What is Git & GitHub?

Git is a tool which is primarily used for two main reason – for tracking history & for collaboration. Whereas, GitHub is basically a UI that supplements the git tool. It is used for storing and managing code using git.

What is Commit?

It is basically a process to final the changes. For example – in real life the commitment process of relationships completes in two steps – engagement & wedding. Similarly for git to finalize the change we have to follow two steps – add & commit. Add – tells the change is ready to be commit. And commit – tells the change is now final (i.e., permanent). Commits plays a crucial role in tracking history.

Note: On GitHub there is no add step, we can directly commit our changes.

Configuring Parameter – Global & Local

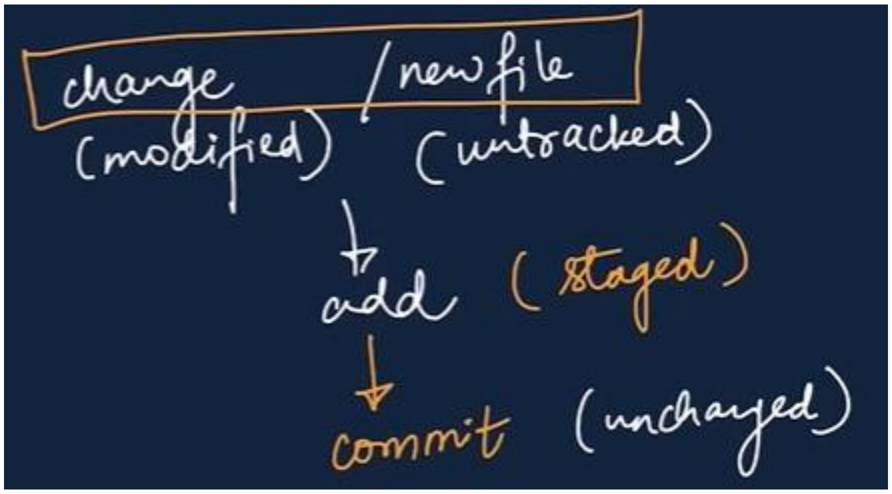
Global – tells git to use the provided values for all the future changes.

Local – use it when you want to commit changes in a specific repo/project from your different GitHub account.

```
git config --list #for seeing what configurations we have set.
```

Git Status – Types

4 types – untracked, modified, staged, unmodified.



Git Add & Commit

add – adds new or changed file in your working directory to the Git staging area.

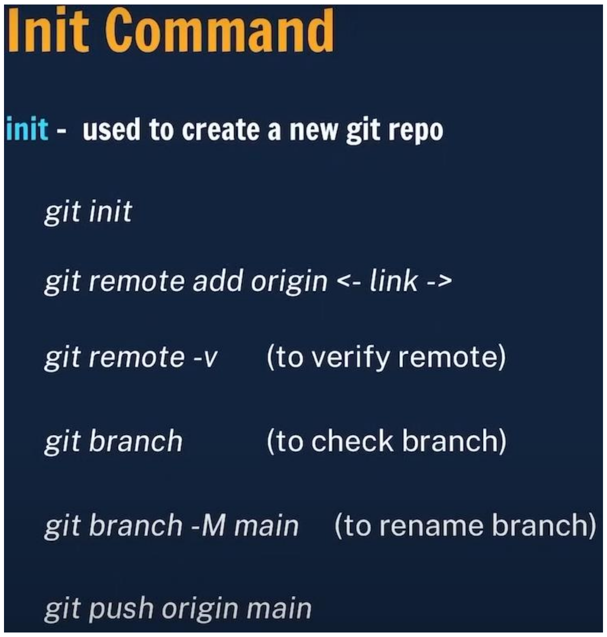
commit – it's the record of change.

difference – git add changes can be easily unstaged without affecting the working directory, whereas undoing a commit involves more steps and can affect the commit history. git commit creates a snapshot of your project at that point, which helps in tracking history.

Git Stages

1. **Working Directory**: This is where you make changes to your files. It's your local copy of the project.
2. **Staging Area (Index)**: This is where you place changes that you want to include in the next commit. You use `git add` to move changes from the working directory to the staging area.
3. **Local Repository**: This is where your commits are stored. When you commit changes, they move from the staging area to the local repository. You use `git commit` to move changes from the staging area to the local repository.
4. **Remote Repository**: This is a version of your project hosted on a server, which can be shared with others. You use `git push` to send your commits from the local repository to the remote repository.

