



Sri Lanka Institute of Information Technology

# APPLICATION FRAMEWORKS

## VERSION CONTROLLING

### LECTURE 02

Faculty of Computing

Department of Software Engineering

Module Code: SE3040

# VERSION CONTROLLING

- What and why?
- Terminology
- Best practices
- GIT vs GITHUB



# WHAT?

- Version control is the practice of tracking and managing software code changes.

like a label (e.g., "Revision 42") so you can easily refer to a specific version of the code.

- Changes are identified using a *revision number*.

Each revision comes with extra info—namely, a timestamp (when the change happened) and the name of the person who made it

- Each revision has its *timestamp* as well as *the person who done the change*.

roll back to an earlier version if something goes wrong

combine changes from different versions

- Revisions can be restored, compared and merged.

- “Management of multiple revisions of the same unit of information”

version control is all about organizing and handling different versions of the same piece of code (or data) efficiently.

# WHY?

- **Centralized source code repository.** all the code is stored in one central place,
- **Easier backups**
- **Easy collaborative development.** a centralized repository lets everyone access and contribute to the same codebase
- **Overview of changes performed to a file.** keeps a full history of every change made to each file (using revision numbers, timestamps, and author details).
- **Access control.** lets you decide who can view, edit, or manage the code. You can set permissions
- **Helps in conflict resolution.** When multiple people edit the same code, conflicts can happen. The repository tracks these changes and provides tools to compare, merge, or resolve conflicts, ensuring nothing breaks.

# BENEFITS OF VERSION CONTROL

- **Security:** Protects against accidental loss or corruption.
- **Clean History:** Keeps track of changes and authors.
- **Collaboration:** Enables teams to work on the same project simultaneously.
- **Branching & Merging:** Allows feature development without affecting the main code.
- **Scalability:** Works efficiently with small and large projects.

# VERSION CONTROL TERMINOLOGY

- **Repository** – The central location where project files are stored.

older systems like SVN

modern ones like Git

- **Trunk (Master/Main Branch)** – The most stable version of the project.

- **Stage** – The process of marking files for tracking changes.

you select which files or changes you want the version control system to track. In Git, for example, you "stage" changes (using git add)

- **Commit** – A snapshot of changes in the repository.

permanent record of what you've done at that moment,

- **Merge** – Combining branches together.

combining changes from different versions (or branches) of code into one. if you worked on a new feature separately, merging brings it back into the main project.

- **Branch** – A copy of the code used for feature development or bug fixes.

separate copy of the code that lets you work on something (like a new feature or bug fix) without messing with the main version

- **Merge Conflict** – Occurs when multiple changes conflict and need manual

**resolution.**

when changes from different branches clash—like if two people edit the same line of code differently. The system can't decide which change to keep, so it flags a conflict, and you have to manually sort it out (e.g., pick one change or blend them).



# GIT - THE POPULAR VERSION CONTROL SYSTEM

features

**Distributed version control system.** every client (developer) gets a full clone of the repository, including all the code and its entire history

- Client get a complete clone of the source code. In a disaster situation full source along with all history can be restored from a client.

- Free and open source.

- Multiple branches and tags.

feature branches (for new features) or role branches (e.g., a "production" branch for live code).

- Feature branches, role branches (production).

- Faster comparing to other systems (works on a linux kernel and written in C).

- Support multiple protocols

Web-based access (e.g., cloning from GitHub).

- HTTP, SSH

Secure, encrypted access for pushing/pulling code.

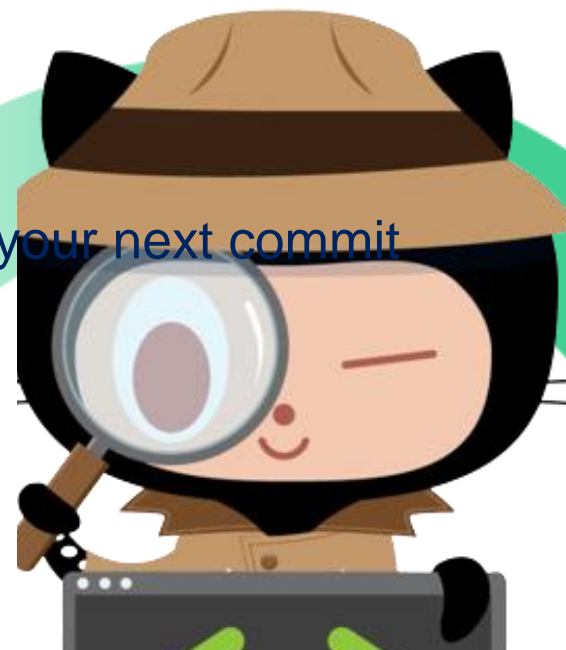
- Staging area, local commits and stashing.

- Staging area - Mark files to be committed. git add - It's like picking which changes to pack into your next commit

- Local commit - Commit code locally without pushing into the remote branch.

- Stashing - Keep file changes in Stash and apply them in a later.

temporarily saved changes you're not ready to commit



# GIT VS GITHUB

## Git

- A version control system.
- Works locally on your machine.
- Used for tracking changes and collaboration.

