Got it! I'll break down each of the topics you mentioned in terms of **What**, **Why**, and **How**, along with simple examples where possible. Let's go step by step:

**1. Introduction to SCM and Git**

**What:**  
SCM stands for **Source Code Management**. It's a system that helps manage and track changes to code over time. Git is a popular SCM tool used to keep track of changes in software development.

**Why:**  
Git helps developers work on different parts of the code simultaneously without interfering with each other's work. It allows version control, meaning you can keep track of every change made to the code.

**How:**  
Git tracks changes using commits, which record snapshots of your code. For example, when you update a file, you can commit the change and Git stores that version.

**2. Installation of Git**

**What:**  
Installing Git allows you to use the tool to manage source code versions.

**Why:**  
Without Git, it would be difficult to manage multiple versions of your project, collaborate with others, and track code changes effectively.

**How:**  
On **RHEL Linux** (Red Hat Enterprise Linux), you can install Git using the following command:

sudo yum install git

After installation, you can check the version with:

git --version

**3. Git File Systems**

**What:**  
Git stores files and their history in a special file system, divided into three areas: **Working Directory**, **Staging Area**, and **Repository**.

**Why:**  
This system helps you organize files at different stages of their lifecycle, like changes that are being worked on, changes ready to be committed, and the final committed code.

**How:**

* **Working Directory**: Where you make changes to files.
* **Staging Area**: Files here are marked to be committed.
* **Repository**: Stores the committed changes and full history.

**4. Creating a Local Repository**

**What:**  
A local repository is where Git stores all versions of your project on your local machine.

**Why:**  
Creating a local repository lets you track and manage changes before pushing to a remote server like GitHub.

**How:**  
To create a new repository, navigate to your project folder and run:

git init

**5. Basic Git Configurations**

**What:**  
Basic Git configurations include setting your name, email, and editor.

**Why:**  
These settings help identify who made each commit and how Git behaves on your machine.

**How:**  
Set your name and email with these commands:

git config --global user.name "Your Name"

git config --global user.email "your.email@example.com"

**6. Adding Files to a Project**

**What:**  
Adding files to a project means putting them in the **staging area** to be committed.

**Why:**  
Before you commit changes, you need to tell Git which files to include.

**How:**  
Use:

git add <filename>

For all files:

git add .

**7. Status of the Project**

**What:**  
The git status command shows the current state of your working directory and staging area.

**Why:**  
It helps you see which files are changed, which are staged, and which are untracked.

**How:**  
Run:

git status

**8. Committing to Git**

**What:**  
Committing is how you save changes in Git. Each commit has a message describing the changes.

**Why:**  
Commits allow you to save the history of your project so you can revert back if needed.

**How:**  
To commit changes, use:

git commit -m "Your commit message"

**9. Ignoring Specific File Types**

**What:**  
You can use .gitignore files to tell Git to ignore certain files (e.g., log files or temporary files).

**Why:**  
You don't want to track files that are not important to your project, like compiled code or temporary files.

**How:**  
Create a .gitignore file in your project and list files or patterns to ignore:

\*.log

\*.tmp

**10. Tags**

**What:**  
Tags are used to mark specific points in history, like version releases.

**Why:**  
Tags help in identifying significant versions, making it easy to reference or release a version.

**How:**  
To create a tag:

git tag v1.0

**11. Branching**

**What:**  
Branching allows you to work on different versions of a project in parallel without affecting the main codebase.

**Why:**  
Branches help developers work on new features or fixes independently.

**How:**  
To create a new branch:

git branch new-feature

To switch to the branch:

git checkout new-feature

**12. Merging**

**What:**  
Merging brings changes from one branch into another.

**Why:**  
It allows you to integrate work from different branches into the main project.

**How:**  
To merge a branch into the current branch:

git merge new-feature

**13. Reverting**

**What:**  
Reverting undoes a commit and creates a new commit to reverse the changes.

**Why:**  
Reverting is safer than deleting commits because it preserves the commit history.

**How:**  
To revert a commit:

git revert <commit-id>

**14. Using diff Command**

**What:**  
The git diff command shows differences between files or commits.

**Why:**  
It allows you to see what exactly has changed before committing.

**How:**  
To see changes in your working directory:

git diff

**15. Garbage Collection**

**What:**  
Git's garbage collection removes unnecessary data to optimize the repository.

**Why:**  
It helps keep your repository small and efficient.

**How:**  
Run:

git gc

git gc –aggressive Runs a more thorough optimization.

**16. Git Logging and Auditing**

**What:**  
Git logs show the history of commits, which helps in auditing the project's progress.

**Why:**  
It helps you track changes and understand who made them.

**How:**  
Use:

git log

**17. Cloning Repositories**

**What:**  
Cloning creates a copy of a repository on your machine.

**Why:**  
It allows you to start working on an existing project.

**How:**  
To clone a repository:

git clone <repository-url>

**18. Forking**

**What:**  
Forking is copying a repository from someone else's GitHub account to your own.

**Why:**  
It allows you to make changes without affecting the original project.

**How:**  
On GitHub, click the **Fork** button on a repository page.

**19. Securing GitHub Account with SSH Keys**

**What:**  
SSH keys provide a secure way to connect to GitHub without using your password.

**Why:**  
It is more secure than using a password and avoids entering credentials every time.

**How:**  
Generate an SSH key:

ssh-keygen -t rsa -b 4096 -C "your\_email@example.com"

Then add the key to GitHub.

**20. Webhooks**

**What:**  
Webhooks send HTTP requests when certain events happen on GitHub.

**Why:**  
Webhooks allow integration with external services like CI/CD tools.

**How:**  
Create a webhook in your GitHub repository settings and specify the URL for notifications.

**21. Push, Pull, and Tracking Remote Repos**

**What:**  
Push and pull commands sync local changes with a remote repository.

**Why:**  
Pushing and pulling are how you share your work with others and stay up-to-date with the project.

**How:**  
Push to a remote repository:

git push origin main

Pull updates from the remote:

git pull origin main

**22. Managing Conflicts**

**What:**  
Merge conflicts happen when two people make changes to the same part of a file.

**Why:**  
You need to resolve conflicts to make sure changes from different branches work together.

**How:**  
Git will mark the conflict in the file. You manually edit the file to fix it and commit the resolution.

Continuing from where we left off:

**23. GitHub Training**

**What:**  
GitHub training involves learning how to use GitHub's features for managing repositories, collaborating with teams, and administering projects.

**Why:**  
GitHub is a popular platform for hosting Git repositories, and training helps you maximize its potential for collaboration and project management.

**How:**  
GitHub offers tutorials and documentation. You can sign up and start experimenting with creating repositories, forking projects, and collaborating with others.

**24. GitLab Basics**

**What:**  
GitLab is another platform for Git repositories, offering features similar to GitHub but with added tools for Continuous Integration (CI), Continuous Deployment (CD), and project management.

**Why:**  
GitLab allows teams to manage their code, track issues, and automate testing and deployment workflows.

**How:**  
To use GitLab:

1. Sign up for an account at [GitLab](https://gitlab.com/).
2. Create a new repository or import an existing one.
3. GitLab provides built-in CI/CD pipelines for automating your code lifecycle.

**25. Download and Install GitLab for Local Use**

**What:**  
You can set up a self-hosted GitLab server on your machine to manage repositories locally.

**Why:**  
Running GitLab locally gives you full control over your repositories and the ability to customize its settings and features.

**How:**

1. Follow the [official GitLab installation guide](https://about.gitlab.com/install/).
2. After installation, access GitLab locally by visiting http://localhost in your browser.

**26. Adding Users and Groups to GitLab**

**What:**  
GitLab allows you to add users and organize them into groups to manage permissions and access control.

**Why:**  
Grouping users helps in managing access to repositories, ensuring that only the right people have the right permissions.

**How:**

* **Adding Users:** Go to the GitLab admin interface and click on **Users**, then add users by their email addresses.
* **Creating Groups:** Navigate to **Groups** and create a new group. Add users to the group for access control.

**27. Creating and Managing Projects in GitLab**

**What:**  
In GitLab, projects are repositories where your code is stored and managed.

**Why:**  
Projects help organize and track different versions of your code, issues, and merge requests.

**How:**

* To create a project, click **New Project** and choose whether to create a blank project or import one from an external Git repository.
* Once created, you can push your local code to the GitLab project using Git commands like git push.

**28. Push Changes and Merge with GitLab**

**What:**  
Pushing changes to GitLab and merging changes from different branches are key steps in collaborative development.

**Why:**  
Pushing ensures your local changes are shared with the team. Merging brings changes from other branches together.

**How:**

* **Push to GitLab:**

git push origin main

* **Merge in GitLab:**  
  You can create a merge request in GitLab’s web interface to merge branches.

**29. GitHub Signed Commits**

**What:**  
Signed commits ensure that the author of the commit is verified using cryptographic signatures.

**Why:**  
Signed commits provide security by proving that the commit came from the person it claims to be from, ensuring integrity.

**How:**  
To sign a commit, use:

git commit -S -m "Commit message"

You’ll need to set up GPG keys for signing. GitHub provides instructions on adding a GPG key to your account.

**30. GitHub LFS (Large File Storage)**

**What:**  
Git LFS is an extension for Git that helps manage large files by replacing them with lightweight references inside your Git repository.

**Why:**  
Large files like images, videos, and datasets can bloat your repository, making it slow. Git LFS helps keep your repository size manageable.

**How:**

1. **Installation:**
2. git lfs install
3. **Track specific file types:**
4. git lfs track "\*.mp4"
5. **Commit and push large files:** When you commit and push large files, Git LFS manages them in the background.

**31. What is Git Large File Storage (LFS)?**

**What:**  
Git LFS is a tool that replaces large files in your repository with pointers, while storing the actual files in a separate storage.

**Why:**  
Git LFS helps keep the size of your Git repository small and manageable, especially when working with binary files or large assets.

**How:**  
When you clone a repository with LFS-enabled files, Git LFS fetches the actual file content from the server:

git lfs clone <repository-url>

**32. Installation of Git LFS**

**What:**  
Git LFS needs to be installed separately from Git.

**Why:**  
Without Git LFS, Git doesn't natively manage large binary files efficiently.

**How:**  
Install Git LFS:

* **On Linux:**
* sudo apt-get install git-lfs

or for RHEL:

sudo yum install git-lfs

**33. Select the File Types You'd Like Git LFS to Manage**

**What:**  
Git LFS allows you to specify which file types should be managed by LFS.

**Why:**  
This ensures that only large files are handled by LFS, preventing unnecessary overhead for smaller files.

**How:**  
Track a file type:

git lfs track "\*.mp4"

You can edit the .gitattributes file directly for advanced configurations.

**34. Including/Excluding Git LFS Files**

**What:**  
You can choose which files should be included or excluded from LFS tracking.

**Why:**  
You might not need LFS for every large file (e.g., you may only want to track images but not temporary files).

**How:**  
To stop tracking a file type:

git lfs untrack "\*.mp4"

**35. Locking Git LFS Files**

**What:**  
Git LFS supports locking files to prevent others from editing them simultaneously.

**Why:**  
Locking helps prevent merge conflicts by ensuring that only one person can edit a large file at a time.

**How:**  
To lock a file:

git lfs lock <file>

To unlock a file:

git lfs unlock <file>

**36. GitHub Administration**

**What:**  
GitHub administration involves managing repositories, teams, and users in an organization.

**Why:**  
Effective administration ensures proper permissions, security, and collaboration across teams.

**How:**

* **Repository permission levels**: Admins can control who has access to repositories.
* **Team permission levels**: Organize users into teams and set roles such as read, write, or admin.

**37. Best Practices for Team-Level Administration**

**What:**  
Best practices for managing teams within a GitHub organization ensure smooth collaboration and security.

**Why:**  
It’s important to set the right access levels, create efficient workflows, and ensure that all team members follow best practices.

**How:**

* Use **teams** to organize users.
* Set clear **permission levels** for each team (e.g., read-only, write, admin).
* Regularly **audit** repository access and activity.

