Section 3: GitHub Actions - Basic Building Blocks & Components

03 August 2025 00:37

GitHub Actions: Getting Started – Fundamentals & Key Building Blocks

Overview

This section introduces the core concepts and building blocks of GitHub Actions, aiming to provide a solid foundation for creating and understanding GitHub Action workflows.

Core Concepts

What Are GitHub Actions?

- GitHub Actions is a powerful feature that automates workflows directly in your GitHub repositories.
- You can define steps to run jobs automatically in response to events (like pushing code or opening pull requests).

Key Elements Covered in This Section

- **Workflows:** The highest-level component, a workflow is an automated process that you define in your repository.
- **Jobs:** Each workflow consists of one or more jobs. Jobs run in parallel by default and are made up of steps.
- **Steps:** Steps are the smallest unit and define either a shell command or an action to run in a job.

Learning Approach

- The course focuses on hands-on learning with concrete workflows and real demonstrations, not just theory or slides.
- You'll see practical examples to help reinforce each concept and practice using GitHub Actions practically.

Pre-requisites

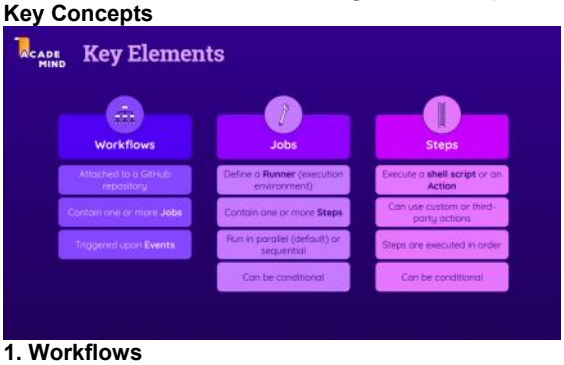
- GitHub Account:
- Required for following along; you can create one for free if you don't have it already.
- Basic Knowledge of Git and GitHub:
- If you're not familiar, refer to the Git-GitHub Crash course section recommended by the instructor.

What to Expect Next

- Deep dive into workflows, jobs, and steps.
- Explanations and example demonstrations of how to use and structure these building blocks in your own repositories.
  The coming lectures will provide detailed, step-by-step guidance and code examples to help you fully understand and leverage GitHub Actions in your projects.

GitHub Actions: Core Building Blocks Explained

Key Concepts



1. Workflows

- **Definition:** Automated processes attached to a GitHub repository.
- **Placement:** Stored in your repository and responsible for managing automation sequences.
- **Multiplicity:** You can create as many workflows as you want per repository.
- **Purpose:** The starting point—each automation process begins with a workflow.
- **Triggers (Events):** Workflows run in reaction to specific events, such as:
  - Manual activation,
  - A new commit being pushed,
  - Opening a pull request, etc.
- **Structure:** Each workflow contains one or more jobs.

2. Jobs

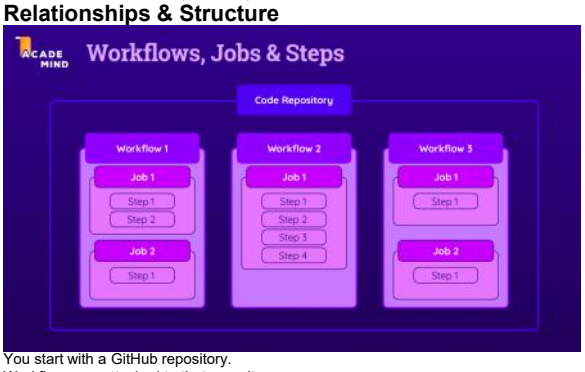
- **Definition:** A unit of work that runs inside a workflow.
- **Multiplicity:** Each workflow contains one or more jobs.
- **Execution:**
  - By default, multiple jobs run in parallel.
  - They can be configured to run sequentially or only under certain conditions (conditional jobs).
- **Runner:**
- Every job specifies a runner, which is the environment (machine & operating system) where the job runs.
  - **Runners can be:**
    - GitHub-hosted (provided by GitHub for Linux, macOS, or Windows).
    - Self-hosted (your custom machine or cloud instance).
- **Contents:** Each job contains one or more steps to execute within its runner.

3. Steps

- **Definition:** The smallest unit of work inside a job.
- **Multiplicity:** One or more steps per job.

- **Execution Order:** Steps are always executed in order, from top to bottom—not in parallel.
- **Types:** Each step is either:
  - A shell command (run as a script/command-line instruction),
  - Or an action (a reusable script or program, which could be written by you or a third party).
- **Conditional Steps:** Steps can be executed based on conditions.
- **Examples:**
  - Download code (first step),
  - Install dependencies (second step),
  - Run automated tests (third step).

Relationships & Structure

- 

- You start with a GitHub repository.
- Workflows are attached to that repository.
- Workflows contain jobs.
- Jobs contain steps.
- Each level can have multiple items: multiple workflows per repo, multiple jobs per workflow, multiple steps per job.

```
Repository
└── Workflow(s)
    └── Job(s)
        └── Step(s)
```

Additional Notes

- Flexibility: There's no fixed limit for the number of workflows, jobs, or steps—a structure can be as simple or as granular as needed.
- Real Usage: You'll commonly set up:
- Several workflows for different automation tasks (e.g., CI, deployment, code analysis).
- Jobs split logically (e.g., test vs. build vs. deploy).
- Steps for each action the job should perform—executed in sequence.

Summary Table

Component	Description	Notes
Workflow	Full automation process definition	Attached to repo; triggered by events
Job	Unit of work within a workflow, runs on a runner	Multiple per workflow; parallel by default
Step	Single operation/command within a job	Shell command or action; executed in order

In the next part of your course, you'll observe these concepts in practice with live demonstrations and concrete workflow examples, reinforcing these fundamental ideas through hands-on experience.

Creating Your First GitHub Actions Workflow (Step-by-Step)

1. Setting Up the Repository

- Using a GitHub account on the free plan.
- Create a new repository named **gh-first-action**.
  - The repository name does not matter.
  - Make it public.
  - Add a README file.
- Do not connect to or clone a local repository —stay on GitHub for this example.

2. Accessing Actions

- Once the repository is created, you land on a screen showing the README.
- Do not clone locally; you can, but that's not done here.
- Click the **Actions** tab.
  - This is where you view and create actions (workflows) for this repository.
- Currently, "action" is used as a synonym for "workflow", which is sufficient for now.
- What you create here is actually a workflow.

3. Creating a Workflow from a Template

- Select the simple workflow template and click **Configure**.
  - Alternatively, you can start with other workflow templates for more complex tasks.
  - For now, stick with the simple workflow.

4. Editing the Workflow

- The editor opens in the browser.
- You can also create actions locally, but for now, use the browser editor.
- **File Name:** Name it **first-action**.
- **Location:** The workflow file is stored in **.github/workflows/first-action.yml.**
  - .github folder → workflows subfolder → first-action.yml file.
- **Format:** YAML file (text formatting language).
- **Requirement:** Store workflows in .github/workflows/ and as YAML files for GitHub to detect them.

5. Writing the Workflow File

- Delete the template content and write the workflow from scratch for understanding.

Steps, with explanation inline:

1. **Workflow Name:**
   - Add name: First Workflow
2. **Trigger (Event):**
   - Add on: workflow_dispatch
   - This means the workflow is manually triggered by the user.
3. **Define Jobs:**
   - Add jobs: (must be plural, not job)
   - Indent and add your first job (job name of your choice, e.g., first-job)
4. **Job Runner:**
   - Under the job, add runs-on: ubuntu-latest
     - This selects an Ubuntu runner (Linux environment).
5. **Steps:**
   - Under the job, add steps:
   - Add steps as a dash -
   - Each step needs a name and a run command.
     - First step: Name = Print greeting, Command = echo "Hello World!"
     - Second step: Name = Print goodbye, Command = echo "Done - bye!"

Final Code:

```
name: First Workflow
on: workflow_dispatch
jobs:
  first-job:
    runs-on: ubuntu-latest
    steps:
      - name: Print greeting
        run: echo "Hello World!"
      - name: Print goodbye
        run: echo "Done - bye!"
```

Note:
Indentation in YAML is important. Each section must be indented as shown.

Committing and Running Your First Workflow

- With the job defined, start a commit to save the workflow file and add it to the repository.
  (This is important because GitHub Action workflows are part of your code —they are defined in files inside your Git repository, not outside.)
- After adding the workflow:
  - Go back to the **Actions** tab.
  - The appearance of this tab changes; you now see an overview of past workflow runs and a list of all workflows GitHub found in .github/workflows.
- For the workflow you just created:
  - You'll see its name (e.g., **First Workflow**).
  - GitHub indicates this workflow uses the workflow_dispatch trigger, so you can trigger it manually.
    (If you hadn't set workflow_dispatch, the manual run button would not appear.)
- You can now run the workflow manually:
  - Click the button to run the workflow against the main branch.
  - The workflow will execute.
- While it's executing:
  - In the Actions tab:
    - A yellow dot appears while running .
    - When it finishes, you'll see a green checkmark indicating success .
- Click on the workflow run for more details:
  - See the workflow file, event, and the job(s) executed.
  - Click the job to see step-by-step execution details.
    - Steps include automatic Setup and Cleanup, plus your own defined steps ("Print greeting" and "Print goodbye").
    - You can expand each step to see commands and output.
- You can run this workflow as many times as you want —just go back to the workflow and trigger it again.

Running Multi-Line Shell Commands

Thus far, you learned how to run simple shell commands like:

```
run: echo "Something"
```

If you need to run multiple shell commands (or multi-line commands for readability), you can do so by adding the pipe symbol (|) after the run: key.

Example:

```
run: |
  echo "First output"
  echo "Second output"
```

This will run both commands in one step.

Using a More Realistic Project with GitHub Actions

- The demo project from previous lectures has now been cloned to the local system, showing it as a remote repository with a .github folder, which contains the workflows folder and the first -action.yml workflow file.
- Moving on to a different example:
  - A very simple **ReactJS web application** is used as the new project.
  - **You don't need to know ReactJS or web development** to follow along, as no meaningful ReactJS code is written or required to understand GitHub Actions.
  - The purpose of this project is to have a slightly more realistic scenario.
- **Running the project locally (optional):**
  - Make sure you have Node.js installed on your system.
  - Navigate into the project folder (second action react demo or the downloaded folder).
  - Run npm install to install all third-party dependencies.
  - Afterwards, run npm run dev to start a development server and preview the app in the browser.
- **Testing:**
  - The demo project includes a test script for automated tests.
  - Tests can be run with npm test.
  - All tests should pass by default.
  - The specifics of testing and unit test code aren't important for understanding Actions here.
- **Typical project workflow:**
  - These setup and test steps are usually performed locally.
  - It's common to use Git for version control and GitHub for remote repositories, especially for team collaboration.
  - It is practical to want automated tests to run **whenever a new version of the code is pushed to GitHub**.
- **Setting up Git locally:**
  - Initialize a Git repository in the project folder.
    - Note: The .gitignore file is present and, for example, ignores the node_modules folder (which contains installed third-party dependencies that can always be regenerated, so shouldn't be checked into version control).
  - Stage all files and create an initial commit.
- **Setting up on GitHub:**
  - Go to GitHub and add a new repository (e.g., named "second action react demo").
    - The repository name and settings (private/public, description) are up to you.
    - For a change, this example uses a **private repository** and does **not** add new files on GitHub, because all the files exist locally.
- **Connecting local and remote:**
  - Copy the remote repository URL from GitHub.
  - Add the remote repository to your local project with the git remote command.
    - Use origin as the identifier and include your GitHub username in the URL to ensure authentication on push.
  - Push your code to the repository (git push). You may be prompted for a password or personal access token, depending on previous usage.
  - Link your local branch to the remote branch for future pushes.
- **Result:**
  - Once pushed, reloading your GitHub repository page shows the code is now on GitHub.
  - At this point, there are **no GitHub Actions workflows** yet configured for this project.

Adding an Automated Test Workflow Locally (Step-by-Step, All Details Included)

Plan and Approach

- The goal is to **add a GitHub Actions workflow** that automatically runs tests.
- Although you could create this action through GitHub's Actions tab, you can also write the workflow file locally and push it up—there is no difference in structure or required keywords.

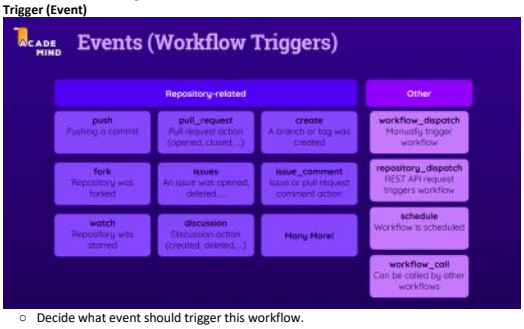
Directory and File Structure

- In your project (second-action-react-demo), create a **subfolder named .github** (the name is required—must be exactly .github).
- Inside .github, create a **subfolder named workflows** (also required—must be exactly workflows).

- Inside .github/workflows/, create a YAML file for your workflow (**the filename is up to you**; in this example, it's called test.yml).

Creating the Workflow File (test.yml)

1. **Workflow Name & Purpose**
   - Add a name key:
     This displays your workflow name in the GitHub Actions UI.
   - Example:
     ```
     name: Test Project
     ```
2. **Trigger (Event)**



   - Decide what event should trigger this workflow.
     Triggers and events are the same thing in GitHub Actions.
   - GitHub supports many events, most related to repository activity (push, pull_request, branch creation, repo forked, issues opened/closed/deleted, etc.).
   - There are also other events like:
     - workflow_dispatch (manual trigger)
     - repository_dispatch (triggered via REST API)
     - Scheduled triggers (run on a schedule, e.g. every day at 8:00 AM)
     - Workflow calling (triggered by other workflows)
   - For full details, the linked GitHub Docs page lists all possible events and their variations.
   - https://docs.github.com/en/actions/reference/workflows-and-actions/events-that-trigger-workflows
   - **For this project, you want the workflow to run whenever you push new commits to the repository**:
     ```
     on: push
     ```
     - push is a frequently used event and can be refined to specific branches, but for now, keep it simple.
3. **Defining Jobs**
   - All workflows require at least one job, defined under the jobs: key (jobs must be plural).
   - For this case, one job named test:
     ```
     jobs:
       test:
     ```
4. **Job Runner**
   - Specify the environment (runner) your job should use.
   - Here, use GitHub's hosted runner for Ubuntu:
     ```
     runs-on: ubuntu-latest
     ```
5. **Job Steps**
   - Steps are defined as a list under steps:, with each step starting with a dash (-).
   - **Naming each step makes logs and UI clearer.**
   - The very first step should **get the code**.
     - **Important:**
       - Workflows/job steps do NOT run inside your actual repository—they run on a separate, fresh server (a runner) provided by GitHub.
       - The runner does NOT have your code by default.
       - **If you want to install dependencies or run your project's actual code, you must first download it as the initial step**.
   - For now, just define the first step (the action to get the code):
     ```
     steps:
       - name: Get code
     ```
     (The actual implementation for "get code" will typically use a specific GitHub action, but this is just the step setup as described here.)

Full Example Structure

```
name: Test Project
on: push
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
```

Summary of Key Points

- **Events:** Workflows can be triggered by many types of events (push, pull_request, schedule, API calls, etc.).
  - You can refine them and define variations (e.g., only PRs opened, not closed).
  - The GitHub Docs page linked in your lecture lists all supported events and variants.
- **Why fetch code as the first step?**
  - Runner servers start "clean"—for code, dependencies, or tests to work, you need to bring code onto the runner.
- **Workflow files can be created locally or directly on GitHub—structure and keywords are always the same.**

This captures every detail from your supplied material with nothing added or missing.

Actions: A Key Building Block in GitHub Actions

- **Actions** are an essential component, separate from workflows, in the GitHub Actions system.
- When we started using GitHub Actions, "Action" and "Workflow" were sometimes used interchangeably, but technically, **a workflow defines automation logic**, while **an action is an individual, reusable application or script that executes a task**.

What is an Action?

- In GitHub Actions, **an Action is a custom or third-party application** that performs a complex or regularly used task.
  - For example, fetching code from a GitHub repository onto a runner machine.
- **Actions** are different from commands defined with run:, which execute shell commands or scripts you write directly in your workflow YAML file.
  - run:: You write and control the shell command/script logic.
  - **Action:** Predefined, reusable task, often for common needs (like "checking out" repo code).
- **Actions** can:
  - Be created and published by you, official maintainers, or the community.
  - Be freely shared and found in the GitHub Marketplace—where you can browse and use thousands of available actions for all sorts of automation needs.

Marketplace for Actions

- The **GitHub Marketplace** is a centralized place for finding and using Actions (and Apps, which are different from Actions).
  - **Most Actions are free**; the Marketplace simply makes finding and confirming the legitimacy of actions easier.
- **Verified Actions** (like the official checkout action) carry a badge indicating they're maintained by trusted creators (such as the GitHub team), offering confidence in their reliability and security.

Example: The Checkout Action

- Downloading the repository code to a runner is a **very common task**.
  - Technically, you could execute a shell command to clone the repo, but this is redundant because a standard GitHub Action already exists for it.
- The **Checkout Action** (actions/checkout) is:
  - Official, created/maintained by GitHub.
  - Verified and trusted.

    - Available in the Marketplace, with instructions for use.
- It's used to fetch your repository's code onto the runner at the start of a job, making it available for later steps (such as installing dependencies or running tests).

Using an Action vs. run:

- Use **run:** to execute custom, simple script commands that you write yourself.
- Use **an Action** (via the uses: keyword in YAML) when you want to incorporate a ready-made, reusable "component" that is maintained by someone else or the community.

Summary:
Actions are reusable, often complex tools distributed through the GitHub Marketplace. The official **Checkout Action** provided by GitHub is a prime example—it's widely used, trusted, and allows you to easily copy your repository code onto a runner without writing custom Git commands yourself. You can find and use thousands of different actions for all sorts of automation tasks w ithin GitHub Actions.

How to Use the Checkout Action in Your Workflow

To use an action (like the checkout action) in your GitHub Actions workflow:

- Instead of the run keyword (used for commands), use the uses keyword.
- The value for uses is the identifier of the action. For the official checkout action, it's actions/checkout.
- You should also specify a version for stability, like @v3.
- Some actions require additional configuration using the with keyword, but for the checkout action (and basic usage), this isn't necessary.

Example Workflow Step

Here is how you add the checkout action as your first step:

```yaml
name: Test Project
on: push
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
```

Key Points:

- The uses: actions/checkout@v3 line tells GitHub to use the official checkout action, version 3.
- Specifying the version (e.g., @v3) locks your workflow to that version and prevents unexpected issues if the action changes in the future.
- Most of the time, no extra configuration is required for this action; it will automatically checkout the repository attached to the workflow.
- If you need advanced behavior or settings, you can add a with: section under uses: and specify options (refer to the official action documentation for available settings).

This step ensures your runner server has your repository's code, ready for any following steps (like installing dependencies or running tests).

Adding the Install Node.js Step with the Setup-Node Action

After downloading your code with the checkout action, the next job step is to ensure Node.js is installed and all dependencie s can be installed. Since the ubuntu-latest runner already comes with Node.js preinstalled, you technically don't need to install it. However, to practice using actions and specifying specific versions, you can use the official setup -node action.

How to Add the Setup-Node Action

- Use the uses keyword to reference the action.
- Specify the version of the action (@v3).
- Use the with keyword to pass configuration options, such as the node-version you want installed.
- In this example, Node.js v18 is installed—even though the runner may already have Node.js, with this step you guarantee the use of the desired version.

Example Workflow with Both Steps

```yaml
name: Test Project
on: push
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install NodeJS
        uses: actions/setup-node@v3
        with:
          node-version: '18'
```

Explanation:

- Get code uses the official checkout action to download your repository's code.
- Install NodeJS uses the setup-node action. With the with block, you set node-version: '18', ensuring Node.js 18 is used.
- Even if Node.js is already present on the runner, specifying the action version and desired Node.js version is great practice for reproducibility and clarity.

Now your runner will have the code and the specific Node.js version needed for your automated test steps.

Extending the Test Workflow with Dependency Installation and Test Execution

Here is the expanded workflow for the test job that now includes installing dependencies and running tests on the runner:

```yaml
name: Test Project
on: push
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install NodeJS
        uses: actions/setup-node@v3
        with:
          node-version: "18"
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test
```

Explanation of New Steps

- **Install dependencies**:
  Runs the command npm ci on the runner.
  This command installs all dependencies specified in the package-lock.json, ensuring exact versions are used for consistency and avoiding unwanted version updates.
  (npm install would work but npm ci is preferred for automated environments.)
- **Run tests**:
  Runs the command npm test to execute your automated tests, just like you would locally.

Important Notes on Committing and Permissions

- After updating your workflow file locally, you must commit the changes and push them to your GitHub repository:
  ```
  git add .github/workflows/test.yml
  git commit -m "added test workflow"
  git push
  ```
- If you encounter an error on push related to permissions—specifically that your personal access token (PAT) does not have permission to create or update workflows—you will need to:
  1. Generate a **new personal access token** with the **workflow scope enabled** (sometimes called "GitHub Actions"

permission).
2. Remove the old token from your local environment or credential manager.
3. Configure your Git client to use the new token.
4. Retry the push, which should now succeed.

- Once the push succeeds, GitHub will detect the new workflow automatically and trigger it (because it is configured to run on push events).

Verifying Workflow Execution

- After push, navigate to the **Actions** tab on GitHub to see your workflow running.
- You will see an overview of the workflow runs, their status (running, success, failure).
- Click into a run to see detailed logs of each step:
  - Verify your dependencies were installed.
  - Verify your tests ran and their results are reported.
- Monitoring these results is critical, especially to detect and debug test failures.

This completes the setup of a meaningful test workflow that automatically runs your tests on every push, mirroring how you wo uld test locally but running in GitHub-hosted runners. The next lecture will cover how to handle workflow failures and inspect those results thoroughly.

Handling Failing Workflows and Tests in GitHub Actions

Simulating a Failed Test

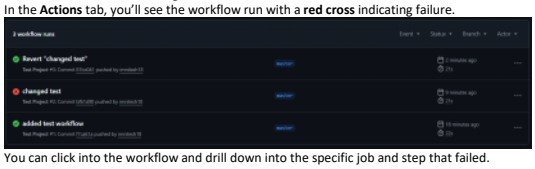
- To demonstrate workflow failure, deliberately break a test:
  - Edit the test file: src/components/MainContent.test.jsx
  - Modify line 19 by inserting .not before .toBeInTheDocument().
  - This changes the expectation, causing the test to fail.
- You can run npm test locally to confirm the test fails with this change.

Committing and Pushing the Broken Test

- Commit the broken test code with an appropriate commit message.
- Push the commit to GitHub.
- Since your workflow is triggered by push, pushing this change starts a new workflow run automatically.

Observing the Failed Workflow on GitHub

- The workflow will start running on GitHub.
- In the **Actions** tab, you'll see the workflow run with a **red cross** indicating failure.



- You can click into the workflow and drill down into the specific job and step that failed.

Debugging the Failure

- GitHub Actions provides detailed logs of each step.
- In this case, the log shows the test runner's output, highlighting which test expectation was not met.
- These logs help understand **why** the workflow or test failed.

Correcting the Failure

- Since the test was deliberately broken, the correct course is to fix it.
- You can revert the breaking commit by running:

```
git log
git revert <commit-id>
```

- This creates a new commit that undoes the faulty changes.

Pushing the Revert Commit

- Push the revert commit to GitHub.
- This triggers another workflow run automatically.
- This time, the workflow and tests should pass successfully because the test is fixed.

Summary of Commands

```
# Local test failure simulation
npm test
# Commit broken test
git add .
git commit -m "Break test to simulate failure"
git push
# After seeing failure on GitHub, revert broken commit
git revert <commit-id>
git push
```

Key Points

- Workflows triggered on push will run on every commit, including ones that introduce failing tests.
- GitHub Actions logs provide detailed error output to troubleshoot failing steps.
- Use git revert to undo problematic commits cleanly.
- This process ensures continuous integration workflows report failures early and help maintain code quality.

Adding Multiple Jobs to a GitHub Actions Workflow: Test and Deploy Jobs

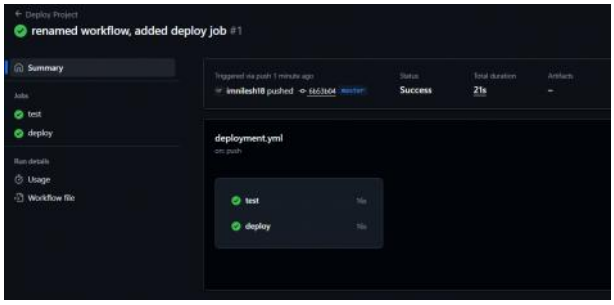
Overview

- You can have **multiple jobs** inside a single workflow.
- Jobs are defined at the same **indentation level** under jobs:.
- Each job has its own unique identifier (like test or deploy).
- Each job runs on its **own runner** (i.e., isolated environment).
- Jobs run **in parallel by default** unless configured otherwise.

Workflow File Details

```yaml
name: Deploy Project
on: push
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install NodeJS
        uses: actions/setup-node@v3
        with:
          node-version: "18"
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install NodeJS
        uses: actions/setup-node@v3
        with:
          node-version: "18"
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test
```

```
    - name: Build project
      run: npm run build
    - name: Deploy
      run: echo "Deploying..."
```



Important Points

- The deploy job shares many steps with the test job (getting code, installing Node.js, installing dependencies, running tests), but then continues with **building** and **deploying**.
- Indentation is critical in YAML:
  - Both job identifiers (test and deploy) should be aligned (same indentation).
  - The steps inside each job are indented under steps.
- Because the jobs run **in parallel**, they start simultaneously, which you can observe in the Actions UI.
- The total runtime is **not** the sum of the individual job runtimes because they execute concurrently.
- Running jobs in parallel optimizes workflow duration but some workflows require sequential execution depending on dependencies.

Triggering and Observing the Workflow

- After committing and pushing this updated workflow file:
  - The **Actions** tab in GitHub shows the renamed Workflow (Deploy Project).
  - The current workflow run lists **two jobs**: test and deploy.
  - Both jobs will run **simultaneously** and their progress is shown independently.
  - If both complete successfully, you see green checks by both jobs.

Summary

- Multiple jobs mean your workflow can perform **different tasks concurrently**, such as testing and deploying.
- To run jobs **sequentially**, you can configure dependencies using job needs: (covered in following lessons).
- Keeping job identifiers consistent and indentation correct is essential for YAML parsing and workflow success.

Running Jobs Sequentially Using needs Keyword and Handling Failures

Running Jobs After Each Other

- You can configure jobs in a workflow to **run sequentially** instead of in parallel.
- Use the **needs** keyword in the dependent job to specify which job(s) must complete before it starts.

deployment.yml

```
name: Deploy Project
on: push
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install NodeJS
        uses: actions/setup-node@v3
        with:
          node-version: "18"
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test
  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install NodeJS
        uses: actions/setup-node@v3
        with:
          node-version: "18"
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test
      - name: Build project
        run: npm run build
      - name: Deploy
        run: echo "Deploying..."
```

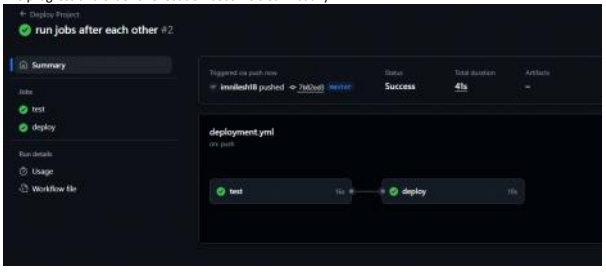
- The job with needs: test will **wait until the test job finishes successfully**.
- If the required job fails, the dependent job will **not start**.
- You can specify multiple jobs to wait for by using square brackets, e.g., needs: [job1, job2], but in this case, there's only one job to wait for.

Effect on Workflow Execution and Runtime

- Jobs will run one after the other, so the **overall workflow duration will increase** compared to running jobs in parallel.
- This setup makes logical sense to **only deploy if tests pass**, avoiding deploying broken or failing code.

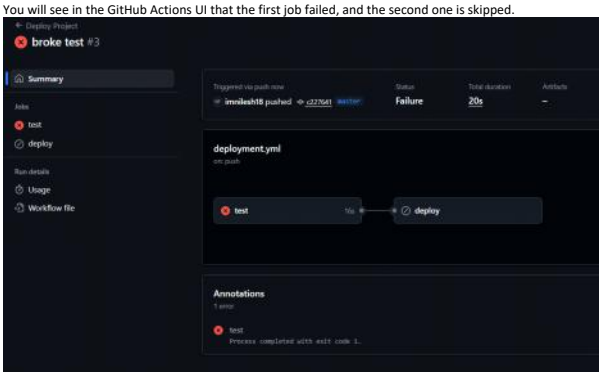
Visual Feedback on GitHub Actions UI

- In the GitHub Actions tab, the UI reflects the sequential dependency:
  - The dependent job (e.g., deploy) will only start after the prerequisite job (e.g., test) finishes successfully.
  - The progress and order of execution become clear visually.



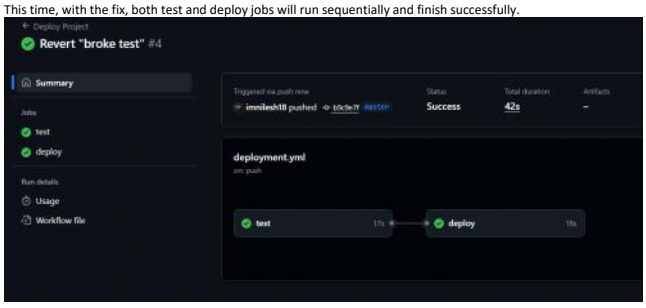
Handling a Failed Test Job

- If the test job **fails** (for example, a test is deliberately broken by adding .not in the code):
  - The deploy job **will not start**.
  - You will see in the GitHub Actions UI that the first job failed, and the second one is skipped.



Reverting a Broken Commit

- To fix a broken test commit:
  1. Get the commit ID of the last broken commit.
  2. Run git revert <commit-id> to create a new commit undoing the breaking change.
  3. Push the revert commit to GitHub.
- After push, a new workflow run starts.
- This time, with the fix, both test and deploy jobs will run sequentially and finish successfully.



Summary

- Use needs to control job execution order in workflows.
- Dependent jobs wait for prerequisite jobs to finish **successfully**.
- This prevents deployment of code if tests fail.
- Sequential job execution increases total workflow runtime but improves deployment safety.
- You can revert breaking commits to restore successful workflow runs.

Multiple Triggers for a Single Workflow

- Until now, each workflow was configured with **a single trigger event** (on:), such as a push to the repository.
- It's often practical to allow a workflow to run on **multiple events**:
  For example, whenever you **push code** or when you want to **trigger it manually** from the GitHub UI.

How to Specify Multiple Triggers

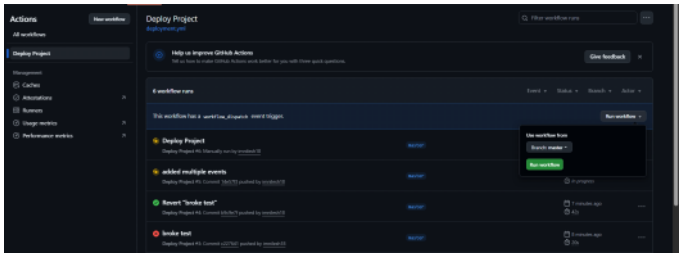
- Wrap the event names in square brackets and list them:
  `on: [push, workflow_dispatch]`
  - push : Runs whenever a commit is pushed to the repository.
  - workflow_dispatch : Enables a button in the GitHub Actions UI to **run the workflow manually**.

Example: Workflow File with Multiple Triggers

```yaml
name: Deploy Project
on: [push, workflow_dispatch]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install NodeJS
        uses: actions/setup-node@v3
        with:
          node-version: "18"
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test
  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install NodeJS
        uses: actions/setup-node@v3
        with:
          node-version: "18"
      - name: Install dependencies
        run: npm ci
      - name: Run tests
        run: npm test
      - name: Build project
        run: npm run build
      - name: Deploy
        run: echo "Deploying..."
```

Behavior with Multiple Triggers

- If you **push** to the repository, the workflow runs automatically.
- If you visit the workflow in GitHub's Actions tab, you now see a **Run workflow** button, letting you manually trigger a run.
- Both triggers are independent:
  You can have multiple workflow executions (manual and automatic) running in parallel or queued.



Next Steps (What's Possible)

- You can add as **many triggers (events)** as you want by listing them in the square brackets.
- GitHub Actions allows configuring each event in more detail (e.g., which branches on push) in advanced usage, which will be covered later.

In summary:
You can make any workflow listen to multiple triggers simply by wrapping the event names in an array under the on: key and li sting each desired event. This adds flexibility and control for your automation.

Expressions and GitHub Context in GitHub Actions

What Are Expressions and Context?

- **Expressions** are special syntax in GitHub Actions to access dynamic data or compute values at runtime rather than output plain text.
- **Context** refers to the metadata information GitHub automatically provides about the workflow run, the event, the runner environment, and more.
- Context enables workflows and steps to understand their surroundings and behave accordingly.

Syntax for Expressions

- Expressions are wrapped inside ${{ ... }}.
- This signals that the content inside should be evaluated and is not just literal text.
- Example of using GitHub context in a step's shell command:
  `run: echo "${{ toJSON(github) }}"`
  - Here, github is a reserved context identifier containing metadata about the workflow and event.
  - toJSON() is a built-in function that converts the context object into a JSON string, making it printable.

What is the GitHub Context?

- The github context object contains data such as:
  - Repository information (name, owner, URL)
  - The event that triggered the workflow
  - Event payload information
  - Workflow, job, and run identification data
- This context is automatically made available to jobs and steps at runtime.

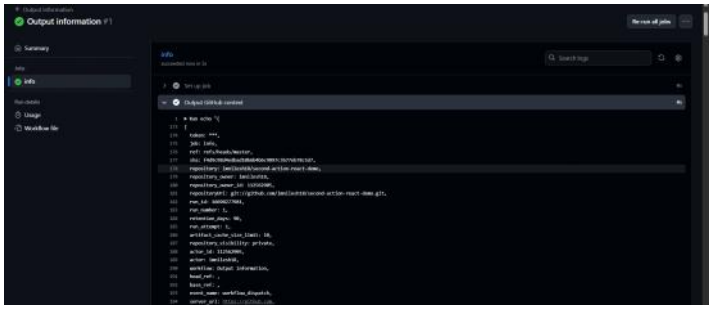
Why Use Expressions and Context?

- They allow you to write dynamic workflows that adapt based on the event, branch, or environment.
- Example use cases include:
  - Accessing repository details for API calls
  - Conditional execution based on event type
  - Using environment variables or runner details
- Using expressions, you can retrieve and format this data for handling in shell commands or passing into other steps.

Summary of the Example Workflow

```yaml
name: Output information
on: workflow_dispatch
jobs:
  info:
    runs-on: ubuntu-latest
    steps:
      - name: Output GitHub context
        run: echo "${{ toJSON(github) }}"
```

- This workflow triggers manually (workflow_dispatch).
- It runs a single job info on an Ubuntu runner.
- The single step prints the serialized github context object, showing the metadata available.



- This showcases how to use expressions to access and output GitHub context data dynamically.

Additional Notes

- The expression syntax and available context objects are a powerful feature to write flexible and environment-aware workflows.
- Common built-in functions include toJSON(), contains(), format(), etc.
- Contexts include github, env, runner, secrets, job, steps, and more.
- This introductory example simplifies the concept; you will see more advanced use cases throughout the course.

