**Day1**

**1. Getting Started**

**I. Introduction**

**Introduction to git**

* Version control is essential for collaborative software development.
* Git is a distributed version control system widely used for tracking changes in source code.

**Why Version Control?**

* Collaborative work: Enables multiple developers to work on the same project simultaneously.
* History tracking: Keeps a record of changes made to the codebase.
* Branching and merging: Facilitates parallel development and code integration.

**II. About Version Control**

**Types of Version Control Systems**

* Centralized VCS: Single repository, requires constant network connection.
* Distributed VCS: Multiple local repositories, independent of network.

**Git as a Distributed VCS**

* Each user has a complete copy of the repository.
* Changes are tracked independently, allowing offline work.

**III. A Short History of Git**

**Inception**

* Created by Linus Torvalds in 2005.
* Developed for managing the Linux kernel source code.

**Key Milestones**

* Git becomes widely adopted in open-source projects.
* GitHub, GitLab, and Bitbucket provide hosting and collaboration.

**IV. What is Git?**

**Definition**

* Git is a distributed version control system.
* Manages and tracks changes in source code during software development.

**Core Concepts**

* Repository: Storage for project files and version history.
* Commit: Snapshot of changes at a specific point in time.
* Branch: Independent line of development.

**V. The Command Line**

**Basic Commands**

* git init: Initializes a new Git repository.
* git add: Adds changes to the staging area.
* git commit: Records changes to the repository.

**Advanced Commands**

* git branch: Manages branches.
* git merge: Integrates changes from different branches.

**VI. Installing Git**

**Platforms**

* Git supports Windows, macOS, and Linux.
* Installation instructions available on the official Git website.

**Verification**

* After installation, verify by running git --version in the command line.

**VII. First-Time Git Setup**

**Configuration**

* Set username and email: git config --global user.name "Your Name" and git config --global user.email "your@email.com".
* Customize Git settings for user preferences.

**VIII. Getting Help**

**Documentation**

* Official Git documentation available online.
* Use git --help for command-specific help.
* Community forums and Q&A sites for additional support.

**IX. Summary**

**Key Takeaways**

* Git is a distributed version control system.
* Essential commands include init, add, and commit.
* Understanding branches is crucial for collaborative development.
* Explore documentation and seek help when needed.

Certainly! Here are detailed PowerPoint notes on the Git Basics topics:

**2. Git Basics**

**I. Getting a Git Repository**

**Initializing a Repository**

* git init: Initializes a new Git repository in the current directory.
* Creates a hidden .git directory to store configuration and version history.

**Cloning a Repository**

* git clone <repository URL>: Downloads an existing Git repository to your local machine.
* Establishes a connection with the remote repository.

**II. Recording Changes to the Repository**

**Tracking Changes**

* git status: Shows the status of changes as untracked, modified, or staged.
* git add <file>: Adds changes to the staging area.

**Committing Changes**

* git commit -m "Commit message": Records changes in the repository.
* Commits create a snapshot in the version history.

**III. Viewing the Commit History**

**Git Log**

* git log: Displays a chronological list of commits.
* Shows commit hash, author, date, and commit message.

**Customizing Log Output**

* git log --oneline: Condenses commit information to one line.
* git log --graph: Visualizes branch history.

**IV. Undoing Things**

**Discarding Changes**

* git restore <file>: Discards changes in a file.
* git restore --source=<commit> <file>: Restores a file to a specific commit.

**Amending Commits**

* git commit --amend: Adds changes to the previous commit.
* Useful for fixing mistakes in the last commit.

**V. Working with Remotes**

**Adding Remotes**

* git remote add <name> <url>: Links a local repository to a remote repository.
* Common remotes include GitHub or GitLab.

**Fetching and Pulling**

* git fetch: Retrieves changes from the remote repository.
* git pull: Fetches changes and integrates them into the current branch.

**VI. Tagging**

**Creating Tags**

* git tag <tagname>: Marks a specific commit with a tag.
* Useful for releasing versions or marking significant points.

**Pushing Tags**

* git push origin <tagname>: Shares tags with the remote repository.

**VII. Git Aliases**

**Creating Aliases**

* git config --global alias.<alias-name> <git-command>: Sets up custom shorthand for Git commands.
* Improves efficiency and reduces typing.

