

Sistemi di versioning (GIT)

e provider (GitHub, GitLab, Bitbucket)

• • •



Contenuto del corso

Introduzione ai sistemi di versioning

Requisiti di sistema

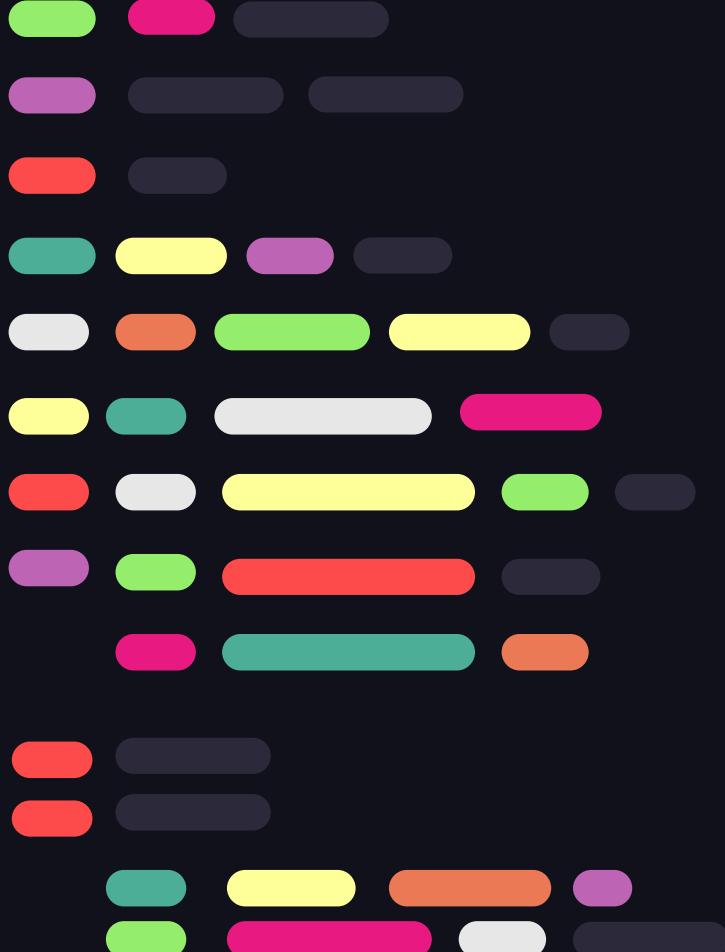
Repository, commit, branch e tag

Inizializzazione di un repository GIT

Creazione e switch tra branch

Lavoro con repository remoti

Utilizzo delle opzioni avanzate di GIT



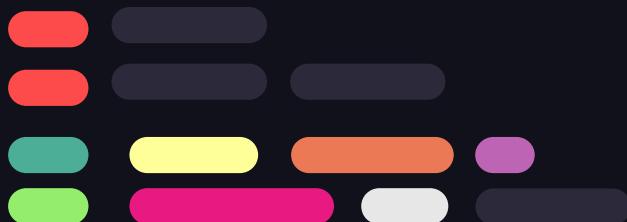
GIT }

< "Git: version, collaborate,
master your code!" >



01 { . .

Introduzione ai
sistemi di versioning



}

. .

Perché non basta salvare i file come v1, v2, v3



- Molti team non usano un sistema di versioning e provano a gestire le versioni rinominando i file: `relazione_finale.docx`, `relazione_finale2.docx`, `relazione_finalissima.docx`. Questo metodo è inefficiente e rischioso:
- Si perde traccia di chi ha cambiato cosa e quando
- Si sovrascrivono contenuti
- Si generano duplicati in cloud ed email
- Nessuno è certo di lavorare sull'ultima versione
- I sistemi di versioning come Git risolvono il problema tracciando automaticamente modifiche, autore e data, permettendo di tornare indietro e collaborare in modo strutturato.



Introduzione ai sistemi di versioning

- I sistemi di versioning consentono di gestire le versioni dei file nel tempo.
- Permettono di tracciare le modifiche, ripristinare versioni precedenti e collaborare in team.
- Due categorie principali: centralizzati (**SVN**) e distribuiti (**GIT**).



Perché Git è lo standard mondiale

Git è oggi lo standard de facto per il controllo di versione:

- È distribuito: ogni sviluppatore ha la cronologia completa
- È veloce nelle operazioni locali
- Gestisce i branch in modo efficiente
- Si integra con workflow moderni (GitHub Flow, GitFlow, trunk-based)
- È supportato da piattaforme come GitHub, GitLab, Bitbucket

Conoscere Git è una competenza fondamentale richiesta in aziende, università e team di ogni dimensione.