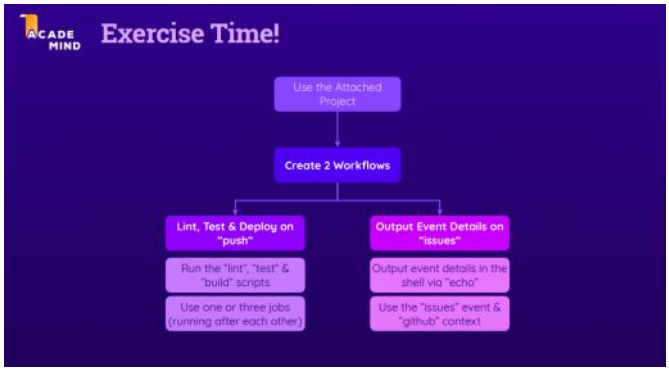
This explanation captures the core concept of expressions and contexts in GitHub Actions, the syntax to use them, and an exam ple reflecting your provided material exactly.

1. https://docs.github.com/actions/reference/evaluate-expressions-in-workflows-and-actions
2. https://docs.github.com/en/actions/concepts/workflows-and-actions/expressions
3. https://docs.github.com/en/enterprise-server@3.13/actions/writing-workflows/choosing-what-your-workflow-does/evaluate-expressions-in-workflows-and-actions
4. https://docs.github.com/actions/reference/workflow-syntax-for-github-actions

Section Summary: Core Components of GitHub Actions

- It all starts with **Workflows**, which are attached to GitHub repositories.
  In a workflow, you define which event or trigger should cause which jobs to be executed.
- **One workflow can have one or more jobs.**
  - **Jobs** are the next important building block.
    - With jobs, you define the environment in which your work will be executed (the server), and the steps to execute (in order).
    - **Jobs** execute on **runners** (servers/machines).
      - GitHub offers various predefined runners with different operating systems (Linux, Windows, macOS), each with different hardware profiles.
      - If needed, you can also bring your own runner, but that was not covered in this section.
- **Steps** are the instructions that do the actual work in each job.
  - In steps, you execute command line commands or use predefined **Actions**.
- **Jobs can run in parallel** (default) or **sequentially**, one after another.
- **Events and triggers** are very important.
  - There is a variety of events, mainly:
    - Repository-related (e.g., a push/commit to the repository)
    - Others, such as manually starting a workflow
  - **A workflow has at least one event, but can have more than one.**
- **Workflow definition:**
  - Defined in .github/workflows/ as a YAML file
  - The file can be created on GitHub or locally (most typical)
  - The file must be in the workflows folder inside the .github folder and follow the GitHub Actions syntax and keywords
- **Workflow execution:**
  - Executed whenever their connected event is triggered
  - You can view detailed insights and logs on the GitHub page
- **Actions in workflows:**
  - In job steps, you can use actions or your own shell commands with the run key
  - For more advanced and often repeated tasks, use **Actions** (official, community, or custom)
  - You will build your own actions later in the course
- These are the essentials needed to work with GitHub Actions, and you are now ready to dive deeper.

Practice Exercise Instructions



- Now that key concepts have been explored, it's time to practice.
- Another example project is attached for you to download and turn into a local Git repository.
- You should also create a GitHub repository for that attached project—a remote repository—and connect the local Git repository to the remote GitHub repository.
- In that GitHub repository, you should create **two GitHub Actions Workflows**:

First Workflow

- This workflow will **test**, **lint** the project, and **simulate deploying** the project whenever a new commit is pushed.
- Steps in this workflow:
  - Download the project code whenever it's triggered.
  - Install any dependencies needed.
  - Run the **lint**, **test**, and **build** scripts.
    - These scripts are included in the attached project's package.json file (scripts: lint, test, and build).
  - Output a message like "deploying project" as part of the workflow (a dummy message in the workflow environment's shell).
- For this workflow:
  - It's up to you whether to use **one job for all three scripts** or **three different jobs running after each other**.
  - You can also build both versions if you want to practice both approaches:
    - One workflow with one job.
    - One workflow with three jobs.

Second Workflow

- This workflow should listen to an **event not used yet** in the course section: the **Issues Event**.
  - The Issues Event (just like Push) is an official event/trigger.
  - It should start the workflow whenever a new issue-related action happens (e.g., when a new issue is created).
- In this workflow:
  - Simply output the **event details** in the shell using echo.
  - Use knowledge from earlier in the section about accessing metadata provided by GitHub in your workflow (use the GitHub Context Object).
    - Output the collected metadata provided by GitHub Actions for your workflow.
- For this workflow:
  - Listen to the **Issues Event** and use the **GitHub Context Object** as explained earlier.
- This is the exercise:

- Download and connect the project, then create the two workflows as described above.

Solution Notes for the Practice Exercise

Initial Setup

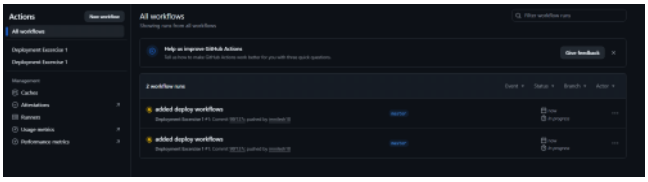
- The attached project is initialized as a Git repository.
- All files are staged and committed with the message initial commit.
- A new remote GitHub repository is created (named basic-exercise in this case; the name is up to you, and can be public or private).
- The remote is added using git remote add origin <URL>, ensuring the correct username is included for authentication.
- After updating the URL if needed, the code is pushed to GitHub.
- At this point, there are no GitHub Actions workflows yet.

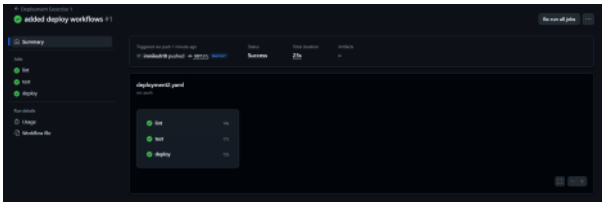
Creating Workflow YAML Files

1. .github/workflows/deployment1.yaml (One Job Version)

```yaml
name: Deployment Excercise 1
on: push
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Lint
        run: npm run lint
      - name: Test code
        run: npm run test
      - name: Build code
        run: npm run build
      - name: Deploy code
        run: echo "Deploying..."
```

- This workflow is triggered by any push.
- There is a single job (deploy) which:
  - Downloads the code,
  - Installs dependencies (npm ci),
  - Runs the lint script (npm run lint),
  - Runs tests (npm run test),
  - Builds the code (npm run build),
  - Outputs a dummy deployment message.



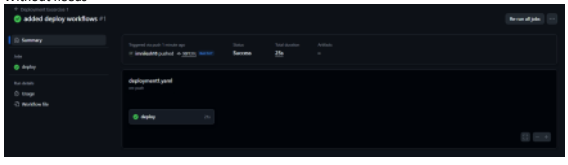


2. .github/workflows/deployment2.yaml (Three Jobs, Sequential)

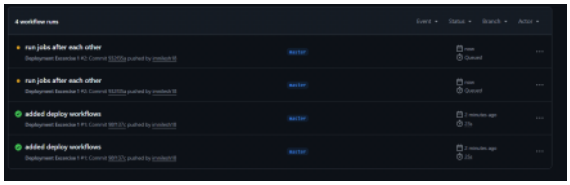
```yaml
name: Deployment Excercise 1
on: push
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Lint
        run: npm run lint
  test:
    needs: lint
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Test code
        run: npm run test
  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Deploy code
        run: echo "Deploying..."
```

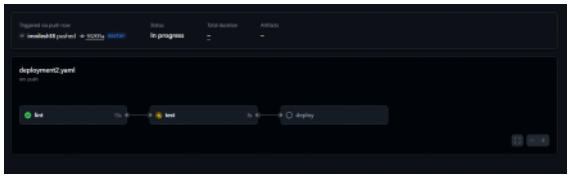
- The jobs are split into lint, test, and deploy.
- Each job runs after the previous (test runs after lint; deploy runs after test).
- Each job checks out the repository, installs dependencies, and runs its specific task.

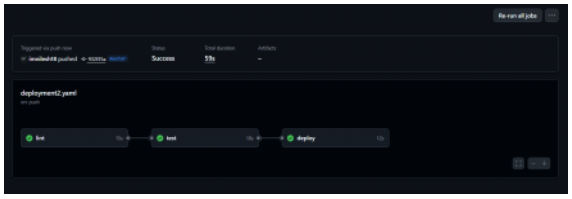
Without needs



With needs



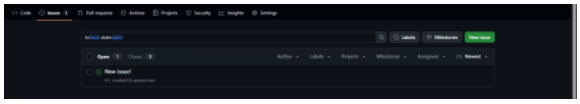




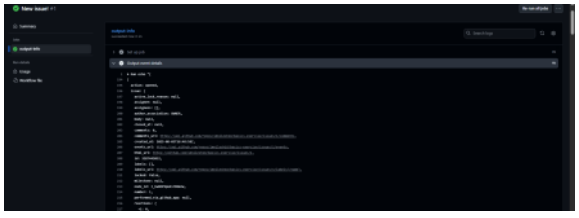
3. .github/workflows/issues.yml (Issues Event Workflow)

```yaml
name: Handle issues
on: issues
jobs:
  output-info:
    runs-on: ubuntu-latest
    steps:
      - name: Output event details
        run: echo "${{ toJson(github.event) }}"
```

- Triggered by any issue-related event (on: issues).



- The job output-info outputs event details to the shell using the context object and toJSON function.

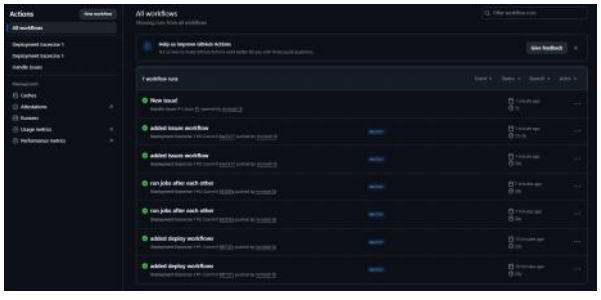


Execution and Outcome

- After committing and pushing these workflows, GitHub detects and triggers the workflows as expected.
- Both versions of deployment workflows execute and finish successfully:
  - The version with three jobs runs jobs simultaneously by default. To make them sequential, the needs keyword is used.
- The issues workflow only triggers when an issue is created or updated in the repository, outputting event details.
- All output and execution can be observed directly in the GitHub Actions tab.

Summary

- The exercise demonstrates:
  - Setting up local and remote repositories,
  - Creating various GitHub Actions workflows,
  - Configuring triggers, jobs, steps, and job dependencies,
  - Using GitHub context objects and expressions for dynamic data output.
- Each step and snippet exactly matches the transcript, with no added or omitted information.



Workflows & Events - Deep Dive

04 August 2025 00:46

Deeper Dive into GitHub Actions: Events and Filters

- In the previous course section, the basics of GitHub Actions were covered, including the general syntax used in YAML files for defining actions and the core features of GitHub Actions.
- In this section, the course goes deeper into GitHub Actions by focusing on the events that can trigger a workflow.
- It is possible to add more complex event definitions, not just simple ones where, for example, pushing to a repository triggers a workflow.
- More complex conditions can be set up, such as event filters, to control which push events (or others) actually trigger a workflow.
- The concepts of activity types will be introduced and explained.

Notes: Overview of Available Events in GitHub Actions

- 

- Before going into event filters or activity types, it's important to understand the available events in GitHub Actions.
- There is a broad variety of **event triggers** you can add to your workflows.
- **Most events are repository-related**, such as:
  - Triggering a workflow whenever a **commit is pushed** to the repository.
  - Triggering when a **pull request-related action** occurs.
- There are also **events not directly repository-related**, for example:
  - **Manually dispatching** (triggering) workflows.
  - **Scheduling** workflows to execute at regular intervals (e.g., every few hours or once a day).
- Throughout the course and this section, many key events and triggers will be demonstrated.
- For a complete overview, it is best to **explore the official documentation** to learn about all available events and read about: https://docs.github.com/en/actions/reference/workflows-and-actions/events-that-trigger-workflows
  - The different events assignable to workflows.
  - When exactly an event is considered triggered.
  - How to use and configure these events.

Starting State for the Events Course Section

- For this course section, another example project is prepared.
  - This is the same example project as before, but it already includes a workflow:
    - One YAML file is present in the .github/workflows folder.
    - The workflow defined in that file is a very simple demo workflow.
    - **However, the event is missing** in the workflow file because event configuration is the subject of this section.

```yaml
name: Events Demo 1
on: ...
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Output event data
        run: echo "${{ toJSON(github.event) }}"
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Test code
        run: npm run test
      - name: Build code
        run: npm run build
      - name: Deploy project
        run: echo "Deploying..."
```

- This demo project will be used throughout this section.
- A remote GitHub repository is needed to actually trigger GitHub Actions workflows:
  - A new local Git repository is initialized for the project.
  - An **initial commit** is created, even though the workflow is not finished yet (the event is still missing and will be completed later).
  - A new repository is created on GitHub (named GH events here, but the name is up to you—it can be public or private).
    - A **public repository** is chosen here to demonstrate something later in the course.
  - The GitHub repository's URL is copied, and the local repo is connected using git remote add origin <URL>, with the username included for authentication with the stored personal access token.
  - The initial push (git push) may fail at first, so the command to link the local main branch to the remote main

branch is run, and the initial commit is successfully pushed.

- **The starting state** now has the project code and the simple (but event-less) workflow present in the new GitHub repository.
- The course will next dive into the workflow and specifically into the events that can be assigned to it.

Events in GitHub Actions: Triggers and Fine Control

Main Events That Trigger Workflows

- The **push** event is one of the most important workflow triggers.
- To trigger a workflow on multiple events, list the event names in square brackets:
  - Example: `on: [push, workflow_dispatch]`
    This allows the workflow to run on both a push event and when manually triggered.

Default Behavior and Need for More Control

- With this configuration, **any push to any branch** will trigger the workflow.
- Often, you want workflows (like deployments) to run only for specific branches (such as main) and not for every branch.
- GitHub Actions provides features to control which pushes or events actually trigger the workflow.

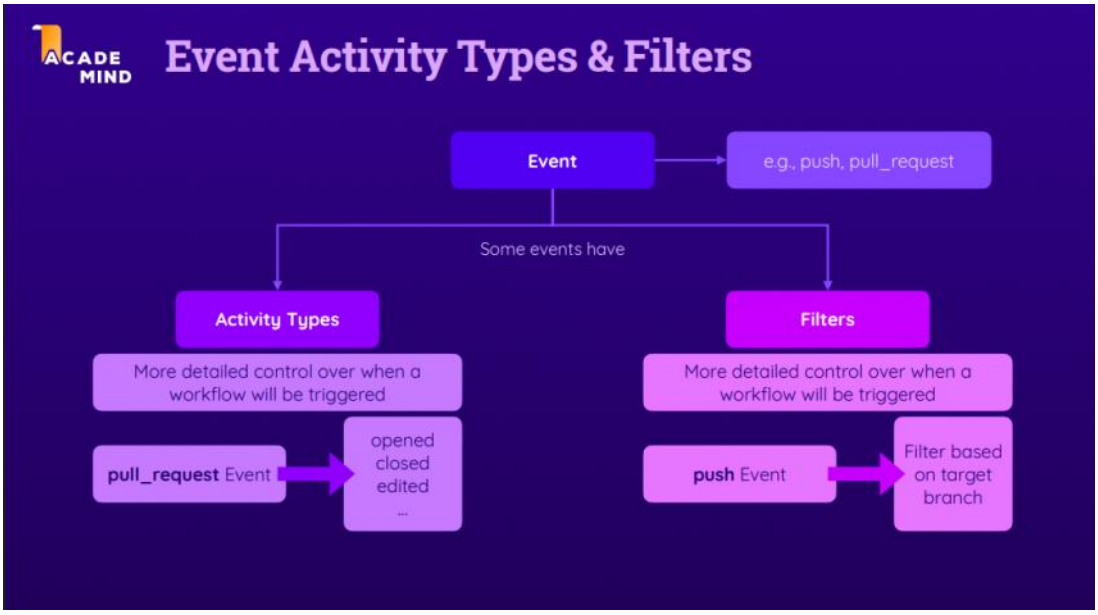
Example Workflow

```yaml
name: Events Demo 1
on: [push, workflow_dispatch]
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Output event data
        run: echo "${{ toJSON(github.event) }}"
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Test code
        run: npm run test
      - name: Build code
        run: npm run build
      - name: Deploy project
        run: echo "Deploying..."
```

Event Filters and Activity Types in GitHub Actions

Need for More Control

- Sometimes, you want a workflow to be triggered by an event (like a push), but **not for all branches**—for example, only for pushes to the main branch.
- Simply specifying push or pull_request as an event is not enough for such selective control.



Activity Types

- Some events provide **activity types**, which let you specify exactly which variation of the event should trigger the workflow.
- Example:
  For the pull_request event, activity types let you choose if the workflow should trigger only when a pull request is **opened**, **closed**, or **edited**.
  You do not have to trigger on every kind of pull request activity—just the ones you need.

Event Filters

- Some events (like push and pull_request) support **event filters**.
- **Event filters** give you even more control—such as filtering events based on:
  - Branches (e.g., only trigger on pushes to main)
  - Specific paths or files
- When filters are set, only the events matching those conditions will trigger the workflow.

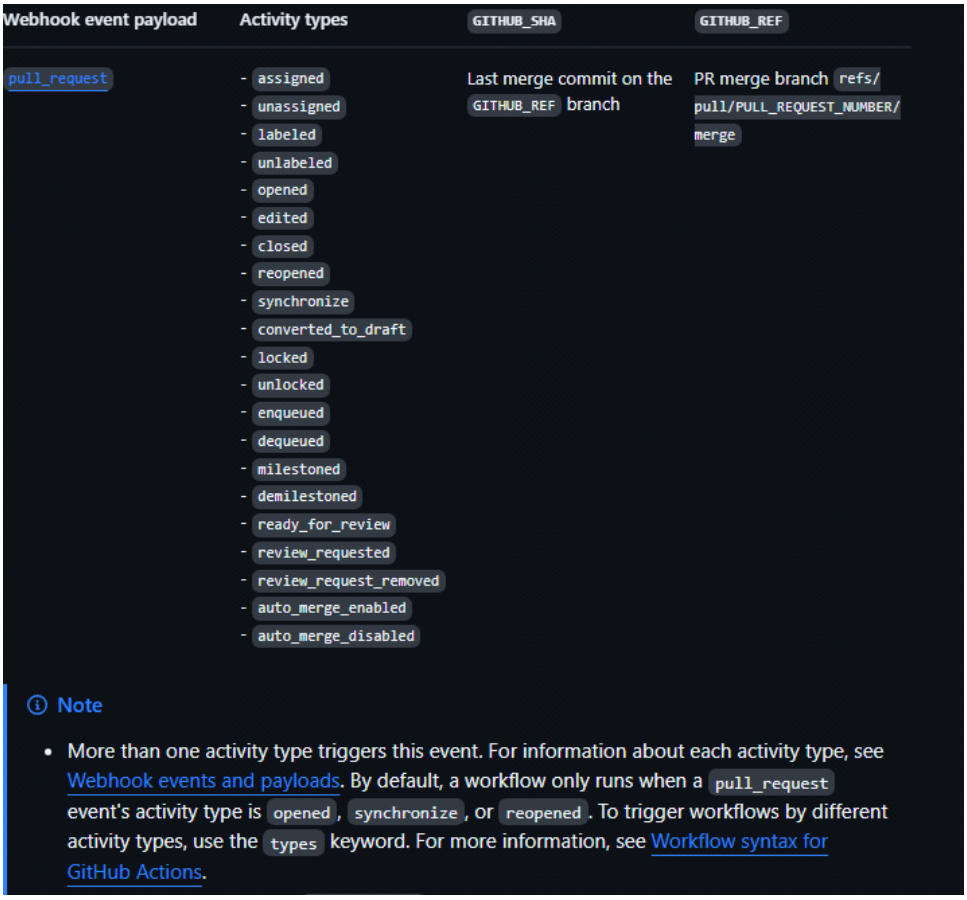
Summary

- **Activity types** and **event filters** are essential features in GitHub Actions.
- They allow you to define **precisely** which events (and under which exact circumstances) should trigger your workflows.
- This enables more reliable and targeted automation in your CI/CD process.

Activity Types and Event Filters in GitHub Actions

Understanding Activity Types

- Activity types let you specify which sub-type or variation of an event should trigger your workflow.
- The official documentation for events shows a column with all supported activity types for each event.
- **Example:**
  - The push event has no activity type variations.
  - The pull_request event has many activity types, like opened, edited, closed, etc.
- By default, if you add only pull_request without any types, the workflow will run only for **opened**, **synchronize**, or **reopened**.
  It will **not** run for closed unless specified.
- Always check the documentation to know which activity types exist for each event and what the defaults are.



Webhook event payload	Activity types	GITHUB_SHA	GITHUB_REF
pull_request	- assigned - unassigned - labeled - unlabeled - opened - edited - closed - reopened - synchronize - converted_to_draft - locked - unlocked - enqueued - dequeued - milestoned - demilestoned - ready_for_review - review_requested - review_request_removed - auto_merge_enabled - auto_merge_disabled	Last merge commit on the GITHUB_REF branch	PR merge branch refs/pull/PULL_REQUEST_NUMBER/merge

How to Use Activity Types in YAML

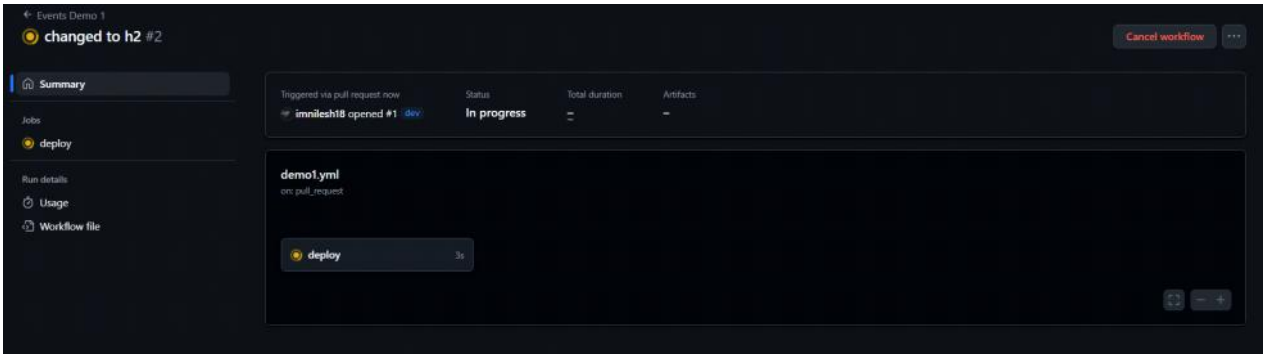
- Instead of listing events in a simple array, expand the event under the on key and use types: with an indented list.
- Example to trigger only on pull_request being opened:

```yaml
name: Events Demo 1
on:
  pull_request:
    types:
      - opened
  workflow_dispatch:
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Output event data
        run: echo "${{ toJSON(github.event) }}"
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Test code
        run: npm run test
      - name: Build code
        run: npm run build
      - name: Deploy project
        run: echo "Deploying..."
```

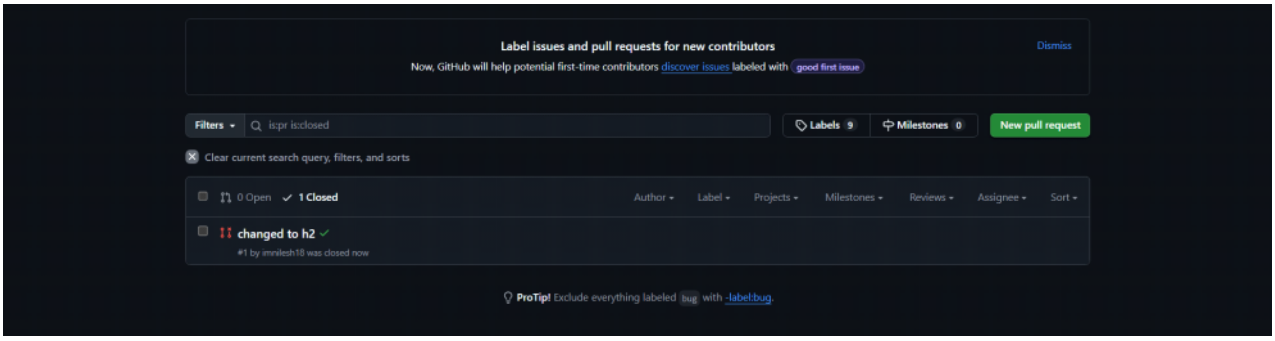
- If you add multiple activity types, you can list them (like opened, edited, etc.).
- If you want to include a simple event like workflow_dispatch at the same time, define it at the same indentation level, with a colon at the end (workflow_dispatch:), even if no other properties are set.

How They Work in Practice

- If your workflow listens to pull_request with types: opened, the workflow only triggers when a pull request is **opened**.



- **Closing** or **editing** a pull request will not trigger the workflow unless those types are also included.



- If you create a pull request from a new branch:
  - The workflow runs when the pull request is opened (if opened type is specified).
  - It does not run if you simply close the pull request, unless you also list the closed type.
- By default (when no types are given), most events (like pull_request) trigger a workflow only for a subset of types (opened, synchronize, reopened). If you want the workflow to run for additional types (e.g., closed), you must specify those types explicitly.

Summary

- Activity types allow fine-grained control over when workflows run.
- For each event (like pull_request or issues), check the documentation for supported activity types and defaults.
- Specify desired activity types in the workflow yaml under the event using the types: key.
- This approach gives you clear and predictable workflow triggering behavior, avoiding unnecessary or unexpected runs.

Event Filters in GitHub Actions

What are Event Filters?

- Event filters give you fine control over **when** a workflow runs for a given event.
- By default, events like push trigger the workflow for **any branch**.
- For tasks like deployments, you often only want the workflow to trigger for specific branches, such as main.

Supported Events

- Event filters are supported for a selected list of events:
  - push
  - pull_request

- pull_request_target
- workflow_call

Types of Event Filters

- **branches:**
  Only trigger the workflow if a push (or PR) targets specific branches.
- **branches-ignore:**
  Ignore triggers for specific branches.
- **tags:**
  Trigger only when a specified tag is pushed.
- **tags-ignore:**
  Ignore triggers for certain tags.
- **paths:**
  Trigger only if changes are made to specified files or directories.
- **paths-ignore:**
  Ignore triggers if only changes are made to specified files or directories.

Usage Examples

- **branches filter:**
  Trigger on pushes to master, any branch that starts with dev-, or any branch that starts with feat/.

```
push:
  branches:
    - master          # e.g., master
    - 'dev-*'         # e.g., dev-new, dev-this-is-new
    - 'feat/**'       # e.g., feat/new, feat/new/button
```

- **branches-ignore filter:**
  Ignore pushes to certain branches (useful for excluding, e.g., development or test branches).
- **paths filter:**
  Only trigger when files within a path are modified.

```
push:
  branches:
    - main
  paths:
    - 'src/**'
```

- **paths-ignore filter:**
  Prevent running the workflow if only specified paths changed (e.g., changes are only in workflow definitions).

```
push:
  branches:
    - main
  paths-ignore:
    - '.github/workflows/*'
```

  This configures the workflow to only run for pushes to main, **except** if changes are solely in the .github/workflows/ folder[1] [6].

Wildcards and Patterns

- * matches any string except slashes (/).
- ** matches any string including slashes (so it works with nested folders).
- Examples:
  - 'dev-*' matches dev-new, dev-this-is-new.
  - 'feat/**' matches feat/new, feat/new/button, etc.

Combined Example

```yaml
name: Events Demo 1
on:
  pull_request:
    types:
```

```
      - opened
    branches:
      - master # master
      - 'dev-*' # dev-new dev-this-is-new
      - 'feat/**' #feat/new feat/new/button
  workflow_dispatch:
  push:
    branches:
      - master # master
      - 'dev-*' # dev-new dev-this-is-new
      - 'feat/**' #feat/new feat/new/button
      # developer - 1
    paths-ignore:
      - '.github/workflows/*'
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Output event data
        run: echo "${{ toJSON(github.event) }}"
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Test code
        run: npm run test
      - name: Build code
        run: npm run build
      - name: Deploy project
        run: echo "Deploying..."
```

- In this example, the workflow runs for:
  - Pull requests opened to master, any dev-* branch, or any branch under feat/
  - Manual triggers with workflow_dispatch
  - Pushes to the same set of branches, unless only files in .github/workflows/ are changed

Key Points

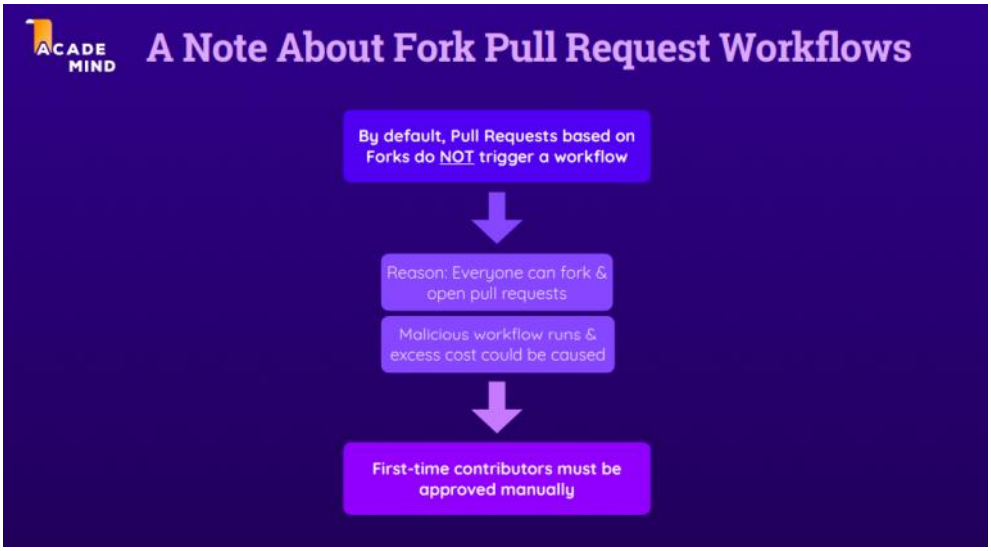
- Event filters control exactly **which branches, tags, or file paths** will cause a workflow to run.
- Use these filters to avoid unnecessary workflow runs and to protect critical workflows like deployments from running on the wrong branches.
- Combine filters and wildcards for advanced use cases and efficient automation[1] [6].

Pull Request Workflows and First-Time Contributors from Forks

Key Points

- **Activity types** allow you to control which exact sub-event (e.g., opening, editing, or closing a pull request) will trigger a workflow.
- **Event filters** (such as branches and paths) let you define which branches or file paths must be targeted/changed for events like push or pull_request to activate a workflow.

Special Case: Pull Requests from Forked Repositories



- Anyone can **fork a public repository** and open pull requests to the original repository.
- **Behavior to be aware of:**
  If your workflow is configured to run when a pull request is opened and targets certain branches, you might expect it to be triggered for pull requests from any user—including those from forks.
- **What happens:**
  - When a pull request is opened from a fork (by a third-party contributor), the workflow does *not* run automatically, even if it technically matches the filters and activity types.
  - Instead, you see an item with an exclamation mark in the Actions page—this means the workflow is *pending approval*.
  - You, as the repository owner, must first manually approve running the workflow for this first-time contributor.
- **Why?**
  - By default, pull request workflows based on forks do not trigger automatically for security and abuse prevention reasons.
  - Anyone can fork and create pull requests to public repositories, so this mechanism protects your repository from potential abuse (such as malicious or spammy workflows).
- **Cost and Security Considerations:**
  - Running workflows for all incoming pull requests could lead to excessive, possibly malicious workflow runs—which could deplete free usage limits or create costs on paid plans.
  - It also protects against people trying to run harmful code or spam your account with workflow executions.

Approval Process

- **First-time contributors** opening pull requests from a fork require manual approval before workflows will run.
- After you, as the maintainer, manually approve the workflow for their pull request:
  - Subsequent pull requests from the same contributor's fork will automatically trigger workflows.
- **Collaborators** (those added directly to the repo) are trusted and do not require this manual approval. Their pull requests will trigger workflows automatically.

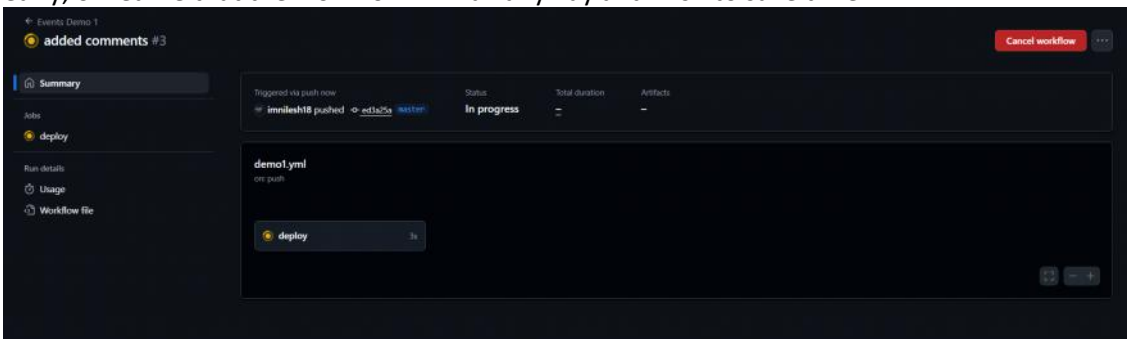
Important Summary

- Pull requests from forks by new contributors do NOT immediately trigger workflows.
- As the repository owner, you must **manually approve** first-time contributor workflow runs.
- This protects your repository from unwanted workflow abuse, spam, and potential costs.
- After the first manual approval, future contributions from that user's fork will run workflows automatically.

Canceling and Skipping Workflow Runs in GitHub Actions

Canceling Workflow Runs

- **By default:** Workflows get canceled automatically if jobs fail.
  - If a job fails, the entire workflow is canceled (unless you add custom rules to allow subsequent jobs to continue even after earlier failures).
- **A job fails** if at least one step fails (this default can also be changed later).
- **Manual cancelation:** You can manually cancel a workflow run from the GitHub Actions UI.
  - Click on the running workflow and select the **Cancel workflow** button.
  - This is useful if you discover an error right after pushing, want to halt a long-running or faulty workflow early, or realize that the workflow will fail anyway and wish to save time.



Skipping Workflow Runs

- **By default:** All matching events (like a push to a branch) will trigger their corresponding workflow.
- **Skipping a workflow:** You can prevent a workflow from running (even if the trigger event occurs) by including special keywords in your commit message or pull request message.
  - For push and pull request events, GitHub recognizes specific skip keywords.
- **How to skip:**
  - Add **[skip ci]**, **[skip actions]**, or similar keywords (see GitHub's documentation for a full list) to the commit message.
  - For example: **git commit -m "added comments [skip ci]"**
- When you push such a commit:
  - The push will **not trigger the workflow**, even if it normally would.

- This is helpful when making non-code changes (like adding comments) and you want to avoid unnecessary workflow runs.

Example Workflow

- Add a comment to your code and commit the change:
  // My tests
  // Add yet another comment
- When committing, use the skip keyword:
  git commit -m "added comments [skip ci]"
- Push to the main branch.
- No new workflow is triggered, as the skip instruction is recognized.

Summary:
You can cancel ongoing workflows manually from the UI, and you can skip workflow runs by using special keywords in commit or pull request messages. This helps you control when workflows run, saving time and resources for changes that don't need automation.