**VIII. Summary**

**Key Takeaways**

* Git repositories are initialized with git init or cloned with git clone.
* Changes are recorded with git add and git commit.
* View commit history with git log and customize output.
* Undo changes with git restore and amend commits.
* Remotes are managed using git remote.
* Tags mark important points in history.
* Git aliases streamline common commands.

Certainly! Here are detailed PowerPoint notes on the Git Branching topics:

**Day2**

**3. Git Branching**

**I. Branches in a Nutshell**

**Introduction**

* Branching is a fundamental concept in Git.
* Allows for parallel development and isolation of features.

**Main Branches**

* master (or main): Default branch, represents the main line of development.
* develop: Common for ongoing development work.

**Feature Branches**

* Created for developing specific features or bug fixes.
* Isolates changes until ready for integration.

**II. Basic Branching and Merging**

**Creating a Branch**

* git branch <branch-name>: Creates a new branch.
* git checkout <branch-name> or git switch <branch-name>: Switches to the new branch.

**Basic Merging**

* git merge <branch-name>: Integrates changes from one branch into another.
* Merges can result in a merge commit.

**III. Branch Management**

**Viewing Branches**

* git branch: Lists all local branches.
* git branch -d <branch-name>: Deletes a branch after merging.

**Switching Branches**

* git checkout <branch-name> or git switch <branch-name>: Moves to a different branch.
* git checkout -b <new-branch> or git switch -c <new-branch>: Creates and switches to a new branch.

**IV. Branching Workflows**

**Feature Branch Workflow**

* Create a branch for each feature.
* Merge back into develop or main when ready.

**Gitflow Workflow**

* Uses master, develop, feature branches, release branches, and hotfix branches.
* Provides a structured approach to software development.

**V. Remote Branches**

**Tracking Remote Branches**

* git fetch origin: Retrieves information about remote branches.
* git branch -r: Lists remote branches.
* git checkout -b <local-branch> origin/<remote-branch>: Creates a local branch tracking a remote branch.

**VI. Rebasing**

**Rebasing vs. Merging**

* git rebase <branch-name>: Incorporates changes from one branch into another by moving or combining commits.
* Results in a linear commit history.

**Interactive Rebasing**

* git rebase -i <commit>: Allows for interactive rebasing, such as squashing commits.

**VII. Summary**

**Key Takeaways**

* Branches provide isolation for development work.
* Basic branching involves creating, switching, and merging branches.
* Branch management includes viewing, deleting, and switching branches.
* Different branching workflows suit various development scenarios.
* Remote branches allow collaboration with others.
* Rebasing can be an alternative to merging, providing a linear history.

Certainly! Here are detailed PowerPoint notes on the Git on the Server topics:

**4. Git on the Server**

**I. The Protocols**

**Git Protocols Overview**

* Git supports several protocols for server communication.
* Common protocols include HTTP, SSH, and Git protocol.

**Choosing a Protocol**

* HTTPS: Secure and widely supported.
* SSH: Offers secure authentication.
* Git Protocol: Lightweight and efficient for read-only access.

**II. Getting Git on a Server**

**Installing Git**

* Ensure Git is installed on the server.
* Available for Windows, macOS, and Linux.

**Basic Configuration**

* Set up a Git user account on the server.
* Configure user details: git config --global user.name "Your Name".

**III. Generating Your SSH Public Key**

**SSH Key Generation**

* Generate SSH keys using ssh-keygen.
* Public key is shared with the server.

**Adding SSH Key to Server**

* Copy the public key to the server's authorized keys file.

**IV. Setting Up the Server**

**Creating a Git Repository**

* Initialize a new repository on the server: git init --bare.
* Bare repositories have no working directory.

**Setting Up Permissions**

* Ensure proper file permissions for security.
* Restrict access to authorized users.

**V. Git Daemon**

**Git Daemon Overview**

* Lightweight and fast Git server.
* Supports the Git protocol.

**Starting Git Daemon**

* Run git daemon on the server.
* Enables Git clients to fetch and clone repositories.

**VI. Smart HTTP**

**HTTP Protocol Configuration**

* Enables Git over HTTP for read and write access.
* Requires a web server (e.g., Apache, Nginx) with Git support.

**Authentication**

* Utilizes HTTP authentication or OAuth for secure access.

**VII. GitWeb**

**GitWeb Overview**

* Web interface for browsing Git repositories.
* Provides a visual representation of branches and commits.

**Configuration**

* Set up GitWeb on the server.
* Configure access and appearance settings.

