Joey's Github Foundations Cheatsheet

Version Control: is a group of systems and processes to manage changes to files, programs and directories. It is useful for anything that changes over time or needs to be shared. Version control is able to track files in different states, combine different versions of files, identify a particular version and revert changes.

Git: It is a popular control system for software development and data projects, open source and reliable. The benefits of git include it storing everything, being able to compare files at different times, seeing what changes were made by who and when and even reverting to previous versions of files.

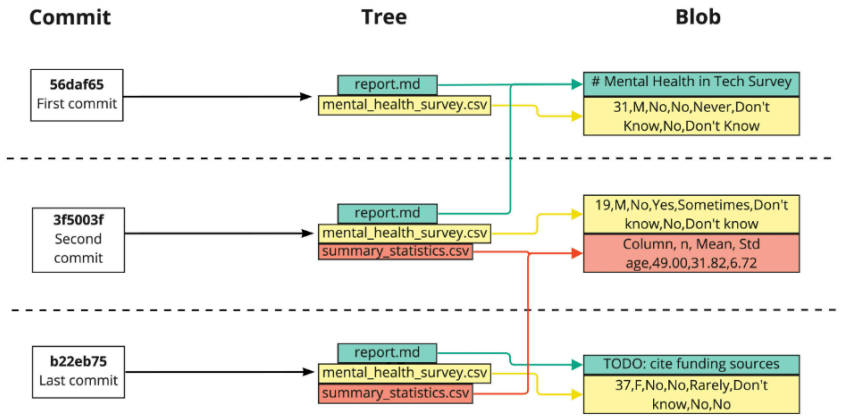
- Using Git: The git commands are run on the shell also known as the terminal. It is a program for executing commands

Repositories: A repository is a directory consisting of files and the Git records about the project's history. The benefits to creating a Git repo, includes the ability to systematically track the versions of files, to revert to previous versions, and to compare versions at different points in time.

- Nested Repositories: A Git repo inside another Git repo, there will be two .git directories which can confuse the Git about which directory needs to be updated when a commit is made.

Git Workflow:

- Edit and save files on the computer
- Add the files to the Git staging area
    - Tracks what has been modified
    - To add a single file called README.md use "git add README.md"
- Commit the files
    - Git takes a snapshot of the files at this point in time
    - Allows us to compare and revert files
    - Git commit structure has three parts
        - Commit: containing the metadata -author, log message, commit time
        - Tree: tracks the names and locations of files and directories in the repo. Like a dictionary - mapping keys to files/ directories.
        - Blob: also known as binary large objects, may contain data of any kind and a compressed snapshot of a file's content.



        - Unique Id called a hash is a 40 character string of numbers and letters. Hashes allow data sharing between repos, if two files are the same then the hashes are the same

Customizing the git log output

- Restricting the number of commits
    - git log - 3: to show the most recent 3 commits
    - git log report.md: to only look at the commit history of the file report.md
    - git show c27fa586: only the first 8-10 characters of the hash is enough to be unique. This command shows the more specific log entry of that commit and a diff output showing changes in that commit.

Using git diff, shows the changes that are being removed (in red) and added (in green)

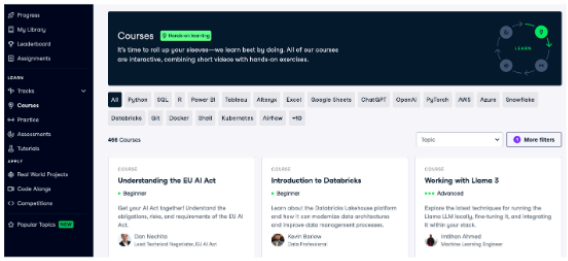
Git command Dictionary:

- Navigation
    - pwd: print working directory, this displays the current working directory of the user.
    - ls: this list the all the files in the current directory that the user is in
    - cd: this helps to change the user's directory
    - git --version: to check which version the user has installed git to be
    - nano README.md: reads the file README
- Repo creations
    - git init mental-workspace: this creates a new repo called "mental-workspace"
    - git init: this creates a repo based on the working directory that it is already in
    - git status: checks the status of the github repo
- Git Workflow
    - git add README.md: this adds a single file to the staging area
    - git . : this adds all modified files in the current directory and subdirectories to the staging area
    - git commit -m "Adding a README.": this allows to include a log message as part of the command, if -m is not used, a text editor will be opened
    - git log: shows the commits from the newest to oldest, it contains the commit number, author date and message. Pressing "space" allows you to view more recent commits. And "q" to exit the log outputs
        - git log - 3: to show the most recent 3 commits

