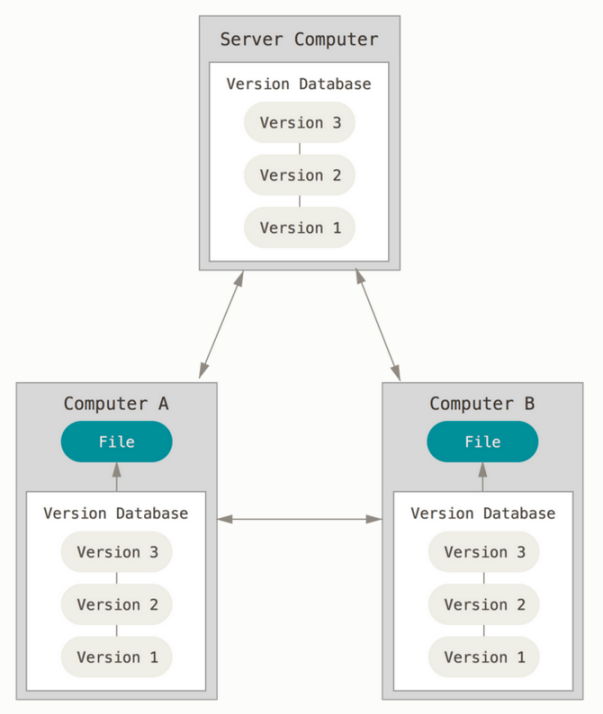
GitHub

Read git book: 1.3 | 2.6

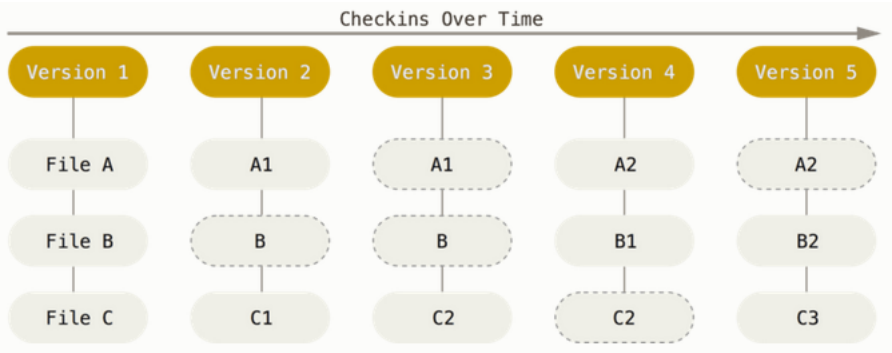
https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control



When you tell git to start keeping track of the files in a folder (or directory), you move the commits to a git **repository**, in which git
stores the snapshots of the files within the working directory.

As mentioned, git is a distributed VCS, but that doesn't mean that, whenever you work on your repository, you are going to change the state of the central server. When you work in a repository, your changes are local, and won't be reflected on the central server until you push those changes.

It only adds data, meaning that, if you remove a file, this operation could be seen as: "add file deletion". So you will not lose data if you commit your changes frequently.
You **ESPECIALLY**
will not lose it if you push your changes to some central server



- **Modified:** You have changed the file but have not committed it to your database yet
- **Staged:** You have marked a modified file in its current version to go into your next commit snapshot. The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit.

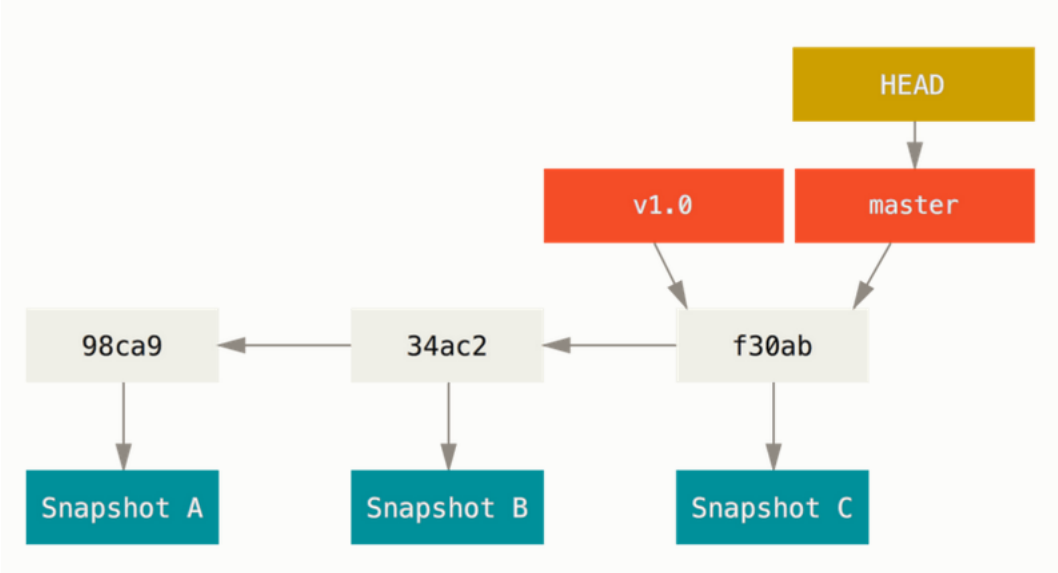
1. The file is created or modified. git compares the changes of the current directory with the last snapshot and notices that there are some changes. Those files are now labeled as **modified** because it changed with respect its last snapshot

