# Learning Github Actions

Yatharth Chauhan

# WELCOME TO MY WORLD

yatharthchauhan.me

CONNECT WITH ME

# WELCOME TO GITHUB ACTIONS

```
AUTHOR:         Yatharth Chauhan (Github: YatharthChauhan2362)
TOPIC:          Github Actions
REPOSITORY:     Github-Actions-Learning
```

# Github Actions

GitHub Actions are automated processes that you can set up in your repository to build, test, package, release, or deploy your code. You can use them to automate your workflow and add custom behaviors to your repository.

## Advantages of GIthub Actions

1. **Automation:**

- GitHub Actions allow you to automate your workflow, so you don't have to manually build, test, package, release, or deploy your code. This can save you time and effort, and help you ensure that your code is always up-to-date and working as expected.

2. **Customization:**

- GitHub Actions are highly customizable, so you can tailor your workflow to fit your specific needs. You can define custom triggers, use different actions and services, and configure your workflow to run on different platforms and environments.

3. **Integration:**

- GitHub Actions integrate seamlessly with your repository, so you can easily set up and manage your workflow from within GitHub. You can also use them in combination with other tools and services, such as deployment platforms, package repositories, and continuous integration servers.

4. **Collaboration:**

- GitHub Actions are collaborative, so you can share your workflow with your team and contribute to the development of open-source projects. You can also use them to automate tasks that involve multiple repositories or organizations.

## Key functionality of GitHub Actions

GitHub Actions is a continuous integration and delivery (CI/CD) platform that allows you to automate your software development workflows.

Some key functionality of GitHub Actions includes:

1. Running tests and code checks automatically when you push code to your repository.
2. Building and deploying your code to various environments automatically.
3. Automating the release process for your code.
4. Running custom scripts or actions in response to specific events such as creating or merging a pull request.

Overall, GitHub Actions allows you to automate many of the tasks involved in the software development process, helping you save time and reduce the chance of errors.

## Events

1. **Push events:**

- Triggered when you push code to a repository.

2. **Pull request events:**

- Triggered when you create or update a pull request.

3. **Issue events:**

- Triggered when you open, close, or update an issue.

4. **Release events:**

- Triggered when you create or publish a release.

5. **Comment events:**

- Triggered when you comment on a commit, issue, or pull request.

6. **Schedule events:**

- Triggered on a schedule that you specify (e.g., daily, weekly).

Example:

```
on:
    push:
        branches:
            - master
pull_request:
    branches:
        - master
```

In this example, the workflow is named "CI" and it is triggered by two types of events: push events and pull request events. The workflow will run only when code is pushed to the master branch or when a pull request is created against the master branch.

# Github Workflows

GitHub Workflows are a way to automate tasks that you use in your software development workflows. Workflows are made up of individual Jobs, which are defined by a series of steps. Each step in a job can run a script, an action, or an external command.

- Workflows can be triggered by a variety of events, such as pushes to a repository, the creation of a pull request, or the opening of an issue.
- A workflow in GitHub Actions is defined in a YAML file that is stored in the .github/workflows directory in your repository. The file must have a .yml or .yaml extension.

1. **Build and test:**

- This Workflow could be triggered whenever you push new code to your repository. It could run your test suite to ensure that the code is working as expected, and then build a production-ready version of your code.

2. **Release:**

- This Workflow could be triggered whenever you create a new release tag in your repository. It could build and package your code, and then upload the package to a package repository or a deployment service.

3. **Deploy:**

- This Workflow could be triggered whenever you push new code to a specific branch (e.g. production). It could build and package your code, and then deploy it to your production environment.

Example:

Here is a simple example of a workflow file that runs a test job whenever code is pushed to the repository:

```
name: CI

    on: push

    jobs:
        test:
        runs-on: ubuntu-latest
        steps:
            - uses: actions/checkout@v2
            - run: npm install
            - run: npm test
```

This workflow is named "CI" and it is triggered by push events. It consists of a single job called "test" that runs on an Ubuntu environment.

The job has three steps:

1. checking out the code
2. Installing dependencies
3. Running tests.

