GIT & GITHUB

Rondus Technologies

Week 1: Introduction to Version Control and Git Basics

Day 1: Introduction to Version Control

- ❑ What is version control?

What is version control?

Version control, also known as source control or revision control, is a system that records changes to a file or set of files over time so that you can recall specific versions later. It enables multiple contributors to work on a project simultaneously, tracking changes, and coordinating their work.

Key features of version control include:

1. History Tracking:

   - Every change made to files is tracked, including who made the change, when it was made, and what changes were made.

2. Collaboration:

   - Multiple developers can work on the same project concurrently without interfering with each other's changes.

3. Branching and Merging:

   - Version control allows the creation of branches, which are independent lines of development. Changes made in branches can be merged back into the main line of development.

4. Reverting Changes:

   - It provides the ability to revert to previous versions of files or the entire project, helping to undo mistakes or address issues.

What is version control?

5. Parallel Development:

- Different teams or individuals can work on different features or bug fixes simultaneously without conflicts.

6. Conflict Resolution:

- In cases where changes overlap, version control systems provide mechanisms to resolve conflicts and merge changes intelligently.

7. Backup and Recovery:

- Version control serves as a backup mechanism, as all changes are stored and can be retrieved in case of data loss or errors.

8. Audit Trail:

- It maintains a detailed audit trail of changes, promoting transparency and accountability.

What is version control?

Version control systems come in two main types:

- **Centralized Version Control System (CVCS):**

  - Uses a central server to store all files and enables users to check out a copy of the files. Examples include CVS (Concurrent Versions System) and Subversion.

- **Distributed Version Control System (DVCS):**

  - Each user has their own copy of the repository, including the complete history. Examples include Git, Mercurial, and Bazaar.

Git is one of the most widely used distributed version control systems. It is known for its speed, flexibility, and robust branching and merging capabilities. Git, combined with platforms like GitHub, GitLab, or Bitbucket, is widely used for collaborative software development.

Day 2: Introduction to Git

- ❑ Installing Git
- ❑ Basic Git commands (init, add, commit, status, log)
- ❑ Understanding the Git repository structure

Installing Git

To install Git, you can follow the instructions based on your operating system:

Installing Git on Windows:

Git Bash (Git for Windows):

1. Download the installer from [Git for Windows](https://gitforwindows.org/).

2. Run the installer.

3. Follow the installation wizard, selecting default options unless you have specific preferences.

4. Choose the default editor and adjust other settings as needed.

5. Complete the installation.

Git GUI (Git for Windows):

- Git for Windows also provides a graphical user interface (GUI) option. You can choose this during the installation process.

Installing Git

Installing Git on macOS:

Using Homebrew:

1. Open Terminal (you can find it in Applications > Utilities > Terminal).

2. Install Homebrew if you don't have it:

```bash
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

3. Install Git using Homebrew:

```bash
brew install git
```

Installing Git

Verifying Git Installation:

After installation, open a terminal or command prompt and run the following command to verify that Git has been installed successfully:

```bash
git --version
```

This command should display the installed Git version.

Once Git is installed, you can configure it with your name and email using the following commands:

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

Now you're ready to use Git for version control!

Basic Git commands (init, add, commit, status, log)

Here are some of the basic Git commands that are commonly used in a typical Git workflow:

1. `git init`

- Initializes a new Git repository. This command should be run in the root directory of your project.

```bash
git init
```

2. `git add`

- Adds changes in the working directory to the staging area. This is a preparatory step before committing changes.

```bash
git add <file_name>
```

To add all changes:

```bash
git add .
```

Basic Git commands (init, add, commit, status, log)

3. `git commit`

- Records changes in the staging area to the repository. It creates a snapshot of the changes along with a commit message.

```bash
***git commit -m "Your commit message"***
```

4. `git status`

- Displays the status of changes as **untracked, modified, or staged.**

```bash
***git status***
```

Basic Git commands (init, add, commit, status, log)

5. `git log`

- Shows a log of commits, including commit hashes, authors, dates, and commit messages.