- **Committed:** The data is safely stored in your local database.

To browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you. It simply reads it directly from your local database. This means you see the project history almost instantly. If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally

2. When you are happy with all the changes you make to the file, you can tell the file to "pose" for the snapshot. So you put it in the **stage** state
3. git takes the snapshot of the file and stores it in the repository. The file is now **committed.** Next time you change any other file in the working directory, it will enter the modified state again

If you want to do a little work, you can commit happily to your ***local*** copy, until you get to a network connection to upload.



Branching

> Branches are movable pointers to commits; You can think of them as a separate path in code development (which you can later merge with another branch)

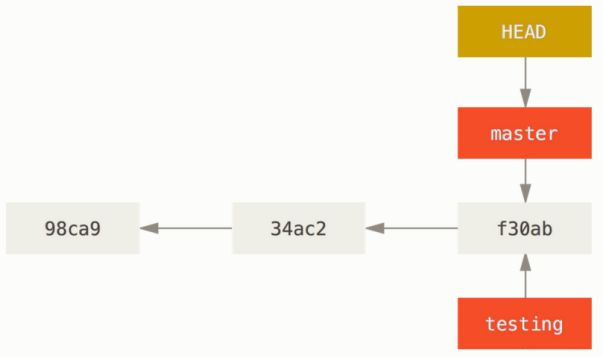
- By default, git creates a branch called main. **You should always keep it as your main branch!**
- Head is a pointer to where in the commit history we currently are
- Work on new features separately from other features and developers (separation is good!)
- Make the whole thing more structured and easy to follow
- **Not pollute the master branch** with untested/experimental/work in progress (WIP) code

```
git branch NAME_OF_BRANCH
'''a command responsible for handling branches'''
```

- **We are still on the master branch as indicated by head!**
- The new branch is merely a pointer to the branch

```
git checkout testing
git checkout -b NAME_OF_BRANCH
```

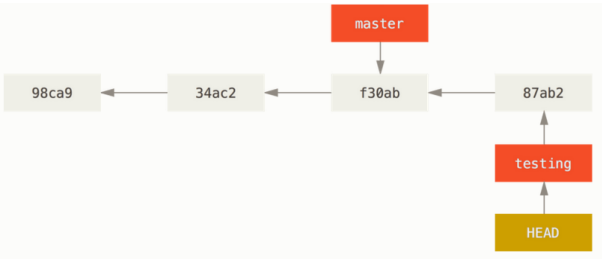
```
'''creates the new branch and checks it out in one command'''
git checkout master '''move back to master'''
```



Now we are on the testing branch (head points towards it) we can do the usual operations like git add, git command on it and come up with results like this:

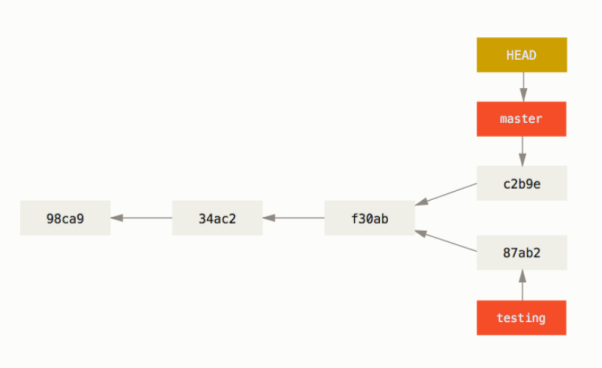
- **Your local changes will go back to how they were on master!**
- **This doesn't mean your files are lost. They are just committed to another branch!**

```
git checkout -b NAME_OF_THE_BRANCH
'''create a branch from the current one and change to it immediately'''
```

