DevOps Shack

GitHub Actions Notes

1. Introduction to GitHub Actions

GitHub Actions enables developers to automate their workflows and manage the complete CI/CD process directly within GitHub. It helps streamline code testing, building, and deployment.

What is GitHub Actions?

GitHub Actions is a flexible automation tool that allows you to execute scripts, run commands, and perform tasks on every code push, pull request, or manual trigger. It supports native CI/CD pipelines and can be extended with thousands of reusable actions available on the GitHub Marketplace.

GitHub Actions integrates with core GitHub services like **Pull Requests**, **Issues**, and **Checks**, making it a powerful tool for developers who use GitHub repositories for managing their code.

Key Terminologies and Concepts

- **Workflows:** A workflow is a collection of jobs that run sequentially or in parallel. Workflows are defined in a YAML file and are triggered by events.
- **Jobs:** A job is a set of steps that execute on the same runner. Jobs can run independently or depend on other jobs.
- **Steps:** Steps are the individual tasks that run within a job, such as executing commands or actions.
- **Actions:** Actions are reusable components in workflows. You can use pre-built actions from the GitHub Marketplace or create your own custom actions.
- **Runners:** Runners are the servers that execute workflows. GitHub provides both hosted and self-hosted runners.

Benefits of GitHub Actions

GitHub Actions brings several benefits to the development process:

- **Native Integration with GitHub:** All workflows and pipeline management happen directly within the repository.
- **Wide Range of Pre-Built Actions:** Thousands of pre-built actions are available to speed up automation and integration tasks.
- **Multi-platform Support:** You can run workflows on Ubuntu, macOS, and Windows environments.

- **Scalability:** GitHub-hosted runners scale automatically based on your workload.
- **Flexibility:** You can create complex workflows with parallel and sequential jobs, condition-based steps, and reusable actions.

---

2. Getting Started with GitHub Actions

Setting up GitHub Actions in Your Repository

GitHub Actions is built into GitHub, and you can set up actions by simply creating a .github/workflows directory in your repository and adding YAML files to define the workflows. Here's how to get started:

1. Go to the **Actions** tab of your GitHub repository.
2. Click **New Workflow** to get started with a pre-configured template or create your custom workflow.
3. Create a .yaml file in .github/workflows in your repository.

```
name: CI Pipeline

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Setup Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '14'
      - run: npm install
      - run: npm test
```

In this example, the workflow triggers on every push or pull request to the main branch. It checks out the code, sets up a Node.js environment, installs dependencies, and runs tests.

Understanding Workflow Triggers

Workflows are triggered by events, which are defined under the on keyword. The most common events are:

- **push**: Triggers the workflow when code is pushed to the repository.
- **pull_request**: Triggers the workflow when a pull request is opened or updated.
- **schedule**: Defines a schedule using cron syntax for periodic executions.

- **workflow_dispatch**: Allows manual triggering from the GitHub Actions UI.

---

3. Deep Dive into Workflows

Workflow Structure and Syntax

A GitHub Actions workflow is defined using a YAML structure. The key components of a workflow include:

- **Name**: Optional name for the workflow.
- **on**: Events that trigger the workflow (push, pull_request, schedule, etc.).
- **jobs**: Defines one or more jobs within the workflow. Each job contains steps that run on a runner.
- **steps**: Defines the individual tasks in each job. These can be commands or reusable actions.

Example:

```
name: My Workflow

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Setup Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '14'
      - run: npm install
      - run: npm test
```

Multi-Job Workflows

GitHub Actions supports multiple jobs running either sequentially or in parallel. Jobs can also be configured with dependencies.