Git Init



Init Command

init - used to create a new git repo

```
git init

git remote add origin <- link ->

git remote -v          (to verify remote)

git branch             (to check branch)

git branch -M main     (to rename branch)

git push origin main
```

Git Push

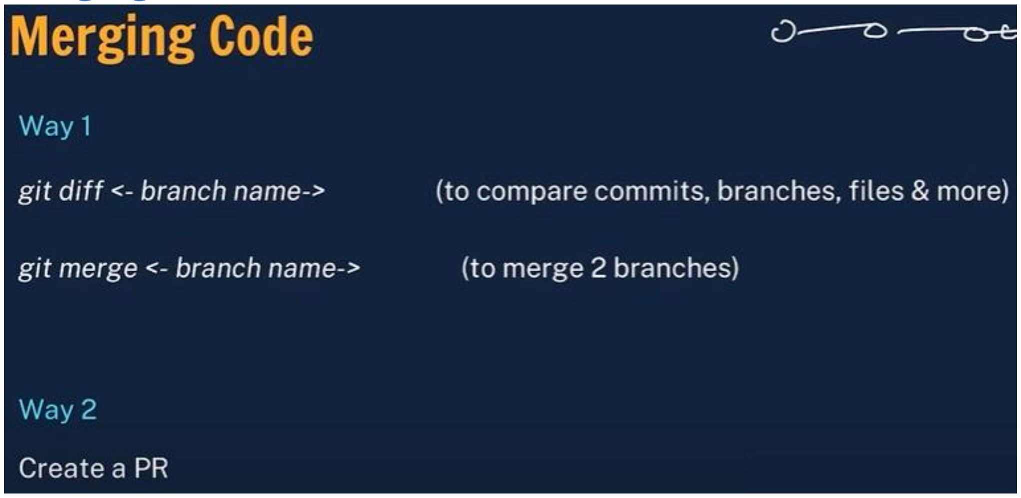
```
git push -u origin main # -u will set the upstream, basically it will tell git that from now onwards we will only write "git push" it will automatically understand where to push. Feasible when we always have to push to "origin main"
```

Initially the default branch name was "master" so at that time we have to change branch name to push changes to main.

Deleting Branch

We can't delete current working branch, first, we have to switch it and then we can delete it.

Merging Code



Merging Code

Way 1

git diff <- branch name-> (to compare commits, branches, files & more)

git merge <- branch name-> (to merge 2 branches)

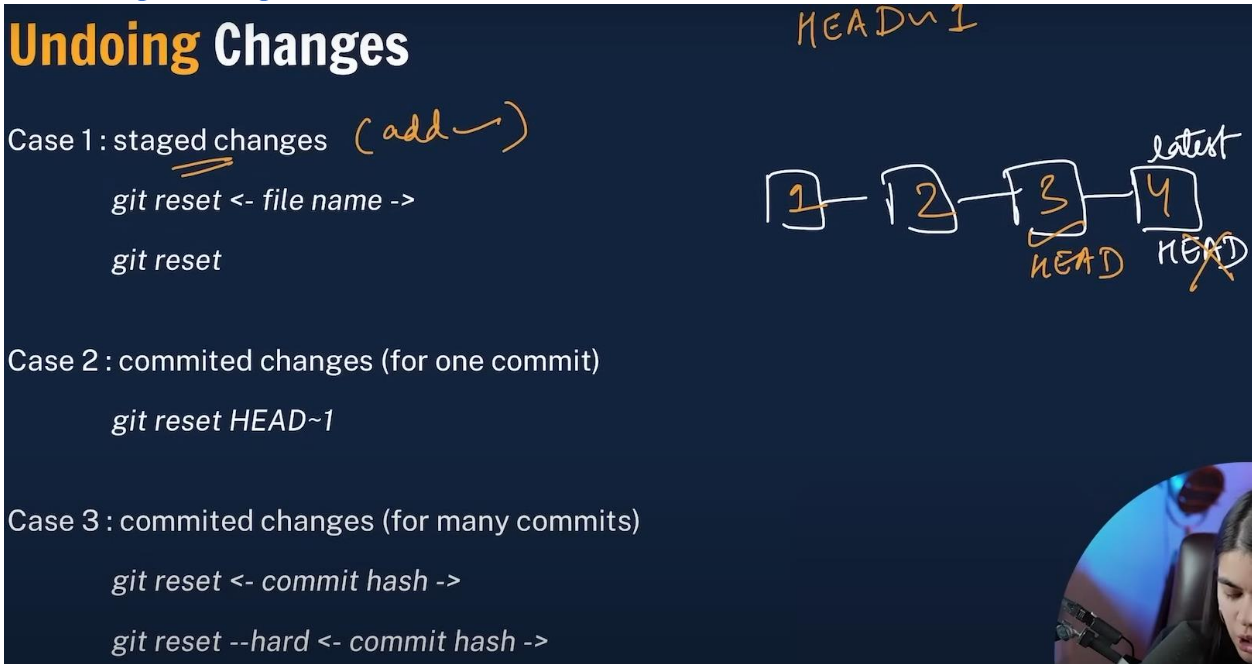
Way 2

Create a PR

Create a Pull Request – It's a way to merge using GitHub. It is recommended when we are working under supervision so we will create a pull request. It will let senior know that we have pushed final changes to a branch and now requesting to merge it into main. It will allow senior to review it. Once senior is satisfied, he will select "merge pull request" so GitHub will start checking whether they can be automatically merge able or not.

automatically merge able – it will check whether we have added a new code or changes an existing code of main. So, if we have changed the existing code, it will show a conflict, and the "Merge Conflict" event will be triggered. In this case we have to manually tell git what changes we want to keep.