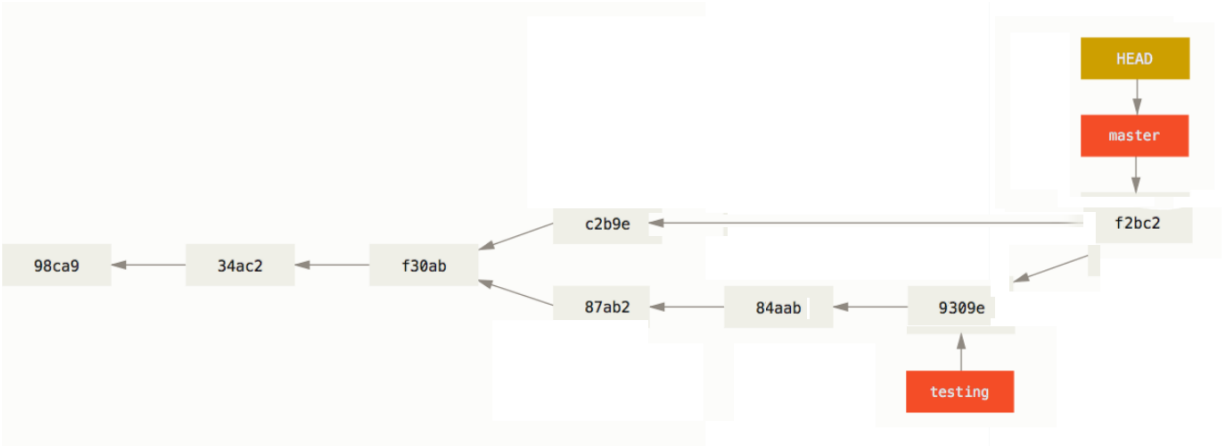


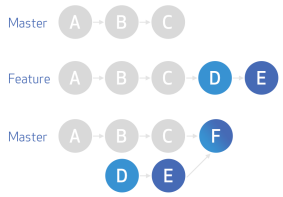
- **Pull all changes from the remote repository before creating a branch with a new feature!** (this will minimize the risk of merge conflicts)

Merging



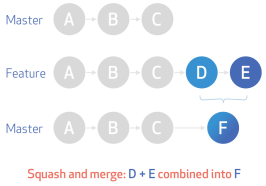
Let's say that you have finished working with the "testing branch. You can implement the changes from "testing" into "master"





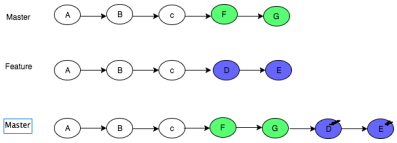
Merge commit: D + E added to Master via F

Merge retains all of the commits in your branch and interleaves them with commits on the base branch



Squash and merge: D + E combined into F

Squash retains the changes but omits the individual commits from history



Rebase moves the entire feature branch to begin on the tip of the master branch, effectively incorporating all of the new commits in master

How To Create A git Repository

```
AiCore_git
cd Desktop
mkdir AiCore_git
cd AiCore_git
git init
ls -a
touch "test_1.txt"
git status
git add test_1.txt
git commit test_1.txt -m "First commit"

git config --global user.name "Your Name"
git config --global user.email "your@email.com"
```

1. Move into the desktop (cd)
2. Create a new directory on your desktop (mkdir)
3. Move into the new directory
4. Run (git init) to set up a repository. This contains a hidden directory containing information about the commits you make
5. Use (ls -a) to check the hidden files. A directory called .git is there
6. Create a txt file (touch)
7. Check the status of git (git status)
8. Add a file to staged (git add)
9. Make a commit (git commit) and write a message (-m)
10. Provide your details to link to your gitHub (git config)

How To Branch And Merge

1. Create a new branch (git checkout -b)
2. Check the branches available (git branch)
3. Switch to the main branch (git checkout). test_2 will stay in the branch
4. Merge the branches together (git merge)

```
git checkout -b testing
git branch
touch "test_2.txt"
git add test_2.txt
git commit test_2.txt -m "Second commit"
git checkout master
git merge testing
```

How To Revert Changes

```
git reset HEAD~
git reset test_2.txt
```

Undoes the last commit/merge/stage. N**o changes to files will be done except that, so don't worry, it WILL NOT delete them!**

GitHub

1. Click on the + in the top right
2. Add .gitignore
3. Click on the green code button
4. Make sure your in HTTPS and copy the URL

.gitignore is a file that prevents adding language-related "junk" (files that are the result of running and not necessary for the project) to git