**VIII. GitLab**

**GitLab as a Git Server**

* GitLab is a web-based Git repository manager.
* Offers additional features like issue tracking, CI/CD, and more.

**Installing GitLab**

* Follow GitLab installation instructions.
* Configure settings and integrate with other services.

**IX. Third Party Hosted Options**

**GitHub, GitLab, Bitbucket**

* Popular platforms for hosting Git repositories.
* Provide collaboration features, issue tracking, and more.

**Choosing a Hosted Option**

* Consider factors like pricing, features, and integration.

**X. Summary**

**Key Takeaways**

* Choose the appropriate protocol based on security and accessibility needs.
* Set up Git on the server, including SSH key generation.
* Git Daemon provides a lightweight server option.
* Smart HTTP enables Git over HTTP with authentication.
* GitWeb and GitLab offer web interfaces for repository management.
* Consider third-party hosted options for convenience and additional features.

**5. Distributed Git**

**I. Distributed Workflows**

**Understanding Distributed Workflows**

* Distributed version control allows for decentralized collaboration.
* Various workflows accommodate different project structures.

**Centralized Workflow**

* Single shared repository.
* Contributors clone, make changes, and push to the central repository.

**Feature Branch Workflow**

* Each feature or bug fix gets its own branch.
* Branches are merged into the main branch when ready.

**Forking Workflow**

* Contributors fork the repository to their accounts.
* Pull requests are used to propose changes to the original repository.

**II. Contributing to a Project**

**Forking a Repository**

* Fork the repository on a platform like GitHub.
* Creates a copy under the contributor's account.

**Cloning and Branching**

* Clone the forked repository to the local machine.
* Create a branch for the intended contribution.

**Making Changes**

* Make and commit changes to the local branch.
* Regularly pull changes from the original repository to stay updated.

**Pushing Changes and Pull Requests**

* Push changes to the forked repository.
* Create a pull request to propose changes to the original repository.

**III. Maintaining a Project**

**Reviewing Pull Requests**

* Project maintainers review and discuss proposed changes.
* Automated testing may be employed to ensure code quality.

**Merging Pull Requests**

* Approved changes are merged into the main branch.
* Contributors may need to resolve conflicts during the merge.

**Versioning and Releases**

* Use tags or branches to mark releases.
* Document changes in a release notes file.

**IV. Summary**

**Key Takeaways**

* Distributed workflows offer flexibility in collaboration.
* Contributors fork, clone, and branch to make changes.
* Pull requests facilitate collaboration and code review.
* Maintainers review, merge, and release new versions.
* Clear documentation and communication are essential for successful collaboration.

Certainly! Here are detailed PowerPoint notes on the GitHub topics:

**6. GitHub**

**I. Account Setup and Configuration**

**Creating a GitHub Account**

* Sign up on the GitHub website.
* Choose a username and configure account settings.

**Configuring Git on Local Machine**

* Set global Git configurations: git config --global user.name "Your Name" and git config --global user.email "your@email.com".
* Optionally, set other configurations like preferred text editor.

**II. Contributing to a Project**

**Forking a Repository**

* Fork a repository on GitHub to create a personal copy.
* Clone the forked repository to the local machine.

**Branching and Making Changes**

* Create a branch for the intended contribution: git checkout -b <branch-name>.
* Make changes, commit them, and push to the forked repository.

**Creating a Pull Request**

* Initiate a pull request on GitHub.
* Describe the changes made and request a review.

**III. Maintaining a Project**

**Reviewing Pull Requests**

* Project maintainers review proposed changes.
* Use comments and reviews to provide feedback.

**Merging Pull Requests**

* Merge approved pull requests into the main branch.
* Resolve conflicts if necessary during the merge.

**Releases and Versioning**

* Use GitHub releases to mark project milestones.
* Document changes in release notes.

**IV. Managing an Organization**

**Creating an Organization**

* Create an organization on GitHub for collaborative projects.
* Invite members to join the organization.

**Repository Permissions**

* Set permissions for organization members based on roles.
* Control access to repositories.

**V. Scripting GitHub**

**GitHub Actions**

* Automate workflows with GitHub Actions.
* Define custom CI/CD processes in a YAML file.

**GitHub API**

* Use the GitHub API for programmatic interaction.
* Automate repetitive tasks or gather repository information.

