Table of Content

UNIT 1: Git and GitHub

Topics Covered

1. What is version control?
2. Benefits of using Git
3. Git vs other VCS (like SVN)
4. Installing Git
5. Configuring Git (username, email)

1. What is Version Control?

Version Control System (VCS) is software that helps developers manage changes to source code over time. It allows you to:

- **Track changes** to files
- **Revert** to a previous version if something breaks
- **Collaborate** with others on the same codebase
- Avoid **conflicts** when working with teammates

Real-World Analogy:

Imagine writing a book in MS Word. You save a version every time you make edits — VCS automates this. You can go back to any version and see who made what changes.

2. Benefits of Using Git

Benefit	Description
History Tracking	Git keeps a full history of changes made to your files
Experiment Safely	You can create branches to test new features without affecting the main code
Team Collaboration	Multiple people can work on the same code without stepping on each other
Easy Rollback	Broke the app? You can go back to a stable commit
Offline Work	Git works locally; you don't need an internet connection to track changes

3. Git vs Other Version Control Systems

Feature	Git (Distributed VCS)	SVN (Centralized VCS)
Local Repository	Yes	❌ No
Offline Commits	Yes	❌ No
Speed	⚡ Very Fast	🐌 Slower
Branching	🚀 Easy and lightweight	😵 Heavy and complex
Dependency on Server	❌ No (unless pushing)	Yes (always)

Conclusion: Git is faster, more flexible, and better suited for modern development workflows.

4. Installing Git

For Windows:

1. Go to https://git-scm.com/downloads
2. Download and run the installer
3. Keep the default settings unless you know what you're doing
4. Open Git Bash (you'll use this instead of CMD)

For Ubuntu/Linux:

```
sudo apt update
sudo apt install git
```

For macOS:

```
brew install git
# OR use Xcode Command Line Tools:
xcode-select --install
```

5. Configuring Git

Once Git is installed, set your global identity (used in all repos):

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

This information will appear in your commit logs.

Verify Configuration

```
git config --list
```

You should see:

```
user.name=Your Name
user.email=your@email.com
```

Core Concepts

Distributed Version Control

Every developer has a full copy of the codebase including its entire history. This allows:

- Offline commits
- Local experimentation
- Safer collaboration

Git Architecture

Git works in **three areas**:

Area	Purpose
Working Directory	Where you write/edit files
Staging Area	Where you mark files for commit using git add
Local Repository	Where commits are saved using git commit

Hands-On Practice

Step-by-Step:

```
# Step 1: Install Git

# Step 2: Configure Git
git config --global user.name "Your Name"
git config --global user.email "you@example.com"

# Step 3: Verify settings
git config --list
```

Exercise

Objective: Setup Git on your system

Task 1: Install Git

- Use your OS-specific method to install Git.

Task 2: Configure Your Identity

- Run:
- git config --global user.name "Your Full Name"
- git config --global user.email "youremail@example.com"

Task 3: Verify Installation

- Run:

git --version
git config --list

📌 **Checkpoint**: If you see Git version and your user info, you're good to go!

Summary

- Git is a distributed version control system for tracking changes in your code.
- It helps with team collaboration, rollback, and parallel development.
- Setup requires installing Git and configuring your username/email.

Unit 2: Git Basics

Topics Covered

1. Initializing a Git repository
2. Tracking files
3. Adding & committing changes
4. Writing good commit messages

1. Initializing a Git Repository

When you're starting a new project, you need to tell Git to track changes in that directory.

Command:

```
git init
```

This creates a **.git/ hidden folder**, which stores all metadata and version history.
Once initialized, the folder is officially a Git repository.

2. Tracking Files

When you create or modify a file, Git doesn't track it **automatically**.

You need to:

- Check the file's status
- Add it to the staging area

Check File Status:

```
git status
```

You'll see:

- **Untracked files** → Not being tracked yet
- **Modified files** → Changed after last commit
- **Staged files** → Ready to be committed

3. Adding Files to Staging Area

Git uses a two-step commit process:

1. Stage the files
2. Commit them

Add a single file:

```
git add filename.txt
```

Add all files in the directory:

```
git add .
```

This moves files to the **Staging Area** (like a "preview basket" before committing).