Whenever code is pushed to the repository, the "CI" workflow will run and execute the steps in the "test" job. This can help ensure that the code is always tested and working properly.

## Monolithic actions

In a monolithic action, all the steps for an action are included in a single file.

This can make it easier to manage and maintain the action, as all the code is in one place. However, monolithic actions can also be more difficult to test and debug, as it can be harder to isolate issues to a specific step.

Here is an example of a monolithic action that builds and pushes a Docker image to a registry:

```
name: Build and push Docker image

    on: [push]

    jobs:
        build:
            runs-on: ubuntu-latest
            steps:
                - uses: actions/checkout@v2
                - name: Build and push image
                run: |
                    docker build -t my-image .
                    docker login -u username -p password
                    docker push my-image
```

This action is triggered whenever a push event occurs on the repository. It consists of a single job with two steps:

1. The **actions/checkout** action retrieves the code from the repository.
2. The **Build and push image** step builds a Docker image from the code and pushes it to a registry.

This action is monolithic because all the steps are included in a single file.

### Static website to GitHub Pages

Here is an example of a GitHub Action that deploys a static website to GitHub Pages:

```
name: Deploy to GitHub Pages

on:
    push:
        branches:
            - master

jobs:
    build:
        runs-on: ubuntu-latest
        steps:
            - uses: actions/checkout@v2
            - name: Build and Deploy
            uses: peaceiris/actions-gh- pages@v3
            with:
                github_token: ${{     secrets.GITHUB_TOKEN }}
                publish_dir: ./public
```

This action is triggered whenever you push new code to the *master* branch of your repository.

It runs on the latest version of Ubuntu, checks out the code, and then uses the *peaceiris/actions-gh-pages* action to build and deploy the code to GitHub Pages.

The *publish_dir* parameter specifies the directory that contains the static files to be deployed, and the *github_token* parameter allows the action to authenticate with GitHub and push the changes to the *gh-pages* branch.

## Github Marketplace

GitHub Marketplace is a platform that allows developers to find and purchase tools and services that integrate with GitHub. Some examples of tools and services that are available on GitHub Marketplace include project management tools, code review tools, continuous integration and deployment services, and performance monitoring tools. To use GitHub Marketplace, you need to have a GitHub account.

You can browse the Marketplace by visiting the GitHub Marketplace website or by accessing the Marketplace from the GitHub website. Github Marketplace

## Starter Workflow

A starter workflow is a pre-configured GitHub Actions workflow that you can use as a starting point for your own workflow. GitHub provides a number of starter workflows that you can use to automate common tasks, such as building and deploying code or running tests.

To use a starter workflow, you can copy the workflow file to your repository and modify it to fit your needs.

For example, here is a starter workflow for building and deploying a static site:

```
name: Deploy

    on:
        push:
            branches: [ master ]

jobs:
    build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Install dependencies
    run: npm install
    - name: Build
    run: npm run build
    - name: Deploy
  uses: peaceiris/actions-gh-pages@v3
     with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: ./public
```

This workflow is triggered whenever a push event occurs on the master branch. It consists of a single job with four steps:

1. The **actions/checkout** action retrieves the code from the repository.
2. The **Install dependencies** step installs the required dependencies for the project.
3. The **Build** step builds the static site.
4. The **Deploy** step uses the **peaceiris/actions-gh-pages** action to deploy the built site to GitHub Pages.

## Action Creation

To create a GitHub Action, you need to create a workflow file in your repository. A workflow is a defined series of steps that can be automated on GitHub.

Here's an example of a simple workflow that runs a Python script whenever a push event occurs on the repository:

```
name: Run Python script

on: [push]

jobs:
    build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Run script
     run: python script.py
```

To create a GitHub Action, you need to create a workflow file in your repository. A workflow is a defined series of steps that can be automated on GitHub.

Here's an example of a simple workflow that runs a Python script whenever a push event occurs on the repository:

Copy code name: Run Python script on: [push] jobs: build: runs-on: ubuntu-latest steps: - uses: actions/checkout@v2 - name: Run script run: python script.py This workflow consists of a single job with two steps:

1. The **actions/checkout** action retrieves the code from the repository.
2. The **Run script** step runs a Python script called **script.py**.