Undoing Changes



Undoing Changes

Case 1 : staged changes

git reset <- file name ->

git reset

Case 2 : commited changes (for one commit)

git reset HEAD~1

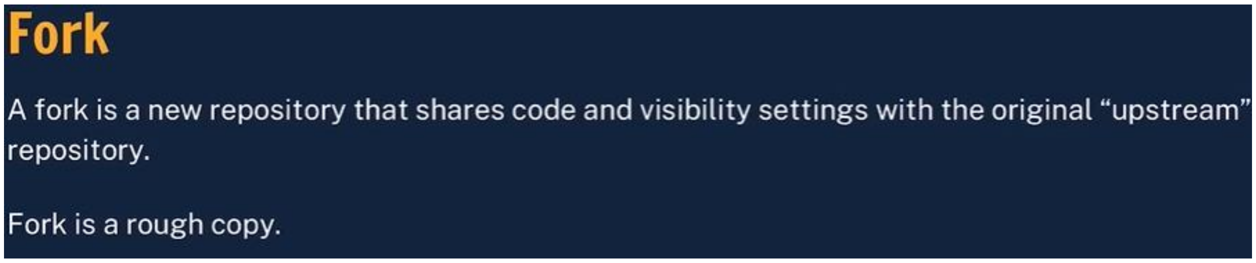
Case 3 : commited changes (for many commits)

git reset <- commit hash ->

git reset --hard <- commit hash ->

[commit hash] – is a automatically assigned hash to every commit. We can get the hash of any commit by running "git log" command. If we don't use the 'hard' parameter we will get back to that point but the changes will be still there so if we check "git status" we will get prompted that current directory/branch is behind main by __ no of commits. So, to completely remove we can use the 'hard' parameter.

Git fork



Fork

A fork is a new repository that shares code and visibility settings with the original "upstream" repository.

Fork is a rough copy.

Use to create a copy of someone else's code like open-source projects. To merge we have to create a pull request in a similar way as we have seen before.

Resetting commit history on GitHub

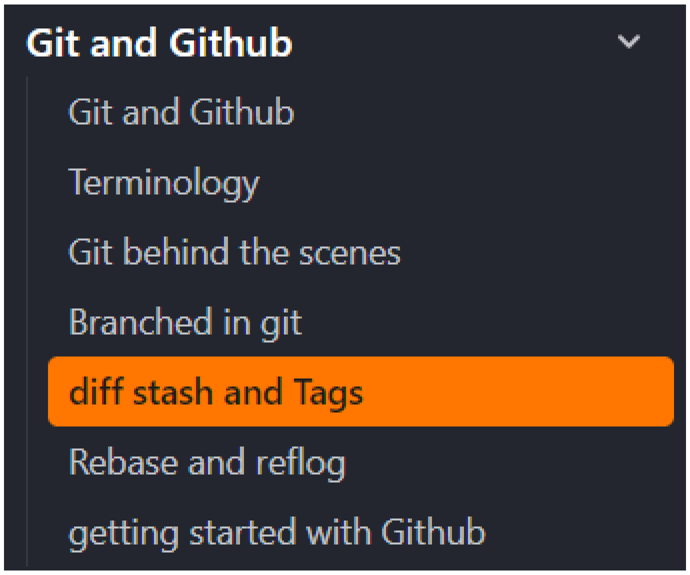
Let's say accidently you have committed and pushed your important credentials on GitHub. So, even you will change the code for the next push but as GitHub keeps track of history anyone can easily get back to that commit and see your credentials. So, in this case follow the following steps:

```
git checkout --orphan new-branch
```

# --orphan is used to create a branch with no commit history. This means the new branch will not have any parent commits, making it a completely fresh start. By using --orphan, you effectively create a new branch that looks like a brand-new repository with only the current state of your files.

Next steps are pretty much straight forward, add files, commit, delete old main branch, rename current branch to "main", push changes.

