

Git: From Beginner to Advanced

Git is a distributed version control system designed to track changes in source code during software development. It enables multiple developers to collaborate efficiently, manage code versions, and maintain a history of changes. This explanation will take you from a beginner's understanding to an advanced level, covering Git setup, GitHub repositories, branching, workflows, commits, pull requests, GitHub Actions, CI/CD pipelines, matrix builds, workflow security, GitHub Secrets, linting, and related tools like golangci-lint and Trivy. I'll also include relevant topics like Git rebasing, merge strategies, and advanced GitHub features to provide a comprehensive guide.

1. Git Setup

Theory and Concept

Git is a distributed version control system (VCS) created by Linus Torvalds in 2005. Unlike centralized VCS (e.g., SVN), every developer has a full copy of the repository, including its history, enabling offline work and faster operations. Git tracks changes to files, allowing you to revert to previous versions, collaborate on code, and manage complex projects.

Setup Process (Beginner)

1. Install Git:

- **Windows:** Download from git-scm.com and install. Use Git Bash for a Unix-like terminal.
- **macOS:** Install via Homebrew (`brew install git`) or Xcode.
- **Linux:** Use package managers (e.g., `sudo apt install git` for Ubuntu).
- Verify: `git --version`.

2. Configure Git:

- Set your name and email (used for commit metadata):
`git config --global user.name "Your Name"`
`git config --global user.email "your.email@example.com"`
- Set a default text editor (e.g., VS Code):
`git config --global core.editor "code --wait"`
- Enable colored output:
`git config --global color.ui auto`

3. Authentication:

- Use SSH or HTTPS for repository access.
- **SSH Setup:**
 - Generate an SSH key: `ssh-keygen -t ed25519 -C "your.email@example.com"`.
 - Add the key to your GitHub account under Settings > SSH and GPG keys.

- Test: `ssh -T git@github.com`.
- **HTTPS:** Use a personal access token (PAT) for authentication (GitHub no longer supports password-based auth).

Advanced Setup

- **Git Aliases:** Simplify commands with aliases.

```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.st status
```

Now, `git co` works instead of `git checkout`.

- **Git Hooks:** Automate tasks (e.g., linting before commits). Hooks are scripts in `.git/hooks/`. Example:

- Create a pre-commit hook to run tests:

```
# .git/hooks/pre-commit
#!/bin/sh
npm test
```

- Make executable: `chmod +x .git/hooks/pre-commit`.

- **Global .gitignore:** Ignore files globally (e.g., `.DS_Store`, `node_modules`):

```
git config --global core.excludesfile ~/.gitignore_global
echo "node_modules/" >> ~/.gitignore_global
```

Example

You're starting a project. After installing Git, you configure it:

```
git config --global user.name "Jane Doe"
git config --global user.email "jane@example.com"
```

You generate an SSH key, add it to GitHub, and verify connectivity. You're now ready to create or clone repositories.

2. GitHub Repository

Theory and Concept

A GitHub repository is a storage space for your project's files, hosted on GitHub's servers. It integrates with Git for version control and provides collaboration features like issues, pull requests, and project boards. Repositories can be public, private, or internal (for organizations).

Beginner Steps

1. **Create a Repository:**
 - On GitHub, click “New” under Repositories.
 - Name it (e.g., `my-project`), add a description, and choose public/private.
 - Optionally initialize with a README, `.gitignore`, or license.
2. **Clone a Repository:**
 - Copy the repository URL (HTTPS or SSH).
 - Clone locally: `git clone git@github.com:username/my-project.git`.
 - Navigate: `cd my-project`.
3. **Basic Workflow:**
 - Add files: `touch index.html`.
 - Stage: `git add index.html`.
 - Commit: `git commit -m "Add index.html"`.
 - Push: `git push origin main`.

Advanced Features

- **Repository Settings:**
 - Enable features like wikis, issues, or discussions.
 - Configure branch protection rules (e.g., require PR reviews before merging).
- **Collaborators:** Invite team members under Settings > Collaborators.
- **Forks:** Create a copy of a repository for experimentation or contribution.
- **Templates:** Mark a repository as a template for reuse.

Example

You create a repository called `my-blog`. After initializing it with a README, you clone it:

```
git clone git@github.com:username/my-blog.git
cd my-blog
echo "# My Blog" >> README.md
git add README.md
git commit -m "Update README"
git push origin main
```

You invite a collaborator and set a branch protection rule requiring one approval for merges to `main`.

