# GitHub Actions: Comprehensive Notes

**GitHub Actions** is a powerful automation tool that allows you to automate tasks within your software development lifecycle, such as continuous integration (CI), continuous delivery (CD), testing, and deployment. It helps you build, test, and deploy code directly from your GitHub repositories.

## 1. What is GitHub Actions?

- **GitHub Actions** enables you to define custom workflows that automate specific actions based on events in your repository.
- It integrates directly with GitHub and can automate everything from code testing to deployment.
- Workflows are defined in **YAML** files and can be triggered by GitHub events like commits, pull requests, and merges.

## 2. Key Concepts in GitHub Actions

### 2.1 Workflows:

- A **workflow** is an automated process made up of one or more jobs that run in response to a specific event.
- Each repository can have multiple workflows, and they are defined in `.github/workflows/` directory.

  Example:

  ```
  name: CI Workflow
  on: [push]

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

### 2.2 Events:

- Events are the triggers that start a workflow, such as `push`, `pull_request`, `release`, or scheduled events (`cron`).

  Common events:

  - `push`: Triggered when code is pushed to the repository.
  - `pull_request`: Triggered when a pull request is opened, updated, or merged.
  - `schedule`: Runs on a regular schedule using a cron-like syntax.

#### 2.3 Jobs:

- A **job** is a collection of steps that execute on the same runner. Jobs run in parallel by default but can be configured to run sequentially.

#### 2.4 Steps:

- **Steps** are the individual tasks within a job. Each step runs a command or uses an action to accomplish a specific task (like running tests, building code, etc.).

#### 2.5 Actions:

- **Actions** are reusable pieces of code that automate common tasks like checking out code, setting up environments, or deploying applications.
- GitHub provides a marketplace for predefined actions, or you can create custom actions.

#### 2.6 Runners:

- A **runner** is a server that runs the jobs in a workflow. GitHub provides hosted runners (e.g., `ubuntu-latest`, `macos-latest`) or you can set up self-hosted runners.

#### 2.7 Artifacts:

- Artifacts are files generated during a workflow run (e.g., logs, build files, test results) that can be saved or shared between jobs.

## 3. Creating a Workflow

- Workflows are defined using YAML configuration files stored in the `.github/workflows/` directory in the repository.
- You can have multiple workflows for different purposes like testing, building, or deploying.

Example of a basic workflow:

```
name: CI Pipeline

on: [push, pull_request]

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

**Explanation:**

- **`name`**: Specifies the name of the workflow.
- **`on`**: Defines events that trigger the workflow. In this case, it runs on `push` and `pull_request`.
- **`jobs`**: A section that defines one or more jobs to run in the workflow.
- **`runs-on`**: Defines the environment for running the job (in this case, `ubuntu-latest`).
- **`steps`**: The tasks the job will execute, such as checking out the code, setting up Node.js, installing dependencies, and running tests.

## 4. Triggers for Workflows

### 4.1 on: push

- Triggers the workflow whenever code is pushed to the repository or a specific branch.

  Example:

```
on:
  push:
    branches:
      - main
```

### 4.2 on: pull_request

- Triggers the workflow whenever a pull request is opened or updated.

  Example:

```
on:
  pull_request:
    branches:
      - develop
```

### 4.3 Scheduled (Cron Jobs)

- You can schedule workflows to run at specific intervals using cron syntax.

  Example:

```
on:
  schedule:
    - cron: '0 0 * * *' # Runs every day at midnight
```

## 5. Types of Actions

### 5.1 Predefined Actions:

- GitHub Actions Marketplace provides a vast collection of predefined actions that you can use directly in your workflows.

  Example:

```
steps:
  - name: Checkout code
    uses: actions/checkout@v2

  - name: Set up Python
    uses: actions/setup-python@v2
    with:
      python-version: '3.x'
```

### 5.2 Custom Actions:

- You can create custom actions using JavaScript or Docker, allowing you to define more complex or specific tasks.

