**GIT ---🡪 GITHUB**

* Git is a open source platform and it is mainly called as a version control system vcm tool

**1**. **Explain the Git workflow you typically use in your projects. Walk me through the steps from feature development to deployment.**

* **Answer with Example:** "In most of my projects, I follow a simplified version of Gitflow or GitHub Flow. It usually starts with a main or production branch that always reflects the stable, deployable code. For new features or bug fixes, I create a new branch off main, typically named feature/new-login-page or bugfix/issue-42.
  + **Development:** I work on this feature branch, committing my changes regularly with descriptive messages like feat: Implement user login functionality or fix: Resolve issue with incorrect button behavior.
  + **Collaboration:** Once the feature is complete and I've performed local testing, I push my branch to the remote repository (git push origin feature/new-login-page).
  + **Review:** I then create a Pull Request (PR) on GitHub targeting the main branch. This allows team members to review my code, provide feedback, and discuss any potential issues. For example, a reviewer might comment on a specific line of code: 'Consider adding input validation here to prevent errors.'
  + **Merge:** After the code review is approved, the PR is merged into the main branch. This can be done via a standard merge or a squash merge to create a cleaner commit history on main.
  + **Deployment:** Finally, the changes on the main branch are deployed to the target environment, often triggered automatically by a CI/CD pipeline that listens for changes on main. For instance, a Jenkins job or a GitHub Actions workflow might build the application, run tests, and then deploy it to a staging or production server."

**2.** **What happens when you run the command git init? What are the key files and directories created, and what is their purpose?**

* **Answer with Example:** "When you run git init in a directory, Git initializes an empty Git repository. It creates a hidden .git directory at the root of your project. This directory contains all the metadata and object database that Git uses to manage your project's history and configuration.
  + **HEAD:** This file points to the currently checked-out commit or branch. For a new repository, it usually points to the main branch (or master in older Git versions). You can see its content with cat .git/HEAD, which might show ref: refs/heads/main.
  + **config:** This file stores the project-specific configuration options, such as your username and email, remote repository URLs, and other settings. You can view and edit this file directly or using git config.
  + **description:** This file is used by some Git tools to provide a short description of the repository.
  + **hooks:** This directory contains script files that Git can execute at various points in its workflow, such as before a commit or after a push. Examples include pre-commit to run linters or tests before allowing a commit, and post-receive on a remote server to trigger deployments.
  + **info:** This directory contains auxiliary information for Git. The exclude file inside it is similar to .gitignore but is not committed to the repository and only affects your local setup.
  + **objects:** This is the core of Git's storage. It's where Git saves all versions of your files (blobs), commit metadata (commits), tree structures representing directories (trees), and tags. These are stored as content-addressable objects, meaning they are identified by the SHA-1 hash of their content."

**3.** **Describe the difference between git clone, git remote add, and git submodule add. When would you use each?**

* **Answer with Example:**
  + **git clone <repository\_url>:** This command is used to create a local copy of a remote repository. It downloads all the project's files, history, and branches.
    - **Example:** git clone https://github.com/octocat/Spoon-Knife.git will create a directory named Spoon-Knife containing a copy of the repository from that URL, along with the .git directory configured to track the origin remote.
  + **git remote add <name> <repository\_url>:** This command is used to create a new connection to a remote repository. You give the remote a name (conventionally origin for the main remote). This allows you to fetch from and push to that remote.
    - **Example:** If you have a local Git repository and want to connect it to a remote repository on GitHub, you would navigate to your local repository directory and run: git remote add origin git@github.com:your-username/your-repo.git. You can then use git push origin main to push your local main branch to the origin remote.
  + **git submodule add <repository\_url> <path>:** This command is used to add an external repository as a subdirectory within your main repository. This is useful when your project depends on a specific version of another project.
    - **Example:** If your project needs a specific library located at https://github.com/example/external-lib.git in a subdirectory named libs/external-lib, you would run: git submodule add https://github.com/example/external-lib.git libs/external-lib. This creates a .gitmodules file and adds the external repository at the specified path. You then need to initialize and update the submodule with git submodule init and git submodule update to fetch its contents."