Section Summary: Workflow Events, Activity Types, Approval, Cancellation, Skipping, and Event Filters

- There is a broad variety of events that can be used to trigger workflows.
  - Most are **repository related** (like pushing, pull requests, or issues).
  - There are also more general events (like scheduling workflows or manually triggering workflows).
- When specifying events:
  - **Many events support activity types**, letting you narrow down which specific event types should trigger a workflow.
    - Example: Configure a workflow to run when a pull request is **opened** or **edited**, or when it's **closed**, according to your needs.
- For **pull requests**:
  - If a pull request comes from a **forked repository**, it will not trigger workflows by default.
    - You must manually approve workflows for unknown contributors before the workflow will run.
    - After approval, future workflows from that contributor will run automatically.
- Once a workflow is running, it can be **canceled**:

    - **Automatically** (for example, when a job or step fails).
    - **Manually** (if you anticipate failure or for any other reason).
- If you want to intentionally skip a workflow before it runs, you can add a **special instruction** to your commit message: **git commit -m "added comments [skip ci]"**
    - This will skip the workflow run that would have started.
- **Event filters** can also be used in your event definitions:
    - These are related to activity types but come with a different purpose.
    - Instead of selecting which version of an event triggers a workflow, filters let you restrict workflow runs to cases where certain conditions are met.
    - **Event filters exist for push and pull request events**.
        - You can filter based on the targeted branch or on affected files.
        - This helps you control exactly when your workflows are executed.

Job Artifacts & Outputs

04 August 2025 02:07

Working with Data and Outputs in GitHub Actions

Focus of This Section

- This course section dives deeper into GitHub Actions, particularly on **handling data within jobs and steps** as well as **producing and using outputs**.

Key Topics Covered

- **Artifacts:**
  - Introduction to what artifacts are.
  - How to work with artifacts in your workflows.
- **Job Outputs:**
  - Understanding how job outputs work.
  - Differences between artifacts and job outputs.
- **Dependency Caching:**
  - Exploring the concept and importance of caching dependencies.
  - Why dependency caching is a key skill when working with GitHub Actions.

You will learn in detail:

- How to manage, create, and use artifacts in your workflows.
- How to produce outputs from jobs and steps, and how these outputs differ from artifacts.
- How to cache dependencies efficiently for faster and more reliable workflow runs.

Setting Up the Example Project and Repository for This Section

Project Setup

- The example project for this section is the same as used before but now includes an example workflow in the .github/workflows folder.
- This workflow is already finished and set to be activated whenever a push occurs to the main branch.
- This workflow will be the one you enhance and work on throughout the section.

Git & Repository Initialization

- Initialize the project with Git locally to create a local repository.
- Create the initial commit.

GitHub Repository

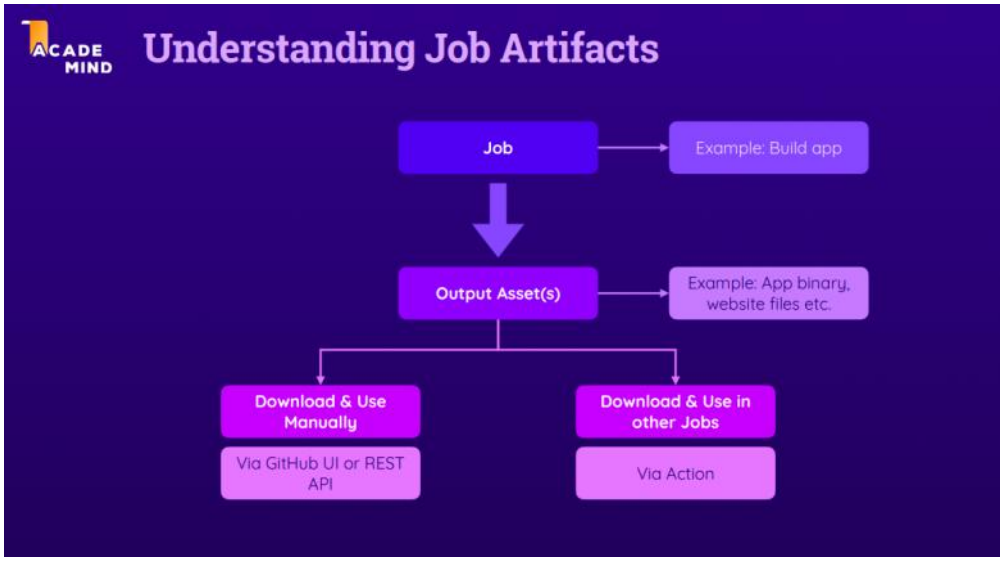
- Create a new repository on GitHub (example name: **gh-data**; you can choose any name, public or private).

- After creating the repository, copy the repository URL.
- Add the repository as a remote to your local setup with the name origin and include the username for authentication purposes.
- Push your main branch after establishing the remote link.

Result

- You now have a GitHub repository with the example workflow set up, and this repository will be used throughout this course section.

Understanding Job Artifacts in GitHub Actions



What Are Job Artifacts?

- **Job Artifacts** are the files and folders produced as outputs by a GitHub Actions job.
- Examples include:
  - Website files ready for deployment to a web hosting provider.
  - App package files for uploading to app stores.
  - Executable files for desktop applications.
  - Log files generated by testing jobs, or any other file output you wish to keep or examine.

Why Are Artifacts Important?

- Artifacts allow you to:
  - **Download the files manually** after a job finishes (as the repository owner, you might want to inspect these or use them elsewhere).
  - **Automatically share files between jobs** in the same workflow (e.g., a build job creates files and a subsequent deploy job uploads them to a server).

When Are Artifacts Used?

- Whenever a job generates outputs that:
  - Need to be deployed or distributed elsewhere.
  - Should be manually inspected.
  - Are required by other jobs later in the workflow.

Typical Use Cases

- Build jobs producing files for deployment.
- Test jobs producing logs for debugging.
- Any job whose outputs are needed by another job, or must be downloaded and used outside of GitHub Actions.

Summary

- **Artifacts** in GitHub Actions are files and folders that are the result of your jobs.
- You can use GitHub Actions features to store, download, and pass these artifacts along for manual use or for consumption by other jobs in your workflow, enabling flexible and powerful automation scenarios.

Workflow Structure and Storing Artifacts

Overview of the Demo Workflow

- **Workflow Name:** Deploy website (the name is not important).
- **Trigger:** Runs when code is pushed to the main branch.
- **Jobs:**
  1. **test**
     - Checks out the code.
     - Installs dependencies.
     - Runs lint and test scripts.
  2. **build**
     - Runs after the test job (needs: test).
     - Checks out the code again.
     - Installs dependencies.
     - Runs the build script (npm run build), which produces output files in a dist folder.
  3. **deploy**
     - Runs after the build job (needs: build).
     - Currently only outputs a dummy "Deploying..." message.

Understanding Artifacts in This Context

- The **build** job, after running npm run build, creates a dist folder.
- The dist folder contains the files that would be uploaded to a web server for hosting the website.
- After the workflow is finished, the runner machine (server) will be shut down and any produced files will be lost unless they are stored elsewhere.
- Therefore, it's important to **store the dist folder as an artifact** so you can:
  - Download it manually from the workflow run page.
  - Inspect, test, or upload these files to a web server.

Next Steps

- To store the contents of the dist folder as an artifact, **add a new step in the build job**.
- This step will be responsible for uploading the dist folder as an artifact.

Current code before artifact step:

```yaml
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
  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Deploy
        run: echo "Deploying..."
```

- The next step is to add an artifact upload step after the build step in the build job, so the dist folder is saved and accessible after the workflow finishes.

Storing Artifacts with the Upload Artifact Action

Adding an Upload Step for Artifacts

- If a job in your workflow produces output you want to store (artifacts), add a new step to upload them.
- The step can be named anything (here: **Upload artifacts**).
- Use the official **Upload Artifact** action: actions/upload-artifact.

Configuration

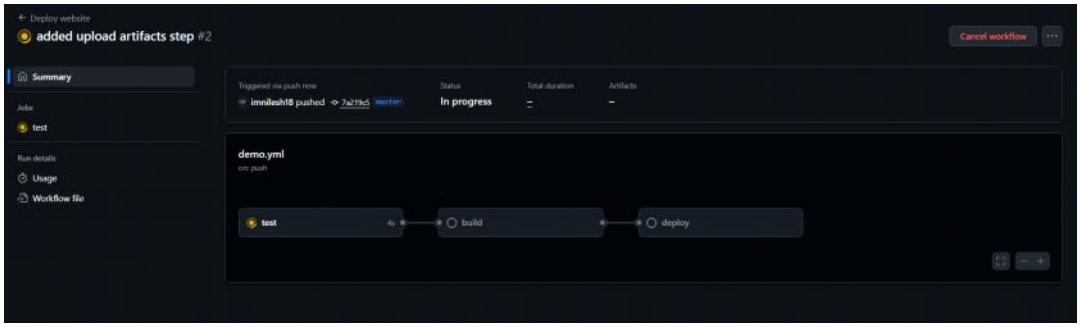
- Specify the artifact **name** so you can retrieve it later (here: dist-files).
- Specify the **path** to the files/folder to upload as the artifact (here: the dist folder).
- You can specify multiple paths using the pipe | and list each path on a new line (e.g., "dist", "package.json"), though usually storing the build output folder alone is enough.

Example Workflow Snippet

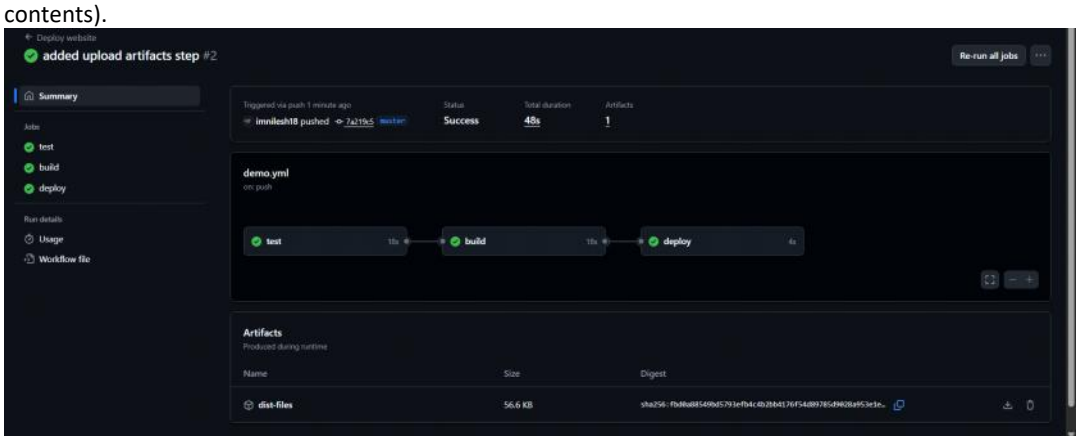
```yaml
name: Deploy website
on:
  push:
    branches:
      - master
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Lint code
        run: npm run lint
      - name: Test 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 artifacts
        uses: actions/upload-artifact@v4
        with:
          name: dist-files
          path: dist
          # path: |
          #   dist
          #   package.json
  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Deploy
        run: echo "Deploying..."
```

How It Works

- When the workflow runs and finishes, a downloadable artifact (dist-files) is available on the workflow run page in GitHub Actions.



- You can click to download this artifact (as a ZIP file) and access all the included files (e.g., the dist folder and its contents).



- The artifact is available because of the actions/upload-artifact action step.

Notes

- Only add the paths you actually want as artifacts (here, just the dist folder is needed). Multiple paths can be listed if required.
- Artifacts can be downloaded for manual inspection or later steps in your workflow.

Using Uploaded Artifacts in a Subsequent Job

Overview

- Artifacts uploaded in one job (like build) can be used in another job (like deploy).
- Each job runs on its own runner (machine), so files from one job are not automatically available in another.
- To use artifacts in a following job, you need to **download them explicitly**.

How to Download Artifacts in a Job

- Add a new step in the job where you want to use the previously uploaded artifacts.
- Use the official **download-artifact** action (actions/download-artifact).
- Specify the same artifact name you used for uploading (e.g., dist-files).

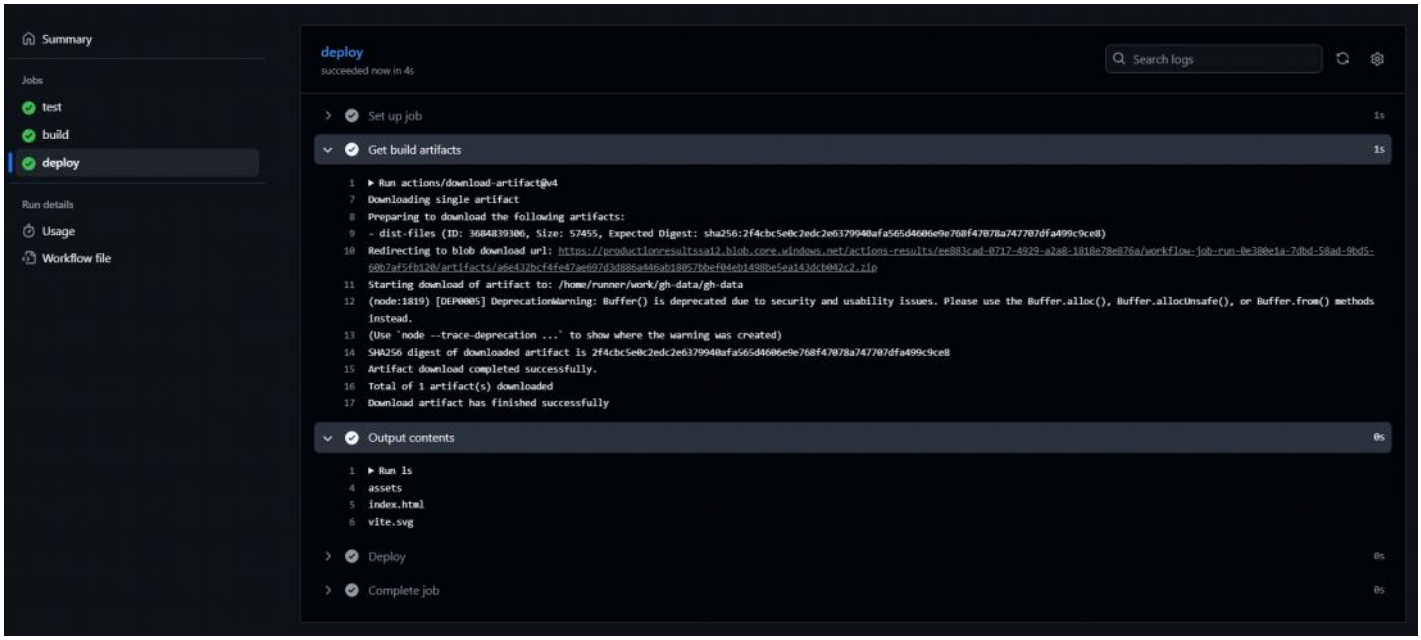
Structure and File Extraction

- When the artifact is downloaded and unpacked in the new job, the contents of the original dist folder are extracted directly into the working folder of the job—**not inside a new dist folder**.
- This means all files that were in dist are now available at the top level of the working directory in the deploy job.

Example Workflow

```yaml
name: Deploy website
on:
  push:
    branches:
      - master
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
```

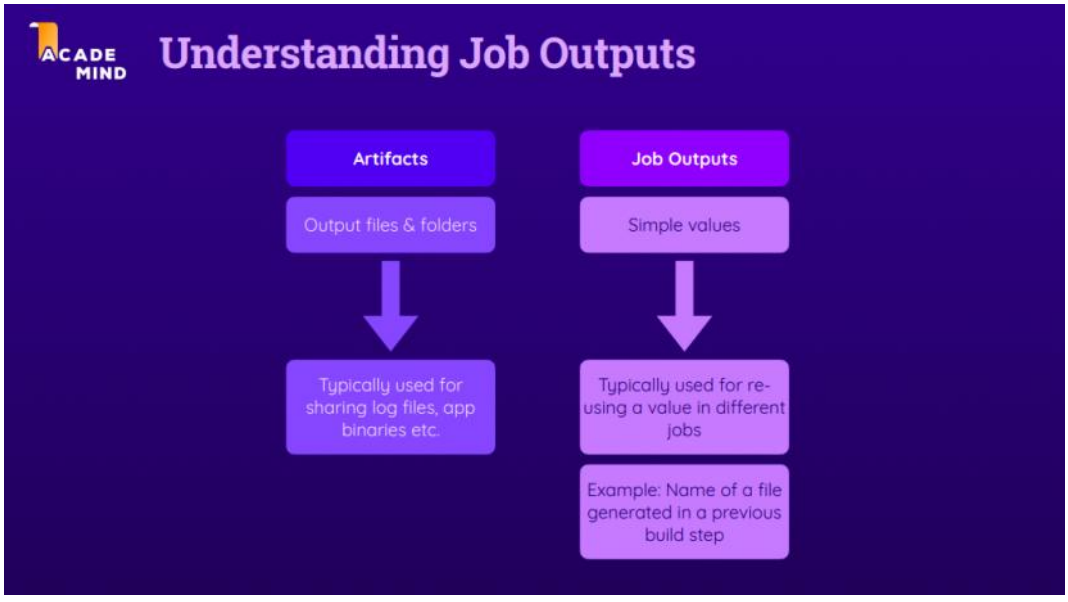
```yaml
    - name: Get code
      uses: actions/checkout@v3
    - name: Install dependencies
      run: npm ci
    - name: Lint code
      run: npm run lint
    - name: Test code
      run: npm run test
build:
  needs: test
  runs-on: ubuntu-latest
  steps:
    - name: Get code
      uses: actions/checkout@v3
    - name: Install dependencies
      run: npm ci
    - name: Build website
      run: npm run build
    - name: Upload artifacts
      uses: actions/upload-artifact@v4
      with:
        name: dist-files
        path: dist
        # path: |
        #   dist
        #   package.json
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..."
```



Key Points

- Use actions/download-artifact to get files produced from a previous job.
- The artifact's files are directly available in the job's working directory.
- This setup allows you to use generated content (like a production build) for tasks like deployment, further processing, or in spection.

Understanding Artifacts vs. Job Outputs in GitHub Actions



Artifacts

- Artifacts are **output files and folders** generated by jobs.
- They are typically used for **sharing**:
  - Log files
  - App binaries
  - Any file/folder output you want to pass on or inspect.
- Artifacts are used to transfer files between jobs or download files manually after workflow runs.

Job Outputs

- Job outputs are **simple values** produced by jobs.
- They are typically used for **reusing a value across jobs**.
- Example use cases:
  - Name of a file generated in a previous build step that is needed in a later job.
  - Dates, hashes, random values, or any variable/result from one job needed in another job.
- Unlike artifacts, job outputs do not transfer actual files—only single values (strings, numbers, etc.).

Key Difference

- **Artifacts:** Actual files and folders (e.g., log files, binaries, build outputs).
- **Job Outputs:** Simple values that can be referenced in later jobs (e.g., file names, IDs, computed values).

Setting and Sharing Job Outputs

Scenario

- In the **build** job, after producing the dist folder and uploading it as an artifact, you also want to capture the name of the generated JavaScript file inside the dist/assets folder.
- The JS file name is not fixed; it can be generated randomly.
- The goal is to make this JS file name available for use in the **deploy** job.

Why Job Outputs?

- Job outputs allow one job to pass **simple values** (like the generated file name) to other jobs.
- Unlike artifacts (which transfer files/folders), job outputs are for sharing key data (strings, numbers, etc.).

How to Set a Job Output

1. **Define Outputs for the Job**
   Add an outputs key at the top level of the job, before defining steps.
   ```
   outputs:
         script-file: ${{ steps.publish.outputs.script-file }}
   ```
2. Here, script-file is the output variable for the job.
3. **Set Output in a Step**
   - Add a step that finds the filename and sets it as an output.
   - Give the step an id (e.g., publish).
   - Use a command like this to echo and write the output:
     ```
     - name: Publish JS filename
       id: publish
       run: find dist/assets/*.js -type f -execdir echo 'script-file={}' >> $GITHUB_OUTPUT ';'
     ```
   - This writes the name of the JavaScript file to the special output file managed by GitHub Actions.
4. **Reference the Step's Output in the Job Output**
   - In the job's outputs, use ${{ steps.publish.outputs.script-file }} to link the step's output to the job's output.
5. **Access the Output in Another Job**
   - The deploy job (or any job needing this value) can now use ${{ needs.build.outputs.script-file }} to get the file name.

Example Workflow

```
name: Deploy website
on:
  push:
    branches:
      - master
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:
      script-file: ${{ steps.publish.outputs.script-file }}
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Build website
        run: npm run build
      - name: Publish JS filename
        id: publish
        run: find dist/assets/*.js -type f -execdir echo 'script-file={}' >> $GITHUB_OUTPUT ';'
      - name: Upload artifacts
        uses: actions/upload-artifact@v4
        with:
          name: dist-files
          path: dist
          # path: |
          #   dist
          #   package.json
  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..."
```

Key Points

- Use job outputs for sharing values between jobs.
- Set step outputs using the syntax with >> $GITHUB_OUTPUT.
- Reference the step output in the job's output declaration.
- Access the job output in subsequent jobs using the needs context.

Accessing and Using a Job Output in Another Job

How to Use a Job Output in a Dependent Job

- To use an output from the build job in the deploy job, add a new step in the deploy job.
- Give the step any name (e.g., **Output filename**).
- Use the echo command to print the output.
- Access the output value from the build job using the expression ${{ needs.build.outputs.script-file }}.

Example

```
    - name: Output filename
            run: echo "${{ needs.build.outputs.script-file }}"
```

- Here, needs.build.outputs.script-file points to the output named script-file published by the build job.

Usage

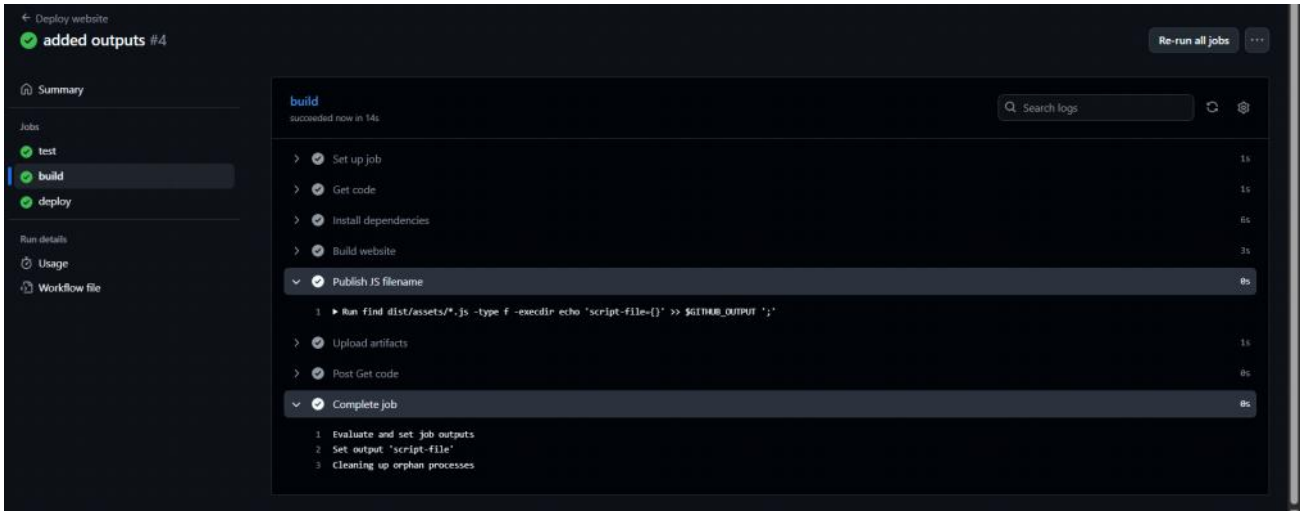
- The output data (such as a dynamically generated file name) is now available for use in this job.
- This technique is helpful for passing simple values from one job to another, enabling flexible and dynamic workflows.

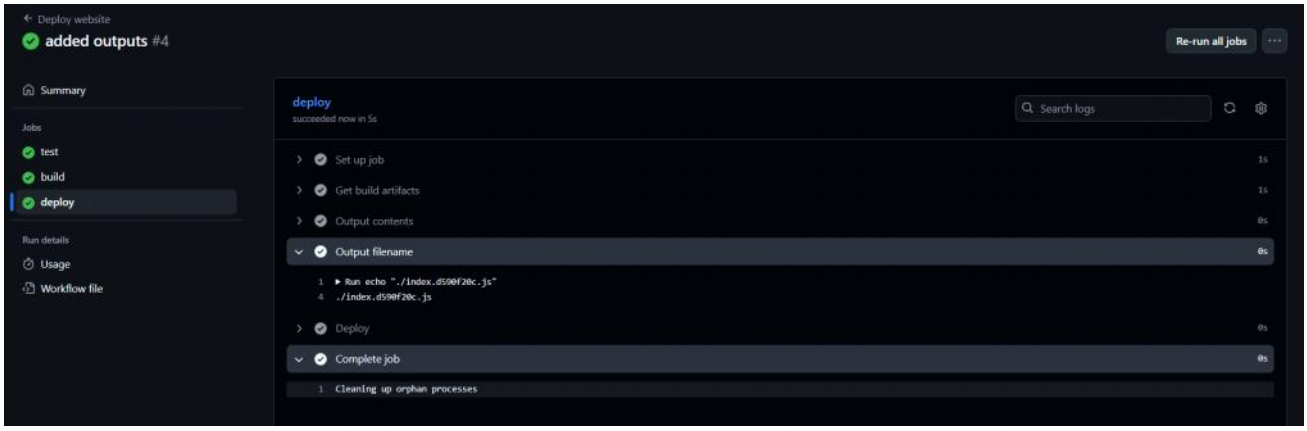
Full Context

```
    name: Deploy website
    on:
      push:
        branches:
          - master
    jobs:
      test:
        runs-on: ubuntu-latest
        steps:
```

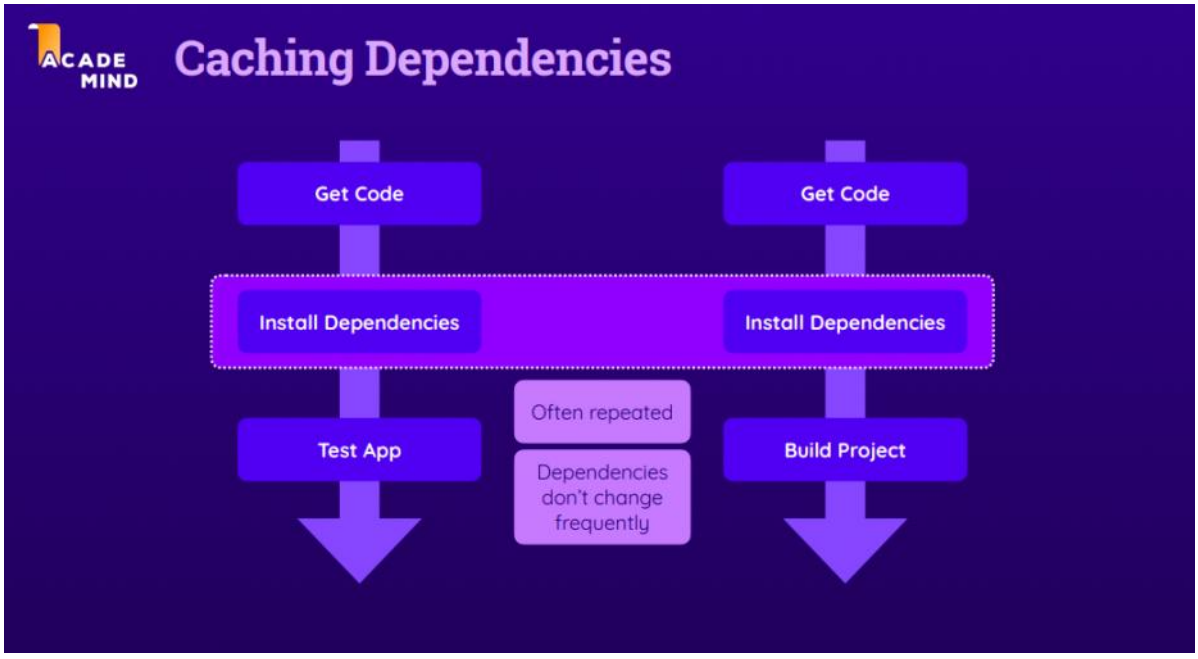
```
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Lint code
        run: npm run lint
      - name: Test code
        run: npm run test
  build:
    needs: test
    runs-on: ubuntu-latest
    outputs:
      script-file: ${{ steps.publish.outputs.script-file }}
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Install dependencies
        run: npm ci
      - name: Build website
        run: npm run build
      - name: Publish JS filename
        id: publish
        run: find dist/assets/*.js -type f -execdir echo 'script-file={}' >> $GITHUB_OUTPUT ';'
      - name: Upload artifacts
        uses: actions/upload-artifact@v4
        with:
          name: dist-files
          path: dist
          # path: |
          #   dist
          #   package.json
  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Get build artifacts
        uses: actions/download-artifact@v4
        with:
          name: dist-files
      - name: Output contents
        run: ls
      - name: Output filename
        run: echo "${{ needs.build.outputs.script-file }}"
      - name: Deploy
        run: echo "Deploying..."
```

- When this workflow runs, the deploy job will successfully output the file name provided by the build job as a job output.





Caching Dependencies



- Workflows take about a minute to execute because jobs run sequentially, with additional setup and cleanup time.
- Running jobs in parallel is not an option when jobs depend on each other.
- Some steps are always repeated, particularly "Get code" and "Install dependencies".
- Each job requires checking out code again, since every job runs on a separate runner.
- The "Get code" step is fast (about one second), but "Install dependencies" is slower (taking 8–12 seconds depending on the job).
- Installing dependencies is the longest step, so reducing its duration could save significant workflow execution time.
- Even without cost concerns (e.g., on a free plan), speeding up workflows means faster deployment.
- Dependency installation is common in most CI/CD workflows, and dependencies don't change frequently.
- Caching dependencies is an effective way to speed up the installation process.
- Caching allows dependencies to be installed once in a job and then reused by other jobs.
- Cached data can also be reused across workflow runs, not just within a single workflow.
- GitHub Actions provides an official cache action for caching files and folders across jobs and workflows.

1. https://docs.github.com/actions/reference/workflow-syntax-for-github-actions
2. https://github.com/actions/cache
3. https://docs.github.com/en/actions/reference/workflows-and-actions/dependency-caching

Caching Dependencies in GitHub Actions

Key Points

- Workflows often take about a minute to execute because jobs run sequentially and each runs setup/cleanup work.

- Jobs cannot run in parallel if they depend on each other.
- Steps like "Get code" and "Install dependencies" are repeated in each job.
- Each job uses its own runner, so getting code is always required, but it's fast (about one second).
- **Installing dependencies is slower** (8–12 seconds per job) and is typically the longest part of the workflow.
- Reducing dependency installation time will significantly reduce total workflow duration.
- Even on a free plan (no cost concern), faster execution means quicker deployment.
- Installing dependencies is commonly repeated in CI/CD, though dependencies change infrequently.
- **Caching dependencies** avoids redundant installations and saves time.
- Using dependency cache:
  - Cache the dependencies after installation in the first job.
  - Reuse the cache in later jobs and workflow runs, unless dependencies change.
- Add the **caching step before installing dependencies** in each job where you want to leverage the cache.

How to Configure Caching

- Use the official GitHub Actions cache action: actions/cache@v3.
- Specify the path to cache (e.g., ~/.npm for npm dependencies).
- Use a key based on the contents of your lock file (e.g., package-lock.json); the key changes whenever dependencies change, ensuring cache is properly invalidated.

Example Step

```
    - name: Cache dependencies
            uses: actions/cache@v3
            with:
              path: ~/.npm
              key: deps-node-modules-${{ hashFiles('**/package.lock.json') }}
```

- This step restores the cache if available and saves a new cache after dependencies are installed.

How It Works

- First run: No cache, so dependencies are installed fully and then cached.
- Subsequent runs: Cache is restored, making installing dependencies much faster.
- If dependencies change (package-lock.json changes): The key changes, stale cache isn't used, and a new cache is created.

Best Practices

- Only cache what you need to keep caches small.
- Use the hash of lock files in the key to invalidate cache when dependencies are updated.
- Periodically update cache keys.
- Do not cache sensitive data.
- Be aware of cache size limits and possible automatic pruning.

Benefits

- Shorter workflow duration.
- Faster feedback cycles and deployment.
- More efficient CI/CD pipelines.
- Cost savings when using paid GitHub Actions minutes.

Summary:

Caching dependencies in GitHub Actions workflows is a powerful optimization for reducing total execution time and boosting efficiency. By restoring cached dependencies on subsequent runs and updating the cache only when dependencies change, you can minimize redundant work and speed up your CI/CD pipelines.

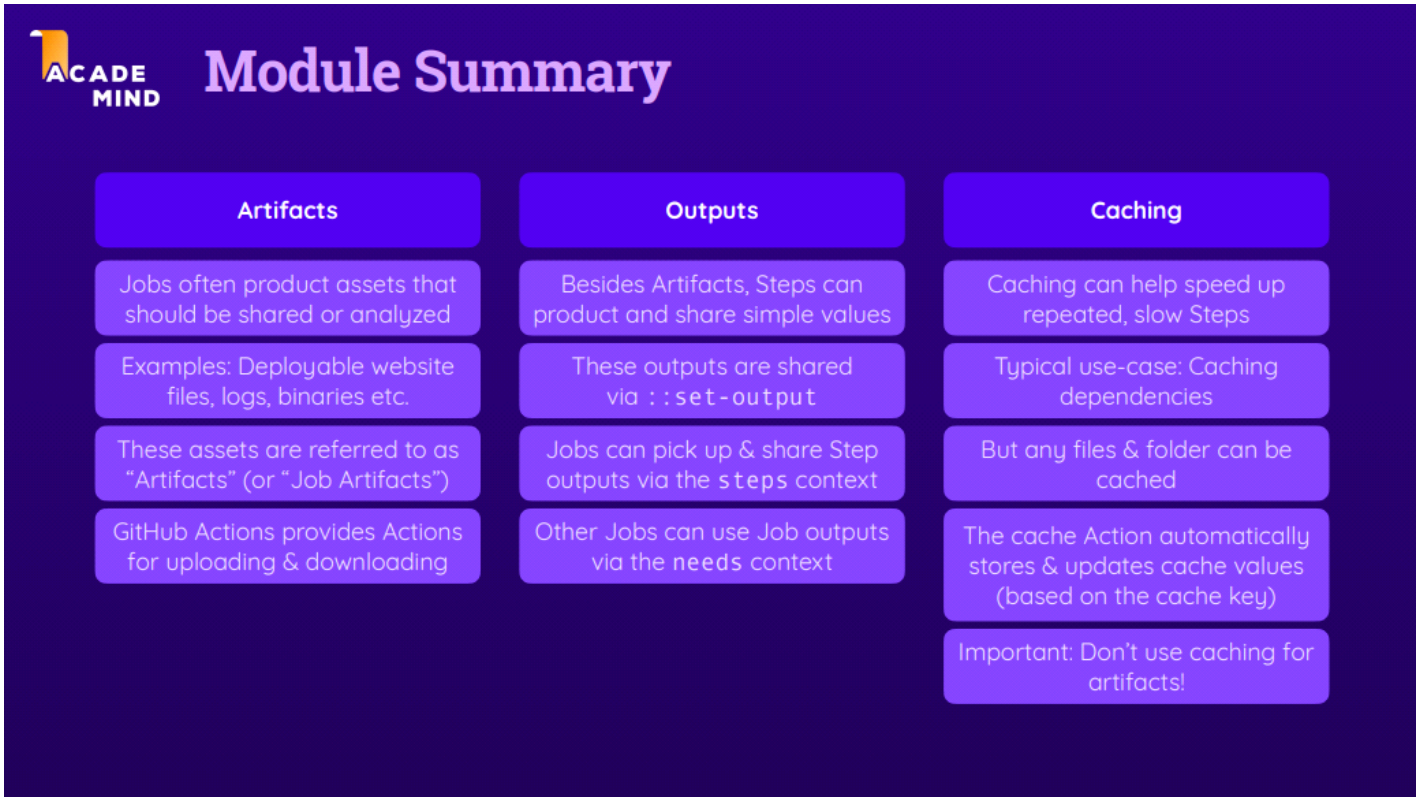
```
name: Deploy website
on:
  push:
    branches:
      - master
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Cache dependencies
        uses: actions/cache@v3
        with:
          path: ~/.npm
```

```
        key: deps-node-modules-${{ hashFiles('**/package.lock.json') }}
    - name: Install dependencies
      run: npm ci
    - name: Lint code
      run: npm run lint
    - name: Test code
      run: npm run test
build:
  needs: test
  runs-on: ubuntu-latest
  outputs:
    script-file: ${{ steps.publish.outputs.script-file }}
  steps:
    - name: Get code
      uses: actions/checkout@v3
    - name: Cache dependencies
      uses: actions/cache@v3
      with:
        path: ~/.npm
        key: deps-node-modules-${{ hashFiles('**/package.lock.json') }}
    - name: Install dependencies
      run: npm ci
    - name: Build website
      run: npm run build
    - name: Publish JS filename
      id: publish
      run: find dist/assets/*.js -type f -execdir echo 'script-file={}' >> $GITHUB_OUTPUT ';'
    - name: Upload artifacts
      uses: actions/upload-artifact@v4
      with:
        name: dist-files
        path: dist
        # path: |
        #   dist
        #   package.json
deploy:
  needs: build
  runs-on: ubuntu-latest
  steps:
    - name: Get build artifacts
      uses: actions/download-artifact@v4
      with:
        name: dist-files
    - name: Output contents
      run: ls
    - name: Output filename
      run: echo "${{ needs.build.outputs.script-file }}"
    - name: Deploy
      run: echo "Deploying..."
```

Notes: Caching Dependencies in GitHub Actions

- After adding caching, future workflow runs benefit from restored dependencies, making installation much faster when the cache is used.
- If you change code (for example, change an h3 to h2 and commit/push), the cache step restores previously saved dependencies, speeding up install dependencies in the workflow.
- On simple code changes (not affecting dependencies), dependency installation is fast due to a cache hit.
- If you update dependencies (for example, using npm update), the package-lock.json changes.
- When package-lock.json changes, the cache key changes, resulting in a cache miss: the cache is not used and dependencies are fully reinstalled.
- After installing the new dependencies, the cache is updated for future workflow runs.
- This system ensures the cache is only used when dependencies have not changed, automatically invalidating and regenerating if dependency files change.
- Behavior summary:
  - Initial workflow run (no cache): full install, then cache saved.
  - Subsequent runs (no change in dependencies): cache restored, install is fast.
  - After dependency update (package-lock.json change): cache miss, full install, new cache stored.
- Proper caching leads to significant time savings in GitHub Actions workflows.