        - git log report.md: to only look at the commit history of the file report.md
        - git log -2 mental_health_survey.csv: to show the most recent 2 commits in the csv file
        - git log --since='Apr 2 2024': to see commits since 2nd April 2024
        - git log --since='Apr 2 2024' --until='Apr 11 2024': searching for commits between dates
            - Possible to use natural language such as "2 weeks ago", "3 months ago" and "yesterday" or even different date formats such as "07-15-2024" or "15 Jul 2024"/ "15 July2024". The recommended format is to follow the ISO Format 6801 "YYYY-MM--DD"
        - git show c27fa586: only the first 8-10 characters of the hash is enough to be unique. This command shows the more specific log entry of that commit and a diff output showing changes in that commit.
    - git diff: shows the difference between versions, it compares the last committed version with the latest version not in the staging area.
        - git diff --staged report.md: to show the differences between the last committed version with the version in the staging area (the one which was git add)
        - git diff --staged: compares all files
        - git diff 25f4b4d 186398f: shows the difference from the first hash to the second hash. Generally the most recent hash should be placed in the second hash.
            - git diff HEAD~1 HEAD: does the same function but compares the second most recent with the most recent commit
        - git diff main summary-statistics: this compares the difference between two branches main and summary-statistics.
        - HEAD: checking the state in latest commit
    - git revert: restoring a repo to the state prior to the previous commit, it reinstates previous versions and restores all files in the given commit.
        - git revert HEAD (it includes making a commit)
            - git revert --no-edit HEAD (to avoid opening the text editor)
            - Ctrl + "o", then enter to save the text editor
            - Ctrl + "x" to exit the text editor
        - git revert -n HEAD: this reverts without committing (bringing the files into the staging area), the -n flag refers to no commit. git revert works on commit, not individual files.
        - git checkout: this helps to revert files or a number of files from a specific commit
            - git checkout HEAD~1 --report.md: refers to the second most recent commit and narrowing the command to a single file
        - git restore --staged summary_statistics.csv: to unstage a single file
            - git restore: to unstage all files

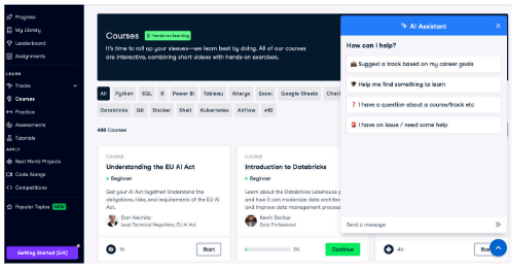
Branches: an individual version of a repo, git uses branches to systematically track multiple versions of files. In each branch: some files might be the same, others might be different and some may not exist at all.

Live system



- Works as expected
- Default branch = `main`

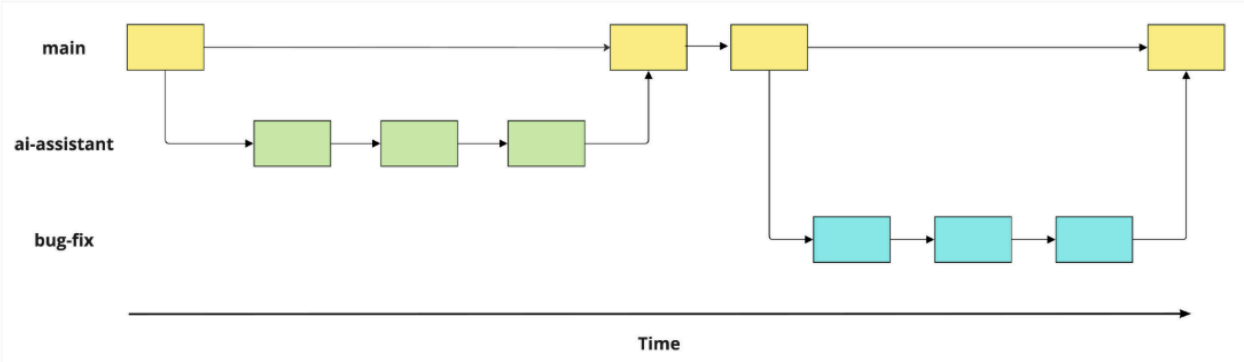
Feature development



- Might encounter issues during development and testing
- Doesn't affect the live system

Branches are beneficial as they allow multiple developers to work on a project simultaneously. Git makes it easy to compare the state of a repo between branches and to combine contents, pushing new features to a live system. Generally each branch has a specific different purpose.

