Java Programming and Selenium Web Automation

Homework – Week 9 – Git Commands

# How to resolve GIT conflict?

**What is it?**

**Conflicting versions** of a file.

**How does it happen?**

When attempting to **merge different changes** made by **multiple people working on the same file**, Git is unable to automatically determine which changes should be kept and which should be discarded, leading to a conflict.

**How to resolve it?**

**Manually edit the file**: open the file and make necessary changes, i.e., decide which changes to **keep** and which to **discard**.

This can be done by **opening the file in a text editor** and looking for sections marked with **conflict markers** (<<<<<<<, =======, and >>>>>>>). Then **add** → **commit** → **push**

**Commands to resolve conflicts:**

* “**git log – merge**”: **list of commits** that are **causing conflicts**
* “**git diff** ”: identify the **difference** between **repositories** or **files**
* “**git checkout**”: **undo** **changes** made to the file, or for **changing** **branches**
* “**git reset -- mixed**”: **undo** **changes** to **working** **directory** and **staging** **area**
* “**git merge -- abort**”: **exiting** **merge** process and **returning back to the state** before merging
* “**git reset**”: **reset the conflicted files** to their original state. Used at time of merge conflict to

# Why are GIT branches needed?

**What are they?**

**Version of repository** that **diverges** from the **main**/**master** working project.

Diagram

Description automatically generated

**Why are they needed?**

Allows **developers to work on different parts** of the project **simultaneously** without affecting the main working branch to.

It enables:

* **Collaboration**: multiple people **work on different parts** of the project simultaneously **without interfering** in each other’s work. Once ready, it can be **merged** to the main branch.
* **Maintenance**: **bug** fixes, **security** patch.
* **Experimentation**: **new** **feature**/**idea** **exploration** without affecting the stability of the main branch.

# What is GIT trunk?

**Definition**: In Git, "trunk" is a term used to refer to the **main branch of a Git repository**, also known as the "**master**" **branch**. The trunk is typically the **default** and **primary** **branch** of the repository where all the main development work takes place.

Timeline

Description automatically generated

Developers will often create **new branches** **off** the trunk for **feature** **development**, **bug** **fixes**, or **experimentation**.

Once changes on a branch have been **tested** and **approved**, they can be **merged** back into the **trunk** to incorporate those changes into the main development line.

# Why is GIT better than SVN?

Git and SVN are both popular version control systems, but there are some key differences that make Git a better choice for many development teams:

* **Distributed architecture:**

Every developer has a **full copy** of the repository, including its entire **history**.

→ **work** **offline** and

→ **collaborate** efficiently

* **Fast performance:**

Git's design is optimized for **speed**, making it **faster** than SVN for most operations, including branching and merging.

* **Branching and merging:**

Git makes branching and merging easier and more flexible than SVN, allowing developers to **create** and **switch** between **branches** **quickly** and **easily**. Git's branching model also allows for more granular control over code changes and more efficient merging of changes between branches.

* **Better support for non-linear development:**

Git is better suited for non-linear development, where multiple branches are being worked on simultaneously. SVN's centralized architecture can lead to **conflicts** and **delays** when multiple developers are working on the same codebase.

* **Better support for large codebases:**

Git is better equipped to handle large codebases than SVN, thanks to its more **efficient** **storage** and **performance** **optimizations**.

Git is a more **modern** and **flexible** version control system than SVN, making it a better choice for most development teams. However, SVN may still be a better choice for some teams, particularly those with simpler codebases and workflows.

# What's difference between GIT fetch vs GIT pull?

|  |  |  |
| --- | --- | --- |
|  | **git fetch** | **git pull** |
| **Similarities** | Both **update local repository** with changes made to a remote repository | |
| **Differences** | **Downloads** **latest** **changes** to local repository but not merge with local branch.  **→** local branch **not yet in sync** with remote branch | Combination of “git fetch” followed by “git merge”: it **downloads and merges** the latest changes from remote repository with local branch.  **→** local branch **in sync** with remote branch |

# When does GIT commit fails?

**What causes it?**

When attempting to **push** local **changes** to the **remote** **repository** **without** **updating** your **local** **repository** with new changes already made to the remote repo.

