Contents

[GITHUB ACTION 3](#_Toc189420520)

[COMMON USE CASES 3](#_Toc189420521)

[KEY BUILDING BLOCKS 3](#_Toc189420522)

[WORKFLOW 3](#_Toc189420523)

[JOBS 3](#_Toc189420524)

[RUNNERS 4](#_Toc189420525)

[STEPS 4](#_Toc189420526)

[CREATING A WORKFLOW 5](#_Toc189420527)

[SIMPLE WORKFLOW 5](#_Toc189420528)

[RUNNING THE WORKFLOW 6](#_Toc189420529)

[EVENTS THAT TRIGGERED WORKFLOWS 7](#_Toc189420530)

[CREATING WORKFLOW FROM CODEBASE 8](#_Toc189420531)

[ACTIONS 8](#_Toc189420532)

[EXAMPLE 8](#_Toc189420533)

[WITH KEYWORD 10](#_Toc189420534)

[MULTIPLE TRIGGERS 10](#_Toc189420535)

[RUNNING JOBS IN PARALLEL VS SEQUENTIAL 11](#_Toc189420536)

[RUNNING JOBS IN PARALLEL 11](#_Toc189420537)

[RUNNING JOBS SEQUENTIALLY 11](#_Toc189420538)

[EXPRESSIONS AND CONTEXT OBJECTS 12](#_Toc189420539)

[COMMONLY USED PROPERTIES OF THE CONTEXT OBJECT 12](#_Toc189420540)

[ACCESSING CONTEXT OBJECT USING EXPRESSION 13](#_Toc189420541)

[CONTEXT OBJECTS 13](#_Toc189420542)

[EXERCISE 13](#_Toc189420543)

[WORKFLOW TRIGGERS / EVENTS 16](#_Toc189420544)

[EVENT FILTERS AND ACTIVITY TYPES 17](#_Toc189420545)

[EVENT FILTERS 18](#_Toc189420546)

[ACTIVITY TYPES 19](#_Toc189420547)

[CANCELLING AND SKIPPING WORKFLOWS 22](#_Toc189420548)

[SKIPING WORKFLOW RUN 23](#_Toc189420549)

[JOB ARTIFACTS(DATA) AND OUTPUTS 23](#_Toc189420550)

[JOB ARTIFACTS 23](#_Toc189420551)

[USE CASE/ EXAMPLE 24](#_Toc189420552)

[JOB OUTPUTS 25](#_Toc189420553)

[EXAMPLE - 1 25](#_Toc189420554)

[EXPLANATION 26](#_Toc189420555)

[NOTE 26](#_Toc189420556)

[EXAMPLE -2 26](#_Toc189420557)

[JOB ARTIFACTS VERSUS JOB OUTPUTS 28](#_Toc189420558)

[DEPENDENCY CACHING 28](#_Toc189420559)

[EXAMPLE 31](#_Toc189420560)

[ENVIRONMENT VARIABLES AND SECRETS 32](#_Toc189420561)

[DEFINING ENVIRONMENT VARIABLES 32](#_Toc189420562)

[WORKFLOW-WIDE VARIABLES 33](#_Toc189420563)

[JOB-SPECIFIC VARIABLES 33](#_Toc189420564)

[STEP-SPECIFIC VARIABLES 33](#_Toc189420565)

[EXAMPLE 33](#_Toc189420566)

[USING SECRETS 40](#_Toc189420567)

[CREATING & STORING SECRETS 40](#_Toc189420568)

[ACCESSING SECRETS 42](#_Toc189420569)

[REPOSITORY ENVIRONMENT 42](#_Toc189420570)

[CREATING ENVIRONMENT SECRETS 43](#_Toc189420571)

[CONTROLLING WORKFLOW AND JOB EXECUTION 45](#_Toc189420572)

[CONDITIONAL JOB AND STEPS 46](#_Toc189420573)

[SPECIAL CONDITIONAL FUNCTIONS 48](#_Toc189420574)

[CONDITION ON STEP LEVEL 48](#_Toc189420575)

[CONDITION ON JOB LEVEL 50](#_Toc189420576)

[MATRIX STRATEGIES 50](#_Toc189420577)

[REUSABLE WORKFLOW 52](#_Toc189420578)

[EXAMPLE 53](#_Toc189420579)

[CUSTOM ACTIONS 54](#_Toc189420580)

[TYPES OF CUSTON ACTIONS 54](#_Toc189420581)

# GITHUB ACTION

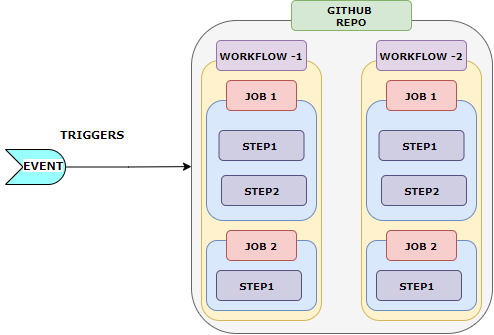
**GitHub Actions allow us to automate various tasks and workflows directly within their repositories. It provides a way to build, test, and deploy code in a consistent and efficient manner**.

* With GitHub Actions, we can define custom workflows using YAML syntax, which are triggered by specific events such as pushes, pull requests, or scheduled intervals.
* These workflows can include a series of steps, which can be executed on different operating systems, environments, or even in parallel.

## COMMON USE CASES

1. **Continuous Integration (CI):** Automatically building and testing code every time changes are made to a repository, helping to catch bugs and issues early on.
2. **Continuous Deployment (CD):** Deploying applications or services to various environments (such as staging or production) when specific conditions are met, such as passing CI tests or merging pull requests.
3. **Code quality and security checks**: Running linters, code formatters, vulnerability scanners, or other static analysis tools to ensure code quality and security standards are met.
4. **Release automation**: Automating the process of creating release notes, tagging versions, and publishing artifacts or packages.

## KEY BUILDING BLOCKS



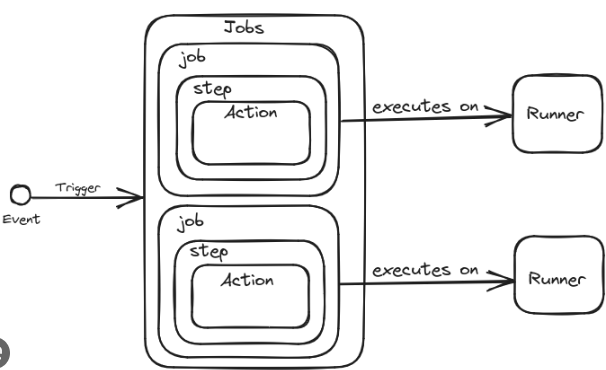
WORKFLOW

* A workflow is a configurable automated process that we can define in the repository using a YAML file**.** We can have multiple workflows defined in a repository.
* Workflows are attached to GitHub repositories, which contains one or more jobs.
* The workflow can be triggered by specific events, such as **push, pull request, or a scheduled interval**. These events are called **triggers**.

JOBS

* Within a workflow, we can define one or more jobs. Jobs contain the steps that will be executed.
* Based on configuration **- The Jobs can run in parallel(default), sequential or conditional**.
* Each job can have its own environment, such as the operating system, required tools, and dependencies called Runners. Runners are the execution environment(the machine and operating system) that will be used for executing these steps.

### RUNNERS



* A runner is a machine or virtual environment that executes a workflow. Runners are responsible for running the jobs defined in the workflow and executing the steps within those jobs.
* GitHub offers a selection of different runner types and operating systems, such as Ubuntu, Windows, and macOS.
* **Runners**: [actions/runner-images: GitHub Actions runner images](https://github.com/actions/runner-images)
* The **runs-on** keyword is used in a workflow file to specify the type of runner and operating system on which the job should run. It determines the environment in which the steps of a job will be executed.
* A job represents a set of steps that run on the same runner.

|  |  |
| --- | --- |
| **jobs:**  **my\_job:**  **runs-on: ubuntu-latest**  **steps:**  **- name: Step 1**  **run: echo "Hello, world!"**  **- name: Step 2**  **run: echo "This is another step"** | * In this example, the my\_job job will run on an Ubuntu runner. The runs-on keyword specifies the **ubuntu-latest** as runner, which is a GitHub-hosted runner with the latest version of Ubuntu. * By specifying the runs-on value, we can control the execution environment for our jobs and ensure that they run on the desired operating system and runner type. |

STEPS

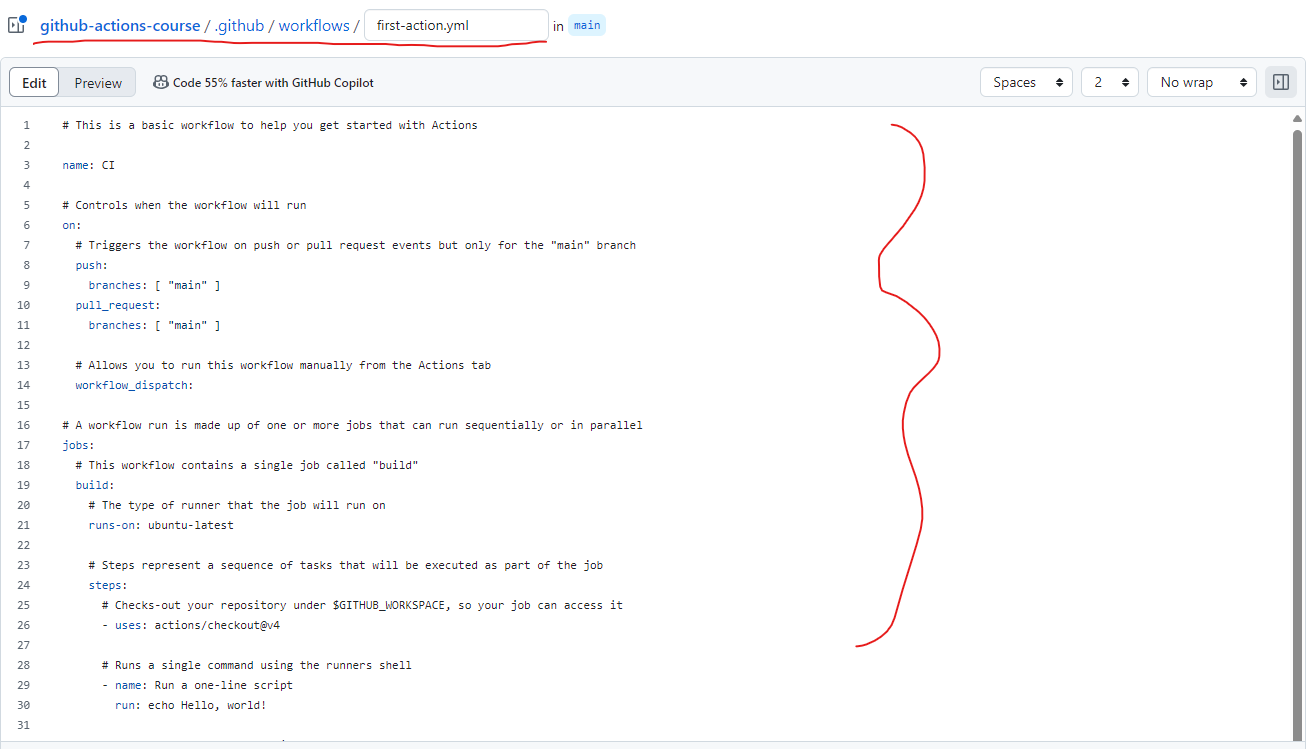
* Steps define the individual tasks that make up a job.
* **Steps can be a shell scripts or an Action** Each step runs in its own shell or container environment and can execute commands or actions.
* Steps can be written in various languages, such as JavaScript, Python, Ruby, or shell scripts.
* We can use pre-defined actions from the GitHub Marketplace or create our own custom actions to reuse across workflows.