## 6. Environment Variables and Secrets

- You can pass **environment variables** and **secrets** into your workflows to securely handle sensitive data (e.g., API keys, tokens).

  Example:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to production
        run: ./deploy.sh
        env:
          API_KEY: ${{ secrets.API_KEY }}
```

- **`secrets`**: GitHub allows you to securely store sensitive information using repository secrets, which can be accessed using `${{ secrets.<SECRET_NAME> }}`.

## 7. Matrix Builds

Matrix builds allow you to run the same job across multiple combinations of environments, such as different versions of Node.js, Python, or operating systems.

Example:

```
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        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 example, the tests will be run across Node.js versions 12, 14, and 16.

## 8. Error Handling and Failures

GitHub Actions provides several ways to handle errors and failures in workflows:

- **Failing a Step**: If a command fails, the step and job will fail unless handled.

  Example:

  ```
  - run: npm test
    continue-on-error: true
  ```

- **Using `if` conditions**: You can control the execution of steps based on conditions.

  Example:

  ```
  - name: Run tests only on Linux
    if: runner.os == 'Linux'
    run: npm test
  ```

- **Post-job actions**: You can define steps that should run even if the job fails, using `always()` or `failure()` conditions.

## 9. Artifacts and Caching

- **Artifacts**: You can store and upload files from a workflow to use in later jobs or to download after the workflow completes.

  Example:

  ```
  - name: Upload build artifacts
    uses: actions/upload-artifact@v2
    with:
      name: build-files
      path: build/
  ```

- **Caching**: You can cache dependencies or other data to speed up workflow execution.

  Example:

  ```
  - name: Cache node modules
    uses: actions/cache@v2
    with:
      path: node_modules
      key: ${{ runner.os }}-node-${{ hashFiles('package-lock.json') }}
  ```

## 10. GitHub Actions Marketplace

- GitHub has a **Marketplace** for reusable actions created by GitHub and the community. You can search for actions to easily integrate into your workflows, such as testing frameworks, CI/CD tools, and deployment scripts.

## 11. Best Practices for GitHub Actions

- **Keep Workflows Small**: Break down large workflows into smaller, independent jobs.
- **Use Secrets for Sensitive Data**: Always store sensitive information in GitHub Secrets.
- **Leverage Caching**: Cache dependencies to improve workflow speed.
- **Test Locally**: Test your scripts and commands locally before adding them to your workflows.
- **Use `actions/checkout`**: Always include the `actions/checkout@v2` step to pull your repository code when running a workflow.

## 12. Conclusion

GitHub Actions is a powerful CI/CD tool that is natively integrated with GitHub. It provides an efficient way to automate software development workflows, from testing to deployment. By using workflows, jobs, steps, and actions, you can automate any task in your development lifecycle with flexibility and efficiency.

## Continuous Deployment (CD) Workflow Using GitHub Actions

Below is an example of a GitHub Actions workflow file that implements Continuous Deployment (CD). This workflow automatically deploys the application to an AWS Elastic Beanstalk environment whenever code is pushed to the `main` branch. I'll provide the workflow file and explain each step in detail.