```bash
git log
```

To display a more condensed log:

```bash
git log --oneline
```

Understanding the Git repository structure

Understanding the Git repository structure is crucial for grasping how Git manages and stores information about your project. A Git repository consists of several key components:

1. Working Directory:

- This is the directory on your file system where you have all your project files.

- It includes the actual files and directories you're currently working on.

2. Staging Area (Index):

- The staging area is a space where changes are marked for the next commit.

- Before committing, you use `git add` to move changes from the working directory to the staging area.

3. Local Repository:

- This is the `.git` directory within your project, created when you run `git init`.

- It contains the complete history and configuration of the repository.

- The repository holds three main components:

Understanding the Git repository structure

a. Object Database:

- Git stores objects (commits, trees, and blobs) in this database.

b. HEAD:

- A reference to the latest commit in the current branch.

c. Branches and Tags:

- Pointers to specific commits. Branches are mutable and move as new commits are added, while tags are fixed references to specific commits.

Git Object Types:

1. Blob:

- Represents a file. It stores the content but doesn't contain any metadata.

2. Tree:

- Represents a directory. It contains pointers to blobs and other trees, forming the directory structure.

Understanding the Git repository structure

3. Commit:

- Represents a snapshot of the project at a specific point in time.

- It contains metadata (author, date, commit message) and points to the root tree of the project.

Git Commands and Their Impact on the Repository Structure:

1. `git init`:

- Initializes a new Git repository in the current directory.

- Creates a hidden `.git` directory.

2. `git add`:

- Adds changes from the working directory to the staging area.

3. `git commit`:

- Creates a new commit with the changes from the staging area.

- Updates the HEAD reference to point to the latest commit.

Understanding the Git repository structure

4. `git clone`:

- Copies an existing repository, including the entire commit history, into a new directory.

5. `git pull`:

- Fetches changes from a remote repository and merges them into the local branch.

6. `git push`:

- Pushes local commits to a remote repository.

7. `git branch`:

- Creates, lists, or deletes branches.

8. `git merge`:

- Combines changes from different branches.

9. `git tag`:

- Creates a tag for a specific commit, providing a stable reference to that commit.

Stop here

Day 3: Branching and Merging

- ❑ Creating branches (branch, checkout)
- ❑ Merging branches (merge, rebase)
- ❑ Resolving merge conflicts

Creating branches (branch, checkout)

Creating and working with branches is a fundamental aspect of Git, allowing you to work on different features or bug fixes independently. Here are the basic Git commands for creating and switching branches:

1. Creating a Branch (`git branch`):

To create a new branch, you can use the `git branch` command followed by the branch name.

```bash
**git branch <branch_name>**
```

Example:

```bash
**git branch feature-branch**
```

Creating branches (branch, checkout)

3. Creating and Switching to a Branch in One Command (`git switch`):

In Git versions 2.23 and later, you can use the `git switch` command to create and switch to a branch in one step.

```bash
***git switch -c <branch_name>***
```

Example:

```bash
***git switch -c feature-branch***
```

Creating branches (branch, checkout)

4. Viewing Branches (`git branch`):

To see a list of all branches in your repository and identify the currently active branch, you can use the `git branch` command without any arguments.

```bash
git branch
```

The active branch will be highlighted or marked with an asterisk (`*`).

Creating branches (branch, checkout)

5. Deleting a Branch (`git branch -d`):

Once you have merged the changes from a branch into the main branch or another target branch, you can delete the branch using the `git branch -d` command.

```bash
***git branch -d <branch_name>***
```

Example:

```bash
***git branch -d feature-branch***
```

Note:

- Always create and switch to a new branch when working on a new feature or bug fix to keep your changes isolated.

- Regularly merge or rebase your feature branch with the main branch to incorporate the latest changes.

Merging branches (merge, rebase)

Merging branches is a fundamental operation in version control systems like Git. Two common approaches to merging branches are "merge" and "rebase." Let's explore each method:

1. Merge:

Purpose:

- Combines changes from one branch into another while preserving the commit history of both branches.

Steps:

1. Checkout the Destination Branch:

```bash
***git checkout destination_branch***
```

2. Merge the Source Branch:

```bash
***git merge source_branch***
```

- Git creates a new merge commit that has two parent commits, preserving the history of both branches.

Merging branches (merge, rebase)

3. Resolve Conflicts (if any):

- If there are conflicting changes between the branches, Git will prompt you to resolve conflicts manually.

4. Commit the Merge:

- After resolving conflicts, commit the changes to complete the merge.

Pros:

- Preserves the original commit history of both branches.
- Simple to understand and use.

Cons:

- May result in a more complex commit history if there are frequent merges.

Merging branches (merge, rebase)

2. Rebase:

Purpose:

- Rewrites the commit history of the source branch onto the tip of the destination branch.

Steps:

1. Checkout the Source Branch:

```bash
**git checkout source_branch**
```

2. Rebase onto the Destination Branch:

```bash
**git rebase destination_branch**
```

- Git reapplies each commit from the source branch onto the tip of the destination branch.

3. Resolve Conflicts (if any):

- Similar to merging, conflicts may arise that need to be resolved.

Merging branches (merge, rebase)

4. Complete the Rebase:

- After resolving conflicts, continue the rebase with:

```bash
git rebase --continue
```

Pros:

- Results in a linear and cleaner commit history.

- Reduces the number of merge commits.

Cons:

- Alters commit history, which can cause issues if the branches are shared with others.

- Conflicts may be more challenging to resolve due to the linear reapplication of commits.

Merging branches (merge, rebase)

Choosing Between Merge and Rebase:

1. **Use Merge When:**
   - Collaborating with others on a shared branch.
   - A feature or bugfix branch is relatively short-lived.
   - Commit history clarity is a lower priority.

2. **Use Rebase When**:
   - Working on a feature branch that you want to appear as a linear sequence of commits.
   - Cleaning up your local branch before pushing it to a shared repository.
   - Commit history clarity is a higher priority.

Merging branches (merge, rebase)

Important Considerations:

- **Shared Branches:**

  - Avoid rebasing branches that are shared with others, as it can cause conflicts and disrupt their work.

- Force Push:

  - After rebasing, a force push (`git push --force`) may be necessary to update the remote branch. Exercise caution when force pushing, especially on shared branches.

- Consistency:

  - Choose a consistent approach within your team to maintain a clean and understandable project history.

Ultimately, the choice between merging and rebasing depends on the specific use case, team workflow, and the importance of maintaining a clean and linear commit history.

Resolving merge conflicts

Merge conflicts occur when Git cannot automatically merge changes from two branches due to conflicting modifications in the same part of a file. Resolving merge conflicts involves manually addressing these conflicts and finalizing the merge. Here's a step-by-step guide on how to resolve merge conflicts:

1. **Initiate Merge:**

   - Suppose you are on the branch where you want to merge changes (e.g., the destination branch).

   ```bash
   git checkout destination_branch
   git merge source_branch
   ```

2. **Conflict Notification:**

   - If conflicts occur, Git will notify you about the conflicting files.

3. **View Conflicts:**

   - Open the conflicted file(s) in a text editor. Within the file, Git marks the conflicting sections with conflict markers `<<<<<<<`, `=======`, and `>>>>>>>`.

Resolving merge conflicts

4. **Understanding Conflict Markers:**

   - The section between `<<<<<<<` and `=======` represents your local changes.

   - The section between `=======` and `>>>>>>>` represents changes from the branch being merged.

5. **Resolution:**

   - Manually edit the file to resolve conflicts. Decide which changes to keep, modify, or discard.

6. **Remove Conflict Markers:**

   - After resolving conflicts, remove the conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) and ensure the file reflects the desired changes.

7. **Mark as Resolved:**

   - Mark the file as resolved:

   ```bash
   git add conflicted_file
   ```

Resolving merge conflicts

8. **Continue Merge:**
   - Complete the merge process:
   ```bash
   ***git merge --continue***
   ```

9. **Commit the Merge:**
   - If not already done, commit the resolved changes:
   ```bash
   ***git commit***
   ```

10. **Verification:**
    - Verify that all conflicts are resolved:
    ```bash
    ***git status***
    ```

Resolving merge conflicts

Example: Resolving a Merge Conflict:

Suppose you have a conflict in a file `example.txt`. After opening the file, you see:

```plaintext
<<<<<<< HEAD
This is the content on the destination branch.
=======
This is the content on the source branch.
>>>>>>> source_branch
```

You decide to keep both changes, so you modify the file to:

```plaintext
This is the content on the destination branch and the source branch.
```

After saving the file, you mark it as resolved:

```bash
***git add example.txt***
```

Resolving merge conflicts

Then, you continue the merge:

```bash
***git merge --continue***
```

And finally, commit the resolved changes:

```bash
***git commit***
```

Additional Tips:

- Aborting the Merge:
  - If you encounter difficulties resolving conflicts, you can abort the merge:

    ```bash
    ***git merge --abort***
    ```

  - This reverts the branch to its state before the merge.

Resolving merge conflicts

- Interactive Merge Tool:

  - Use Git's interactive merge tool for a visual aid in resolving conflicts:

    ```bash
    git mergetool
    ```

- Review Changes:

  - After resolving conflicts and completing the merge, thoroughly review the changes to ensure the desired modifications are included.

Resolving merge conflicts is a common part of collaborative development. Clear communication and coordination within the team help manage conflicts effectively.

Day 4: Collaborating with Remote Repositories

- ❑ Cloning repositories
- ❑ Pushing and pulling changes (push, pull)
- ❑ Fetching changes (fetch)

Cloning repositories

Cloning a repository in Git involves creating a local copy of a remote repository. This is a common operation when you want to work on a project, contribute to open-source projects, or collaborate with others. Here's a step-by-step guide on how to clone a Git repository:

1. Get Repository URL:

   - Obtain the URL of the Git repository you want to clone. This can usually be found on the project's hosting platform (e.g., GitHub, GitLab, Bitbucket).

2. Open Terminal (or Command Prompt):

   - Open the command-line interface on your computer.

3. Navigate to Desired Directory:

   - Navigate to the directory where you want to clone the repository.

   ```bash
   cd path/to/desired/directory
   ```

Cloning repositories

4. Clone Repository:

   - Use the `git clone` command followed by the repository URL.

   ```bash
   git clone repository_url
   ```

   - For example:

   ```bash
   git clone https://github.com/example/repo.git
   ```

5. Authentication (if required):

   - If the repository is private or requires authentication, Git may prompt you to enter your username and password or use other authentication methods (SSH keys, personal access tokens).

6. Verify Cloning Success:

   - Once the cloning process is complete, you'll have a local copy of the repository in the specified directory.

Cloning repositories

Example:

```bash
Navigate to the directory where you want to clone the repository
cd path/to/desired/directory

Clone the repository (replace repository_url with the actual URL)
git clone https://github.com/example/repo.git
```

Additional Cloning Options:

- Clone into a Specific Directory:

  ```bash
  git clone repository_url target_directory
  ```

- Clone a Specific Branch:

  ```bash
  git clone -b branch_name repository_url
  git clone –b dev https://github.com/itoroukpe/Rondus003.git
  ```

Cloning repositories

- Clone a Specific Branch:

```bash
git clone -b branch_name repository_url
```

- Shallow Clone (Limited History):

```bash
git clone --depth 1 repository_url
```

- Clone with Submodules:

```bash
git clone --recursive repository_url
```

Cloning repositories

Notes:

- The `git clone` command creates a new directory with the same name as the repository (or the specified target directory) and downloads all the files from the remote repository.

- The local copy is a fully functional Git repository with its own copy of the commit history.

- After cloning, you can make changes locally, commit them, and push them back to the remote repository.

- Ensure you have Git installed on your machine before attempting to clone a repository.

Cloning is an essential operation when working with Git, providing you with a local working copy of a repository that you can modify and contribute to.

Pushing and pulling changes (push, pull)

Pushing and pulling changes are fundamental Git operations used to synchronize changes between a local Git repository and a remote repository (such as GitHub, GitLab, or Bitbucket). These operations allow you to share your local changes with others and fetch updates made by collaborators. Here's a guide on how to push and pull changes:

Pushing Changes:

1. Commit Your Changes:

   - Before pushing, ensure you have committed your changes locally.

     ```bash
     git commit -m "Your commit message"
     ```

2. Push to Remote Repository:

   - Use the `git push` command to send your committed changes to the remote repository.

     ```bash
     git push origin branch_name
     ```

   - Replace `branch_name` with the name of the branch you are pushing.

Pushing and pulling changes (push, pull)

Example:

```bash
Push changes to the master branch
git push origin master
```

Notes:

- The `origin` is the default name given to the remote repository. You may have a different remote name.

- If you are pushing a new branch for the first time, you may need to set the upstream branch:

```bash
git push -u origin branch_name
```

Pushing and pulling changes (push, pull)

"Pulling Changes:

1. Navigate to Your Local Repository:

   - Open a terminal and navigate to your local repository.

2. Fetch Changes from Remote Repository:

   - Use the `git fetch` command to fetch changes from the remote repository. This operation does not merge the changes into your local branch.

    ```bash

    git fetch origin

    ```

View remote branches

    git branch -r

3. Merge Changes into Local Branch:

   - Use the `git merge` or `git pull` command to merge the fetched changes into your local branch.

    ```bash

    git merge origin/branch_name

    ```

   - Alternatively, use `git pull` to fetch and merge in one command:

    ```bash

    git pull origin branch_name

    ```

Pushing and pulling changes (push, pull)

Use git branch -r to see the remote branches and their latest commits.

```bash
  git branch -r
```

Example:

```bash
Fetch changes from the remote repository
git fetch origin

Merge changes into the local branch
git merge origin/master
```

Notes:

- The `origin/branch_name` represents the branch on the remote repository.

- If you've set up tracking branches, you can use `git pull` without specifying the remote and branch.

Pushing and pulling changes (push, pull)

Tips:

- Pull with Rebase:

  - Use `git pull --rebase` to incorporate remote changes while keeping a linear commit history.

- Force Push (Caution):

  - If you need to force-push your changes (overwrite remote history), use `git push --force` with caution, especially on shared branches.

- Fetch vs. Pull:

  - `git fetch` fetches changes without merging, providing an opportunity to review changes before merging. `git pull` fetches and merges in one step.

- Review Changes Before Merging:

  - Always review changes fetched from the remote repository before merging to avoid unexpected conflicts.

These commands are crucial for collaborative development, enabling you to share your work with others and incorporate changes made by your team into your local repository.

Day 5: Git Best Practices

- ❑ Gitignore files
- ❑ Undoing changes (reset, revert, checkout)

Gitignore files

A `.gitignore` file is used to specify intentionally untracked files and directories that Git should ignore. This is particularly useful for excluding files generated during the build process, temporary files, and sensitive information like passwords or API keys from being accidentally committed to the version control system. Here's a guide on how to use a `.gitignore` file:

1. Create a `.gitignore` File:

   - Create a file named `.gitignore` in the root directory of your Git repository.

2. Add Patterns to Ignore:

   - Add file and directory patterns to the `.gitignore` file, specifying what Git should ignore.

3. Basic Patterns:

   - Use the following basic patterns:

     - `*.log` - Ignores all files with the `.log` extension.

     - `/logs/` - Ignores the entire `logs/` directory.

     - `/temp` - Ignores a directory named `temp` in the root.

     - `secret.txt` - Ignores a file named `secret.txt`.

Gitignore files

4. Comments:
   - Add comments by starting lines with a `#` symbol.

   ```plaintext
   Ignore compiled binaries
   *.exe
   ```

5. Wildcards:
   - Use wildcards for broader matching.

   ```plaintext
   Ignore all .a files
   *.a
   ```

6. Negation (!):
   - Use negation to exclude certain files or directories from being ignored.

   ```plaintext
   Ignore all files except .md files
   *
   !*.md
   ```

Gitignore files

7. Templates:

- Use pre-configured templates for common project types, like Node, Python, or Visual Studio. You can find these templates on [GitHub's Gitignore repository](https://github.com/github/gitignore).

Example `.gitignore` File:

```plaintext
Ignore compiled binaries
*.exe
*.dll
*.so
*.dylib

Ignore dependencies
/node_modules/
/bower_components/
```

Gitignore files

Ignore dependencies

/node_modules/

/bower_components/

Ignore log files and temporary files

/log/

/tmp/

/temp/

Ignore sensitive information

secrets.txt

credentials.json

```

Applying `.gitignore`:

- The rules specified in the `.gitignore` file only apply to untracked files. If a file is already tracked by Git, adding it to `.gitignore` will not untrack it. You may need to remove the file from the repository.

- After adding or modifying a `.gitignore` file, you may want to run `git rm --cached <file>` to untrack files that match the patterns in the `.gitignore` file.