To create this workflow, you would create a file called **main.yml** in the **.github/workflows** directory of your repository, and copy the above code into the file.

There are many other options and configurations that you can use to customize your workflow. You can find more information about creating GitHub Actions in the GitHub Actions documentation.

## Github Action Governance

The ability to run code comes the need for governance to ensure that Actions are used safely and responsibly. Here are some ways you can practice good governance when using GitHub Actions:

1. Use the least privilege principle: when creating a workflow, make sure to only grant the permissions that are absolutely necessary for the Action to run. This helps to reduce the risk of unintended consequences.

2. **Use secret scanning:**

- GitHub provides a secret scanning feature that can help you detect and prevent the accidental exposure of secrets such as API keys in your code.

3. **Use gated workflows:**

- you can set up a gated workflow that requires a code review and/or approval before it can be run. This can help to ensure that the code being run is reviewed and approved by multiple people before it is executed.

4. **Use branch protection rules:**

- you can set up branch protection rules to require that all code changes are reviewed before they are merged. This can help to prevent malicious code from being merged into your repository.

5. **Monitor your workflows:**

- it is important to regularly monitor your workflows to ensure that they are running as expected and to detect any issues or security concerns.

## Troubleshooting Tools and Techniques

1. **Workflow logs:**

- The most basic tool for troubleshooting issues with your workflows is the workflow logs. You can access the logs for a workflow by clicking on the "Actions" tab in your repository, selecting the workflow you want to view, and then clicking on the specific run you want to view the logs for. The logs will show you the output from each action in the workflow, as well as any errors that occurred.

2. **Debugging actions:**

- You can use the debug action in your workflow to print debug messages to the workflow logs. This can be helpful for understanding what is happening in your workflow and identifying the source of any issues.

3. **Using environment variables:**

- You can use environment variables to pass information between actions in your workflow. For example, you might set an environment variable in one action and then use it in a later action to customize its behavior.

4. **Using workflow commands:**

- GitHub Actions provides a set of workflow commands that you can use to control the execution of your workflows. For example, you can use the if command to conditionally execute actions based on the value of an environment variable.

5. **Using third-party tools:**

- There are a number of third-party tools that can be helpful for troubleshooting issues with your workflows. For example, you might use a linting tool to check your workflow file for syntax errors, or a testing tool to validate the behavior of your actions.

### Debug Workflows

There are a few different ways you can debug your GitHub Actions workflows:

1. **Print debug messages:**

- One of the most basic ways to debug your workflows is to use the echo command or the debug action to print debug messages to the workflow logs. This can be helpful for understanding what is happening in your workflow and identifying the source of any issues.

2. **Set breakpoints:**

- You can use the workflow-debug command in your workflow to set a breakpoint, which will pause the execution of your workflow at that point. This can be useful for inspecting the state of your workflow and debugging issues.

3. **Use environment variables:**

- You can use environment variables to pass information between actions in your workflow. For example, you might set an environment variable in one action and then use it in a later action to customize its behavior.

4. **Use third-party tools:**

- There are a number of third-party tools that can be helpful for debugging issues with your workflows. For example, you might use a linting tool to check your workflow file for syntax errors, or a testing tool to validate the behavior of your actions.

# CI-CD Workflows

CI/CD (continuous integration/continuous delivery). With GitHub Actions, you can define a workflow that automatically builds, tests, and deploys your code to a specific environment every time you push a commit or create a pull request.

Here's an example of a simple CI/CD workflow that you could use with GitHub Actions:

1. When you push a commit to the repository, the workflow is triggered and runs a series of tasks, such as building the code and running tests.
2. If all of the tasks are successful, the workflow proceeds to the next step, which is to deploy the code to a staging environment.
3. Once the code is deployed to the staging environment, you can manually test it to ensure that everything is working as expected.
4. If the code is working as expected, you can then use the workflow to deploy it to the production environment.

## CI Workflow

GitHub Actions can be used to set up a continuous integration (CI) workflow, which is a series of automated steps that are triggered when code is pushed to a repository.