Git and GitHub

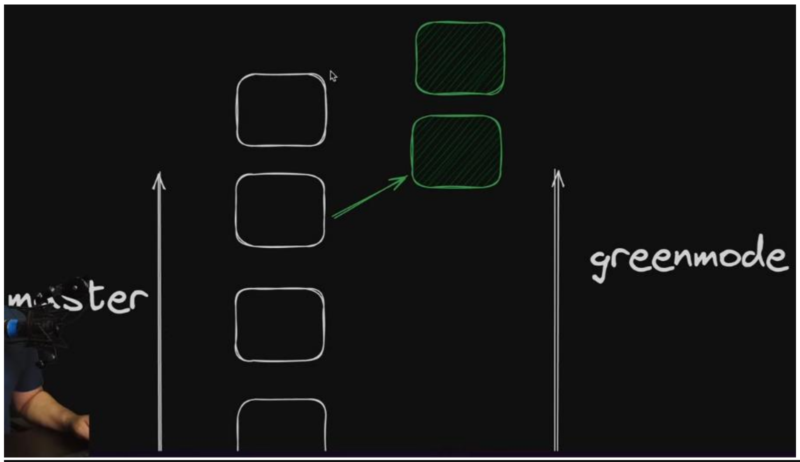


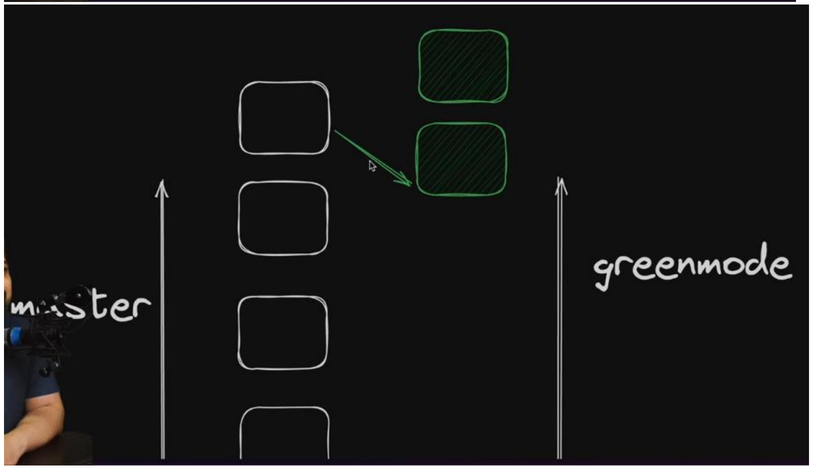
Git and Github

- Git and Github
- Terminology
- Git behind the scenes
- Branched in git
- diff stash and Tags
- Rebase and reflog
- getting started with Github

These topics are explained here, and here is the tutorial.

Rebase concept –



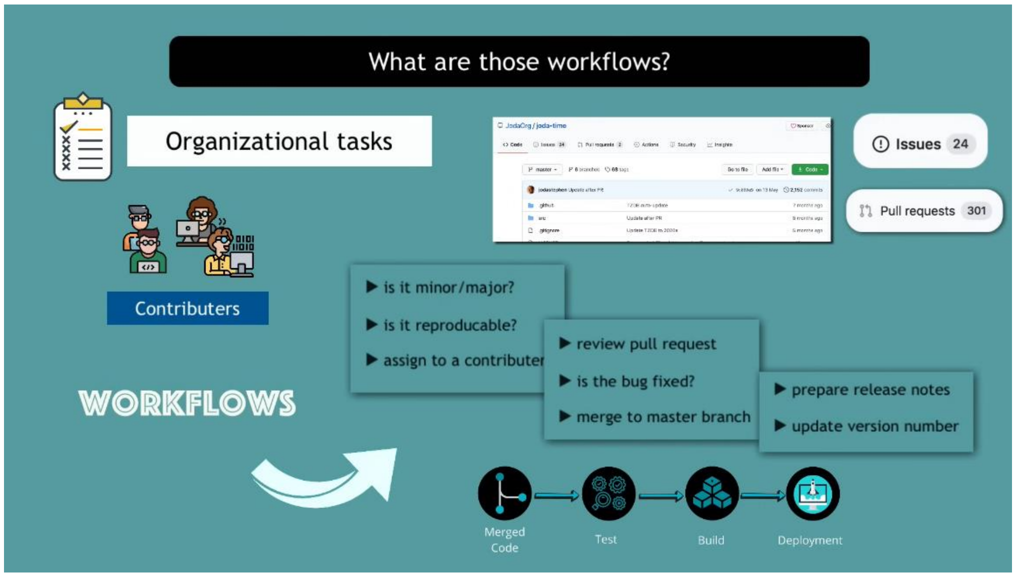


GitHub Actions

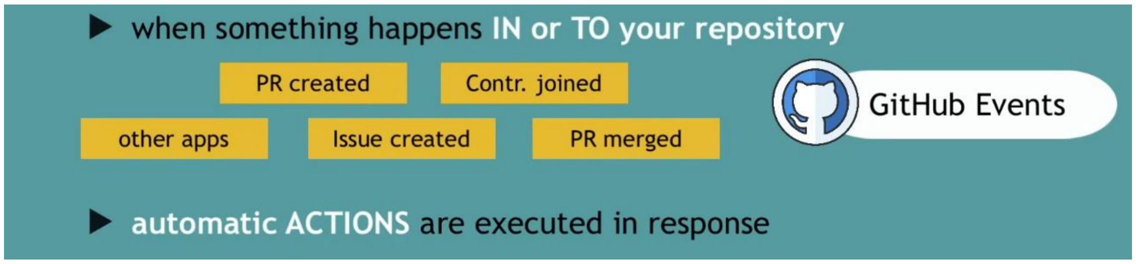
The tutorial is available here. Platform to automate developer workflows. CI/CD – is one of many workflows.

