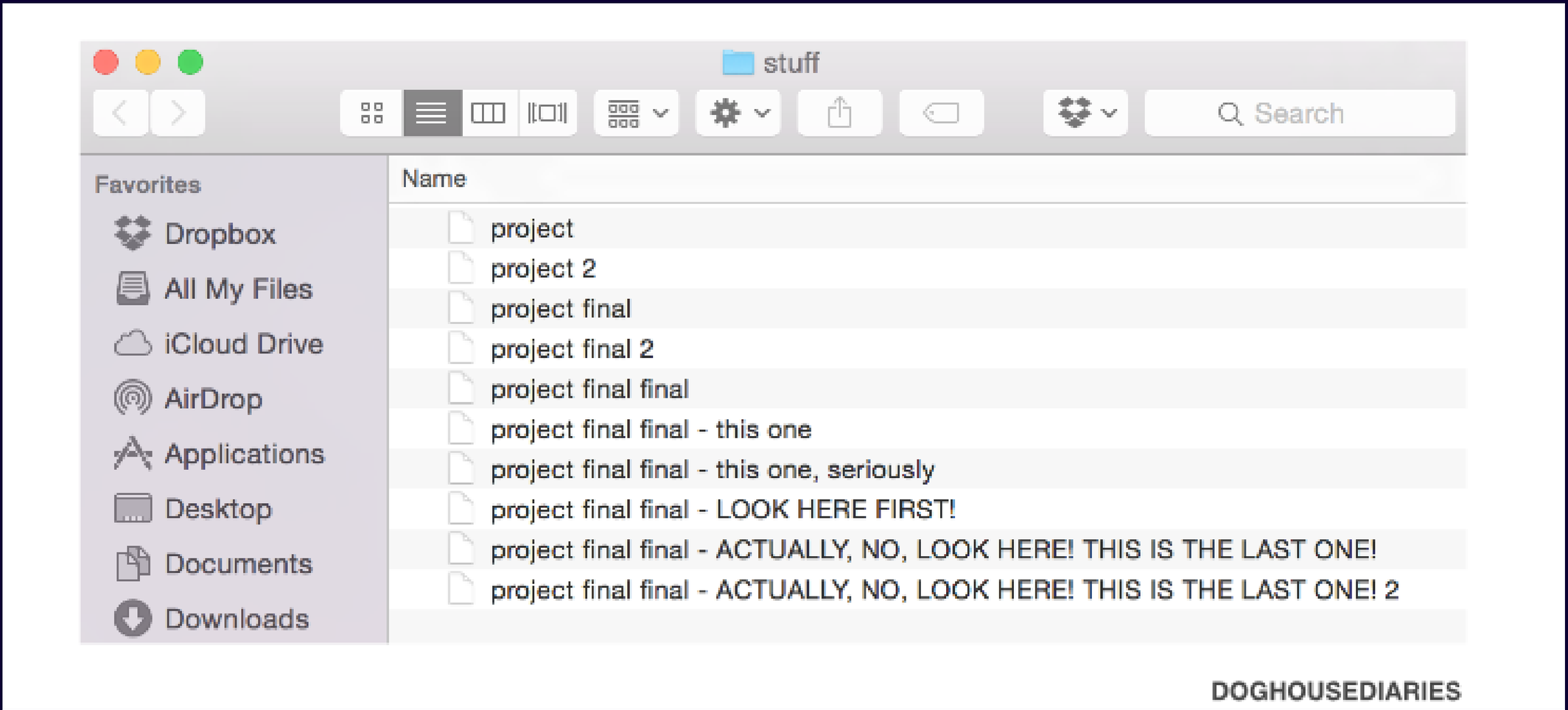
Introduction to Source Code Management with Git

Presented by

Nessim Saidi

SOA PEOPLE
Inspire Performance

Source Code Management: Why Do We Need it?

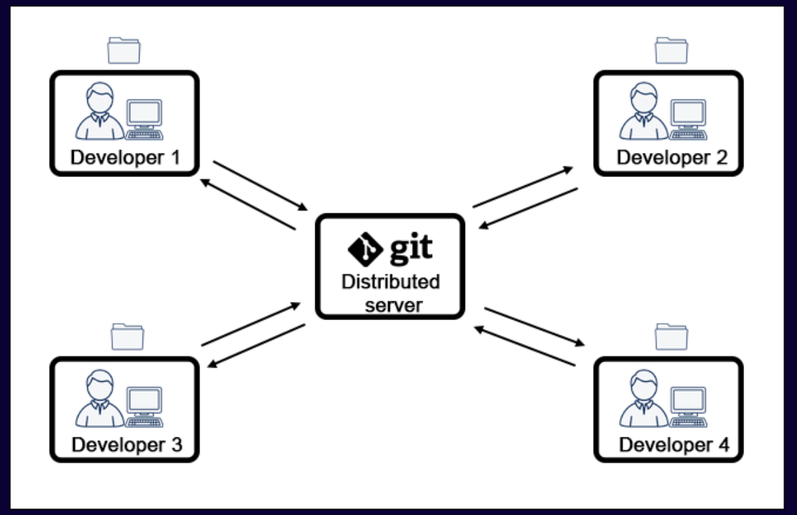


Source Code Management: Why Do We Need it?