2GB size limit on a repository

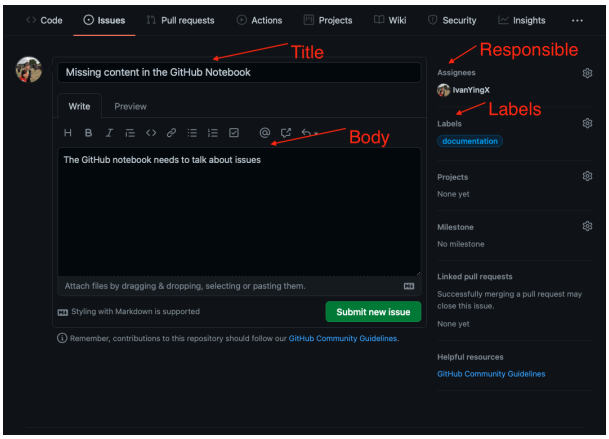
5. Clone your repository (git clone). This will create a local version of the code in a folder with the same name as the repo
6. Use git push to push

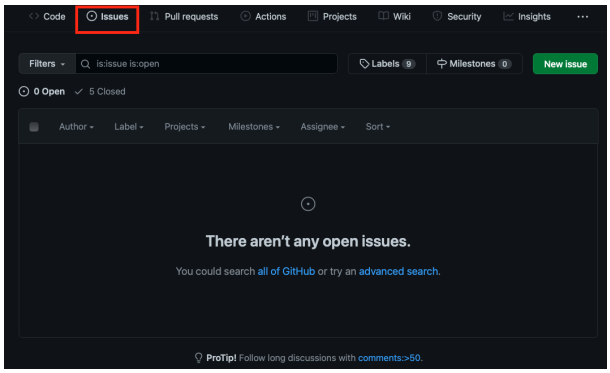
Pull request (PR) means we are asking the repository/project owner (or anyone with appropriate status) to **merge** changes located on our branch **upstream** (to the main branch

```
git clone <URL you just copied>
git push -u origin BRANCH_YOU_ARE_ON
```

Issues

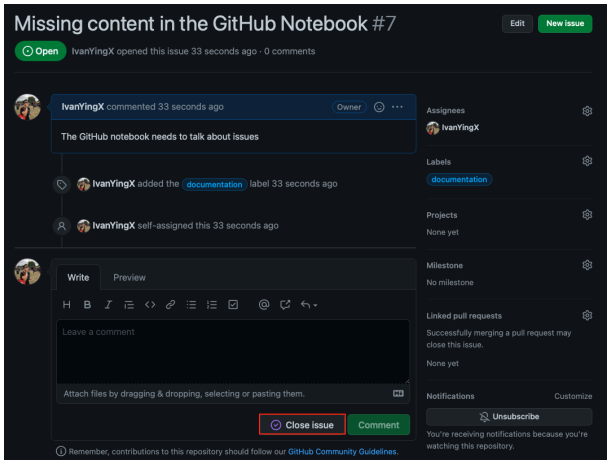
GitHub Issues is a tool for keeping track of what are the steps to develop your repo. It is very helpful especially when you are working in teams and you need a list of tasks to do. Each issue can be tagged with a label to define whether the issue consists of fixing something or adding new features.





You can even create your own tags! You can create issues by clicking on the "Issues" tab in your repo And then clicking on the "New issue" green button. In the next window, you can add a title and a description of your issue. Additionally, you can label your issue with a tag and make someone in your team responsible for fixing this issue. When you are happy with it, click on "Submit new issue". You will be redirected to the summary of the issue.

But a nice way to do it is by using a keyword followed by the number of the issue in a commit message. These keywords are: `close, closes, closed, fix, fixes, fixed, resolve, resolves, resolved` So, if I make a commit and add, for example, the following message: `"Add issues content. This fix #7"` It will close this issue automatically



Getting Started

Checksums

Everything in Git is checksummed before it is stored and is then referred to by that checksum. This means it's impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can't lose information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.

Git Basics

Commit Messages

Style: markup syntax | wrap margins | grammar | capitalization | punctuation

Content: what information should the body of the commit message (if any) contain | what should it *not* contain?

Metadata: How should issue tracking IDs | pull request numbers | be referenced?

Spell these things out, remove the guesswork, and make it all as simple as possible. The end result will be a remarkably consistent log that's not only a pleasure to read but that actually *does get read* on a regular basis.

Not every commit requires a subject and body. A single line is fine, especially when the change is so simple that no further context is necessary.

If you're having a hard time summarizing, you might be committing too many changes at once

Git never wraps text automatically. When you write the body of a commit message, you must mind its right margin, and wrap text manually.