- Verify the effectiveness of your `.gitignore` file by using `git status` to see if untracked files are correctly ignored.
```

Gitignore files

Global `.gitignore`:

- You can create a global `.gitignore` file that applies to all your Git repositories by configuring it in your Git configuration:

```bash
git config --global core.excludesfile ~/.gitignore_global
```

Then, create a `.gitignore_global` file in your home directory.

Note:

- `.gitignore` files are case-sensitive. Ensure that the patterns match the case of the files or directories you want to ignore.

- `.gitignore` is not a security feature. Never rely on it to secure sensitive information. Always use proper access controls and encryption for sensitive data.

The effective use of `.gitignore` files helps keep your repositories clean and avoids unintentional inclusion of files that should not be tracked by version control.

Undoing changes (reset, revert, checkout)

Undoing changes in Git can be done using various commands like **`git reset`, `git revert`, and `git checkout`**. Each command serves a different purpose and should be used based on your specific requirements. Here's a guide on how to undo changes using these commands:

1. `git reset` - Discard Commits:

Use `git reset` to move the branch pointer backward, effectively discarding commits. There are different modes of `git reset`:

Soft Reset:

- Preserves changes in your working directory and staging area.

```bash
Soft reset to undo the last commit
git reset --soft HEAD^
```

Mixed Reset (Default):

- Preserves changes in your working directory but resets the staging area.

```bash
Mixed reset to unstage changes
git reset HEAD^
```

Undoing changes (reset, revert, checkout)

Hard Reset:

- Discards changes in your working directory, staging area, and commits.

```bash
Hard reset to completely remove the last commit
git reset --hard HEAD^
```

2. `git revert` - Create Revert Commits:

Use `git revert` to create a new commit that undoes a previous commit.

```bash
Revert the last commit
git revert HEAD
```

Undoing changes (reset, revert, checkout)

3. `git checkout` - Discard Local Changes:

Use `git checkout` to discard changes in your working directory. This command is not used to undo commits but to discard local changes.

Discard Changes in a File:

```bash
Discard changes in a specific file
git checkout -- file_name
```

Discard All Changes:

```bash
Discard all changes in the working directory
git checkout -- .
```

Undoing changes (reset, revert, checkout)

Tips:

- Use `git reset` for Local Branches:

  - Use `git reset` when working with local branches, especially for undoing commits that have not been pushed.

- Use `git revert` for Shared Branches:

  - Use `git revert` when working with shared branches to avoid rewriting history and affecting collaborators.

- Be Cautious with `git reset --hard`:

  - Avoid using `git reset --hard` if the changes you want to undo have not been committed or pushed.

- Create Backups:

  - Before performing any reset or revert, consider creating a backup branch or making a copy of your repository.

- Review Changes Before Undoing:

  - Use `git log` and `git diff` to review changes before deciding to undo them.

Undoing changes (reset, revert, checkout)

Example Workflow:

Suppose you made a commit that you want to undo:

```bash
View commit history
git log

Choose the appropriate undo method
For example, use git reset to discard the last commit
git reset --hard HEAD^
```

Remember to replace `HEAD^` with the appropriate commit reference or branch name based on your situation.

Choose the appropriate method based on your workflow and whether the changes are local or shared with collaborators. Always be cautious when undoing changes, especially if the changes have been shared with others.

Week 3: Introduction to GitHub

Day 11: Introduction to GitHub

- ❑ Overview of GitHub and its features
- ❑ Creating a GitHub account

Overview of GitHub and its features

GitHub is a web-based platform that provides version control and collaborative features for software development. It is widely used for hosting and managing Git repositories. Here's an overview of GitHub and its key features:

1. Repository Hosting:

   - GitHub hosts Git repositories, providing a centralized location for code storage and collaboration.

2. Collaboration:

   - Multiple developers can work on a project simultaneously, contributing to different branches and features.

3. Version Control:

   - GitHub uses Git for version control, allowing developers to track changes, revert to previous states, and collaborate efficiently.

4. Pull Requests:

   - Developers can propose changes by creating pull requests, enabling others to review, comment, and suggest modifications before merging.

Overview of GitHub and its features

5. Branching:

- GitHub supports branching, allowing developers to work on features or bug fixes in isolated branches before merging into the main branch.

6. Issues and Bug Tracking:

- Users can create issues to report bugs, request features, or discuss improvements. Issues can be assigned, labeled, and linked to pull requests.

7. Milestones:

- Milestones help organize and track progress by grouping related issues and pull requests together.

8. Wikis:

- GitHub provides wikis for documenting project details, guidelines, and other relevant information.

9. Actions:

- GitHub Actions enables automated workflows, including continuous integration and delivery (CI/CD), allowing developers to automate build, test, and deployment processes.

Overview of GitHub and its features

9. Actions:

- GitHub Actions enables automated workflows, including continuous integration and delivery (CI/CD), allowing developers to automate build, test, and deployment processes.

10. Code Review:

- GitHub facilitates code reviews through inline comments, suggesting changes, and reviewing pull requests.

11. Security Alerts:

- GitHub automatically scans repositories for known vulnerabilities and provides security alerts.

12. GitHub Pages:

- Users can host static websites directly from their GitHub repositories using GitHub Pages.

13. Projects:

- GitHub Projects provide a visual way to manage and organize work, including tasks, notes, and a kanban board.

Overview of GitHub and its features

14. Community and Social Features:

- GitHub fosters a community around code repositories, allowing users to follow repositories, star projects, and contribute to open-source projects.

15. Graphs and Insights:

- GitHub provides graphs and insights to track repository activity, contributors, and code frequency.

16. GitHub Gist:

- GitHub Gist allows users to share and collaborate on snippets of code, text, or markdown files.

17. Search and Explore:

- Users can search for repositories, topics, and developers, making it easy to discover and explore projects.

Overview of GitHub and its features

18. API and Integrations:

- GitHub offers a comprehensive API, enabling integrations with various tools and services.

19. Organization Accounts:

- Organizations on GitHub allow teams to collaborate and manage access to repositories.

20. Desktop Application:

- GitHub provides a desktop application for a user-friendly interface to manage repositories and perform Git operations.

GitHub plays a crucial role in modern software development, providing a platform that integrates version control, collaboration, and automation. Its user-friendly interface and extensive features make it a preferred choice for individual developers, teams, and open-source communities.

Creating a GitHub account

Creating a GitHub account is a straightforward process. Follow these steps to create your GitHub account:

Visit the GitHub Website:

Open your web browser and go to the GitHub website: https://github.com/

Click on "Sign Up":

On the GitHub homepage, click on the "Sign Up" button in the top right corner.

Provide Your Information:

Fill in the required information in the provided form. This typically includes:

Your username (this will be part of your GitHub URL).

Your email address.

A strong and secure password.

Stopped here.

Day 12: GitHub Repositories and Forking

- ❑ Creating repositories on GitHub
- ❑ Forking repositories

Creating repositories on GitHub

1. Sign In to Your GitHub Account:
   - Open your web browser and go to GitHub. Sign in to your GitHub account if you're not already logged in.

2. Navigate to Your Profile:
   - Click on your profile picture in the top right corner of the GitHub homepage. From the dropdown menu, select "Your repositories.“

3. Click on "New":
   - On the "Your repositories" page, click on the green "New" button.

4. Fill in Repository Information:

- Provide the following information for your new repository:
  - Repository Name: Choose a unique and descriptive name for your repository.
  - Description (Optional): Add a brief description of your project.
  - Public or Private: Choose whether the repository will be public (visible to everyone) or private (accessible only to selected collaborators).
  - Initialize this repository with a README (Optional): If checked, GitHub will create an initial README file in your repository.

Forking repositories

Forking a repository on GitHub allows you to create a personal copy of someone else's project. This copy is independent of the original repository, and you can make changes to your fork without affecting the original project. Here's how to fork a repository on GitHub:

1. Visit the Repository:

   - Go to the GitHub repository you want to fork by navigating to its URL (e.g., `https://github.com/original-owner/original-repo`).

2. Fork the Repository:

   - In the top right corner of the repository page, click the "Fork" button.

3. Choose Your Account:

   - GitHub will prompt you to choose where to fork the repository. Select your GitHub account as the destination.

4. Wait for the Fork to Complete:

   - GitHub will create a fork of the repository under your account. This process may take a few moments.

5. Navigate to Your Fork:

   - After forking is complete, you'll be redirected to your forked copy of the repository under your GitHub account. The URL will be `https://github.com/your-username/original-repo`.

Now you have a personal copy (fork) of the original repository. You can clone your fork to your local machine, make changes, and push them back to your fork on GitHub. If you want to contribute your changes back to the original repository, you can create a pull request.

Forking repositories

Cloning Your Fork:

1. Open a Terminal:
   - Open a terminal on your local machine.

2. Clone Your Fork:
   - Use the `git clone` command to clone your fork to your local machine.

   ```bash
   git clone https://github.com/your-username/original-repo.git
   ```

   - Replace `your-username` with your GitHub username and `original-repo` with the name of the repository you forked.

3. Navigate to the Cloned Repository:
   - Change into the cloned repository's directory.

   ```bash
   cd original-repo
   ```

Now you have a local copy of your fork, and you can start making changes to the code.

Forking repositories

Syncing Your Fork with the Original Repository:

If the original repository is updated, you might want to sync your fork with those changes. Here's how you can do it:

1. Add the Original Repository as a Remote:

   - Add the original repository as a remote with a name like `upstream`.

     ```bash

     ***git remote add upstream https://github.com/original-owner/original-repo.git***

     ```

2. Fetch Changes from the Original Repository:

   - Fetch changes from the original repository.

     ```bash

     git fetch upstream

     ```

Forking repositories

3. Merge or Rebase Changes:

- Merge or rebase the changes from the original repository into your local branch.