Example:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Building the project"

  test:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - run: echo "Running tests"
```

In this case, the test job depends on the build job and will only execute after build completes successfully.

Workflow Examples

To cater to different levels of expertise, we can expand workflow examples for **beginner**, **intermediate**, and **advanced** use cases.

---

4. Jobs, Steps, and Actions

Jobs: What Are They?

A **job** is a set of steps that runs in the same environment on a runner. Jobs are independent, and workflows can contain multiple jobs that either run in parallel or have dependencies.

Steps: Building Blocks of Workflows

A **step** is an individual task in a job, such as running a shell command or executing a pre-built action. Steps run sequentially within a job.

Example of steps:

```
steps:
  - run: echo "Running the first step"
  - uses: actions/checkout@v2
  - run: npm install
  - run: npm test
```

Using Pre-built Actions

GitHub Marketplace provides thousands of pre-built actions, such as checking out a repository or setting up specific programming languages. These can be integrated into your workflows using the uses keyword.

Example:

```
steps:
  - uses: actions/checkout@v2
  - uses: actions/setup-node@v2
    with:
      node-version: '16'
```

---

5. Advanced Workflow Features

Matrix Builds for Multi-Configuration Testing

Matrix builds allow you to run jobs with multiple configurations. This is useful for testing across different operating systems, software versions, or environments.

Example:

```
jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        node-version: [12, 14, 16]
    steps:
      - uses: actions/checkout@v2
      - name: Setup Node.js
        uses: actions/setup-node@v2
        with:
          node-version: ${{ matrix.node-version }}
      - run: npm install
      - run: npm test
```

In this case, the workflow tests across three operating systems and three Node.js versions.

Parallel and Sequential Job Execution

By default, jobs in GitHub Actions run in parallel. However, you can configure jobs to run sequentially by defining dependencies using the needs keyword.

6. Runners in GitHub Actions

GitHub-hosted vs. Self-hosted Runners

Runners are the virtual machines that execute the workflows defined in GitHub Actions. GitHub provides two types of runners:

- **GitHub-hosted Runners:** These are virtual machines managed and maintained by GitHub. They come pre-installed with popular software tools and libraries, making it easier to get started with minimal setup. You can choose the operating system (Ubuntu, macOS, or Windows), and GitHub manages the lifecycle of the runners.
- **Self-hosted Runners:** If you need more control over the runner environment or require custom hardware/software, you can configure self-hosted runners. This allows you to run your workflows on your infrastructure, such as bare metal servers, virtual machines, or cloud instances.

GitHub-hosted Runners

GitHub-hosted runners provide a pre-configured environment with many popular tools like Docker, Node.js, Python, Java, and more, which means you can run workflows without worrying about

dependencies. GitHub-hosted runners automatically scale based on workload and provide different environments:

- **Ubuntu:** ubuntu-latest
- **Windows:** windows-latest
- **macOS:** macos-latest

Example of using a GitHub-hosted runner:

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

Self-hosted Runners

Self-hosted runners offer more flexibility by allowing you to configure the environment. You can control which tools and dependencies are available, and they are particularly useful for specialized tasks, long-running jobs, or workflows that require hardware resources not provided by GitHub-hosted runners.

1. **Setting Up a Self-hosted Runner:** To set up a self-hosted runner, follow these steps:
    - Navigate to the **Settings** tab of your repository.
    - Under **Actions**, click **Runners**, then click **Add self-hosted runner**.
    - Follow the instructions to download and configure the runner on your machine.
2. **Using a Self-hosted Runner in a Workflow:** Example:

```
jobs:
  build:
    runs-on: self-hosted
    steps:
      - uses: actions/checkout@v2
      - run: ./build.sh
```

Managing Runners

You can manage runners at three levels: repository, organization, or enterprise. GitHub provides a **Runner API** to list, add, remove, and update runners programmatically, offering flexibility in managing self-hosted environments.

---

7. Using Caches and Artifacts

Caching Dependencies

GitHub Actions supports caching to speed up workflow execution by storing dependencies or other files. Caching is especially useful in CI/CD pipelines where you repeatedly install the same dependencies, such as in Node.js or Python applications.

1. **Setting Up Cache:** Example of caching node_modules:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Cache Node.js modules
        uses: actions/cache@v2
        with:
          path: node_modules
          key: ${{ runner.os }}-node-${{ hashFiles('package-lock.json') }}
          restore-keys: |
            ${{ runner.os }}-node-
      - run: npm install
      - run: npm test
```

In this example, the cache is restored based on the package-lock.json hash. If the dependencies haven't changed, they are restored from the cache, speeding up the workflow.