3. Git Branching

Theory and Concept

Branching allows you to work on different versions of a project simultaneously. The `main` branch (or `master`) is typically the production-ready branch. Other branches are used for features, bug fixes, or experiments. Branches are lightweight pointers to commits.

Beginner Branching

1. **Create a Branch:**

```
git branch feature/add-login
```

2. **Switch to a Branch:**

```
git checkout feature/add-login
```

Or combine: `git checkout -b feature/add-login`.

3. **Work and Commit:**

- Edit files, stage, and commit as usual.

4. **Push Branch:**

```
git push origin feature/add-login
```

5. **Merge Branch:**

- Switch to main: `git checkout main`.
- Merge: `git merge feature/add-login`.
- Push: `git push origin main`.

6. **Delete Branch:**

- Local: `git branch -d feature/add-login`.
- Remote: `git push origin --delete feature/add-login`.

Advanced Branching

- **Rebasing:** Rebase a branch to integrate changes from `main` cleanly:

```
git checkout feature/add-login
git rebase main
```

Resolve conflicts if any, then force-push: `git push --force-with-lease`.

- **Interactive Rebase:** Rewrite commit history:

```
git rebase -i HEAD~3
```

Edit, squash, or reorder the last three commits.

- **Branch Strategies:**

- **Git Flow:** Uses `main` (production), `develop` (integration), feature branches, release branches, and hotfixes.
- **GitHub Flow:** Simplifies to `main` and feature branches with pull requests.
- **Trunk-Based Development:** Short-lived branches merged directly to `main`.

Example

You're adding a login feature:

```
git checkout -b feature/add-login
touch login.js
git add login.js
git commit -m "Add login functionality"
git push origin feature/add-login
```

You realize `main` has new changes, so you rebase:

```
git fetch origin
git rebase origin/main
git push --force-with-lease
```

4. Pull Request (PR)

Theory and Concept

A pull request is a GitHub feature to propose, review, and merge changes from one branch to another (e.g., feature branch to `main`). PRs facilitate code review, discussion, and collaboration.

Beginner PR Workflow

1. **Create a PR:**
 - Push your branch: `git push origin feature/add-login`.
 - On GitHub, click “Compare & pull request” for the branch.
 - Add a title, description, and reviewers.
2. **Review:**
 - Reviewers comment, suggest changes, or approve.
3. **Merge:**
 - Merge via GitHub (merge commit, squash, or rebase).
 - Delete the branch after merging.

Advanced PR Features

- **Draft PRs:** Mark a PR as “Draft” to indicate it’s not ready for review.
- **Merge Strategies:**

- **Merge Commit:** Combines histories, creating a merge commit.
- **Squash and Merge:** Combines all commits into one, cleaner history.
- **Rebase and Merge:** Applies commits directly to the target branch.
- **Auto-Merge:** Enable auto-merge to merge PRs automatically when conditions (e.g., approvals, passing CI) are met.
- **Linked Issues:** Reference issues (e.g., **Fixes #123**) to auto-close them on merge.

Example

You push `feature/add-login` and create a PR. A reviewer suggests changes. You update:

```
git checkout feature/add-login
echo "Fix: Add validation" >> login.js
git add login.js
git commit -m "Add validation per review"
git push origin feature/add-login
```

The PR is approved, and you squash-merge it into `main`.

5. GitHub Workflow

Theory and Concept

The GitHub Workflow (often GitHub Flow) is a lightweight branching strategy emphasizing feature branches, pull requests, and continuous deployment. It's ideal for teams with frequent releases.

Workflow Steps

1. Create a feature branch from `main`.
2. Make changes and commit.
3. Push the branch and create a PR.
4. Review and merge the PR into `main`.
5. Deploy `main` to production.
6. Delete the feature branch.

Advanced Workflow

- **Protected Branches:** Prevent direct pushes to `main` and require PRs with approvals.
- **Code Owners:** Assign specific team members to review certain files (via `CODEOWNERS` file).
- **Issue Templates:** Standardize issue and PR descriptions for consistency.

- **Labels and Milestones:** Organize PRs and issues for project management.

Example

Your team follows GitHub Flow. You create `feature/add-search`, push changes, and open a PR. The PR requires two approvals and passing CI checks. After merging, `main` is deployed automatically via a CI/CD pipeline.