4. Committing Changes

Once files are staged, you **commit** them — this means saving a snapshot to the repository.

Commit:

```
git commit -m "Short but clear message"
```

Git creates a commit object with:

- Author info
- Commit message
- Timestamp
- Snapshot of all staged files

Git best practice: **Commit often, in small logical chunks.**

5. Best Practices for Commit Messages

Writing clear commit messages is crucial for team collaboration and debugging later.

Golden Rules:

- Use the **imperative** mood: “Add login button”, not “Added” or “Adding”
- Keep it short and meaningful (ideally < 50 characters)
- Describe **why** something changed (not just what)

Examples:

- Add user registration form
- Fix typo in README
- ❌ Changed something
- ❌ Final commit lol

Hands-On Exercise

Objective: Create a simple project folder, track a file, and make your first commit.

Step-by-step:

```
# 1. Create a folder and navigate to it
mkdir my-first-repo
cd my-first-repo

# 2. Initialize Git
git init

# 3. Create a file
echo "Hello Git!" > hello.txt

# 4. Check the status
git status

# 5. Stage the file
git add hello.txt

# 6. Commit it
git commit -m "Add hello.txt with welcome message"
```

Congrats! You've made your first Git commit.

Troubleshooting Tips

Problem	Solution
fatal: not a git repository	Run git init first
nothing to commit	Add files using git add
File not showing in commit	Ensure it's staged with git status

Summary

- git init initializes a Git repository.
- Use git add to stage changes and git commit to save them.
- Use git status often to see what's staged/untracked.
- Write clear, short, and action-based commit messages.

Challenge Yourself

Try doing this for 3 files, then update one and commit only that specific change. Explore how Git tracks each file.

Unit 3: Working with Remote Repositories

Topics Covered

1. Setting up a GitHub account
2. Creating a GitHub repository
3. Connecting local repo to remote (GitHub)
4. Push/pull workflow
5. Cloning repositories

1. GitHub Account Setup

Steps:

1. Visit ☞ https://github.com/
2. Sign up with an email, username, and password
3. (Optional) Enable 2FA (two-factor authentication) for security
4. Personalize your profile (name, bio, profile pic)

You'll need a GitHub account to **host remote repositories** and collaborate with others.

2. Create a Repository on GitHub

Steps:

1. After logging in, click the "+" icon at the top-right → **New repository**
2. Enter a **repository name**
3. Choose **Public** or **Private**
4. *Do NOT* initialize with README, .gitignore, or license (for now)
5. Click **Create repository**

💡 GitHub now shows you instructions to link a local project — follow them or continue with the guide below.

3. Connect Local Repo to Remote

Assume you've already created a local repo using:

```
git init
```

Step 1: Add the remote origin

```
git remote add origin https://github.com/your-username/your-repo-name.git
```

💡 This tells Git *where* to push and pull from.

Step 2: Push local code to GitHub

```
git push -u origin main
```

-u sets origin as the default remote for main branch, so next time you can just do git push.

4. Push/Pull Workflow

Action	Command	Description
Push changes	git push origin main	Uploads local commits to GitHub
Pull changes	git pull origin main	Downloads latest changes from GitHub
Check remotes	git remote -v	Displays current remote URL(s)
Rename branch	git branch -M main	Rename current branch (e.g., from master)

🔄 **Pull before you push** if you're working on a team — this prevents conflicts.

5. Cloning a Remote Repo

If someone already created a GitHub repo and you want to work on it:

```
git clone https://github.com/username/repo-name.git
```

This:

- Downloads all the files
- Sets up the .git folder
- Connects to the remote origin

Now you can cd repo-name and start working immediately.

Hands-On Exercise

🎯 **Objective**: Create a GitHub repository and connect it to a local project

Step-by-Step Instructions:

```
# 1. Create a local folder
mkdir github-demo
cd github-demo
git init

# 2. Create a file
echo "# My GitHub Demo" > README.md
git add README.md
git commit -m "Initial commit"

# 3. Create a remote repo on GitHub (via web browser)

# 4. Add remote origin
git remote add origin https://github.com/YOUR_USERNAME/github-demo.git
```

```
# 5. Push to GitHub
git branch -M main  # Optional: rename default branch to 'main'
git push -u origin main
```

From another device:

```
git clone https://github.com/YOUR_USERNAME/github-demo.git
```

Now you can cd github-demo and start working from there.