1. Separate subject from body with a blank line
    a. a single short (less than 50 characters) line summarizing the change
    b. followed by a blank line
    c. a more thorough description
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
    a. this means 'an instruction or command'
    b. If applied, this commit will *your subject line here*
6. Wrap the body at 72 characters
7. Use the body to explain what and why
    a. leave out how a change has been made. Code is self-explanatory
    b. the reasons why you made the change
    c. the way things worked before the change (and what was wrong with that)
    d. the way they work now, and why you decided to solve it the way you did

Tagging

Git has the ability to tag specific points in a repository's history as being important. Typically, people use this functionality to mark release points (`v1.0`, `v2.0`).

```
git tag #lists all tags alphabetically
git tag -l "v1.8.5*" #lists all tags close to v1.8.5 | wildcards can only be used in -l or --list
git tag 'v1.4' #create lightweight tag
git tag -a v1.4 -m "my version 1.4" #createannotated tag
git show #shows tag and its data
git tag -a v1.2 9fceb02 #specify the checksum to add a tag to a previous commit
```

Lightweight: like a branch that doesn't change — it's just a pointer to a specific commit.

Annotated: stored as full objects in the Git database. They're checksummed; contain the tagger name, email, and date; have

```
git push origin <tagname> #push tags to a remote server
git push --tags #pushes all tags that arnt already there at once (both types)
git push <remote> --follow-tags #push all annotated tags
git tag -d v1.4-lw #deletes a tag (not from the remote servers)
git push origin :refs/tags/v1.4-lw #deletes a tag from remote servers)
git push origin --delete v1.4-lw #also deletes a tag from the remote server
```

a tagging message, and can be signed and verified with GNU Privacy Guard (GPG).

It's generally recommended that you create annotated tags so you can have all the information; but if you want a temporary tag or for some reason don't want to keep the other information, lightweight tags are the alternative

By default, the `git push` command doesn't transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them.

If you want to view the versions of files a tag is pointing to, you can do a `git checkout` of that tag, although this puts your repository in "detached HEAD" state, which has some ill side effects. In "detached HEAD" state, if you make changes and then create a commit, the tag will stay the same, but your new commit won't belong to any branch and will be unreachable, except by the exact commit hash. Thus, if you need to make changes — say you're fixing a bug on an older version, for instance — you will generally want to create a branch. If you do this and make a commit, your `version2` branch will be slightly different than your `v2.0.0` tag since it will move forward with your new changes, so do be careful.

Verifying

You can sign commits and tags locally, to give other people confidence about the origin of a change you have made. If a commit or tag has a GPG or S/MIME signature that is cryptographically verifiable, GitHub marks the commit or tag "Verified" or "Partially verified."

Commits and tags have the following verification statuses, depending on whether you have enabled vigilant mode. By default, vigilant mode is not enabled.

VIGILANT ENABLED

Verified	Commit is signed - signature successfully verified - user is the only author who has enabled vigilant mode
Partially Unverified	Commit is signed - signature successfully verified - user either: a) is not the committer b) has enabled vigilant mode. So commit signature doesn't guarantee consent of the author - commit is partially verified
Unverified	Commit signed but signature could not be verified - Commit not signed but user enabled vigilant mode - commit not signed and author has enabled vigilant mode

VIGILANT NOT ENABLED

Verified	The commit is signed and the signature was successfully verified
Unverified	The commit is signed but the signature could not be verified
No Verification Status	The commit is not signed

Repository administrators can enforce required commit signing on a branch to block all commits that are not signed and verified.

GitHub will automatically use GPG to sign commits you make using the GitHub web interface. Commits signed by GitHub will have a verified status on GitHub.

Organizations and GitHub Apps that require commit signing can use bots to sign commits. If a commit or tag has a bot signature that is cryptographically verifiable, GitHub marks the commit or tag as verified.

Authorization Token

1. Go to profile settings
2. Click on developer settings
3. Personal access tokens
4. Create new token
5. Copy token into request

Actions

GitHub Actions allow us to automate various stages of software development if GitHub is employed as the version-control system.

- **Event-driven:** Within a GitHub repository, it is possible to automate the execution of a piece of code (e.g. pull request) based on event triggers.
- **Configuration-oriented:** `.yaml` is employed to configure the necessary **workflows.**
- **Customisable:** We can create custom GitHub Actions with `docker` or JavaScript.
- **Supports simple continuous integration (CI):** Each pull request/merge can be automatically tested.
- **Supports simple continuous deployment/delivery (CD):** After successful testing, we can create a pipeline that automatically deploys the code for others to use.

Continuous Integration

The integration of software changes from multiple contributors.

- assessing the code quality (via automated code assessment, reviews or both).
- assessing adherence to the project's best practices.

All of the above and more can be achieved using GitHub Actions in order to streamline the code-merging process (e.g. assign labels automatically).

Continuous Deployment

The testing of production code to ensure that it is always ready for deployment (i.e. bug-free).