## GitHub

- A cloud-based hosting service for Git repositories.
- Provides UI and additional features like issue tracking, CI/CD integration.

Feature	Git	GitHub
What It Is	A distributed version control system	A cloud-based hosting service for Git repositories
Where It Works	Locally on your machine	Online (cloud-based)
Main Purpose	Tracks changes and enables collaboration	Hosts Git repos with added features
Installation	Requires local installation (e.g., via CLI)	No installation—accessed via browser or Git client
Cost	Free and open-source	Free tier available; paid plans for private repos or extras
Key Functions	Commits, branches, merges, staging	Repo hosting, UI, pull requests, collaboration tools
Speed	Fast (written in C, works locally)	Depends on internet speed for access
Protocols Supported	HTTP, SSH (for remote interaction)	HTTP, SSH (plus GitHub-specific APIs)
Collaboration	Local clones; manual sharing via remotes	Centralized hub for team access, forks, and reviews
Extra Features	Staging area, local commits, stashing	Issue tracking, CI/CD (e.g., GitHub Actions), wikis
Backup	Full history in every clone	Relies on GitHub servers (plus local clones)
Access Control	Managed locally or via remote setup	Built-in permissions (public/private repos, teams)



# GITHUB ALTERNATIVES

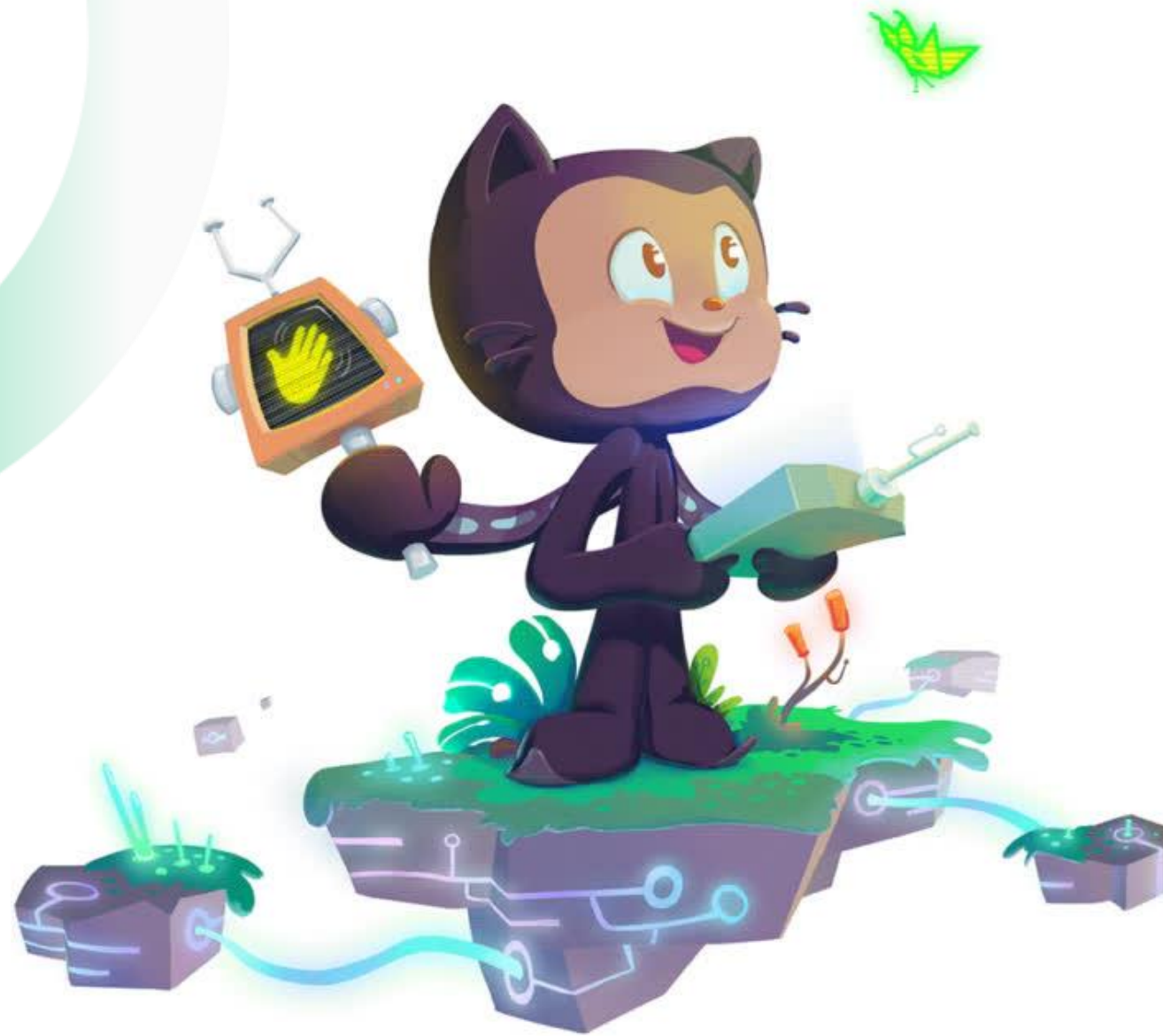
## Git Alternatives

- **Fossil** – A distributed version control system with built-in bug tracking and wiki.
  - **Mercurial** – A high-performance distributed version control system.
  - **Subversion (SVN)** – A centralized version control system used in legacy projects. this is older system
- existing project that is old

## GitHub Alternatives

- **GitLab** – Provides CI/CD tools and DevOps integration.
- **Bitbucket** – Supports Git and Mercurial, ideal for teams using Atlassian tools.
- **AWS CodeCommit** – A managed Git repository service for AWS users.
- **Azure Repos** – A cloud-based repository service for Microsoft Azure.