Tips

- You can have multiple remotes (e.g., origin, upstream)
- Use SSH URLs for better authentication (git@github.com:user/repo.git)
- Always pull before pushing on team projects to prevent merge issues

Summary

Concept	Command
Add remote	git remote add origin <repo-url>
Push code	git push -u origin main
Pull latest changes	git pull origin main
Clone repo	git clone <repo-url>
Check remote URL	git remote -v

Unit 4: Branching Basics

1. Why Use Branches?

In Git, a **branch** is a lightweight movable pointer to a commit.
It allows developers to:

- Work on **new features** or **bug fixes** without affecting the main code
- Try experimental ideas safely
- Collaborate without interfering with others' work

Example Use Cases:

```
main → Production-ready code
feature/login → New login system
bugfix/navbar → Fixing broken navbar
```

Think of branches as separate "sandboxes" — you can experiment freely without breaking the original code.

2. Creating and Switching Branches

Create a new branch:

```
git branch new-feature
```

This creates a branch **but doesn't switch** to it yet.

Switch to a branch:

```
git switch new-feature
```

OR

```
git checkout new-feature   # Older syntax, still works
```

💡 When switching, Git updates your working directory with that branch's latest state.

3. List All Branches

To see which branches exist:

```
git branch
```

- The current active branch will have a * next to it.
- Use descriptive names like:
  - feature/signup-page
  - hotfix/logout-crash
  - bugfix/registration-username-validation

- experiment/image-compression

4. Hands-On Exercise

Objective: Create a new branch, make changes, and switch between branches.

Step-by-step:

```
1. Create a working repo
mkdir branch-demo
cd branch-demo
git init
echo "This is main branch" > index.html
git add .
git commit -m "Add index.html on main"

2. Create a new branch
git branch new-feature

3. Switch to the new branch
git switch new-feature

4. Modify the file in this branch
echo "This is new-feature branch" >> index.html
git add index.html
git commit -m "Update index.html in new-feature branch"

5. Switch back to main branch
git switch main

6. View the difference in content
cat index.html   # Should show only main branch content
```

You've now experienced **independent development** across branches!

5. Common Git Branch Names & Use Cases:

Branch Name	Purpose / Use Case
main or master	**Production-ready code** – always stable and deployable
dev or develop	**Ongoing development** – base for new features
feature/<name>	New features – isolated from other codeExample: feature/login-page
bugfix/<name>	Fixing bugs in dev or productionExample: bugfix/missing-cart-icon
hotfix/<name>	Urgent fixes for production bugsExample: hotfix/login-crash
test/<name>	Testing new experiments or code spikesExample: test/ui-animation
release/<version>	Prepares code for release, testing & packagingExample: release/v1.2.0

Typical Branching Workflow:

```
main
  ├── dev
  │     ├── feature/login
  │     ├── feature/cart
  │     └── bugfix/cart-button
  └── hotfix/payment-error
```

When to Use Each:

Scenario	Create...
Building a new module	feature/module-name
Fixing a reported issue	bugfix/issue-name
Quick production patch	hotfix/fix-name
Preparing a version release	release/vX.Y.Z
Deploying to live site	Merge into main
Collaborating with teammates	Use shared dev branch

Best Practices:

- Keep main always clean & deployable.
- Use clear, descriptive branch names.
- Delete branches after merging to avoid clutter.

Example: Branching in a To-Do App Project

Assume your **main** branch contains the initial version of a simple To-Do app with basic features like:

- Add a task
- Delete a task
- Mark task as done

Now, let's say you're developing more features and fixing issues. Here's how you would structure your branches:

1. main

Production-ready To-Do app with basic task management
Keep this branch clean, tested, and always deployable.

2. dev

Main development branch — integrates features before merging into main.