Summary: Artifacts, Outputs, and Caching in GitHub Actions



- **Artifacts:**
  - Assets produced by jobs, such as files for deployment, logs, binaries.
  - Stored and shared using upload and download actions in GitHub Actions.
  - Can also be manually accessed or downloaded via the GitHub website or REST API.
- **Outputs:**
  - Simple values (not files) that need to be shared between steps or jobs.
  - Set in steps with the set output command using echo in a run step.
  - Job outputs are defined in the job using the outputs key and steps context.
  - Other jobs use these outputs via the needs context.
- **Caching:**
  - Used to speed up frequently repeated and slow steps such as installing dependencies.
  - Managed by the official cache action, which stores, restores, and updates cache automatically based on a key.
  - The cache key should reflect when the cache needs to be updated (e.g., when dependencies change).
  - **Artifacts and cache have different purposes:**

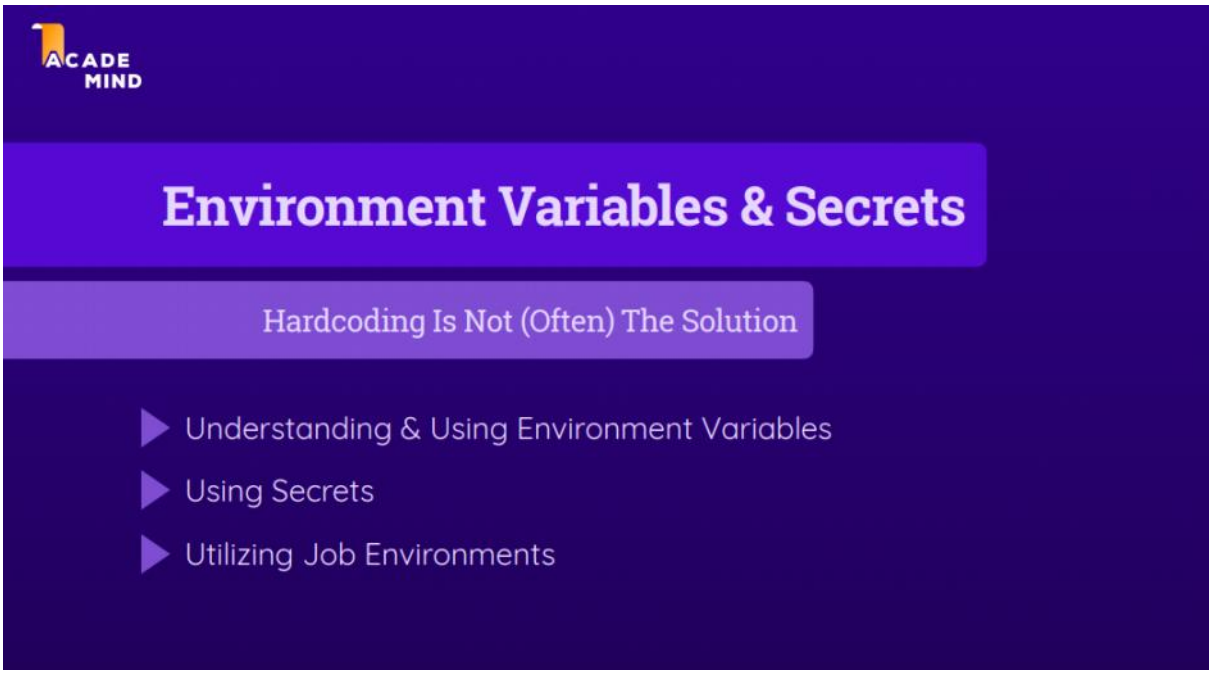
        - Artifacts are for output files produced by your workflows—the results you want to share, analyze, or deploy.
        - Cache is for speeding up workflow steps (like dependency installation), not for storing build artifacts.
    - **Do not use caching for your artifacts.**

In summary, use **artifacts** to store and share valuable output files between jobs or for later download, and use **cache** to speed up workflow execution by reusing files that don't change frequently, such as dependencies.

Using Environment Variables & Secrets

05 August 2025 01:51

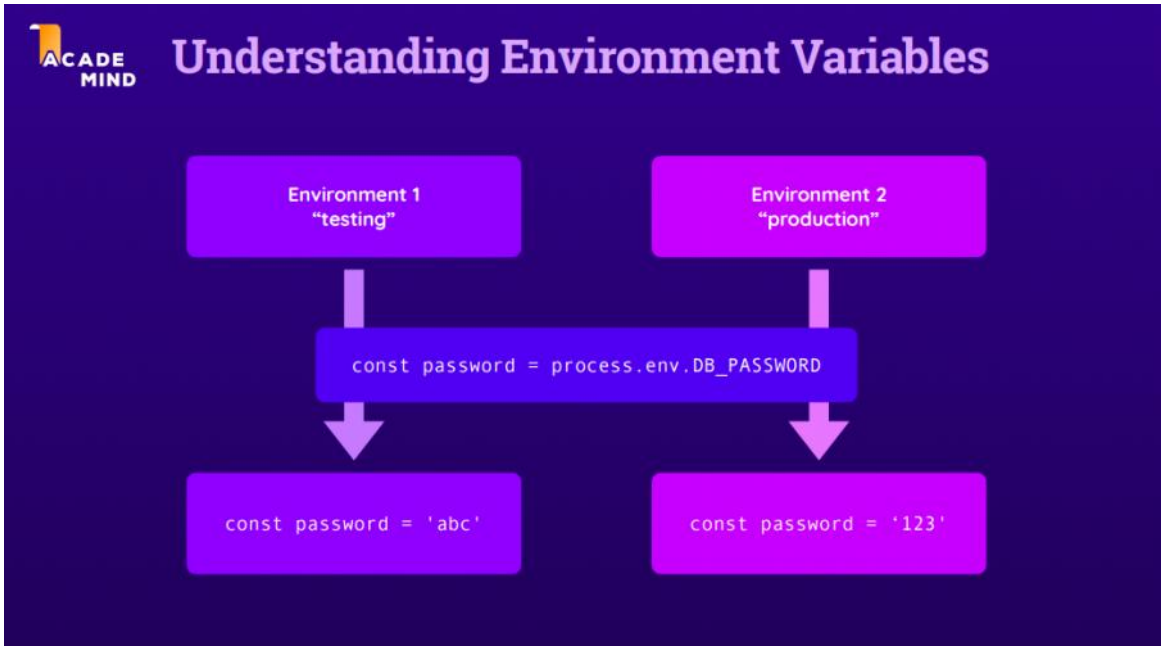
Using Environment Variables & Secrets in GitHub Actions



- This course section explores how to use **Environment Variables** and **Secrets** in GitHub Actions workflows.
- First, it covers what **Environment Variables** are, why they are needed, and how to use and work with them in workflows.
- Next, it explores the concept of **Secrets**—what they are and how they can be used.
- The section also dives into **GitHub Actions Environments**, explaining what problems this feature can solve and how it may be used in workflows.

Environment Variables, Secrets, and Environments in GitHub Actions

- For this section, a new demo project (REST API with Node and Express) is used.
- The project employs **environment variables** in the code, accessed via process.env (e.g., process.env.MONGODB_CLUSTER_ADDRESS, process.env.MONGODB_USERNAME, etc.).
- Environment variables provide dynamic values, such as database credentials, which might vary by environment:
    - Example: Testing and production environments needing different database addresses, usernames, or passwords.



  - This separation ensures tests use a separate database, avoiding accidental changes to production data.
- The use of environment variables is evident in files like database.js (for building connection URIs), app.js (for setting the server port), and the test config file (for test server configuration).
- The demo project also includes a GitHub Actions Workflow (not fully finished) that must start the server and run end-to-end tests. This setup requires **supplying environment variable values during workflow runs**.
- **Why use environment variables?**
  - Code often has values (credentials, ports, URLs) that differ between environments (dev, test, prod).
  - Avoids hardcoding sensitive or environment-specific data in code.
  - Promotes safer, more flexible deployments and testing.
- In summary:
  - **Environment variables** allow you to configure values dynamically per environment, supporting different setups for development, testing, and production.
  - This pattern is common in practice and essential to safely run tests, deployments, and integrate external services.
  - Supplying these variable values in CI/CD pipelines (like GitHub Actions) is a requirement for any automated workflow involving environment-specific data.

Providing Environment Variable Values in GitHub Actions

- **Environment variables can be provided at different levels** in your GitHub Actions workflow:
  - **Workflow-level:** Use the env key at the top level (alongside name and on). All jobs in the workflow will have access to these variables unless a job defines its own variable of the same name.
  - **Job-level:** Use the env key under a job. Only that job will have access to the variables defined here.
- **Example scenario:**
  - Set the database name (MONGODB_DB_NAME) at the workflow level. This value will be shared by all jobs, and is a sensible default for both testing and production (e.g., "gha-demo").
  - Job-specific credentials (like MONGODB_CLUSTER_ADDRESS, MONGODB_USERNAME, MONGODB_PASSWORD) are set at the job level (e.g., for a testing job). This is because test and production environments should use different servers and credentials.
- **How it looks in the workflow file:**

```
name: Deployment
on:
  push:
    branches:
      - main
      - dev
```

```
env:
  MONGODB_DB_NAME: gha-demo
jobs:
  test:
    env:
      MONGODB_CLUSTER_ADDRESS:
      MONGODB_USERNAME:
      MONGODB_PASSWORD:
    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:XYZ
      - name: Run tests
        run: npm test
      - name: Output information
        run: echo "..."
  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Output information
        run: |
          echo "..."
```

- **Key points:**
  - Variables set at the workflow level apply to all jobs.
  - Variables at the job level override those at the workflow level for that job.
  - This structure supports clear separation between shared and job-specific configuration.
  - Suitable for organizing credentials and environment-specific setup for different stages or jobs in your workflow.

Example: Setting Up a Database and Environment Variables in GitHub Actions

- A new MongoDB database is created using MongoDB Atlas (cloud-based, free tier).
- Account is set up and a new cloud database instance is created.
- A user is created, with username and autogenerated password (these are the first environment variable values to store—though storing credentials directly in the workflow is not recommended and will be addressed later).
- The database connection address is obtained via the "connect" button in the Atlas UI and set as the MONGODB_CLUSTER_ADDRESS variable.
- The workflow also requires the current machine's IP (or allow all IPs) in Atlas network access so GitHub Action runners can connect.
- All these environment variables (cluster address, username, password) are added at the job level in the workflow.

- The MONGODB_DB_NAME environment variable is set at the workflow level for all jobs, as all jobs share the database name.
- The PORT environment variable, used in the app code and in Playwright config, is also provided at the job level (e.g., PORT: 8080 or PORT: 3000) to ensure the server starts correctly for the tests.
- Example workflow structure:

```yaml
name: Deployment
on:
  push:
    branches:
      - main
      - dev
env:
  MONGODB_DB_NAME: gha-demo
jobs:
  test:
    env:
      MONGODB_CLUSTER_ADDRESS: cluster0.urmifeb.mongodb.net
      MONGODB_USERNAME: imnilesh18
      MONGODB_PASSWORD: 8pIbXjGlMQ8WtgRt
      PORT: 8080
    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:XYZ
      - name: Run tests
        run: npm test
      - name: Output information
        run: echo "..."
  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Output information
        run: |
          echo "..."
```

- This setup ensures that:
  - All environment variable values (DB name, cluster, user, password, port) are available as needed,
  - The values are provided at the correct workflow or job scope,
  - The application and CI job have what they need to build, start, and test against a real database,
  - The configuration supports secure and flexible environment setup.

Using Environment Variables in GitHub Actions Workflows

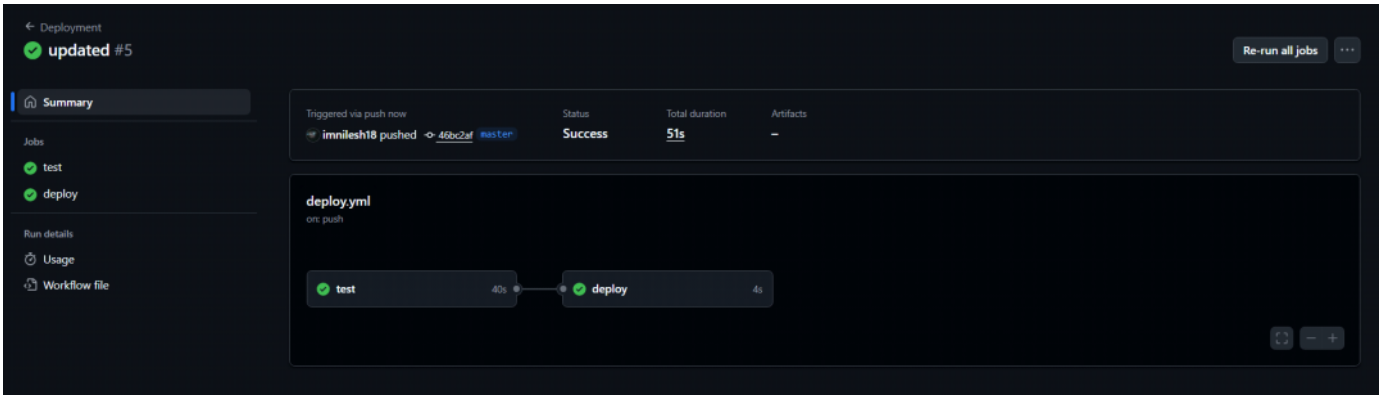
- Environment variables can be used not only in your code but also in your Workflow definition file.
- **Interpolating environment variables:**
  - In run commands, use the appropriate syntax for your shell ($PORT for Linux shell, ${{ env.PORT }} with

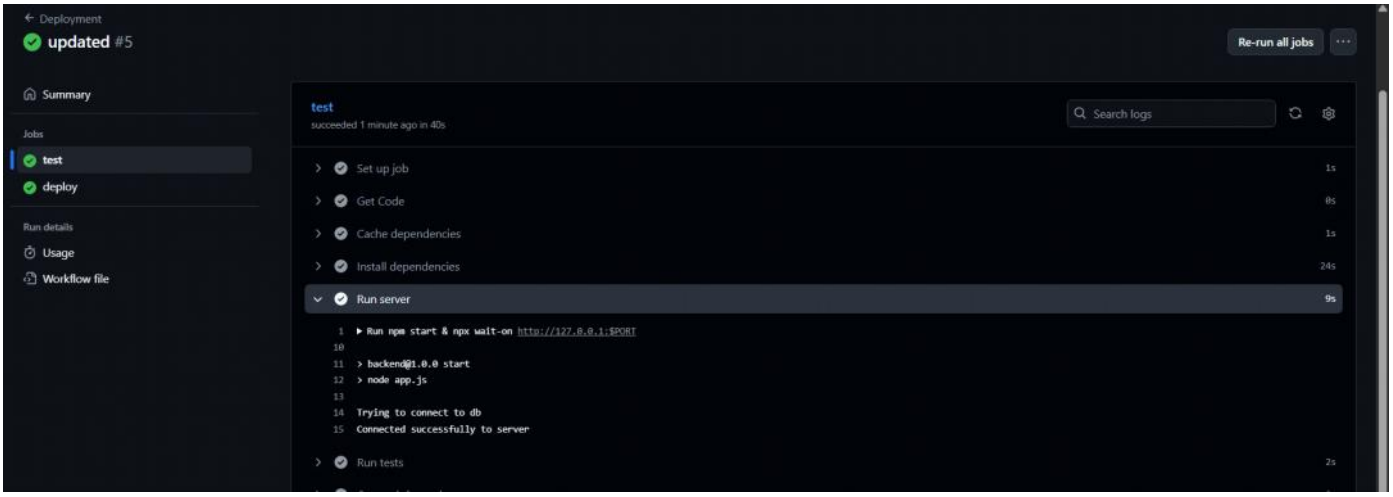
GitHub Actions expression syntax).
    - On Linux (default GitHub runner), use $VARIABLE to access a value in shell commands.
    - For accessing in GitHub YAML expressions, use ${{ env.VARIABLE }}.
    - On Windows (PowerShell), use a different syntax, like $env:PORT.
- **Using environment variables across jobs:**
    - Workflow-level variables (env: at the top) are available in all jobs.
    - Job-level variables (env: under a specific job) are only available in that job; other jobs cannot access them.
    - Step-level variables can also be defined for use in just a single step.
- **Example:**
    - In the test job, job-level env variables are provided (MONGODB_CLUSTER_ADDRESS, MONGODB_USERNAME, MONGODB_PASSWORD, PORT). These are used in the code as well as in run commands.
    - In the deploy job, only the workflow-level variable (MONGODB_DB_NAME) is accessible, not the job-specific variables from the test job, illustrating variable scope.
- **Common checks:**
    - If environment variables are not working, ensure the database is set up correctly, IP access is configured, and the credentials are correct.
- **Sample Workflow Snippet:**

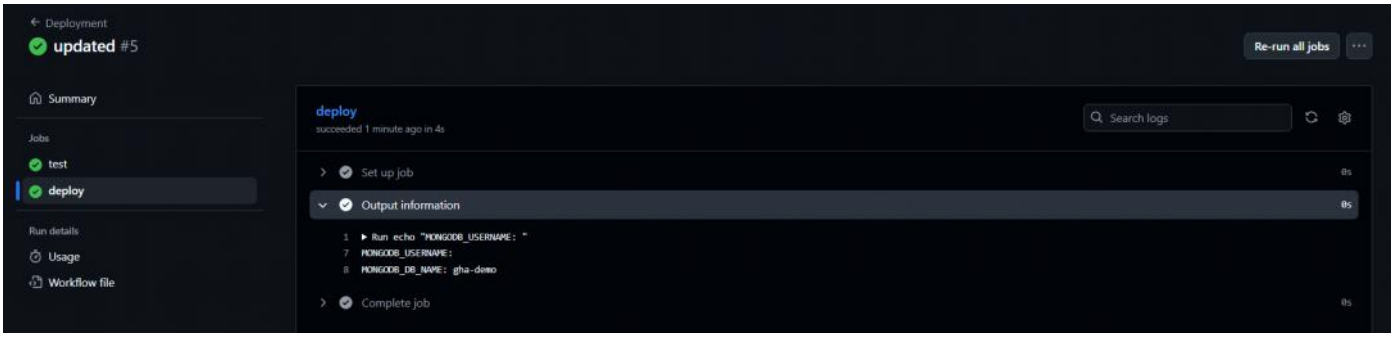
```
name: Deployment
on:
  push:
    branches:
      - master
      - dev
env:
  MONGODB_DB_NAME: gha-demo
jobs:
  test:
    env:
      MONGODB_CLUSTER_ADDRESS: cluster0.yourcreds
      MONGODB_USERNAME: yourusername
      MONGODB_PASSWORD: yourpassword
      PORT: 8080
    runs-on: ubuntu-latest
    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_USERNAME }}"
echo "MONGODB_DB_NAME: $MONGODB_DB_NAME"
```

- **Summary:**
  - Use appropriate syntax to access env vars in scripts and YAML.
  - Workflow env vars are global; job env vars are scoped to the job.
  - Only workflow-level variables are accessible in all jobs; job-specific variables remain private to that job.
  - Correct setup allows the application and workflow to use and display the correct environment variable values in logs and output.



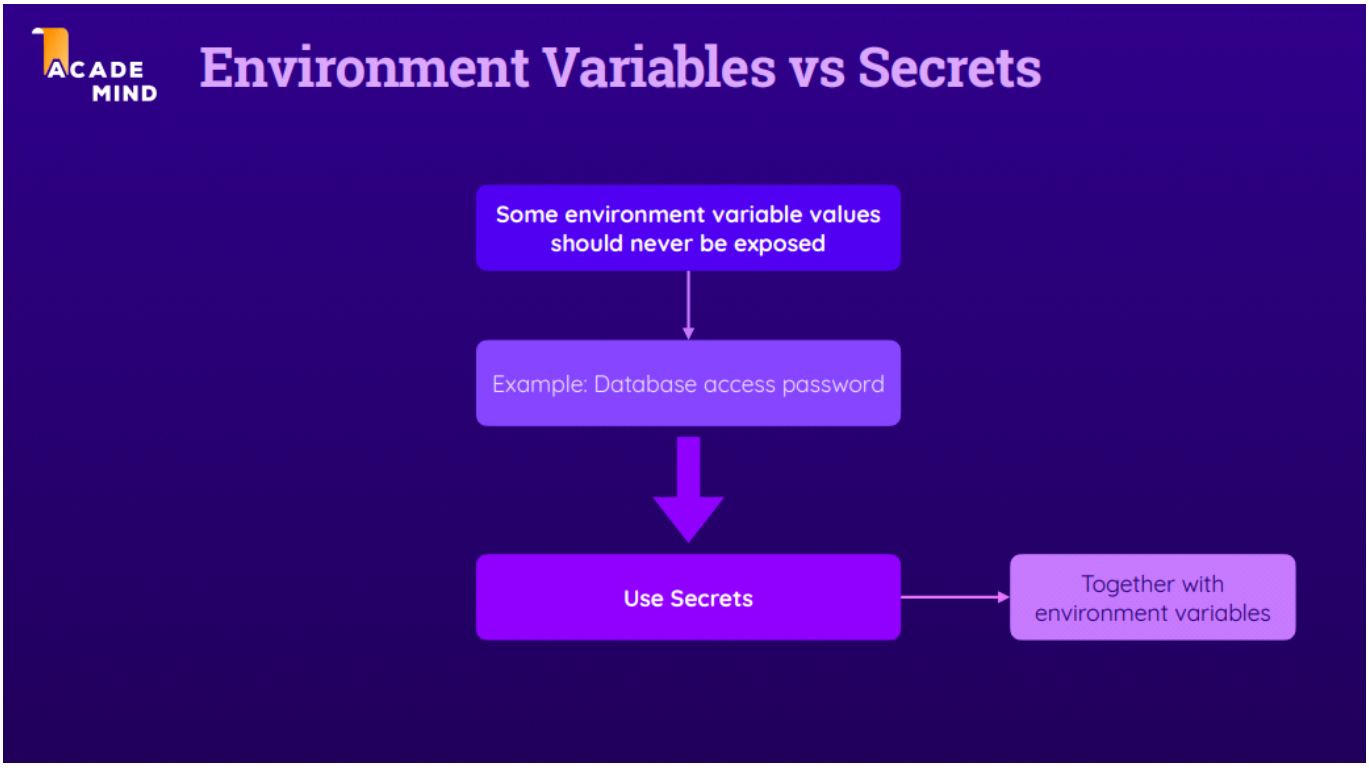




Default Environment Variables
You learned how to set and use your own environment variables.

GitHub Actions also provides a couple of default environment variables that are set automatically: https://docs.github.com/en/actions/learn-github-actions/environment-variables#default-environment-variables
These environment variable can, for example, give you quick access to the repository to which the workflow belongs, the name of the event that triggered the workflow and many other things.

Environment Variables vs Secrets

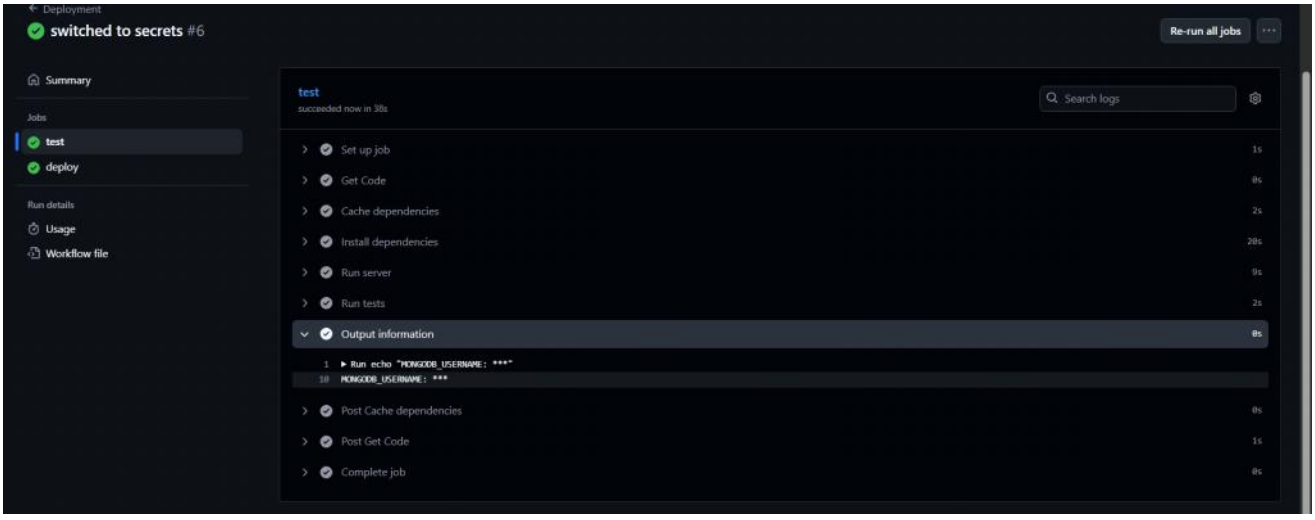


- Environment variables are useful for injecting values into workflows, but some values (like credentials) should not be exposed in the workflow file.
- Storing credentials such as usernames and passwords directly in the workflow file is unsafe, as anyone with repository access can view them.
- Even for testing databases, exposing credentials in code or workflow files is discouraged.
- *Secrets* solve this problem: they are like environment variables, but securely stored and not accessible to anyone viewing the repository or logs.
- GitHub Actions (and similar CI/CD providers) allow you to store secrets:
  - **Organization-level** (if using a GitHub org account)
  - **Repository-level** (for any repository)
- To add a secret in GitHub:
  - Go to repository Settings > Security > Secrets.
  - Add secrets for values like MongoDB username and password.
  - Names do not have to match environment variable names, but matching names is good practice.
  - Once added, secrets can only be updated (not viewed).
- In your workflow definition, reference secrets using the secrets context (${{ secrets.SECRET_NAME }}).
  - Example:
    - MONGODB_USERNAME: ${{ secrets.MONGODB_USERNAME }}
    - MONGODB_PASSWORD: ${{ secrets.MONGODB_PASSWORD }}
- When the workflow runs, values stored as secrets are automatically injected.

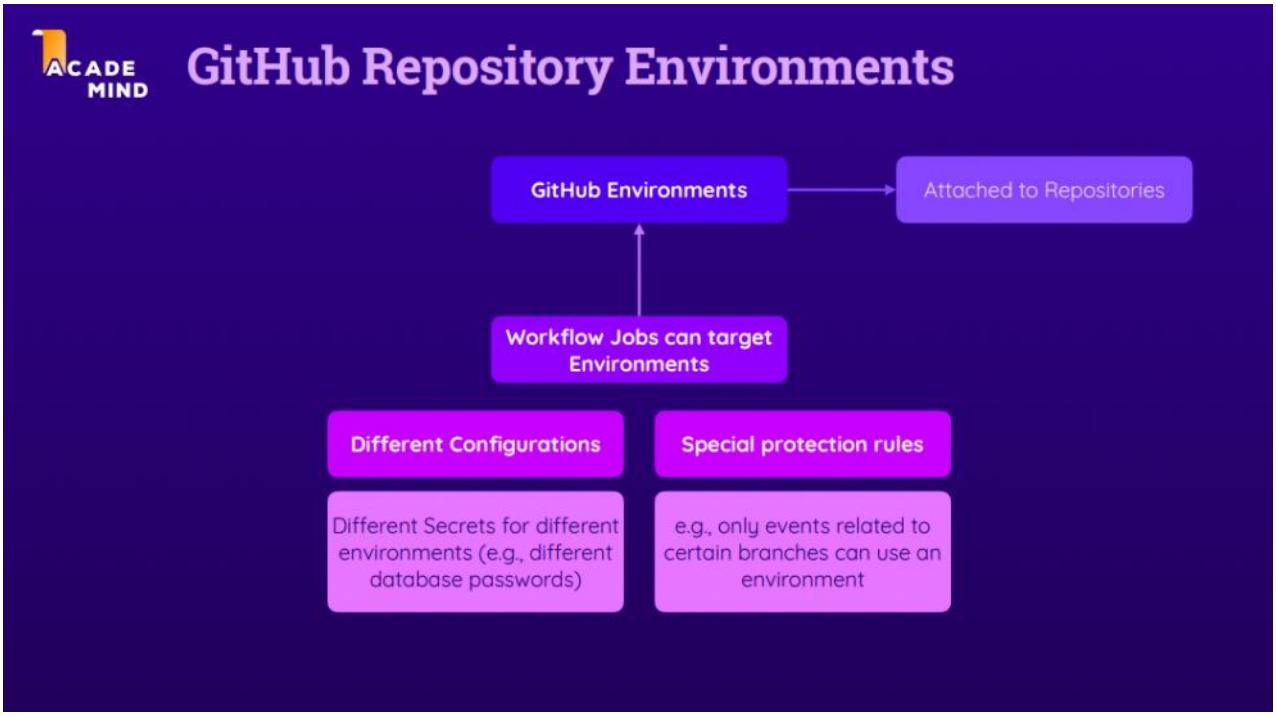
- If you attempt to print a secret value, GitHub will mask it in the logs to preserve its secrecy.
- With secrets added and workflow updated, both jobs run successfully. GitHub hides secret values in the logs, ensuring credentials remain secret.

```yaml
name: Deployment
on:
  push:
    branches:
      - master
      - dev
env:
  MONGODB_DB_NAME: gha-demo
jobs:
  test:
    env:
      MONGODB_CLUSTER_ADDRESS: cluster0.yourcluster
      MONGODB_USERNAME: ${{ secrets.MONGODB_USERNAME }}
      MONGODB_PASSWORD: ${{ secrets.MONGODB_PASSWORD }}
      PORT: 8080
    runs-on: ubuntu-latest
    steps:
      - name: Get Code
        uses: actions/checkout@v3
      - name: Cache dependencies
        uses: actions/cache@v3
        with:
          path: ~/.npm
          key: npm-deps-${{ 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_USERNAME }}"
          echo "MONGODB_DB_NAME: $MONGODB_DB_NAME"
```

- Secrets truly remain secret—GitHub ensures these values are protected from accidental exposure in logs or code.