Fixing a bug

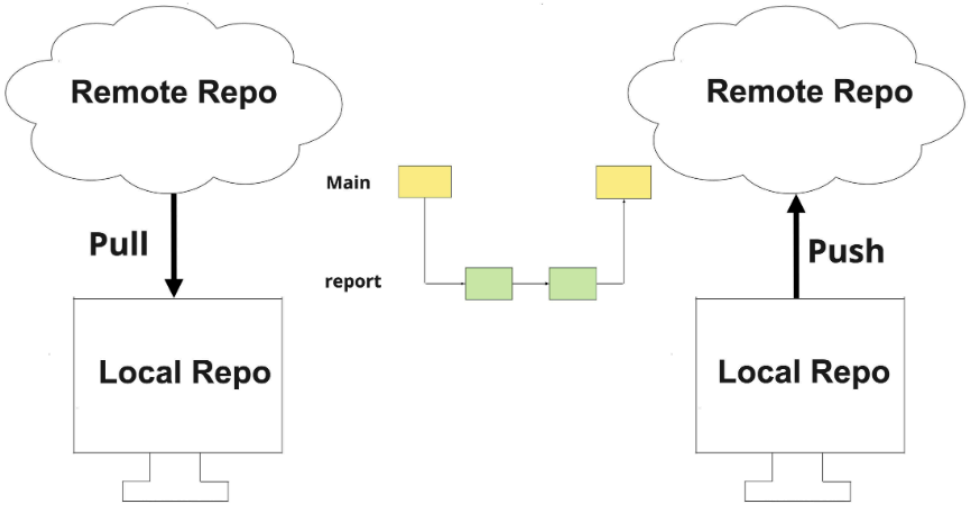


The left titles represent a branch while the green and blue boxes represent a commit.

Extra Git command Dictionary:

- Branches
  - Branch Navigation
    - git branch: this lists all the different branches in the repo. The asteris * tells you which branch that you are currently in.
    - git switch main: this allows you to switch between branches
    - git branch speed-test: this creates a new branch called speed-test
    - git switch -c speed-test: creates a new branch called speed-test and switches to it
  - Branch Editing
    - git branch -m feature_dev chatbot: changes the branch name from feature_dev to chatbot
    - git branch -d chatbot: this deletes the branch called chatbot, usually branches are deleted once we are finished with them. If a branch has not been merged to main, the deletion of a branch will produce an error.
      - git branch -D chatbot: deletes a branch that hasn't been merged.

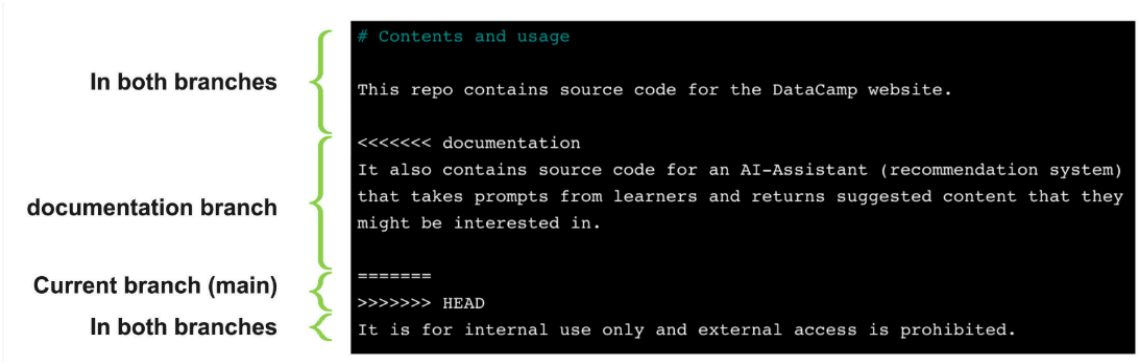
    - Branch comparison
        - git diff branch_A branch_B: to compare the difference between these two branches
    - Branch Merging (source into the destination)
        - git switch main: Switch into the detestation branch
        - git merge ai-assistant: ai-assistant is the source
        - Alternative Method
            - Git merge ai-assistant main: following the order of (git merge source destination)
- Cloning repo
    - git clone path-to-project-repo: making a local copy of the repo
        - git clone /home/george/repo new_repo: this renames the clone repo into a new name called new_repo
        - git clone https://github.com/datacamp/project
- Working with remotes
    - git remote: lists all remotes associated with the repo
        - git remote add george https://github.com/george_datacamp/repo: this creates a remote called george, it is tagging the repo link to a name called george
        - git remote -v: gets more information about the remote(s)
    - git fetch origin: fetch from the origin remote
        - git fetch origin main: fetching only from the origin remote's main branch
    - git merge origin: merging origin remote's default branch (main) into the local repo's current branch
    - git pull origin: this fetch and merge from the remote's default (main) into the local repo's current branch
        - git pull origin dev: this fetches the dev branch from the origin remote. This will automatically open a nano text editor and ask us to add a message for the merge.
        - git pull --no-edit origin main: pulls without editing the message, however this is not recommended
    - git push origin main: this saves changes locally, this can be also seen as pushing changes into origin from the local repo's main branch.