A CI workflow can help you automatically build, test, and deploy your code, saving you time and reducing the risk of errors.

To set up a CI workflow using GitHub Actions, you will need to create a configuration file called a "workflow" in your repository. This file is written in YAML and defines the steps that make up your CI workflow.

Here is an example of a more complex CI workflow that builds and deploys a static website to GitHub Pages:

```
name: CI

on: [push]

jobs:
    build:
        runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2
    - name: Install dependencies
      run: |
        npm install
    - name: Build site
    run: |
        npm run build
    - name: Deploy to GitHub Pages
    uses: JamesIves/github-pages-deploy-action@3.6.1
    with:
        ACCESS_TOKEN: ${{ secrets.ACCESS_TOKEN }}

        BRANCH: gh-pages
        FOLDER: build
```

This workflow is triggered whenever code is pushed to the repository (on: [push]). It consists of a single job called "build" that runs on the latest version of Ubuntu. The job has four steps:

1. The first step checks out the code from the repository.
2. The second step installs the dependencies for the project using npm.
3. The third step builds the static website using the command npm run build.
4. The fourth step deploys the built website to GitHub Pages using the github-pages-deploy-action Action.

To use this workflow, you will need to create a personal access token and store it as a secret in your repository. You can then reference the secret in your workflow using ${{ secrets.ACCESS_TOKEN }}. This allows you to authenticate with the GitHub API and deploy your website to GitHub Pages.

## CD Workflow

Here's an example of a simple CD workflow using GitHub Actions:

1. Create a new repository in GitHub and push your code to it.
2. In the root of your repository, create a new directory called **.github/workflows**.
3. In the **.github/workflows** directory, create a new file called **deploy.yml**. This file will contain the instructions for your CD workflow.
4. In the **deploy.yml** file, define your workflow by specifying a name and the trigger that will start the

   workflow: name: Deploy on: push: branches: - master

This workflow will be triggered every time you push to the master branch of your repository.

5. Add a build step to your workflow by using the **actions/setup-node** action and the **actions/npm** action:

   jobs: build: runs-on: ubuntu-latest steps: - uses: actions/setup-node@v1 with: node-version: 12 - run: npm install - run: npm run build

This will install the necessary dependencies and build your code.

6. Add a step to test your code by using the **actions/setup-node** action and the **actions/npm** action:
   - run: npm test

This will run any tests you have defined in your project.

7. Add a step to deploy your code to a hosting platform, such as GitHub Pages or Heroku:
   - name: Deploy to GitHub Pages uses: peaceiris/actions-gh-pages@v3 with: github_token: ${{ secrets.GITHUB_TOKEN }} publish_dir: ./build

This will publish the contents of the build directory to GitHub Pages.

```
name: Deploy

on:
    push:
        branches:
            - master

jobs:
    build:
        runs-on: ubuntu-latest
        steps:
            - uses: actions/setup-node@v1
             with:
                node-version: 12
            - run: npm install
            - run: npm run build
            - run: npm test
            - name: Deploy to GitHub Pages
            uses: peaceiris/actions-gh-pages@v3
            with:
                github_token: ${{ secrets.GITHUB_TOKEN }}
                publish_dir: ./build
```

## Super Linter

Super Linter is an open-source project that provides a unified interface for running a variety of code linters. It can be used to lint code written in languages such as:

- Bash
- CoffeeScript
- Go
- JSON
- Python
- Ruby
- Terraform and many more

To use Super Linter in GitHub Actions, you will need to create a workflow file in your repository.

This file defines a set of tasks that will be run whenever a certain trigger occurs (e.g. when you push code to your repository). Here is an example workflow file that uses Super Linter to lint the code in your repository:

```
name: Lint Code

on: [push]

jobs:
    lint:
        runs-on: ubuntu-latest
        steps:
            - uses: actions/checkout@v2
            - name: Install Dependencies
              run: |
                apt-get update
                apt-get install -y curl
            - name: Install Super Linter
            run: |
                curl -L <https://git.io/super-linter> | bash
            - name: Lint Code
            run: |
                super-linter
```

This workflow will run whenever you push code to your repository. It will install the necessary dependencies (e.g. curl), install Super Linter, and then run the super-linter command to lint your code.