## GitHub Actions Workflow File (`.github/workflows/deploy.yml`)

```
name: CD Pipeline


# Trigger the workflow on push events to the 'main' branch

on:

  push:

    branches:

      - main


# Define environment variables used in the workflow

env:

  AWS_REGION: "us-west-2"

  AWS_S3_BUCKET: "my-app-bucket"

  APPLICATION_NAME: "MyApp"

  ENVIRONMENT_NAME: "MyApp-env"


jobs:

  build:

    runs-on: ubuntu-latest


    steps:
```

```
# Step 1: Check out the code from the repository
- name: Checkout code
  uses: actions/checkout@v3

# Step 2: Set up Node.js environment for building the application
- name: Set up Node.js
  uses: actions/setup-node@v3
  with:
    node-version: '14'

# Step 3: Install application dependencies
- name: Install Dependencies
  run: npm install

# Step 4: Build the application (for a React/Node.js app)
- name: Build Application
  run: npm run build

# Step 5: Archive build artifacts
- name: Archive Production Artifacts
  run: zip -r my-app.zip build

# Step 6: Upload artifacts to Amazon S3
- name: Upload to S3
  run: |
        aws s3 cp my-app.zip s3://$AWS_S3_BUCKET/my-app.zip
      env:
        AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
        AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}

    # Step 7: Deploy the build to AWS Elastic Beanstalk
    - name: Deploy to Elastic Beanstalk
      run: |
        aws elasticbeanstalk create-application-version --application-name $APPLICATION_NAME --
version-label $GITHUB_SHA --source-bundle S3Bucket=$AWS_S3_BUCKET,S3Key=my-app.zip
        aws elasticbeanstalk update-environment --environment-name $ENVIRONMENT_NAME --version-
label $GITHUB_SHA
      env:
        AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
        AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

## Explanation of Each Step

### 1. Workflow Name and Trigger

```
name: CD Pipeline
```

- The `name` attribute defines the name of the workflow. It’s displayed in the Actions tab of the repository.

```
on:
  push:
    branches:
      - main
```

- The `on` section specifies that this workflow should be triggered every time a `push` event occurs on the `main` branch. You can modify this to trigger on other branches or events (e.g., `pull_request`).

## 2. Environment Variables

```
env:
  AWS_REGION: "us-west-2"
  AWS_S3_BUCKET: "my-app-bucket"
  APPLICATION_NAME: "MyApp"
  ENVIRONMENT_NAME: "MyApp-env"
```

- The `env` section defines environment variables used throughout the workflow. These values are referenced in the deployment steps, making it easier to modify configuration details like AWS region, S3 bucket name, or Elastic Beanstalk environment.

## 3. Job Configuration

```
jobs:
  build:
    runs-on: ubuntu-latest
```

- The `jobs` section defines the jobs in the workflow. In this example, there is a single job named `build`.
- `runs-on` specifies the runner (virtual machine) to execute the job. Here, `ubuntu-latest` is used, which provides a Linux environment with common tools pre-installed.

## 4. Checkout the Code

```
steps:
  - name: Checkout code
    uses: actions/checkout@v3
```

- This step checks out (clones) the repository's code so that subsequent steps have access to the codebase.
- `uses: actions/checkout@v3` is a predefined action that checks out the code in the current GitHub repository.

## 5. Set Up Node.js

```
- name: Set up Node.js
  uses: actions/setup-node@v3
  with:
    node-version: '14'
```

- This step sets up the Node.js environment using the `setup-node` action, which is necessary for building Node.js applications.
- The `node-version` specifies which version of Node.js to use. Here, it's set to version `14`.

### 6. Install Dependencies

```
- name: Install Dependencies
  run: npm install
```

- The `run` command installs the dependencies specified in the `package.json` file using `npm install`. This step is required to ensure all necessary packages are available for building the application.

### 7. Build the Application

```
- name: Build Application
  run: npm run build
```

- This step builds the application using the `npm run build` command. The actual build command may vary depending on your project's configuration.
- For React applications, this command generates production-ready static files in the `build` directory.

### 8. Archive Production Artifacts

```
- name: Archive Production Artifacts
  run: zip -r my-app.zip build
```

- This step archives the `build` directory into a ZIP file named `my-app.zip`. This ZIP file will later be uploaded to an S3 bucket for deployment.

### 9. Upload to Amazon S3

```
- name: Upload to S3
  run: |
    aws s3 cp my-app.zip s3://$AWS_S3_BUCKET/my-app.zip
  env:
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

- This step uploads the `my-app.zip` archive to an S3 bucket.
- The `aws s3 cp` command copies the ZIP file to the specified bucket (`$AWS_S3_BUCKET`), using environment variables defined earlier.
- `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` are pulled from **GitHub Secrets**, which securely stores credentials to avoid exposing sensitive information in the workflow file.