**VI. Summary**

**Key Takeaways**

* GitHub is a platform for version control and collaborative software development.
* Forking and branching are essential for contributing to projects.
* Pull requests facilitate code review and collaboration.
* Maintainers review, merge, and release changes.
* Organizations help manage collaborative projects.
* GitHub provides automation through GitHub Actions and an API.

**7. Git Tools**

**I. Revision Selection**

**Git Log**

* git log: Displays commit history.
* Options like --oneline, --graph, and --since provide different views.

**Git Show**

* git show <commit>: Displays details of a specific commit.
* Useful for reviewing changes introduced in a commit.

**II. Interactive Staging**

**Git Add -p**

* git add -p: Interactively stage changes.
* Allows selecting specific changes within a file.

**III. Stashing and Cleaning**

**Git Stash**

* git stash: Temporarily saves changes.
* Useful for switching branches without committing.

**Git Clean**

* git clean: Removes untracked files from the working directory.
* Use with caution to avoid unintentional data loss.

**IV. Signing Your Work**

**Git Commit -S**

* git commit -S: Sign commits using GPG.
* Adds a cryptographic signature to each commit.

**V. Searching**

**Git Grep**

* git grep <pattern>: Searches for a pattern in the repository.
* Options like --line-number and --count provide additional information.

**VI. Rewriting History**

**Git Rebase**

* git rebase <branch>: Rewrites commit history.
* Squash, edit, or reorder commits interactively.

**VII. Reset Demystified**

**Git Reset**

* git reset <commit>: Unstages changes or moves the branch pointer.
* Options like --soft, --mixed, and --hard control the reset behavior.

**VIII. Advanced Merging**

**Git Merge Strategies**

* git merge -s <strategy>: Utilize different merge strategies.
* Strategies include recursive, octopus, and ours.

**IX. Rerere**

**Git Rerere**

* git rerere: Reuse recorded resolution of conflicted merges.
* Automates conflict resolution based on past resolutions.

**X. Debugging with Git**

**Git Bisect**

* git bisect: Binary search for bugs.
* Identify the commit that introduced a bug.

**XI. Submodules**

**Git Submodules**

* git submodule: Embed external repositories within a repository.
* Useful for managing dependencies.

**XII. Bundling**

**Git Bundle**

* git bundle: Package a repository as a single file.
* Transfer Git data without direct network access.

**XIII. Replace**

**Git Replace**

* git replace: Replace an object with another one.
* Useful for correcting mistakes in the commit history.

**XIV. Credential Storage**

**Git Credential Storage**

* Configure credential storage for authentication.
* Options include caching, credential managers, and helpers.

**XV. Summary**

**Key Takeaways**

* Revision selection tools like git log and git show provide insights into commit history.
* Interactive staging (git add -p) allows for granular control over changes.
* Stashing and cleaning tools help manage work in progress.
* Signing commits adds a layer of authentication using GPG.
* Searching with git grep enables pattern matching.
* Rewriting history with git rebase allows for a cleaner commit history.
* Advanced merging strategies and rerere automate conflict resolution.
* Debugging tools like git bisect help identify the source of bugs.
* Submodules manage dependencies, and bundling facilitates data transfer.
* git replace corrects mistakes, and credential storage enhances authentication.

**8. Customizing Git**

**I. Git Configuration**

**Global Configuration**

* git config --global: Set global configurations.
* Includes user details, default text editor, and color settings.

**Repository Configuration**

* git config: Set configurations specific to a repository.
* Includes repository-specific user details and custom aliases.

**II. Git Attributes**

**Git Attributes Overview**

* Define attributes for files in a repository.
* Control line endings, language, and merge strategies.

**.gitattributes File**

* Specify attributes in a .gitattributes file.
* Applied to files based on patterns and rules.

**III. Git Hooks**

**Git Hooks Introduction**

* Scripts triggered by specific Git events.
* Located in the .git/hooks directory.

**Common Git Hooks**

* pre-commit: Run before each commit.
* post-commit: Run after a commit is made.

**IV. An Example Git-Enforced Policy**

**Commit Message Policy**

* Enforce a commit message format.
* Use a pre-commit hook to check and enforce the policy.

**Implementation Steps**

1. Create a pre-commit hook script.
2. Define the policy checks in the script.
3. Configure the repository to use the hook.