Branch merging: When merging two branches, the last commits from each branch are called parent commits. The main branch is always assumed to be the “ground truth”. One branch will be known as the source (the branch we want to merge from) and the other branch will be known as the destination (the branch we want to merge into).

- Branch Merging steps:
    - git switch main: Switch into the detestation branch
    - git merge ai-assistant: ai-assistant is the source

Conflicts: A conflict occurs when Git is unable to resolve differences in the contents of one or more files between branches. For example when editing the same file in two branches and then merging them together, Git does not know what version to keep.



Remote repo is a repo stored in the cloud through an online repo hosting service such as GitHub. The benefits of using remote are that everything is backed up, collaboration can be done regarding the location. Having a repo copy on our local computer makes this a local remote. When cloning a repo git remembers where the original was by storing a remote tag in the new repo’s configuration.

The remote repo should always be the project’s source of truth, where the latest versions of files that are not draft can be located.

GitHub: It is a cloud-based hosting service that allows users to store and track their work, aka version control. Cloud-based means that Github provides on-demand resources to its users over the internet, or the cloud. The primary use of GitHub is to store and keep track of projects and files and collaborate with others.

GitHub vs Git: Git is a version control software that can be used without GitHub or other hosting platform, GitHub is a platform that enhances Git to make it easier to manage projects and collaborate. The .git file stores the history of the file, together with the files that are stored, these made up a repo. Since the GitHub repo is stored on the internet, it is also known as a remote repo. Git defines a local repo as saved on our local computer and a remote repo as saved on the internet, or cloud.

README file: regardless of the number of files we have, the README will always be shown after the list of files. This is a markdown file therefore it uses markdown syntax. The view can be switched between an edit file which shows the markdown syntax and preview.

Markdown Syntax Fundamentals:

- # Heading1: this creates a heading, 1 to 6 hashtag shows the different size of headings
- **bold**: bolding a word
- *italics*: converting a word to be italics type
- [Premier League](https://www.premierleague.com/): the square bracket contains the text and the parentheses will contain the desired URL
- ![image](asdkas.jpg): this is to add an image, the text is used by screen readers and is shown if the image is unable to be displayed

Writing a README:

- Need to be descriptive as anyone should understand our project
- List the contents of the repository
- Clearly explain the project to others
- README fundamentals
    - Title
    - Description of technology and why?
    - Description of the process and why?
    - Table of contents
- README extras
    - How the project came about
    - The motivation
    - Limitations
    - Challenges
    - What problem it hopes to solve
    - What the intended use is
    - Credits

You can create a README file as a placeholder when creating a new folder in GitHub as they wont allow users to create an empty folder. references/README.md to create a new folder called references.

Branch protection rules:

- Require a pull request before merging
- Require that pull requests are approved before merging
- Restrict who can delete a protected branch
- This adds as a layer of protection against introducing code errors into the main branch and when requiring review can help improve code quality.

Personal Access Tokens (PATs): PAT is an alternative authentication to using passwords in the terminal and is required since August 2021 instead of passwords. They are used as they are more secure. The PAT is used in replacement when being asked for a password when attempting to clone a repo remotely.