# GIT COMMANDS



- Git init
- Git clone
- Git add
- Git commit
- Git push

<https://confluence.atlassian.com/bitbucketserver/basic-git-commands-776639767.html>

# BASIC GIT COMMANDS

## 1. Initializing & Cloning

- `git init` – Initialize a new repository.
- `git clone <repository_url>` – Clone an existing repository.

## 2. Staging & Committing

- `git add <file>` – Stage a specific file.
- `git add .` – Stage all changes.
- `git commit -m "Commit message"` – Commit staged changes with a message.

# STAGING IN GIT

## What is Staging?

When you edit files in your project (your "working directory"), those changes don't automatically get saved to the repository's history. Instead, staging lets you pick and choose which changes you want to package up before making them official with a commit.

Staging in Git is the process of preparing your changes before committing them to the repository. It acts as a middle step between modifying files and permanently saving those changes in Git.

## Why is Staging Important?

- Allows you to review your changes before committing.
- Helps in grouping related changes into a single commit.
- Ensures that only selected files are committed. You might have 10 changed files, but only want to commit 3. Staging lets you select exactly what goes in, leaving the rest for later.

## How Staging Works in Git?

1. Modify files in your working directory.
2. Stage the files using `git add` (this moves the changes to the staging area).
3. Commit the changes to save them in the repository using `git commit`.

# STAGING IN GIT

Staging is temporary and flexible (you can add or remove things), while committing is permanent (it's locked into the repository's history).

## Staging vs. Committing

This is the prepare step.

The changes stay in a temporary holding area (the staging area) and aren't part of the repository's history until you commit.

**Staging** :- Prepares files for commit but does not save them permanently.

**Committing** :- Saves the staged changes into the repository with a commit message.

This is the save step.

When you commit, you take whatever's in the staging area and make it a permanent snapshot in the repository, complete with a message (e.g., "Fixed footer alignment").

## Git Commands for Staging

- **git add <file>** → Stage a specific file
- **git add .** → Stage all modified files
- **git status** → Check which files are staged lists files that are staged (ready to commit), modified but unstaged or untracked
- **git reset <file>** → Unstage a file Unstages a file. If you added style.css by mistake (git add style.css), running git reset style.css pulls it back out of the staging area, leaving it in your working directory.



# BASIC GIT COMMANDS

## 3. Branching & Merging

- `git branch <branch_name>` - Create a new branch.
- `git checkout <branch_name>` - Switch to a branch.
- `git merge <branch_name>` - Merge a branch into the current branch.
- `git branch -d <branch_name>` - Delete a branch (after merging)

If you're on main and run `git merge feature-login`, it brings all changes from feature-login into main.

d flag ensures it only deletes if the branch is fully merged (safe deletion).

## 4. Working with Remote Repositories

This covers how your local Git repo interacts with a remote one

"origin" as the nickname for that remote URL.

- `git remote add origin <repository_url>` - Link a local repository to a remote one.
- `git push origin <branch_name>` - Push changes to the remote repository.
- `git pull origin <branch_name>` - Pull changes from the remote repository.
- `git fetch` - Fetch updates from the remote repository.

Downloads updates from the remote repo (like new branches or commits) but **doesn't merge them**

**fetch** is cautious (**just looks**), while **pull** is proactive (**looks and merges**).

Downloads changes from the remote branch and merges them into your current local branch.



# BASIC GIT COMMANDS

## 5. Undoing Changes

Remove the commit, but don't touch the changes I made.

targets the last commit.

- **git reset --soft HEAD~1** – Undo the last commit while keeping changes. Undo the last commit but keeps your changes in the staging area (and working directory).

Delete the commit and all its changes

- **git reset --hard HEAD~1** – Undo the last commit and discard changes.

- **git checkout -- <file>** – Discard changes in a specific file.

Discards all uncommitted changes in a specific file, reverting it to the last committed version.

tells Git this is a file, not a branch

### --soft vs. --hard :

- **--soft** is gentle—it keeps your changes so you can rework them.
- **--hard** is brutal—it erases everything from that commit, no take-backs (unless tracked elsewhere).

### reset vs. checkout :

- **reset** rewinds commits (history-level changes).
- **checkout -- <file>** fixes individual files in your working directory without touching the commit history.

# GIT BRANCHING

way to create separate “paths” or copies of your codebase within the same repository

- Allows developers to work on different features

separately.