GitHub Repository Environments

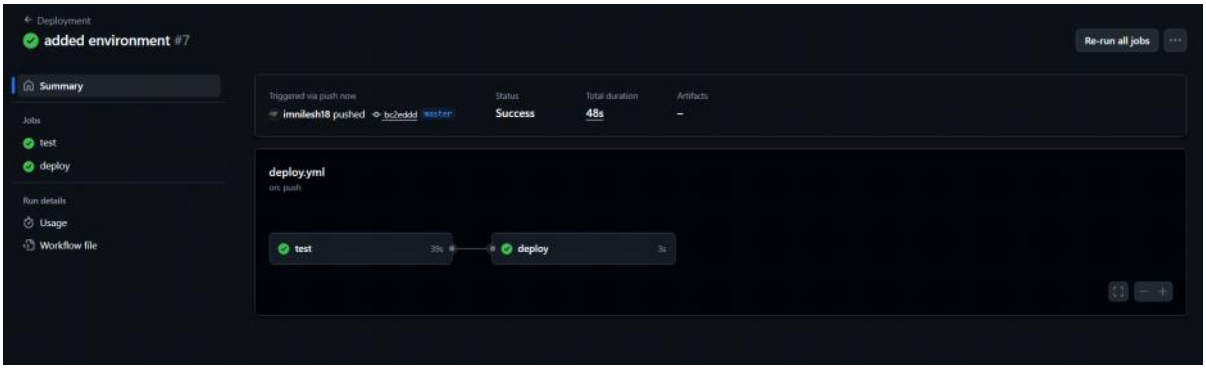


- GitHub provides an environments feature that can be attached to repositories.
- Workflow jobs can target specific environments by specifying the environment key in the job.
- Environments allow:
  - **Different configurations:** Use different secrets for different environments (e.g., testing vs. production can have different database passwords).
  - **Special protection rules:** Restrict which branches/events can use an environment (e.g., only allow jobs on the main branch to use the testing environment).
- To set up environments:
  - Go to repository Settings → Environments.
  - Add environments like "testing", "staging", "production".
  - Each environment can have its own secrets and configuration.
- When a job in a workflow targets an environment, it uses the secrets and configuration of that environment.
- You can enforce extra protections, such as:

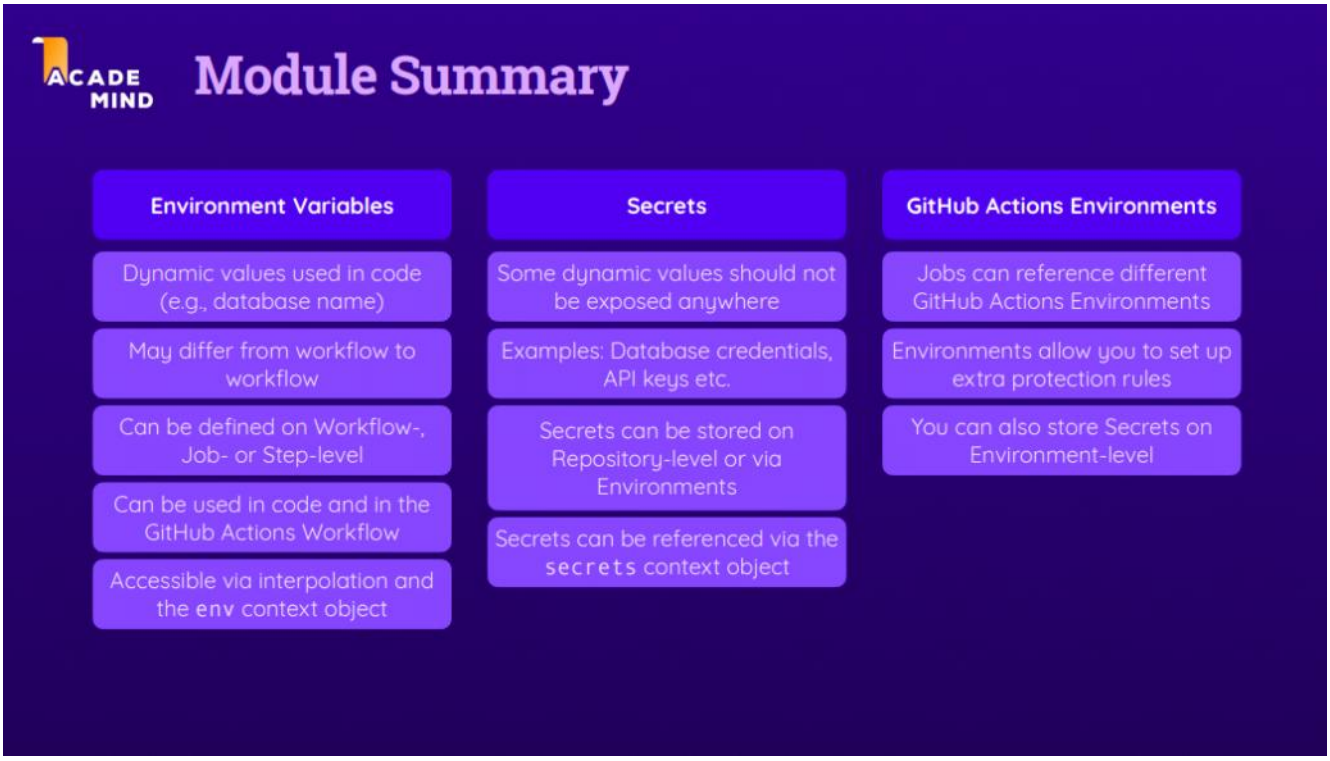
    - Required reviewers (manual approval before running)
    - Wait timers (delays before the job runs)
    - Branch protections (restrict jobs to specific branches)
- Example: The test job uses the "testing" environment, so it gets the secrets from that environment. With a protection rule allowing only the main branch, running the workflow from another branch (e.g., dev) causes the job to be blocked.
- This setup enables safer, more flexible deployments by ensuring that only approved branches and users deploy to sensitive targets, and by safely compartmentalizing secrets per environment.

```yaml
name: Deployment
on:
  push:
    branches:
      - master
      - dev
env:
  MONGODB_DB_NAME: gha-demo
jobs:
  test:
    environment: testing
    env:
      MONGODB_CLUSTER_ADDRESS: cluster0.urmifeb.mongodb.net
      MONGODB_USERNAME: ${{ secrets.MONGODB_USERNAME }}
      MONGODB_PASSWORD: ${{ secrets.MONGODB_PASSWORD }}
      PORT: 8080
    runs-on: ubuntu-latest
    steps:
      - name: Get Code
        uses: actions/checkout@v3
      - name: Cache dependencies
        uses: actions/cache@v3
        with:
          path: ~/.npm
          key: npm-deps-${{ 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_USERNAME }}"
          echo "MONGODB_DB_NAME: $MONGODB_DB_NAME"
```

- Workflows triggered on main succeed using environment secrets.
- Workflows triggered on, e.g., dev, are blocked if the branch is not allowed by environment rules.



Environment Variables, Secrets, and Environments Summary



- **Environment Variables:**
  - Used to inject dynamic values into your code and workflow definition files.
  - Values can differ per workflow and are defined at the Workflow, Job, or Step level.
  - Step-level environment variables can be added if a value is only required for a specific step (not shown in this section, but possible).
  - Values are automatically provided to the code running on the job's runner and can be interpolated or accessed using the env context object in workflow files.
- **Secrets:**
  - Store sensitive information such as database credentials, API keys, etc., that must not be exposed.
  - Can be defined at Repository or Environment-level, and for organizations, also at Organization-level.
  - Reference these secrets using the secrets context object in your workflow file.
- **Environments:**
  - Allow different jobs to reference different GitHub Actions Environments.
  - In Environments, you can add additional protection rules (e.g., deny job execution in certain scenarios).
  - You can also define secrets at the Environment level, enabling different jobs to use different environments

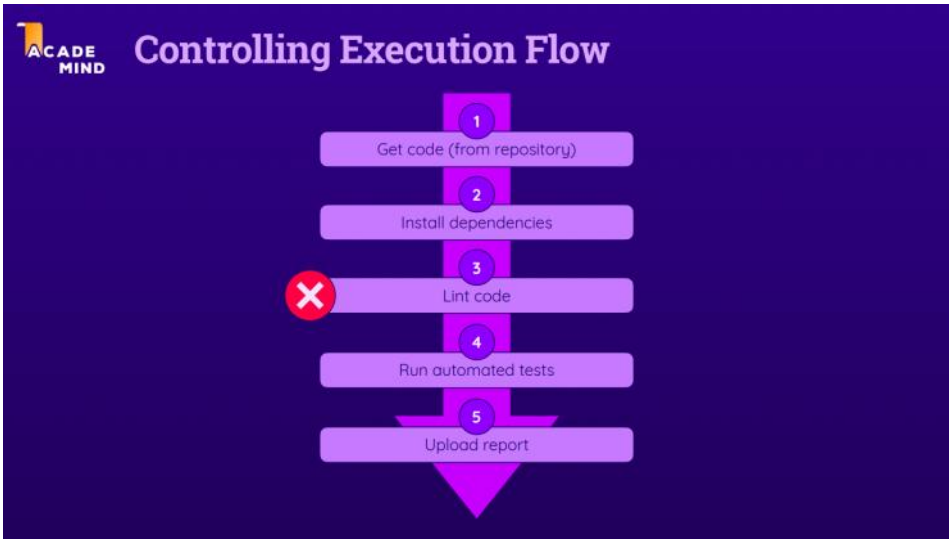
and thus get distinct secret values.

This module covered defining and using environment variables and secrets, associating jobs with environments, using environment-specific secrets, and applying environment-level protection rules to control which jobs can execute and when.

Controlling Workflow & Job Execution

05 August 2025 03:44

Controlling Execution Flow in GitHub Actions

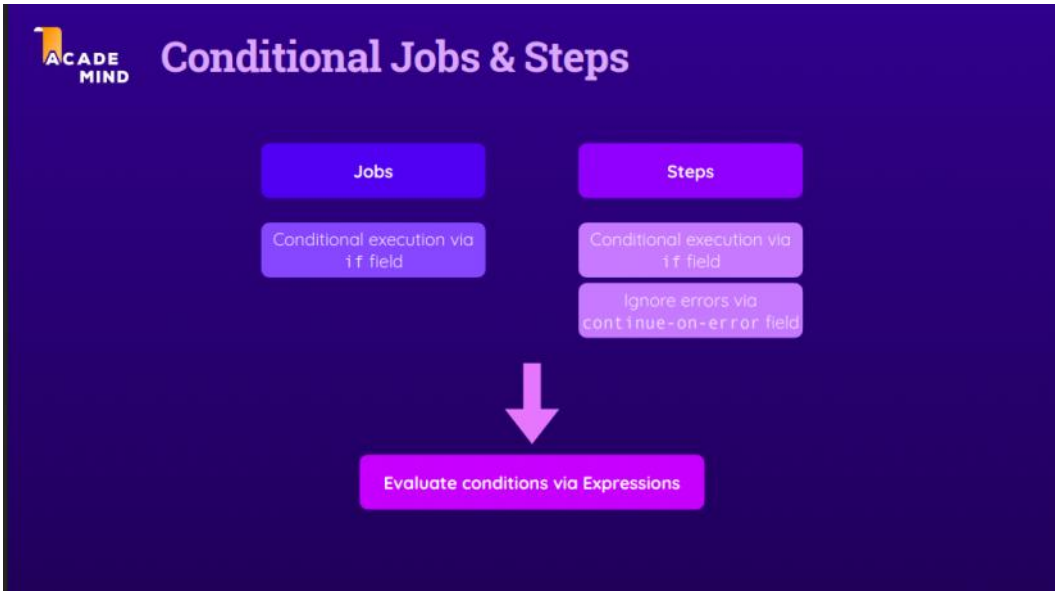


- At this point in the course, you can write solid GitHub Actions workflows that perform various tasks.
- Current workflows always execute from start to finish (A to C) and stop if an error occurs.
- Default behavior:
  - If a step fails, the job is canceled.
  - Subsequent steps in the job are not executed.
  - All other jobs are not executed if one job fails due to a failing step.
- This default error-handling is often useful, but sometimes you may want to continue executing steps even after a failure.
- Some steps may need to execute only if another step fails.
- More generally, you often want greater control over when jobs and steps execute;

it's not always necessary to execute all jobs and steps in every workflow run.

- Often, there are *conditions* for execution.
- This course section covers controlling execution flow:
  - **Controlling when jobs and steps execute**
  - **Running jobs and steps conditionally**
  - **Tools provided by GitHub Actions for execution flow control**
- Other execution flow topics included:
  - **Execution matrix / job matrix:** What it is and when to use it.
  - **Reusable workflows:** How they help save time and reduce redundant writing.

Conditional Execution of Jobs and Steps in GitHub Actions



- You can execute jobs and steps conditionally using two main tools:

1. The if Field

- The if field can be added to both jobs and steps.
- The execution of a job or step happens only if the specified condition is met.

2. The continue-on-error Field (Steps Only)

- The continue-on-error field can be added to steps.
- This allows other steps in the job to continue executing even if the current step fails.

Conditions and Expressions

- Both the if and continue-on-error fields can be configured with custom conditions using expressions.

As shown in the attached image:

- Jobs: Conditional execution via if field.
- Steps: Conditional execution via if field; error ignoring via continue-on-error field.
- Evaluation of conditions is done using expressions.

All these features together give you fine-grained control over workflow execution—when jobs and steps run, continue on error, or are skipped based on logical conditions.

Workflow Structure

- Back in a brand new project (attached).
- The project is actually the same React application as before.
- The focus is not the app, but the workflow.
- A **starting workflow** is provided to you, and it works as-is.
- **The workflow triggers on pushes to the main branch.**
- **Lint job:** Runs linting script to check code style.
- **Test job:**
  - Runs automated tests.
  - Uploads the test report using the upload artifact action.
    - Test report: Due to configuration in vite.config, a test.json file is generated.
    - This test.json is uploaded via the upload artifact action (as learned earlier).
- **Build job:**
  - Runs the build command.
  - Uploads the build output files.
- **Deploy job:**
  - Downloads build files (these would be uploaded to a hosting provider in a real workflow).
  - Outputs dummy content (does not actually deploy).
- **Note regarding the test job:**
  - Uploading the test report is something not covered earlier.
  - The test script in this project produces a report (test.json), which is then uploaded as an artifact.
- **Summary:**
  - This is the demo workflow as provided.
  - Currently, there is **nothing conditional** in this workflow.
  - Adding conditional logic will be explored in the rest of the course section.

```yaml
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-${{ hashFiles('**/package-
lock.json') }}
      - name: Install dependencies
        run: npm ci
      - name: Test code
        run: npm run test
      - name: Upload test report
        uses: actions/upload-artifact@v3
        with:
          name: test-report
          path: test.json
  build:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Cache dependencies
        id: cache
        uses: actions/cache@v3
        with:
          path: ~/.npm
          key: deps-node-modules-${{ 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@v3
        with:
          name: dist-files
          path: dist
  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Get build artifacts
        uses: actions/download-artifact@v3
        with:
          name: dist-files
      - name: Output contents
        run: ls
      - name: Deploy
        run: echo "Deploying..."
```

Conditional Execution: Upload Test Report Only on Failure

- The goal is to upload the test report (test.json) only if the test step fails.
- By default, if the "Test code" step fails, the following steps in the job (including "Upload test report") are not executed, and dependent jobs ("build" and "deploy") are also skipped.
- The "lint" job runs in parallel and completes regardless of test failures.

How to Add the Condition

- You should add an if field to the "Upload test report" step, with a condition that checks if the previous step ("Test code") failed.
- The correct syntax for the if condition is:
  if: failure()
  or, more specifically, if you want to check the outcome of the test step:
  if: ${{ failure() && steps.test_code.outcome == 'failure' }}
  (Assuming the "Test code" step has an id of test_code.)
- This if condition ensures:
  - "Upload test report" runs only when the job is in a failed state due to the test step.
  - It overrides default behavior, so the step executes, even when preceding steps fail.

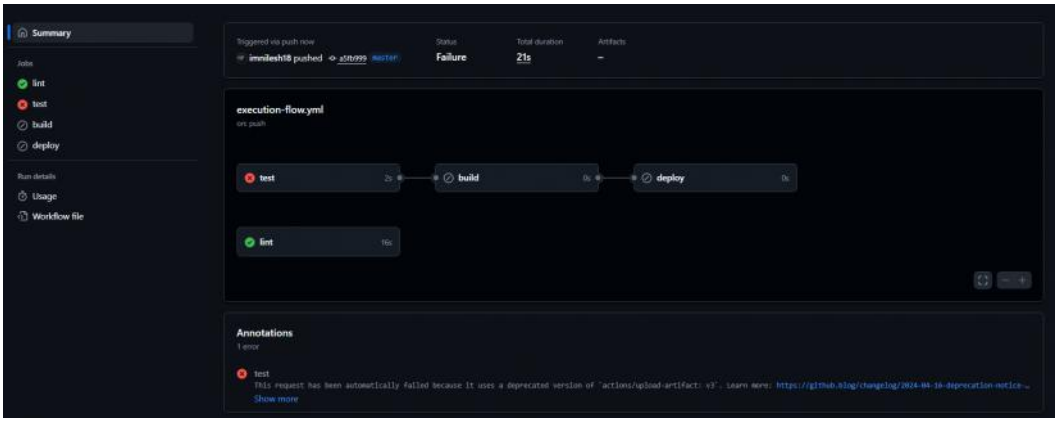
Example Fixed Section

Here's how you can add the condition to your workflow:

```
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
        run: npm run test
      - name: Upload test report
        if:
        uses: actions/upload-artifact@v3
        with:
          name: test-report
          path: test.json
```

Notes

- This setup ensures the test report is uploaded only if tests fail, helping further debugging or sharing of failure details.
- The rest of the workflow behavior remains unchanged, with failure in the "test" job causing "build" and "deploy" jobs to be skipped.



Adding an If Condition to Upload Test Report Step

- To make the upload test report step execute only when tests fail, add the **if** field to the step.
- The condition in the **if** field uses GitHub Actions expression syntax.
- With **if** you can use expressions or context objects—if hard-coded to true/false, you could omit the field entirely.
- Typically, you want a dynamic value, so you can write the condition as a simple expression; the dollar sign and curly braces are optional for the **if** field.

Referencing Another Step's Outcome

- Assign an **id** (e.g., run-tests) to the step you want to check (the "Test code" step).
- In the **if** condition, use the **steps** context object to reference the step by its id and check its outcome:
  - steps.run-tests.outcome == 'failure'
  - The outcome could be: success, failure, cancelled, or skipped.
  - Use == 'failure' to check specifically for a failed step.
  - Wrap the string comparison in single quotes.

Adding the Condition

- With only this condition, the step is **still skipped** when the previous step fails, due to default job behavior.
- To ensure the "Upload test report" executes even if the test step fails, you **must also use the failure() function** in the if condition.

Final Syntax Example

```yaml
- name: Test code
        id: run-tests
        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
```

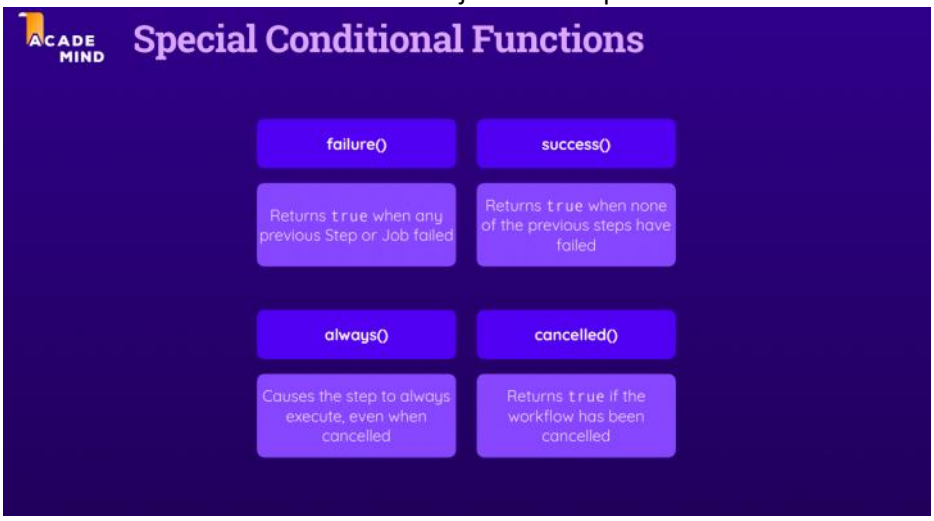
- failure() is a special GitHub Actions function that returns true if the workflow is currently in a failed state.
- Combine it with your outcome check using && (logical AND).

- Other operators you can use in if conditions: ==, !=, >, <, >=, <=, and logical &&/||.

Summary

- By setting an id for the "Test code" step and checking its outcome with the steps context, combined with the failure() function, the test report will only be uploaded if the tests fail.
- This approach gives you fine-grained control over when a step executes, even overriding the default cancellation behavior of GitHub Actions.

GitHub Actions: Special Conditional Functions for Execution Control

- GitHub Actions provides **four special functions** which can be used in your if conditions to control the execution of jobs and steps:
- 

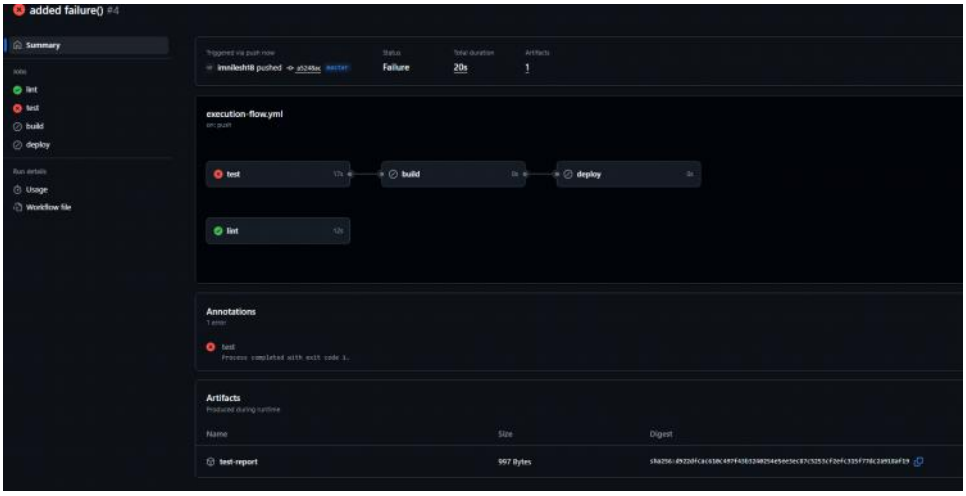
  1. **failure()**
     - Returns true if any previous step or job failed.
     - Enables running a step only when there was a failure earlier (for example, uploading a test report only if tests fail).
  2. **success()**
     - Returns true when none of the previous steps have failed.
     - Ensures execution only if everything succeeded up to that point.
  3. **always()**
     - Always returns true.
     - Ensures a job or step runs every time, regardless of any failures or cancellations.
  4. **cancelled()**
     - Returns true if the workflow has been canceled.
     - Useful for executing cleanup or notification steps on workflow cancellation.
- **How they work in your workflow:**
  - By adding failure() to the condition:
    - GitHub Actions will evaluate that step even if previous steps failed.
    - Combined with logical operators (e.g., && steps.run-tests.outcome == 'failure'), you can narrow the condition to run a step only when a specific preceding step failed.
    - The step (like "Upload test report") will execute if the test step actually failed, but not if earlier steps (like "Install dependencies") failed or were skipped.
- **Example Implementation:**

```yaml
      - name: Install dependencies
        run: npm ci
      - name: Test code
        id: run-tests
        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
```

- **Result:**
  - If the test step fails, "Upload test report" will execute and upload the artifact, allowing you to access the report for debugging purposes.
  - If other steps fail, the condition ensures the upload only happens when the test step is the reason for failure.

Function	Behavior
failure()	Executes when any previous step/job failed
success()	Executes when none of the previous steps/jobs failed
always()	Always executes
cancelled()	Executes if the workflow was cancelled

- These conditional functions provide precise control so you can build robust workflows that respond intelligently to success, failure, and cancellation events, ensuring steps run exactly when needed and as designed.



Using the if Field at the Job Level in GitHub Actions

- The if field can be used on jobs (not just steps) in your workflow to conditionally run a job based on specific criteria.
- When added to a job, the if field controls whether the entire job runs, based on the evaluated condition.

Example Scenario

- Suppose you add a new job report with:
  - A step called "Output information" that echoes a message and GitHub context.
  - An if: failure() condition, aiming for the job to run only if another job fails.

Behavior

- **Without the needs key:**
  - The report job runs in parallel with all jobs.
  - It starts execution immediately, before other jobs have completed.
  - Since no other jobs have failed yet, the failure() function returns false and the job is skipped.
- **With the needs key:**
  - Specify dependencies (e.g., needs: [lint, deploy]).
  - This causes the report job to wait until both lint and deploy jobs have finished.
  - Now, GitHub Actions evaluates failure() after all specified jobs (and their dependencies) have executed.
  - If any job in the dependency chain fails, failure() returns true, and the report job runs.
  - If there are no failures, the report job is skipped.
- The failure() function checks the status of all jobs in the dependency chain, not just the direct dependencies.

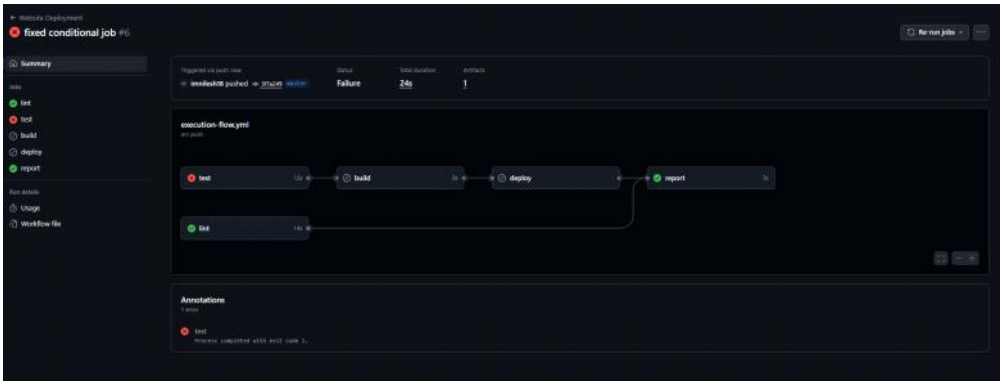
Usage Example

```yaml
report:
  needs: [lint, deploy]
  if: failure()
  runs-on: ubuntu-latest
  steps:
    - name: Output information
      run: |
        echo "Something went wrong"
        echo "${{ toJSON(github) }}"
```

- The report job will run only after both lint and deploy jobs (and all their dependency chain jobs) are complete, and only if any of those jobs failed.

Notes

- You can use other conditions in the if field at the job level, not just failure(), for advanced control (e.g., checking outputs of other jobs).
- This allows workflows to include reporting or notification steps that only execute on error, or other conditional flows as needed.



Optimizing Caching and Conditional Step

Execution in GitHub Actions

- Instead of caching the global npm folder, you can cache the entire node_modules folder, which contains all local project dependencies.
- When node_modules is successfully restored from cache, you can skip the "Install dependencies" step (npm ci) altogether, saving even more time.
- To do this, add an if condition to the "Install dependencies" step. This checks for a cache hit and only runs the step if dependencies were NOT restored from cache.
- The cache action (actions/cache@v3 or @v4) provides a boolean output cache-hit that signals if the cache was used.
- Reference the cache step by its id (e.g., id: cache) and access the output as steps.cache.outputs.cache-hit.
- The condition used:
  if: steps.cache.outputs.cache-hit != 'true'
  - This checks if we did NOT get a cache hit. If true, then npm ci runs; if false, the step is skipped.
- Apply this same condition to every job that uses caching and dependency installation.
- First workflow run:
  - node_modules is not cached, so cache is saved and dependencies are installed.
- Subsequent runs:
  - Cache is available and restored, so "Install dependencies" is skipped and jobs are faster.
- Example (applied to all jobs)

```
- name: Cache dependencies
        id: cache
        uses: actions/cache@v3
        with:
          path: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-
lock.json') }}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
```

- This shows another important use of if conditions: you can base execution on the output of previous steps, not just on success or failure.
- In this example, check for a "cache hit" before deciding whether to install dependencies.
- Also, when all jobs succeed, the conditional "report" job does not execute (because its condition if: failure() is not met), further streamlining the workflow.

Summary:

- Use the if condition at the step level to skip dependency installation when node_modules is restored from cache, optimizing workflows for speed.
- You are not limited to failure checks—if conditions let you base execution on any previous step's output.
- This technique can be applied everywhere you use caching and installation steps.

if vs continue-on-error in GitHub Actions

- The if field allows you to run steps and jobs conditionally—based on expressions you define.
- The continue-on-error field lets jobs continue executing even if a particular step fails.

How continue-on-error Works

- To use, add continue-on-error: true to a step (often set directly to true).
- When set, if that step fails, **the job continues** and the next steps (and subsequent jobs) also execute.
- This differs from the usual behavior, where a failing step would halt execution of the remaining steps in the job and dependent jobs.
- You may also use an expression, but typically this is a hardcoded choice.

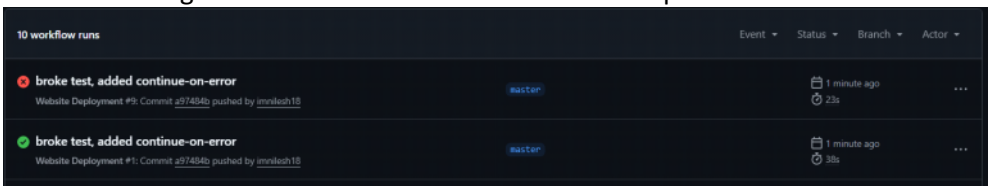
```
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Cache dependencies
        id: cache
        uses: actions/cache@v3
        with:
          path: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-
lock.json') }}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Test code
        continue-on-error: true
        id: run-tests
        run: npm run test
      - name: Upload test report
        uses: actions/upload-artifact@v4
        with:
          name: test-report
          path: test.json
```

Comparison With if

- With an if check, you control *whether specific steps* (or jobs) run based on condition(s). The rest of the workflow behaves normally, meaning if a step fails, its job is still marked as failed, and dependent jobs do not run.
- With continue-on-error: true, a failed step is treated as a success for workflow purposes, so all subsequent steps/jobs proceed, even if the step technically failed.

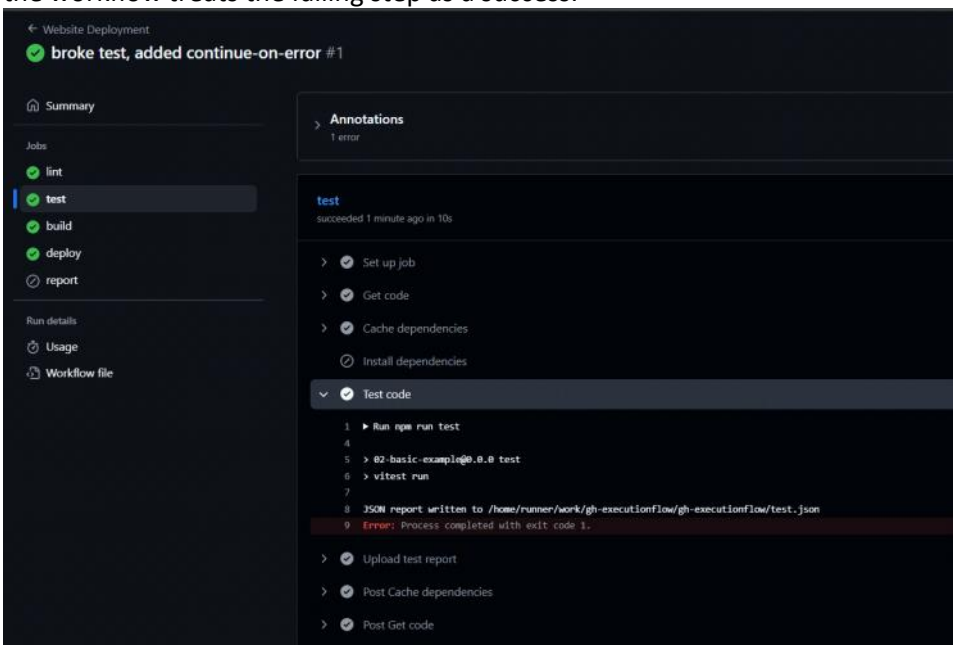
Example Demonstration

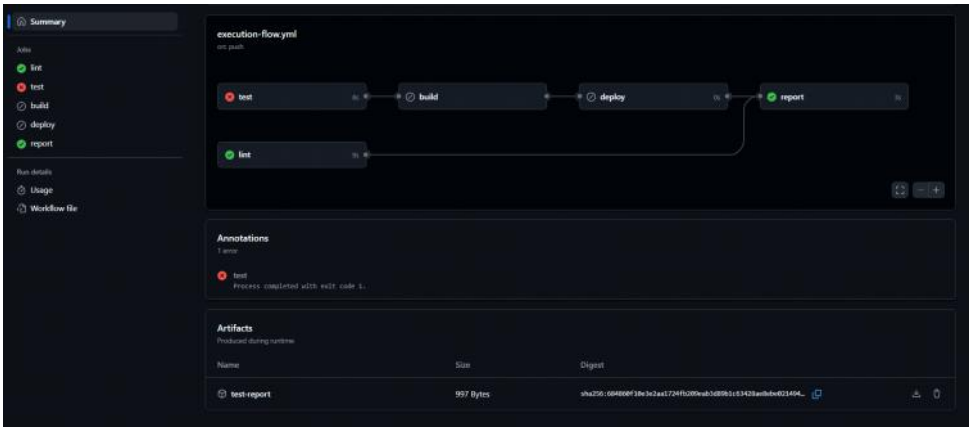
- Two workflows were created and run based on the same commit:
  - One using if for conditional upload of the test report.
  - One using continue-on-error: true for the test step.



Results:

- The workflow with an if condition fails if "Test code" fails since the job is marked as failed; only the conditional step is executed, and following jobs (build/deploy) are skipped.
- The workflow with continue-on-error: true succeeds overall, with a green check mark. All subsequent steps and jobs run—even though "Test code" failed—because the workflow treats the failing step as a success.





Context of outcome vs conclusion

- **outcome:** The value before continue-on-error is evaluated (actual result of the step—could be failure).
- **conclusion:** The value after continue-on-error is applied (often success, unless

technical or cancellation errors).

Should You Use continue-on-error or if?

- Use continue-on-error if you want your entire workflow to continue even if that step fails.
- Use if for finer control—if you want only selected steps or jobs to run, but not the overall workflow.

Matrix Jobs in GitHub Actions

- **Matrix jobs** allow you to run the same job with multiple configurations, such as different Node.js versions, operating systems, or any set of values.
- This is done using the strategy field with a matrix key inside your job.
- Common use cases:
  - Running tests/builds across different environments (OS, dependency versions, etc.)
  - Quickly check cross-platform or cross-version compatibility

How Matrix Works

- Define your job (e.g., build) and use strategy.matrix for multiple keys:
  - For example: node-version: [12,d operating-system: [ubuntu-latest, windows-latest]`
- Use the matrix values in your job definition using ${{ matrix.<key> }} syntax.
- This creates parallel jobs for **all combinations** of the matrix values.

Example Matrix Job

```yaml
name: Matrix Demo
on: push
jobs:
  build:
    continue-on-error: true
    strategy:
      matrix:
        node-version: [12, 14, 16]
        operating-system: [ubuntu-latest, windows-latest]
    runs-on: ${{ matrix.operating-system }}
    steps:
      - name: Get Code
        uses: actions/checkout@v3
      - name: Install NodeJS
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}
      - name: Install Dependencies
        run: npm ci
      - name: Build project
        run: npm run build
```

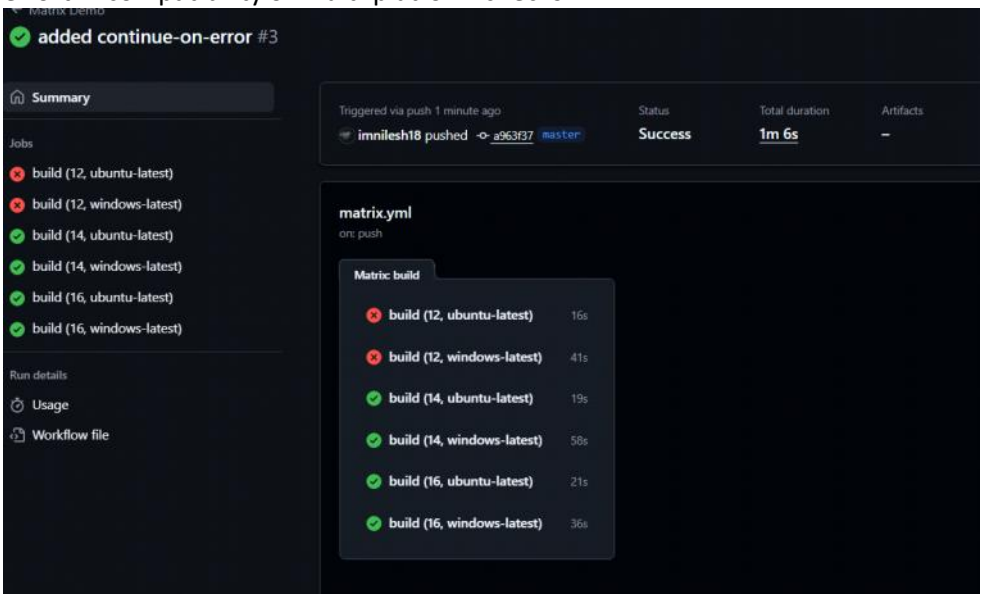
Key Points

- With the above config, the workflow triggers 6 jobs:
  - Ubuntu 12, Ubuntu 14, Ubuntu 16
  - Windows 12, Windows 14, Windows 16
- The jobs run **in parallel by default**.
- If a particular Node version/OS combination fails, by default, other jobs may be canceled—unless continue-on-error: true is set at the job level.

- Setting continue-on-error: true lets other matrix jobs continue running even if one or more fail.

Summary

- Matrix strategy is useful for testing or building your application across multiple environments or tool versions automatically and efficiently.
- You can control job-level error handling with continue-on-error.
- All combinations specified in the matrix will be executed in parallel, saving time and effort in compatibility or multi-platform checks.



Matrix Configuration: include and exclude in GitHub Actions

- In matrix jobs, you can use the **include** key to add specific job combinations to your matrix **without** producing all possible combinations from the lists above.
  - For example, to manually add node-version: 18 and operating-system: ubuntu-latest, use:

```
include:
          - node-version: 18
            operating-system: ubuntu-latest
```

  - This creates a single job with Node v18 on Ubuntu, regardless of other values in the lists.
- You can also add brand new keys and values via **include** that do not exist in your main matrix lists.
- The **exclude** key is used to omit specific combinations from your otherwise auto-generated matrix.
  - For example, to prevent a job from running with node-version: 12 on operating-system: ubuntu-latest:

```
exclude:
          - node-version: 12
            operating-system: ubuntu-latest
```

  - This removes that combination from the matrix.