Cloning: Similarly to copy and paste, it creates a copy on a local computer and allows updates to go back and forth. With Git, the user pushes changes back to the original repo and pulls changes into the user's local version.

Forking: This clones a repo without a link back to the original repo. This creates an independent copy of it on our Github. This means we can run experiments on it without the risk of anything reaching the original repo.

GitHub Issues: These are messages to track problem fixes, plans, important tasks and communications for a project.

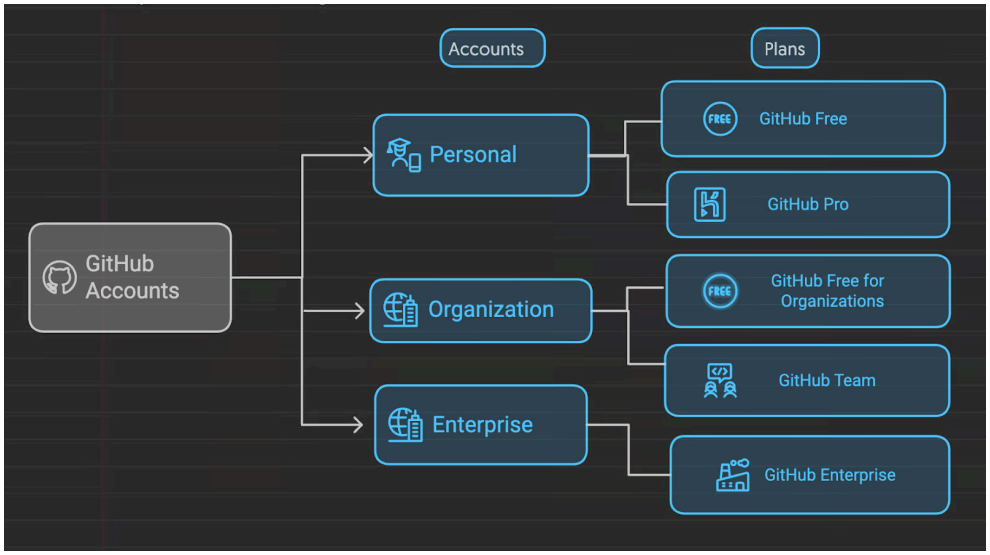
GitHub issue practices:

- Assign: Who should work on the issue
- Tag: WHo needs to read the issue
- Using # to link to other issues
- Using @ to tag users within the issue
- Using > to quote in a comment

A pull request (PR) is a way to notify others about changes, it allows the repo owner to check changes before they are added. It is best practice to add changes to a branch that is not the main branch. The base branch is the branch we want to add our updates to, where the compare branch is the branch that contains the update.

Pull Request (PR) practices:

- Assignee: Approves the PR
- Reviewer: Looks at the changes before the PR is merged
- Comment: Feedback, suggestions and are not required to be incorporated
- Request Changes: confirms that the feedback needs to be incorporated
- It is good practice to delete the branch that has been merged, the deleted branch can be restored if needed. This can be done through the “restore” button in the PR history.



Choosing which github account to use.

GitHub Projects: The benefits of using this helps to organize tasks, track progress, enhance collaboration and simplify tracking. It is able to integrate with issues and PRs, centralized management and customizable boards.

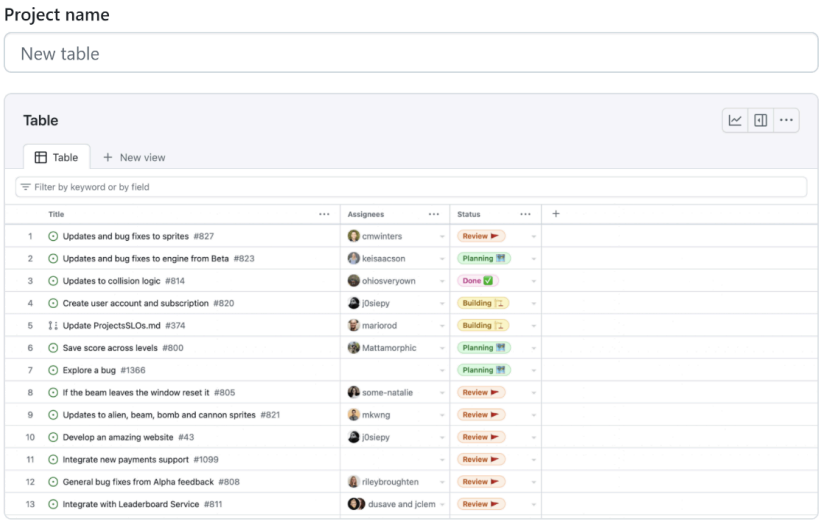
GitHub Projects vs Projects (Classic):