```bash
***git merge upstream/main***
```

or

```bash
***git rebase upstream/main***
```

4. Push Changes to Your Fork:

- Push the changes back to your fork on GitHub.

```bash
***git push origin main***
```

Now your fork is synced with the latest changes from the original repository.

Forking is a common practice in open-source collaboration, allowing developers to contribute to projects without having direct write access to the original repository.

Day 13: GitHub Pull Requests

- ❑ Creating and managing pull requests
- ❑ Code reviews on GitHub

Creating and managing pull requests

Creating and managing pull requests (PRs) is a fundamental aspect of collaborative development on GitHub. Pull requests allow you to propose changes to a repository and ask that they be reviewed and merged into the main codebase. Here's a step-by-step guide on how to create and manage pull requests:

Creating a Pull Request:

1. Fork the Repository:

   - If you haven't already, fork the repository you want to contribute to. Follow the steps mentioned earlier for forking a repository.

2. Clone Your Fork:

   - Clone your forked repository to your local machine using `git clone`.

3. Create a New Branch:

   - Create a new branch in your local repository to work on your changes.

   ```bash
   git checkout -b feature-branch
   ```

Creating and managing pull requests

4. Make Changes:

- Make the necessary changes to the codebase. Commit your changes with descriptive messages.

```bash
git add .
git commit -m "Implement new feature"
```

5. Push Changes to Your Fork:

- Push your changes to your fork on GitHub.

```bash
git push origin feature-branch
```

6. Create a Pull Request:

- Visit your fork on GitHub. GitHub will detect the new branch and display a "Compare & pull request" button. Click on it.

Creating and managing pull requests

7. Review Changes:

- Review the changes you made in the pull request. Provide a title and a description summarizing your changes.

8. Create Pull Request:

- Click the "Create pull request" button to submit your pull request.

Managing Pull Requests:

Once a pull request is created, you can manage and collaborate on it:

1. Review Changes:

- Reviewers and collaborators can review the changes by looking at the diff, adding comments, and providing feedback.

2. Discussion:

- Engage in discussions with collaborators through comments in the pull request. Discuss code changes, propose alternatives, and address concerns.

3. Continuous Integration (CI):

- If the repository has CI configured, GitHub will run tests on your changes. Ensure that the CI build passes.

Creating and managing pull requests

4. Address Feedback:

- If feedback is provided, make the necessary changes in your local branch, commit, and push them to your fork. The pull request will be automatically updated.

5. Rebase or Merge from Upstream (Optional):

- If changes were made to the original repository after you forked it, you might need to rebase or merge those changes into your branch.

6. Squash Commits (Optional):

- If you have multiple small commits, consider squashing them into a single commit before merging. This helps keep the commit history clean.

7. Merge Pull Request:

- Once the pull request is approved and passes all checks, you or a repository collaborator can merge it.

8. Delete Branch (Optional):

- After merging, you can choose to delete the branch associated with the pull request, especially if it was a feature branch.

Pull requests are a collaborative and transparent way to propose and discuss changes in a Git repository. They provide a structured workflow for code review and integration, making it easier for teams to work together on projects.