Usage Summary

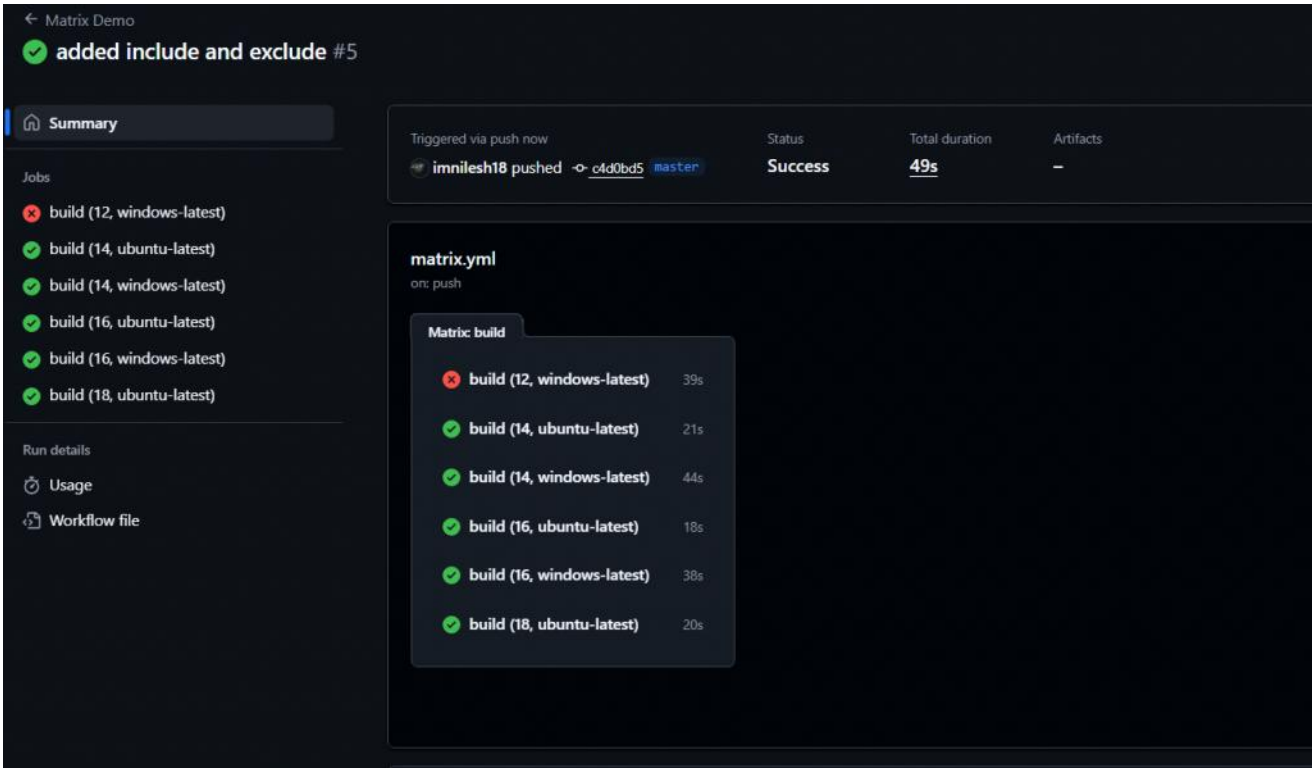
- **include**: Add a standalone combination (or keys) to your matrix, which would not appear if you simply extend the main lists.
- **exclude**: Remove a specific combination from your matrix, when you do not want every permutation.
- The include and exclude keys can be used together for precise control over which jobs are run by the matrix strategy.

Example in Context

```yaml
name: Matrix Demo
on: push
jobs:
  build:
    continue-on-error: true
    strategy:
      matrix:
        node-version: [12, 14, 16]
        operating-system: [ubuntu-latest, windows-latest]
        include:
          - node-version: 18
            operating-system: ubuntu-latest
        exclude:
          - node-version: 12
            operating-system: ubuntu-latest
    runs-on: ${{ matrix.operating-system }}
    steps:
      - name: Get Code
        uses: actions/checkout@v3
      - name: Install NodeJS
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}
      - name: Install Dependencies
        run: npm ci
      - name: Build project
        run: npm run build
```

- Result: All version/OS combinations are created **except** node v12 + Ubuntu, **plus** an extra job for node v18 + Ubuntu.
- Useful for fine-tuning test/build matrices without manually listing every combination.

Summary:
Use include to add custom combinations; use exclude to prevent unwanted combinations. This gives you flexible and powerful control over your matrix jobs in GitHub Actions.



Reusable Workflows in GitHub Actions

- **Reusable workflows** allow you to define a workflow once and then call it from other workflows, making it easy to manage common tasks used across multiple workflows.

How Reusable Workflows Work

- You define a reusable workflow in a file (e.g., reusable.yml).
  - The workflow must use the **workflow_call** event in its on field to allow other workflows to trigger it:

```yaml
name: Reusable Deploy
on: workflow_call
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Output information
        run: echo "Deploying & uploading..."
```

  - In another workflow (e.g., using-reusable-workflow.yml), you can call the reusable workflow in a job using the uses key at the **job** level:

```yaml
deploy:
    needs: build
    uses: ./.github/workflows/reusable.yml
```

  - The given path is relative from the root of your repository, pointing to the reusable workflow file.
  - You can also use a path to a workflow in a different repository.
- When your main workflow runs, jobs like "lint", "test", and "build" execute as usual.
- When it reaches the deploy job, it triggers the reusable workflow (reusable.yml), executing the steps and jobs defined there.
- The reusable workflow does not run on its own when a push or PR happens, but is triggered explicitly from another workflow job.

Example Usage

Reusable Workflow File (reusable.yml):

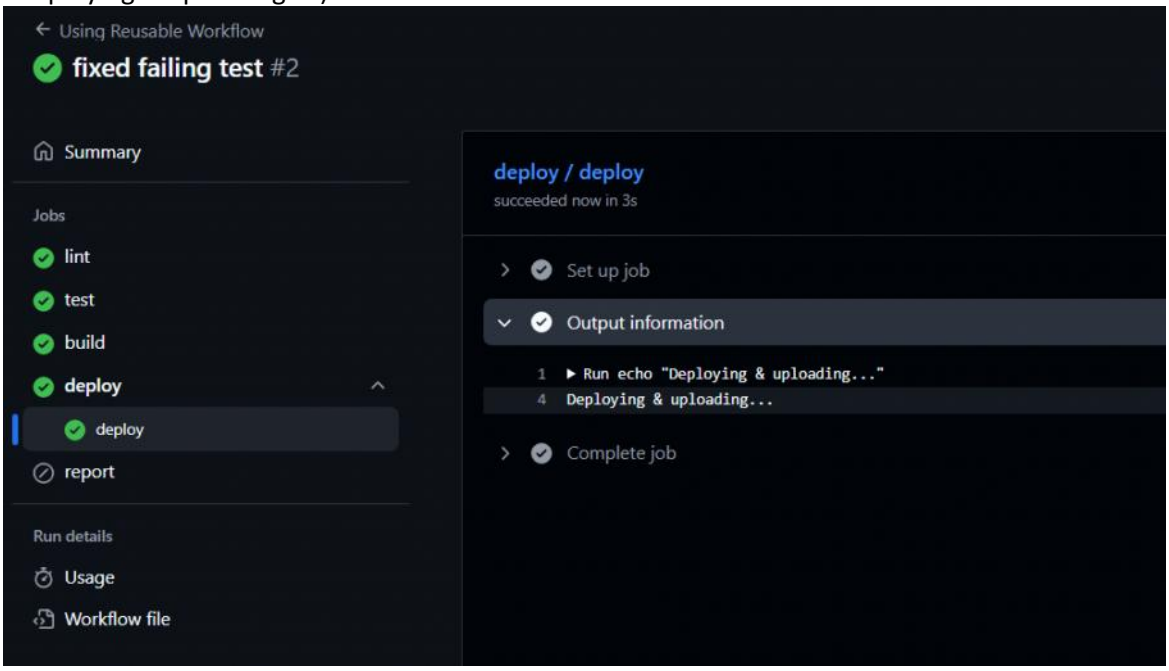
```yaml
name: Reusable Deploy
on: workflow_call
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Output information
        run: echo "Deploying & uploading..."
```

Main Workflow File (using-reusable-workflow.yml):

```yaml
name: Using Reusable Workflow
on:
  push:
    branches:
      - master
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: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Lint code
        run: npm run lint
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Cache dependencies
        id: cache
        uses: actions/cache@v3
        with:
          path: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Test code
        id: run-tests
        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
```

```yaml
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Cache dependencies
        id: cache
        uses: actions/cache@v3
        with:
          path: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Build website
        id: build-website
        run: npm run build
      - name: Upload artifacts
        uses: actions/upload-artifact@v4
        with:
          name: dist-files
          path: dist
  deploy:
    needs: build
    uses: ./.github/workflows/reusable.yml
  report:
    needs: [lint, deploy]
    if: failure()
    runs-on: ubuntu-latest
    steps:
      - name: Output information
        run: |
          echo "Something went wrong"
          echo "${{ toJSON(github) }}"
```

- When deploy runs, the reusable deploy workflow starts and executes its steps (such as outputting "Deploying & uploading...").



Key Points

- Reusable workflows must use the on: workflow_call trigger.
- You reference reusable workflows at the job level in other workflows using the uses field.
- The reusable workflow can be defined in your repo or in another repo.
- This feature is useful for code reuse and maintaining DRY (Don't Repeat Yourself) workflow definitions for common tasks like deployment.

Passing Inputs to Reusable Workflows in GitHub Actions

- **Reusable workflows** can accept *inputs* to make them flexible and adaptable to different situations, like when you need to use a particular artifact name that might change across workflows.

How to Define and Use Inputs in a Reusable Workflow

1. **Define Inputs in the Reusable Workflow**
   - In your reusable workflow (e.g., reusable.yml), use the workflow_call trigger and add an inputs section.
   - Each input can have a description, be set as required or optional, have a default value, and specify a type.

```yaml
name: Reusable Deploy
on:
  workflow_call:
    inputs:
      artifact-name:
        description: The name of the deployable artifact files
        required: false
        default: dist
        type: string
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Get Code
        uses: actions/download-artifact@v4
        with:
          name: ${{ inputs.artifact-name }}
      - name: List files
        run: ls
      - name: Output information
        run: echo "Deploying & uploading..."
```

   - Here, artifact-name is the input. If a calling workflow doesn't provide a value, it defaults to dist.
2. **Use the Input in Your Workflow Steps**
   - Access the input via the inputs context object: ${{ inputs.artifact-name }}.
   - This value is used dynamically in your workflow, such as when specifying what artifact to download.
3. **Pass Inputs When Calling the Reusable Workflow**
   - In the workflow that calls the reusable workflow, provide the input value using the with key at the job level.
   - For example, if you upload an artifact as dist-files, you must pass this as the input:

```yaml
deploy:
    needs: build
    uses: ./.github/workflows/reusable.yml
    with:
      artifact-name: dist-files
```

   - This ensures the reusable workflow downloads and acts on the correct artifact.

What Happens Without Providing an Input

- If a calling workflow does **not** provide an input and your reusable workflow expects one (with a

default), the default value is used.

- In the example above, if artifact-name is not given, it uses dist as the artifact name, which would fail if the artifact was actually named dist-files.

Recap of the Steps

- Define any needed inputs in your reusable workflow with the inputs key under on: workflow_call.
- Access the input using the inputs context object.
- In your calling workflow, pass values for those inputs with the with field; if omitted, the default is used (which may or may not be correct for your use case).
- This allows for flexible, adaptable, and DRY (don't repeat yourself) workflow definitions that respond to context and variable needs.

use-reuse.yml

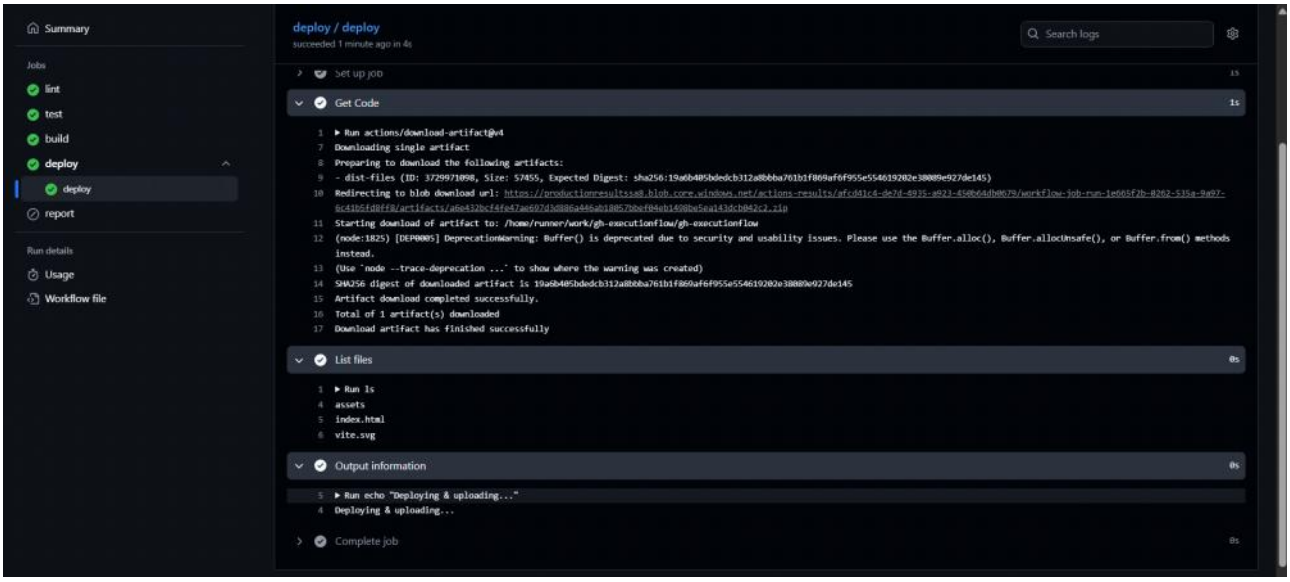
```yaml
name: Using Reusable Workflow
on:
  push:
    branches:
      - master
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Cache dependencies
        id: cache
        uses: actions/cache@v3
        with:
          path: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Lint code
        run: npm run lint
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Cache dependencies
        id: cache
        uses: actions/cache@v3
        with:
          path: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Test code
        id: run-tests
        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: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Build website
        id: build-website
        run: npm run build
      - name: Upload artifacts
        uses: actions/upload-artifact@v4
        with:
          name: dist-files
          path: dist
  deploy:
    needs: build
    uses: ./.github/workflows/reusable.yml
    with:
      artifact-name: dist-files
  report:
    needs: [lint, deploy]
    if: failure()
    runs-on: ubuntu-latest
    steps:
      - name: Output information
        run: |
          echo "Something went wrong"
          echo "${{ toJSON(github) }}"
```

resusable.yml

```
name: Reusable Deploy
on:
  workflow_call:
    inputs:
      artifact-name:
        description: The name of the deployable artifact files
        required: false
        default: dist
        type: string
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Get Code
        uses: actions/download-artifact@v4
        with:
          name: ${{ inputs.artifact-name }}
```

```
  - name: List files
    run: ls
  - name: Output information
    run: echo "Deploying & uploading..."
```



Passing Secrets to Reusable Workflows in GitHub Actions

- **Reusable workflows** can expect—besides inputs—**secrets** for secure data like API keys, passwords, etc.
- To make a reusable workflow accept secrets, add a secrets key under workflow_call, on the same level as inputs.

How to Define and Use Secrets in a Reusable Workflow

In the reusable workflow file (reusable.yml):

```
name: Reusable Deploy
on:
  workflow_call:
    inputs:
      artifact-name:
        description: The name of the deployable artifact files
        required: false
        default: dist
        type: string
    # secrets:
    #   some-secret:
    #     required: false
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Get Code
        uses: actions/download-artifact@v4
        with:
          name: ${{ inputs.artifact-name }}
      - name: List files
        run: ls
      - name: Output information
        run: echo "Deploying & uploading..."
```

- You can specify if a secret is required. If required is true, the workflow will fail unless the secret is provided.
- You **cannot** set a default for a secret.

To provide a secret when using the reusable workflow:

- Use the secrets key in the job that calls the reusable workflow, at the same level as with and uses.

```
deploy:
  needs: build
  uses: ./.github/workflows/reusable.yml
  with:
    artifact-name: dist-files
  # secrets:
  #   some-secret: ${{ secrets.some-secret }}
```

- The secret's identifier must match what was declared in the reusable workflow's secrets section.
- Use the secrets context (${{ secrets.some-secret }}) to pass a secret from your repository/workflow into the reusable workflow.

Key Points

- **Inputs** and **secrets** are both supported for reusable workflows; inputs are passed with with, secrets with secrets.
- This mechanism allows you to pass secure, sensitive values to reusable workflows without exposing them in plain text.
- In your workflow steps, access the secret using the secrets context object: ${{ secrets.some-secret }}.
- You can mark secrets as required or optional, but you cannot set a default value for secrets.

Summary:

- Declare expected secrets under workflow_call > secrets: in your reusable workflow.
- When calling the reusable workflow, provide secrets at the job level with the secrets key.
- This setup supports secure, maintainable, and flexible CI/CD automation across multiple workflows.

Outputs in Reusable Workflows (GitHub Actions)

- Reusable workflows can **return outputs**—not just accept inputs or secrets.
- Outputs allow you to pass results (such as the outcome of a deployment) from the reusable workflow back to the workflow that called it.

How to Set Up Outputs

1. Declare Outputs in the Reusable Workflow

- In your reusable workflow (reusable.yml), under the workflow_call event, add an outputs key.
- Example:

```
outputs:
      result:
        description: The result of the deployment operation
        value: ${{ jobs.deploy.outputs.outcome }}
```

2. Set Outputs in a Job

- In the job (e.g., deploy), use the outputs field to expose step outputs:

```
jobs:
  deploy:
    outputs:
      outcome: ${{ steps.set-result.outputs.step-result }}
```

- For the step that produces the output (e.g., set-result), use the special ::set-output command in a shell script:

```
      - name: Set result output
```

```
        id: set-result
        run: echo "::set-output name=step-result::success"
```

- Assign the value produced in the step (step-result) as an output of the job (outcome), and then reference it in the workflow output (result).

3. Use the Output in the Calling Workflow

- In the workflow using the reusable workflow, add a job that depends on the previous job (via needs) and references the job output:

```
  print-deploye-result:
    needs: deploy
    runs-on: ubuntu-latest
    steps:
      - name: Print deploy output
        run: echo "${{ needs.deploy.outputs.result }}"
```

- This retrieves the output (result, e.g., "success") set by the reusable workflow and prints it in the dependent job.

Workflow Chain Example

- **Reusable workflow** sets and exports outputs.
- **Calling workflow** uses the reusable workflow and accesses its outputs as job outputs via needs.<job>.outputs.<output-name>.

Summary

- Outputs let reusable workflows send results (like deployment status) to calling workflows.
- Define outputs under workflow_call in the reusable workflow.
- Set step outputs using ::set-output, expose them at the job level, and reference them at the workflow level.
- Access workflow outputs from the calling workflow with the needs context.

This pattern allows for return values, making workflows flexible and enabling cross-workflow communication and chaining.

reusable.yml

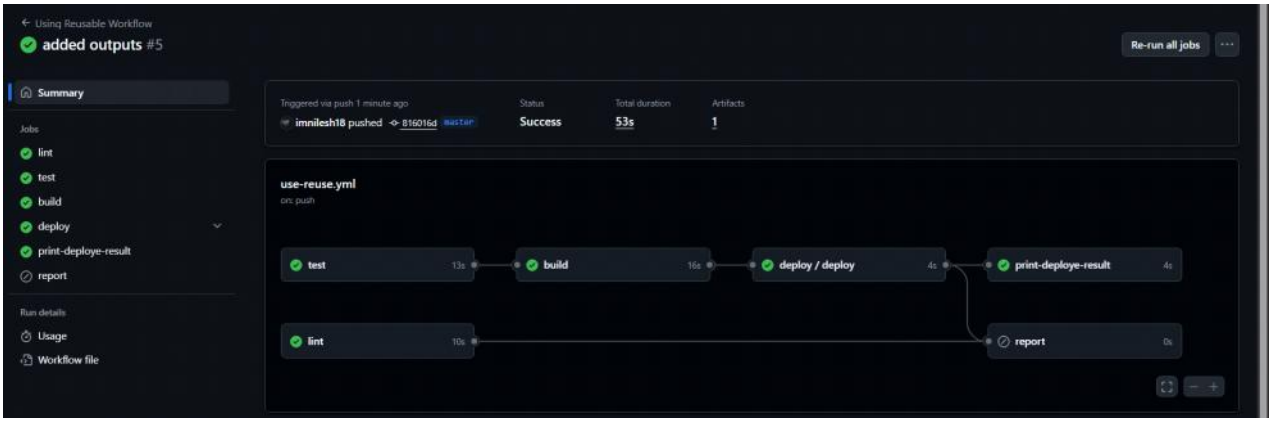
```
name: Reusable Deploy
on:
  workflow_call:
    inputs:
      artifact-name:
        description: The name of the deployable artifact files
        required: false
        default: dist
        type: string
    outputs:
      result:
        description: The result of the deployment operation
        value: ${{ jobs.deploy.outputs.outcome }}
    # secrets:
    #   some-secret:
    #     required: false
jobs:
  deploy:
    outputs:
      outcome: ${{ steps.set-result.outputs.step-result }}
    runs-on: ubuntu-latest
    steps:
      - name: Get Code
```

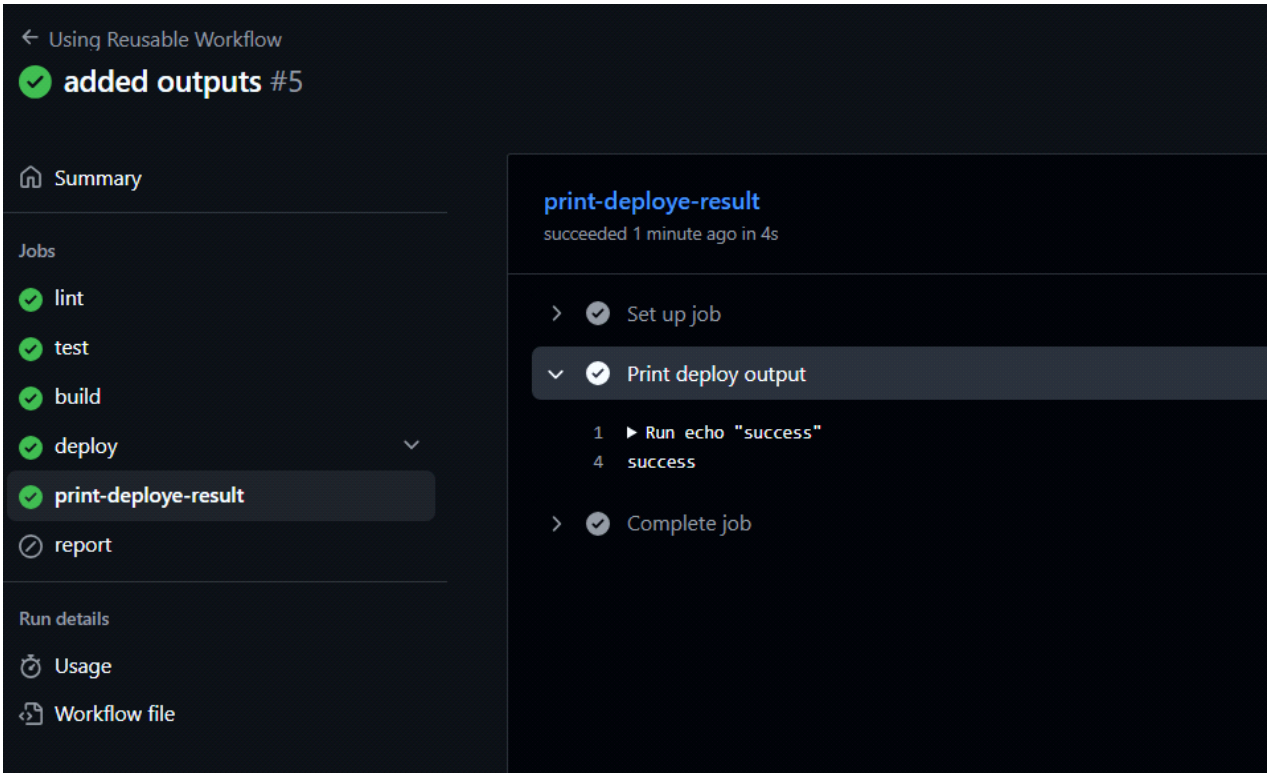
```
        uses: actions/download-artifact@v4
        with:
          name: ${{ inputs.artifact-name }}
      - name: List files
        run: ls
      - name: Output information
        run: echo "Deploying & uploading..."
      - name: Set result output
        id: set-result
        run: echo "::set-output name=step-result::success"
```

use-reuse.yml

```
name: Using Reusable Workflow
on:
  push:
    branches:
      - master
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: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Lint code
        run: npm run lint
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Cache dependencies
        id: cache
        uses: actions/cache@v3
        with:
          path: node_modules
          key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
      - name: Install dependencies
        if: steps.cache.outputs.cache-hit != 'true'
        run: npm ci
      - name: Test code
        id: run-tests
        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:
```

```yaml
  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: node_modules
        key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
    - name: Install dependencies
      if: steps.cache.outputs.cache-hit != 'true'
      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
  uses: ./.github/workflows/reusable.yml
  with:
    artifact-name: dist-files
  # secrets:
  #   some-secret: ${{ secrets.some-secret }}
print-deploye-result:
  needs: deploy
  runs-on: ubuntu-latest
  steps:
    - name: Print deploy output
      run: echo "${{ needs.deploy.outputs.result }}"
report:
  needs: [lint, deploy]
  if: failure()
  runs-on: ubuntu-latest
  steps:
    - name: Output information
      run: |
        echo "Something went wrong"
        echo "${{ toJSON(github) }}"
```





Section Summary: Conditional Execution, Matrix Jobs, and Reusable Workflows in GitHub Actions

- **Conditional Jobs & Steps**
  - You can control which steps or jobs run with the if field and dynamic expressions.
  - This is useful when not all jobs/steps should always run, or when you want to run them only in certain scenarios.
  - Special functions—failure(), success(), cancelled(), and always()—let you change the default execution behavior.
    - Example: Run a step only on failure, or ensure a cleanup step runs always, or execute a job if the workflow was canceled.
  - Use continue-on-error on steps to ignore failures and proceed as if the step succeeded.
- **Matrix Jobs**
  - Easily run the same job with different configurations by creating a matrix of combinations (e.g., different Node versions, OSes).
  - You can add or remove individual combinations using include and exclude in the matrix config.
  - Control whether a single job failure cancels other jobs in the matrix by using the continue-on-error setting at the job level.
- **Reusable Workflows**
  - Reusable workflows allow you to define logic (jobs, steps) once and use it inside other workflows.
  - Make a workflow reusable by adding the workflow_call event.
  - Any workflow with workflow_call can be called from others.
  - You can pass any required data to reusable workflows:
    - **Inputs** (for passing values, options, artifact names, etc.)
    - **Outputs** (to return results from the reusable workflow back to the calling workflow)
    - **Secrets** (to securely share sensitive data)
  - This feature enables you to create modular, DRY workflows and share logic across multiple projects or pipelines.

These features together provide powerful tools for controlling, customizing, and reusing CI/CD logic with GitHub Actions, boosting flexibility and maintainability.

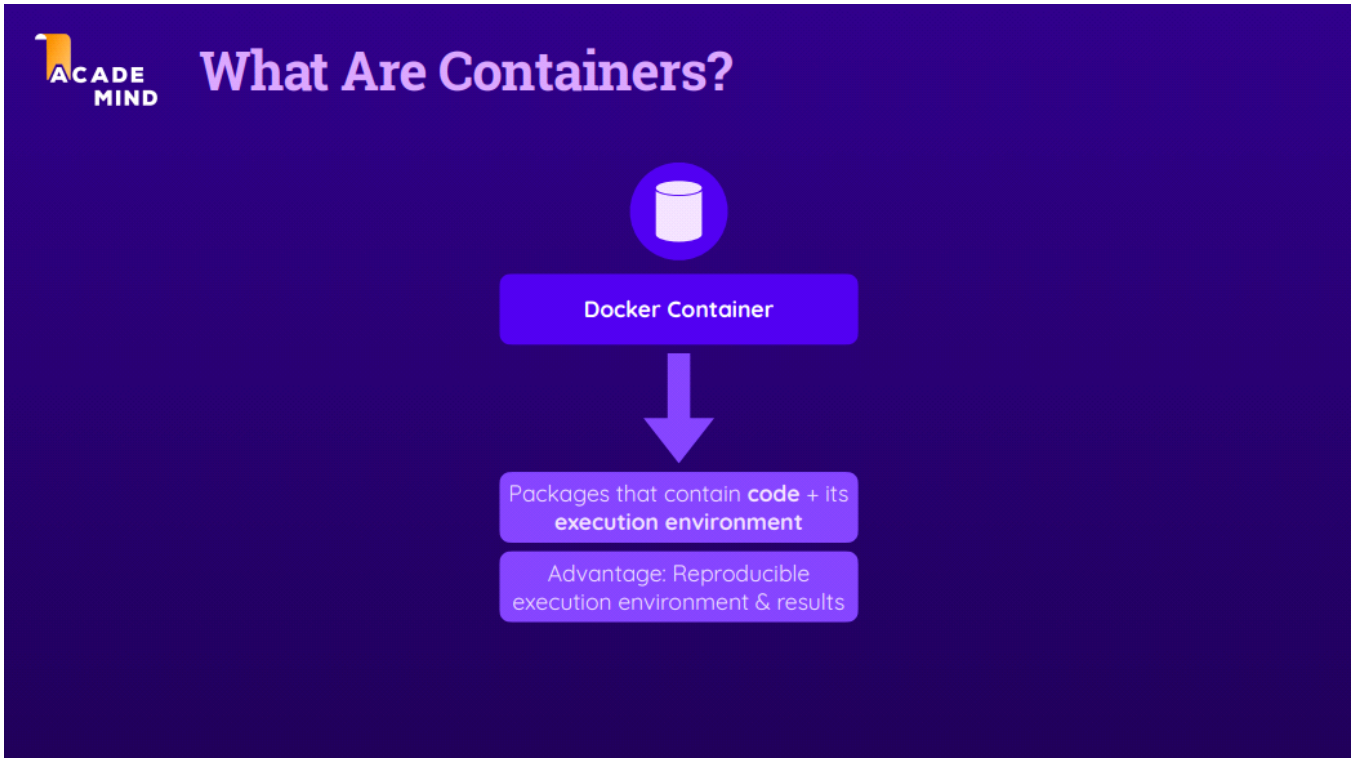
Jobs & Docker Containers

11 August 2025 02:25

Using Containers with GitHub Actions

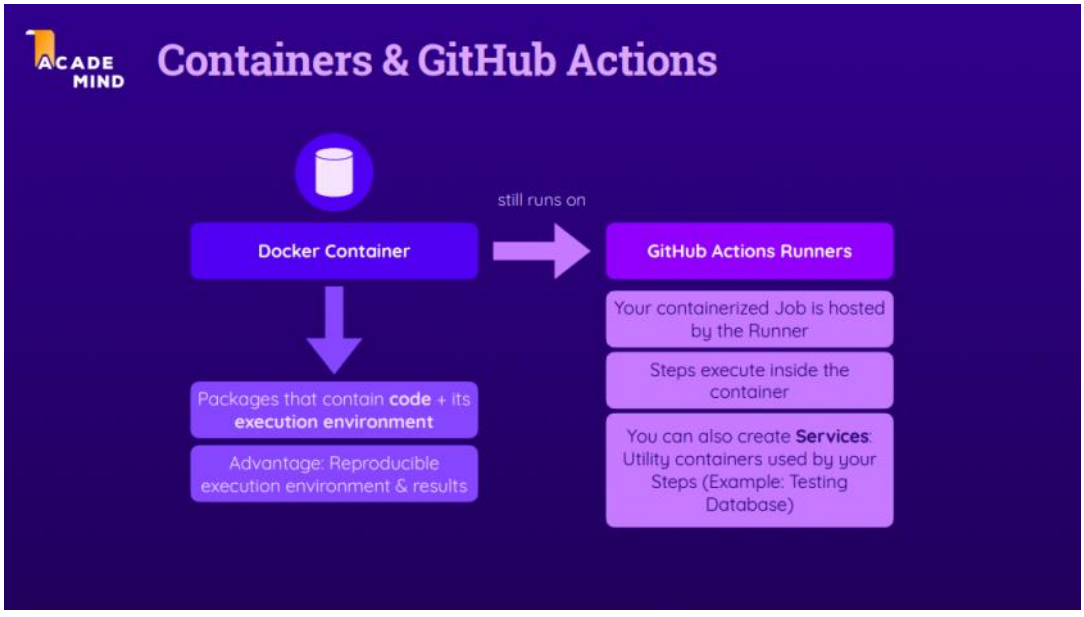
- In this course section, we will explore how you can use **containers** with GitHub Actions.
- A very brief refresher on what containers are (and why you might want to use them) will be provided.
- You will get the most out of this section if you already have some knowledge about containers.
- If you do not, you can also skip this section.
- The section will reintroduce you to the container concept.
- You will learn how to **utilize containers for running jobs in containers** instead of just on the runner machines, which you can specify.
- A concept called **Service Containers** (offered by GitHub Actions) will also be explored.
- Service Containers are a concept that can help in automated workflows.

What Are Containers? (Docker)



- Containers (specifically Docker containers in this section) are packages that contain both your application or website code and the environment needed to execute or build that code.
- The idea behind containers is the same regardless of whether they use Docker or another technology when discussed in development contexts.
- **Why use containers?**
  - Instead of just having your code and running it on different machines (which have different configurations, operating system versions, and installed software), containers provide isolated environments with clearly defined software.
  - This approach creates **reproducible execution environments**, ensuring your code always runs in exactly the same way, no matter where you run the container.
- With containers, you avoid issues caused by environment differences, achieving predictable results and reliable builds and executions across any platform.

Docker Containers with GitHub Actions



- Docker containers are defined via a Dockerfile that specifies how the package (container) should be set up.
- A container starts from a base image (e.g., Linux with Node.js installed).
- Environment variables and setup steps in the Dockerfile prepare a reproducible environment to run the application on any Docker-capable machine.
- This is not a full container course; if containers are unclear, the section can be skipped. GitHub Actions supports containers but they are optional.

Why use containers in GitHub Actions?

- Jobs and steps can run on:
  - The runner machine directly ("just the runner"), or
  - Inside a container defined by you.
- Advantages of containers over running directly on the runner:
  - Full control over the environment, installed software, and setup steps.
  - Avoid reliance on the runner's predefined OS version and preinstalled software.
  - If required tools/versions are missing or differ, you can define them in your container instead of adding ad-hoc install steps in workflows.
  - Reuse the same preconfigured container across different workflows.

How containers run in GitHub Actions

- Containers run on top of GitHub-provided runners; the runner hosts the containerized job.
- The job's steps execute inside the container's isolated environment you defined.

Service containers

- Service (utility) containers can be started alongside your job to provide required services (e.g., databases) needed by the main build/test job.
- This feature will be explored later with a concrete example.