Differenze tra sistemi centralizzati e distribuiti



- **Sistemi centralizzati:** tutti i file e la cronologia sono memorizzati su un server centrale.
- **Sistemi distribuiti:** ogni utente ha una copia completa del repository.
- GIT è un sistema distribuito, offrendo maggiore flessibilità e affidabilità.



Perché scegliere GIT?

- **Velocità:** GIT è estremamente rapido nelle operazioni locali.
- **Branching efficiente:** creazione e fusione di branch senza impatto sulle prestazioni.
- **Distribuito:** possibilità di lavorare senza connessione internet.
- **Affidabilità:** protezione da perdite di dati accidentali.



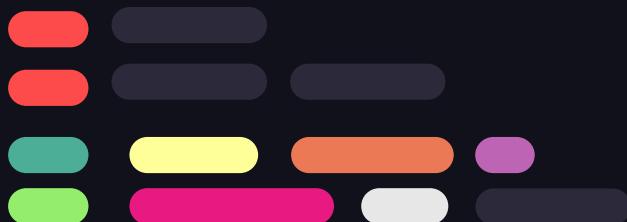
Provider più diffusi

- **GitHub:** Il più popolare, con strumenti avanzati per il versionamento e la collaborazione.
- **GitLab:** Alternativa open-source con funzionalità avanzate di CI/CD.
- **Bitbucket:** Integrato con Jira, ideale per team aziendali.



01 { . .

Installazione e
configurazione di GIT



} . .

Requisiti di sistema per l'installazione

- **Windows:** richiede Git Bash o Git for Windows.
- **macOS:** disponibile tramite Homebrew (`brew install git`).
- **Linux:** disponibile tramite package manager (`sudo apt install git`).

Installazione di GIT

- **Windows:** scaricare l'installer dal sito ufficiale e seguire la procedura guidata.
- **macOS:** installazione con Homebrew (`brew install git`).
- **Linux:** installazione con il package manager (`sudo apt install git`).
- Verifica dell'installazione: `git --version`

Configurazione iniziale di GIT

- **Impostazione nome e email:**
 - `git config --global user.name "Nome Utente"`
 - `git config --global user.email "email@example.com"`
- **Configurazione dell'editor predefinito:**
 - `git config --global core.editor "vim"`

Configurazioni avanzate e alias

Creazione di alias per comandi comuni:

- `git config --global alias.st status`
- `git config --global alias.cm commit -m`

Configurazione del colore dell'output:

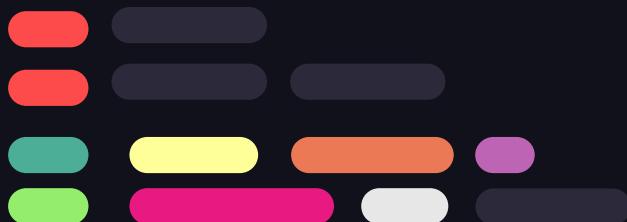
- `git config --global color.ui auto`

Verifica della configurazione: `git config --list`



03 { ..

Concetti fondamentali di GIT



..

Struttura di un repository GIT: Working Tree, Staging Area, Repository



- **Working Tree:** l'area in cui si lavora sui file attuali.
- **Staging Area:** un'area temporanea in cui vengono preparate le modifiche prima del commit.
- **Repository:** il database in cui vengono salvate in modo permanente le versioni dei file.



Differenza tra file tracciati, non tracciati e modificati

- **File tracciati:** sono sotto controllo di GIT e possono essere versionati.
- **File non tracciati:** file nuovi che non sono ancora stati aggiunti al controllo di versione.
- **File modificati:** file già tracciati che sono stati alterati e non ancora salvati nel repository.



Working Directory → Staging Area → Repository

Il ciclo base di Git si fonda su tre aree distinte:

- **Working Directory** → dove si modificano i file
- **Staging Area** → dove si selezionano le modifiche pronte per il commit
- **Repository** → dove le modifiche diventano permanenti nella cronologia

Il percorso tipico è: modifica → git add → git commit.



Primo approccio ai comandi: git init, git status, git add, git commit

- **git init:** inizializza un nuovo repository GIT in una cartella.
- **git status:** mostra lo stato dei file (modificati, non tracciati, in staging).
- **git add:** aggiunge file alla Staging Area per essere committati.
- **git commit:** registra le modifiche nel repository.