## CREATING A WORKFLOW

|  |  |
| --- | --- |
|  | 1. Navigate to Repo 🡪 Actions 🡪 Select a Workflow template 2. This will create an editor with some initial workflow structure & content in “YAML” file |

1. The path where the work will be created is **<repo>/.github/workflows/<work-file-name>.yml**

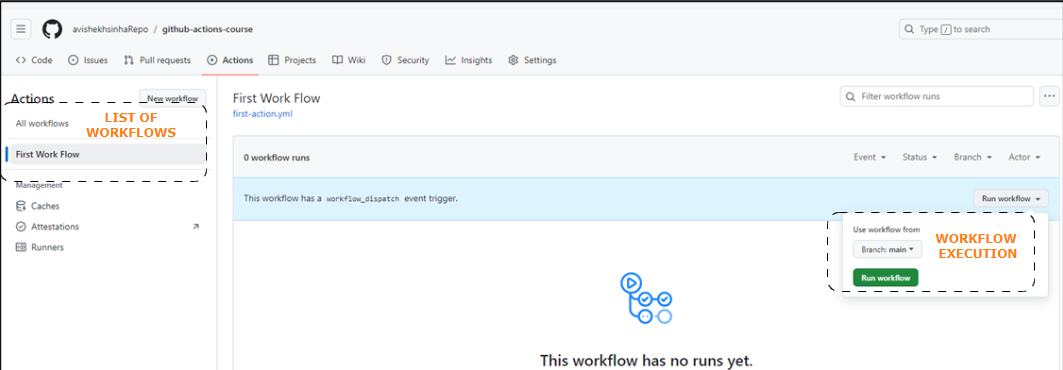
***Note****: We can create multiple workflows by creating multiple ‘yaml’ files in ‘workflow’ directory*

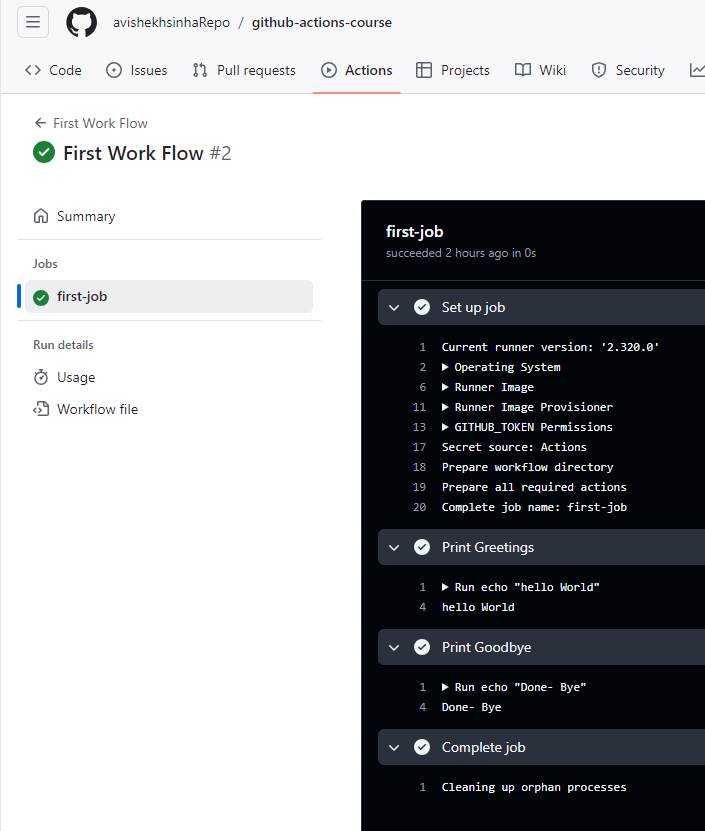


### SIMPLE WORKFLOW

|  |  |
| --- | --- |
| **name**: First Work Flow  **on**: workflow\_dispatch  **jobs**:  first-job:  **runs-on**: ubuntu-latest  **steps**:  - **name**: Print Greetings  **run**: echo "hello World"  - **name**: Print Goodbye  **run**: echo "Done- Bye"  *So, when we manually trigger the "First Work Flow" workflow, it will run the "first-job" job on the latest version of Ubuntu. The job consists of two steps: printing the greeting message "hello World" and printing the farewell message "Done- Bye" to the console.* | * **name**: First Work Flow: This sets the name of the workflow to "First Work Flow". * **on**:The "on" keyword in a GitHub Actions workflow configuration file specifies the event or set of events that will trigger the workflow. It determines when the workflow will be executed. * GitHub Actions supports various other events that can be used with the "on" keyword, such as:   + push: Triggered when code is pushed to the repository.   + pull\_request: Triggered when a pull request is created or updated.   + schedule: Triggered at specific scheduled intervals.   + repository\_dispatch: Triggered by a custom event using the GitHub API.   In the example   * **workflow\_dispatch**: This specifies that the workflow should be triggered manually using the "workflow\_dispatch" event. This means we can manually run this workflow from the GitHub Actions UI. * **jobs**: This section defines the jobs that will be executed as part of the workflow. In this case, there is one job named "first-job".   + **first-job**: This section defines the properties and steps for the "first-job" job.   + **runs-on**: ubuntu-latest:     - This specifies that the job should run on the latest version of the Ubuntu operating system.     - GitHub Actions provides a variety of runner environments to choose from, and in this case, it is using the Ubuntu environment. * **steps**: This section defines the individual steps that will be executed as part of the job.   + **name**: Print Greetings: This sets the name of the step to "Print Greetings".   + **run**: echo "hello World": This is the command that will be executed as part of the step. In this case, it is using the echo command to print the message "hello World" to the console.   + **name**: Print Goodbye: This sets the name of the step to "Print Goodbye".   + **run**: echo "Done- Bye": This is the command that will be executed as part of the step. It uses the echo command to print the message "Done- Bye" to the console. |

### RUNNING THE WORKFLOW





|  |
| --- |
| NOTE  If we need to run multiple shell commands (or multi-line commands, e.g., for readability), we can easily do so by adding the pipe symbol (|) as a value after the run: key.  Like this:   1. ... 2. run: | 3. echo "First output" 4. echo "Second output"   **This will run both commands in one step.** |

## EVENTS THAT TRIGGERED WORKFLOWS

* Reference : <https://docs.github.com/en/actions/writing-workflows/choosing-when-your-workflow-runs/events-that-trigger-workflows>

## CREATING WORKFLOW FROM CODEBASE

|  |  |
| --- | --- |
|  | To organize the work flow with the Repo   * We can create a **.github** folder at the root level of the repository. * Inside the .github folder, we can create a workflows folder to store all workflow files. * Each workflow file should have a “yml” extension and contain the definition of a specific workflow. * We can create multiple workflow files within the workflows folder to define different workflows for different purposes; |

## ACTIONS

* In GitHub Actions, actions are the individual units of work/application that performs a (typically complex) frequently repeated task ,that can be combined to create workflows.
* Actions are reusable, standalone tasks that can be executed within a workflow.
* **They are defined in YAML files and can be created by the GitHub community or by the repository owner.**
* Actions can perform a wide range of tasks, **such as building and testing code, deploying applications, sending notifications, or running custom scripts**. They are designed to automate various parts of the software development lifecycle.
* Actions are typically written in JavaScript, but they can also be created in other languages like Python or Shell. **They can be triggered by events, such as a push to a repository, a pull request, or a scheduled interval.**
* When defining a workflow in GitHub Actions, we can use pre-built actions from the GitHub Marketplace (<https://github.com/marketplace?type=actions> ) or create our own custom actions. These actions can be combined and configured to create a workflow that fits your specific needs.
* Actions in GitHub Actions provide a way to modularize and reuse common tasks, making workflows more efficient and easier to maintain. They contribute to the automation and streamlining of various development processes, enabling teams to achieve continuous integration and delivery.

### EXAMPLE

In this example – we want to run the test cases using Github Action for a React Application

#### STEPS IN GITHUB ACTION

The workflow run in runner machine/ environment hence we need following steps in the workflow

|  |  |
| --- | --- |
|  | 1. **Download /checkout the code in Runner environment** 2. **Install the required node version** 3. **Install the dependencies** 4. **Run the test cases** |

|  |  |
| --- | --- |
| **name: The Test Case Workflow**  **on: [push]**  **jobs:**  **test:**  **runs-on: ubuntu-latest**  **steps:**  **- name: Get Code**  **uses: actions/checkout@v3**  **- name: Install Node JS**  **uses: actions/setup-node@v2**  **with:**  **node-version: '18'**  **- name: Install Dependencies**  **run: npm ci**  **- name: Run Tests**  **run: npm test** | * The workflow is triggered by a push event to the repository, meaning it will run whenever changes are pushed to the repository. * The workflow consists of a single job named test, which runs on the latest version of Ubuntu (ubuntu-latest). The job is broken down into several steps: * **Get Code**:   + This step uses the **actions/checkout@v3** action to check out the repository's code.   + This action ensures that the workflow has access to the codebase that triggered the workflow. * **Install Node JS**:   + This step uses the actions/setup-node@v2 action to set up Node.js version 18.   + This action configures (using **with**) the environment with the specified version of Node.js, which is necessary for running npm commands and the project's JavaScript code. * **Install Dependencies**:   + This step runs the **npm ci** command to install the project's dependencies.   + The npm ci command is optimized for continuous integration environments and ensures that the exact versions of dependencies specified in the [package-lock.json](vscode-file://vscode-app/c:/Program%20Files/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) file are installed. * **Run Tests**   + The final step runs the npm test command to execute the project's test suite. This command typically runs any tests defined in the project, ensuring that the code changes do not introduce any regressions or issues. |

**NOTE**

|  |  |
| --- | --- |
| on:  push:  branches:  **- 'main'**  **- 'releases/\*\*'** | * **This trigger will run the workflow when the code is pushed to main or to a branch that starts with releases/.** |

**npm ci**

* npm ci is a command that is used to install dependencies for a Node.js project based on the package-lock.json.
* It is primarily used in continuous integration (CI) or automated build environments to ensure reproducible and deterministic dependency installations.

**How npm ci works?**

* It deletes the node\_modules directory and any existing package-lock.json or npm-shrinkwrap.json file.
* It installs the exact versions of dependencies specified in the package-lock.json or npm-shrinkwrap.json file, ensuring that the installed dependencies match the exact versions specified in the lock file. This helps to create a consistent and reproducible environment for your project.
* It does not modify the package.json file or update the versions of the dependencies listed in it. Instead, it strictly relies on the lock file to install the dependencies.
* Unlike npm install, which allows for more flexibility in updating dependencies based on the semver ranges specified in the package.json file, npm ci is designed to be used in CI or automated build environments where reproducibility and consistency are essential.
* It is recommended to run npm ci instead of npm install in your CI pipeline or when setting up a new development environment to ensure that the exact versions of dependencies are installed as specified in the lock file.

