TABLE OF CONTENTS GIT BASIC

ADVANTAGES OF GIT VERSION CONTROL

1. Version Control and History

- **Track Changes**: Git keeps a complete history of changes, allowing you to see who made changes, what was changed, and when.
- **Revert to Previous States**: If something goes wrong, you can easily revert to a previous version of your project.
- **Branching and Merging**: Git allows you to create branches to work on new features or experiments without affecting the main codebase. You can later merge these branches back into the main branch.

2. Collaboration

- **Multiple Contributors**: Git enables multiple developers to work on the same project simultaneously without overwriting each other's work.
- **Conflict Resolution**: Git provides tools to resolve conflicts when changes from different contributors overlap.
- **Remote Repositories**: Platforms like GitHub, GitLab, and Bitbucket allow teams to collaborate on projects hosted in remote repositories.

3. Backup and Redundancy

- **Distributed System**: Every developer's working copy of the code is a full backup of the repository, reducing the risk of data loss.
- **Remote Repositories**: Remote repositories act as additional backups, ensuring that your code is safe even if local copies are lost.

4. Branching and Experimentation

- **Feature Branches**: You can create separate branches for new features, bug fixes, or experiments, keeping the main branch stable.
- **Easy Merging**: Git makes it easy to merge branches back into the main codebase once the work is complete and tested.

5. Efficiency and Performance

- **Fast Operations**: Git is designed to be fast, even for large projects with extensive histories.
- **Offline Work**: You can commit changes, create branches, and view history without an internet connection. You only need to connect to sync with remote repositories.

6. Open Source and Community Support

- **Wide Adoption**: Git is widely used in the software development community, meaning there is extensive documentation, tutorials, and community support available.
- **Integration**: Git integrates with many development tools, including IDEs, continuous integration/continuous deployment (CI/CD) pipelines, and project management tools.

7. Flexibility and Workflow Customization

- **Custom Workflows**: Git supports various workflows (e.g., Git Flow, GitHub Flow) that can be tailored to your team's needs.
- **Hooks and Automation**: Git allows you to set up hooks to automate tasks like running tests or deploying code when certain actions are performed.

8. Auditability and Accountability

- **Detailed Logs**: Git provides detailed logs of who made changes and when, which is useful for auditing and accountability.
- **Code Reviews**: Platforms like GitHub and GitLab facilitate code reviews, ensuring that changes are reviewed and approved before being merged.

9. Scalability

- **Handles Large Projects**: Git is designed to handle projects of all sizes, from small personal projects to large enterprise-level applications.
- **Efficient Storage**: Git uses compression and delta encoding to store changes efficiently, reducing the storage footprint.

10. Cross-Platform Compatibility

- **Works on Multiple OS**: Git is compatible with Windows, macOS, and Linux, making it versatile for teams using different operating systems.

11. Open Source

- **Free to Use**: Git is open-source software, meaning it's free to use and can be modified to suit specific needs.

12. Community and Ecosystem

- **Rich Ecosystem**: Git has a rich ecosystem of tools and extensions that enhance its functionality, such as GUI clients, CI/CD integrations, and more.

13. Security

- **Data Integrity**: Git uses SHA-1 hashing to ensure the integrity of your data, making it difficult for changes to go unnoticed.
- **Access Control**: Remote repositories often provide access control mechanisms to restrict who can view or modify the code.

14. Documentation and Metadata

- **Commit Messages**: Commit messages provide a way to document changes, making it easier to understand the history and rationale behind changes.
- **Tags and Releases**: Git allows you to tag specific points in history as important (e.g., releases), making it easy to reference and deploy specific versions.

Conclusion

Git is a powerful and versatile tool that enhances collaboration, efficiency, and reliability in software development. Its distributed nature, robust branching model, and extensive ecosystem make it an essential tool for developers and teams of all sizes.

SETTING UP GIT

Initialize a new repository

```
git init
```

Example:

```
git init my-project
# This will create a new folder called my-project and initialize git
```

Clone an existing repository

```
git clone <repository-url>
```

Example:

```
git clone https://github.com/user/repo.git
```

BASIC WORKFLOW

Check the status of your repository

```
git status
```

Add files to the staging area

```
git add <file-name>
```

Example:

```
git add index.html
```

To add all changes

```
git add .
```

Commit changes

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

Example:

```
git commit -m "Added index.html file"
```

BRANCHING AND MERGING

Create a new branch

```
git branch <branch-name>
```

Example:

```
git branch feature-login
```

Switch to a branch

```
git checkout <branch-name>
```

Example:

```
git checkout feature-login
```

Create and switch to a new branch

```
git checkout -b <branch-name>
```

Example:

```
git checkout -b feature-signup
```

List all branches

```
git branch
```

Merge a branch into the current branch

```
git merge <branch-name>
```

Example:

```
git merge feature-login
```

Delete a branch

```
git branch -d <branch-name>
```

Example:

```
git branch -d feature-login
```

REMOTE REPOSITORIES

Add a remote repository

```
git remote add <name> <repository-url>
```

Example:

```
git remote add origin https://github.com/user/repo.git
```

View remote repositories

```
git remote -v
```

Push changes to a remote repository

```
git push <remote-name> <branch-name>
```

Example:

```
git push origin main
```

Pull changes from a remote repository

```
git pull <remote-name> <branch-name>
```

Example:

```
git pull origin main
```

VIEWING HISTORY

View commit history

```
git log
```

View a simplified commit history

```
git log --oneline
```

View changes in a specific commit

```
git show <commit-hash>
```

Example:

```
git show abc1234
```

UNDOING CHANGES

Unstage a file:

```
git reset <file-name>
```

Example:

```
git reset index.html
```

Revert changes in a file

```
git checkout -- <file-name>
```

Example:

```
git checkout -- index.html
```

Amend the last commit

```
git commit --amend
```

Revert a commit:

```
git revert <commit-hash>
```

Example:

```
git revert abc1234
```

STASHING

Stash changes

```
git stash
```

Apply stashed changes:

```
git stash apply
```

List all stashes

```
git stash list
```

TAGS

Create a tag

```
git tag <tag-name>
```

Example:

```
git tag v1.0.0
```

Push tags to remote

```
git push --tags
```

FETCHING AND PULLING

Fetch changes from remote

```
git fetch <remote-name>
```

Example:

```
git fetch origin
```

Pull changes and merge

```
git pull <remote-name> <branch-name>
```

Example:

```
git pull origin main
```

IGNORING FILES

Create a `.gitignore` file to exclude files/folders from being tracked:

```
# Example .gitignore file
node_modules/
.env
*.log
```

SAMPLE WORKFLOW

Clone a repository

```
git clone https://github.com/user/repo.git
```

Create a new branch

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

Make changes and stage them

```
git add .
```

Commit changes

```
git commit -m "Added new feature"
```

Push changes to remote

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

Merge changes into main

```
git checkout main
git pull origin main
git merge feature-branch
git push origin main
```