Visualizzazione dello storico con git log e differenze tra versioni con git diff

- **git log**: mostra la cronologia dei commit con autore e messaggio.
- **git diff**: confronta due versioni di un file per visualizzare le modifiche.



git init

Sintassi: `git init`

Serve a:

Creare un nuovo repository Git in una cartella locale.

Cosa fa internamente:

Crea la cartella nascosta `.git` con:

- oggetti
- index
- riferimenti ai branch
- configurazione locale



git clone

Sintassi:

`git clone <url>`

oppure

`git clone <url> <nome_cartella>`

Serve a:

Scaricare un repository remoto creando una copia locale completa (con cronologia)

git config

Sintassi:

```
git config --global user.name "Nome"  
oppure  
git config --global user.email "email@example.com"  
oppure  
git config -list
```

Serve a:

Configurare:

- identità
- editor
- comportamento default



git status

Sintassi: `git status`

Mostra:

- file modificati
- file staged
- file non tracciati
- branch corrente



git diff

Sintassi:

`git diff`

oppure

`git diff --staged`

oppure

`git diff <commit1> <commit2>`

Confronta le differenze nel versioning



git log

Sintassi:

`git log`

oppure

`git log -oneline`

oppure

`git log --graph -all`

Visualizza la cronologia; molto utile per mostrare la struttura dei branch.



git show

Sintassi: `git show <hash>`

Mostra il contenuto dettagliato di un commit.

git add

Sintassi:

`git add file.txt` // file specifico

oppure

`git add .` //tutti i files della directory corrente (e sottodirectory)

oppure

`git add -A` //tutti i files dell'intero progetto

Sposta file nello staging area

git commit

Sintassi:

`git commit -a` // prende tutte le modifiche dei file già tracciati (modificati/eliminati) e li aggiunge automaticamente all'area di staging oppure

`git commit -m «messaggio»` // permette di inserire il messaggio di commit direttamente dalla riga di comando tra virgolette, evitando l'apertura dell'editor di testo

oppure

`git commit -am «Messaggio»` // combinazione delle options -a e -m

Crea uno snapshot del progetto.

commit = oggetto immutabile (contiene hash, autore, data, parent)



git rm

Sintassi:

```
git rm file.txt
```

Rimuove il file e lo mette staged per eliminazione



git mv

Sintassi:

```
git mv old.txt new.txt
```

Sposta o rinomina file mantenendo tracciamento



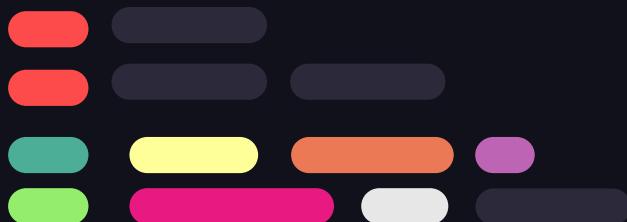
Mini esercizio: il ciclo base di Git

- Creare un file README.md
- Aggiungerlo con git add README.md
- Registrare la modifica con git commit -m "Primo commit"
- Modificare il file, controllare lo stato con git status, aggiungere di nuovo e fare un secondo commit



04 { ..

Creazione e gestione
di un repository GIT



}

..

Creazione di un repository locale: git init



- Creazione di un nuovo repository con git init.
- Struttura della cartella .git.
- Differenza tra repository vuoto e repository con file esistenti.



Clonazione di un repository remoto con git clone

- **git clone <url>**: Copia un repository remoto in locale.
- Configurazione dell'URL remoto.
- Differenze tra clonazione HTTPS e SSH.

Aggiunta, modifica e rimozione di file con git add e git rm



- **git add <file>**: Aggiunge file alla Staging Area.
- **git rm <file>**: Rimuove un file dalla Staging Area e dal repository.
- Aggiunta di più file con git add ..



Creazione di commit e loro modifica (git commit, git commit --amend)

- **git commit -m "Messaggio"**: salva le modifiche con un messaggio descrittivo.
- **git commit --amend**: modifica l'ultimo commit. Permette di unire modifiche dimenticate (precedentemente aggiunte alla staging area con git add) al commit precedente, oppure di correggere il messaggio dell'ultimo commit. In realtà, questo comando crea un nuovo commit che sostituisce il precedente

Come esplorare la cronologia dei commit (git log, git show)

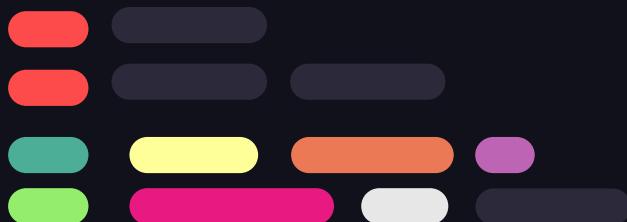


- **git log**: Elenco dei commit effettuati.
- **git show <commit>**: Mostra i dettagli di un commit specifico.