Example Project Setup and Preparation (Pre-Containers)

- Attached example: backend API (same as earlier, slightly altered) plus a Dockerfile.
- Dockerfile purpose:
  - Build a Docker image defining the environment where the code runs.
  - Containers are then run from that image on any machine with Docker installed.
  - Not required for this section; shown to illustrate local/server Docker use.
- Goal here is not to build/run containers locally; focus is on showing the definition and then using GitHub Actions (which also supports Docker containers).
- Repository setup:
  - Initialize Git repository and create a public remote repository.
  - Using public repo to leverage the environments feature for environment-specific secrets consumed by the workflow.
- Push project to GitHub:
  - The attached project already contains a workflow that:
    - Triggers on pushes to main.
    - Starts a web server in the test job, connects to MongoDB, and runs tests.
- Initial run failed because:
  - Secrets not yet added.
  - MongoDB cluster not running/accessible (earlier section setup was shut down).
- Required MongoDB setup steps:
  - Sign in to MongoDB, ensure a cluster exists.
  - Allow access from all IP addresses (so GitHub Actions runners can connect).
  - Ensure a database user exists; assign a new password if needed.
  - Store credentials as environment-specific secrets in the repository's Environments:
    - testing environment
    - Secrets: MongoDB password (newly generated), MongoDB username (e.g., imnilesh18).
- Rerun actions:
  - After adding secrets and ensuring network access, rerun all jobs.
  - If failing, verify the cluster address via "Connect → Connect your application" and use that exact address.
  - After fixing cluster address and pushing changes, workflow connects successfully, runs tests, and completes.

- Next step: Move on to using containers in GitHub Actions.

Workflow (as provided)

```yaml
name: Deployment (Container)
on:
  push:
    branches:
      - master
      - dev
env:
  CACHE_KEY: node-deps
  MONGODB_DB_NAME: gha-demo
jobs:
  test:
    environment: testing
    runs-on: ubuntu-latest
    env:
      MONGODB_CONNECTION_PROTOCOL: mongodb+srv
      MONGODB_CLUSTER_ADDRESS: cluster0.urmifeb.mongodb.net
      MONGODB_USERNAME: ${{ secrets.MONGODB_USERNAME }}
      MONGODB_PASSWORD: ${{ secrets.MONGODB_PASSWORD }}
      PORT: 8080
    steps:
      - name: Get Code
        uses: actions/checkout@v3
      - name: Cache dependencies
        uses: actions/cache@v3
        with:
          path: ~/.npm
          key: ${{ env.CACHE_KEY }}-${{ 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 # requires MongoDB Atlas to accept requests from anywhere!
      - name: Run tests
        run: npm test
      - name: Output information
        run: |
          echo "MONGODB_USERNAME: $MONGODB_USERNAME"
  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Output information
        env:
          PORT: 3000
        run: |
          echo "MONGODB_DB_NAME: $MONGODB_DB_NAME"
          echo "MONGODB_USERNAME: $MONGODB_USERNAME"
          echo "${{ env.PORT }}"
```

Running a Job in a Docker Container (GitHub Actions)

- Current setup: test job runs directly on a GitHub-hosted Ubuntu runner. This is fine and often sufficient.
- Why containers might help:
  - Full control over environment and installed software.
  - Avoid installing tools on each run; reuse a prebuilt image.
  - Example from academind.com: Playwright tests use an official Playwright image with browsers preinstalled to save time and cost.

- For our example, we'll run the test job in a container, even though the runner would work.

How to run a job in a container

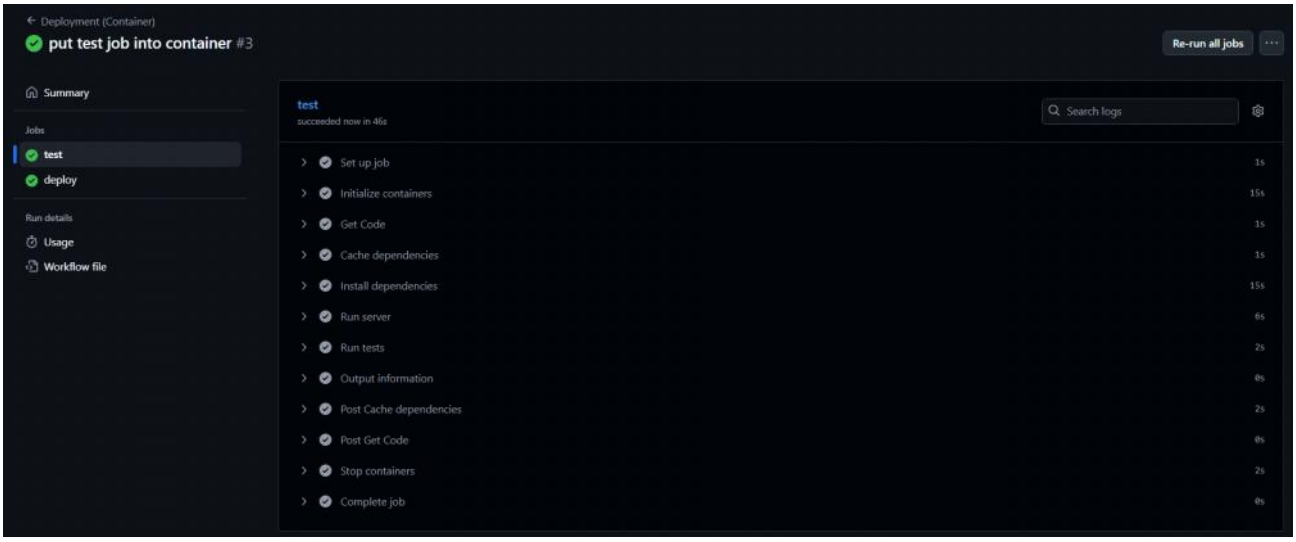
- Add the container key to the job.
- Set the image, e.g., node or node:16. Images come from public registries like Docker Hub (or your own published images).
- You can use:
  - Short form: container: node:16
  - Long form with nested image: (useful when also passing container-level env variables required by the image itself).
- Distinguish between:
  - Container-level env: variables the image itself needs.
  - Job env: variables for steps running inside the container (remain under the job's env).

What actually happens

- The runner still hosts the container; steps execute inside the container's isolated environment (not directly on the runner).
- Actions like checkout and cache still work; GitHub Actions handles them inside the container context.
- Workflow output: initialization downloads the specified image (e.g., node:16), sets up Docker, creates the container, and then runs all steps inside it.

Result

- Same functional outcome as before, but steps run in a controlled, reproducible environment—useful whenever environment control matters.

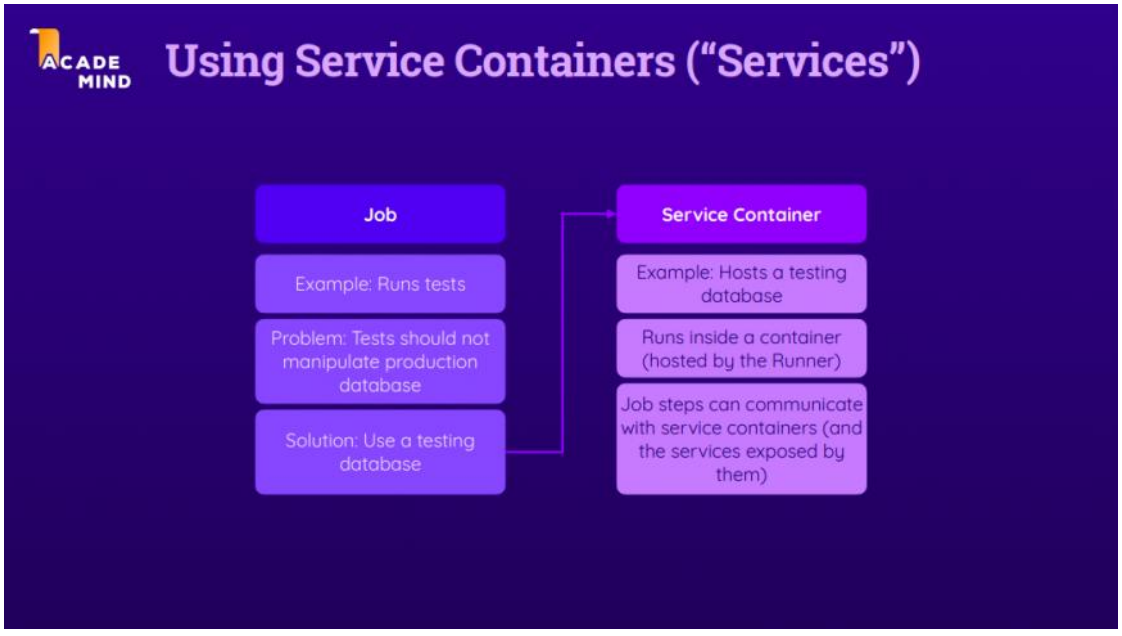


Workflow snippet used

```yaml
name: Deployment (Container)
on:
  push:
    branches:
      - master
      - dev
env:
  CACHE_KEY: node-deps
  MONGODB_DB_NAME: gha-demo
jobs:
  test:
    environment: testing
    runs-on: ubuntu-latest
    container:
      image: node:16
    env:
      MONGODB_CONNECTION_PROTOCOL: mongodb+srv
```

```
      MONGODB_CLUSTER_ADDRESS: cluster0.urmifeb.mongodb.net
      MONGODB_USERNAME: ${{ secrets.MONGODB_USERNAME }}
      MONGODB_PASSWORD: ${{ secrets.MONGODB_PASSWORD }}
      PORT: 8080
    steps:
      - name: Get Code
        uses: actions/checkout@v3
      - name: Cache dependencies
        uses: actions/cache@v3
        with:
          path: ~/.npm
          key: ${{ env.CACHE_KEY }}-${{ 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 # requires MongoDB Atlas to accept
requests from anywhere!
      - name: Run tests
        run: npm test
      - name: Output information
        run: |
          echo "MONGODB_USERNAME: $MONGODB_USERNAME"
  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Output information
        env:
          PORT: 3000
        run: |
          echo "MONGODB_DB_NAME: $MONGODB_DB_NAME"
          echo "MONGODB_USERNAME: $MONGODB_USERNAME"
          echo "${{ env.PORT }}"
```

Service Containers in GitHub Actions



- Jobs and steps can run inside a container for full environment control.
- Beyond that, GitHub Actions can spin up **service containers** alongside a job.

Why service containers?

- Some steps benefit from an auxiliary service to interact with (e.g., a database for automated tests).
- Tests should not use the production database:
  - Avoid accidental data manipulation.
  - Prevent unnecessary load on the production server.
- A separate testing database is the solution, but maintaining a dedicated server is extra work and cost.

What service containers provide

- Run temporary services (e.g., databases) in containers, side-by-side with jobs/steps.
- The service exists only while the workflow runs and is shut down afterward.
- Jobs/steps can communicate with these services during execution.
- Common use: spin up a testing database container for integration tests.
- Applicable to any service required by a job, not just databases.
- Summary: Service containers let workflows provision ephemeral, containerized services that jobs can use during execution, eliminating the need for persistent test infrastructure.

Adding a MongoDB Service Container to a Job

- Service containers can be used whether the job itself runs in a container or directly on the runner.
- Services are defined per job using the services key (plural). Multiple services can be added.

Configure the MongoDB service

- Under services, add a label (e.g., mongodb) for the service.
- Services always run in containers; specify the base image with image:.
  - Use the official MongoDB image: image: mongo.
- Some images require environment variables. For MongoDB:
  - MONGO_INITDB_ROOT_USERNAME (e.g., root)
  - MONGO_INITDB_ROOT_PASSWORD (e.g., example)
- These credentials must match the job's environment variables used by the app/tests so connections succeed.

Effect

- While the job runs, GitHub Actions starts the MongoDB container side-by-side.
- Steps like starting the web server and running tests can connect to this temporary database.
- The service container shuts down automatically after the workflow finishes.

Workflow

```yaml
name: Deployment (Container)
on:
  push:
    branches:
      - master
      - dev
env:
  CACHE_KEY: node-deps
  MONGODB_DB_NAME: gha-demo
jobs:
  test:
    environment: testing
    runs-on: ubuntu-latest
    container:
      image: node:16
    env:
      MONGODB_CONNECTION_PROTOCOL: mongodb+srv
      MONGODB_CLUSTER_ADDRESS: cluster0.urmifeb.mongodb.net
      MONGODB_USERNAME: root
      MONGODB_PASSWORD: example
      PORT: 8080
    services:
      mongodb:
```

```
        image: mongo
        env:
          MONGO_INITDB_ROOT_USERNAME: root
          MONGO_INITDB_ROOT_PASSWORD: example
    steps:
      - name: Get Code
        uses: actions/checkout@v3
      - name: Cache dependencies
        uses: actions/cache@v3
        with:
          path: ~/.npm
          key: ${{ env.CACHE_KEY }}-${{ 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 # requires MongoDB Atlas to accept
requests from anywhere!
      - name: Run tests
        run: npm test
      - name: Output information
        run: |
          echo "MONGODB_USERNAME: $MONGODB_USERNAME"
  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Output information
        env:
          PORT: 3000
        run: |
          echo "MONGODB_DB_NAME: $MONGODB_DB_NAME"
          echo "MONGODB_USERNAME: $MONGODB_USERNAME"
          echo "${{ env.PORT }}"
```

Communicating with Service Containers (Containerized vs. Non-Containerized Jobs)

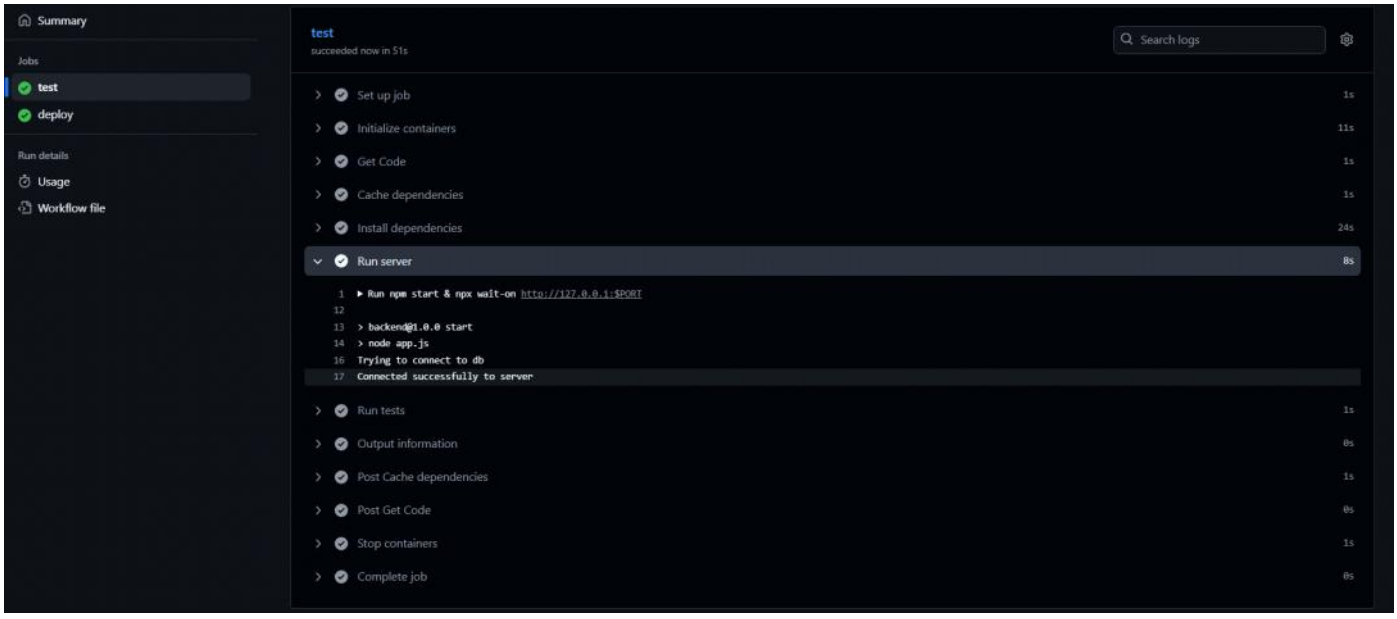
- Service containers provide temporary services (e.g., MongoDB) for jobs. How to connect depends on whether the job itself runs inside a container.

When the job runs in a container

- GitHub Actions creates an internal network where the service label acts as the host.
- Use the service label (e.g., mongodb) as the connection address.
- For this MongoDB example:
  - Update protocol to mongodb (Mongo-specific requirement).
  - Use the same credentials defined for the service container.
  - No port mapping is necessary; the container network handles it.
- Result: Fully isolated workflow relying only on the service container, no external DB needed.

When the job runs on the runner (not in a container)

- Service label hostnames are not available.
- Connect via localhost IP: 127.0.0.1.
- Expose the service container's port(s) with ports: so the runner can reach the service.
  - For MongoDB, map 27017:27017.
- Update connection string to mongodb://127.0.0.1:27017 (with appropriate credentials).
- Workflow still uses the service container successfully without running the job in a container.



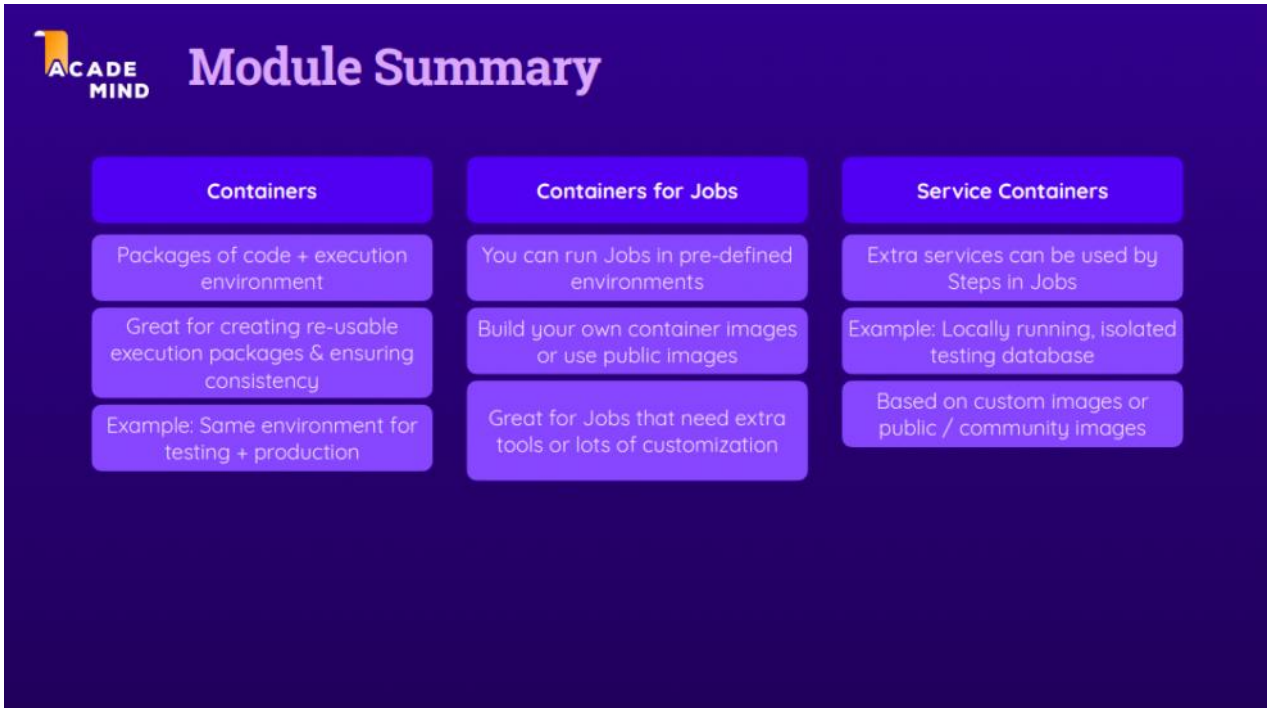
Workflow example (job on runner with service container)

```yaml
name: Deployment (Container)
on:
  push:
    branches:
      - master
      - dev
env:
  CACHE_KEY: node-deps
  MONGODB_DB_NAME: gha-demo
jobs:
  test:
    environment: testing
    runs-on: ubuntu-latest
    # container:
    #   image: node:16
    env:
      MONGODB_CONNECTION_PROTOCOL: mongodb
      MONGODB_CLUSTER_ADDRESS: 127.0.0.1:27017
      MONGODB_USERNAME: root
      MONGODB_PASSWORD: example
      PORT: 8080
    services:
      mongodb:
        image: mongo
        ports:
          - 27017:27017
        env:
          MONGO_INITDB_ROOT_USERNAME: root
          MONGO_INITDB_ROOT_PASSWORD: example
    steps:
      - name: Get Code
        uses: actions/checkout@v3
      - name: Cache dependencies
        uses: actions/cache@v3
        with:
          path: ~/.npm
          key: ${{ env.CACHE_KEY }}-${{ 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 # requires MongoDB Atlas to accept
requests from anywhere!
      - name: Run tests
        run: npm test
      - name: Output information
        run: |
          echo "MONGODB_USERNAME: $MONGODB_USERNAME"
  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: Output information
        env:
          PORT: 3000
        run: |
          echo "MONGODB_DB_NAME: $MONGODB_DB_NAME"
          echo "MONGODB_USERNAME: $MONGODB_USERNAME"
          echo "${{ env.PORT }}"
```

- Choose the approach (containerized job or runner-based job) based on preferences and requirements; both support service containers.

Module Summary: Containers, Containers for Jobs, and Service Containers



- Containers
  - Packages of code plus the environment needed to execute that code.
  - Great for creating reusable execution packages and ensuring consistency.
  - Example: Build an image once and use the same environment for testing and production.
- Containers for Jobs (GitHub Actions)
  - Run jobs in containers instead of directly on the runner.
  - Lets you set up exactly the environment you want and reuse it across jobs and workflows.
  - Use your own container images or public images hosted on a Docker registry.

    - Especially useful for jobs that need extra tools installed or lots of customization.
- Service Containers (GitHub Actions)
    - Separate feature to spin up extra services that steps in jobs can use—even if the job itself is not running in a container.
    - Example: A locally running, isolated testing database available only while the workflow runs.
    - Service containers can be based on public/community images or your own images, as long as they're hosted on a Docker registry.
- Closing note
    - This section briefly reintroduced containers, a full course exists for deeper learning about Docker and containers.

Building & Using Custom Actions

15 August 2025 16:49

Building Custom Actions (Overview)

- We've used many built-in/community actions in workflows:
  - Examples: cache, checkout, upload-artifact.
  - Actions encapsulate tasks of any complexity into single steps (e.g., upload files for later jobs).
- Until now, only existing public actions were used; we didn't create our own.
- This section covers:

  - What custom actions are and why to build them.
  - The different types of custom actions available (there's more than one kind).
  - How to build and use custom actions, seeing the concepts in practice.

Why Build Custom Actions?

- Public actions exist (e.g., upload-artifact) and many more are available in the Marketplace, but custom actions can still be valuable.

Main Reasons

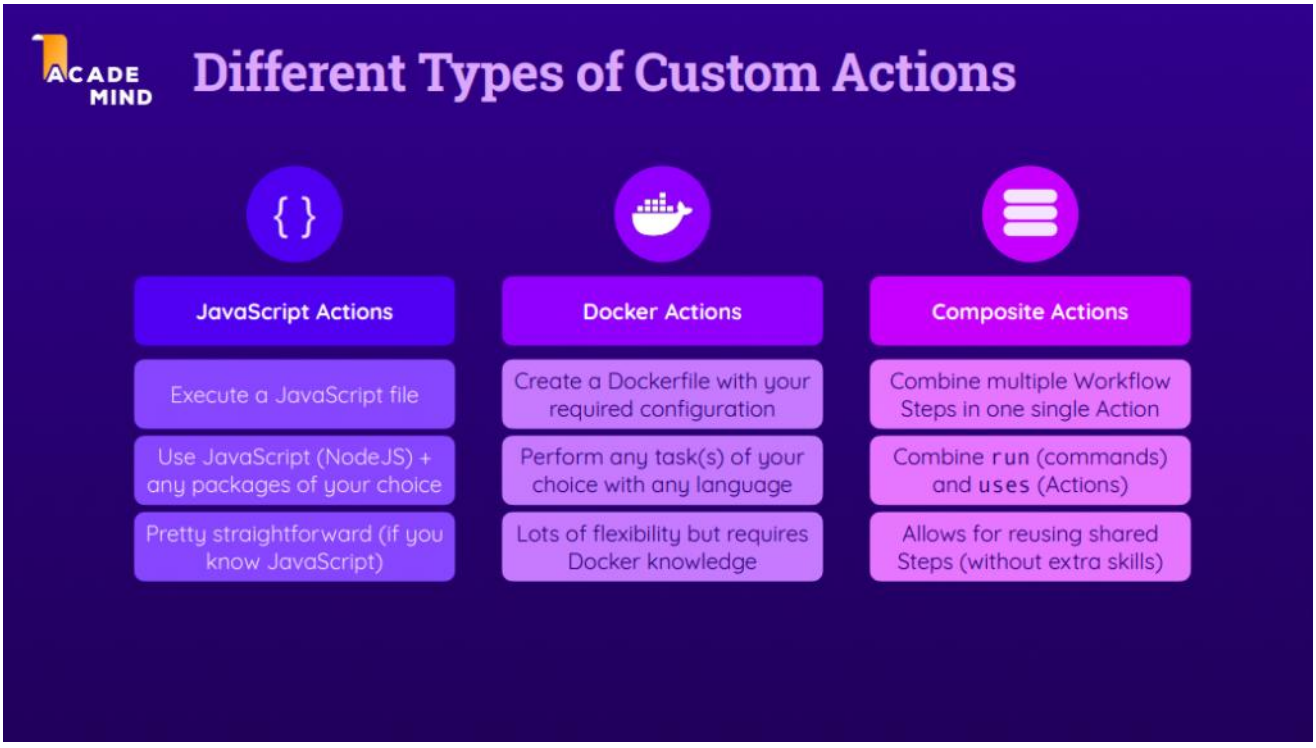
- Simplify workflow steps
  - Replace multiple, possibly complex, repeated step definitions with a single custom action.
  - Reuse the same logic across multiple jobs to avoid copy-paste.
  - Example: Group “cache dependencies” + “install dependencies” used in lint, test, and build jobs into one action.
- No suitable existing action
  - Public actions might not solve the exact problem or the way it’s needed in a workflow.
  - A custom action can implement any logic required, with the preferred programming language.

Additional Motivation

- After creating a custom action, it can be published to the Marketplace to help others and contribute to the community.

Types of Custom Actions in GitHub Actions

- There are three main types of custom actions:
  - JavaScript Actions
  - Docker Actions
  - Composite Actions



JavaScript Actions

- Write the action's logic in JavaScript.
- A JavaScript file is executed whenever the action runs.
- Uses Node.js features and any npm packages.
- Straightforward if you know JavaScript.
- Not ideal if you don't know JavaScript (you must code in JS for this type).

Docker Actions

- Build a containerized action defined by a Dockerfile.
- GitHub builds and runs a container based on your image definition.
- Perform any task using any programming language inside the container.
- Very flexible (choose base image, environment, tools).
- Requires basic Docker knowledge.

Composite Actions

- No programming language required.
- Combine multiple workflow steps into a single, reusable action.
- Can include both run commands and uses (other actions).
- Ideal for grouping repeated step sequences across jobs/workflows.

Key Takeaway

- All three approaches produce actions used like public actions; the difference lies in how you build them:
    - Code-centric (JavaScript),
    - Container-centric (Docker),
    - Step-composition (Composite).

Getting Started with Custom Actions (Composite Actions First)

- The starting project is a simple React app with an existing workflow file in .github/workflows; no new features are used yet.
- Repository setup steps taken:

- Initialize the local repository and create the initial commit.
- Create a new remote repository (public or private; choice doesn't matter).
- Link the local repository to the remote and push (establish main-to-main connection).
- The included workflow triggers and finishes successfully, but doesn't use custom actions yet.

Next Step: Build a Custom Composite Action

- Begin custom action development with a Composite Action (the easiest type to start with).
- Composite Actions will be introduced and implemented in the next step to extend the current workflow.

Creating a Composite Action and Preparing to Use It

- **Goal:** Simplify repeated steps (cache dependencies + install dependencies) by creating a composite action.

Where to place the action

- In the existing project, create .github/actions/cached-deps/.
- Actions inside a repo are usable only by workflows in that same repo.
- Every action must have an action.yml file that defines the action.

Composite action definition (action.yml)

- Name and description define how the action appears (e.g., in Marketplace if published).
- Use runs.using: composite to declare a composite action.
- Define steps under runs.steps.
- When a step uses run, add shell: (e.g., bash) in composite actions.
- It's fine to use other actions inside a composite action.

Example action.yml:

```yaml
name: 'Get & Cache Dependencies'
description: 'Get the dependencies (via npm) and cache them.'
runs:
  using: 'composite'
  steps:
    - name: Cache dependencies
      id: cache
      uses: actions/cache@v3
      with:
        path: node_modules
        key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
    - name: Install dependencies
      if: steps.cache.outputs.cache-hit != 'true'
      run: npm ci
      shell: bash
```

Current workflow (before replacing with the action)

- The workflow repeats "cache deps + install deps" in lint, test, and build jobs.
- Next step: replace those repeated step pairs with a single uses reference to the new composite action.

Using a Local Composite Action in Your Workflow

- Replace repeated steps with a single custom action call.

How to reference the action

- Use the uses key (same as with public actions).
- Value depends on where the action is stored:
  - If in a separate repository: owner/repo.

- If local in the same repository: path from the repository root (not from the workflow file).

Example (local path):

```
- name: Load & cache dependencies
  uses: ./.github/actions/cached-deps
```

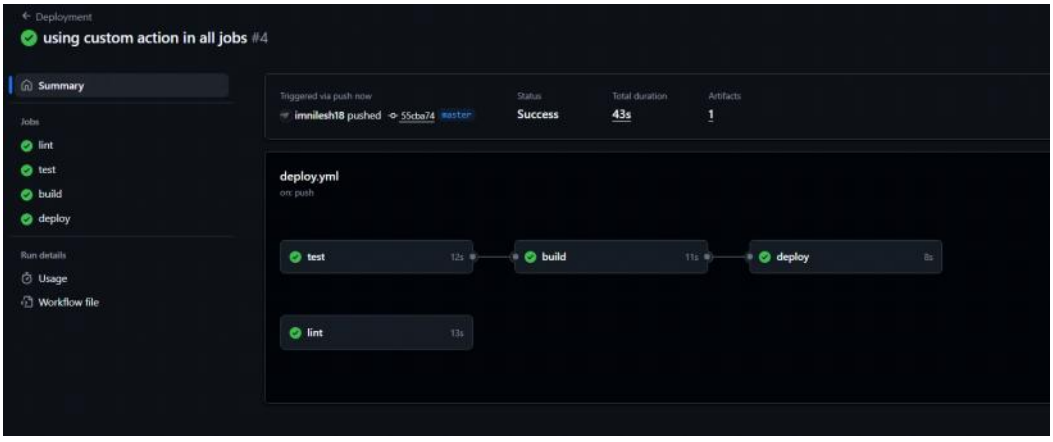
- GitHub will automatically look for action.yml inside that folder.

Apply across jobs

- Replace the two-step sequence (cache + install) in each job (lint, test, build) with the single action invocation.
- Benefits:
  - DRY workflow: removes duplicate step definitions.
  - Clearer workflow files.
  - Centralized logic (e.g., conditional install when cache miss).

Result

- After committing and pushing:
  - The workflow triggers and the custom action runs in the lint job (and others once added).
  - Cache is used when available; install runs only on cache miss.
  - Final runs show all jobs succeeding with the simplified action usage.



deploy.yml

```yaml
name: Deployment
on:
  push:
    branches:
      - master
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Load & cache dependencies
        uses: ./.github/actions/cached-deps
      - name: Lint code
        run: npm run lint
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Get code
        uses: actions/checkout@v3
      - name: Load & cache dependencies
        uses: ./.github/actions/cached-deps
      - name: Test code
```

```yaml
        id: run-tests
        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: Load & cache dependencies
        uses: ./.github/actions/cached-deps
      - name: 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 code
        uses: actions/checkout@v3
      - name: Get build artifacts
        uses: actions/download-artifact@v4
        with:
          name: dist-files
          path: ./dist
      - name: Output contents
        run: ls
      - name: Deploy site
        run: echo "Deploying..."
```

action.yml

```yaml
name: 'Get & Cache Dependencies'
description: 'Get the dependencies (via npm) and cache them.'
runs:
  using: 'composite'
  steps:
    - name: Cache dependencies
      id: cache
      uses: actions/cache@v3
      with:
        path: node_modules
        key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
    - name: Install dependencies
      if: steps.cache.outputs.cache-hit != 'true'
      run: npm ci
      shell: bash
```

Adding and Using Inputs in a Composite Action

- Make actions configurable by defining inputs so the same action can behave differently across jobs.

Define inputs in action.yml

- Add an inputs section alongside name, description, and runs.
- For each input:
  - Provide a description.
  - Set required: true|false.
  - Optionally supply a default when not required.

Example:

```yaml
name: "Get & Cache Dependencies"
description: "Get the dependencies (via npm) and cache them."
inputs:
  caching:
    description: "Whether to cache dependencies or not."
    required: false
    default: "true"
runs:
  using: "composite"
  steps:
    - name: Cache dependencies
      if: inputs.caching == 'true'
      id: cache
      uses: actions/cache@v3
      with:
        path: node_modules
        key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
    - name: Install dependencies
      if: steps.cache.outputs.cache-hit != 'true' || inputs.caching != 'true'
      run: npm ci
      shell: bash
```

- Use the inputs context to reference input values in if conditions.
- Logic here:
  - Cache step runs only when caching is 'true'.
  - Install step runs on cache miss OR when caching is disabled.

Pass inputs from workflow

- Provide values with: when using the action.
- Customize per job or step.