### WITH KEYWORD

* **The `with` keyword in a GitHub Action workflow is used to provide inputs or parameters to an action or step.**
* **It allows us to pass values or arguments to the action or step, which can be used to customize its behavior.**

EXAMPLE

|  |  |
| --- | --- |
| **name: Example Workflow**  **on:**  **push:**  **branches:**  **- main**  **jobs:**  **example-job:**  **runs-on: ubuntu-latest**  **steps:**  **- name: Step 1**  **uses: actions/checkout@v2**    **- name: Step 2**  **run: echo "This is step 2."**    **- name: Step 3**  **uses: some-action@v1**  **with:**  **input1: value1**  **input2: value2**  **input3: ${{ secrets.SOME\_SECRET }}** | `example-job` job consists of three steps:   1. The first step checks out the repository using the `actions/checkout` action. 2. The second step runs a simple shell command to echo a message. 3. The third step uses a custom action named `some-action@v1`. It utilizes the `with` keyword to provide inputs (`input1`, `input2`, and `input3`) and their corresponding values to the action.      * **The `with` keyword allows us to pass values from various sources, including static values, environment variables, and even secrets stored in GitHub Secrets.** * **For example, to reference environment variables using the `${{ env.VARIABLE\_NAME }}` syntax, and secrets using the `${{ secrets.SECRET\_NAME }}` syntax.** |

## MULTIPLE TRIGGERS

|  |  |
| --- | --- |
| **on: [push,workflow\_dispatch]** | * We are allowing the workflow to be triggered both by a push event on the specified branch and through manual triggering via the workflow dispatch event. |

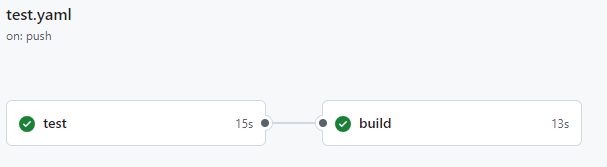
## RUNNING JOBS IN PARALLEL VS SEQUENTIAL

### RUNNING JOBS IN PARALLEL

|  |  |
| --- | --- |
|  | * Parallel execution allows multiple jobs to run simultaneously, potentially reducing the overall execution time of your workflow. * This is useful when the jobs are independent of each other and can be executed concurrently without any dependencies or conflicts. For example - tasks like building and testing multiple configurations or running tests in parallel against different environments. |

### RUNNING JOBS SEQUENTIALLY

* Sequential execution ensures that jobs are executed one after another, in a defined order.
* This is useful when the jobs have dependencies on each other, and the output of one job is required as an input for the next job.
* Sequential jobs can be defined by specifying the **needs** property in the workflow file, which ensures that a job runs only after its dependencies have completed successfully.
* Sequential jobs can be helpful when you need to perform tasks like deploying an application after successful build and test jobs, or running a set of steps in a specific order.



|  |  |
| --- | --- |
|  | `name: The Test Case and Deployment Workflow  on: [push,workflow\_dispatch]  jobs:  test:  runs-on: ubuntu-latest  steps:  - name: Get Code  uses: actions/checkout@v3  - name: Install Node JS  uses: actions/setup-node@v4  with:  node-version: '18'  - name: Install Dependencies  run: npm ci  - name: Run Tests  run: npm test  build:  **needs: test**  runs-on: ubuntu-latest  steps:  - name: Get Code  uses: actions/checkout@v3  - name: Install Node JS  uses: actions/setup-node@v4  with:  node-version: '18'  - name: Install Dependencies  run: npm ci  - name: Build Project  run: npm run build |
| Note:  We can use the needs keyword in GitHub Actions to wait for multiple jobs to finish before starting a subsequent job or step.  **needs: [job1, job2]** |

## EXPRESSIONS AND CONTEXT OBJECTS

* The context object in GitHub Actions provides information about the execution environment and the event that triggered the workflow.
* It contains several properties that you can access within the workflow to retrieve details about the repository, the event, and other relevant information.

### COMMONLY USED PROPERTIES OF THE CONTEXT OBJECT

|  |  |
| --- | --- |
| **github.repository** | Contains information about the repository where the workflow is running, such as the repository owner and name. |
| **github.event** | Represents the payload of the GitHub event that triggered the workflow run. This includes data specific to the event type, such as push, pull request, issue, etc |
| **github.sha** | Specifies the commit SHA associated with the event that triggered the workflow run. |
| **github.ref** | Provides the Git reference (branch or tag) associated with the event that triggered the workflow run. |
| **github.actor** | Represents the username of the user or the name of the GitHub App that initiated the event |

### ACCESSING CONTEXT OBJECT USING EXPRESSION

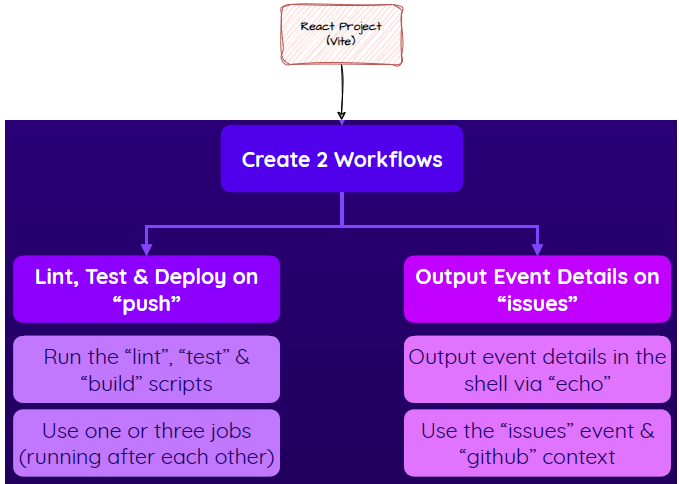
* We can access these properties using the Github object in the workflow steps.
* **For example, github.repository would be accessed as ${{ github.repository }}.**

### CONTEXT OBJECTS

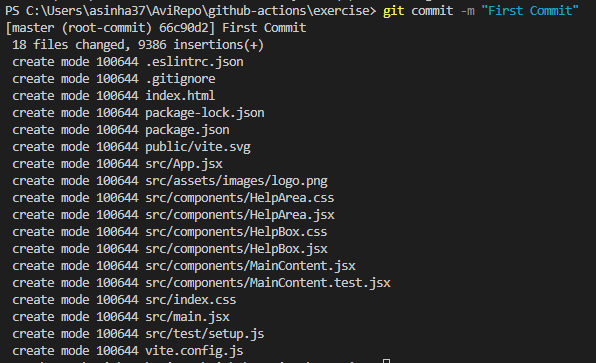
**REFERENCE**

* **Context Object** : <https://docs.github.com/en/enterprise-cloud@latest/actions/writing-workflows/choosing-what-your-workflow-does/accessing-contextual-information-about-workflow-runs#about-contexts>
* **Expression**: <https://docs.github.com/en/enterprise-cloud@latest/actions/writing-workflows/choosing-what-your-workflow-does/evaluate-expressions-in-workflows-and-actions#about-expressions>

## EXERCISE



1. Step 1: **CREATE A NEW REMOTE REPO**. Example: [**github-action-react**](https://github.com/avishekhsinhaRepo/github-action-react) - <https://github.com/avishekhsinhaRepo/github-action-react.git>
2. Step 2: Initialize the local repo : **git init**
3. **Step 3: Commit the code** 
   1. **git add .**
   2. **git commit -m “First Commit”**



1. Step 4: Push to the remote repository
   1. git remote add origin <https://github.com/avishekhsinhaRepo/github-action-react.git>
   2. git branch -M main
   3. git push -u origin main
2. Step 5: Create a workflow using “Simple Workflow” template

A screenshot of a computer

Description automatically generated

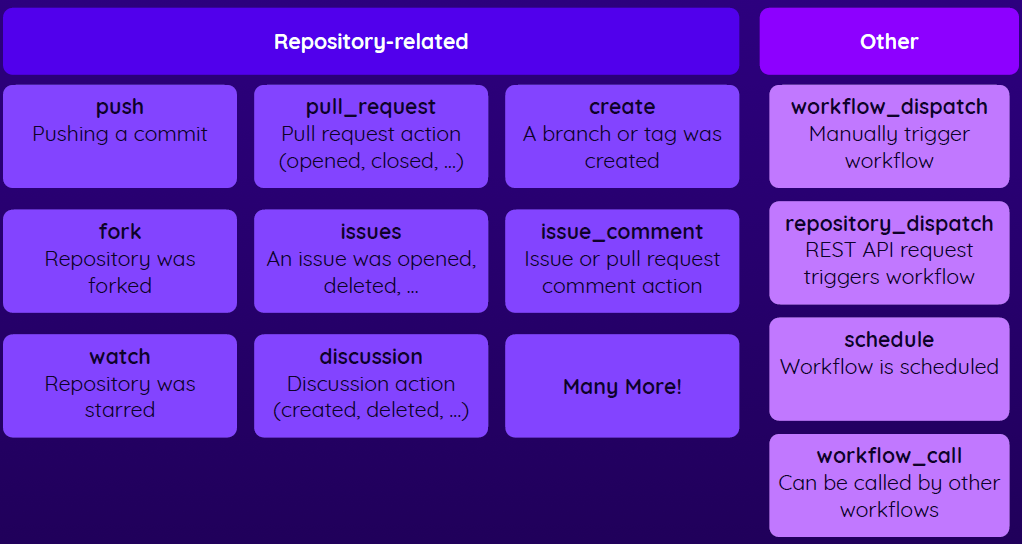
|  |  |
| --- | --- |
| **Release.yaml** | Package.json |
| **name: React App Test Build and deploy Workflow**  **on: [workflow\_dispatch, push]**  **jobs:**  **lint:**  **runs-on: ubuntu-latest**  **steps:**  **- name: Checkout code**  **uses: actions/checkout@v2**  **- name: Install dependencies**  **run: npm ci**  **- name: Run tests**  **run: npm run lint**  **test:**  **runs-on: ubuntu-latest**  **steps:**  **- name: Checkout code**  **uses: actions/checkout@v2**  **- name: Install dependencies**  **run: npm ci**  **- name: Run tests**  **run: npm test**  **build:**  **needs: test**  **runs-on: ubuntu-latest**  **steps:**  **- name: Checkout code**  **uses: actions/checkout@v2**  **- name: Install dependencies**  **run: npm ci**  **- name: Build**  **run: npm run build**  **deploy:**  **needs: build**  **runs-on: ubuntu-latest**  **steps:**  **- name: Deploy**  **run: echo "Deployed"** | **{**  **"name": "02-basic-example",**  **"private": true,**  **"version": "0.0.0",**  **"type": "module",**  **"scripts": {**  **"dev": "vite",**  **"lint": "eslint --ext .jsx --fix src",**  **"build": "vite build",**  **"preview": "vite preview",**  **"test": "vitest run"**  **},**  **"dependencies": {**  **"prop-types": "^15.8.1",**  **"react": "^18.2.0",**  **"react-dom": "^18.2.0"**  **},**  **"devDependencies": {**  **"@testing-library/jest-dom": "^5.16.5",**  **"@testing-library/react": "^13.3.0",**  **"@testing-library/user-event": "^14.4.3",**  **"@types/react": "^18.0.17",**  **"@types/react-dom": "^18.0.6",**  **"@vitejs/plugin-react": "^2.0.1",**  **"eslint": "^8.23.0",**  **"eslint-plugin-react": "^7.31.1",**  **"jsdom": "^20.0.0",**  **"vite": "^3.0.7",**  **"vite-plugin-eslint": "^1.8.1",**  **"vitest": "^0.22.1"**  **}**  **}** |
| **ISSUE WORKFLOW**  **name: React issues Workflow**  **on:**  **issues:**  **types:**  **- opened**  **- closed**  **jobs:**  **react-issues:**  **runs-on: ubuntu-latest**  **steps:**  **- name: Checkout code**  **uses: actions/checkout@v2**  **- name: Install dependencies**  **run: npm ci**  **- name: Run tests**  **run: npm run test** | * It will be triggred whenever a New Issue is opened or existing issue is closed. |

A screenshot of a computer

Description automatically generated

# WORKFLOW TRIGGERS / EVENTS

* Events are triggers that initiate the execution of workflows.
* The events are mostly either **repository related** like push, pull\_request or **other events like workflow\_dispatch**
* Documentation : <https://docs.github.com/en/actions/writing-workflows/choosing-when-your-workflow-runs/events-that-trigger-workflows>



|  |  |
| --- | --- |
| MULTPLE TRIGGERS FOR A WORKFLOW | name: Events Demo 1  on**: [workflow\_dispatch, push]**  jobs:  …. |