- inspecting changes against integration tests (or a whole suite of them).
- testing every feature that is to be merged.
- rapid deployment of necessary fixes (users always have the best possible version of the product available).
-

Continuous Delivery

The automatic deployment or code after testing and approval. This approach may not be suitable for every project and may be difficult to achieve. However, it offers significant benefits if done correctly.

- automated/easy rollbacks (to catch critical flaws when tests fail),
- test optimization to ensure that only the necessary tests are conducted to improve speed and reduce cost,
- automated deployment of the changes to production (e.g. creating images and containers, and deploying them on Kubernetes as a new version of the software).

Concepts

Events: Anything that occurs inside the repository, e.g. new issue, pull request, and merging (**external events are allowed via webhooks**).

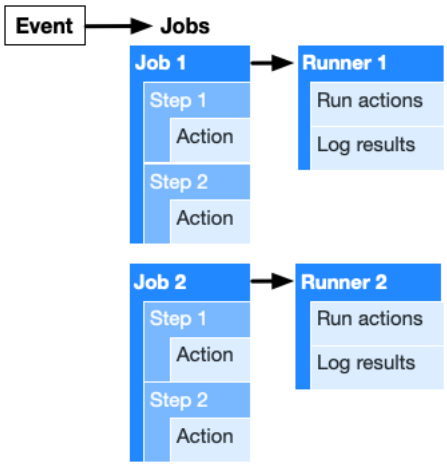
Workflow: Automated processes added to the repository.

Jobs: Set of steps running on the same machine.

Steps: Individual tasks that can run **action**/shell commands on the machine.

Action: Standalone commands **set up individually in separate/the same GitHub repository.**

Runner: A server with Github Actions runner installed.



- **Jobs in a workflow run in parallel by default.**
- **Dependencies can be created between jobs** (e.g. deployment relies on the success of the testing job).
- **Any publicly set-up Action can be used**
- **Runners are provided by GitHub for free**
- It is also possible to set up custom runners

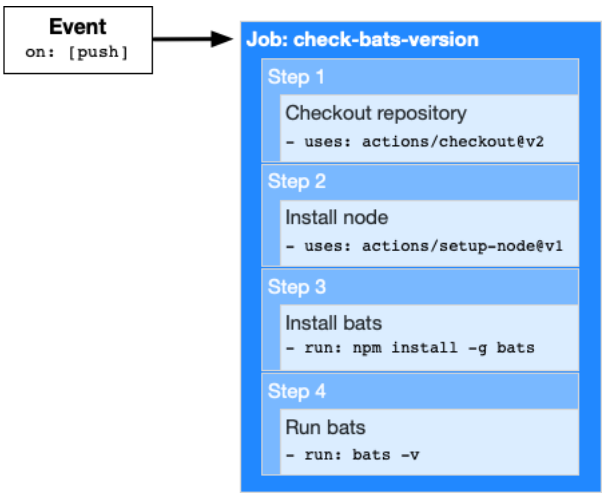
Structure

To use GitHub Actions, the following are required:

- an appropriate folder in our repository containing workflows (`/.github/workflows`).
- workflow `.yaml` files.

```
name: learn-github-actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-node@v1
      - run: npm install -g bats
      - run: bats -v
```

- `name`: specifies the name of the workflow
- `on`: event on which it will run
- `jobs`: list of jobs that will run in parallel, unless otherwise specified
  - `check-bats-version`: our first job
    - `runs-on`: runs on the latest Ubuntu worker...
    - ... and contains the following `steps`:
      - `uses`: checks out our repository
      - `uses` `node`: for running `npm` commands
      - `run`s JS installation of bats package
      - `run`s the command that checks the `bats` version



Fields

on = Enables the specification of multiple events and their details based on which the workflow is run

There are a few possibilities for running the workflow based on predefined events:

- On `push` / `pull_request` branches and/or tags, including their regex exclusion/inclusion.
- On `push` / `pull_request` to a specific file/directory.

```
on:
  # Trigger the workflow on push or pull request,
  # but only for the main branch
  push:
    branches:
      - main
  pull_request:
  # Also trigger on page_build, as well as release created events
  page_build:
  release:
    types: [published, created, edited]
  schedule:
    # Since * is a special character in YAML, the string must be w
    - cron:  '30 5,17 * * *'
```

env = Allows us to set the availability of environment variables

Availability can be set for different stages:

- all jobs
- single job
- single step

```
env:
  SERVER: production
jobs:
  job1:
    env:
      FIRST_NAME: Mona
    steps:
      - name: My first action
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          FIRST_NAME: Mona
          LAST_NAME: Octocat
```

jobs = specify the jobs that must achieve completion before the current one runs. This approach enables the sequential running of jobs separately.

