to more reliable processes.

5. Integration Capabilities

* Custom actions can interface with external services and APIs, allowing for richer functionality and integration into your broader development ecosystem.

6. Flexibility and Adaptability

* You can easily modify or extend custom actions to meet changing project requirements, making it simple to adapt your workflows as needed.

7. Modularity

* Custom actions support a modular approach, enabling different team members to work on distinct components of a workflow independently.

8. Community Sharing

* If you publish your custom actions, you contribute to the open-source community, allowing others to benefit from your work and potentially collaborate on improvements.

Conclusion

Custom actions in GitHub Actions empower you to create efficient, tailored, and maintainable automation workflows, enhancing your development process and enabling better collaboration within teams.

**Different type of custom action**

Here are several types of custom actions you can create in GitHub Actions, along with specific use cases for each:

1. JavaScript Actions

Use Case: Automated Testing

* Description: Write a custom action in JavaScript to run unit tests.
* Example: This action could check out the code, install dependencies, and execute tests using a testing framework like Jest.
* A screenshot of a computer program

  Description automatically generated

2. Docker Actions

Use Case: Build and Push Docker Images

* Description: Create a Docker action that builds a Docker image from your code and pushes it to a container registry.
* Example: Automate the deployment of a microservice whenever there’s a new commit.
* A screenshot of a computer program

  Description automatically generated

3. Composite Actions

Use Case: Multi-Step Workflow for Deployment

* Description: Combine multiple steps into a single reusable action, like checking out code, setting up environment variables, and deploying to a cloud provider.
* Example: Streamline the deployment process for an application.
* A screenshot of a computer

  Description automatically generated

4. Scripted Actions

Use Case: Custom Notification

* Description: Write a shell script that sends notifications (e.g., via Slack or email) after a successful build.
* Example: Notify your team whenever the CI pipeline completes successfully.
* A computer screen shot of a computer

  Description automatically generated

5. Event-Driven Actions

Use Case: Triggering Actions on Pull Requests

* Description: Create an action that automatically labels pull requests based on the presence of certain keywords in the title.
* Example: Automatically label PRs as “urgent” if they contain the word "urgent" in the title.
* A screenshot of a computer code

  Description automatically generated

Conclusion

These examples illustrate different types of custom actions in GitHub Actions, each tailored to specific use cases. By leveraging these actions, you can create efficient, reusable, and maintainable workflows that streamline your development process.

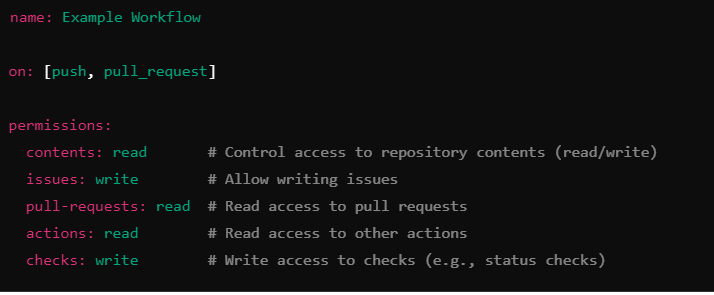
**Git Hub Token and Permission**

GitHub Actions Permissions

You can control permissions for workflows at both the organization/repository level and within individual workflows.

**Workflow Permissions**

You can specify permissions directly in your workflow YAML file using the permissions key. Here's a breakdown:



Default Permissions

By default, the permissions for the GITHUB\_TOKEN in a workflow are:

* contents: read
* issues: read
* pull-requests: read
* actions: read

You can adjust these permissions to restrict access to only what's necessary.

GITHUB\_TOKEN

* Description: The GITHUB\_TOKEN is automatically generated by GitHub for each workflow run. It allows you to interact with the GitHub API without needing to create your own token.
* Usage: Access it in your workflows like this:

A screen shot of a computer

Description automatically generated

Personal Access Tokens (PATs)

For more control over permissions, you can create a Personal Access Token (PAT):

1. Create a PAT:
   * Go to GitHub settings → Developer settings → Personal access tokens → Tokens (classic).
   * Generate a new token with the required scopes (permissions).
2. Scopes: When creating a PAT, choose scopes according to your needs:
   * repo: Full control of private repositories.
   * workflow: Update GitHub Actions workflow files.
   * admin:org: Manage the organization (admin access).
   * write:packages: Publish packages.
3. Use in Workflows:
   * Store the PAT as a secret in your repository (e.g., MY\_PAT).
   * Access it in your workflow:
   * A screen shot of a computer

     Description automatically generated

Best Practices

* Limit Permissions: Grant only the permissions necessary for your workflows.
* Use Environment Secrets: For sensitive information related to environments, use GitHub Environments to manage secrets and require approvals.
* Regularly Review Tokens: Periodically review and rotate your tokens for security.

By understanding and effectively managing permissions and tokens in GitHub Actions, you can enhance the security and efficiency of your workflows. If you have further questions or need more specific examples, let me know!