## Environments

In GitHub Actions, environments are a way to define a set of variables that can be used in your workflow.

They are useful for storing information such as API keys, connection strings, or other secrets that your workflow needs to use. Environments can be defined in your repository and then referenced in your workflow file.

To create an environment in your repository, go to the "Actions" tab, then click on the "Environments" link in the left sidebar. From here, you can create a new environment by clicking the "New environment" button. You can then give your environment a name and define the variables that should be included in the environment.

Once you have defined an environment, you can use it in your workflow by using the env keyword and the name of the environment.

For example:

```
jobs:
    build:
        runs-on: ubuntu-latest
        env:
            MY_SECRET: ${{ env.MY_ENVIRONMENT_NAME }}
        steps:
            - run: echo "The value of MY_SECRET is $MY_SECRET"
```

Environments are a great way to separate sensitive information from your code and keep it secure. They are also useful for managing different configurations for different environments, such as staging and production environments.

## Protection rules

Protection rules in GitHub are a way to ensure that certain actions cannot be performed on a repository unless certain conditions are met.

For example, you might want to require that all pull requests be reviewed by at least one other person before they can be merged, or that all pushes to the **master** branch must be made by a designated group of maintainers.

To create protection rules for a repository, go to the "Settings" tab of the repository, then click on the "Branches" link in the left sidebar. From here, you can choose a branch for which you want to set protection rules, and then click the "Add rule" button. You can then specify the conditions that must be met before certain actions can be performed on the branch.

Some common protection rules include:

- Requiring a certain number of approving reviews on pull requests
- Requiring that pull requests be up to date with the target branch before they can be merged
- Requiring that only certain users or teams can push to the branch
- Enabling status checks and requiring that all checks pass before a pull request can be merged

## Deployment Logs

In GitHub Actions, deployment logs are a way to view the details of a deployment that was performed as part of a workflow. You can access the deployment logs for a workflow by going to the "Actions" tab of the repository, then clicking on the workflow that you want to view.

Once you are **viewing the details** of a workflow, you can click on the "Deployments" tab to view the deployment logs. This will show you a list of all deployments that were performed as part of the workflow, along with information about the deployment environment and the status of the deployment.

If you click on a **specific deployment**, you can view more detailed information about the deployment, including the log output from the deployment steps. This can be helpful for troubleshooting any issues that may have occurred during the deployment process.

You can also use the **GitHub API** to retrieve deployment logs programmatically. This can be useful if you want to automate the process of accessing and analyzing deployment logs as part of your continuous integration and delivery process.

# Runners

In GitHub Actions, a runner is a specific instance that runs a workflow. When you set up a workflow, you specify the environment in which the workflow will run, such as a specific operating system and version of a programming language. When the workflow is triggered, GitHub creates a runner for the specified environment, and the runner executes the steps in the workflow.

## Github hosted runners vs self hosted runners

***Hosted runners:**

- These runners are provided and maintained by GitHub. You don't need to worry about setting up or maintaining the infrastructure for these runners. Hosted runners are available for all public repositories and for private repositories owned by organizations that have a paid GitHub plan.

**Self-hosted runners:**

- These runners are installed and maintained by you. You can use self-hosted runners to run your workflows on your own infrastructure, on premesis, or in a cloud provider that GitHub Actions doesn't support. Self-hosted runners can be used with any repository, public or private.

## Configured self hosted runner

To configure a self-hosted runner in GitHub Actions, you need to do the following:

1. Install the runner software on the machine where you want to run the workflow. The runner software is available for Windows, Linux, and macOS.

2. On the machine where you installed the runner software, run the **config.cmd** or **config.sh** script to register the runner with GitHub. This script is located in the **bin** directory of the runner software installation directory.

3. When you run the script, you will be prompted to enter the URL of the repository that you want to use with the runner, as well as an authentication token. You can get the repository URL and authentication token from the repository's settings page in GitHub.