Feature Branches

Branch Name	Purpose
feature/edit-task	Add functionality to edit a task
feature/due-date	Add a due date to each task
feature/filter-completed	Allow filtering tasks (e.g., completed/incomplete)
feature/user-auth	Add login/signup system
feature/share-todo	Allow users to share their to-do list with others

Bugfix Branches

Branch Name	Purpose
bugfix/duplicate-task	Fix bug where adding a task twice causes a crash
bugfix/deletion-delay	Fix slow response when deleting tasks

Hotfix Branches (urgent production issues)

Branch Name	Purpose
hotfix/task-not-saving	Fix broken task saving feature on deployed app
hotfix/page-crash	Emergency fix for app crashing on homepage

Release Branches

Branch Name	Purpose
release/v1.0.0	Prepares first full-featured release for deployment
release/v1.1.0	Includes due dates and edit task feature

Test/Experiment Branches

Branch Name	Purpose
test/ui-redesign	Try out a new UI layout for the task list
test/drag-drop	Experiment with drag-and-drop for reordering tasks

Sample Workflow:

1. Create new feature:

```
git checkout -b feature/edit-task
```

2. Do the work → commit → push
3. Merge into dev for testing
4. After all features are ready, merge dev into main for production

6. Summary

Task	Command
Create a branch	git branch new-branch-name
Switch to a branch	git switch new-branch-name
List all branches	git branch
Rename current branch	git branch -m new-name

Unit 5: Branch Management & Merging

1. What is Merging?

Merging combines changes from one branch into another.

Common use case:

Merge feature-branch into main after completing a feature.

Command:

```
git merge feature-branch
```

This command means: “Take the changes from feature-branch and apply them on top of the current branch.”

2. Merge Conflicts

Conflicts happen when:

- The same line of a file was changed differently in two branches
- Git can’t decide which version to keep

What it looks like in a file:

```
<<<<<<< HEAD
This is the main branch version
=======
This is the feature branch version
>>>>>>> feature-branch
```

You must **manually edit** this to decide what stays.

Resolving a conflict:

1. Open the file and delete the conflict markers (<<<<<<<, =======, >>>>>>>)
2. Keep the correct content
3. Stage the resolved file:

```
git add .
```

4. Finalize with:

```
git commit
```

3. Fast-forward vs No-fast-forward Merge

Fast-forward:

If main hasn't moved ahead, Git can just “move the pointer”:

main → same as feature-branch

Looks like:

A → B → C ← main, feature-branch

Git applies no new merge commit — it just moves main forward.

No-fast-forward (merge commit):

When main and feature-branch have diverged:

```
      → C (main)
     /
A → B
     \
      → D (feature)
```

Git **creates a new merge commit** with both histories:

```
git merge feature-branch
```

You’ll get a commit like:

Merge branch 'feature-branch' into main

Hands-On Exercise

🎯 **Goal**: Create two branches, introduce a conflict, resolve it, and merge into main.

🛠 Step-by-step:

```
# 1. Set up a repo and commit a file
mkdir merge-demo && cd merge-demo
git init
echo "Hello from main" > file.txt
git add . && git commit -m "Initial commit on main"

# 2. Create and switch to feature branch
git checkout -b feature
echo "Hello from feature branch" > file.txt
git add . && git commit -m "Edit in feature branch"

# 3. Switch back to main and edit same line
git switch main
echo "Hello from main branch updated" > file.txt
git add . && git commit -m "Edit in main branch"

# 4. Try merging (will cause a conflict)
git merge feature
```

Now you'll see a **merge conflict** in file.txt.

Resolve the conflict:

Open file.txt, you'll see:

```
<<<<<<< HEAD
Hello from main branch updated
=======
Hello from feature branch
>>>>>>> feature
```

📝 Edit to something like:

Hello merged version!

Then finalize:

```
git add file.txt
git commit -m "Resolve conflict and merge feature into main"
```

Done! You've completed a manual merge with conflict resolution.

Best Practices

Tip	Why it matters
Commit often	Smaller changes = fewer merge headaches
Pull before merge	Keeps your branch up-to-date
Use clear messages	Understand what the merge was for later

Summary

Task	Command
Merge branches	git merge branch-name
Resolve conflict	Edit file manually, then commit
List branches	git branch
See history	git log --oneline --graph

Extra Tip

To avoid merge conflicts on teams:

- Communicate which files you're editing
- Keep branches short-lived and frequently merged

Unit 6: Advanced Branching & Collaboration

1. Rebase Basics

What is Rebase?

Rebasing takes your feature branch and **replays your commits on top of another branch**, like main.

Use Case: You want your feature branch to stay up-to-date and have a clean linear history.

Rebase vs Merge

Operation	What it does	History
Merge	Combines histories, creates merge commits	Graph tree
Rebase	Rewrites history, avoids merge commits	Linear

Rebase Syntax:

```
git switch feature
git rebase main
```

This applies your feature branch commits **after the latest commit in main**.

If conflicts occur:

1. Git will pause on conflict
2. Manually resolve the conflict in files
3. Run:
4. git add .
5. git rebase --continue

Exercise 1: Rebase a Feature Branch

```
# On main branch
echo "main version" > app.js
git add . && git commit -m "Main edit"

# On feature branch (after some initial commit)
git switch feature
echo "feature version" > app.js
git add . && git commit -m "Feature edit"

# Now try rebasing
git rebase main
# If conflict occurs: resolve, then
git add app.js
git rebase --continue
```

2. Pull Requests (PRs)

What is a PR?

A **Pull Request** (on GitHub) is a way to:

- Review code
- Discuss changes
- Approve and merge into main

Steps to create a PR:

1. Push your feature branch to GitHub:
2. git push origin feature
3. On GitHub:
    - Click **"Compare & pull request"**
    - Add title and description
    - Submit PR to main branch
4. Team members can:
    - Review
    - Comment
    - Request changes
    - Approve & merge

PRs = safer & cleaner collaboration!

3. GitFlow & Feature Branch Strategies

These are **branching models** for teams:

Git Flow:

- main → Production
- develop → Active development
- feature/xyz → Individual features
- release/1.0, hotfix/urgent-fix branches when needed

Feature Branch Strategy (simpler for beginners):

- Always work on new branches:
    - feature/navbar-redesign
    - bugfix/login-error
    - refactor/db-models

Merge into main only after review/approval.

4. .gitignore Essentials

Use .gitignore to **skip unnecessary or sensitive files** from being tracked by Git.

Common use cases:

```
.env (API keys, secrets)
node_modules/
dist/, build/ folders
```

Add .gitignore file:

```
touch .gitignore
```

Example .gitignore content:

```
node_modules/
.env
secret.env
.DS_Store
```

Track it:

```
git add .gitignore
git commit -m "Add .gitignore"
```

Exercise 2: Create and Use .gitignore

```
# Create sensitive or temp file
touch secret.env
echo "API_KEY=12345" > secret.env

# Add to .gitignore
echo "secret.env" >> .gitignore

# Try adding files
git add .
git status   # secret.env will be ignored
```

Exercise 3: Create a Pull Request

1. Create a GitHub repo
2. Clone it locally and create a branch:
3. git checkout -b feature/readme
4. echo "# GitHub Demo" > README.md
5. git add . && git commit -m "Add README"
6. git push origin feature/readme
7. Go to GitHub → Open a PR from feature/readme to main

Best Practices

Tip	Reason
Rebase before PR	Cleaner history
Use .gitignore	Prevent leaking sensitive files
Create PRs, not direct merges	Enables review & feedback
One feature per branch	Easier to test and revert if needed

Summary

Task	Command / UI Action
Rebase a branch	git rebase main
Continue after resolving	git rebase --continue
Create .gitignore	touch .gitignore + list unwanted files
Push feature branch	git push origin feature-name
Create PR	GitHub UI: Compare & pull request

MCQs

Unit 1: Introduction to Git

1. **What is Git?**
   A. Programming language
   B. Centralized version control system
   C. Distributed version control system
   D. Text editor
2. **Which of the following is a primary benefit of version control?**
   A. Automatic software testing
   B. Tracking code changes
   C. Compiling code
   D. Hosting websites
3. **Which command is used to configure a Git user's email globally?**
   A. git user.email
   B. git config email
   C. git config --global user.email
   D. git set email
4. **Which of these is *not* a Git area?**
   A. Working Directory
   B. Staging Area
   C. Remote Server
   D. Build Cache
5. **Git is classified as which type of version control system?**
   A. Centralized
   B. Distributed
   C. Local-only
   D. Hybrid

Unit 2: Git Basics

6. **Which command initializes a new Git repository?**
   A. git start
   B. git init
   C. git new
   D. git create
7. **What does git status show?**
   A. Current GitHub issues
   B. Status of remote repositories
   C. Modified and staged files
   D. Running processes
8. **Which command stages all files in the directory?**
   A. git add -all
   B. git add /

C. git add .
D. git push .

9. **Which of these is a good commit message?**
   A. "done"
   B. "Fix: correct login validation logic"
   C. "update"
   D. "stuff added"
10. **What is the default branch name in Git (after 2020)?**
    A. master
    B. head
    C. main
    D. origin

Unit 3: Remote Repos

11. **Which command connects your local repo to a GitHub repo?**
    A. git connect
    B. git remote add origin <url>
    C. git link
    D. git push origin
12. **What does `git clone` do?**
    A. Deletes the repo
    B. Creates a backup
    C. Copies a remote repo to local
    D. Forks the repo
13. **What does `git push` do?**
    A. Pulls from remote
    B. Sends commits to remote
    C. Stages changes
    D. Creates a PR
14. **To pull changes from GitHub, which command is used?**
    A. git send
    B. git update
    C. git pull origin main
    D. git upload
15. **What is required to push to a GitHub repo?**
    A. Docker
    B. Python
    C. Remote URL and access
    D. Web server

Unit 4: Branching Basics

16. **What is the purpose of branching in Git?**
    A. To store credentials
    B. To experiment without affecting main
    C. To download updates
    D. To access GitHub

17. **Which command creates a new branch named `dev`?**
    A. `git add dev`
    B. `git new branch dev`
    C. `git branch dev`
    D. `git create dev`
18. **Which command switches to a branch called `login-feature`?**
    A. `git change login-feature`
    B. `git move login-feature`
    C. `git switch login-feature`
    D. `git jump login-feature`
19. **Which command lists all branches?**
    A. `git branches`
    B. `git list-branch`
    C. `git show-branches`
    D. `git branch`
20. **What happens if you commit on a new branch?**
    A. It updates the main branch
    B. It only affects that branch
    C. It syncs with GitHub
    D. It discards previous commits

Unit 5: Merge & Conflicts

21. **What does `git merge` do?**
    A. Deletes a branch
    B. Applies changes from one branch to another
    C. Conflicts branches
    D. Creates a PR
22. **Merge conflicts occur when:**
    A. Same files exist in two branches
    B. You clone a repo
    C. You use `.gitignore`
    D. You rename a branch
23. **Which marker is shown in a conflict file?**
    A. `+++ CONFLICT`
    B. `====>`
    C. `<<<<<<< HEAD`
    D. `!conflict`
24. **After resolving a conflict, you must:**
    A. Commit the resolution
    B. Run `git merge --abort`
    C. Delete the branch
    D. Clone again
25. **Fast-forward merge means:**
    A. Merging two unrelated histories
    B. Git creates a new merge commit
    C. Git moves branch pointer ahead
    D. Git discards changes

Unit 6: Advanced Topics

26. **What does `git rebase main` do (on a feature branch)?**
    A. Deletes main
    B. Creates a PR
    C. Reapplies feature commits on top of main
    D. Pushes to main
27. **What is the purpose of `.gitignore`?**
    A. Hide folders from GitHub
    B. Avoid tracking unwanted files
    C. Delete local files
    D. Encrypt credentials
28. **Which file should be in `.gitignore`?**
    A. index.html
    B. main.js
    C. .env
    D. README.md
29. **Pull Request (PR) is used to:**
    A. Push directly to main
    B. Start a Git repo
    C. Propose changes and get them reviewed
    D. Track issues
30. **Which is a good Git strategy for team collaboration?**
    A. Everyone commits on main
    B. Rebase to hide mistakes
    C. Use feature branches and PRs
    D. Push untested code quickly

MCQ Answer Key (1–30)

Q1. C	Q2. B	Q3. C	Q4. D
Q5. B	Q6. B	Q7. C	Q8. C
Q9. B	Q10. C	Q11. B	Q12. C
Q13. B	Q14. C	Q15. C	Q16. B
Q17. C	Q18. C	Q19. D	Q20. B
Q21. B	Q22. A	Q23. C	Q24. A
Q25. C	Q26. C	Q27. B	Q28. C
Q29. C	Q30. C		