**When does it occur?**

* **No changes** to commit.
* **Merge conflicts** between the changes in the **current branch** and in the **branch** we are trying **to** **merge**.
* **Permission** and **authentication issues.**
* **Commit message** is not properly **formatted.**
* **Repository** is in **bad state**: issues with **file structure** or **Git configuration.**

# How does GIT clone command work?

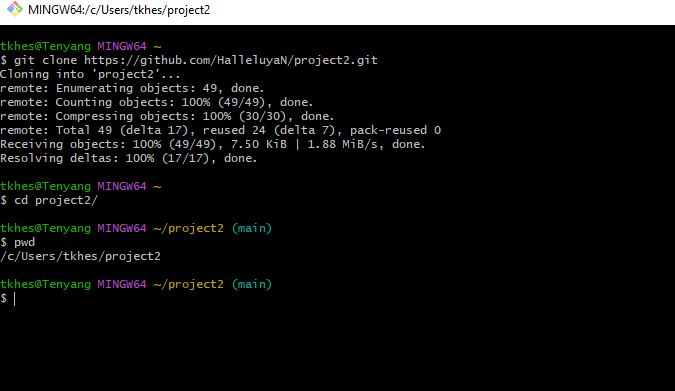
**What is it?**

Git clone allows you to **copy** and **contribute** to an existing **repository**. Cloning gives you **access** to all the **files**, including previous **versions** and data in that repository. Cloning makes the files of that repository **accessible** on your local machine and can **work** and **push** **offline**.

**Syntax:**

**git clone <repository\_url>**

**Example:**



# How can we check GIT update?

Git doesn’t have a specific “**update**” command. Rather, the process of updating a repository is **broken** into few **different** git **commands**.

* **“git add”** keeps the new file in a staging area, ready to be committed.
* **“git commit”** moves the file from the staging area to the local machine.
* **“git push”** takes the file from the local machine to the remote repository.
* **“git pull”** will allow you to pull the updated repository.

The following commands can be used in **checking any updates** in **Git**:

**git fetch**: downloads latest changes without merging to local branch

**git log origin/master**: shows changes that have been made in the remote repository’s main branch.

**git merge**: merges the changes made.

# What other tools/technology like GIT in the market?

**Other** popular **Version Control Systems** (**VCS**) are:

* **Subversion** (**SVN**): **Centralized VCS**

Developers **work** **simultaneously** on the **same codebase**.

Often used in **enterprise** **environments**.

More **traditional** compared to Git (ex: locking files to prevent concurrent modifications.

* **Mercurial** (**Hg**): **Distributed VCS**

Shares **similarities** with **Git**, including the ability to perform branching and merging.

**Ease** of use

**Scalability** → popular choice for larger projects.

* **BitBucket**: **Web-based VCS** by Atlassian

Provides both **Git** and **Mercurial** **repositories**.

Provides features such as **code** **review**, **issue** **tracking**, and **integration** with other **Atlassian** **tools**.

* **Perforce** (**P4**): **Centralized VCS**

**Commonly** used in **enterprise** **environments** (specifically in software development) due to its **scalability** and **security** features.

It provides features such as **branching** and **merging**, **access** **control**, and **file** **locking**.

* **SourceSafe**: **Legacy** **VCS** by Microsoft

**No longer actively** **developed** or **supported**, but it is still used by **some** organizations for managing older projects.

* **Microsoft Team Foundation Server** (**TFS**): **VCS** by Microsoft

**Tightly** integrated with **Microsoft's** development **tools**.

Often used in **enterprise** **environments**.

**Allows** for **continuous** **integration** and **deployment**.

**Code review**, **build** **automation**, and **integration** with other **Microsoft** **tools** such as Visual Studio.

* **Concurrent Versions System** (**CVS**): **Centralized** **CVS**

**Older** but is still used by some organizations for **managing** **legacy** **code**.

**Provides** **basic** **version** **control** **features** such as **branching** and **merging**.

* **Apache's Allura**: It is a **web-based VCS**

Supports **Git**, **SVN**, and **Mercurial**, among other tools.