4. After you have entered the repository URL and authentication token, the script will configure the runner and start it. The runner will automatically connect to GitHub and start running workflows as soon as they are triggered.

5. If you want to use the runner with multiple repositories, you can register the runner multiple times, each time with a different repository URL and authentication token.

Note that you can also use the GitHub Actions API to register and manage self-hosted runners programmatically. This can be useful if you want to automate the process of setting up and configuring runners.

## Runner groups

In GitHub Actions, a runner group is a collection of runners that are configured to run the same type of workload. You can use runner groups to manage the resources that are available for running your workflows.

For example, suppose you have a self-hosted runner that is configured to run Linux workloads, and another self-hosted runner that is configured to run Windows workloads. You can create a runner group for each runner, and then specify the runner group that you want to use for each workflow in your repository. This way, you can ensure that the workflow is run on the appropriate runner, depending on the environment that is required.

You can create runner groups and manage the runners within them using the GitHub Actions API or the GitHub Actions web UI. You can also use the actions/setup-runner action in a workflow to automatically set up and configure a runner in a runner group.

Using runner groups can help you better manage the resources that are available for running your workflows, and can make it easier to ensure that your workflows are run in the appropriate environment.

## Runner node configuration

In GitHub Actions, a runner is a specific instance that runs a workflow. When you set up a workflow, you specify the environment in which the workflow will run, such as a specific operating system and version of a programming language. This environment is called the "runner node".

To configure the runner node for a workflow, you need to specify the runner node in the workflow file. The runner node is defined using the runs-on key. Here is an example of a workflow file that specifies a runner node:

```
name: CI

on: [push]

jobs:
    build:
        runs-on: ubuntu-latest
        steps:
            - uses: actions/checkout@v2
            - name: Run a one-line script
            run: echo Hello, world!
```

In this example, the runner node is an Ubuntu machine with the latest version of Ubuntu installed. You can specify any of the supported operating systems as the runner node for your workflow.

You can also specify a specific version of an operating system as the runner node. For example, you can use ubuntu-20.04 to specify that the runner node should be an Ubuntu machine with Ubuntu 20.04 installed.

### Security risk of public runner

There is a security risk associated with using a public runner in that you are sharing the environment with other users and have no control over who is using the runner at the same time as you. This means that there is a possibility that someone else could potentially access your code or data if they are able to compromise the runner.

To mitigate this risk, it is important to ensure that you are following best practices for securing your code and data, such as using secure credentials and environment variables, and not committing sensitive information to your repository. You should also consider using a self-hosted runner or a private runner, which provides more control over the environment and can reduce the risk of external access to your code and data.

### Dynamically scale runners

There are two main ways to dynamically scale runners in GitHub Actions:

1. **Self-hosted runners:**

- Self hosted runners can set up and manage your own virtual machines or physical servers to run your jobs. This allows you to customize the hardware and software environment to meet the needs of your workload, and you can scale the number of runners up or down as needed.

**Private runners:**

- Private runners are managed by GitHub and run on dedicated virtual machines. They offer the convenience of a public runner, but with the added security and isolation of a private environment. You can dynamically scale private runners by changing the number of concurrent jobs that the runner can handle.

To dynamically scale runners in GitHub Actions, you will need to set up the runner infrastructure and configure your workflow to use the runners. You can then use the GitHub API or the Actions workflow syntax to scale the runners up or down as needed.

## Github secrets

In GitHub Actions, secrets are encrypted environment variables that you can use in your workflows to store sensitive information, such as passwords and API keys. Secrets are stored in the GitHub repository where your workflow is defined and are encrypted at rest and in transit.

To use secrets in your GitHub Actions workflow, you will first need to define the secrets in the repository settings. To do this, go to the "Settings" tab of your repository, then click on "Secrets" in the left menu. From here, you can add a new secret by giving it a name and entering its value.