**4.** **Explain the concepts of staging area (index) and working directory in Git. How do they relate to each other?**

* **Answer with Example:** "The **working directory** is the set of files and directories you see in your file system. It's where you make changes to your project's files. Git tracks changes made in the working directory but doesn't consider them part of the repository history until you explicitly tell it to.
* The **staging area** (also known as the index or cache) is a temporary area where you prepare changes from your working directory to be included in the next commit. Think of it as a buffer between your working directory and the Git repository.
* **Relationship:** When you make changes to files in your working directory, Git is aware of these modifications. However, these changes are not yet part of a commit. To include these changes in your next commit, you need to use the git add command. This command moves the changes from your working directory to the staging area.
  + **Example:**
    1. You modify a file named README.md in your working directory.
    2. You run git status, and it will show README.md under "Changes not staged for commit".
    3. You then run git add README.md. This stages the changes.
    4. Running git status again will now show README.md under "Changes to be committed".
    5. Finally, you run git commit -m "Update README" to create a new commit containing the changes that were in the staging area. The working directory can then be further modified for the next set of changes."

**5.** **What are Git branches, and why are they important in a collaborative development environment? Describe a branching strategy you've used (e.g., Gitflow, GitHub Flow).**

* **Answer with Example:** "Git branches are lightweight, movable pointers to a specific commit in the repository's history. They allow you to diverge from the main line of development and work on isolated features, bug fixes, or experiments without affecting the stability of the main codebase.
* **Importance in Collaboration:** Branches are crucial for collaboration because they enable multiple developers to work on different parts of the project simultaneously without stepping on each other's toes. Each developer can work on their own branch, make commits, and then merge their changes back into the main branch (or another integration branch) when their work is complete and reviewed. This prevents conflicts and keeps the main codebase stable.
* **GitHub Flow Example:** I've often used GitHub Flow, which is a simpler branching strategy suitable for continuous deployment.
  1. **main branch:** This is the primary branch, always deployable.
  2. **Create branches for every feature or fix:** When starting work on something new, I create a descriptive branch off main (e.g., feature/user-profile).
  3. **Commit to your feature branch locally and regularly.**
  4. **Push your local branch to the remote repository.**
  5. **Open a Pull Request (PR) on GitHub:** This initiates code review and discussion.
  6. **Discuss and review the code:** Team members provide feedback on the PR.
  7. **Merge the branch into main:** Once the review is satisfactory and any issues are resolved, the feature branch is merged back into main.
  8. **Deploy main:** Changes on main are then deployed to production.
  9. **Delete the feature branch:** Once merged and deployed, the feature branch can be deleted to keep the repository clean."

**6.** **How do you merge branches in Git? Explain the difference between git merge and git rebase. What are the potential advantages and disadvantages of each? When would you choose one over the other?**

* **Answer with Example:** "In Git, you can integrate changes from one branch into another using either git merge or git rebase.
  + **git merge:** This command takes the changes from a source branch and integrates them into the target branch by creating a new "merge commit". This commit has two parent commits, representing the heads of the two branches being merged.
    - **Example:** If you are on the main branch and want to merge changes from the feature/new-api branch, you would run: git merge feature/new-api. This will create a new commit on main that combines the histories of both branches. The commit history will show the divergence and the subsequent merge.
    - **Advantages:** Preserves the full history of both branches, making it easier to track when and why changes were introduced. It's also generally simpler to understand for beginners.
    - **Disadvantages:** Can lead to a more cluttered commit history with many merge commits, especially in active repositories with frequent merges.
  + **git rebase:** This command takes the commits from the source branch and reapplies them on top of the target branch. It essentially rewrites the history of the source branch as if it was branched off the target branch more recently.
    - **Example:** If you are on the feature/new-api branch, which branched off main some time ago, and main has moved ahead with new commits, you can run git rebase main. This will take your commits on feature/new-api and replay them on top of the current state of main, resulting in a linear commit history.
    - **Advantages:** Creates a cleaner, linear commit history, which can be easier to follow. It avoids the "fork in the road" appearance of merges.
    - **Disadvantages:** Rewrites history, which can be problematic if the rebased branch has already been pushed to a shared remote repository, as it can cause issues for other collaborators. Resolving conflicts during a rebase can also be more complex as you're replaying commits one by one.
  + **When to choose:** Use **git merge** when you want to preserve the exact history of when changes were integrated and when working on long-lived feature branches that have already been shared.
    - Use **git rebase** on local feature branches that haven't been shared yet to create a cleaner history before merging into a main branch. It's often preferred for integrating short-lived feature branches into main (as in GitHub Flow) to keep the main branch history linear."