- GitHub Projects
    - Flexible and customizable
    - Independent of repositories
    - Advanced features
- Projects (Classic)
    - Tied to repositories
    - Basic tracking (disabled in Aug 2024)

Managing Project Visibility and Access:

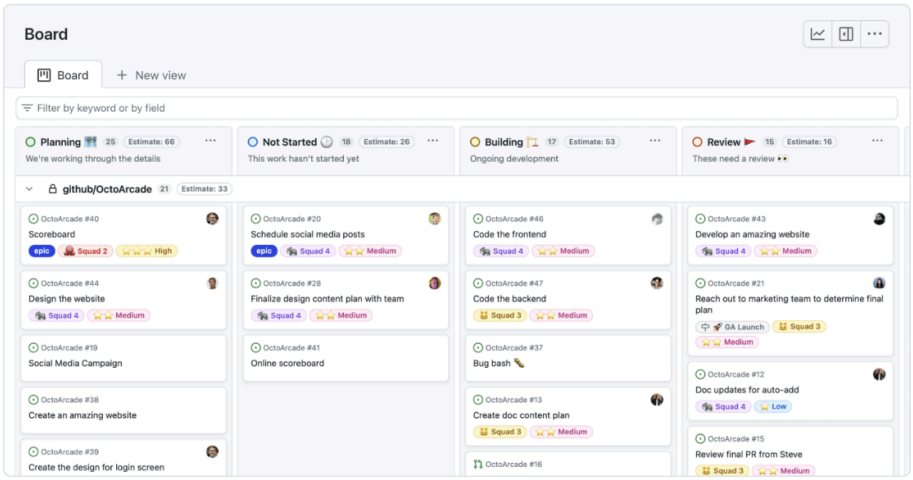
- Visibility Settings:
    - Public: Visible to everyone
    - Private: Restricted to team and collaborators
- Access Roles:
    - Admin: Full control (Head of data, senior colleagues)
    - Write: Can modify (Data team members)
    - Read: View only (Product team members)
    - No Access (Other departments