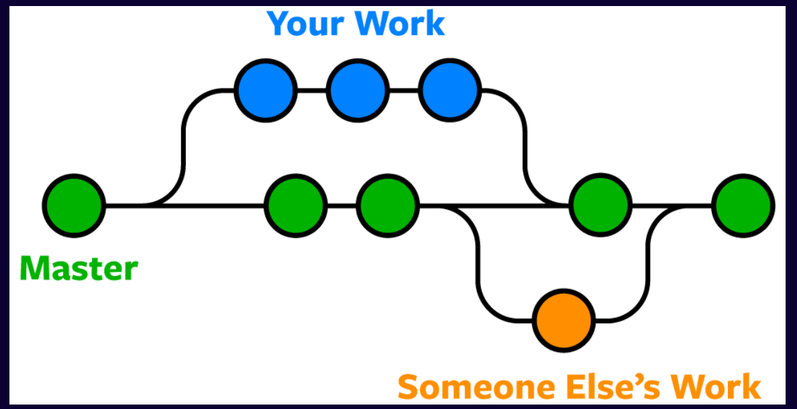
- **Track Changes & History**
  - Restore previous versions, audit changes, and avoid lost work.
- **Collaborate Efficiently**
  - Multiple developers can work on the same project without conflicts.
- **Develop in Parallel**
  - Work on features independently without disrupting the main codebase.
- **Ensure Code Quality**
  - Use branches, reviews, and automated tests to maintain integrity.
- **Automate & Deploy**
  - Enable CI/CD for smooth testing, builds, and releases.

What is Git?

- **Distributed Version Control**
  - Every developer has a “full” copy of the repository.
- **Tracks Changes**
  - Maintains a history of code modifications over time.
- **Branching & Merging**
  - Enables parallel development and integration of changes.
- **Collaboration & Code Sharing**
  - Facilitates teamwork with remote repositories.
- **Rollback & Recovery**
  - Restore previous versions and undo mistakes.
- **Automation & CI/CD Integration**
  - Works with pipelines for testing and deployment.



Source: https://phoenixnap.com/kb/what-is-git



Source: https://www.nobledesktop.com/learn/git/git-branches

Source Code Management: An SAP Perspective

- **Classic version management for ABAP:**
  - Source code stored in central database
  - Environment is mostly bound / specific to system (excluding customization)
  - Trackable versions on Transport Request Release
  - No conflict-free collaboration

- **“Modern” file-based version management (UI5, CAP, etc.) with Git**
  - everything is stored in files within decentralized repositories (with local copies)
  - Versions are stored in Git’s metadata (.git folder) as diff files
  - Environment can be managed in files (configurations, Infrastructure as Code, Dev containers, versioned database, dependencies, pipelines/automation, policies, orchestration & deployment, monitoring/logging, etc.)
  - Trackable versions on commit (version check-in at any time possible)
  - Parallel development and collaboration at heart; conflict-resolution on merge

Source Code Management: An SAP Perspective

Git for ABAP

- **abapGit**
  - Exports ABAP objects as text files
    → enables limited version control, reviews, collaboration, …
  - Main use case: transport development from system to system (e.g. onPremise to BTP ABAP environment; across BTP accounts; partner to customer systems, …)

- **gCTS (Git-enabled Change and Transport System)**
  - Evolution of classic CTS landscape
  - Uses Git workflows (branching, merging) for managing transports
  - Main use cases:
    - Basis for distribution (e.g. transport from DEV to TST)
    - Integration of different sources into one code base as means for Continuous Integration

Installing Git

Use your terminal

Check if Git already exists:

```
git --version
```

Download the Git installer from https://git-scm.com/download/

First-time setup: provide your display name and email address

```
git config --global user.name "John Doe"
git config --global user.email johndoe@soapeople.com
```