What are those workflows?

Let's say you have created a library and this library has some users and contributors. Some users found an issue and reported that. So these are the tasks/workflows which will you have to follow to fix an issue.



So as the project gets bigger you will be having more & more headache to process these tasks. So, you would want to automate these as much as possible. Just for this reason GitHub actions are! Whenever an event occurs these action gets triggered automatically.

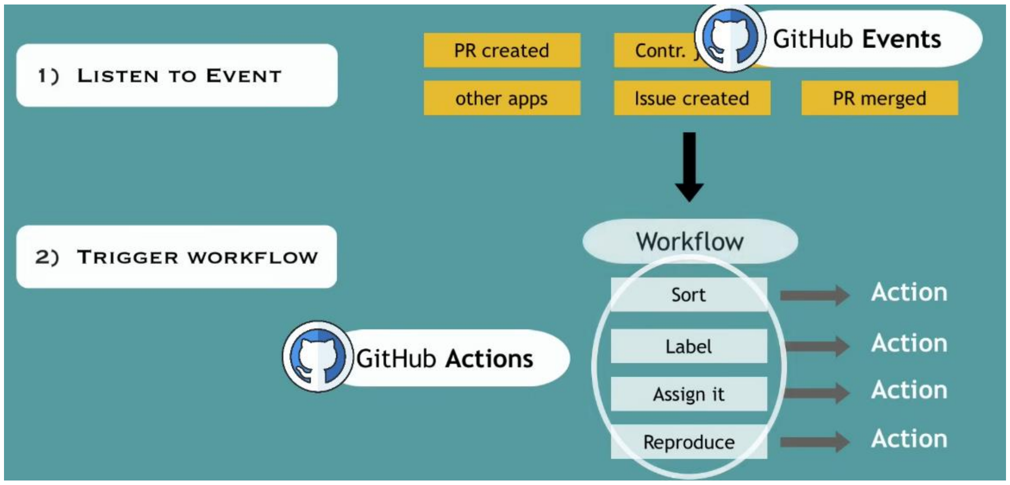


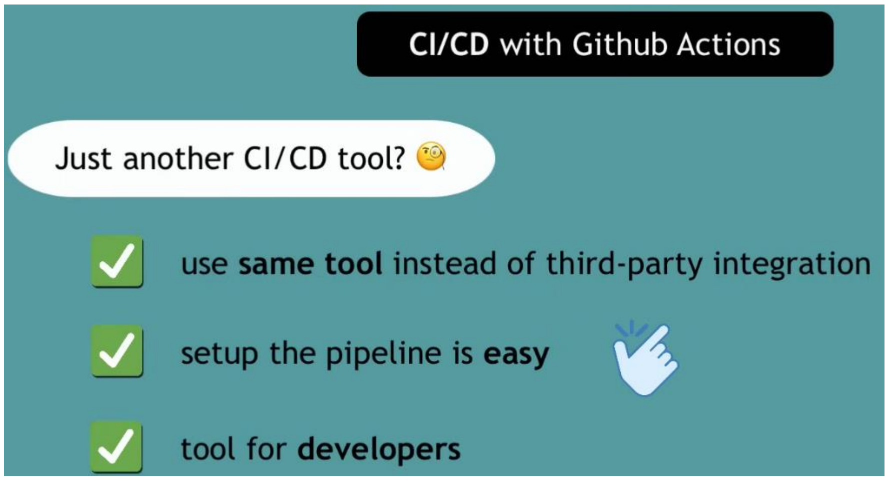
▶ when something happens IN or TO your repository