Once you have defined your secrets, you can use them in your workflow by referencing the secret name in your workflow file. For example:

```
jobs:
    build:
        steps:
        - uses: actions/checkout@v2
        - name: Set up Node.js
          uses: actions/setup-node@v2
          with:
            node-version: 12.x
        - name: Install dependencies
          run: npm install
        - name: Build the project
          run: npm run build
          env:
            MY_SECRET_ENV_VAR: ${{ secrets.MY_SECRET }}
```

In this example, the secret named "MY_SECRET" is used to set an environment variable named "MY_SECRET_ENV_VAR" in the build step of the workflow. The value of the secret is encrypted and stored securely, and is only decrypted and made available to the workflow when the job is run.

It is important to keep in mind that secrets are stored in the repository, so anyone with access to the repository can view the names of the secrets, but not their values. You should not commit sensitive information to your repository or use secrets to store information that you do not want other repository collaborators to be able to see.

## Organization secrets and Repository secrets

There are two types of secrets in GitHub Actions: repository secrets and organization secrets.

1. **Repository secrets**

- Repository secrets are specific to a single repository and can only be used in workflows that are defined in that repository. Repository secrets are stored in the repository settings and are only accessible to people with access to the repository.

2. **Organization secrets**

- Organization secrets are shared across all repositories in an organization and can be used in any workflow in any repository in the organization. Organization secrets are stored in the organization settings and are only accessible to people with permission to manage the organization.

Both repository secrets and organization secrets are encrypted at rest and in transit, and can only be accessed by the workflow when the job is run. However, organization secrets have the added advantage of being able to be shared across multiple repositories, which can make it easier to manage secrets for large organizations with many repositories.

## Environment secrets

GitHub Actions allow you to set environment variables for your workflow. These environment variables are encrypted and only made available to the actions in your workflow.

To set an environment variable, you can use the env key in your workflow file. For example:

```
jobs:
    build:
        runs-on: ubuntu-latest
        steps:
        - name: Set an environment variable
            MY_VAR: "hello"
        - name: Print the environment variable

        run: |
            echo $MY_VAR
```

You can also set environment variables using the **secrets** key. Secrets are encrypted environment variables that are stored in the GitHub Secrets Manager. To set a secret, you can use the**GITHUB_TOKEN**secret to authenticate with the GitHub API and then call the**/repos/:owner/:repo/actions/secrets/:secret_name** endpoint to set the value of a secret.

For example, you can use the following script to set a secret named **MY_SECRET**:

```
# Set the value of the secret
curl -X PUT -H "Authorization: token ${GITHUB_TOKEN}" -d '{"value":"my secret
value"}' <https://api.github.com/repos/OWNER/REPO/actions/secrets/MY_SECRET>
```

You can then use the secret in your workflow by referencing it as an environment variable. For example:

```
jobs:
    build:
        runs-on: ubuntu-latest
        steps:
        - name: Print the secret
        run: |
            echo $MY_SECRET
```

It is important to note that secrets are only made available to the actions in your workflow and are not passed to subsequent runs of the workflow. They are also not made available to forks of your repository.

## Secrets Limitations

There are a few limitations to be aware of when using secrets in GitHub Actions:

1. **Maximum number of secrets per repository:**

- You can store up to 100 secrets per repository. This includes secrets that are used in GitHub Actions and also those that are used in GitHub Packages.

2. **Size limit of secrets:**

- Secrets are limited to 64 KB in size. If you need to store larger amounts of data, you should consider using an external secret management service or storing the data in a private Git repository.

3. **Visibility of secrets:**

- Secrets are only visible to the repository's maintainers. You cannot share secrets with other users or organizations.

4. **Secrets in forks:**

- Secrets are not available in forks of a repository. If you want to use secrets in a fork, you need to create the secrets in the forked repository.

5. **Editing or Removing Secrets:**

- Once added, a secret cannot be edited directly, you would need to remove the secret and re-create it with the new value.

6. **Limitations in usage:**

- Secrets can only be used in GitHub Actions workflow files, they cannot be accessed via the REST API, Github UI, or other tools. Additionally, secrets are not available when using the GitHub Actions REST API or the GitHub REST API.

It's important to keep in mind that you should only store sensitive information, such as API keys and passwords, in secrets. You should not store information that is sensitive and must be kept confidential, like a private key. This kind of data should be stored in a secure offline location or a password manager or other specialized software.

Thank You.