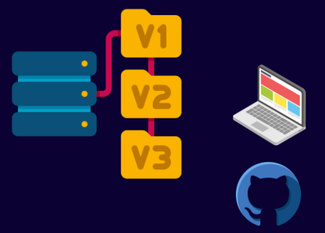
Command-line help

```
git --help
```

Git: Fundamentals

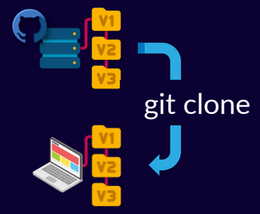
Repository

Server / Local Copy



Cloning

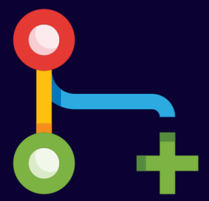
Server



Local Copy

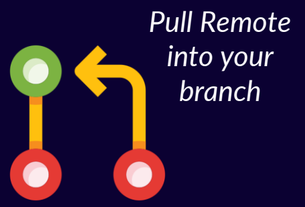
Branching

Main Branch



Your Branch

Pulling



Your Branch

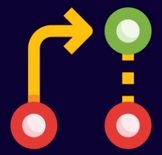
Remote Branch

Committing

Check in new version ("snapshot") as delta

Your Branch

Pushing



Push local changes into Remote

Your Branch

Remote Branch

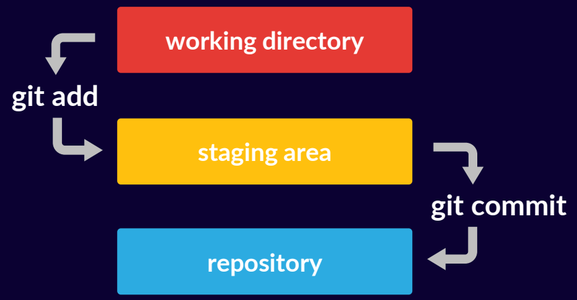
Merging

Merge branch B into A

Branch A

Remote B

Staging



Credits: https://www.flaticon.com/free-icons/commit-git - Commit git icons created by edt.im, Flat Icons, vectorsmarket15 - Flaticon

Lab: First Steps

Use your terminal

Initialize current folder as Git-managed (create a new repository)

```
git init                    # initialize local repo
```

View status of Git repository

```
git status
```

Add file(s) to staging area

```
git add .                   # add all untracked files
git add <filename>          # add specific file
```

Commit all staged files

```
git commit -m “This is a commit message 🥳“
```

Display log / history

```
git log
```

Show last commit

```
git show
```

Open Terminal:

Windows-Key *type* “Terminal”

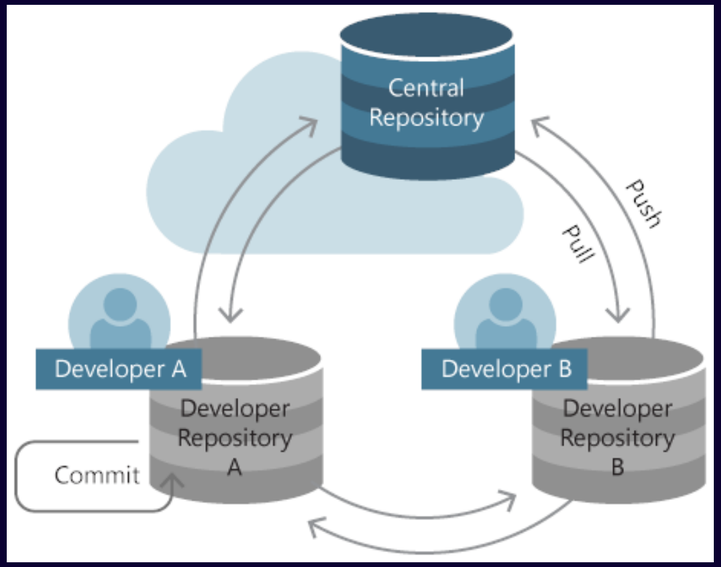
open “Terminal”

Terminal Cheat Sheet:

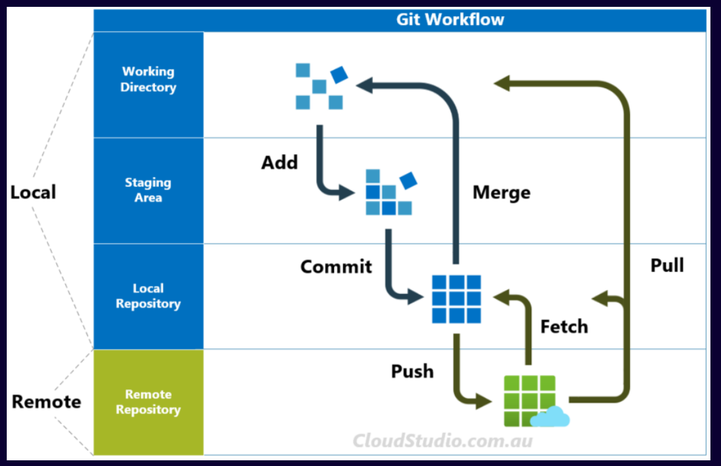
Action	Windows PowerShell	Linux/macOS Command
List files	`dir` *or* `ls`	`ls`
Change directory	`cd <folder_name>`	`cd <folder_name>`
Go up one folder level	`cd ..`	`cd ..`
Create folder	`mkdir <folder_name>`	`mkdir <folder_name>`

Collaboration: Remote Repositories

- **What is a Remote Repository?**
  - A version of your Git repository stored on a remote server (e.g., GitHub, GitLab, Bitbucket).
- **Why Use Remotes?**
  - Enables collaboration, backup, and centralized code sharing across teams.
- **Cloning a Repository**
  - `git clone <repo-url>` creates a local copy of a remote repository.
- **Fetching & Pulling Changes**
  - `git fetch` retrieves updates, while `git pull` fetches and merges changes.
- **Pushing Changes**
  - `git push origin <branch>` uploads local commits to the remote repository.
- **Managing Remotes**
  - `git remote -v` lists linked remote repositories, and `git remote add <name> <url>` adds a new remote.
- **Forks & Pull Requests**
  - Forking a repository creates an independent copy, and pull requests enable contributions to the original repository.