Different layouts



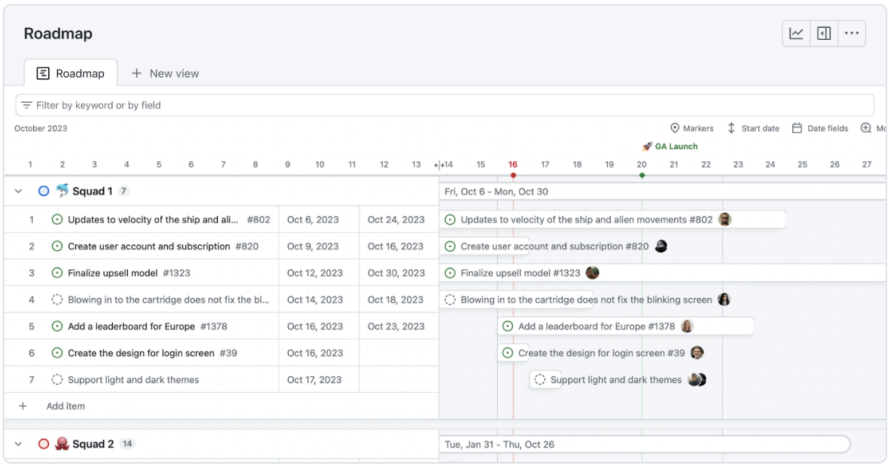
Project name

New board

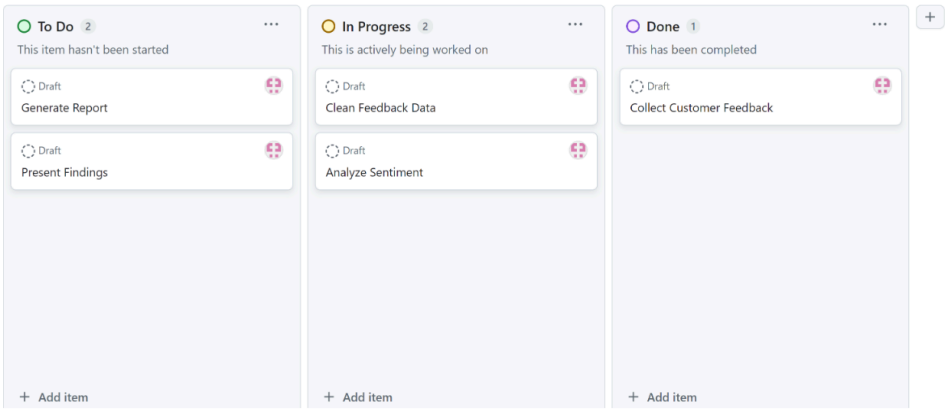


Project name

New roadmap



Customizable layout



GitHub Tools

- GitHub Actions:
    - Streamlines: Automates workflows
    - Automation: Simplifies processes
    - Insights:
- Insights:
    - Real-time data: Tracks progress
    - Spot issues: Spot bottlenecks
    - Smart choices: Informs decisions

Automations Tools in GitHub Projects:

- Built-in Automations:
    - Auto-move tasks based on events and keeping boards updated automatically based on events, like moving tasks when issues are closed or pull requests are merged.
- GitHub Actions:
    - Automate complex workflows like testing, deployment and tagging issues based on criteria
    - Advanced automation with this is using YAML (yet another markup language) files which defines the actions. This triggers tasks on code events like code pushes.
    - It can help streamline routine tasks like setting up environments, scheduling data pipelines, or deploying machine learning models to production.

Project Insights:

- Visualize progress: Real-time and historical data charts
- Spot bottlenecks: Identify issues early with insights
- Chart types: Includes both Current and Historical charts

GitHub Organization:

- Centralized workspace: Manage multiple projects
- Role-based access: Control permissions
- Security policies: Protect our data

Roles and Responsibilities of an organization in GitHub:

- Owner: Full control of the organization
- Member: Standards access for collaboration
- Outside Collaborator: Limited repo access
- Additional Roles:
    - Moderator: Manages interactions in public repos
    - Billing Manager: Handles billing settings
    - Security Manager: Oversees security settings

Managing Teams: Teams simplify management and can be created by owners and members. The teams can be grouped by projects, roles or departments. A nested team allows the mirroring of internal hierarchies, such as a data team with sub teams like data engineers and data scientists. A best practice is to assign permissions to teams, not individuals for easier management.

- Team Roles
    - Team Members: Collaborates on projects
    - Team Maintainer: Manages settings and members

Administration at Repo level:

- Read: View and discuss
- Triage: Manage issues/ PRs
- Write: Push code
- Maintain: Manage the repo without sensitive actions
- Admin: Full control including security management and deletion

Authentication Models:

- Two-factor authentication (2FA): Adds extra security
    - 2FA Methods:
        - Security Keys: The most secure option
        - Time-based One-time Password (TOTP): Get codes via a trusted app
        - SMS Codes: Receive codes through text
        - GitHub Mobile: A convenient option
- Secured Socket Shell (SSH) keys: Secure, key-based access
- Personal Access Tokens (PATs): Controlled API access, they are codes that act like passwords for API access. The use cases are for automating tasks and integrating with Jupyter notebooks.
    - Classic PATs: Broad access, less control
    - Fine-grained PATs: Precise control, tighter security
- Identity Providers (IdP): Centralized user management
- SAML Single Sign-On (SSO): One secure login for multiple apps, it requires a Security Assertion Markup Language (SAML) Identity provider like MIcrosoft Entra ID, Okta or OneLogin. This improves security by centralizing user authentication and reducing the need for multiple passwords.
- OAuth Authorization Model: It is a method for granting third-party access to our GitHub account. It does not require sharing a password, but it gives apps the specific permissions they need while keeping our credentials safe. It is useful for connecting GitHub to data tools like Apache Airflow, allowing specific access to datasets for automated workflows without exposing our entire account.

Team Synchronization:

- Automation: Links GitHub teams to Identity Provider (IdP) groups, automating updates
- Efficiency: Reduces manual work by syncing access with IdP
- Requirements: GitHub Enterprise Cloud, IdP like Okta or Azure AD
- Benefits:
    - Security: Automatically adjusts access based on IdP changes
    - Scalability: Manages large teams without manual updates
    - Efficiency: Automates tasks, reducing manual work

Onboarding (Secure Access Setup):

- Inviting the person to join the organization
- Assign member role
- Request 2FA setup for security
- Adding the person to the respective group in our Identity Provider (IdP)
- Team Synchronization automatically adds the person to the respective team in GitHub. Being added to the team, the person has the correct permission from the start.