**7.** **What are merge conflicts, and how do you typically resolve them? Describe a situation where you encountered a complex merge conflict.**

* **Answer with Example:** "Merge conflicts occur when Git cannot automatically integrate changes from two or more branches because the same lines in the same file have been modified differently. Git marks the conflicting sections in the affected file.
* **Typical Resolution:**
  1. **Identify the conflicting files:** After attempting a merge or rebase, Git will indicate which files have conflicts.
  2. **Open the conflicting files:** Examine the files in a text editor or an IDE with merge conflict resolution tools. Git will insert special markers (<<<<<<<, =======, >>>>>>>) to delineate the conflicting changes from different branches.
  3. **Manually edit the files:** Decide which changes to keep. You might choose changes from one branch, combine changes from both, or write completely new code. Remove the Git conflict markers after resolving the conflict.
  4. **Stage the resolved files:** Once you've edited the files to your satisfaction, use git add <conflicted\_file> to mark them as resolved in the staging area.
  5. **Complete the merge/rebase:**
     + For a merge, use git commit to create a new merge commit with your resolved changes. You can often edit the default commit message provided by Git.
     + For a rebase, use git rebase --continue to move to the next commit in the rebase sequence. If you encounter more conflicts, repeat the resolution process. If you want to skip a commit, use git rebase --skip. If you want to abort the rebase entirely, use git rebase --abort.
* **Complex Merge Conflict Example:** "I once worked on a project where two developers were working on different parts of the same large configuration file (application.yml) for several days. One developer added new database connection settings, while the other refactored the logging configuration and also modified some shared properties. When we tried to merge their branches into the develop branch, we encountered a significant conflict in this file. The conflict wasn't just on a few lines; it involved multiple overlapping sections with different indentation and structural changes. To resolve this, we had to:
  1. Open the application.yml file and carefully examine all the conflicting blocks.
  2. Communicate extensively with the other developer to understand the intent behind their changes.
  3. Manually reconstruct the file, ensuring that both the new database settings and the refactored logging configuration were correctly integrated, and that the shared properties were consistent. This involved not just picking one version over the other but carefully merging the logic and syntax.
  4. We used an IDE with a three-way merge tool, which helped visualize the changes from the base commit and the two branches.
  5. After resolving all the conflicts, we tested the application thoroughly in a local environment to ensure that both the database connections and logging were working as expected before finally committing the merged changes."