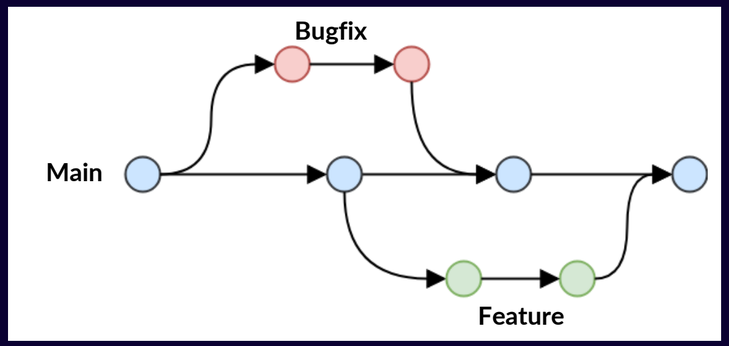
Source: https://learn.microsoft.com/en-us/devops/develop/git/set-up-a-git-repository



Source: https://cloudstudio.com.au/2021/06/26/git-command/

Working with Branches

- **What is Branching?**
  - Create independent lines of development without affecting the main codebase.
- **Why Use Branches?**
  - Work on new features, bug fixes, or experiments without disrupting the stable version.
- **Types of Branches (examples):**
  - **Feature Branches** – Develop new functionality separately before merging.
  - **Hotfix Branches** – Apply urgent fixes directly to production.
  - **Release Branches** – Prepare stable versions before deployment.
- **Key Commands:**
  - `git branch <name>` – Create a new branch.
  - `git checkout <name>` – Switch to a branch.
  - `git merge <branch>` – Merge changes from another branch.
  - `git branch -d <name>` – Delete a branch after merging.



Source: https://docs.wavemaker.com/learn/blog/2021/09/17/git-branching-strategy/

Lab: Branching 1

Clone existing Repository

Use your terminal

```
git clone https://github.com/SOAPeople-NessimSaidi/git-introduction.git
cd git-introduction                          # change directory to new project
```

Create a new branch

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

or

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

Switch to branch

```
git checkout <your-branch-name>          # switch to branch, -b for create & switch
```

Create a new file

```
echo “Hello Git!” > hello.txt
```

Add file to staging area

```
git add hello.txt
```

Commit your change (create a snapshot)

```
git commit -m “Added hello.txt with a welcome message”
```

Lab: Branching 2

Pulling changes from Remote Repository *(optional, but best practice!)*

Use your terminal

```
git pull origin <your-branch-name>          # get updates from remote
```

Pushing changes to Remote Repository in separate branch

```
git push origin <your-branch-name>          # send your updates to remote
```

Merging Workflow *(get changes, change branch, merge push)*

```
git pull origin main                        # get updates of main from remote
git checkout main                           # switch to main branch
```

Merge your branch into main

```
git merge <your-branch-name>     # take contents of your branch and merge into main
```

```
git push origin main                        # push main branch to remote
```

Working with Branches: Best Practices

Best Practices:

- Work in branches to keep the main branch clean and stable.
- Keep branches short-lived and focused. Merge into *main* as often as possible/practicable.
- Regularly sync with *main* to avoid conflicts (i.e. merge *main* into *your branch*).
- Use meaningful branch names (e.g., feature-login, bugfix-404).
- Commit small, incremental changes instead of large updates.
- Use clear commit messages to document what you changed.
  - DO THIS: “Fix login bug by updating authentication logic”, NOT THIS: “some changes”
  - Pro-tip: Use Conventional Commits e.g.: “feat(api): send an email to the customer when a product is shipped”
- Always pull the latest changes before pushing your work!
- IF big changes were done: Consider merging main into your branch prior to merging into main to resolve merge errors in your branch.

Lab: Managing the Repository Lifecycle in Git

View Git history

Use your terminal

```
git log                                          # view history
git log --oneline --graph --decorate --all       # condensed log
git log –p -2                                    # show diff for the last 2 commits
git log --author="Nessim"                        # filter log for author
git log --grep="basic"                           # filter log for message content
```

Comparing and Reviewing Changes

```
git diff                              # see changes that haven’t been committed yet
git diff main feature-branch          # compare diff between two branches
git blame <filename>                  # see who last changed each line in a file
```

Handling Stashes (*“Save your current changes without committing”*)

```
git stash push -m "WIP: Fixing login bug"     # stash with a custom message
git stash list                                # show all saved stashes
git stash apply                               # Apply last stash (keeps it in the stash list)
git stash pop                                 # Apply and delete from stash list
```

Undoing Commits & Manipulating History in Git

Undoing a Commit locally (But Keeping Changes)

Use your terminal

If you've committed changes but haven't pushed them yet, you can **undo the commit while keeping your changes**:

```
git reset --soft HEAD~1          # Reset to previous commit (keep changes staged)
```

✅ **Safe to use in any setting** because it doesn't rewrite shared history.