6. Git Commits

Theory and Concept

A commit is a snapshot of changes in your repository, stored with a unique SHA hash. Commits include metadata (author, timestamp, message) and are the building blocks of Git's history.

Beginner Commits

- **Stage Changes:** `git add file.txt` or `git add .` (all files).
- **Commit:** `git commit -m "Descriptive message"`.
- **View History:** `git log` or `git log --oneline`.

Advanced Commits

- **Amending Commits:** Modify the last commit:

```
git commit --amend -m "Updated message"
```
- **Signed Commits:** Add GPG signatures for security:
 - Configure GPG key: `gpg --gen-key`.
 - Link to Git: `git config --global user.signingkey <key-id>`.
 - Sign commits: `git commit -S -m "Signed commit"`.
- **Atomic Commits:** Keep commits small and focused (e.g., one commit per logical change).
- **Commit Messages:** Follow conventions (e.g., “type(scope): description”):

```
feat(auth): add login endpoint
fix(ui): correct button alignment
```

Example

You accidentally commit with a vague message:

```
git commit -m "Changes"
```

You amend it:

```
git commit --amend -m "feat(auth): implement user login"
```

7. GitHub Actions

Theory and Concept

GitHub Actions is a CI/CD platform integrated into GitHub. It automates workflows (e.g., testing, linting, deployment) triggered by events like pushes or PRs. Workflows are defined in YAML files under `.github/workflows/`.

Beginner Workflow

1. Create a Workflow:

- Create `.github/workflows/ci.yml`:

```
name: CI
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Set up Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
      - run: npm install
      - run: npm test
```

2. **Triggers:** Run on `push`, `pull_request`, or `schedules`.

3. **Jobs and Steps:** A job runs on a virtual machine (e.g., `ubuntu-latest`). Steps are individual tasks.

Advanced Features

- **Matrix Builds:** Test across multiple environments:

```
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [16, 18, 20]
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
```



```

    with:
      node-version: ${ matrix.node-version }
    - run: npm test

```

- **Custom Actions:** Create reusable actions in your repository or use public ones.
- **Caching:** Speed up workflows:

```

- uses: actions/cache@v4
  with:
    path: ~/.npm
    key: ${ runner.os }-node-${ hashFiles('**/package-lock.json') }

```

Example

You set up a workflow to lint and test a Node.js app on every push. The matrix build tests Node.js versions 16, 18, and 20, ensuring compatibility.

8. CI/CD Pipeline

Theory and Concept

CI/CD (Continuous Integration/Continuous Deployment) automates building, testing, and deploying code. CI ensures code is tested on every change, while CD automates deployment to staging or production.

Beginner CI/CD

- **CI:** Run tests and linting on every push/PR.
- **CD:** Deploy to a server after merging to main.

```

name: Deploy
on:
  push:
    branches: [main]
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Deploy to Server
        run: ssh user@server 'cd /app && git pull && npm install && pm2 restart all'

```

Advanced CI/CD

- **Environments:** Use GitHub Environments for deployment targets (e.g., staging, production).
- **Approval Gates:** Require manual approval for production deployments.
- **Blue-Green Deployment:** Deploy to a new environment, test, then switch traffic.
- **Canary Releases:** Deploy to a subset of users before full rollout.

Example

Your CI/CD pipeline tests a Go app, builds a Docker image, and deploys to AWS:

```
name: CI/CD
on: [push]
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-go@v5
        with:
          go-version: '1.21'
      - run: go test ./...
      - name: Build Docker Image
        run: docker build -t my-app .
      - name: Push to Registry
        run: |
          docker login -u ${{ secrets.AWS_ACCESS_KEY }} -p ${{ secrets.AWS_SECRET_KEY }}
          docker push my-app
```

9. Workflow Security

Theory and Concept

Workflow security ensures that CI/CD pipelines and GitHub Actions are protected from malicious code or unauthorized access. This includes securing secrets, limiting permissions, and auditing workflows.

GitHub Secrets

- **Purpose:** Store sensitive data (e.g., API keys, passwords) securely.
- **Setup:**
 - Go to Repository > Settings > Secrets and variables > Actions.

- Add a secret (e.g., `AWS_ACCESS_KEY`).
- **Usage:**

```
steps:
  - run: aws configure set aws_access_key_id "${ secrets.AWS_ACCESS_KEY }"
```

Advanced Security

- **Dependabot:** Automatically update dependencies to fix vulnerabilities.
- **Code Scanning:** Use CodeQL to detect security issues in code.
- **Workflow Permissions:** Limit Actions' access (e.g., read-only for PRs from forks).

```
permissions:
  contents: read
  pull-requests: write
```

- **Signed Commits:** Enforce signed commits for verified changes.
- **Secret Scanning:** GitHub scans for leaked secrets in public repositories.

