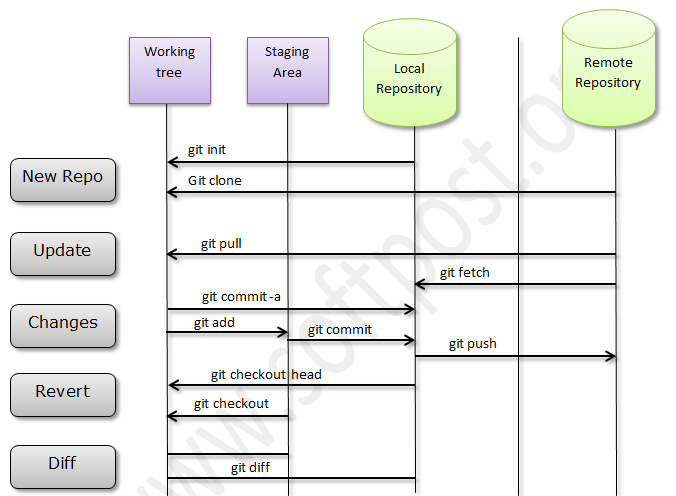
IRAM ABID

GIT ARCHITECTURE

# **GIT ARCHITECTURE**

Git is a version control system that helps keep track of changes made to files and directories. This feature is crucial for maintaining a detailed history of your project’s evolution and offers a safety net for reverting changes if necessary.

When a developer creates new repository or clones existing one, he gets his own copy of the entire repository on his machine. He can make commits to local repository and once all changes are finished, he can push that to remote repository. Below image shows how the entire architecture of Git looks like



**The Key Components of Git:**

Central to Git’s functionality is its object store, a repository component that encapsulates project’s data—including files, log messages, author details, and the metadata necessary to reconstruct any version or branch of your project.

Git employs four primary object types within the object store: blobs, trees, commits, and annotated tags. These elements are the building blocks of Git’s complex data structures, each serving a unique purpose in version control and project management:

* **Blob:** This object type represents the contents of a file within your repository, storing the data without any metadata. Each blob is uniquely identified by a SHA-1 hash, making it a fundamental component for tracking file changes.
* **Tree:** Acting as a directory, a tree object organizes the structure of a project by referencing blobs and other trees. It details filenames and file permissions, structuring the repository’s hierarchy.
* **Commit:** The commit object captures the state of a project at a given moment. It links to the project’s top-level tree for that snapshot, includes metadata such as author and commit messages, and connects to previous commits to maintain a continuous history.
* **Annotated Tag:** Serving as a milestone marker, an annotated tag points to specific commits, often used to denote version releases. It includes additional information like the tagger’s details and a message.

These objects, stored in the .git/objects directory and identified by their SHA-1 hash, lay the groundwork for Git’s versioning capabilities. They enable the precise tracking of project evolution, support state management, and enhance collaborative workflows.

**How Git Tracks Content**

It is crucial to understand Git’s role not just as a Version Control System (VCS) but as a sophisticated content tracking system. This system transcends traditional file name tracking, focusing instead on the content within those files, a method that significantly enhances efficiency and accuracy in version control. As projects evolve, the object store grows with new edits, additions, and deletions. To optimize disk space and network bandwidth, Git efficiently compresses these objects into pack files within the same store.

Git operates as a distributed version control system (DVCS), enabling each contributor to maintain a local repository clone. This clone encompasses a complete copy of the central repository, allowing for offline project work on individual computers. Such a structure not only decentralizes the development process but also enhances productivity by enabling contributors to work independently before synchronizing changes with the central repository.

This approach diverges significantly from centralized version control systems (VCS) like SVN and CVS, where all project files reside on a central server. Centralized systems manage changes through a single “trunk” for stable versions and “branches” for new features, with each branch merging back into the trunk upon completion. However, this model necessitates constant communication with the central server, potentially slowing down the development process.

Git’s distributed nature affords it a considerable speed advantage over centralized counterparts, primarily because it minimizes the need for constant remote server communication. Optimizing data transmission through delta encoding, Git only sends changes between files across the network. This efficiency is particularly noticeable in large repositories, where transmitting entire files would be impractically time-consuming.

1. **Staging Index**

Within the intricate Git concepts architecture, the staging index, or simply the index, emerges as a pivotal component for managing changes before they are committed to the repository. This binary file, nestled within the .git directory at your project’s root, acts as a conduit between your working directory and the repository, capturing a snapshot of your intended changes.

Initially, the index acts as a holding area—a snapshot of your modifications poised for the next commit. This is where git add comes into play, moving changes from your working directory to the staging index, readying them for inclusion in your project’s history. It’s akin to an “out basket” filled with completed tasks, meticulously organized for a final review before being committed.

Moreover, during merges, the index helps in segregating changes that merge smoothly from those that don’t, facilitating a focused approach to resolving conflicts. Beyond tracking changes, the index also monitors file permissions, ensuring that the project’s structural integrity is preserved. The index stands as a bridge to a refined and organized repository state.

1. **Repositories**

At the core of Git concepts architecture lies the Git repository, often simply called a “repo.” A repository in Git serves as a comprehensive database that meticulously stores all necessary information to manage and track the revisions and history of a project. Unique to Git, and setting it apart from most version control systems, is that a repository houses not only a full working copy of all project files but also an entire copy of the repository data itself.

First, each repository is entirely local to your system, including all files, branches, and the complete change history. This local repository is managed through the .git directory, a critical component where Git stores all repository data, including objects such as blobs, trees, commits, and tags, as well as refs like branches and tags. Creating a new Git repository can be achieved through two main approaches: initialization of a new repo with the git init command for directories not under version control, or cloning an existing repository using the git clone command, perfect for contributing to ongoing projects.

Moreover, while eGit repository is self-contained on your local machine, it’s designed to seamlessly connect with remote repositories. Git’s architecture supports this distributed nature, maintaining configuration values and repository settings on a per-site, per-user, and per-repository basis, which are not copied during cloning operations. Within the repository, Git manages two primary data structures: the object store and the index, both stored in the .git directory at the root of your working directory.

1. **Git Object Store**

The Git object store is organized and implemented as a content-addressable storage system. Specifically, each object in the object store has a unique name produced by applying SHA1 to the contents of the object, yielding an SHA1 hash value. Because the complete contents of an object contribute to the hash value and the hash value is believed to be effectively unique to that particular content, the SHA1 hash is a sufficient index or name for that object in the object database. Any tiny change to a file causes the SHA1 hash to change, causing the new version of the file to be indexed separately.

SHA1 values are 160-bit values that are usually represented as a 40-digit hexadecimal number, such as 9da581d910c9c4ac93557ca4859e767f5caf5169. Sometimes, during display, SHA1 values are abbreviated to a smaller, unique prefix. Git users speak of SHA1, hash code, and sometimes object ID interchangeably.

Globally Unique Identifiers An important characteristic of the SHA1 hash computation is that it always computes the same ID for identical content, regardless of where that content is. In other words, the same file content in different directories and even on different machines yields the exact same SHA1 hash ID. Thus, the SHA1 hash ID of a file is an effective globally unique identifier.

A powerful corollary is that files or blobs of arbitrary size can be compared for equality across the Internet by merely comparing their SHA1 identifiers.

1. **The Working Tree**

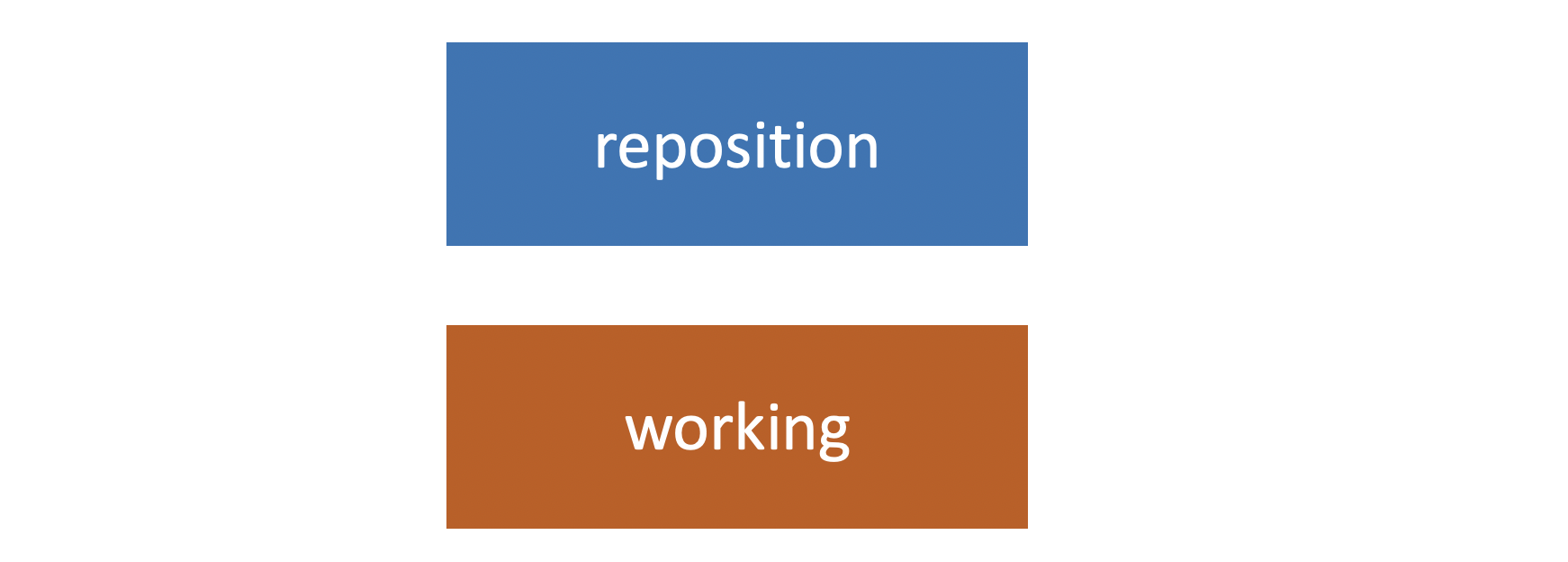
The working tree, or working directory, marks the space where developers actively engage with their project files. It’s where modifications are made before being officially recorded in the repository’s history.

The working tree encompasses the files and directories you’re currently modifying, acting as a sandbox for experimentation and changes prior to committing them to the Git repository. Your working tree reflects the immediate state of your project, showcasing active developments and adjustments outside the .git folder, which houses the repository’s data but remains distinct from the working tree itself.

**Three-Tree Architecture**

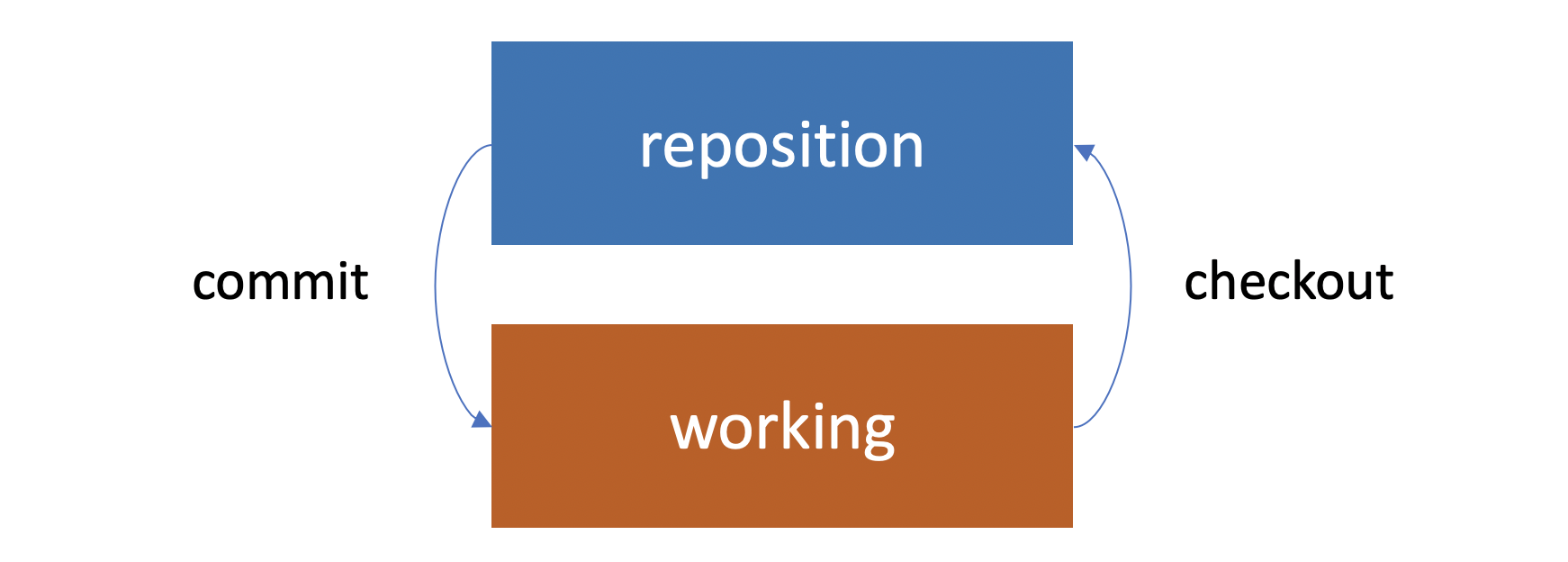
**1. Two-tree architecture**

Let’s start by taking a look at a typical two-tree architecture. This is what a lot of other source version control systems use. They have a repository and a working copy. These are the two trees.



We call them trees because they represent a file/folder structure. The main project directory is at the top, and below it might be 4 or 5 different folders with a few files inside. Maybe a few more folders, each of those folders has a few more folders inside. You can imagine that if you map that out, each of those folders would branch out like the branches of a tree.

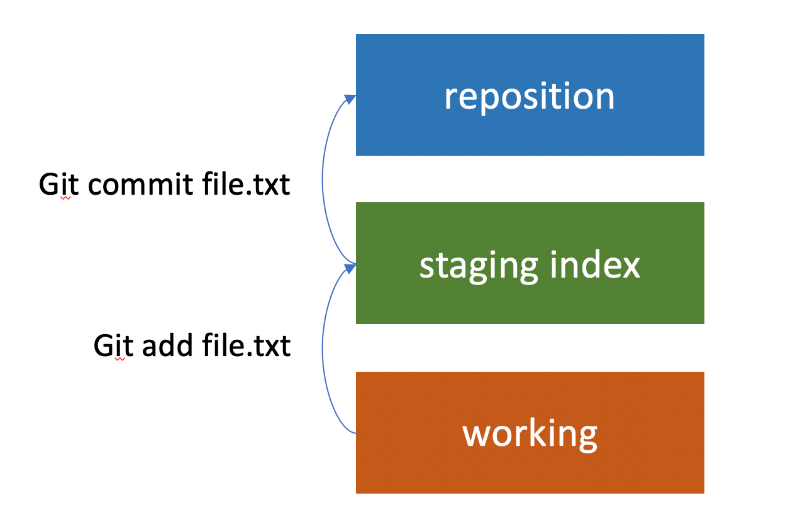
The repository also has a set of files/folders in it, also arranged as a tree. When we want to move files between the repository and the working copy, we checkout copies, that’s the term we use. We check it out from our repo into the working directory, and when we finish making our changes to the files, we commit those changes back to the repo. We have two distinct trees because the files may not be the same.



Imagine if we check out a copy from the repository. We make some changes to it, we save those changes on our hard drive. Those changes are now permanent. They’re saved in our working copy but they’re not yet committed to the repository. Our working copy would look different from the repository. Both are saved, but they’re in different states.

We can imagine another case. If the repository is a shared repository and many people are working from it, they may add their own changes to the repository. If we haven’t checked out a copy recently, then our working copy won’t have its changes. Now that’s a typical two-tree architecture.

**2. Git uses a three-tree architecture**

It still has the repository and the working copy, but in between is another tree, which is the staging index. When we made our first commit, we didn’t just perform a commit. First, we used the add command. We added, then we committed. It was a two-step process (added our files to the staging index, and then from there we committed to the repository).

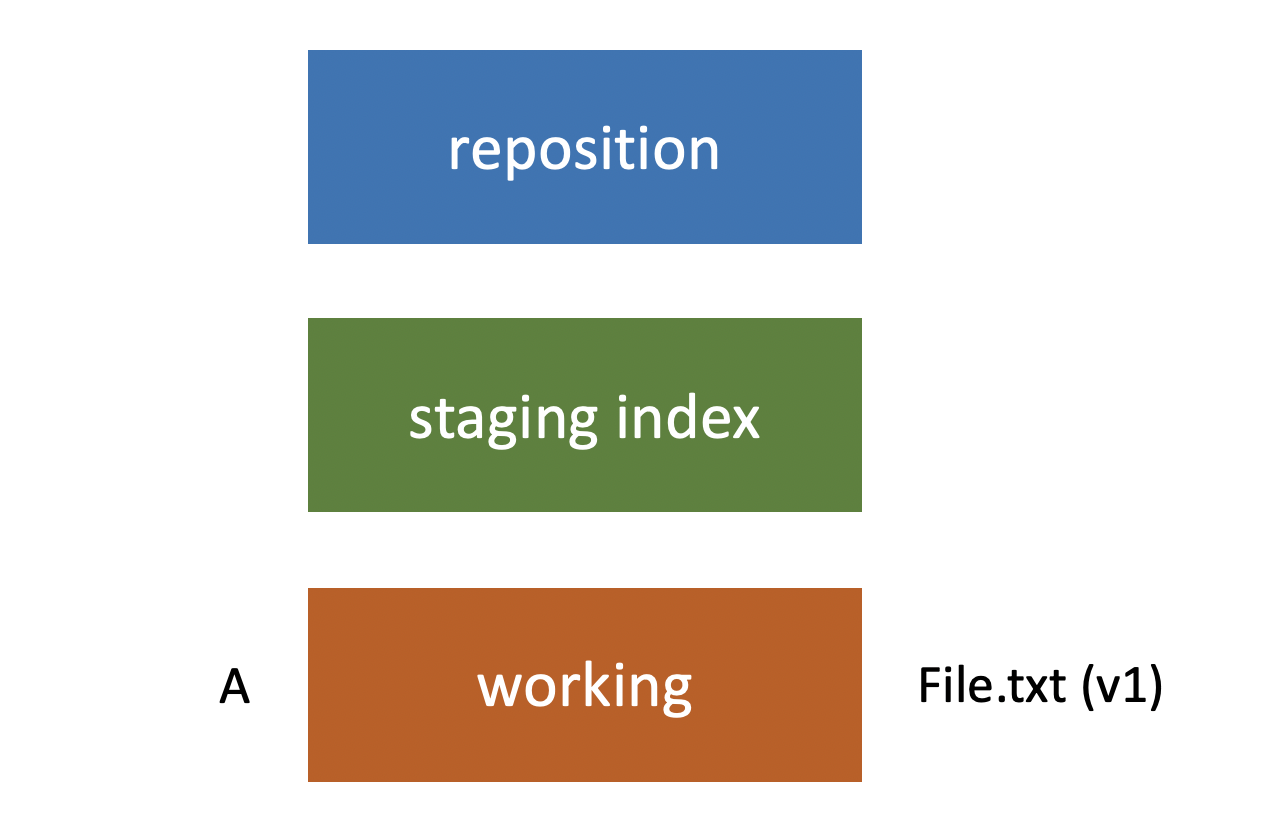
It is possible to just commit directly to the repository and skip adding them to the staging index. But it’s important to understand that this is part of the architecture of Git. It’s a really nice feature because we can make changes to ten different files in our working copy, and then selectively commit five of the changes as one set. That’s why it’s called staging. We have the chance to add the changes that we want to the staging, and then get them ready before we commit them. And we can check out changes from the repository the same way.

It’s also possible to pull them from the repository into the staging index, and then from the staging index to the working directory, but most of the time, that’s not what we do. Usually, we go ahead and pull them straight from the repository, down into the working directory. In the process, the staging index will also be updated too.

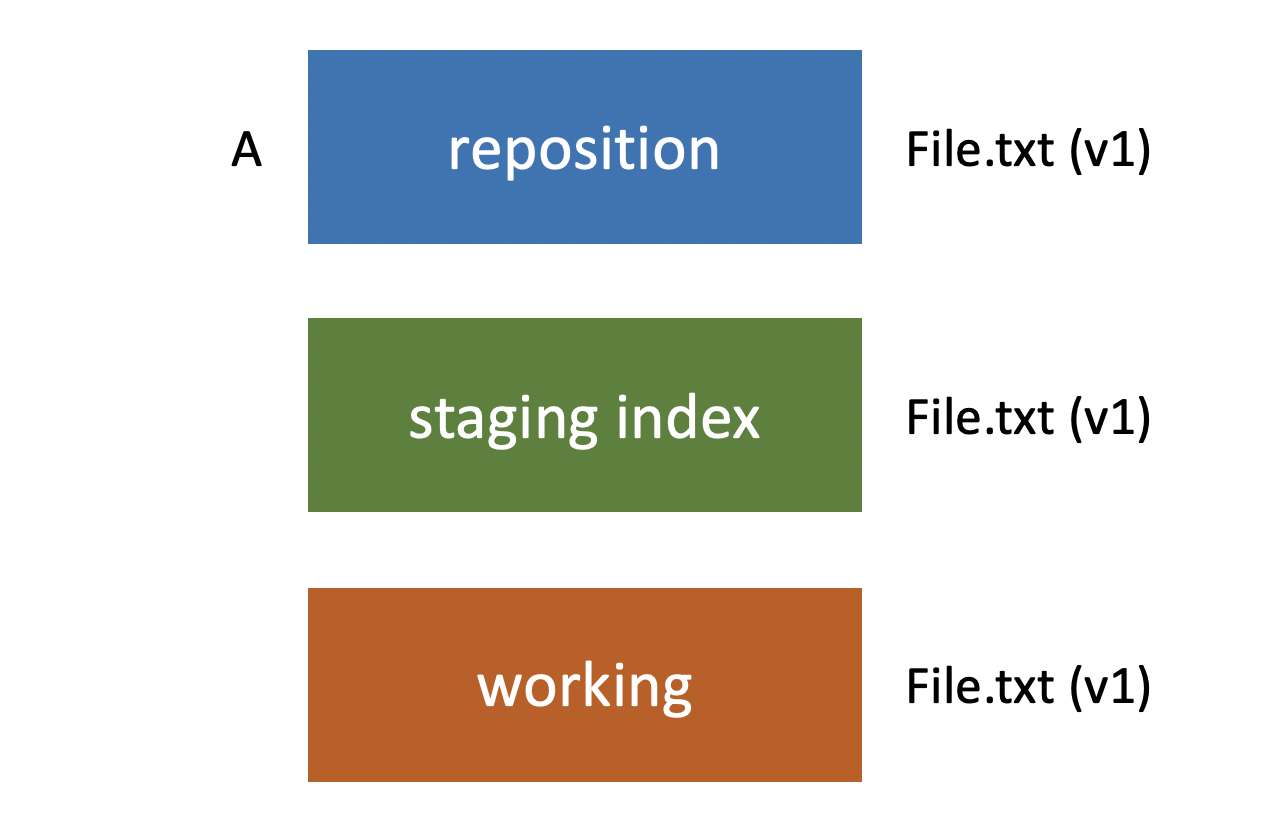
As we’re working with Git, it’s useful to keep these three different trees in mind. There’s our working directory, which contains changes that may not be tracked by Git yet, there’s the staging index, which contains changes that we’re about to commit into the repository, and then there’s the repository, and that’s what’s actually being tracked by Git.

**Git Workflow**

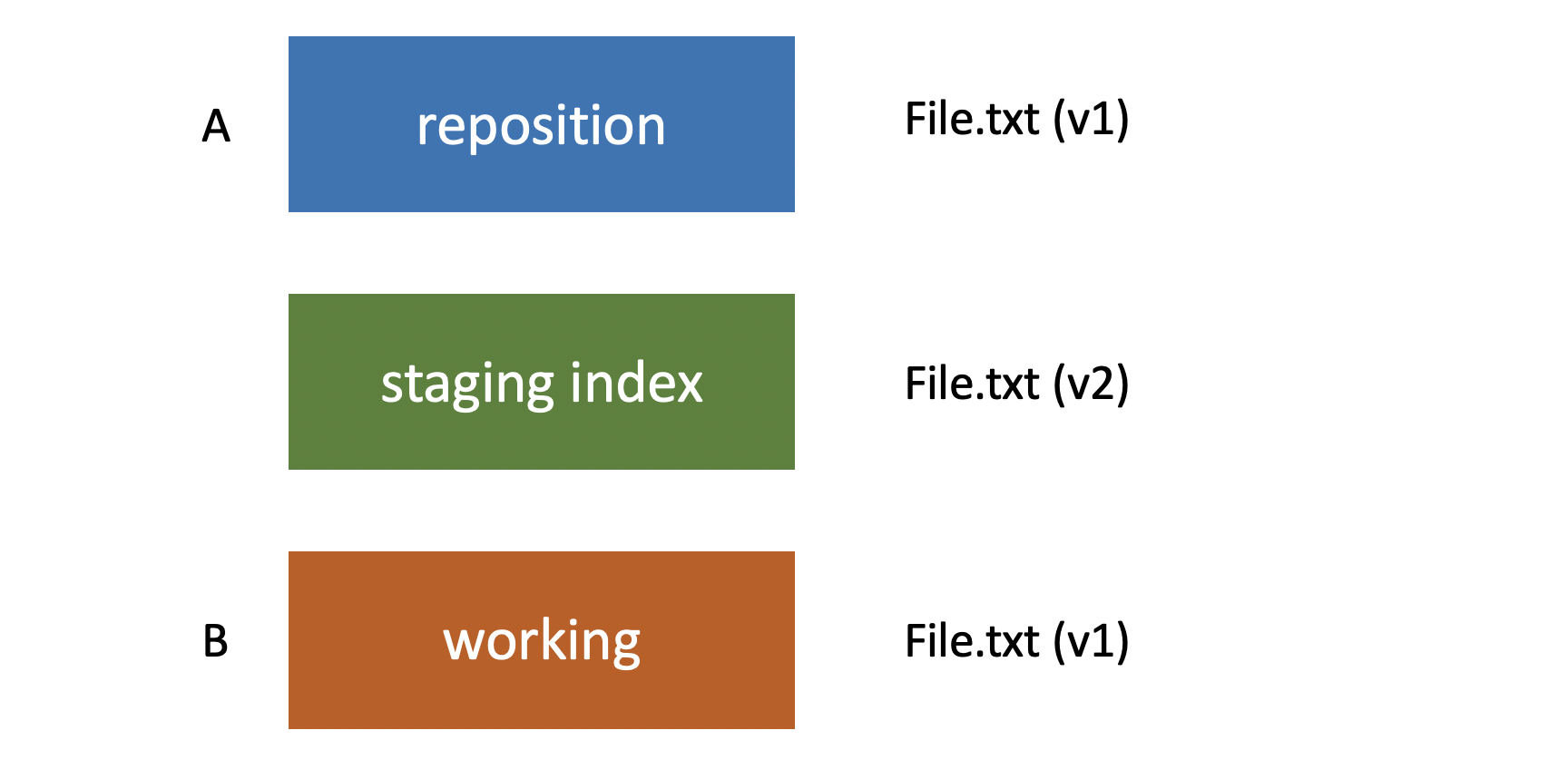
We have our three trees, the Git repository, the staging index, and our working directory. But to begin with let’s keep it simple, and let’s say that we have a file called file.txt that we’ve created in our working directory.



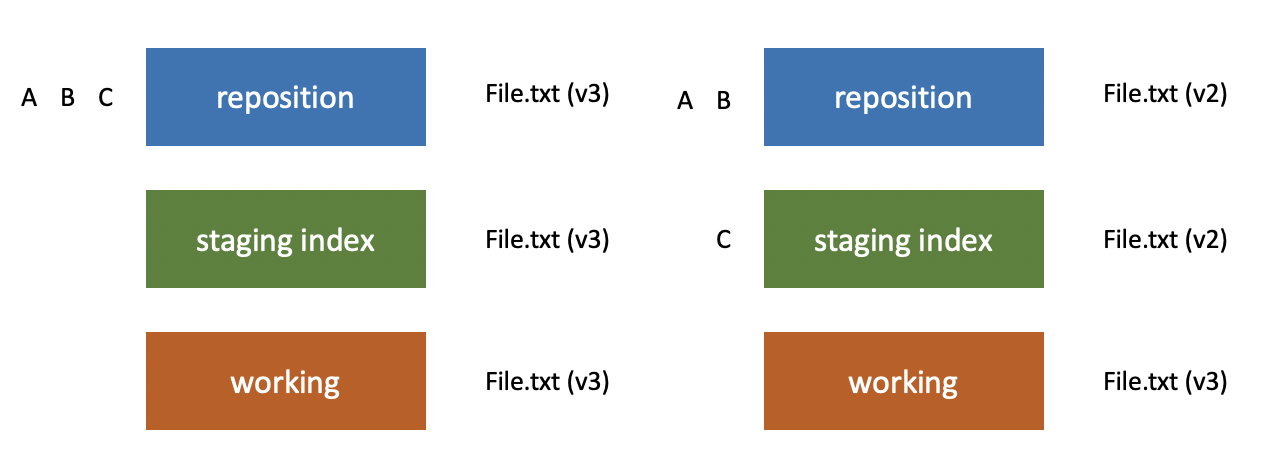
Let’s refer to this set of changes that we’ve made as A. So, it’s in the working directory right now, not in the staging index or the repository. After that, we use git commit in order to push that change set into the repository. Now the repository has the same file and it’s the same version as what’s in our staging index and our working directory.

Let’s imagine that we’re going to make changes to that file. So, we have version two of file.txt. We’re going to refer to this change set as being B.

Once Again, We Have It in Our Working Directory



And then the same thing would be true if we made the third set of changes called c. We would use git add to push those to the staging index and then git commit to push them into the repository. This is the typical workflow you’re going to be using to make commits. And then you could use git log in order to view those commits (A, B & C) and to see what changed between each one.



Now, of course, Git doesn’t refer to them as changes a, b and c. It has a unique ID number that applies to each one. In the next post, let’s talk more about that ID.

**Conclusion**

Three-tree architecture in Git: The working directory (containing changes that may not be tracked by Git), the staging index (containing changes that are about to be committed into the repository), and the repository (being tracked by Git).

When we finish working with our files in the project, commit them to the staging stage and then push them to the repository