Example usage:

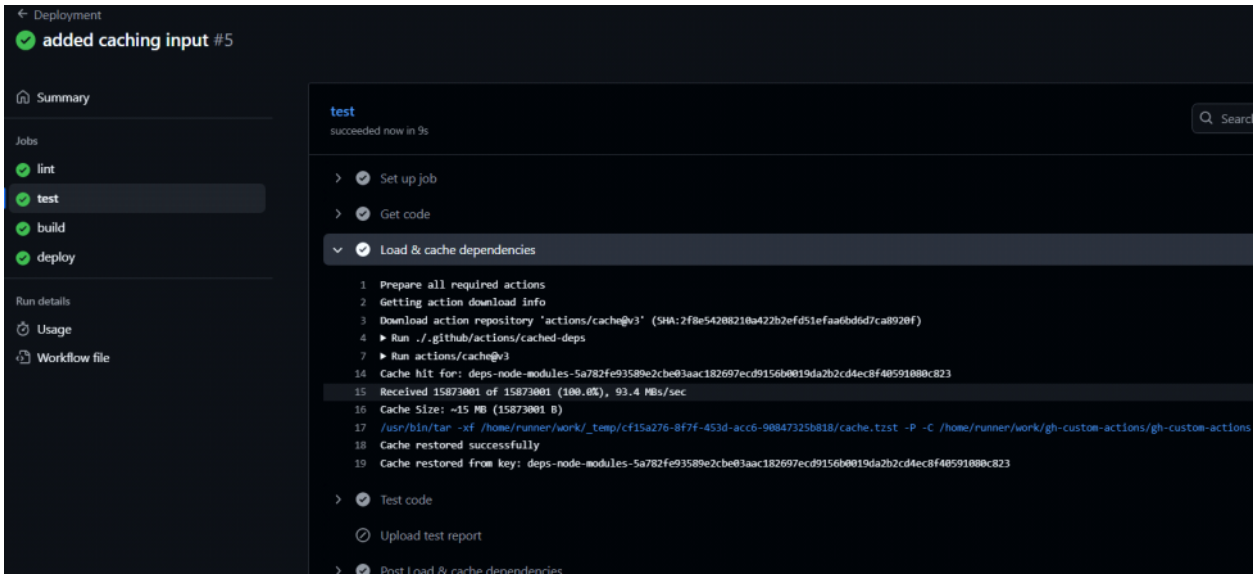
```yaml
- name: Load & cache dependencies
  uses: ./.github/actions/cached-deps
  with:
    caching: 'false' # disables caching for this job/step
```

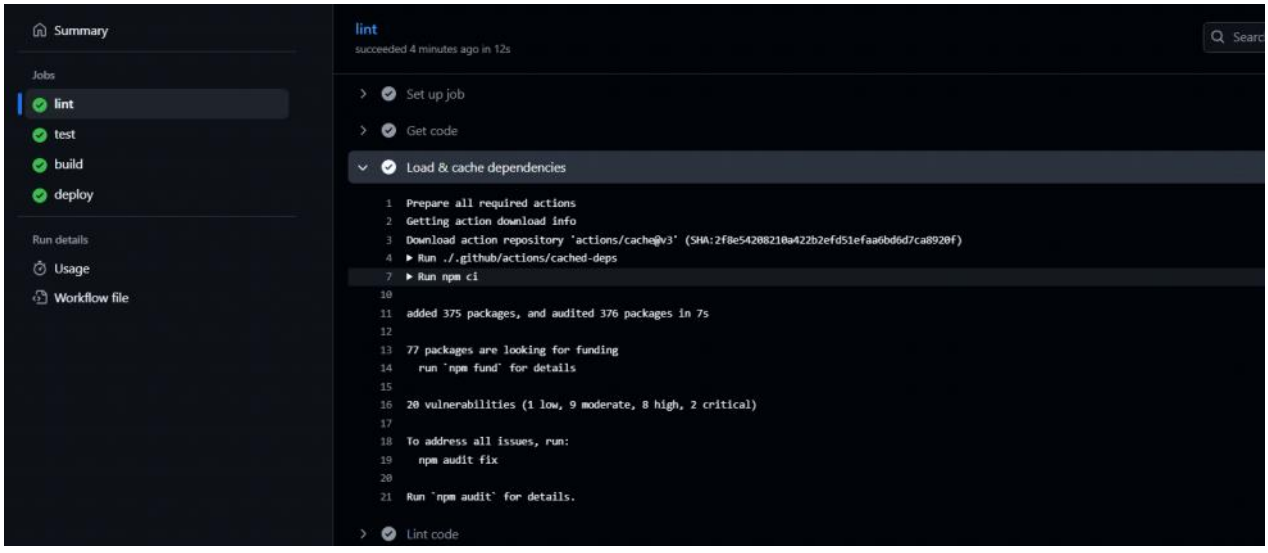
- Other jobs can omit with: and use the default ('true'), enabling caching.

Expected behavior

- For jobs where caching: 'false':
  - Cache step is skipped.
  - Dependencies are installed every run.
- For other jobs:
  - Cache is restored when available; install runs only on cache miss.

This pattern makes the composite action reusable and configurable across different jobs and workflows.





Adding Outputs to a Composite Action

- Custom actions can expose outputs that workflows can read and use in later steps.

Define outputs in action.yml

- Add an outputs section alongside name, description, inputs, and runs.
- For each output:
  - Provide a description.
  - Set value: using a dynamic expression that references a step output.

Example:

```yaml
name: "Get & Cache Dependencies"
description: "Get the dependencies (via npm) and cache them."
inputs:
  caching:
    description: "Whether to cache dependencies or not."
    required: false
    default: "true"
outputs:
```

```
  used-cache:
    description: "Whether the cache was used"
    value: ${{ steps.install.outputs.cache }}
runs:
  using: "composite"
  steps:
    - name: Cache dependencies
      if: inputs.caching == 'true'
      id: cache
      uses: actions/cache@v3
      with:
        path: node_modules
        key: deps-node-modules-${{ hashFiles('**/package-lock.json') }}
    - name: Install dependencies
      id: install
      if: steps.cache.outputs.cache-hit != 'true' || inputs.caching != 'true'
      run: |
        npm ci
        echo "cache='${{ inputs.caching }}'" >> $GITHUB_OUTPUT
      shell: bash
```

- The install step sets a step-level output named cache via $GITHUB_OUTPUT.
- The action's used-cache output maps to steps.install.outputs.cache.

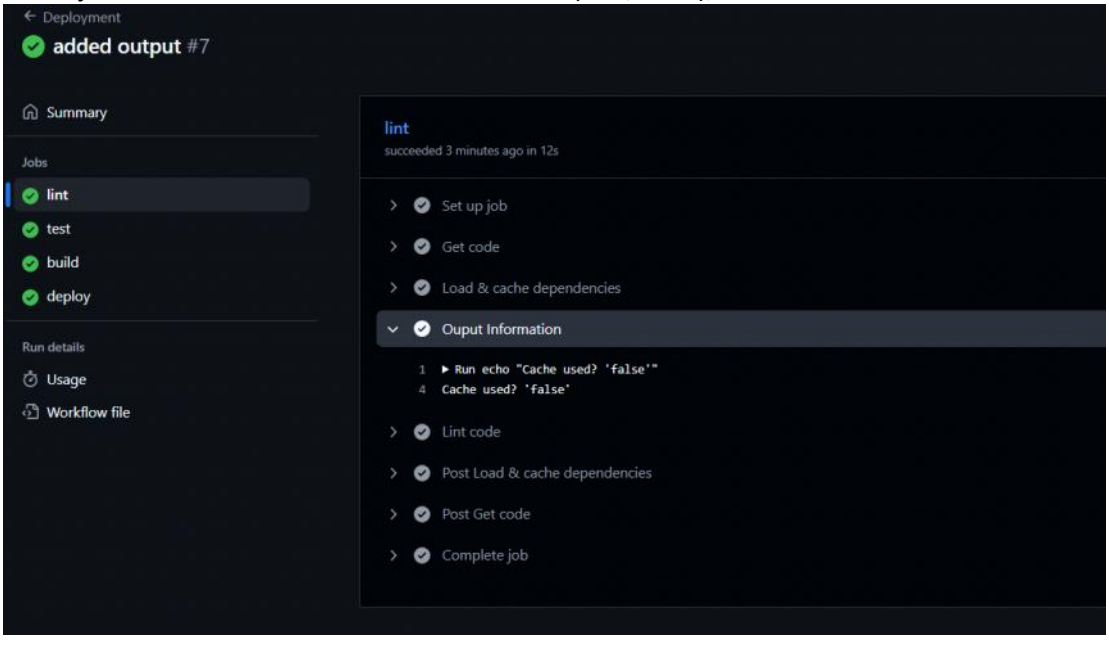
Use the action output in a workflow

- Assign an id to the step that uses the custom action.
- Reference its outputs via steps.<id>.outputs.<output-name>.

Example usage:

```
      - name: Load & cache dependencies
        id: cache-deps
        uses: ./.github/actions/cached-deps
        with:
          caching: 'false'
      - name: Ouput Information
        run: echo "Cache used? ${{ steps.cache-deps.outputs.used-cache }}"
```

- Outcome:
  - The workflow prints the value of the custom action's used-cache output.
  - Other jobs continue to use the action as before (test/build), while lint shows the customized behavior.



Starting a Custom JavaScript Action

- We've built a composite action; next, we'll build a custom JavaScript action.
- JavaScript actions require basic JavaScript/Node.js knowledge, but the example will stay simple and key code will be provided.
- Purpose of the new action: take the built website files and upload them to Amazon S3 (static website hosting on AWS).
    - No AWS account is required to learn the GitHub Actions concepts here.
    - If following along end-to-end, an AWS account (credit card required) will be needed to create an S3 bucket.

Project setup for the JavaScript action

- Location: create a new folder under the repository's actions directory:
    - Path: .github/actions/deploy-S3-javascript
    - This keeps the action local to the current repository (usable only within this repo's workflows).
- Every action—regardless of type—must include an action.yml file.
    - Create action.yml inside .github/actions/deploy-S3-javascript/.
    - This file will define the action's metadata (name/description), inputs/outputs, and how it runs (JavaScript entry file to execute).

What comes next

- Define the action metadata in action.yml.
- Add the JavaScript implementation that will:
    - Read inputs (e.g., bucket name, region, paths, credentials).
    - Upload the built site files to S3.
- Integrate and call this action from the workflow after the build step.

Notes: Define the JavaScript Action Metadata (action.yml)

- Create a new local action directory: .github/actions/deploy-S3-javascript
- Inside it, add action.yml with:
    - name: a concise title for the action
    - description: brief purpose
    - runs: configuration indicating it's a JavaScript (Node) action
    - main: the JavaScript entry file that will execute when the action runs

Example action.yml:

```yaml
name: 'Deploy to AWS S3'
description: 'Deploy a static website via AWS S3.'
runs:
  using: 'node16'
  main: 'main.js'
```

- using: 'node16' declares a JavaScript action running on Node.js 16 (use an appropriate supported Node major version).
- main: 'main.js' points to the entry script located in the same folder as action.yml.
- pre and post scripts are optional and only needed if extra setup/cleanup must run before or after main; not required for this example.

Next step: implement main.js to read inputs, perform the S3 upload, and wire this action into the workflow after the build step.

Notes: Writing and Using a Basic JavaScript Action

- Implement a minimal JavaScript action entry file (main.js):
  - Import GitHub Actions toolkit packages.
  - Define a run function and invoke it.
  - Log a message to the workflow logs.

Example files

1. action metadata (action.yml)

```yaml
name: 'Deploy to AWS S3'
description: 'Deploy a static website via AWS S3.'
runs:
  using: 'node16'
  main: 'main.js'
```

2. action implementation (main.js)

```javascript
const core = require('@actions/core');
const github = require('@actions/github');
const exec = require('@actions/exec');
function run() {
  core.notice('Hello from my custom JavaScript Action!');
}
run();
```

- Initialize the action folder as a Node project and install toolkit deps:
  - From .github/actions/deploy-s3-javascript:
    - npm init -y
    - npm install @actions/core @actions/github @actions/exec
- Important repository considerations:
  - Commit all files needed to execute the action, including node_modules for this local JavaScript action (since runners won't install dependencies for local JS actions automatically).
  - Ensure .gitignore does not exclude required nested files (e.g., dist inside a package).

Using the local JavaScript action in a workflow

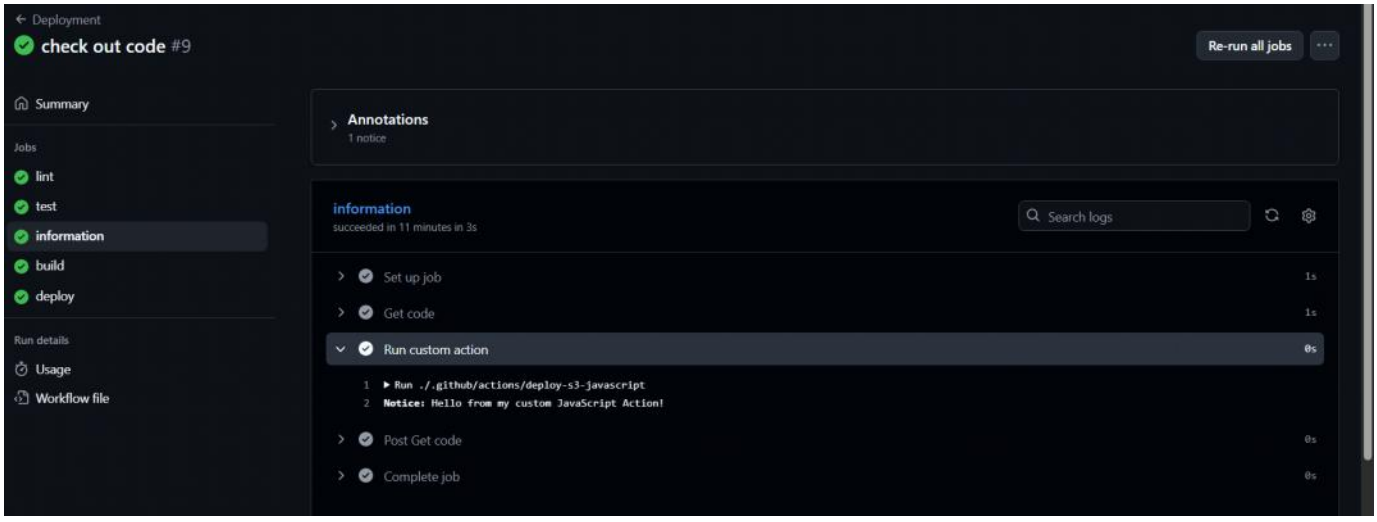
- Add a separate job (runs in parallel with others) and check out the repo before using the local action.
- Reference the action via a path from the repository root.

Example workflow snippet

```yaml
information:
  runs-on: ubuntu-latest
  steps:
    - name: Get code
      uses: actions/checkout@v3
    - name: Run custom action
      uses: ./.github/actions/deploy-s3-javascript
```

- If you see "cannot find action.yml," make sure the job checks out the repository before using the local action.
- If you hit missing module errors, verify required files aren't ignored by .gitignore; adjust rules so nested node_modules (and any needed dist assets) are committed.

Outcome: The job executes the action and prints "Hello from my custom JavaScript Action!" to the logs—confirming the JS action wiring works. Next, expand the action to actually deploy to AWS S3.



Module Summary: Custom Actions

- Custom actions help simplify workflows, remove repeated steps, and solve problems not covered by public Marketplace actions. They can also be shared with the community as open source.

Types of Custom Actions

- Composite actions: bundle multiple workflow steps (mix of run and uses) into a reusable step group.
- JavaScript actions: implement any logic using Node.js and the GitHub Actions Toolkit.
- Docker actions: package logic and its execution environment in a container via a Dockerfile; useful when a specific environment or non-JS language is needed.

Key Capabilities Across Action Types

- Support for inputs and outputs to make actions configurable and chain data between steps.

- Ability to perform arbitrary logic that fits CI/CD needs.

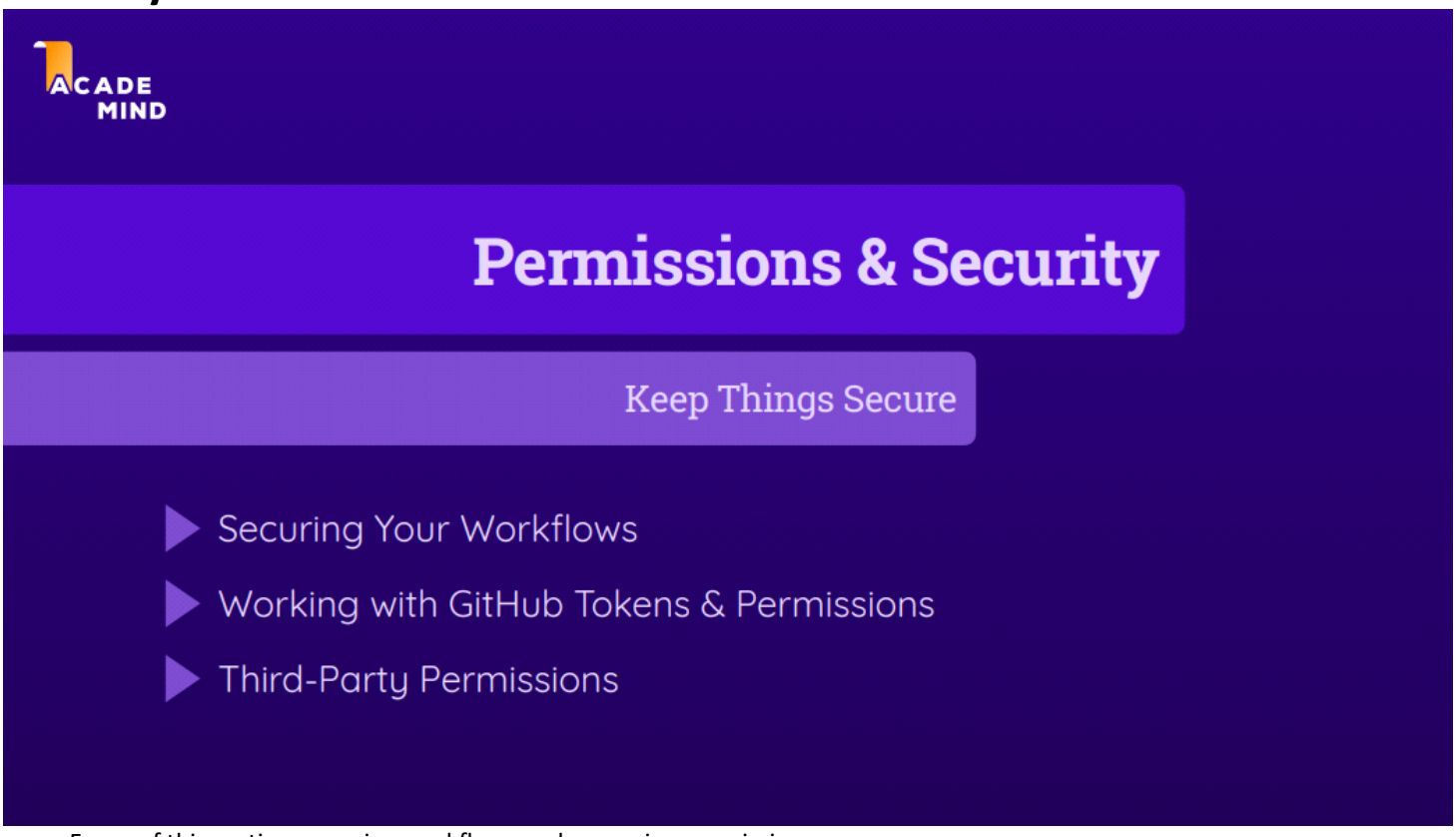
Takeaways from this module

- Start with composite actions to DRY up repeated steps.
- Use JavaScript actions when custom logic in Node.js makes sense.
- Choose Docker actions for non-JS languages or when full control over the environment is required.
- Actions can be published and shared so others can benefit from your solutions.

Security & Permissions

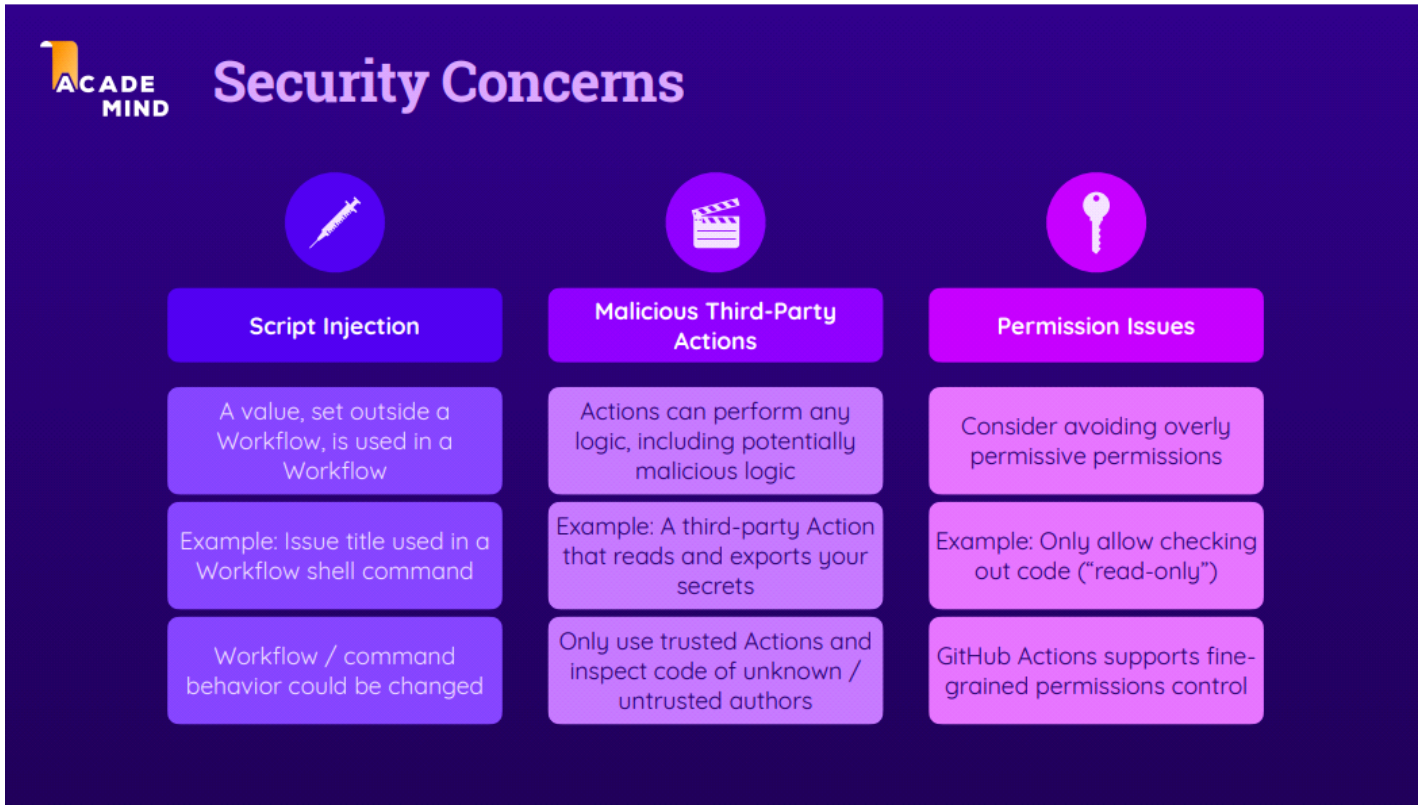
24 August 2025 19:13

Security and Permissions in GitHub Actions



- Focus of this section: securing workflows and managing permissions.
- What will be covered:
  - How to secure workflows and what to consider when doing so.
  - How to configure and manage permissions for workflows.
  - Using GitHub tokens and setting the right permissions for workflows.
  - Managing permissions for third-party platforms or sites that workflows must access.
- Goal: be able to secure workflows, set appropriate permissions, and handle external service access safely.

Workflow Security — Key Topics



- Scope of this section
  - Overview of potential problems and topics to be aware of in workflow security.
  - Includes a lecture with further reading materials, official docs, and blog posts for deeper exploration.
- Topics covered in this section
  - Script Injection
  - Malicious Third-Party Actions
  - Permissions and related issues

Script Injection

- Concept: A value set outside a workflow is used inside the workflow.
- Example scenario: A workflow runs on new issues; the issue title is used in the workflow. Someone could create an issue whose title injects code into the workflow, potentially breaking it or causing harm.
- Note: A concrete example is covered in the next lecture.

Malicious Third-Party Actions

- Actions can perform any logic, including harmful logic (e.g., code that steals secrets).
- Mitigation approach:
  - Use only trusted actions.
  - Inspect the code of actions from unknown or untrusted authors.
- This topic will be revisited later in the section.

Permissions

- Goal: Avoid overly permissive permissions.
- Reason: If script injection or other bad behavior occurs, restrictive permissions can still prevent damage.
- Example: A workflow that only checks out code and runs tests likely needs read-only permissions to read code and does not need permissions to add issues or similar operations.
- GitHub Actions supports fine-grained permissions; configuration details will be covered later in the section.
- Closing note
  - These are selected aspects to be aware of.
  - The end of the section includes links for further reading on additional security problems and official recommendations.

Script Injection in GitHub Actions (Example and Mitigation)

- Focus: Show how a workflow can be affected by script injection and how to defend against it.

Vulnerable workflow example

- Workflow triggers on new issues.
- Step stores the issue title in a shell variable, then checks if it contains “bug”.
- Works for normal titles, e.g., “something is wrong” → outputs “Issue is not about a bug”.

```
name: Label Issues (Script Injection Example)
on:
  issues:
    types:
      - opened
jobs:
  assign-label:
    runs-on: ubuntu-latest
    steps:
      - name: Assign label
        run: |
          issue_title="${{ github.event.issue.title }}"
          if [[ "$issue_title" == *"bug"* ]]; then
          echo "Issue is about a bug!"
          else
          echo "Issue is not about a bug"
          fi
```

- Injection demo:
  - Create an issue with title: A"; echo got your secrets"
  - Result: The injected echo command runs in the workflow logs.
  - Why it works: The title is interpolated into the shell script before execution, breaking the quoting and injecting a new command.
- Impact:
  - An attacker could exfiltrate secrets or run arbitrary commands (e.g., curl to send environment variables, manipulate repo via API).

How to defend

- Prefer using an action instead of inline shell when possible (inputs are handled and not executed in a shell).
- If using shell, assign user-controlled values to environment variables and reference them with native shell syntax.

```yaml
name: Label Issues (Script Injection Example)
on:
  issues:
    types:
      - opened
jobs:
  assign-label:
    runs-on: ubuntu-latest
    steps:
      - name: Assign label
        env:
          TITLE: ${{ github.event.issue.title }}
        run: |
          if [[ "$TITLE" == *"bug"* ]]; then
          echo "Issue is about a bug!"
          else
          echo "Issue is not about a bug"
          fi
```

- Result after fix:
  - The same malicious title no longer executes the injected command.
  - Example log shows the TITLE environment variable value and the safe branch output:
    - TITLE: a";echo Got your secrets"
    - “Issue is not about a bug”

Key takeaways

- Script injection occurs when untrusted data (e.g., issue titles) is interpolated into shell scripts executed by workflows.
- Mitigate by:
  - Using purpose-built actions where feasible.
  - Storing untrusted input in environment variables and referencing them with shell-native syntax instead of embedding expressions directly in run scripts.

Using Actions Securely

- Malicious third-party actions are a potential problem; actions can perform any logic, including harmful behavior.

Security options for using actions

- Highest security: only use your own actions
  - Pros: Full knowledge of what actions do; no unknown code.
  - Cons: Considerable effort to write and maintain everything.
- Use actions by verified creators
  - Marketplace shows a blue check mark after the creator name.
  - Example: upload-artifact by a verified creator (e.g., GitHub team).
  - Benefit: Creator was manually approved by GitHub, so generally trustworthy.
  - Caveat: Still not a 100% guarantee; accounts or codebases could be compromised.
- Use all public actions
  - Broadest functionality but lowest security.
  - GitHub does not monitor or validate actions when they're published.
  - Any action could theoretically do anything.

Practical guidance

- When using actions from non-verified creators or from people you don't know:
  - Analyze the action's code first by visiting its repository; all actions are open source.

- This extra review effort lets you go beyond only your own actions while keeping a relatively high security level.

Permissions in GitHub Actions (Restricting What Jobs Can Do)

- Purpose: Control what a workflow/job is allowed to do (issues, pull requests, contents, actions, etc.) instead of giving full permissions by default.

Example workflow behavior

- Triggers on new issues.
- Checks the issue title safely with an if expression (no shell injection).
- If title contains "bug", sends a POST request to the GitHub API to add the "bug" label to the issue.
- Uses an authorization header with the GitHub token to authenticate.

Why set permissions

- Least privilege: the job only needs permission to write to issues.
- Limits blast radius if a script injection or a malicious action occurs.
- Prevents unintended access (e.g., reading repository contents, changing actions settings).

How permissions are applied

- Permissions can be set:
  - At the workflow level (applies to all jobs), or
  - At the job level (applies to that job only).
- If any permissions are specified, unspecified scopes default to none for that job/workflow.

Example: restrict to issues:write

```yaml
name: Label Issues (Permissions Example)
on:
  issues:
    types:
      - opened
jobs:
  assign-label:
    permissions:
      issues: write
    runs-on: ubuntu-latest
    steps:
      - name: Assign label
        if: contains(github.event.issue.title, 'bug')
        run: |
          curl -X POST \
          --url https://api.github.com/repos/${{ github.repository }}/issues/
${{ github.event.issue.number }}/labels \
          -H 'authorization: Bearer ${{ secrets.GITHUB_TOKEN }}' \
          -H 'content-type: application/json' \
          -d '{
              "labels": ["bug"]
            }' \
          --fail
```

- Result:
  - Job can write issue labels.
  - Job cannot read/modify other areas (e.g., repository contents, actions).
  - Behavior remains the same functionally, but with tighter security.

GitHub Token and Permissions Behavior

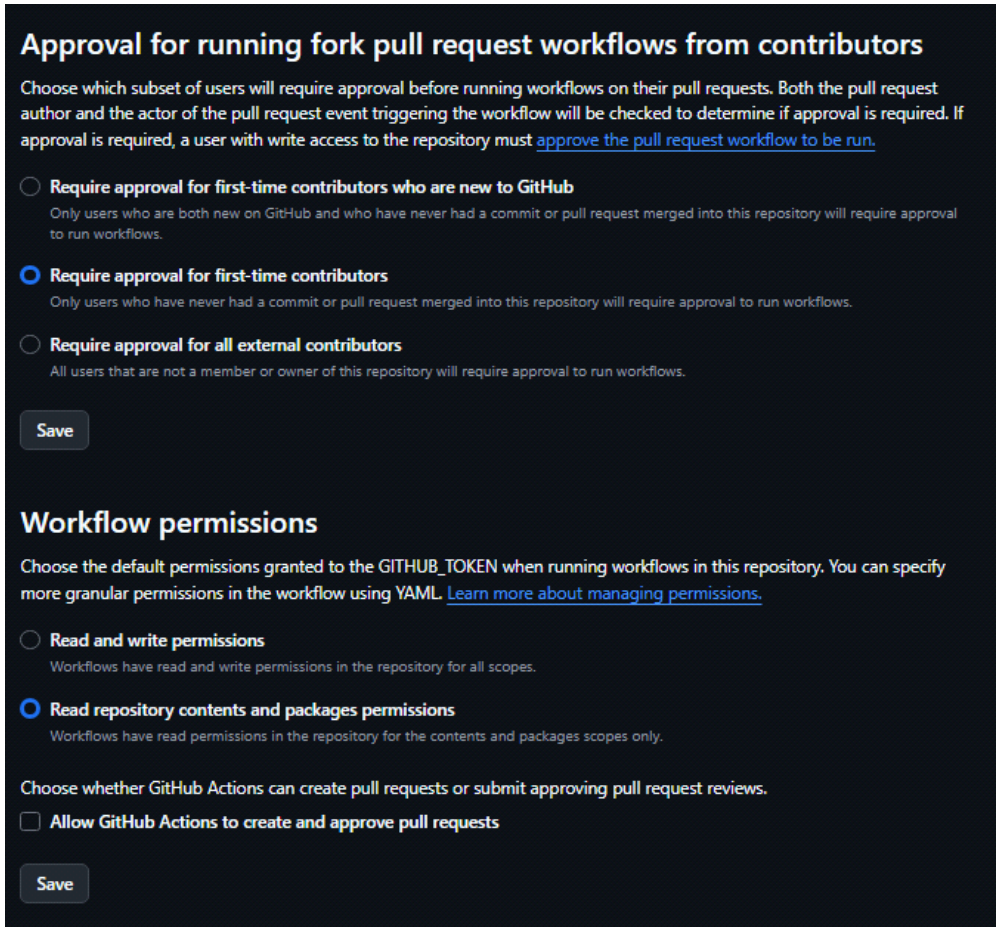
- The token used:
  - The special secret referenced in the workflow is an automatically generated short-lived token provided by GitHub for each job run.
  - It authenticates requests to GitHub's API and expires after the job finishes.
- Relationship to permissions:
  - The token's access is scoped by the permissions configured in the workflow/job.

    - Setting permissions to read only (e.g., issues: read) prevents write actions like adding labels.
    - Restoring issues: write enables the POST request to succeed.
- Implications:
    - Even when not referenced explicitly before, many actions (e.g., checkout) implicitly use this token under the hood to interact with GitHub APIs.
    - If permissions are overly restrictive (e.g., only issues: read), steps that need other scopes (like reading repository contents) will fail.
- Example (read-only fails):

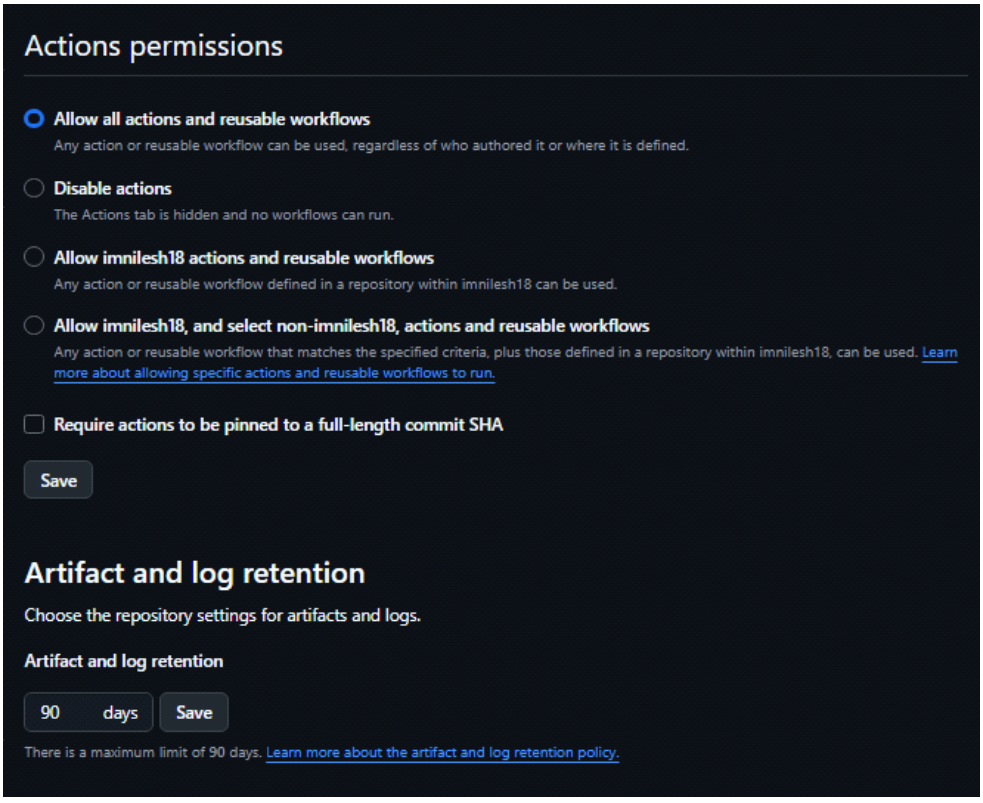
```
name: Label Issues (Permissions Example)
on:
  issues:
    types:
      - opened
jobs:
  assign-label:
    permissions:
      issues: read
    runs-on: ubuntu-latest
    steps:
      - name: Assign label
        if: contains(github.event.issue.title, 'bug')
        run: |
          curl -X POST \
          --url https://api.github.com/repos/${{ github.repository }}/issues/
${{ github.event.issue.number }}/labels \
          -H 'authorization: Bearer ${{ secrets.GITHUB_TOKEN }}' \
          -H 'content-type: application/json' \
          -d '{
              "labels": ["bug"]
            }' \
          --fail
```

- Result: The POST fails because the token's permissions don't allow writing to issues.
- Takeaway:
    - Use least-privilege permissions and adjust per job. Grant only what a job needs so the token can perform required actions and nothing more.

Repository Settings That Affect Workflow Security



- Pull request capabilities
  - Option to allow GitHub Actions to automatically create or approve pull requests.
  - Risk: a bug in a workflow could accidentally approve a PR.
  - Safer choice: disable so only humans can create/approve PRs.
  - If highly automated and confident in workflows, enabling can support auto-approve/merge workflows.
- Fork pull request approvals
  - Control approval requirements for PRs from forks:
    - Require approval for first-time contributors new to GitHub.
    - Require approval for all external contributors.
  - Reason: forks of public repos can submit malicious changes; approval prevents automatic execution of potentially harmful workflows.



- Allowable actions scope
  - Configure which actions can be used in the repository:
    - Allow all actions and reusable workflows.
    - Allow only actions from the account/organization.
    - Disable actions entirely.
  - Trade-off: broader allowance increases capability but also risk; restricting aligns with “use actions securely.”
- Takeaway
  - Review and set these settings to match the repository’s risk tolerance:
    - PR creation/approval by workflows
    - Approval policy for forked PRs
    - Which actions are permitted (and whether to disable actions)