**38. GitHub's Authentication Options**

**What:**  
GitHub supports multiple authentication methods, including passwords, two-factor authentication (2FA), and SSH keys.

**Why:**  
Authentication methods protect your GitHub account from unauthorized access.

**How:**

* **SSH keys**: Use for a more secure connection to repositories.
* **2FA**: Enable for an extra layer of security.

**39. Repository Permission Levels**

**What:**  
Repository permission levels define who can access, modify, or administer a repository.

**Why:**  
Permission levels control access to sensitive data and prevent unauthorized changes.

**How:**

* **Read**: Can view the repository.
* **Write**: Can make changes (push, merge).
* **Admin**: Can configure settings and manage access.

**40. Team Permission Levels**

**What:**  
Team permission levels manage what members of a team can do in repositories.

**Why:**  
It ensures that each team member can only perform actions relevant to their role.

**How:**  
Set team permissions via the GitHub web interface under **Teams** settings.

**41. Organization Permission Levels**

**What:**  
Organization-level permissions control who can access, manage, or create repositories within an organization on GitHub.

**Why:**  
It’s essential to structure organization-level permissions based on roles like admin, member, or owner.

**How:**  
Admin roles can modify repository settings, while members have limited access.

**42. Repository Security and Management**

**What:**  
Managing security involves configuring access controls, keeping sensitive data safe, and ensuring best practices in coding and commits.

**Why:**  
Repository security prevents unauthorized access, data loss, and potential breaches.

**How:**  
Enable 2FA, limit access via permissions, and use branch protection rules.

**43. Git Hooks**

**What:**  
Git Hooks are scripts that run automatically at certain points in the Git workflow.

**Why:**  
They help automate tasks like verifying commit messages,

running tests, or sending notifications.

**How:**  
To use Git Hooks, navigate to the .git/hooks directory and modify hook scripts. For example, to prevent commits with bad messages, modify the pre-commit hook.

**44. How to Use Git Hooks**

**What:**  
Git Hooks help automate actions based on specific events like commits or pushes.

**Why:**  
They ensure consistency in coding practices and workflows.

**How:**  
Create a pre-commit hook to check the commit message format:

#!/bin/sh

# Reject commits with missing ticket ID

if ! git commit -m | grep -q "TICKET-"; then

echo "Commit message must include a ticket ID"

exit 1

fi

**45. Git Config - Command Introduction**

**What:**  
Git config is used to configure Git settings, like user name, email, or editor.

**Why:**  
It personalizes your Git environment and controls how Git behaves.

**How:**  
Use the git config command to set configurations. For example, to set your Git editor:

git config --global core.editor "nano"

**46. Git Config Levels and Files**

**What:**  
Git configurations can be set globally, locally, or system-wide, depending on your needs.

**Why:**  
This allows you to customize Git settings for a single project or across all projects.

**How:**

* **Global config**: Applies to all repositories on your machine.

git config --global user.name "Your Name"

* **Local config**: Applies only to the current repository.

git config user.name "Repo Specific Name"

**47. Writing a Value**

**What:**  
Writing a value in Git config means setting a specific configuration option.

**Why:**  
It customizes your Git experience, like choosing a default text editor or setting color outputs.

**How:**  
For example, to set the default editor to vim:

git config --global core.editor "vim"

**48. Merge Tools and Configure It in Your System**

**What:**  
Merge tools help resolve conflicts when merging branches.

**Why:**  
They provide a user-friendly interface to visualize and resolve merge conflicts.

**How:**  
Configure a merge tool like vimdiff:

git config --global merge.tool vimdiff

**49. Colored Outputs**

**What:**  
Git can display colored outputs for different commands to make them easier to read.

**Why:**  
Colored outputs help differentiate between changes, errors, and warnings visually.

**How:**  
Enable color in Git with:

git config --global color.ui auto

**50. Formatting & Whitespace**

**What:**  
Git allows you to control how whitespace issues are handled during commits.

**Why:**  
Managing formatting and whitespace ensures code consistency across commits.

**How:**  
You can configure how Git handles whitespace errors:

git config --global core.whitespace cr-at-eol