PR created | Contr. joined | other apps | Issue created | PR merged

GitHub Events

▶ automatic ACTIONS are executed in response





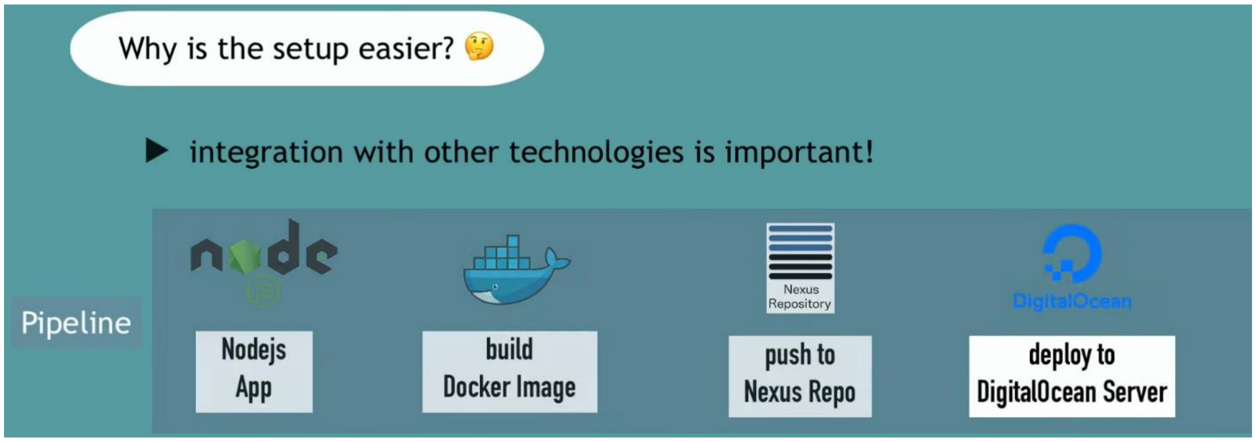
CI/CD with Github Actions

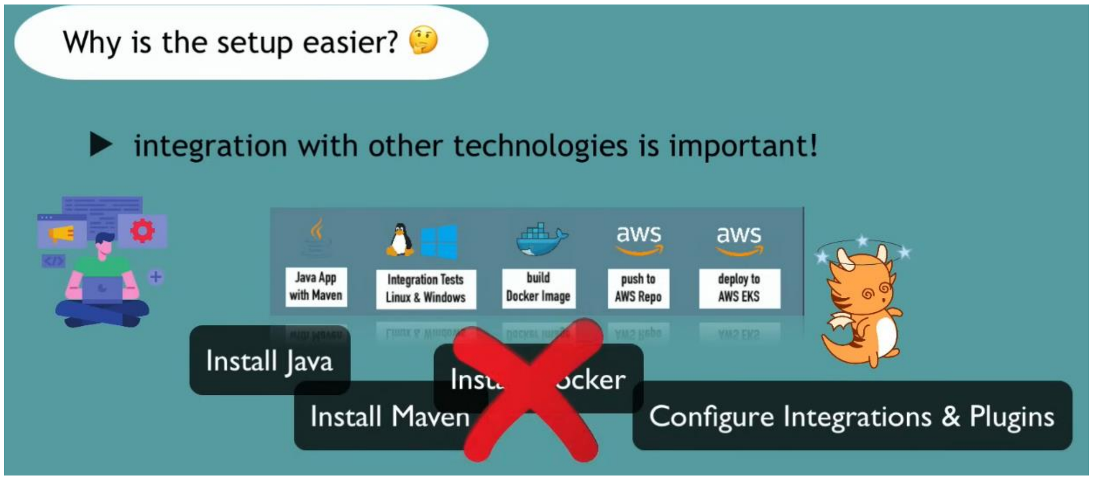
Just another CI/CD tool? 🧐

✓ use **same tool** instead of third-party integration

✓ setup the pipeline is **easy**

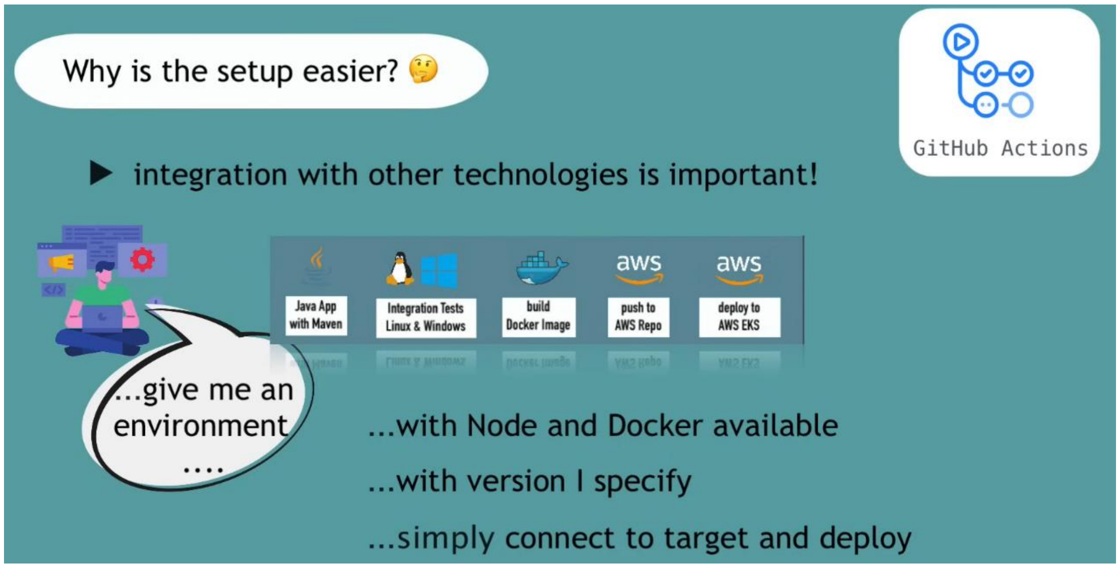
✓ tool for **developers**


Why is the setup easier? 🤔
▶ integration with other technologies is important!
Pipeline
Nodejs App
build Docker Image
Nexus Repository
push to Nexus Repo
DigitalOcean
deploy to DigitalOcean Server