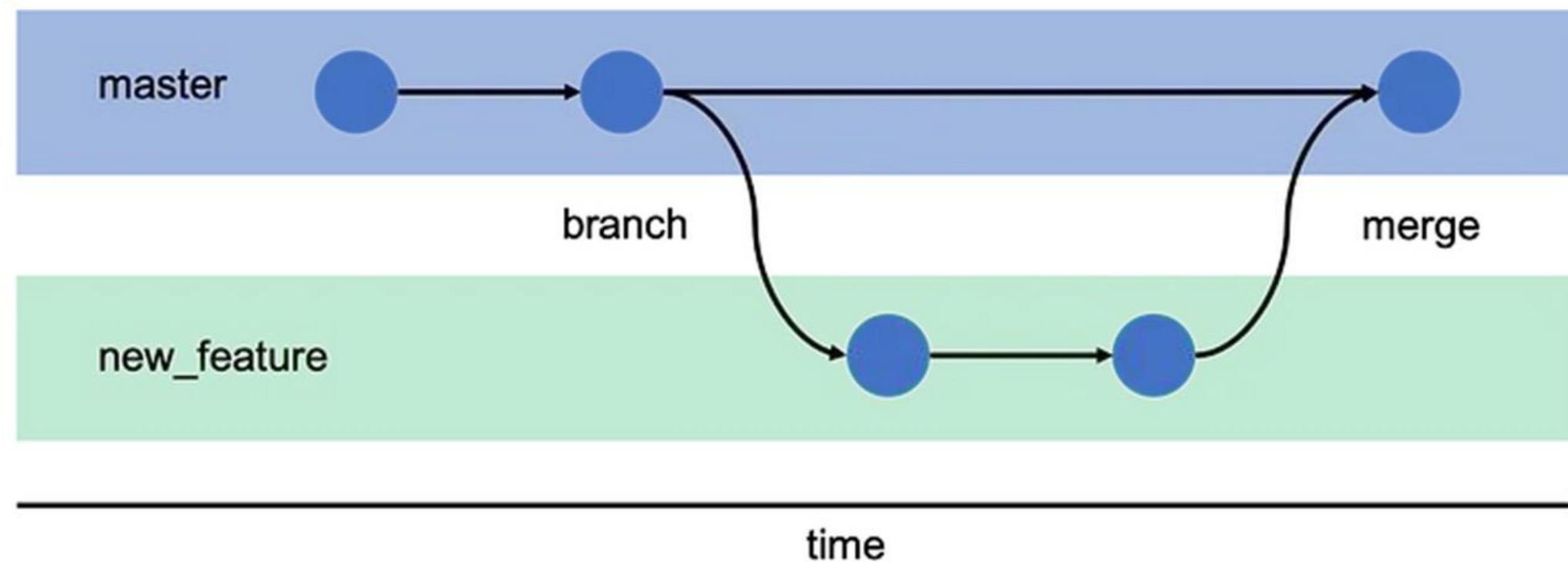
you can create a branch for a specific task (e.g., feature-login for a login page) and work on it without affecting other parts of the project.

- Helps maintain a stable main branch.

main branch (often called main or master) stays clean and functional—only tested, working code gets merged into it.

- Supports multiple branches for development.

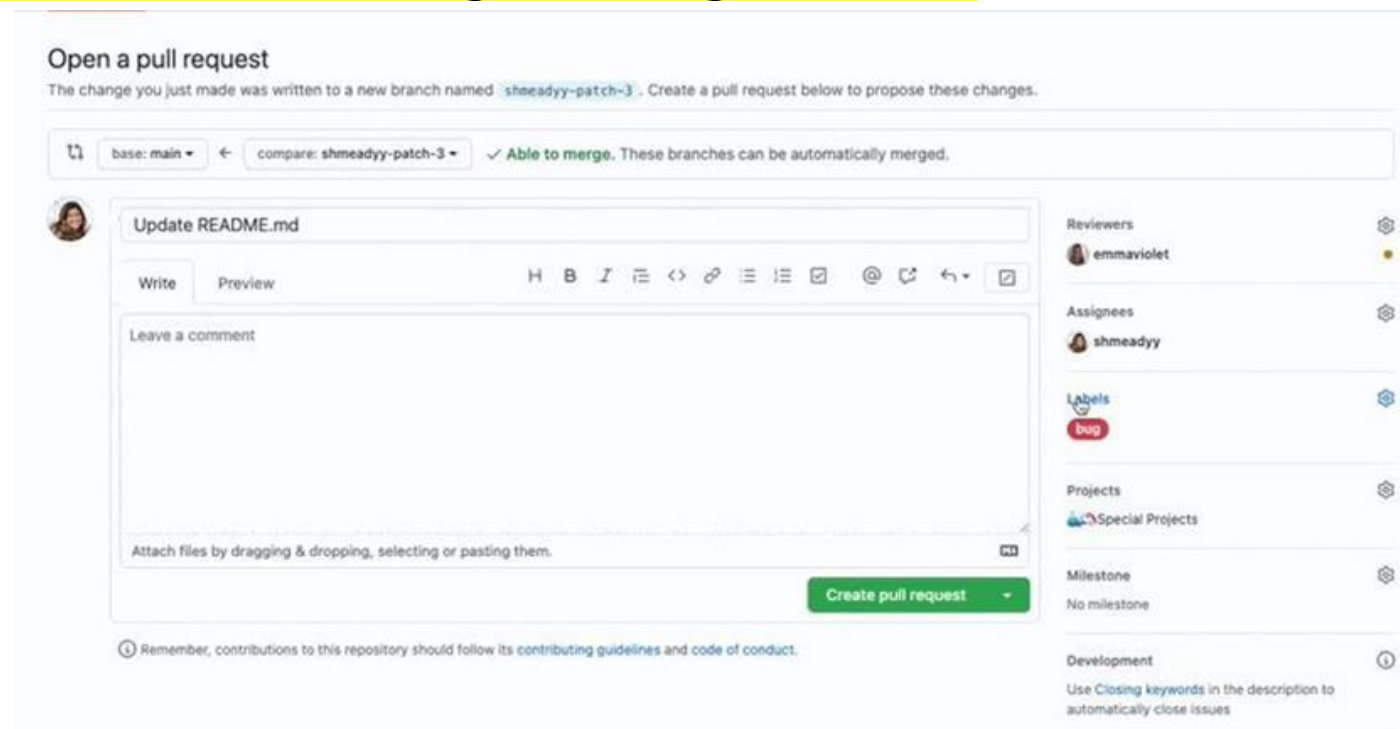
lets you create as many branches as you need—for features, bug fixes, experiments, or even roles (e.g., production, staging).