- Test our application with multiple Python versions (using matrix).
- If all the tests are successful, move to the deployment stage.

```
jobs:
  job1:
  job2:
    needs: job1
  job3:
    needs: [job1, job2]
```

```
jobs:
  job1:
  job2:
    needs: job1
  job3:
    if: always()
    needs: [job1, job2]
```

By default, all required jobs must finish successfully before the job awaiting their completion runs.

```
runs-on: ${{ matrix.os }}
strategy:
  matrix:
    os: [ubuntu-18.04, ubuntu-20.04]
    node: [10, 12, 14]
steps:
  - uses: actions/setup-node@v2
    with:
      node-version: ${{ matrix.node }}
```

strategy and matrix: It is possible to specify a **strategy** that specifies the matrix of the jobs to be run. we can easily create many similar jobs via variable substitution

- The maximum number of jobs that can be generated this way is `256`.
- Ordering matters; the first defined option will be the first job.

We can also add some specific configurations (e.g. a specific mix of OS and `node`, which is outside of the matrix) using `include` (or exclude with `exclude`):

```
name: Node.js CI
on: [push]
jobs:
  build:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [macos-latest, windows-latest, ubuntu-18.04]
        node: [8, 10, 12, 14]
        include:
          # includes a new variable of npm with a value of 6
          # for the matrix leg matching the os and version
          - os: windows-latest
            node: 8
            npm: 6
         exclude:
          # excludes node 8 on macOS
          - os: macos-latest
            node: 8
```

- `strategy.fail-fast`: either `true` or `false`; cancels the whole job if `true` (default: `true`).
- `strategy.max-parallel`: the number of parallel jobs in the matrix. By default, as many as possible (depending on the number of cores).

The variable for continuing if one job fails

```
runs-on: ${{ matrix.os }}
continue-on-error: ${{ matrix.experimental }}
strategy:
  fail-fast: false
  matrix:
    node: [13, 14]
    os: [macos-latest, ubuntu-18.04]
    experimental: [false]
    include:
      - node: 15
        os: ubuntu-18.04
        experimental: true
```

```
name: Greeting from Mona

on: [push, pull_request]

jobs:
  my-job:
    name: My Job
    runs-on: ubuntu-latest
    steps:
      - name: Print a greeting
        env:
          MY_VAR: Hi there! My name is
          FIRST_NAME: Mona
          MIDDLE_NAME: The
          LAST_NAME: Octocat
        run: |
          echo $MY_VAR $FIRST_NAME $MIDDLE_NAME $LAST_NAME.
      - name: My first step
        if: ${{ github.event_name == 'pull_request' && github.even
        run: echo "This event is a pull request and has no assigne
```

steps: Step is the essential unit that defines the role of the workflow; thus, it offers a few additional fields

Each step

- can (but does not have to) run an action (note that all actions run as steps).

```
steps:
  # Reference a specific commit
  - uses: actions/setup-node@c46424eee26de4078d34105d3de3cc4992202
  # Reference the major version of a release
  - uses: actions/setup-node@v1
  # Reference a minor version of a release
```

- has access to the system defined in the `runs-on` clause.
- runs a separate process.

Additionally, the number of steps is **unlimited**

```
  - uses: actions/setup-node@v1.2
  # Reference a branch
  - uses: actions/setup-node@main
```

uses= These are employed to select an action to run as part of a step in your job (an action is a standalone piece of code that performs a specific task, e.g. sets up a specific version of Python).

- Specify the version directly (similar to the case with Docker tags).
- Specify the major version for fixes.

Additionally, it is worth noting that the `uses` field often comes with the `with` nested field, which allows us to specify arguments for the action.

```
jobs:
  my_first_job:
    steps:
      - name: Check out repository
        uses: actions/checkout@v2
        with:
          repository: octocat/my-private-repo
          ref: v1.0
          token: ${{ secrets.PERSONAL_ACCESS_TOKEN }}
          path: ./.github/actions/my-private-repo
      - name: Run my action
        uses: ./.github/actions/my-private-repo/my-action
```

The `workflow` below specifies the checkout from the private repository

```
steps:
  - name: Clean temp directory
    run: rm -rf *
    working-directory: ./temp
```

```
steps:
  - name: Display the path
    run: echo $PATH
    shell: bash
```

run: The run field specifies the shell command that will run on the OS.

Simply pass the `shell` command into the field, and optionally specify the working directory with respect to the repository's root:

You could also specify a different shell to run the command (`bash` is the default shell), e.g.

Python is available as an optional shell; therefore, it is possible to run Python commands directly.