Why is the setup easier? 🤔

▶ integration with other technologies is important!



Why is the setup easier? 🤔

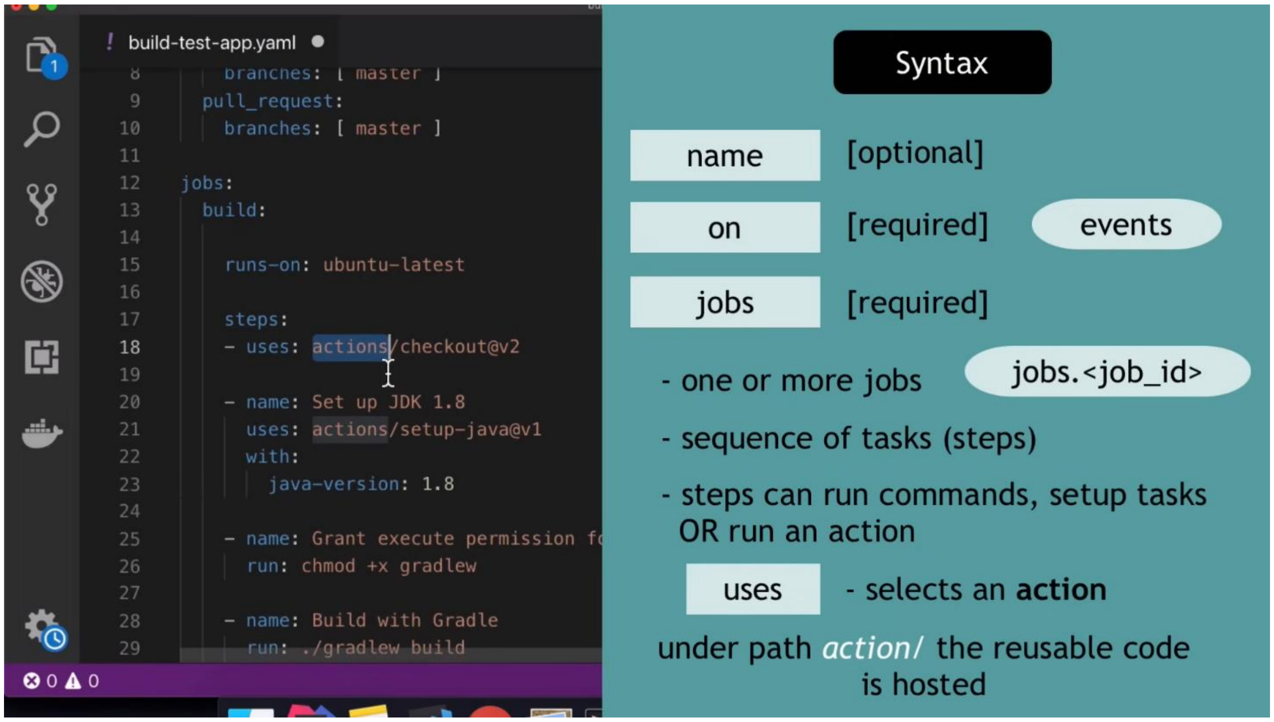
▶ integration with other technologies is important!

...with Node and Docker available

...with version I specify

...simply connect to target and deploy

GitHub events documentation [here](#).



```
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Set up JDK 1.8
      uses: actions/setup-java@v1
      with:
        java-version: 1.8

    - name: Grant execute permission f
      run: chmod +x gradlew

    - name: Build with Gradle
      run: ./gradlew build
```