# PULL REQUESTS (PR)

After finishing work in a branch (e.g., feature-login), you push it to the remote repo (git push origin feature-login). Then, on GitHub (or similar), you create a PR to merge feature-login into main.

- A request to merge changes from one branch into another.
- Helps in the code review process.
- Ensures quality control before integrating code.



The screenshot shows the GitHub 'Open a pull request' page. At the top, it says 'Open a pull request' and 'The change you just made was written to a new branch named shmeaddy-patch-3. Create a pull request below to propose these changes.' Below this, there's a dropdown menu showing 'base: main' and 'compare: shmeaddy-patch-3', with a green checkmark and the text 'Able to merge. These branches can be automatically merged.' The main content area has a title 'Update README.md' and a 'Write' tab. Below the title is a 'Leave a comment' section and a file attachment area. On the right side, there are sections for 'Reviewers' (listing emmaviolet), 'Assignees' (listing shmeaddy), 'Labels' (with a 'bug' label), 'Projects' (with 'Special Projects'), 'Milestone' (set to 'No milestone'), and 'Development' (with a note about closing keywords). A green 'Create pull request' button is at the bottom right. A footer note says 'Remember, contributions to this repository should follow its contributing guidelines and code of conduct.'

# MERGE CONFLICTS

## What is a Merge Conflict?

Imagine two people (or you at different times) edit the same line or same section of a file in different ways across two branches. Git doesn't know which version to pick, so it stops and asks for your help.

A merge conflict occurs when Git cannot automatically merge changes because two branches have modified the same part of a file in different ways.

- ✓ If the changes do not overlap, Git automatically merges them.
- ✗ If there is an overlap, Git flags a merge conflict that must be manually resolved.

# ◆ Example of a Merge Conflict

## Scenario

- Alice and Bob are working on the same project.
- They both **modify the same line** in a file called index.html in **different branches**.

## Steps Leading to a Conflict

Alice's changes (in **feature-branch**):

**<h1>Welcome to Our Website!</h1>**

Bob's changes (in **main branch**):

**<h1>Welcome to My Website!</h1>**

Bob commits his changes to **main**.

Alice tries to merge her **feature-branch** into **main**.

git checkout main

git merge feature-branch

 **Git detects a conflict and stops the merge!**

# ◆ Example of a Merge Conflict

## ◆ Resolving the Merge Conflict

### Identify Conflicted Files

Run `git status` to see which files have conflicts.

Git marks the conflicting section in index.html:

```
<<<<<<< HEAD
<h1>Welcome to My Website!</h1>
=====
<h1>Welcome to Our Website!</h1>
>>>>>>> feature-branch
```

HEAD represents the `main branch` version.

`feature-branch` version appears below the separator

<<<<<<< HEAD to ===== is the current branch's version (`main` ).  
===== to >>>>>>> is the incoming branch's version ( `feature-branch` ).



## ◆ Example of a Merge Conflict

## ◆ Options to Resolve the Conflict

### 1. Accept Current Change (HEAD)

Keep the version from the **main branch** (Bob's version).