Contexts and Expressions

Contexts are a set of defined (**or predefined**) variables used within workflows.

They can be accessed using the **expression syntax** and **index** access

Here are a few commonly used contexts:

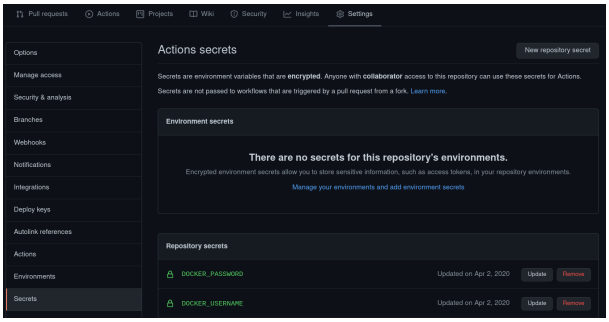
```
name: CI
on: push
jobs:
  prod-check:
    # Expression syntax; GitHub is the context, and ref is the key
    # We could use github.ref instead
    if: ${{ github['ref'] == 'refs/heads/main' }}
    runs-on: ubuntu-latest
    steps:
      - run: echo "Deploying to production server on branch $GITHU
```

- `github`: docs; GitHub-related info and info about the current run.
- `env`: docs; environment variables created within the workflow using `env`.
- `job`: docs; variables related to the current job.
- `steps`: docs; variables related to the step(s).
- `secrets`: docs; allow the specification of secrets, such as passwords, in the repository.__

Specifically, secrets can be specified in `settings`

- **Note that the secrets specified in `actions` will be available in the context for others to use.**
- Additionally, these secrets will be accessible repository-wide, but will not be encrypted

Most of the data required for custom actions and/or our scripts have already been provided by GitHub (e.g. branch name, event name, runner name, etc.).




```
steps:
  - name: Hello world action
    with: # Set the secret as an input
      super_secret: ${{ secrets.SuperSecret }}
    env: # Or as an environment variable
      super_secret: ${{ secrets.SuperSecret }}
```

Literals

When using expressions, we can employ a set of values for comparison.

Operator	Description
()	Logical grouping
[]	Index
.	Property dereference
!	Not
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal
&&	And
||	Or

```
env:
  myNull: ${{ null }}
  myBoolean: ${{ false }}
  myIntegerNumber: ${{ 711 }}
  myFloatNumber: ${{ -9.2 }}
  myHexNumber: ${{ 0xff }}
  myExponentialNumber: ${{ -2.99-e2 }}
  myString: ${{ 'Mona the Octocat' }}
  myEscapedString: ${{ 'It''s open source!' }}
```

- GitHub casts data to numerical types if they do not match.
- Comparison is case insensitive.
- Objects are equated based on instance, not data.__

Functions

Besides predefined environment variables, configurable context

and others, GitHub Actions provide a few functions to use within the `.yaml`.

- `contains(iterable, item)`: `true` if the `iterable` contains an `item`, e.g. `contains(github.event.issue.labels.*.name, 'bug')`.
- `{starts, ends}With(string, string)`: for example, `startsWith('Hello world', 'He')`.
- `format(string, r1, r2, ..., rN)`: similar to the case in Python, e.g. `format("Current reference on branch: {0}", github['ref'])`.

> Note: In github.event.issue.labels.*.name, any matching key in place of * will be returned. For example, [bug, first-issue, fix].

- `{to, from}JSON(value)`: returns the JSON pretty-printed value of `value`, e.g. `toJSON(job)` might return `{ "status": "Success" }`.

These functions can be exploited to pass data between jobs:

```
name: build
on: push
jobs:
  job1:
    runs-on: ubuntu-latest
    outputs:
      matrix: ${{ steps.set-matrix.outputs.matrix }}
    steps:
      - id: set-matrix
        run: echo "::set-output name=matrix::{\"include\":[{\"proj
  job2:
    needs: job1
    runs-on: ubuntu-latest
    strategy:
      matrix: ${{fromJSON(needs.job1.outputs.matrix)}}
    steps:
      - run: build
```

STATUS-CHECK FUNCTIONS

These allow us to determine the status of previous jobs and to branch with the `if` conditional based on the status.

Consider the most common status-check function, `success()`

```
steps:
  ...
  - name: The job has succeeded
    if: ${{ success() }}
```

- `failure()`
- `cancelled()` (manually)
- `always()` (is always `true`, regardless of the previous status)

Errors

```
# Error
Another git process seems to be running in this repository, e.g.
an editor opened by 'git commit'. Please make sure all processes
are terminated then try again. If it still fails, a git process
may have crashed in this repository earlier:
remove the file manually to continue.
# Try
rm -f .git/index.lock
```