Daily Work Flow (Secure Repo Access):

- PAT: The person generates a PAT
- Integration: Connetis GitHub repo to a workbook
- This setup allows the person to run code, commit changes and push efficiently

Offboarding:

- Removing the person from IdP which automatically revokes her GitHub access
- Archive and review the person's contributions

Open Source: this is a software model where code is publicly available for anyone to use, modify and share.

- Benefits of Open Source
    - Rapid Innovation: Progress through divers input
    - High Quality: Peer-reviewed, reliable code
    - Global Access: Free tools for everyone
    - Strong Community: Enhanced collaboration and solutions

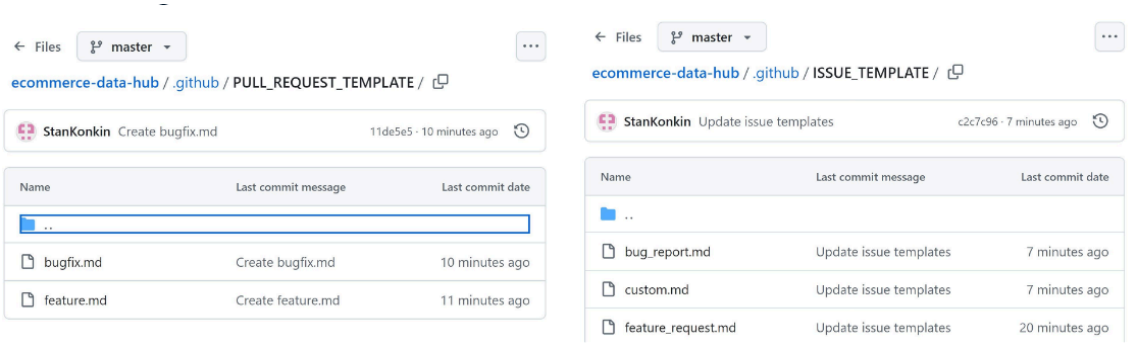
Making Open Source Contributions

- Contribute: Add feature, fix bugs and improve docs
- Skill Growth: Boost your technical expertise
- Build Reputation: Gain recognition in data community
- COllaborate Globally: Work with experts, impact key tools

InnersSource: A development model for internal collaboration that boosts teamwork and sharing through enhanced collaboration and ensures transparency by keeping projects open to all teams.

Organisations have the option to select the repo option of internal, this automatically grants read access to all organization members.

- Standards
    - Clear TItles: use descriptive names like data-pipelines or ml-models
    - Issue Templates: Standardize issue reporting
    - PR Templates: Guide how changes are submitted



Creating .md files in .github help standardize contributions.

Innersource Challenges and Limitations:

- Resistance: Teams may prefer traditional methods
- Security: Risk of exposing sensitive data
- Governance: Needs strong policies
- Resources: Requires extra resources and training

Secure Development: This is essential for all projects to keep the code secure. By automating security tasks, we can reduce manual work and maintain strong security with minimal effort.

- The Essentials
    - Code Scanning: Identify vulnerabilities in our code
    - Secret Scanning: Detect and protect sensitive data
    - Dependency Graph & Depnedabot: Manage dependencies and automate security updates
        - Dependency Graph: It provides a complete view of our project's dependencies, helping us visualize all the libraries our project relies on. It identifies vulnerabilities in our dependency chain and allows us to monitor updates and changes. This tool is crucial for maintaining a secure and up-to-date project
        - Dependabot: Get notifications on vulnerabilities and automatically fix with PRs

Setting Up Security Policies: Creation of a SECURITY.md file can help guide users on how to report security issues and outlines the team's actions. It includes details like contact info and reporting guidelines. Customizing this file ensures clear communication and keeps the project secure.

Removing Sensitive Data: Deleting or editing the file is not enough as the sensitive data is still saved in the commit history. Other tools can be used instead.

- git filter-repo:
    - Control: Detailed management of history
    - Complex: Best for intricate cases
    - Versatile: Supports various tasks
- BFG Repo-Cleaner:
    - Quick: Fast and simple to use
    - Bulk: Great for large-scale deletions
    - Open Source: Focused on speed