Reverting a Commit (Safe for Shared Repositories)

If you have already pushed the commit and need to **undo its changes without modifying history**, use:

```
git revert <commit-hash>          # Undo specific commit (keep history intact)
```

This creates a **new commit** that negates the effects of the specified commit.

✅ **Use in shared repositories** because history remains intact.

Interactive Rebase (Rewriting Multiple Commits – Use with Caution!)

To modify, reorder, squash, or remove multiple commits:

```
git rebase -i HEAD~3          # Interactively rebase the last 3 commits
```

❌ Avoid rebasing shared commits! Use for feature branches before merging to main.

Avoid in Shared Repositories (Unless You're Coordinating with Your Team)

If you have already pushed the commit , **DON'T DO** the following:

```
git reset --hard <commit>          # Deletes history permanently
git rebase –i                      # Rewrites history, causes conflicts if already pushed
git push –force                    # Overwrites remote history, affecting teammates.
```

What to Track in Git – And What to Ignore

What to Version Control

- Source code (e.g., .js, .py, .java, .abap)
- Configuration files (if essential, e.g., package.json, pom.xml, .env.example)
- Documentation (e.g., README.md, API specs)
- Infrastructure as Code (e.g., Terraform scripts, Dockerfiles)
- CI/CD configurations (e.g., GitHub Actions, .gitlab-ci.yml)
- Test scripts & test data
- Essential binary assets like logos, icons, design files

What NOT to Version Control

- **Secrets & credentials** (e.g., .env, config.json with API keys, passwords, tokens, etc.)!
- Generated files & binaries (e.g., node_modules/, target/, dist/, .class, .exe)
- Log files & cache (e.g., .log, .cache/, *.swp)
- Personal editor settings (e.g., .vscode/, .idea/)

.gitignore – Keeping Your Repo Clean

To exclude unnecessary files, use a .gitignore file. Example:

```
# Ignore dependencies and build artifacts
node_modules/
dist/
*.log

# Ignore sensitive data
.env
config/secrets.json
```

Git vs. GitHub

Git (Version Control System)

- A Distributed version control system (DVCS) for tracking code changes.
- Works locally on a developer’s machine.
- Enables branching, merging, and history tracking.
- Can be used with any remote repository service (GitHub, GitLab, Bitbucket, etc.).
- Free, open-source

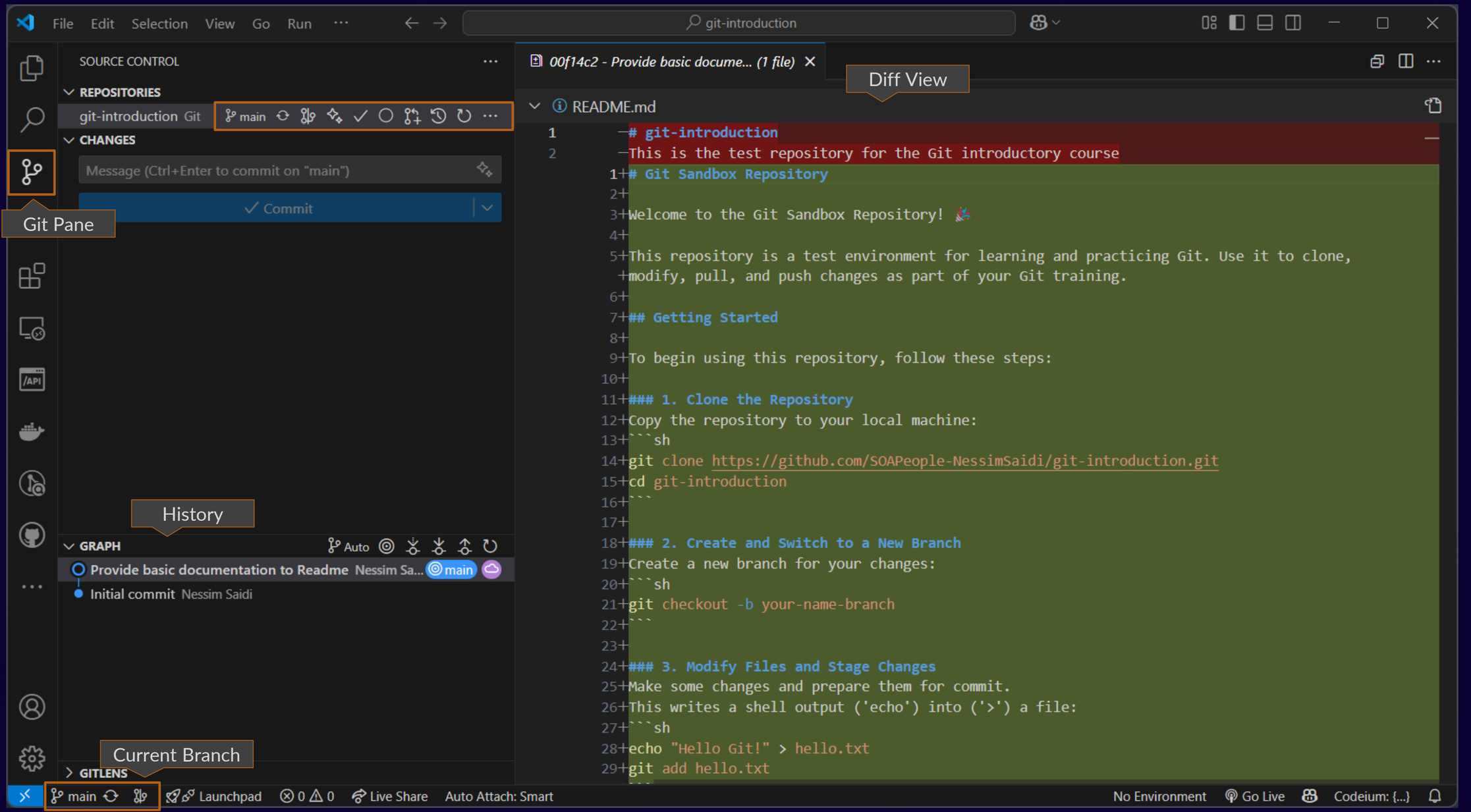
→ Version Management Tool

GitHub (Hosting Service)

- A cloud-based platform for hosting Git repositories.
- Provides collaboration features like pull requests, issue tracking, and discussions.
- Includes CI/CD tools, security scanning, and repository insights.
- Supports public and private repositories with access controls.
- Additional services like Issue tracking, project management, documentation, automation
- Payed services

→ Service Provider and Hoster

Using Git in Visual Studio Code / Business Application Studio



Classic Git Flow

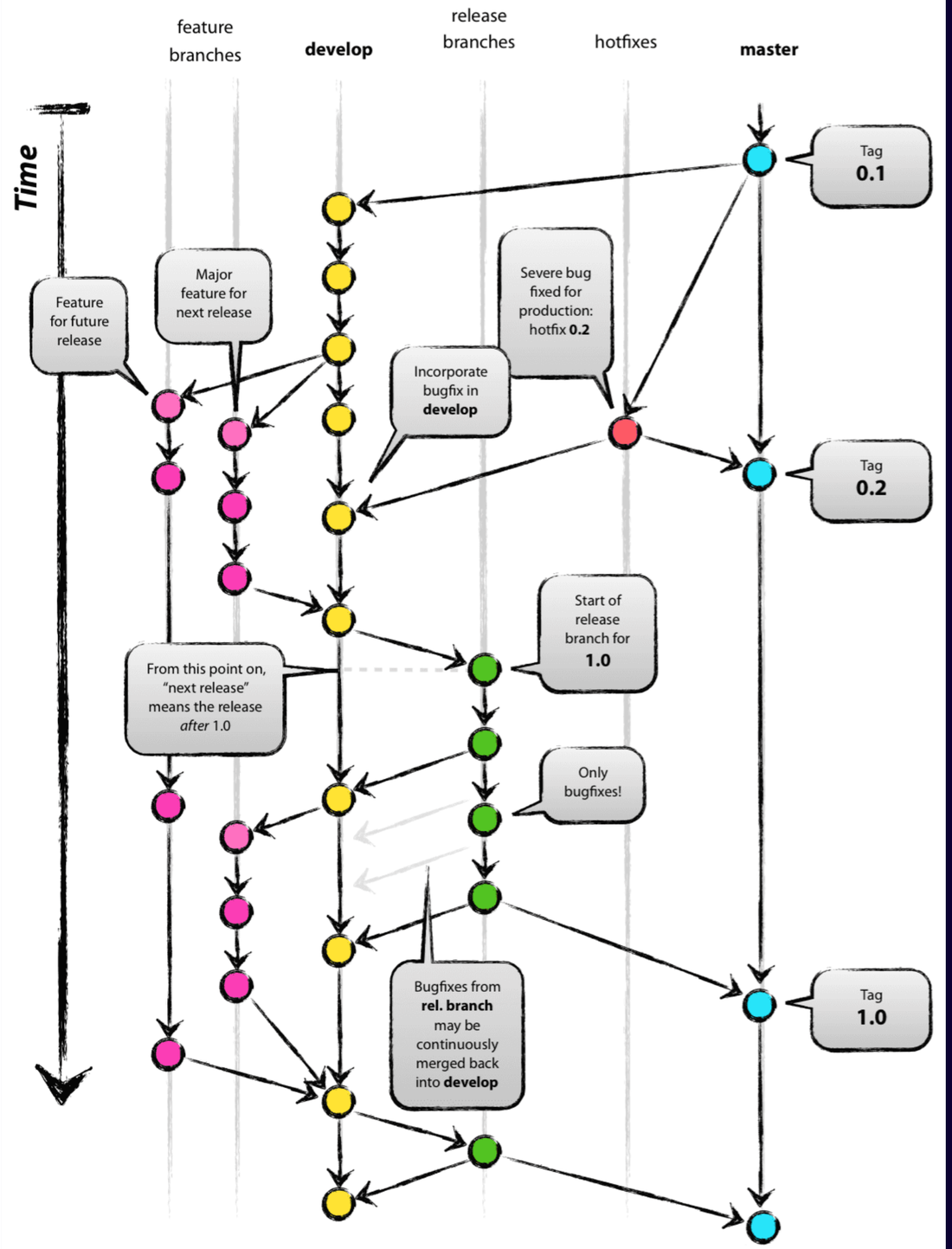
`main` – Stable branch containing production-ready code.

`develop` – Integration branch for ongoing development.

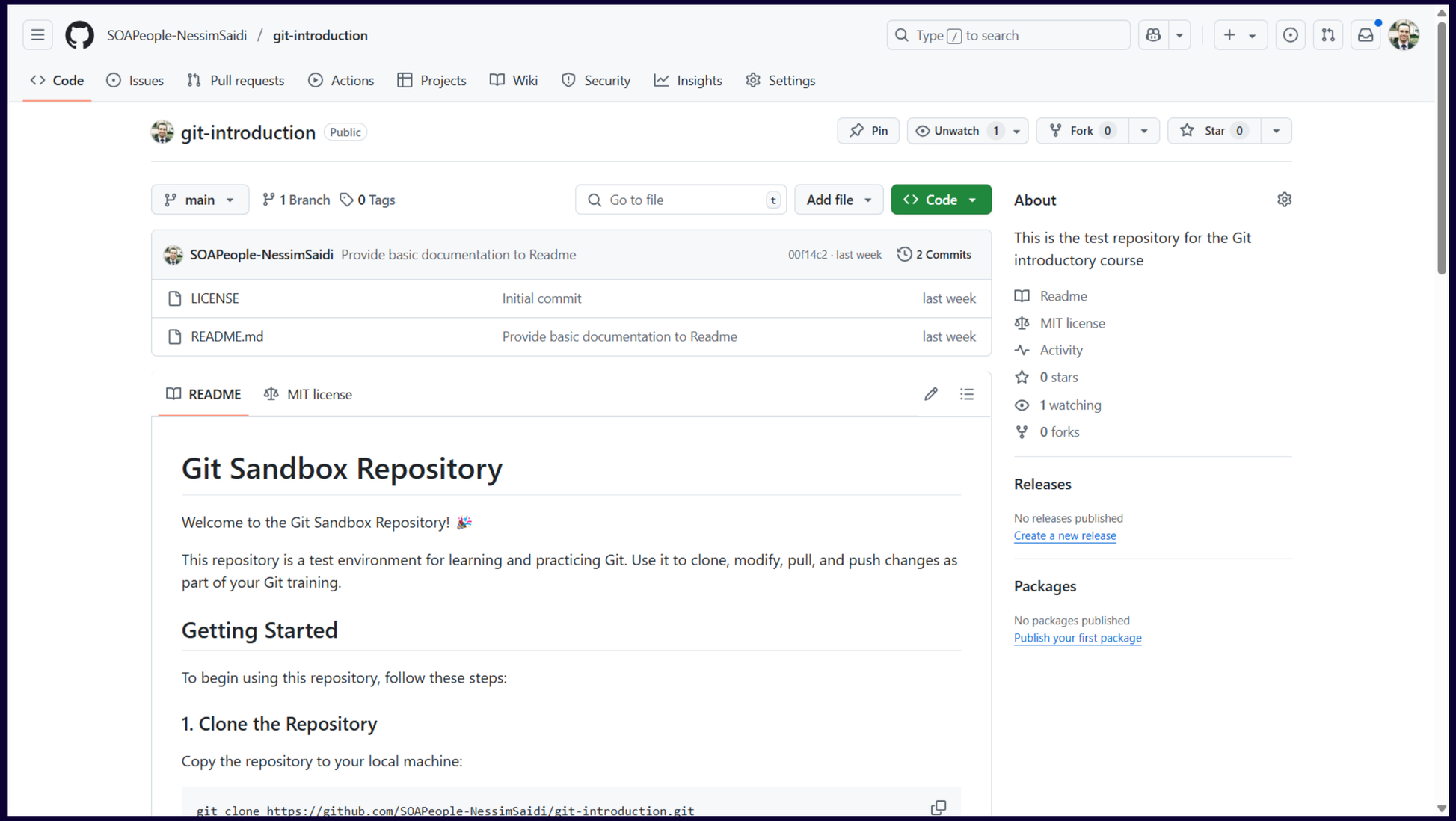
`feature``/*` – Branches for new features, merged into develop after completion.

`release``/*` – Prepares a new production release, branched from develop, and merged into main once stable.

`hotfix``/*` – Emergency fixes for production issues, branched from main, then merged back into main and develop.



Sneak Peek: GitHub



Get SOA People GitHub Account:

- Internal guidelines
- How to request

[GitHub User Guide]

GitHub Flow

`main` – Always deployable, production-ready branch.

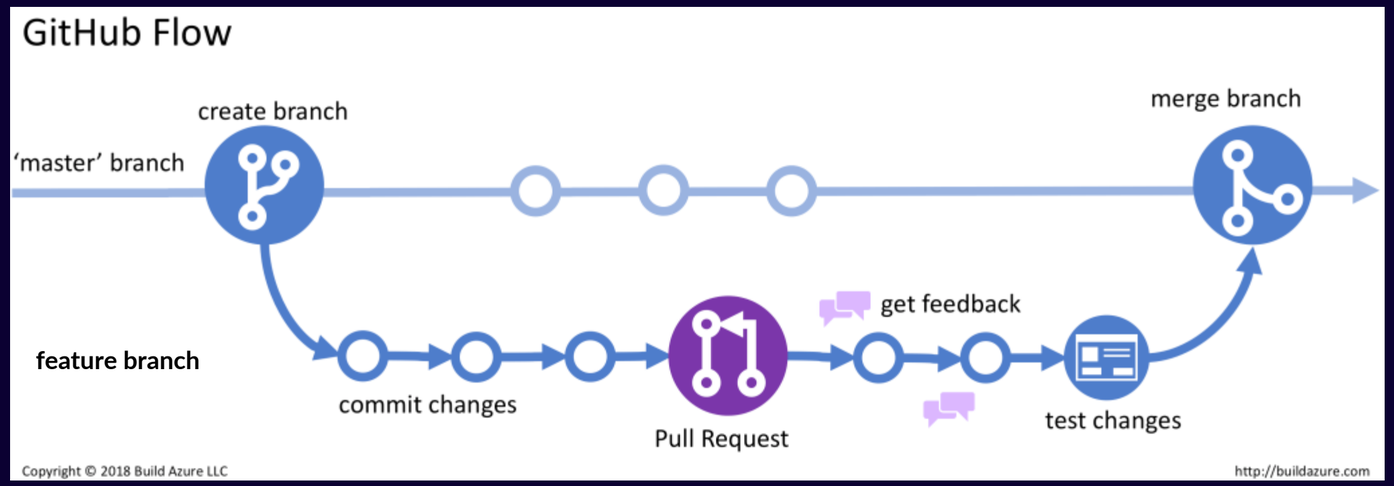
`feature``/*` – Short-lived branches for new features or fixes, branched from main.

`Commit & Push` – Frequent commits pushed to the feature branch.

`Pull Request (PR)` – Code review and discussion before merging.

`Merge to main` – Once approved, the feature branch is merged into main.

`Deploy Continuously` – Changes in main trigger automatic deployments.



→ Simpler than Classic Git Flow

Can be extended e.g. for CI/CD:

- Use automation, trigger on Pull Request
- Run test suites
- Deploy to test/prod environment

Integrating GitHub with Development Tools

Integrated Development Environments (IDEs)

- VS Code, IntelliJ, Eclipse, etc. have built-in GitHub integration
- Enable Git operations like commits, branches, and pull requests within the UI
- Use extensions (e.g., GitHub Copilot, GitLens for VS Code)

Authentication Methods

- Personal Access Tokens (PATs) for secure API authentication — Managing Personal Tokens
  - Treat tokens like passwords! Make sure not to check in to agit repository.
- ⭐ **SSH Keys** for secure, passwordless authentication from local machines — Connecting with SSH
  - Creates trust between your local dev machine and GitHub
  - Recommended method
- Git Credential Manager (GCM) for seamless HTTPS authentication

Outlook: Advanced features

- **Git**
  - Rebasing (git rebase) – Rewrite commit history for a cleaner linear timeline.
  - Squashing (git rebase -i) – Combine multiple commits into one before merging.
  - Cherry-picking (git cherry-pick) – Apply specific commits from one branch to another.
  - Submodules (git submodule) – Manage external repositories inside a project.
- **GitHub**
  - GitHub Actions – Automate workflows with CI/CD
  - GitHub Packages – Host and manage dependencies
  - Code Owners – Define required reviewers for Pull Requests
  - Branch Protection Rules – Enforce rules on main branches
  - GitHub Projects – Manage tasks and issues like a Kanban board
  - Security & Dependabot – Automated dependency scanning and updates
  - Code Scanning – Detect security vulnerabilities in your code

SOA PEOPLE
Inspire Performance

Thank you !

Nessim SAIDI
Developer, D&I Community Lead

nessim.saidi@soapeople.com