## EVENT FILTERS AND ACTIVITY TYPES

**A screenshot of a computer

Description automatically generated**

* **EVENT FILTERS AND ACTIVITY TYPES ARE USED TO DEFINE SPECIFIC CONDITIONS THAT MUST BE MET FOR A WORKFLOW TO RUN**
* **Events can have activity types and filters.**
* Event filters and activity types are defined in the "on" section of the workflow file, where you specify the event that triggers the workflow

### EVENT FILTERS

* **Event filters allow us to specify conditions related to the event that triggers a workflow**.
* **Filter Pattern Cheat Sheet** : <https://docs.github.com/en/actions/writing-workflows/workflow-syntax-for-github-actions#filter-pattern-cheat-sheet>

#### COMMONLY USED EVENT FILTERS

|  |  |
| --- | --- |
| **BRANCHES**   * We can specify one or more branches using glob patterns or regular expressions. * Workflows will only run if the event occurs on the specified branch(es). * **- main**: This line specifies that the workflow should trigger when a push event occurs on the "main" branch. * **- 'feature/\*'**: This line specifies that the **workflow should trigger when a push event occurs on any branch that starts with "feature/".** * **- 'release/\*\*'**: This line specifies that the workflow should trigger when a **push event occurs on any branch that starts with "release/" and has any number of subdirectories.** * **- 'dev-\*'**: This line specifies that the workflow should trigger when a push event occurs on any branch that starts with "dev-". | **on:**  **push:**  **branches:**  **- main**  **- ‘feature/\*’**  **- ‘release/\*\*’**  **- ‘dev-\*’**   * Workflow will run when a push event occurs on the "**main**" branch or **any branch starting with "feature/".** |
| **Note**   * The characters \*, [, and ! are special characters in YAML. If we start a pattern with \*, [, or !, we must enclose the pattern in quotes. * Also, if we use a flow sequence with a pattern containing [ and/or ], the pattern must be enclosed in quotes | **# Valid**  **paths:**  **- '\*\*/README.md'**  **# Invalid - creates a parse error that**  **# prevents your workflow from running.**  **paths:**  **- \*\*/README.md**  **# Valid**  **branches: [ main, 'release/v[0-9].[0-9]' ]**  **# Invalid - creates a parse error**  **branches: [ main, release/v[0-9].[0-9] ]** |
| **TAGS**   * We can specify one or more tags using glob patterns or regular expressions. * Workflows will only run if the event occurs on a tag that matches the specified pattern(s). | **on:**  **push:**  **tags:**  **- v1.\***  **- v2.0**   * The workflow will run when a push event occurs on a tag that matches the pattern "v1.\*" or the exact tag "v2.0". |
| **PATHS**   * We can specify one or more file paths using glob patterns. * Workflows will only run if the event involves changes to files that match the specified path(s). | **on:**  **push:**  **paths:**  **- 'src/\*\*'**   * The workflow will run when a push event involves changes to files in the "src" directory or any of its subdirectories. |

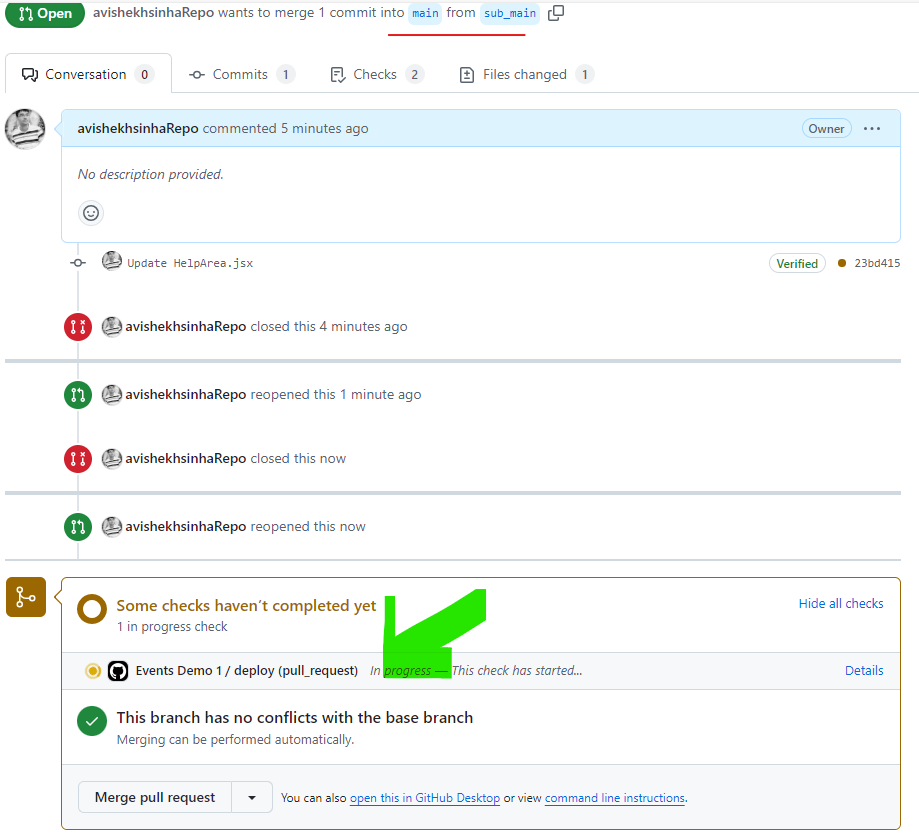
### ACTIVITY TYPES

* **ACTIVITY TYPES ALLOW US TO FURTHER FILTER WORKFLOWS BASED ON THE SPECIFIC ACTIVITY THAT TRIGGERED THE EVENT.**

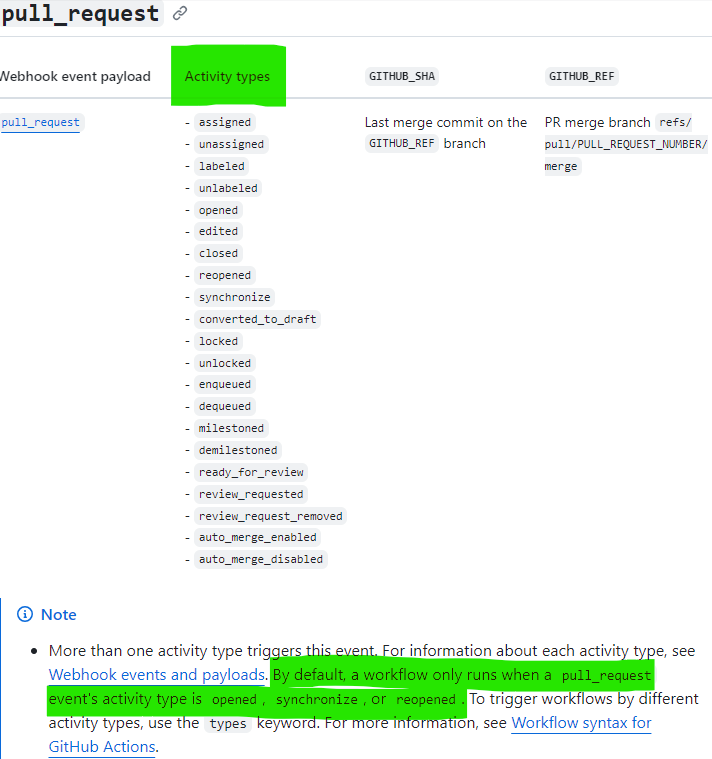
#### COMMONLY USED ACTIVITY TYPES

1. **Created**: This activity type filters workflows based on whether a new branch, tag, or pull request was created.
2. **Pushed**: This activity type filters workflows based on whether a commit was pushed to the repository.
3. **Closed**: This activity type filters workflows based on whether an issue, pull request, or check run was closed.
4. **Opened**: This activity type filters workflows based on whether an issue, pull request, or check run was opened.

|  |  |  |
| --- | --- | --- |
| **VARIATION 1** | **VARIATION 2** | **VARIATION 3** |
| **name: Events Demo 1**  **on:**  **pull\_request:**  **types:**  **- opened**  **- synchronize**  **- reopened**  **…** | **name: Events Demo 1**  **on:**  **pull\_request:**  **types: [opened, synchronize, reopened]**  **…** | **name: Events Demo 1**  **on:**  **pull\_request:**  **types: opened**  **..** |



SAMPLE ACTIVITY TYPE DOCUMENTATION



|  |
| --- |
| * By combining event filters and activity types, we can create more fine-grained rules for when workflows should run. For example, we can configure a workflow to only run when a commit is pushed to the "master" branch and the changes involve files in a specific directory. |

EXAMPLE

|  |
| --- |
| name: Events Demo 1  on:    pull\_request:      types:        - opened        - synchronize        - reopened      branches:        - main        - release/\*\*    push:      branches:        - main        - release/\*\*    workflow\_dispatch: |

EXAMPLE -2



## CANCELLING AND SKIPPING WORKFLOWS

|  |  |
| --- | --- |
|  | * By default, the workflow gets cancelled of Job fails * By default, a Job fails if at least one step fails * MANUALALLY CANCELLING WORKFLOW   Screenshot showing the summary for a workflow that is currently running. The "Cancel workflow" button is highlighted with a dark orange outline. |

### SKIPING WORKFLOW RUN

* Documentation : <https://docs.github.com/en/actions/managing-workflow-runs-and-deployments/managing-workflow-runs/skipping-workflow-runs>
* We can add a special comment in our commit message to skip a workflow run. if we add any of the following strings to the commit message in a push the workflow run will be skipped.

1. [skip ci]
2. [ci skip]
3. [no ci]
4. [skip actions]
5. [actions skip]

|  |  |
| --- | --- |
| EXAMPLE | git commit -m "event added **[skip ci**]" |

# JOB ARTIFACTS(DATA) AND OUTPUTS

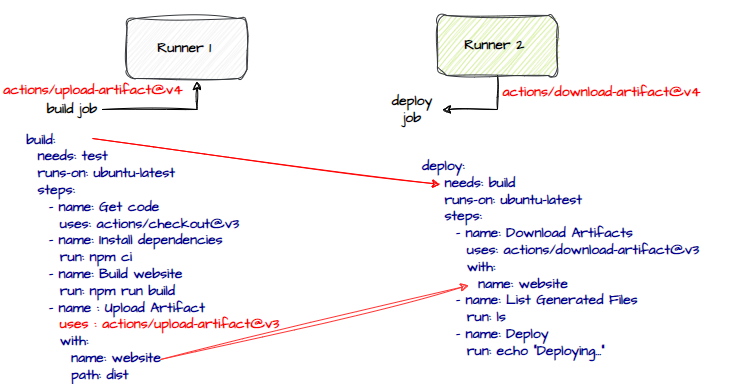
## JOB ARTIFACTS

* **Job artifacts are files or directories generated during the execution of a job**
* We can use the Job artifacts to persist and make available for download or use in subsequent steps or workflows.