Uploading and Downloading Artifacts

Artifacts in GitHub Actions allow you to persist data generated during a workflow, such as build outputs or test results. Artifacts can be uploaded during the workflow and downloaded later.

1. **Uploading Artifacts:** Example:

```
steps:
  - name: Upload build output
    uses: actions/upload-artifact@v2
    with:
      name: build-artifact
      path: ./build
```

2. **Downloading Artifacts:** Example:

```
steps:
  - name: Download build artifact
    uses: actions/download-artifact@v2
    with:
      name: build-artifact
```

Using caching and artifacts helps optimize workflow performance and allows teams to share build results across jobs and workflows.

---

8. Secrets and Environment Variables

Managing Secrets Securely

GitHub provides a secure way to manage sensitive data such as API keys, passwords, and access tokens through the **Secrets** feature. Secrets are encrypted and stored securely, and you can reference them in your workflows.

1. **Storing Secrets:** To store secrets, navigate to the **Settings** tab of your repository, and under **Secrets**, click **New repository secret**. Add the name and value for the secret.

2. **Using Secrets in Workflows:** Example:

```
steps:
  - name: Deploy to Production
    run: deploy --token ${{ secrets.DEPLOY_TOKEN }}
```

In this example, the secret DEPLOY_TOKEN is used securely within the workflow.

Using Environment Variables

Environment variables can be defined globally for a workflow or specific to individual jobs and steps.

1. **Defining Environment Variables at the Workflow Level:**

```
jobs:
  build:
    runs-on: ubuntu-latest
    env:
      NODE_ENV: production
    steps:
      - run: npm install
```

2. **Defining Environment Variables at the Job or Step Level:** You can set environment variables at different levels depending on the scope.

---

9. CI/CD Pipelines with GitHub Actions

Building a CI/CD Pipeline for Node.js Applications

A typical CI/CD pipeline for a Node.js application involves several steps: testing, building, and deployment.

1. **Testing:**

```
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-node@v2
        with:
          node-version: '14'
      - run: npm install
      - run: npm test
```

2. **Building:** After testing, the next step is building the project (e.g., bundling assets for a React app):

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-node@v2
        with:
          node-version: '14'
```

```
    - run: npm install
    - run: npm run build
```

3. **Deploying to AWS S3:** Deployment can be integrated into the pipeline using AWS CLI:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Deploy to S3
        uses: aws-actions/s3-sync-action@v0.5.0
        with:
          args: --acl public-read
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

Deploying to Kubernetes

GitHub Actions can also be used to deploy applications to a Kubernetes cluster. Here's a simplified pipeline for deploying a Dockerized application to a Kubernetes cluster:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Kubernetes
        uses: Azure/setup-kubectl@v1
      - name: Deploy to AKS
        run: |
          kubectl apply -f deployment.yaml
          kubectl apply -f service.yaml
```

In this example, kubectl is used to deploy Kubernetes manifests for the application.

---

11. Security Best Practices for GitHub Actions

Secure Token Management

Always use GitHub's secrets to manage sensitive information. Never hardcode sensitive values like API keys or tokens in your workflow files.

Using Branch Protection Rules

To enforce security in your repositories, configure branch protection rules that ensure workflows must pass before merging pull requests. You can also restrict who can push to protected branches.

Securing Workflows from Untrusted Code

For public repositories, make sure to use pull_request_target events with caution. Malicious actors can exploit workflows by submitting pull requests with unsafe code. Always validate code before running workflows triggered by external contributions.

---

12. Debugging and Monitoring Workflows

Viewing Logs and Troubleshooting Errors

Every workflow run in GitHub Actions generates detailed logs, allowing you to see each step's output and errors. You can increase the verbosity of logs by enabling **debug mode** using ACTIONS_STEP_DEBUG.

Using Debugging Tools

You can use the set-output and echo commands to print variables, outputs, and debugging information during a workflow.

Example of setting a debug message:

```
steps:
  - run: echo "::debug::This is a debug message"
```