### 10. Deploy to AWS Elastic Beanstalk

```
- name: Deploy to Elastic Beanstalk
  run: |
    aws elasticbeanstalk create-application-version --application-name
$APPLICATION_NAME --version-label $GITHUB_SHA --source-bundle
S3Bucket=$AWS_S3_BUCKET,S3Key=my-app.zip
    aws elasticbeanstalk update-environment --environment-name
$ENVIRONMENT_NAME --version-label $GITHUB_SHA
  env:
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

- **Step 1**: Creates a new application version in AWS Elastic Beanstalk using the ZIP file in S3 (`my-app.zip`). The `version-label` is set to the commit SHA (`$GITHUB_SHA`), ensuring each deployment is uniquely identified.
- **Step 2**: Updates the specified Elastic Beanstalk environment (`$ENVIRONMENT_NAME`) to use the newly created application version.
- The deployment is secured using AWS credentials stored in GitHub Secrets.

## How to Use This Workflow

1. **Create a GitHub Repository**: Set up a new repository or use an existing one.
2. **Create `.github/workflows/deploy.yml`**: Add the workflow file to your repository.
3. **Configure GitHub Secrets**:
   - Go to your repository settings in GitHub.
   - Navigate to `Secrets` > `Actions` and add the following secrets:
     - `AWS_ACCESS_KEY_ID`
     - `AWS_SECRET_ACCESS_KEY`
4. **Push Changes to the `main` Branch**: Push your changes, and the CD pipeline will automatically trigger.

## Continuous Integration (CI) Workflow Using GitHub Actions

Below is a basic example of a GitHub Actions CI workflow file that automatically runs tests, builds the application, and checks code quality on every pull request or code push. I'll provide the workflow file and explain each step in detail.

## GitHub Actions CI Workflow File (`.github/workflows/ci.yml`)

```
name: CI Pipeline



# Trigger the workflow on push events to any branch and on pull requests

on:

  push:

    branches:

      - "**"  # Runs on all branches

  pull_request:

    branches:

      - main  # Runs on pull requests targeting the main branch



jobs:

  test-and-build:

    runs-on: ubuntu-latest



    strategy:

      matrix:

        node-version: [12, 14, 16]  # Test the application across different Node.js versions



    steps:

      # Step 1: Check out the code from the repository
```

```yaml
- name: Checkout code
  uses: actions/checkout@v3

# Step 2: Set up Node.js environment
- name: Set up Node.js
  uses: actions/setup-node@v3
  with:
    node-version: ${{ matrix.node-version }}  # Use the node version from the matrix

# Step 3: Cache dependencies to speed up build times
- name: Cache Node modules
  uses: actions/cache@v3
  with:
    path: ~/.npm
    key: ${{ runner.os }}-node-${{ matrix.node-version }}-${{ hashFiles('package-lock.json') }}
    restore-keys: |
      ${{ runner.os }}-node-${{ matrix.node-version }}

# Step 4: Install dependencies
- name: Install dependencies
  run: npm ci  # npm ci is faster and ensures a clean slate

# Step 5: Run tests
- name: Run Tests
  run: npm test
```

```
# Step 6: Lint the code for quality

- name: Lint Code

  run: npm run lint


# Step 7: Build the application

- name: Build Application

  run: npm run build


# Step 8: Upload build artifacts for further use

- name: Upload Build Artifacts

  uses: actions/upload-artifact@v3

  with:

    name: build-files

    path: build/
```

## Explanation of Each Step

### 1. Workflow Name and Triggers

```
name: CI Pipeline
```

- The `name` attribute defines the name of the workflow. It's displayed in the Actions tab of the repository.