**8.** **Explain the purpose of git commit. What are best practices for writing commit messages?**

* **Answer with Example:** "The git commit command is used to record snapshots of your staged changes in the repository's history. Each commit represents a specific point in time and includes metadata like the author, committer, date, and a message describing the changes.
* **Best Practices for Commit Messages:**
  + **Separate subject from body with a blank line:** The first line should be a concise summary of the changes (50 characters or less). The body can provide more detailed context.
  + **Capitalize the subject line:** Start the subject line with a capital letter.
  + **Do not end the subject line with a period:** Keep it brief and to the point.
  + **Use the imperative, present tense in the subject line:** Instead of "Fixed bug" or "Fixing bug," use "Fix bug." Examples: "Add new user authentication method," "Refactor payment processing logic," "Fix typo in documentation."
  + **Use the body to explain the *what* and *why* of the change, not just the *how*:** Provide context and reasoning behind the changes, especially if they are not immediately obvious from the code itself.
  + **Wrap the body at 72 characters:** This improves readability on various Git tools.
  + **Consider using conventional commit formats:** Projects might adopt a specific format like Angular's commit message conventions (e.g., feat(auth): Implement user login, fix(user): Resolve issue with profile update). This can be useful for automated changelog generation and other tooling.
* **Example of a good commit message:**
* feat(api): Implement endpoint for retrieving user details
* This commit introduces a new API endpoint at /users/{id} that allows
* clients to retrieve detailed information about a specific user.
* The implementation includes:
* - Creation of the UserDetailsController.
* - Implementation of the getUserDetails method.
* - Unit tests to verify the endpoint's functionality and error handling.
* This feature is required for the new user profile page in the frontend.
* ```"

**9.** **How can you view the history of changes in a Git repository? Describe different git log options you find useful.**

* **Answer with Example:** "The primary command to view the history of changes in a Git repository is git log. It displays a list of commits in reverse chronological order. However, there are many useful options to customize the output:
  + **git log --oneline:** Shows each commit on a single line, displaying the abbreviated commit hash and the first line of the commit message. This provides a concise overview.
    - **Example Output:** a1b2c3d Initial commit, e5f6g7h Fix: Resolved issue #10, i8j9k0l feat: Implement user authentication
  + **git log --graph --oneline --decorate --all:** This is one of my go-to commands for visualizing the branch and merge history.
    - --graph: Displays an ASCII graph of the branch and merge history.
    - --oneline: Shows each commit on a single line.

**Core Git Concepts (Continued):**

* --decorate: Shows references (branches, tags, HEAD) that point to each commit.
  + --all: Shows commits from all branches.
  + **Example Output:**
  + \* e5f6g7h (HEAD -> main, origin/main) Fix: Resolved issue #10
  + |\
  + | \* i8j9k0l (feature/auth) feat: Implement user authentication
  + |/
  + \* a1b2c3d Initial commit
* **git log -p or git log --patch:** Shows the textual diff for each commit, displaying the changes made in that commit. This is useful for understanding the specific code modifications.
  + **Example Output (partial):**

Diff

--- a/README.md

+++ b/README.md

@@ -1,4 +1,4 @@

# My Project

This is a sample project.

-It is very simple.

+It is incredibly simple and awesome.

* **git log --stat:** Shows a brief statistics for each commit, indicating how many files were changed and the number of insertions and deletions.
  + **Example Output:**
  + commit e5f6g7h Fix: Resolved issue #10
  + 1 file changed, 1 insertion(+), 1 deletion(-)
  + commit i8j9k0l feat: Implement user authentication
  + 3 files changed, 50 insertions(+), 10 deletions(-)
* **git log --author="John Doe":** Filters the commit history to show only commits made by the specified author.
* **git log --since="2 weeks ago" or git log --until="yesterday":** Filters commits based on a date range. You can use various formats for the date.
* **git log <file\_path>:** Shows the commit history for a specific file.
  + **Example:** git log src/app.js will show all commits that modified the src/app.js file.
* **git log --follow <file\_path>:** Shows the commit history of a file, even if it was renamed.
* **git log --grep="refactor":** Filters commits based on a pattern in the commit message.

1. **What is the purpose of Git tags? How do you create and manage them?**
   * **Answer with Example:** "Git tags are used to mark specific points in the repository's history as important, typically for release versions. They are like bookmarks that are meant to be permanent and easily referenceable.
   * **Creating Tags:**
     + **Lightweight Tags:** These are simple pointers to a specific commit and don't contain any extra metadata. You create them using: git tag v1.0.0.
     + **Annotated Tags:** These are more robust and store extra information such as the tagger's name, email, date, and a tagging message. It's generally recommended to use annotated tags for releases. You create them using the -a flag followed by the tag name, -m for the message: git tag -a v1.0.1 -m "Stable release v1.0.1 with critical bug fixes.".
   * **Managing Tags:**
     + **Listing Tags:** You can list all tags in your repository using: git tag. You can also filter tags using patterns, e.g., git tag -l "v1.\*" to list tags starting with "v1.".
     + **Showing Tag Information:** To see details about an annotated tag (including the message), use: git show v1.0.1. For a lightweight tag, it will just show the commit it points to.
     + **Tagging a Previous Commit:** If you forgot to tag a release, you can tag a previous commit by specifying its commit hash: git tag -a v0.9.0 <commit\_hash> -m "Previous release.".
     + **Pushing Tags:** Tags are not automatically pushed when you run git push. You need to explicitly push tags using:
       - git push origin <tag\_name> to push a specific tag.