Syntax

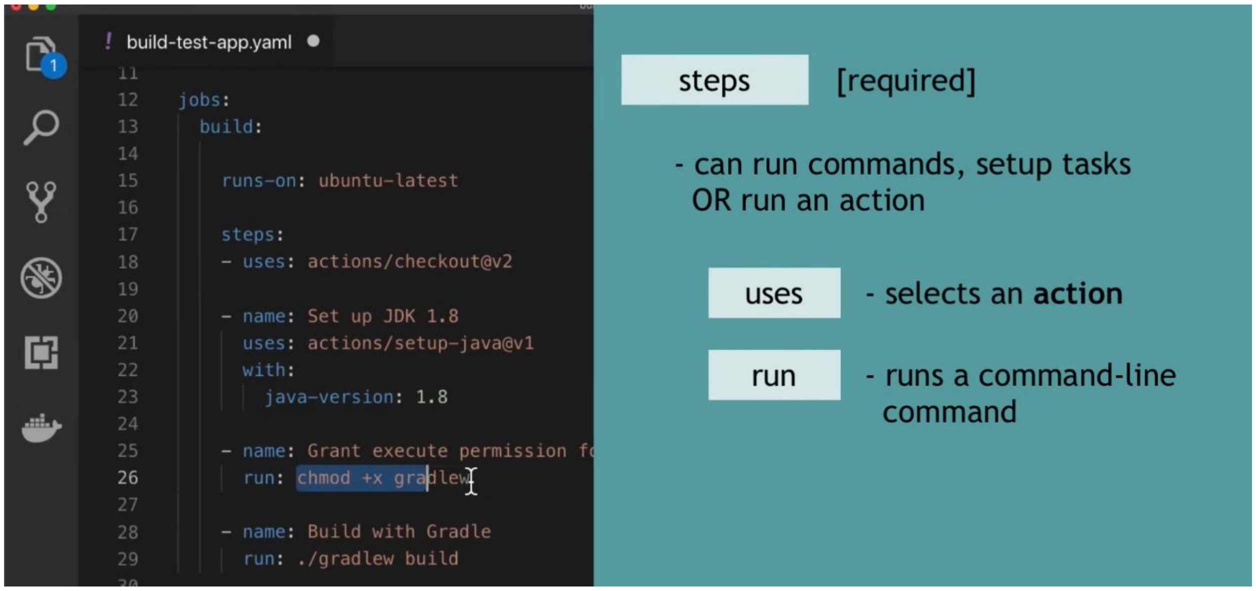
name	[optional]	
on	[required]	events
jobs	[required]	

- one or more jobs jobs.<job_id>
- sequence of tasks (steps)
- steps can run commands, setup tasks OR run an action

uses - selects an **action**

under path *action/* the reusable code is hosted

Predefined actions are listed [here](#).



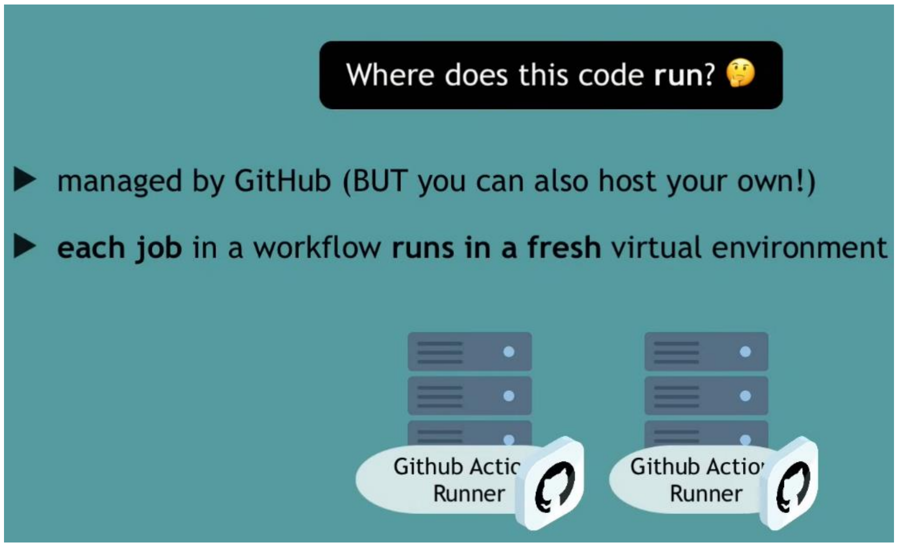
```yaml
jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Set up JDK 1.8
      uses: actions/setup-java@v1
      with:
        java-version: 1.8

    - name: Grant execute permission f
      run: chmod +x gradlew

    - name: Build with Gradle
      run: ./gradlew build
```

steps [required]

- can run commands, setup tasks OR run an action

uses - selects an **action**

run - runs a command-line command



Where does this code run? 🤔

- managed by GitHub (BUT you can also host your own!)
- **each job** in a workflow **runs in a fresh** virtual environment

Github Actic
Runner
Github Actio
Runner

If we have multiple jobs and the second job depends on first one so we have specify the "needs" attribute in very first line. Like below:

```
Build:
-list jobs here
Publish:
needs: Build
```

If we want to test our app on all available virtual servers we can do this like described below:

```yaml
runs-on: ${{matrix.os}}
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest, macOS-latest]
```

If we have specify multiple commands we can as described below. The "Pipe" is multiline syntax in yaml.

```yaml
run: |
  docker login cred
  docker build ...
```