# What are Git challenges?

Here are some **pros** and **cons** of Git:

**Pros**:

* **Collaboration**: It allows developers to **work** **together** on the **same** **codebase**, **share** **ideas**, and **track** **changes**.
* **Version control**: Developers can **track** the **history** of changes made to the codebase and **roll** **back** to previous versions if needed.
* **Open-source community**: Developers can **contribute** to open-source projects, **learn** from other developers' code, and get **involved** in the broader programming community.
* **Integration with other tools**: GitHub **integrates** with many other **tools** and **services**, such as **CI/CD** tools, **project management** tools, and **code review** tools. This makes it **easy** to **build** a complete **development** **workflow**.
* **Security**: GitHub provides several security features, such as **two-factor authentication** and **code scanning**, that help ensure the **safety** of the codebase.

**Cons**:

* **Learning curve:** **Many features** that take time to learn for **beginners**.
* **Cost:** While GitHub offers **free hosting** for **public** repositories, **private** repositories require a **subscription**. This can be a significant expense for small development teams or individual developers.
* **Limited control over server:** GitHub is a **cloud-based service**, which means developers have **limited control** over the underlying server infrastructure.
* **Dependency on internet:** GitHub requires an **internet** **connection** to access and use the platform. This can be a disadvantage if you don't have a stable internet connection.
* **Limitations for large files:** GitHub has a file size limit of **100 MB** per file. This can be a disadvantage for large media files, such as **video** or **audio** **files**.

Overall, GitHub is an **excellent platform** for **collaborating** on coding **projects**, but it's not without its **limitations**. It's important to weigh the pros and cons before deciding whether to use GitHub for your project.

# What are GIT best practices?

Git **best practices** include:

* Use **meaningful** **commit** **messages**: **Clear** and **concise** **explaining** changes.
* **Commit frequently**: Committing **small** and **frequent** **changes** → easier to track and revert changes.
* Use **branches**: Work on **different features** or **bug** **fixes** without interfering with the main codebase.
* Keep the **main branch clean**: **Always** contain **working code** that can be **deployed** at any time.
* **Review** **code changes**: Ensures changes are **high quality** and do not introduce any issues.
* Use **pull requests**: To **initiate** code **reviews** and **track** changes.
* Keep your **repository small**: **Avoid** adding **unnecessary** **files** to your repository, as this can **slow down** cloning and branching.
* Use “**.gitignore”**: **Excludes** files (**build** and **configuration**) from being committed
* **Backup regularly**: **Avoid** **losing** **work** in case of a hardware failure or other issue.
* **Stay up to date**: Keep your **Git version** and **tools** up to date to take advantage of **new features** and **bug fixes**.

# How often team should do merge to master?

The **frequency of merging** to the master branch can vary **depending** on the **project** and **team**.

However, a common **best practice** is to merge to the master branch **frequently**, ideally after completing a **small** **unit** of work or a **feature**.

→ **continuous** **integration**

→ **helps prevent** **conflicts** and **issues** from merging **large** and **complex** changes at once.

In **agile** **development** methodologies, merging occurs at the **end** of each **sprint** or **development** **cycle**.

# How GIT helps in CI/CD pipeline?

Git plays an **essential** **role** in **Continuous Integration**/**Continuous Deployment** (**CI**/**CD**) pipeline by providing **version** **control**, **collaboration**, and **deployment** **automation** capabilities.

* **Version Control**: **track** and **manage** changes:

→ **work** **concurrently**

→ **code** **quality**

→ **improve** **productivity**

* **Collaboration**: code **sharing**, **review** changes, **collaborate**
* **Continuous Integration**: **automate** process of **building**, **testing** and **merging** code into shared repository.

→ **up to date** code

→ **reduce** **risk** of conflicts/bugs

* **Continuous Deployment**: **automate** process of **deploying** code into various environments (staging, testing and production)

→ **seamlessly**

→ **quickly**