       - git push origin --tags to push all local tags to the remote repository.
     + **Deleting Tags:**
       - **Locally:** git tag -d <tag\_name>
       - **Remotely:** git push origin --delete <tag\_name>
   * **Example Scenario:** "When we release a new version of our application, say version 2.0.0, we create an annotated tag on the commit that represents the final state of that release. We use the command git tag -a v2.0.0 -m "Major release v2.0.0 with new features and improvements.". This tag helps us easily identify and refer back to this specific version in the future, for instance, when we need to investigate a bug report related to that release."

**Advanced Git Concepts and Troubleshooting (Continued):**

1. **How would you revert a commit that has already been pushed to a remote repository? What are the potential implications of doing so?**
   * **Answer with Example:** "Reverting a commit that has already been pushed to a shared remote repository requires careful consideration because it affects the history that others have already based their work on. There are two main approaches:
     + **git revert <commit\_hash>:** This is the generally recommended approach for public or shared repositories. It creates a new commit that undoes the changes introduced by the specified commit. This new commit is then added to the history.
       - **Example:** If you want to revert the commit with the hash abcdef12345, you would run git revert abcdef12345. Git will usually open an editor with a default commit message explaining the reversion. After saving and closing, a new commit will be created. You then need to git push origin main (or your relevant branch) to share this reversion with the remote repository.
       - **Implications:** This method preserves the original commit in the history, which can be useful for auditing. However, it adds a new commit, so the history grows.
     + **git reset --hard <parent\_of\_bad\_commit> followed by git push --force origin main:** This approach rewrites history by moving the branch pointer back to the commit before the bad one. **This is highly discouraged on shared branches** because it can cause significant problems for other collaborators who have already pulled the bad commit. Their local repositories will diverge, and they might have to perform complex rebases or resets to reconcile.
       - **Example (Use with Extreme Caution):** If abcdef12345 is the bad commit, you would first find its parent commit hash (e.g., uvwxyz7890). Then you would run git reset --hard uvwxyz7890 followed by git push --force origin main.
       - **Implications:** This creates a cleaner history locally, but it can lead to data loss and severe synchronization issues for collaborators. **Only use --force if you are absolutely sure you understand the consequences and are working on a private branch.**
   * **Best Practice:** In most collaborative scenarios, using git revert is the safer and preferred method for undoing changes that have been pushed."
2. **Explain the concept of git cherry-pick. When might you use it? What are some potential pitfalls?**
   * **Answer with Example:** "git cherry-pick allows you to select specific commits from one branch and apply them onto your current branch. It's like picking a cherry (a commit) from one tree (branch) and planting it in your current tree.
   * **When to Use:**
     + **Applying a specific bug fix to a stable branch:** If a bug fix is committed to a development branch, you might want to cherry-pick that specific commit onto a production or maintenance branch without merging the entire development branch.
       - **Example:** A critical security fix is implemented in the develop branch in commit fedcba98765. To apply this fix to the release-1.0 branch without merging other ongoing development work, you would switch to the release-1.0 branch (git checkout release-1.0) and then run git cherry-pick fedcba98765.
     + **Undoing a commit that was accidentally merged:** If a feature branch was prematurely merged, you could cherry-pick the commit *before* the merge onto the main branch to effectively undo the merge in a more targeted way than a full revert.
     + **Porting specific features between long-lived branches:** In rare cases, you might want to bring a particular feature from one long-lived branch to another without a full merge.
   * **Potential Pitfalls:**
     + **Duplicate Commits:** Cherry-picking creates a new commit with the same changes but a different commit hash and author date. This can lead to a confusing history if overused.
     + **Contextual Issues:** A cherry-picked commit might rely on other changes that are not present in the target branch, leading to build failures or unexpected behavior. You might need to manually resolve conflicts or even cherry-pick related commits.
     + **Loss of History:** Cherry-picking doesn't bring over the full branch history, which can make it harder to understand the evolution of the changes.
     + **Merge Conflicts:** Cherry-picking can still result in merge conflicts if the changes in the picked commit overlap with changes in the target branch."
3. **How can you find a specific commit that introduced a bug? Describe the git bisect command and how it can be used.**
   * **Answer with Example:** "Finding the commit that introduced a bug can be challenging in a large codebase with a long history. git bisect is a powerful Git tool that helps automate this process using a binary search algorithm. You tell Git a "bad" commit (the one where the bug is present) and a "good" commit (a known commit before the bug was introduced), and Git will repeatedly check out commits in between, asking you to mark each one as either "good" or "bad" until it pinpoints the exact commit that introduced the change.
   * **How to Use git bisect:**
     + **Start bisecting:** Navigate to your Git repository in the terminal and run git bisect start.
     + **Mark a known bad commit:** Identify a commit where the bug is present (usually the current HEAD) and tell Git: git bisect bad.
     + **Mark a known good commit:** Identify an older commit where you are sure the bug was not present (e.g., a previous release tag) and tell Git: git bisect good <good\_commit\_hash>.
     + **Git checks out a commit in the middle:** Git will automatically check out a commit halfway between the good and bad commits.