A diagram of a job

Description automatically generated

### USE CASE/ EXAMPLE



* When the jobs runs, they run on their own runner environments
* Now – let’s consider an application (e.g a React Application) which has **build** job and a **deploy** job .
* The use case, where job artifact can be leveraged can be **the artifacts which get generated in build steps(compiled JS and HTML) can be used in the deploy step** so that it can be deployed to a web server

|  |  |
| --- | --- |
| **name: Deploy website**  **on:**  **push:**  **branches:**  **- main**  **jobs:**  **test:**  **runs-on: ubuntu-latest**  **steps:**  **- name: Get code**  **uses: actions/checkout@v3**  **- name: Install dependencies**  **run: npm ci**  **- name: Lint code**  **run: npm run lint**  **- name: Test code**  **run: npm run test**  **build:**  **needs: test**  **runs-on: ubuntu-latest**  **steps:**  **- name: Get code**  **uses: actions/checkout@v3**  **- name: Install dependencies**  **run: npm ci**  **- name: Build website**  **run: npm run build**  **- name : Upload Artifact**  **uses : actions/upload-artifact@v3**  **with:**  **name: website**  **path: dist**  **deploy:**  **needs: build**  **runs-on: ubuntu-latest**  **steps:**  **- name: Download Artifacts**  **uses: actions/download-artifact@v3**  **with:**  **name: website**  **- name: List Generated Files**  **run: ls**  **- name: Deploy**  **run: echo "Deploying..."** | **build**   * **Step: Upload Artifact**:   + This step uses the `**actions/upload-artifact**` action to upload the `dist` directory (built website) as an artifact named "website".     **deploy**   * `**Download Artifacts**`: This step uses the `**actions/download-artifact**` action to download the artifact named "website" that was uploaded in the `build` job. * `List Generated Files`: This step runs the `ls` command to list the files in the downloaded artifact directory. * The `build` job builds the website, uploads the `dist` directory as an artifact, and depends on the successful completion of the `test` job. * The `deploy` job downloads the artifact from the `build` job, lists the generated files, and can be extended to include deployment logic. **The `deploy` job depends on the successful completion of the `build` job.** |

A screenshot of a computer

Description automatically generated

## JOB OUTPUTS

* Job outputs in GitHub Actions allow us to pass data from one job to another within a workflow.
* This can be useful when we want to share computed values (like API responses or build metadata) between different stages of the workflow.

### EXAMPLE - 1

|  |  |
| --- | --- |
| name: Job Output Example  on: push  jobs:  job1:  runs-on: ubuntu-latest  outputs:  **result: ${{ steps.step1.outputs.result }}**  steps:  name: Generate Output  id: **step1**  run: echo "result=Hello from Job 1" >> $GITHUB\_OUTPUT  job2:  runs-on: ubuntu-latest  needs: **job1**  steps:  name: Use Job Output  run: echo "Job 1 Output: **${{ needs.job1.outputs.result** }}" | **Step 1: Define the First Job (Producer Job)**  In this job, you compute or generate some data that you want to share.  Use $GITHUB\_OUTPUT to set the output value.  **Step 2: Declare Job Outputs**  Specify job outputs in the job’s outputs section.  Link the job outputs to the step outputs.  **Step 3: Reference the Output in the Downstream Job**  Use needs.<job\_id>.outputs.<output\_name> to access the output. |

### EXPLANATION

|  |  |
| --- | --- |
| **Job 1 (Producer Job)**  outputs:  result: ${{ **steps**.step1.outputs.result }} | * Declares an output named result for job1. * Connects the job output to the step output from step1. |
| run: echo "result=Hello from Job 1" >> $GITHUB\_OUTPUT | * Sets the step output by writing to $GITHUB\_OUTPUT. |
| **Job 2 (Consumer Job)**  needs: job1  Specifies that job2 depends on job1.  run: echo "Job 1 Output: ${{ **needs**.job1.outputs.result }}" | Accesses the output from job1 using **needs.job1.outputs.result.** |

### NOTE

* The “**steps**” and “**needs**” are github context objects

#### STEPS CONTEXT OBJECT

* steps: <https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/accessing-contextual-information-about-workflow-runs#steps-context>
* The steps context contains information about the steps in the current job that have an [id](https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#jobsjob_idstepsid) specified and have already run using its various properties
* E.g :
  + steps.<step\_id>.outputs
  + steps.<step\_id>.outcome

#### NEEDS CONTEXT OBJECT

* Document : <https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/accessing-contextual-information-about-workflow-runs#needs-context>

### EXAMPLE -2

In the below example we are printing the JS files generated after the build jobs of a react app

A computer screen shot of a program

Description automatically generated

|  |
| --- |
| **name: Deploy website**  **on:**  **push:**  **branches:**  **- main**  **jobs:**  **test:**  **runs-on: ubuntu-latest**  **steps:**  **- name: Get code**  **uses: actions/checkout@v3**  **- name: Install dependencies**  **run: npm ci**  **- name: Lint code**  **run: npm run lint**  **- name: Test code**  **run: npm run test**  **build:**  **needs: test**  **runs-on: ubuntu-latest**  **outputs:**  **result: ${{steps.publish.outputs.script-files}}**  **steps:**  **- name: Get code**  **uses: actions/checkout@v4**  **- name: Install dependencies**  **run: npm ci**  **- name: Build website**  **run: npm run build**  **- name : Upload Artifact**  **uses : actions/upload-artifact@v4**  **with:**  **name: website**  **path: dist**  **- name: Publish JS filename**  **id: publish**  **run: echo "script-files=$(find dist/assets/\*.js -type f)" >> $GITHUB\_OUTPUT**  **deploy:**  **needs: build**  **runs-on: ubuntu-latest**  **steps:**  **- name: Download Artifacts**  **uses: actions/download-artifact@v4**  **with:**  **name: website**  **- name: List Generated Files**  **run: ls**  **- name: Output JS files**  **run: echo "${{needs.build.outputs.result}}"**  **- name: Deploy**  **run: echo "Deploying..."** |

## JOB ARTIFACTS VERSUS JOB OUTPUTS

|  |  |  |
| --- | --- | --- |
| Aspects | Job Outputs | Job Artifacts |
| **Purpose** | Pass small piece of data between jobs | Store and share files(e.g. logs and build files) |
| **Data Size** | Limited to small text values | Suitable for large files and datasets |
| **Persistence** | Temporary with the workflow | Can be downloaded after workflow completes |
| **Scope** | Available within workflow | Can be accessed after the workflow completes |
| **Access Syntax** | **needs.job\_id.outputs.output\_name** | Download artifacts in steps |
| **Use Case** | Share API responses and computation results | Share build artifacts, test, reports, deployment packages. |

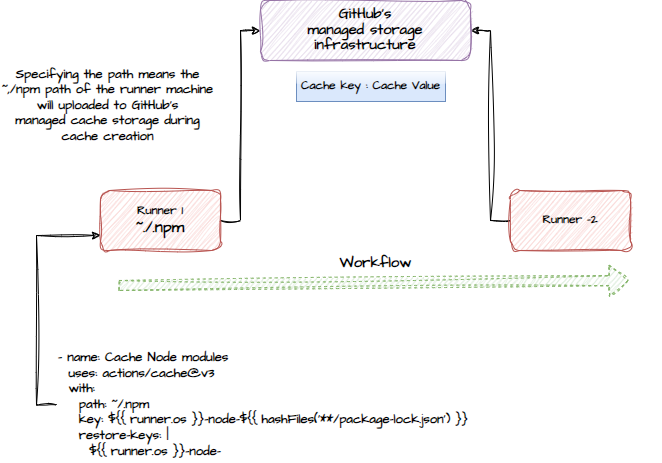
# DEPENDENCY CACHING

* Dependency caching in GitHub Actions allows us to cache dependencies, such as package managers, build tools, or other dependencies required by the workflow, to speed up subsequent workflow runs.
* Caching can help reduce the time it takes to download and install dependencies, especially when working with large dependencies or slow network connections.
* To use dependency caching in a GitHub Action workflow, we can utilize the **actions/cache action** provided by GitHub.

A screenshot of a computer

Description automatically generated

|  |  |
| --- | --- |
| **name: Deploy website**  **on:**  **push:**  **branches:**  **- main**  **jobs:**  **test:**  **runs-on: ubuntu-latest**  **steps:**  **- name: Get code**  **uses: actions/checkout@v3**  **- name: Install dependencies**  **run: npm ci**  **- name: Lint code**  **run: npm run lint**  **- name: Test code**  **run: npm run test**  **build:**  **needs: test**  **runs-on: ubuntu-latest**  **steps:**  **- name: Get code**  **uses: actions/checkout@v4**  **- name: Install dependencies**  **run: npm ci**  **- name: Build website**  **run: npm run build**  **- name : Upload Artifact**  **uses : actions/upload-artifact@v4**  **with:**  **name: website**  **path: dist**  **deploy:**  **needs: build**  **runs-on: ubuntu-latest**  **steps:**  **- name: Download Artifacts**  **uses: actions/download-artifact@v4**  **with:**  **name: website**  **- name: List Generated Files**  **run: ls**  **- name: Deploy**  **run: echo "Deploying..."** | * As in this workflow, we only install dependencies once in one job and then cache them so that other jobs can reuse those cached dependencies and save time * Like dependencies we can cache, files and folders in general across workflow executions(*if the files and folders that don't change at all across different workflows*), so that they don't have to be re-downloaded or recreated all the time. |



* When we use cache actions in GitHub Actions across multiple jobs and runners, the cache is stored **in GitHub's managed storage infrastructure, not locally on any specific runner**. Here's how it works:

**Key Points:**

1. **Global Storage:** The cache is uploaded and stored on GitHub's servers, not on individual runners.
2. **Cache Key:** The cache is identified by a specific key (which you define) and is accessible by any runner as long as the key matches.
3. **Multiple Runners:** If multiple jobs run on different runners (whether self-hosted or GitHub-hosted), they can still share the same cache as long as the cache key matches and the cache is available in the GitHub cache storage.
4. **Cache Retrieval:** When a job starts, GitHub Actions fetches the cache from its storage and downloads it to the runner.
5. **Cache Scope:**
   * Caches are scoped to the repository and branch by default.
   * You can override this with restore-keys if needed.

|  |  |
| --- | --- |
| **Example Configuration:**  **- name: Cache Node modules**  **uses: actions/cache@v3**  **with:**  **path: ~/.npm**  **key: ${{ runner.os }}-node-${{ hashFiles('\*\*/package-lock.json') }}**  **restore-keys: |**  **${{ runner.os }}-node-** | * In this example, all runners can access the cache if they match the cache key. * The path in the GitHub Actions cache action (actions/cache) defines the location of the files or directories on the runner that we want to cache. * The Cache key is created dynamically using hashFile(). The hashFile create a new hash whenever the file content changes. * Hence when a new dependency is added to package-lock.json 🡪 Change the file 🡪 new Hash generated * This make sure that a new cache is created when a dependencies changes (in package-lock.json) |

**WHY IS PATH IMPORTANT?**

The path tells the caching system what files or directories to:

* Upload to GitHub's managed cache storage during cache creation.
* Restore from the cache during subsequent workflow runs.

**Why Use ~/.npm as the Cache Path?**

* In Node.js projects, the ~/.npm directory (on Linux/macOS; C:\Users\<User>\AppData\Roaming\npm-cache on Windows) contains the NPM cache, which stores downloaded packages.

**Important Notes:**

* Caches are **immutable**; once created, they cannot be updated. You must create a new cache with a new key if changes occur.
* If two jobs with the same cache key run simultaneously, they might not find the cache created by each other immediately.

|  |
| --- |
| The hashFiles() function in GitHub Actions is used to compute a hash (SHA-256 checksum) of one or more files. This hash is then used to create a unique cache key.  **Why Use hashFiles()?**   * To **invalidate caches** when certain files change. * Typically used with dependency-related files, like package-lock.json for Node.js projects, to ensure a fresh cache when dependencies are updated.   **How It Works**  If the contents of the files specified in hashFiles() change, the computed hash will also change. This results in a new cache key, forcing the creation of a new cache.  **Example Usage**  key: ${{ runner.os }}-node-${{ hashFiles('\*\*/package-lock.json') }}  **Explanation:**   * ${{ runner.os }} ensures the cache key is OS-specific. * hashFiles('\*\*/package-lock.json') computes a hash of package-lock.json. * If package-lock.json changes, a new cache will be created. |

* The above example is for Node based projects . For other project set up refer : <https://github.com/actions/cache?tab=readme-ov-file#implementation-examples>

## EXAMPLE

|  |  |
| --- | --- |
| **name: Deploy website**  **on:**  **push:**  **branches:**  **- main**  **jobs:**  **test:**  **runs-on: ubuntu-latest**  **steps:**  **- name: Get code**  **uses: actions/checkout@v3**  **- name: Cache dependencies**  **uses: actions/cache@v4**  **with:**  **path: ~/.npm**  **key: deps-node-modules1-${{ hashFiles('\*\*/package-lock.json') }}**  **- name: Install dependencies**  **run: npm ci**  **- name: Lint code**  **run: npm run lint**  **- name: Test the code**  **run: npm run test**  **build:**  **needs: test**  **runs-on: ubuntu-latest**  **outputs:**  **result: ${{steps.publish.outputs.script-files}}**  **steps:**  **- name: Get the code**  **uses: actions/checkout@v4**  **- name: Cache dependencies**  **uses: actions/cache@v4**  **with:**  **path: ~/.npm**  **key: deps-node-modules1-${{ hashFiles('\*\*/package-lock.json') }}**  **- name: Install dependencies**  **run: npm ci**  **- name: Build website**  **run: npm run build**  **- name : Upload Artifact**  **uses : actions/upload-artifact@v4**  **with:**  **name: website**  **path: dist**  **- name: Publish JS filename**  **id: publish**  **run: echo "script-files=$(find dist/assets/\*.js -type f)" >> $GITHUB\_OUTPUT**  **deploy**  **z:**  **needs: build**  **runs-on: ubuntu-latest**  **steps:**  **- name: Download Artifacts**  **uses: actions/download-artifact@v4**  **with:**  **name: website**  **- name: List Generated Files**  **run: ls**  **- name: OutPut JS files**  **run: echo "${{needs.build.outputs.result}}"**  **- name: Deploy**  **run: echo "Deploying..."** | * We are caching the dependencies for each job * The cache action must be placed before the “npm ci” job      * When the workflow runs the “test” job will create the cache which will be further used by “build” job, hence the it will take relatively less time to execute * In next run of the workflow – even the “test” also will use the same cache hence overall workflow execution time will be less. * ***If the dependency get updated,🡪 updates the package-lock.json 🡪 New cache key will get generated 🡪 Old cache gets invalidated 🡪 New cache created*** |

# ENVIRONMENT VARIABLES AND SECRETS

* The value of environment variables varies with the environment . For instance, when running a program in a development environment, the values of environment variables may be set differently compared to when the same program is deployed in a production environment.

## DEFINING ENVIRONMENT VARIABLES

### WORKFLOW-WIDE VARIABLES

|  |  |
| --- | --- |
| **name: Example Workflow**  **on: [push]**  **jobs:**  **example-job:**  **runs-on: ubuntu-latest**  **env:**  **NODE\_ENV: production**  **API\_URL:** [**https://api.example.com**](https://urldefense.com/v3/__https:/api.example.com__;!!IqUcNYopQPk7!JsXMB_aW0dMqkdRB7jPu0q-ng6WKhvmM9clwnZNRng9t0ppbmoBq8ehHafaSPQeBpcF8xR3BcNRns8C8chijZQBiWMkn$)  **steps:**  **- name: Print Environment Variables**  **run: echo "Running in $NODE\_ENV mode. API: $API\_URL"** | * The environment variable defined on workflow level will be accessible to all jobs and steps in the workflow * The value can be overridden when a environment variable with same name is defined at the Job level |

### JOB-SPECIFIC VARIABLES

|  |  |
| --- | --- |
| The job specific environment variable is accessible within the job itself | **jobs:**  **example-job:**  **runs-on: ubuntu-latest**  **env:**  **NODE\_ENV: development**  **steps:**  **- name: Print NODE\_ENV**  **run: echo "NODE\_ENV is $NODE\_ENV"** |

### STEP-SPECIFIC VARIABLES

|  |  |
| --- | --- |
| Similarly, the step specific environment variable is accessible within the step itself | **steps:**  **- name: Set Variable for Step**  **run: |**  **export STEP\_VAR="This is a step variable"**  **echo "Variable is $STEP\_VAR"** |

### EXAMPLE

* In the below example – we have a Express app which connect with MongoDB .
* In the Github action, the test job can execute when the server is up and running.
* The application need a MongoDB database , so that test cases can execute
* The DB connection string is using environment variables for DB connection configuration.
* We can configure the value of environment variable using GitHub Action.

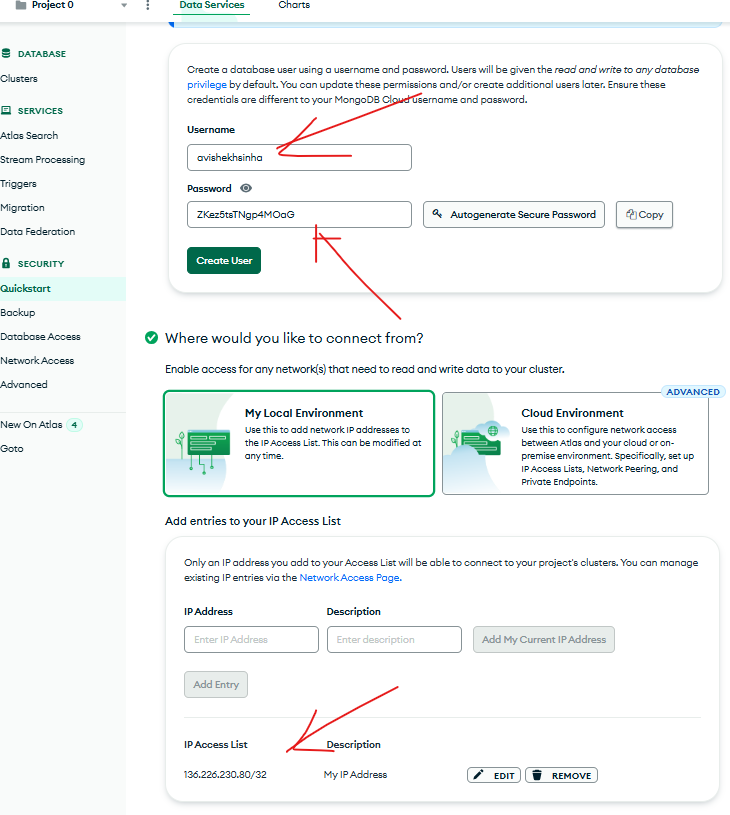
A screen shot of a computer code

Description automatically generated

#### SETTING UP MONGODB DATABASE

A screenshot of a computer

Description automatically generated

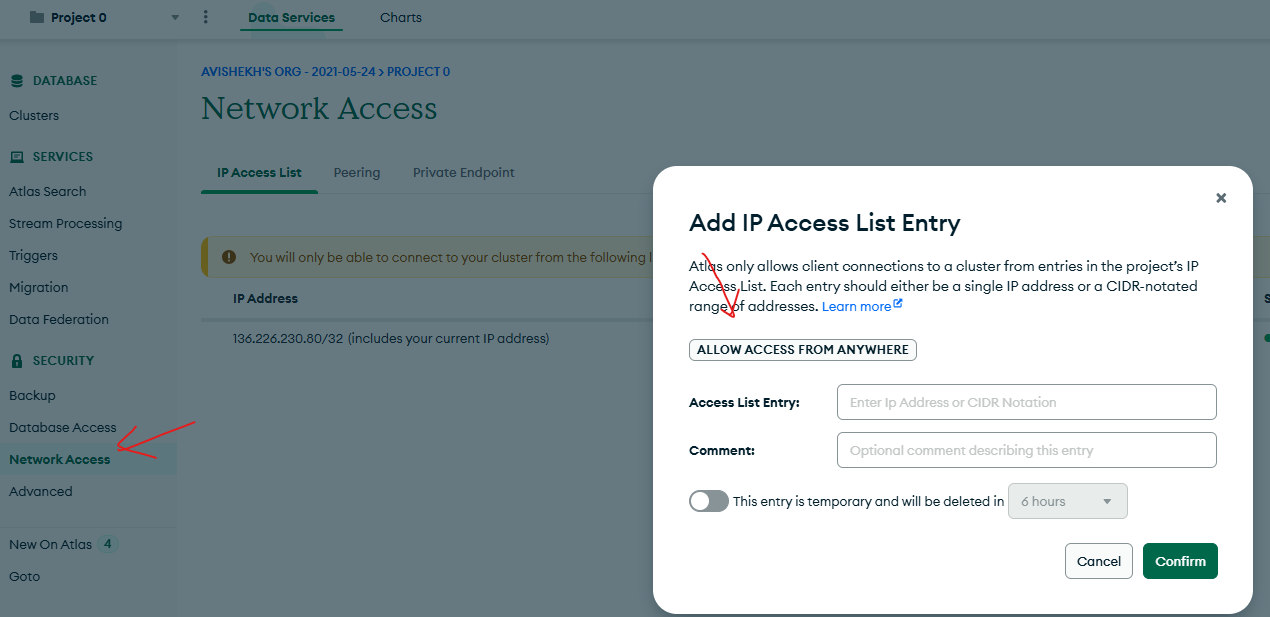


A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated



#### WORKFLOW

|  |
| --- |
| name: Deployment  on:    push:      branches:        - main        - dev  env:  **MONGODB\_DB\_NAME: gha-demo**  jobs:    test:      env:  **MONGODB\_CLUSTER\_ADDRESS: cluster0.9toch.mongodb.net**  **MONGODB\_USERNAME: avishekhsinha**  **MONGODB\_PASSWORD: ZKez5tsTNgp4MOaG**        PORT: 3000      runs-on: ubuntu-latest      steps:        - name: Get Code          uses: actions/checkout@v3        - name: Cache dependencies          uses: actions/cache@v3          with:            path: ~/.npm            key: npm-deps-${{ hashFiles('\*\*/package-lock.json') }}        - name: Install dependencies          run: npm ci        - name: Run server          run: npm start & npx wait-on http://127.0.0.1:**$PORT**        - name: Run tests          run: npm test  **- name: Output information**  **run: echo "MONGODB\_USERNAME ${{env.MONGODB\_USERNAME}}"**    deploy:      needs: test      runs-on: ubuntu-latest      steps:        - name: Output information          run: |  **echo "MONGODB\_USERNAME ${{env.MONGODB\_DB\_NAME}}"** |

* The environment variables can be accessed using interpolation “$”(**$PORT**) or using “env”( **${{env.MONGODB\_USERNAME}}"**) context object

|  |
| --- |
| * The env context object provides access to environment variables that are set in the workflow environment. * It allows us to retrieve the values of environment variables and use them within the workflow steps/ jobs. * To refer an environment variable using the syntax ${{ env.VARIABLE\_NAME }}. * For example, to access the value of an environment variable named MY\_VARIABLE, we would use ${{ env.MY\_VARIABLE }}. |