**<h1>Welcome to My Website!</h1>**

### 2. Accept Incoming Change (feature-branch)

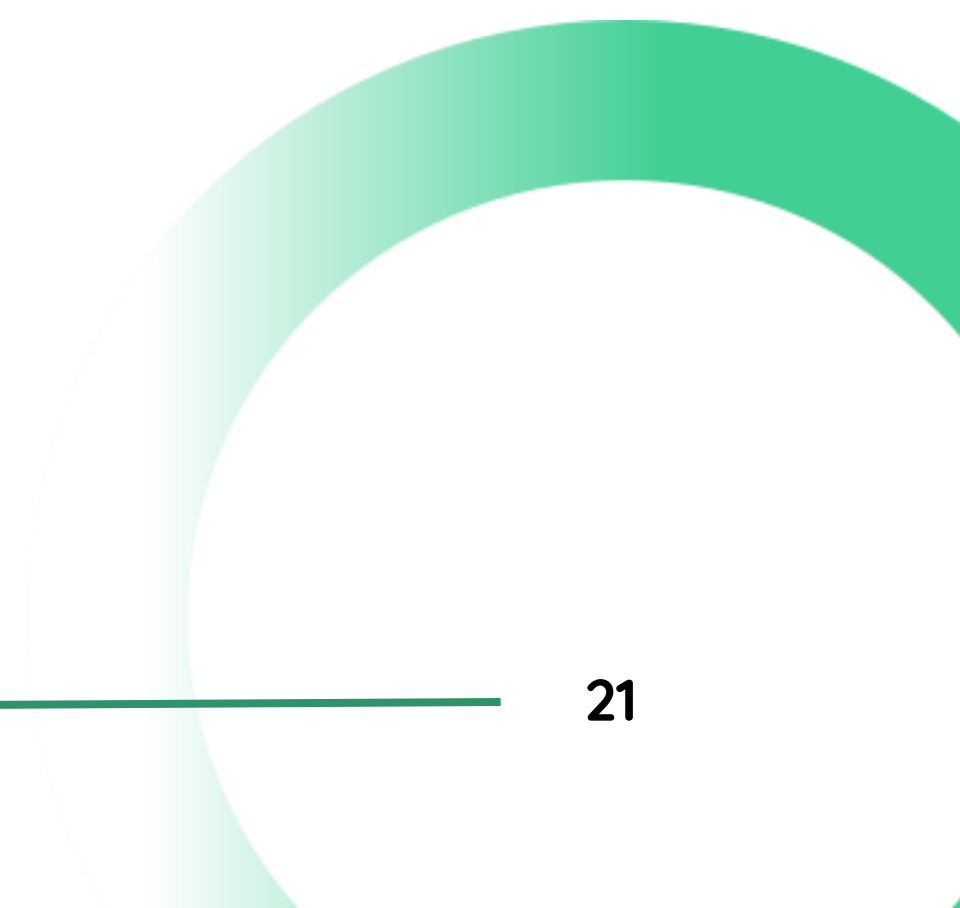
Keep the version from Alice's **feature branch**.

**<h1>Welcome to Our Website!</h1>**

### 3. Accept Both Changes (Merge Manually)

Combine both versions into a **new version**.

**<h1>Welcome to My Website! (Our Website!)</h1>**



## ◆ Example of a Merge Conflict

### ◆ Finalizing the Merge Resolution

After manually fixing the conflict in index.html:

```
git add index.html
```

```
git commit -m "Resolved merge conflict in index.html"
```

```
git push origin main
```

✓ The conflict is resolved, and changes are merged successfully.

# BEST PRACTICES

- Use meaningful commit messages.
- Do not commit sensitive data. Avoid adding passwords, API keys, or personal info to the repo.
- Use .gitignore to ignore unnecessary files. gitignore file to tell Git which files or folders to skip (e.g., temp files, build outputs).
- Always make sure to have the latest version of the file.
- Get the Latest Source Code at Least Once a Day (For Distributed Version Control Systems). In systems like Git, sync with the remote repo daily. Why: Keeps your local copy fresh, reducing merge conflicts and keeping you aligned with the team.  
Example: Start your day with git fetch and git pull.
- Merge code with the development branch at least once per day.
- Always make sure code is working as expected and it is not causing any other code to break. Test your changes before committing or merging to ensure functionality and stability.
- Follow a formal review process when merging. Use pull requests (PRs) or similar for peer review before merging into key branches.

# How to Push a Local Git Repository to GitHub (Step-by-Step)

## 1. Create a Repository on GitHub

- i. Go to GitHub → [GitHub New Repository](#)
- ii. Enter a Repository Name (e.g., MyProject).
- iii. Select Public or Private.
- iv. **DO NOT** check "Initialize this repository with a README" (to avoid conflicts).
- v. Click Create Repository.
- vi. GitHub will show a URL like ( <https://github.com/your-username/MyProject.git> )

# How to Push a Local Git Repository to GitHub (Step-by-Step)

## 2. Open Git Bash and Navigate to Your Local Project

If you already have a local Git repository, open Git Bash and go to your project folder

```
cd /path/to/your/local/repository
```

## 3. Initialize Git (If Not Already Done)

If your project is not a Git repository yet, initialize Git:

```
git init
```

# How to Push a Local Git Repository to GitHub (Step-by-Step)

## 4. Add Files to Git

Check the status of files:

**git status**

Add all files to the staging area:

**git add .**

## 5. Commit Your Code

Commit your changes with a meaningful message:

**git commit -m "Initial commit"**



# How to Push a Local Git Repository to GitHub (Step-by-Step)

## 6. Add GitHub Repository as a Remote

Replace your-username and MyProject with your GitHub details:

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

OR

```
git remote add origin https://<your-github-username>:<your-personal-access-token>@github.com/your-username/MyProject.git
```

First time setting a remote repo

```
git remote add origin <URL>
```

Changing an existing remote URL

```
git remote set-url origin <URL>
```

Updates the URL of an existing remote (like origin) to a new address.