**V. Summary**

**Key Takeaways**

* Git configuration is customizable globally and per repository.
* Git attributes allow fine-tuning file behaviors.
* Git hooks enable automation of custom scripts at specific events.
* Examples like commit message policies can be enforced using Git hooks.
* Customization enhances consistency and workflow efficiency in Git.

# Day3

**Understanding Git Basics**

**Introduction**

* **Version Control**: System for tracking changes in source code over time.
* **Git**: Distributed version control system.
* **Core Concepts**:
  + **Repository**: Storage for project files and version history.
  + **Commit**: Snapshot of changes at a specific point.
  + **Branch**: Independent line of development.

**Cloning a Repository and Performing Basic Operations**

**Cloning**

* git clone <repository URL>: Downloads a repository to the local machine.

**Basic Operations**

* git init: Initializes a new Git repository.
* git add <file>: Stages changes for commit.
* git commit -m "Commit message": Records changes in the repository.

**Fetching and Pulling Content**

**Fetch**

* git fetch: Retrieves changes from a remote repository.
  + Example: git fetch origin

**Pull**

* git pull origin <branch>: Fetches and integrates changes from a remote branch.
  + Example: git pull origin main

**Pushing Code**

* git push origin <branch>: Pushes changes to a remote repository.
  + Example: git push origin feature-branch

**Git Branching**

**Creating a Branch**

* git branch <branch-name>: Creates a new branch.
  + Example: git branch feature-branch
* git checkout <branch-name>: Switches to the new branch.
  + Example: git checkout feature-branch

**Merging**

* git merge <branch>: Integrates changes from one branch into another.
  + Example: git merge feature-branch

**Git Merging**

* git merge <branch>: Combines changes from different branches.
  + Example: git merge feature-branch

**Git Stash**

* git stash: Temporarily saves changes for later use.
  + Example: git stash

**Git Add Interactive**

* git add -i: Interactively stages changes.
  + Example: git add -i

**Git Rebase**

* git rebase <branch>: Rewrites commit history by moving or combining commits.
  + Example: git rebase main

**Working With Multiple Repositories**

* git remote add <name> <repository URL>: Links a local repository to a remote.
  + Example: git remote add origin https://github.com/user/repo.git

**Pull Requests**

* Feature in platforms like GitHub to propose and discuss changes.
  + Create a branch, make changes, and create a pull request on GitHub.

**Git Log**

* git log: Displays commit history.
* Options: --oneline, --graph, --since.
  + Example: git log --oneline

**Git Hooks**

* Custom scripts triggered by Git events.
* Examples: pre-commit, post-commit.
  + Example of a pre-commit hook:
  + #!/bin/bash
  + echo "Running pre-commit checks..."

# Add custom checks here

# Day4.

**GitHub Overview**

* Web-based platform for hosting Git repositories.
* Facilitates collaboration, issues, and pull requests.

**SSH Authentication**

* Generate SSH keys: ssh-keygen.
* Add SSH key to GitHub in the user's settings.

**GitHub Repository**

* Create a repository on GitHub.
* Clone repository locally.
  + Example: git clone https://github.com/user/repo.git

**Forking**

* Fork a repository to contribute without direct access.
  + Click "Fork" on GitHub.

**GitHub Repository Branches**

* Manage branches on GitHub.
* Create, delete, and merge branches.
  + Example: Creating a branch on GitHub.

**GitHub Tags and Releases**

* Tagging versions for releases.
* Create releases with changelog.
  + Example: Creating a release on GitHub.

**Understanding and Resolving Git Merge Conflicts**

* Conflicts occur when changes overlap.
* Resolve conflicts manually or using tools.
  + Example: Manual conflict resolution in a text editor.

**Implementing Git Ignore Files**

* .gitignore: Specify files or patterns to ignore.
* Ignore build artifacts, logs, etc.
  + Example: Ignoring .DS\_Store files in .gitignore.

**Git Reverting**

* git revert <commit>: Create a new commit to undo changes.
  + Example: git revert HEAD

**Git Resetting**

* git reset <commit>: Unstage or move the branch pointer.
* Options: --soft, --mixed, --hard.
  + Example: git reset --hard HEAD

**Git GUI Tools**

**Gitk**

* Basic Git GUI tool for visualization.
  + Example: gitk

**Gitkraken**

* Visual Git GUI with advanced features.
  + Example: gitkraken

**P4merge**

* Visual merge and diff tool.
  + Example: p4merge

**GitViz**

* Visual representation of Git history.
  + Example: gitviz