## USING SECRETS

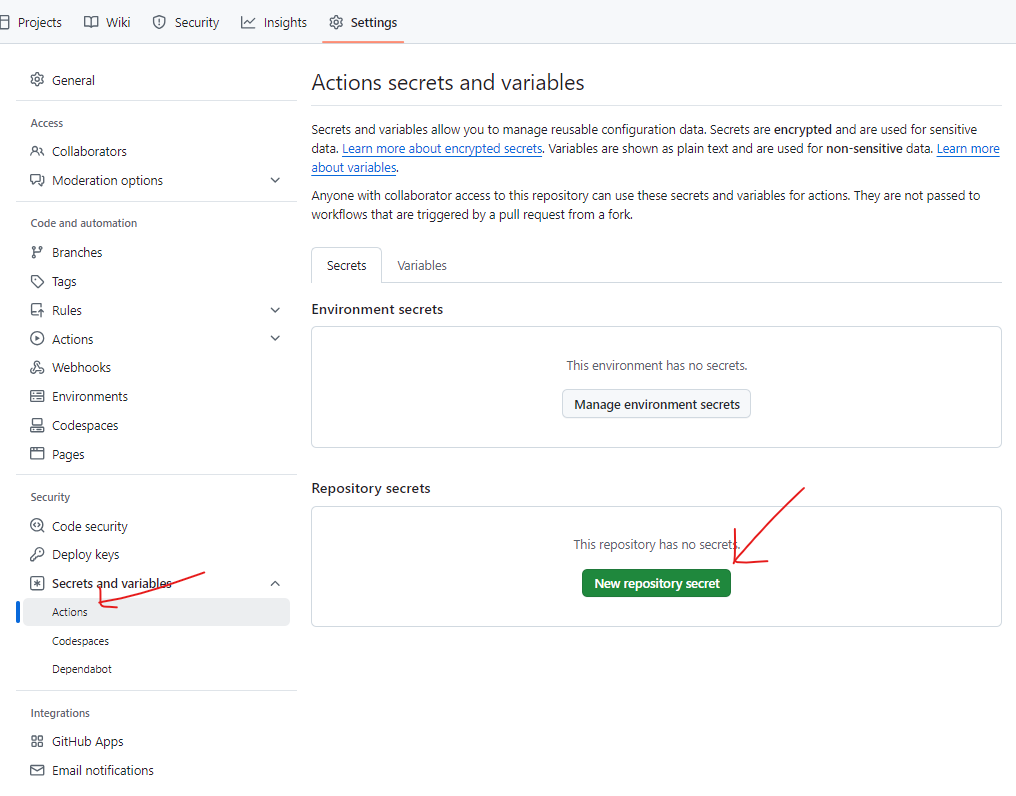
* In the above example – the username and password are sensitive information. Hence rather than having it in workflow yml we use GitHub Secrets:

|  |
| --- |
| * The secrets context object provides access to secrets that have been defined in the GitHub repository or organization. * Secrets are encrypted and can be used to store sensitive information like API keys, credentials, or any other secret values needed by your workflow. * To refer a secret using the syntax ${{ secrets.SECRET\_NAME }}. * For example, if you have a secret named MY\_SECRET, you can access its value using ${{ secrets.MY\_SECRET }}. * Note : that secrets are securely encrypted and can only be accessed within the same repository or organization where they are defined. They are not exposed in the workflow logs or accessible to unauthorized users.     Secrets are not exposed in the workflow logs |

### CREATING & STORING SECRETS

1. Go to Settings of the repository.s
2. Navigate to Secrets and variables > Actions.
3. Click New repository secret and set the key-value pair.

**Note: The secret will be accessible by all the Github actions within the repository.**

****

**A screen shot of a computer

Description automatically generated**

**A close-up of a white rectangle

Description automatically generated**

### ACCESSING SECRETS

|  |
| --- |
| **name: Deployment**  **on:**  **push:**  **branches:**  **- main**  **- dev**  **env:**  **MONGODB\_DB\_NAME: gha-demo**  **jobs:**  **test:**  **env:**  **MONGODB\_CLUSTER\_ADDRESS: cluster0.9toch.mongodb.net**  **MONGODB\_USERNAME: ${{secrets.MONGODB\_USERNAME}}**  **MONGODB\_PASSWORD: ${{secrets.MONGODB\_PASSWORD}}**  **PORT: 3000**  **runs-on: ubuntu-latest**  **steps:**  **- name: Get Code**  **uses: actions/checkout@v3**  **- name: Cache dependencies**  **uses: actions/cache@v3**  **with:**  **path: ~/.npm**  **key: npm-deps-${{ hashFiles('\*\*/package-lock.json') }}**  **- name: Install dependencies**  **run: npm ci**  **- name: Run server**  **run: npm start & npx wait-on http://127.0.0.1:$PORT**  **- name: Run tests**  **run: npm test**  **- name: Output information**  **run: echo "MONGODB\_USERNAME ${{env.MONGODB\_USERNAME}}"**  **deploy:**  **needs: test**  **runs-on: ubuntu-latest**  **steps:**  **- name: Output information**  **run: |**  **echo "MONGODB\_USERNAME ${{env.MONGODB\_DB\_NAME}}"** |

## REPOSITORY ENVIRONMENT

* The secrets will be available to all the workflow irrespective of which environment we are running the workflow.
* Repository environment variable enable as us set up environment specific secrets
* The workflow can then execute for a given environment using the secrets specified for that environment

A diagram of a software company

Description automatically generated

* We can also set up the branches is that it workflow can pick the environment and its corresponding environment secrets
* The code merged to “test” branch will pick the “testing” environment and its environment secrets and “prod” branch code merge will pick the “production” environment secrets

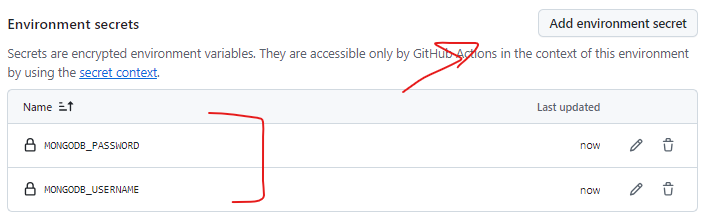
### CREATING ENVIRONMENT SECRETS

A screenshot of a computer

Description automatically generated

A white rectangle with blue lines

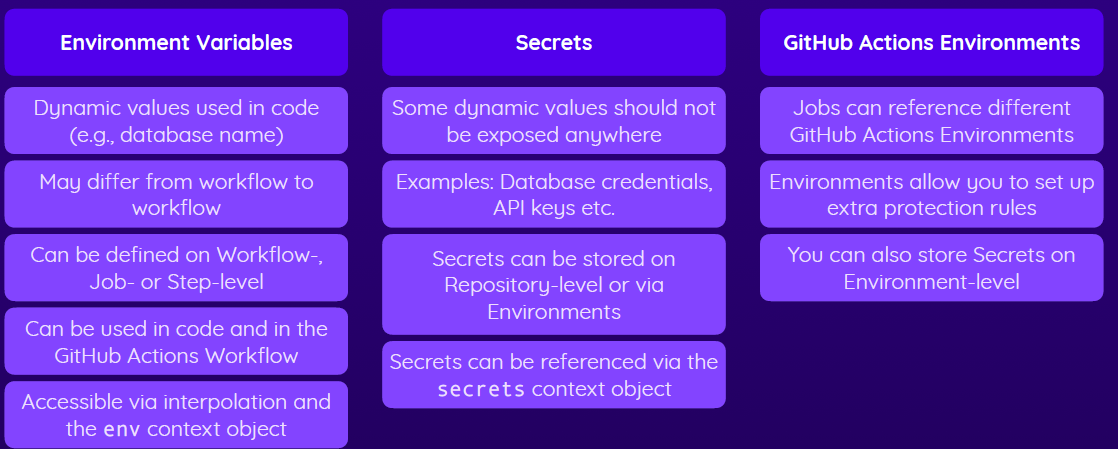
Description automatically generated



|  |
| --- |
| **name: Deployment**  **on:**  **push:**  **branches:**  **- main**  **- dev**  **env:**  **MONGODB\_DB\_NAME: gha-demo**  **jobs:**  **test:**  **environment: testing**  **env:**  **MONGODB\_CLUSTER\_ADDRESS: cluster0.9toch.mongodb.net**  **MONGODB\_USERNAME: ${{secrets.MONGODB\_USERNAME}}**  **MONGODB\_PASSWORD: ${{secrets.MONGODB\_PASSWORD}}**  **PORT: 3000**  **runs-on: ubuntu-latest**  **steps:**  **- name: Get Code**  **uses: actions/checkout@v3**  **- name: Cache dependencies**  **uses: actions/cache@v3**  **with:**  **path: ~/.npm**  **key: npm-deps-${{ hashFiles('\*\*/package-lock.json') }}**  **- name: Install dependencies**  **run: npm ci**  **- name: Run server**  **run: npm start & npx wait-on http://127.0.0.1:$PORT**  **- name: Run tests**  **run: npm test**  **- name: Output information**  **run: |**  **echo "MONGODB\_USERNAME ${{env.MONGODB\_USERNAME}}"**  **echo "MONGODB\_PASSWORD ${{env.MONGODB\_PASSWORD}}"**  **deploy:**  **needs: test**  **runs-on: ubuntu-latest**  **steps:**  **- name: Output information**  **run: |**  **echo "MONGODB\_USERNAME ${{env.MONGODB\_DB\_NAME}}"** |

A screenshot of a computer

Description automatically generated



## CONTROLLING WORKFLOW AND JOB EXECUTION

### CONDITIONAL JOB AND STEPS

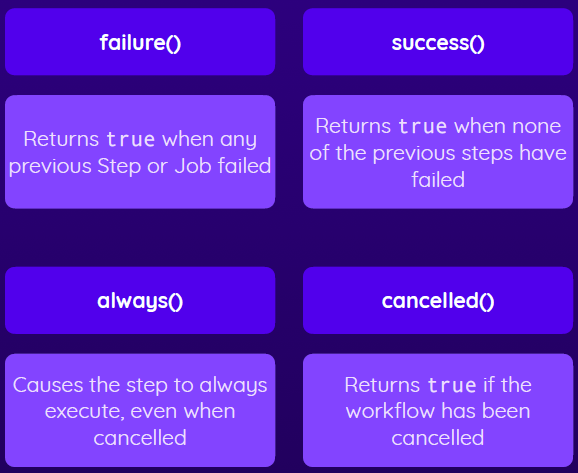
|  |  |
| --- | --- |
|  |  |

* In normal workflow if a job fails, it will stop the execution of next jobs. Conditional execution help in overcoming this issue.
* Conditional execution of jobs and steps can be achieved using the **if field**.
* **The if field can be added to both jobs and steps to specify a condition that must be met for the job or step to be executed.**
* Steps also have the **continue-on-error** field, which allows other steps in the job to continue executing even if a single step fails.
* Expressions can be used to define custom conditions for the if field and the continue-on-error field.

A screenshot of a computer

Description automatically generated

### SPECIAL CONDITIONAL FUNCTIONS



### CONDITION ON STEP LEVEL

* In the below workflow – we wanted to upload the test report only if the test cases fail.