     + **Test the checked-out commit:** You need to build and test your project at this commit to see if the bug is present.
     + **Mark the commit as good or bad:**
       - If the bug is present in the checked-out commit, run git bisect bad.
       - If the bug is not present, run git bisect good.
     + **Repeat steps 4-6:** Git will continue to narrow down the range of commits by checking out another commit in the remaining interval. You keep testing and marking commits until Git identifies the first "bad" commit – the one that introduced the bug.
     + **Reset bisect:** Once the culprit commit is found, run git bisect reset to return your repository to the branch you were on before starting the bisect.
   * **Example Scenario:** "Our QA team reported a new bug in the latest release (tagged v2.0.0). We suspected it was introduced sometime after the previous stable release (v1.5.0). To find the problematic commit, we would:

Bash

git bisect start

git bisect bad v2.0.0

git bisect good v1.5.0

Git would then check out a commit between v1.5.0 and v2.0.0. We would build and test that version. If the bug was present, we'd run git bisect bad. If not, we'd run git bisect good. We'd repeat this process. Git might output something like:

Bisecting: 7 revisions left to test after this (roughly 3 steps)

[abcdefg] Fix: Updated dependency version

We would test this commit. Let's say the bug is still present, so we'd do git bisect bad abcdefg. Git would continue until it eventually outputs the specific commit that introduced the bug, along with its commit message."

1. **What is the difference between a hard reset, a soft reset, and a mixed reset in Git? What are the use cases for each?**
   * **Answer with Example:** "The git reset command is used to move the current branch pointer to a specified commit and optionally modify the staging area and working directory. The three types of resets differ in how they affect these three areas:
     + **git reset --soft <commit>:**
       - Moves the branch pointer (HEAD) to the specified <commit>.
       - **Does not** change the staging area or the working directory. All changes from the commits after the target commit are kept in the staging area.
       - **Use Case:** Useful when you've made several commits that you want to combine into a single commit or when you want to undo the last few commits but keep the changes to restage and modify.
       - **Example:** If you made three commits you want to squash, you could git reset --soft HEAD~3 to move the HEAD back three commits, and all those changes will be in the staging area, ready to be committed again with a new, consolidated message.
     + **git reset --mixed <commit> (This is the default if you just use git reset <commit>):**
       - Moves the branch pointer (HEAD) to the specified <commit>.
       - **Resets the staging area** to match the specified <commit>. The changes from the commits after the target commit are moved to the working directory as unstaged changes.
       - **Does not** change the working directory (the files themselves remain as they were).
       - **Use Case:** Useful when you want to undo some commits and keep the changes in your working directory to potentially modify or discard them selectively before restaging.
       - **Example:** If you committed some files by mistake, you could git reset HEAD~1. This would remove the last commit, and the changes would be in your working directory, unstaged, allowing you to modify or git add them again correctly.
     + **git reset --hard <commit>:**
       - Moves the branch pointer (HEAD) to the specified <commit>.
       - **Resets the staging area** to match the specified <commit>.
       - **Resets the working directory** to match the specified <commit>. **Any uncommitted changes in the working directory are lost!**
       - **Use Case:** Useful when you want to completely discard the last few commits and the changes they introduced, effectively reverting your local repository to a previous state. **Use with extreme caution, especially on shared branches, as it can lead to data loss.**
       - **Example:** If you made a series of experimental commits that you want to completely get rid of, you could use git reset --hard <commit\_before\_experiments>.
2. **How can you ignore specific files or patterns of files from being tracked by Git? Explain the purpose and syntax of .gitignore files.**
   * **Answer with Example:** "The .gitignore file is a plain text file in the root directory of your Git repository (or in subdirectories to apply rules to specific parts of the project). Its purpose is to tell Git which files and directories it should intentionally ignore and not track. This is useful for excluding temporary files, build outputs, configuration files containing sensitive information, IDE-specific files, and other files that are not part of the actual project codebase.
   * **Syntax of .gitignore:**
     + Each line in a .gitignore file specifies a pattern to match against file paths relative to the location of the .gitignore file.
     + Blank lines are ignored.
     + Lines starting with # serve as comments.
     + Standard glob patterns are used:
       - \*: Matches zero or more characters. Example: \*.log ignores all files ending with .log.
       - ?: Matches exactly one character. Example: temp?.txt ignores temp1.txt, tempA.txt, etc.
       - []: Matches any single character within the brackets. Example: [abc].txt ignores a.txt, b.txt, and c.txt. You can also use ranges like [0-9].txt.
       - [^...] or [!...]: Matches any single character *not* within the brackets. Example: [^a-z].txt ignores any .txt file where the first character is not a lowercase letter.
       - /: Used to specify pathnames relative to the .gitignore file. A leading / anchors the pattern to the repository root (if in the top-level .gitignore) or the current .gitignore directory. Example: /build ignores the build directory at the root. subdir/build ignores the build directory within the subdir directory.
       - A trailing / on a pattern indicates it should only match directories. Example: build/ ignores the build directory and its contents, but not a file named build.
       - A pattern starting with ! negates the pattern that precedes it. This can be used to re-include a file that was previously excluded by a broader pattern. Example: