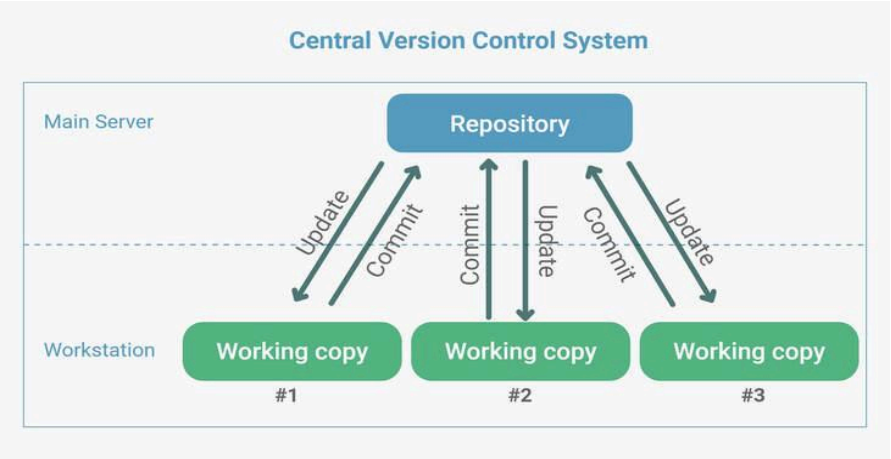
About Version Control

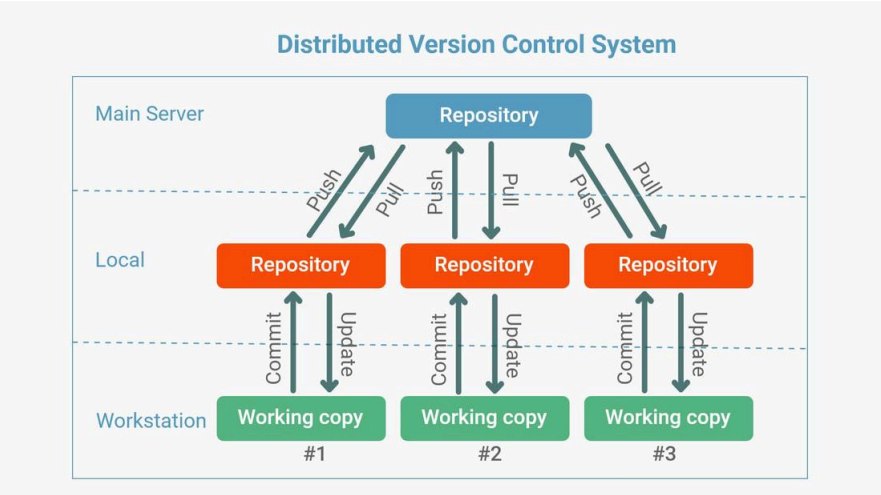
What is “version control”, and why should you care?

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book, you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer. If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

Centralized Version Control Systems The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control. Centralized version control This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it’s far easier to administer a CVCS than it is to deal with local databases on every client. However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they’re working on. If the hard disk the central database is on becomes corrupted, and proper backups haven’t been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCSs suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything



Distributed Version Control Systems This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data. Distributed version control Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models



Git is a distributed version control system (DVCS) that allows developers to track changes in their codebase, collaborate with others, and manage different versions of their projects efficiently. It was created by Linus Torvalds in 2005 and has become the de facto standard for version control in software development due to its speed, flexibility, and distributed nature.

Key features of Git include:

1. **Version Control**: Git tracks changes to files and directories, allowing developers to revert to previous versions, compare changes over time, and collaborate effectively.
2. **Branching and Merging**: Git enables developers to create branches to work on features or fixes independently of the main codebase. Branches can be merged back into the main branch (typically `master` or `main`) once changes are complete and tested.
3. **Distributed Development**: Each developer has a complete copy of the repository, including its entire history. This allows for offline work and enables teams to work collaboratively without being dependent on a central server.

4. **Speed and Efficiency**: Git is designed to be fast, even with large repositories and extensive history. Operations like committing changes, branching, and merging are optimized for performance.

GitHub, on the other hand, is a web-based hosting service for Git repositories. It provides a platform where developers can store, share, and collaborate on their Git repositories, both privately and publicly. GitHub adds several features on top of Git, including:

1. **Remote Repositories**: GitHub hosts remote repositories, allowing developers to push their local changes to a shared repository accessible from anywhere.
2. **Collaboration Tools**: GitHub provides tools for code review, issue tracking, project management, and team collaboration. Features like pull requests enable developers to propose changes, review them, and merge them into the main repository.
3. **Community and Open Source**: GitHub has become a central hub for open-source software development, enabling developers worldwide to contribute to projects, discover new ones, and build software together.
4. **Integration and Extensibility**: GitHub integrates with various third-party tools and services, such as CI/CD pipelines, code quality checks, and deployment platforms, enhancing the development workflow.

In summary, Git is the version control system that tracks changes in your codebase, while GitHub is a platform that hosts Git repositories and provides collaboration tools for teams and communities of developers. Many developers and organisations use Git alongside GitHub to manage their software projects effectively.

A Short History of Git

As with many great things in life, Git began with a bit of creative destruction and fiery controversy. The Linux kernel is an open source software project of fairly large scope. During the early years of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files. In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper. In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were as follows:

- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities. It's amazingly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development

The workflow of Git:

Git workflows define how developers collaborate on a project using Git for version control. There are several Git workflows, each tailored to different team sizes, project complexity, and development goals. Here, I'll outline a commonly used workflow known as the **Feature Branch Workflow**:

Feature Branch Workflow

1. **Create a Feature Branch**:
    - Start by creating a new branch for each new feature or bug fix. Use a descriptive name that indicates the purpose of the branch.

bash
Copy code
```
git checkout -b feature/new-feature-name
```

2. This command creates a new branch (`feature/new-feature-name`) and switches to it (`git checkout`).
3. **Work on the Feature**:
    - Make changes to files, add them to the staging area, and commit them to the feature branch.

bash
Copy code
```
git add .

git commit -m "Implemented feature X"
```

4.
5. **Push the Feature Branch**:
    - Periodically push your feature branch to the remote repository (like GitHub) to share your progress and collaborate with others.

bash
Copy code
```
git push origin feature/new-feature-name
```

6.
7. **Review and Merge**:
    - Once the feature is complete and tested, initiate a pull request (PR) on GitHub (or similar platforms) to merge your changes into the main branch.
    - Request code review from team members to ensure quality and adherence to coding standards.
    - Make necessary changes based on feedback before merging.
8. **Merge and Delete Branch**:

    - After approval, merge your feature branch into the main branch (`master` or `main`). This integrates your changes with the stable codebase.
    - Delete the feature branch both locally and remotely once it's merged and no longer needed.

bash
Copy code
```
git checkout main

git pull origin main  # Fetch latest changes from main branch

git merge --no-ff feature/new-feature-name  # Merge feature branch into main

git push origin main  # Push merged changes to remote

git branch -d feature/new-feature-name  # Delete local feature branch

git push origin --delete feature/new-feature-name  # Delete remote feature branch
```

9.
10. **Update Local Repository**:
    - Periodically update your local repository with changes from the main branch to stay synchronized with the latest developments in the project.

bash
Copy code
```
git checkout main

git pull origin main
```

11.

Benefits of the Feature Branch Workflow:

- **Isolation**: Each feature or fix is developed in isolation, minimizing conflicts with other developers' work.
- **Collaboration**: Enables multiple developers to work on different features concurrently without interfering with each other.
- **Code Quality**: Facilitates code review and testing before integrating changes into the main codebase, ensuring higher quality and stability.
- **Flexibility**: Developers can experiment and iterate on features without impacting the stability of the main branch.

This workflow is widely adopted due to its simplicity, flexibility, and ability to support both small and large teams. Adjustments can be made to suit specific project requirements or

organizational preferences, such as using different branching strategies or incorporating CI/CD pipelines for automated testing and deployment.

Branching in Git

Branching in Git refers to the practice of diverging from the main line of development (often referred to as the `master` or `main` branch) to work on different features, fixes, or experiments independently. It allows developers to isolate changes and work on them without affecting the main codebase until they are ready to integrate those changes back.

Key Concepts of Branching in Git:

1. **Main Branch**: The main branch (often `master` or `main`) represents the stable version of the project. It typically contains the production-ready code.
2. **Creating a Branch**: To create a new branch in Git, you use the `git branch <branch-name>` command. This command creates a new pointer to the current commit (HEAD) where you can start making changes.
3. **Switching Branches**: You can switch between branches using the `git checkout <branch-name>` command. This allows you to work on different features or fixes in separate branches.
4. **Committing Changes**: In each branch, you can make changes to files, add them to the staging area with `git add`, and commit them with `git commit`. These changes are isolated within the branch until you merge them into another branch.
5. **Merging Branches**: Once you have completed work on a branch and tested your changes, you can merge the changes back into the main branch (or any other target branch). This integrates the changes from the source branch into the target branch, combining histories and applying the changes.
6. **Branch Management**: Git provides commands to list branches (`git branch`), delete branches (`git branch -d <branch-name>`), rename branches, and view branch histories (`git log --oneline --graph --all`) to manage branches effectively.

Benefits of Branching:

- **Isolation**: Branching allows developers to work on different features or fixes in isolation, preventing interference with the main codebase until changes are tested and ready.
- **Collaboration**: Teams can collaborate more effectively by working on different aspects of a project concurrently in separate branches.
- **Experimentation**: Branches can be used for experimentation or prototyping without affecting the stability of the main branch.
- **Feature Development**: Feature branches allow for incremental development and testing of new features before they are integrated into the main codebase.

In summary, branching in Git is a powerful feature that enables flexible and efficient software development workflows, supporting collaboration, experimentation, and incremental feature

development. Mastering branching concepts and practices is essential for effectively managing code changes and maintaining project stability in collaborative environments