```
on:
  push:
    branches:
      - "**"
  pull_request:
    branches:
      - main
```

- **`push` Trigger**: Runs the workflow whenever code is pushed to any branch (`"**"` pattern).

- **`pull_request` Trigger**: Runs the workflow on pull requests targeting the `main` branch. This ensures that all tests and quality checks are run before merging the code into the main branch.

## 2. Job Configuration

```
jobs:
  test-and-build:
    runs-on: ubuntu-latest
```

- The `jobs` section defines a single job named `test-and-build`.
- `runs-on` specifies the runner (virtual machine) to execute the job. Here, `ubuntu-latest` is used, which provides a Linux environment.

## 3. Matrix Strategy

```
strategy:
  matrix:
    node-version: [12, 14, 16]
```

- This step defines a **matrix strategy** to run the job across three different Node.js versions: 12, 14, and 16.
- The matrix strategy allows the same workflow to be tested on different configurations (e.g., multiple versions of dependencies, OS).

## 4. Checkout the Code

```
steps:
  - name: Checkout code
    uses: actions/checkout@v3
```

- This step checks out (clones) the repository's code so that subsequent steps have access to the codebase.
- `uses: actions/checkout@v3` is a predefined action that checks out the code in the current GitHub repository.

## 5. Set Up Node.js

```
- name: Set up Node.js
  uses: actions/setup-node@v3
  with:
    node-version: ${{ matrix.node-version }}
```

- This step sets up the Node.js environment using the `setup-node` action, which is necessary for building Node.js applications.
- The `node-version` references the matrix variable `${{ matrix.node-version }}` to specify different Node.js versions to test.

### 6. Cache Dependencies

```
- name: Cache Node modules
  uses: actions/cache@v3
  with:
    path: ~/.npm
    key: ${{ runner.os }}-node-${{ matrix.node-version }}-${{
hashFiles('package-lock.json') }}
    restore-keys: |
      ${{ runner.os }}-node-${{ matrix.node-version }}
```

- Caching is used to speed up the build process by storing dependencies locally on the runner.
- The `key` attribute defines a unique key for each cache based on the operating system, Node.js version, and the `package-lock.json` hash.
- If the cache is found, it's restored to `~/.npm`, reducing the need to re-download dependencies.

### 7. Install Dependencies

```
- name: Install dependencies
  run: npm ci
```

- Installs the dependencies specified in the `package.json` and `package-lock.json` files using `npm ci`. This command is faster and ensures a clean slate.

### 8. Run Tests

```
- name: Run Tests
  run: npm test
```

- Executes the test suite using `npm test`. This step is critical for ensuring code correctness.
- If any tests fail, the workflow will exit and mark the build as failed.

### 9. Lint Code

```
- name: Lint Code
  run: npm run lint
```

- Runs the linter to check for code quality and style issues. Ensures that the code adheres to a specified style guide and prevents common programming errors.

### 10. Build the Application

```
- name: Build Application
  run: npm run build
```

- Builds the application using the `npm run build` command. The build step varies depending on the application. For frontend applications, this usually means creating a production-ready bundle.

### 11. Upload Build Artifacts

```
- name: Upload Build Artifacts
  uses: actions/upload-artifact@v3
  with:
    name: build-files
    path: build/
```

- This step uploads the build output (from the `build/` directory) as an artifact. Artifacts can be used for further testing, deployment, or downloads.
- `uses: actions/upload-artifact@v3` is a predefined action for handling build artifacts.

## How to Use This Workflow

1. **Create a GitHub Repository**: Set up a new repository or use an existing one.
2. **Create `.github/workflows/ci.yml`**: Add the workflow file to your repository.
3. **Push Changes to Any Branch**: Push your changes to any branch, and the CI pipeline will automatically trigger.
4. **Create a Pull Request**: If you create a pull request targeting the `main` branch, this workflow will run to ensure all tests and quality checks pass before merging.