Example

You store an AWS key in GitHub Secrets and use it in a workflow. You enable Dependabot to update `package.json` and configure CodeQL for security scanning.

10. Linting

Theory and Concept

Linting checks code for errors, style violations, or potential bugs, ensuring consistency and quality. Tools like `golangci-lint` (Go) and `Trivy` (security) are commonly used.

`golangci-lint`

- **Purpose:** Aggregates multiple Go linters (e.g., `golint`, `staticcheck`).
- **Setup:**
 - Install: `go install github.com/golangci/golangci-lint/cmd/golangci-lint@latest.`
 - Run: `golangci-lint run.`
- **GitHub Actions Integration:**

```
name: Lint
on: [push]
```

```

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: golangci/golangci-lint-action@v6
        with:
          version: latest

```

Trivy

- **Purpose:** Scans for vulnerabilities in dependencies and container images.
- **Setup:**
 - Install: `brew install trivy` or use a Docker image.
 - Run: `trivy fs .` (filesystem) or `trivy image my-app`.
- **GitHub Actions:**

```

name: Security Scan
on: [push]
jobs:
  scan:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Run Trivy
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: 'my-app:latest'
          format: 'table'
          exit-code: '1'

```

Example

Your Go project uses golangci-lint to enforce style and Trivy to scan for vulnerabilities. Both run in a CI pipeline, failing the build if issues are found.

11. GitHub README

Theory and Concept

A README is the entry point to your repository, explaining the project's purpose, setup, and usage. It's typically written in Markdown (`README.md`).

Beginner README

- Include:
 - Project title and description.
 - Installation and usage instructions.
 - License and contribution guidelines.

- Example:

```
# My Blog
A simple blog built with Node.js.
```

```
## Installation
```bash
npm install
npm start
```


### Usage

Visit <http://localhost:3000>.

### License

MIT ““

## Advanced README

- **Badges:** Show build status, coverage, or version:  
 `! [CI] (https://github.com/username/my-blog/workflows/CI/badge.svg)`
- **Table of Contents:** For large READMEs.
- **Screenshots or GIFs:** Demonstrate features visually.
- **Contributing Guide:** Link to CONTRIBUTING.md for detailed instructions.

### Example

Your README includes a badge for CI status, a setup guide, and a link to a contributing guide, making it welcoming to new contributors.

---

## 12. Additional Topics

### Git Rebasing

- **Purpose:** Reapply commits on top of another base, creating a linear history.

- **Example:**

```
git checkout feature/add-login
git rebase main
git push --force-with-lease
```

- **Pros:** Cleaner history.
- **Cons:** Risk of conflicts; avoid on shared branches.

## Merge Strategies

- **Fast-Forward:** Applies commits directly if no divergence.

```
git merge --ff-only feature/add-login
```

- **No Fast-Forward:** Always creates a merge commit.

```
git merge --no-ff feature/add-login
```

- **Squash:** Combines commits into one.

```
git merge --squash feature/add-login
```

## Git Stash

- **Purpose:** Temporarily save uncommitted changes.
- **Usage:**

```
git stash
git checkout main
git stash pop
```

## Git Tags

- **Purpose:** Mark specific commits (e.g., releases).
- **Usage:**

```
git tag v1.0.0
git push origin v1.0.0
```

## GitHub Projects

- **Purpose:** Kanban boards for task management.
  - **Usage:** Create a project, add issues/PRs, and track progress.
-

## From Beginner to Hero

1. **Beginner:** Install Git, configure it, and learn basic commands (`add`, `commit`, `push`). Create a GitHub repository and make your first PR.
2. **Intermediate:** Master branching, rebasing, and PR workflows. Set up a basic GitHub Actions CI pipeline with tests and linting.
3. **Advanced:** Implement complex CI/CD pipelines with matrix builds, secure workflows with secrets, and integrate tools like `golangci-lint` and `Trivy`. Use advanced Git features (rebasing, tags) and maintain professional READMEs.

By following this guide, you've gone from setting up Git to orchestrating sophisticated workflows, ready to contribute to or lead complex projects with confidence. If you have specific questions or want to dive deeper into any topic, let me know!