# How to Push a Local Git Repository to GitHub (Step-by-Step)

## 7. Push Code to GitHub

Finally, push your local repository to GitHub:

`git branch -M main`

-M: A forceful rename option (short for --move --force). It moves the current branch to the new name (main) and overwrites any existing main branch if it exists.

`git push -u origin main`

Pushes the main branch to the remote repository (origin) and sets it as the upstream branch for future tracking.

-u: Short for --set-upstream. Links your local main branch to the remote main branch,

# How Clone a GitHub Repository to Your Local Machine (Step-by-Step)

## ◆ 1. Find the Repository on GitHub

Go to GitHub and open the repository you want to clone.

Click the "Code" button.

Choose the following:

HTTPS : <https://github.com/your-username/repository-name.git>

## ◆ 2. Open Git Bash or Command Line

On Windows: Open Git Bash.

On Mac/Linux: Use Terminal.

# How Clone a GitHub Repository to Your Local Machine (Step-by-Step)

## ◆ 3. Navigate to Your Desired Local Directory

Before cloning, move to the folder where you want to store the repository:

```
cd /path/to/your/folder
```

## ◆ 4. Clone the Repository

Now, use one of the following commands:

Clone using HTTPS (Easiest) (**Public Repository**)

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

# How Clone a GitHub Repository to Your Local Machine (Step-by-Step)

If you're cloning a private repository, you need authentication:

Why Authentication: Private repos restrict access to protect sensitive code, so Git needs your credentials to verify you're allowed.

✓ Use a **Personal Access Token (PAT)**

If the repository is private, GitHub requires a PAT instead of a password.

git clone https://your-username:your-personal-access-token@github.com/your-username/repository-name.git

# How Clone a GitHub Repository to Your Local Machine (Step-by-Step)

## ◆ 5. Navigate into the Cloned Repository

After cloning, move into the repository folder:

```
cd repository-name
```

## ◆ 6. Verify the Clone

To check if everything is working, run:

```
git status
```

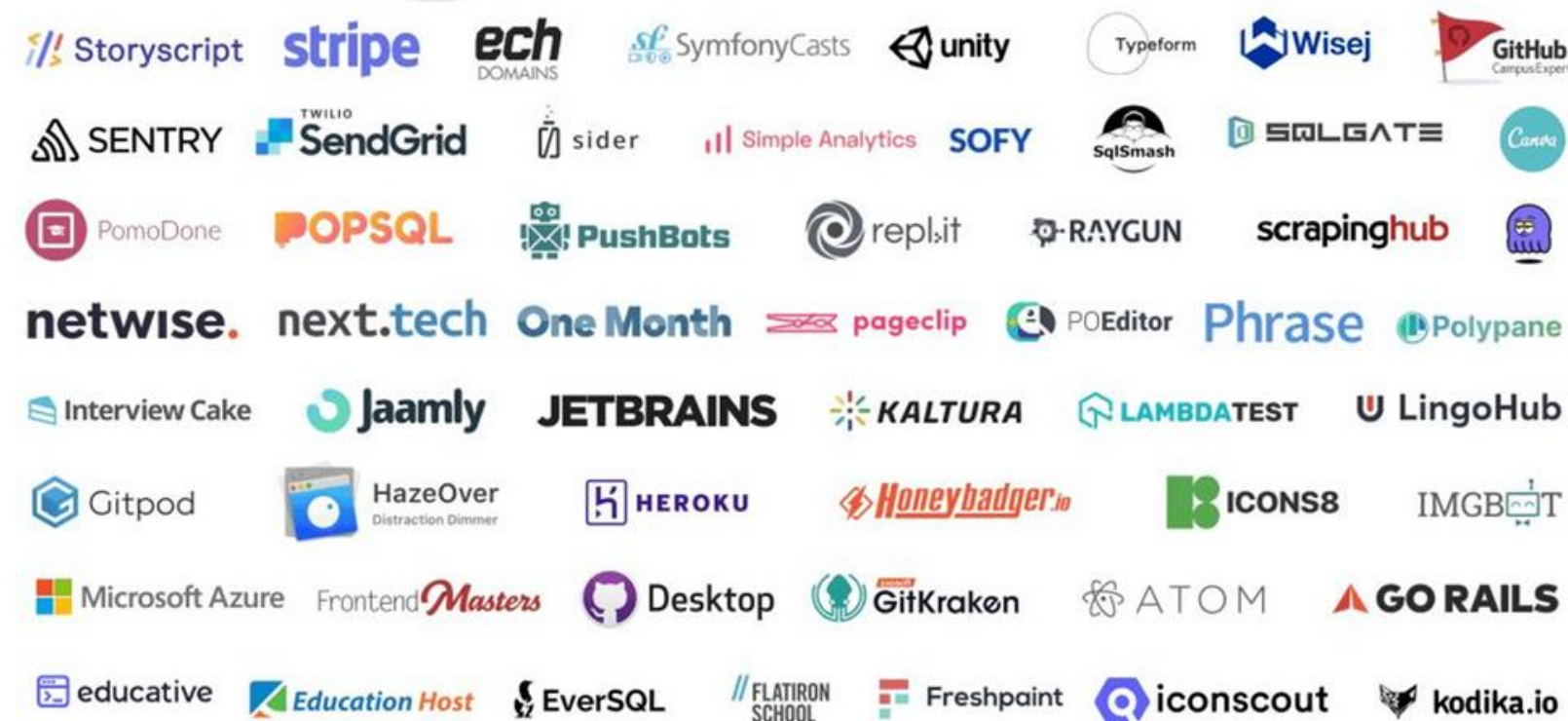




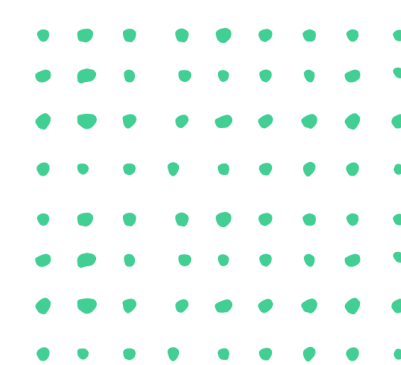
# GITHUB STUDENT DEVELOPER PACK



## GITHUB STUDENT DEVELOPER PACK



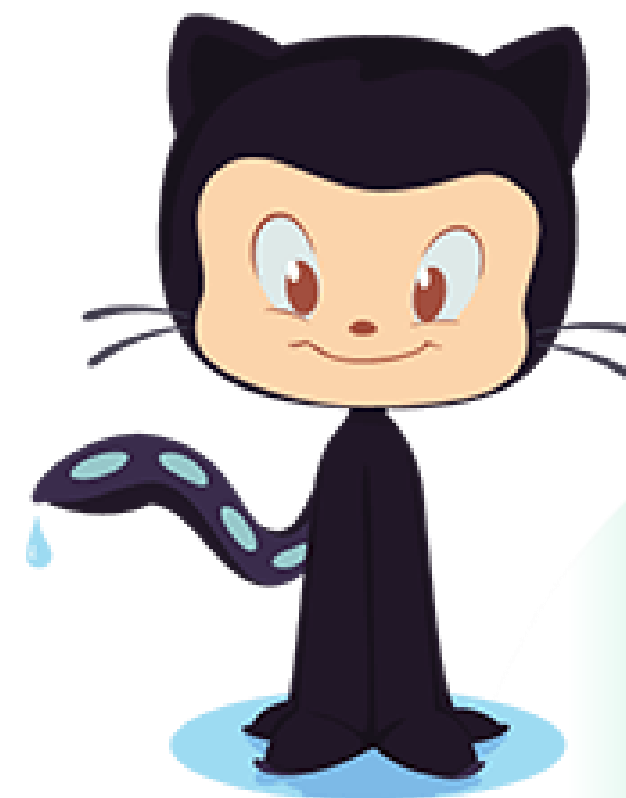
# GIT: INTERACTIVE LEARNING

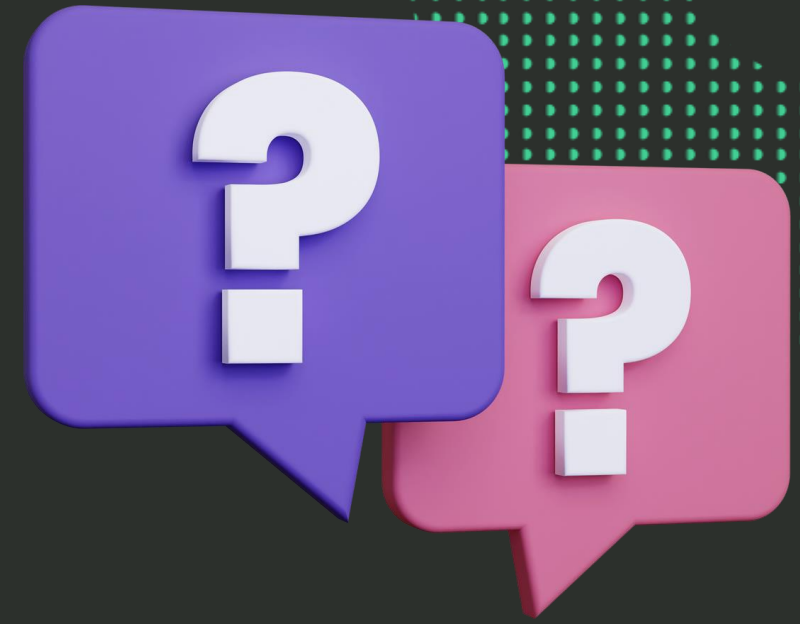


- Following are two good interactive demos for learning git.
- The fundamentals are found in [1] and advanced branching demo is in [2].

[1] <https://try.github.io>

[2] <http://pcottle.github.io/learnGitBranching/>





THAT'S ALL FOLKS !

ANY QUESTIONS ?