**STEPS TO APPLY CONDITIONS ON STEPS FOR THIS USE CASE**

1. Step 1: Give “id” to the step which we want to watch the outcome
2. Step 2: Use the “steps” context object to deduce the outcome of the step
   1. **steps.run-tests.outcome == 'failure'**
3. Step 3: use the failure() conditional function along with step outcome to apply the if condition
   1. **if: failure() && steps.run-tests.outcome == 'failure'**

***Note: failure() returns “true” when any of the previous step or Job failed***

|  |
| --- |
| name: Website Deployment  on:    push:      branches:        - main  jobs:    lint:      runs-on: ubuntu-latest      steps:        - name: Get code          uses: actions/checkout@v3        - name: Cache dependencies          id: cache          uses: actions/cache@v3          with:            path: ~/.npm            key: deps-node-modules-${{ hashFiles('\*\*/package-lock.json') }}        - name: Install dependencies          run: npm ci        - name: Lint code          run: npm run lint    test:      runs-on: ubuntu-latest      steps:        - name: Get code          uses: actions/checkout@v3        - name: Cache dependencies          id: cache          uses: actions/cache@v3          with:            path: ~/.npm            key: deps-node-modules-${{ hashFiles('\*\*/package-lock.json') }}        - name: Install dependencies          run: npm ci        - name: Test code          id: run-tests 🡨 **ID of the test step**          run: npm run test        - name: Upload test report  **if: failure() && steps.run-tests.outcome == 'failure'**          uses: actions/upload-artifact@v4          with:            name: test-report            path: test.json    build:      needs: test      runs-on: ubuntu-latest      steps:        - name: Get code          uses: actions/checkout@v3        - name: Cache dependencies          id: cache          uses: actions/cache@v3          with:            path: ~/.npm            key: deps-node-modules-${{ hashFiles('\*\*/package-lock.json') }}        - name: Install dependencies          run: npm ci        - name: Build website          id: build-website          run: npm run build        - name: Upload artifacts          uses: actions/upload-artifact@v4          with:            name: dist-files            path: dist    deploy:      needs: build      runs-on: ubuntu-latest      steps:        - name: Get build artifacts          uses: actions/download-artifact@v4          with:            name: dist-files        - name: Output contents          run: ls        - name: Deploy          run: echo "Deploying..."  **report:**  **if: failure()**  **needs: [test, deploy]**  **runs-on: ubuntu-latest**  **steps:**  **- name: Report Issues**  **run: echo "Reporting issues...** |

### CONDITION ON JOB LEVEL

* As highlighted above the if condition can also applied at job level . if any of the job fails “report” job will execute else skipped.

“**report” job skipped**

A screenshot of a computer

Description automatically generated

“**report” job executed on failure**

A screenshot of a computer

Description automatically generated

## MATRIX STRATEGIES

* Matrix strategies in GitHub Actions allow us to define a set of variations for a job or a step to run multiple times with different inputs.

**MATRIX**

* A matrix is a way to define a set of variations or combinations for a job or a step to run.
* It allows us to run the same job or step multiple times with different inputs or configurations.
* Each combination in the matrix is represented by a set of key-value pairs.

**SYNTAX**

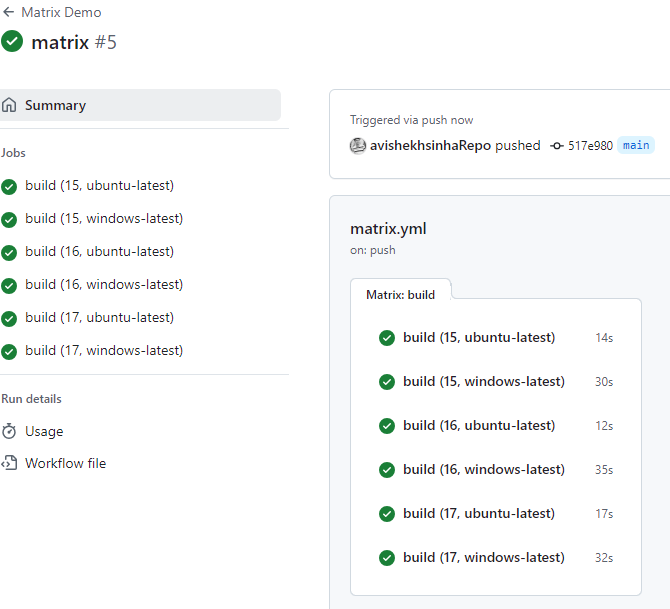
* The matrix is defined using the `**strategy**` keyword within a job or a step.
* The `**strategy**` keyword is followed by an object that contains the `matrix` keyword.
* Inside the `**matrix**` object, we define the key-value pairs for the variations we want to run.

**VARIABLES**

* The key-value pairs in the matrix represent variables that we want to vary for each run of the job or step.
* The values can be individual values, lists, or ranges.
* We can use expressions and environment variables to define the values.

|  |
| --- |
| **name: Matrix Demo**  **on: [push, workflow\_dispatch]**  **jobs:**  **build:**  **strategy:**  **matrix:**  **node-version: [15, 16, 17]**  **operating-system: [ubuntu-latest, windows-latest]**  **runs-on : ${{matrix.operating-system}}**  **steps:**  **- name : Get Code**  **uses: actions/checkout@v4**  **- name: Install Node JS**  **uses: actions/setup-node@v4**  **with:**  **node-version: ${{matrix.node-version}}**  **- name: Install Dependencies**  **run: npm ci**  **- name: Build Code**  **run: npm run build** |

Matrix strategies in GitHub Actions allow for efficient and flexible testing and configuration of jobs or steps by running them with different inputs or variations. They help automate repetitive tasks and make it easier to test our code against multiple environments or configurations.



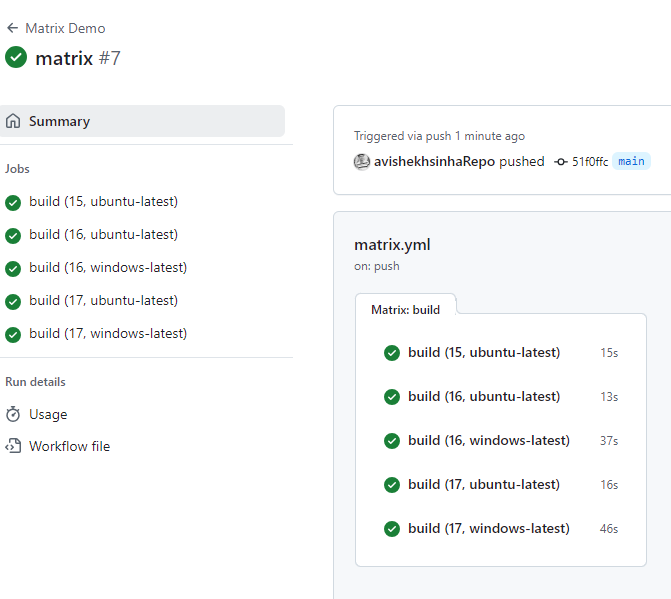
### EXCLUDING AND INCLUDING IN MATRIX STRATEGIES

* Matrix strategies allow us to define a set of dynamic values that can be used to create a matrix of jobs.
* By using the exclude and include keywords within the matrix strategy, we can further customize the matrix and specify which combinations of values **should be included or excluded**.

#### EXCLUDING FROM MATRIX STRATEGIES

* The exclude keyword allows us to exclude specific combinations of values from the matrix.
* We can specify the values to exclude using a combination of expressions, conditions, or patterns.

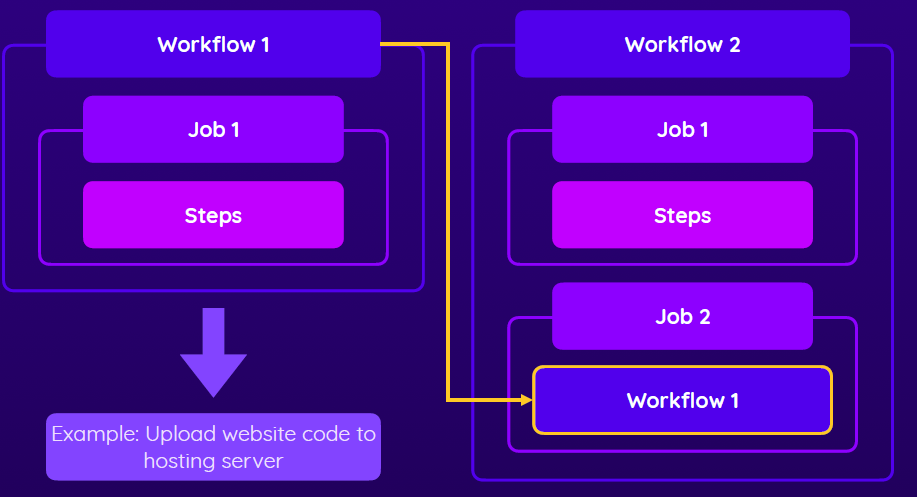
|  |  |
| --- | --- |
| **name: Matrix Demo**  **on: [push]**  **jobs:**  **build:**  **strategy:**  **matrix:**  **node-version: [15, 16, 17]**  **operating-system: [ubuntu-latest, windows-latest]**  **exclude:**  **- node-version: 15**  **operating-system: windows-latest**  **runs-on : ${{matrix.operating-system}}**  **steps:**  **- name : Get Code**  **uses: actions/checkout@v4**  **- name: Install Node JS**  **uses: actions/setup-node@v4**  **with:**  **node-version: ${{matrix.node-version}}**  **- name: Install Dependencies**  **run: npm ci**  **- name: Build Code**  **run: npm run build** | * From all the combinations of matrix defined the combination of node-version: 15 and operating system:window-latest will be excluded in the workflow run |



#### INCLUDING FROM MATRIX STRATEGIES

* The include keyword allows you to explicitly include specific combinations of values in the matrix.
* You can define the values to include using a combination of expressions, conditions, or patterns.
* Here's an example of including specific combinations in a matrix:

## REUSABLE WORKFLOW



* A reusable workflow in GitHub Actions allows us to define a workflow once and then reuse it across multiple repositories within an organization or across different organizations.

Here's an explanation of reusable workflows in GitHub Actions:

1. Workflow Template:

- To create a reusable workflow, you define it as a template that can be referenced by other workflows.

- The template contains the steps and configuration required for the workflow.

- You can define variables and parameters within the template to make it more flexible and customizable.

2. Workflow YAML File:

- The reusable workflow template is typically stored in a separate YAML file, separate from the individual repository workflows.

- This YAML file can be stored in a central location, such as a dedicated repository or a shared location within your organization.

3. Workflow Dispatch:

- To use the reusable workflow in a repository, you trigger it using a workflow dispatch event.

- The workflow dispatch event allows you to manually trigger the workflow and provide any necessary inputs or parameters.

4. Workflow Inputs:

- Workflow inputs can be defined within the reusable workflow template to allow for customization when triggering the workflow.

- These inputs can be used to pass values or configure the behavior of the workflow.

5. Workflow Outputs:

- The reusable workflow can also define outputs that can be used by other workflows or jobs within the same workflow.

- Outputs can be used to pass data or results from one step or job to another.

By creating reusable workflows, you can streamline your workflow management process and ensure consistency across multiple repositories or organizations. It also allows for easier maintenance and updates, as you only need to modify the template file instead of updating individual workflows in each repository. Additionally, reusable workflows promote code reuse and reduce duplication of effort.

### EXAMPLE

1. Create a file called `my-template.yml` in your repository to store the reusable workflow template.

2. Define the reusable workflow template in `my-template.yml`:

```yaml

name: Reusable Workflow

on:

workflow\_dispatch:

inputs:

param1:

description: 'Parameter 1'

required: true

jobs:

my-job:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Print Input

run: echo "Parameter 1: ${{ github.event.inputs.param1 }}"

```

3. In the repository where you want to use the reusable workflow, create a workflow file (e.g., `.github/workflows/use-template.yml`) that triggers the reusable workflow:

```yaml

name: Use Reusable Workflow

on:

push:

branches:

- main

jobs:

use-template:

runs-on: ubuntu-latest

steps:

- name: Trigger Reusable Workflow

uses: ./.github/workflows/my-template.yml

with:

param1: 'Hello, World!'

```

In this example, we have a reusable workflow template defined in `my-template.yml`. The template accepts an input parameter called `param1`. When the workflow is triggered in the repository, it will run the `my-job` job, which checks out the code and prints the value of `param1`.

The `use-template.yml` workflow file triggers the reusable workflow and provides the value `'Hello, World!'` as the `param1` input.

By referencing the reusable workflow template in other workflows, you can easily reuse the same set of steps and configuration across multiple repositories or organizations, while still allowing for customization through inputs.

## CUSTOM ACTIONS

* Custom actions in GitHub Actions allow you to create reusable and shareable units of automation.
* We can create our own custom actions to perform specific tasks or automate workflows

### TYPES OF CUSTON ACTIONS