05 { ..

Gestione dei branch in GIT

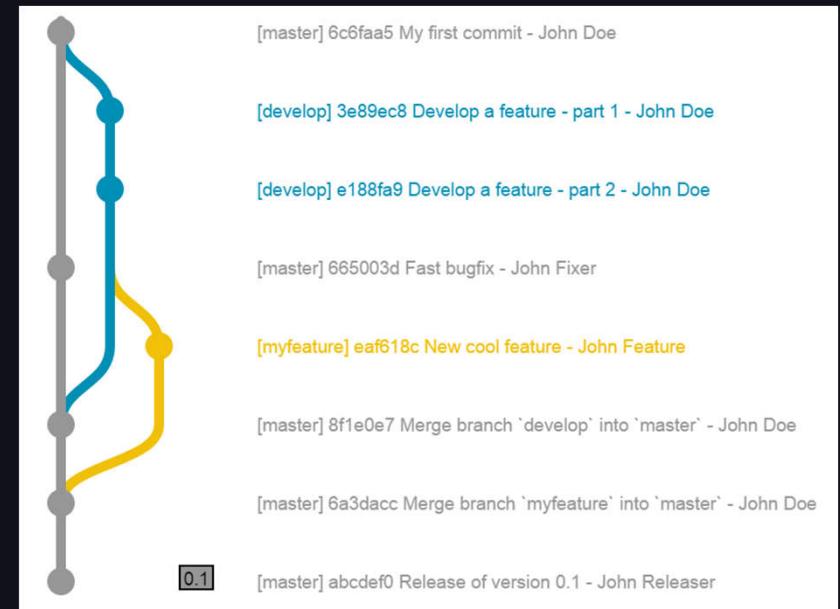


..

Concetto di branching: perché usare i branch?



- Il branching permette di lavorare su più funzionalità contemporaneamente senza interferire con il codice principale.
- Consente di sviluppare nuove feature senza compromettere la stabilità del progetto.
- Si possono creare branch per ogni funzionalità, fix o miglioramento.



git branch

Sintassi:

`git branch`

oppure

`git branch nome_branch` // specifica il branch

oppure

`git branch -d nome_branch` // eliminazione di un branch locale in git

Serve a:

- creare
- visualizzare
- eliminare branch

git checkout

Sintassi:

`git checkout nome_branch`

oppure

`git checkout -b nuovo_branch`

Serve a:

- cambiare branch
- creare e spostarsi su nuovo branch



git switch

Sintassi:

`git switch nome_branch`

oppure

`git switch -c nuovo_branch`

È più chiaro semanticamente di `git checkout`

Unione di branch (git merge) e concetti base di merging



- `git merge nome-branch`: unisce un branch al branch corrente.
- Il merge può essere **fast-forward** (se il branch di destinazione non ha nuove modifiche) o **con commit di merge**.
- Il merge deve essere usato con attenzione per evitare conflitti.

git network graph





git merge

Sintassi:

```
git merge nome_branch
```

Integra un branch dentro quello corrente.

Permette:

- fast-forward
- merge commit
- conflitti



git rebase

Sintassi:

```
git rebase nome_branch
```

«Riscrive» la storia

Differenza tra git merge e git rebase

- `git merge`: unisce i branch preservando la cronologia.
- `git rebase`: riapplica i commit del branch di feature sopra l'ultima versione del branch principale.
- Il rebase crea una cronologia più lineare rispetto al merge.



Risoluzione di conflitti durante il merge

- I conflitti di merge si verificano quando due branch modificano la stessa parte di un file.
- GIT evidenzia i conflitti e richiede una risoluzione manuale.
- Una volta risolto il conflitto, bisogna eseguire `git add` e `git commit`.

git reset

Sintassi:

```
git reset --soft <commit>  
oppure  
git reset --mixed <commit>  
oppure  
git reset --hard <commit>
```

- **Cosa fa:** Sposta il puntatore del branch a un commit precedente, eliminando i commit successivi dalla storia locale.
- **Quando usarlo:** Solo su branch locali, mai su rami condivisi (remote) perché altera la cronologia.
- **Opzioni:** --soft (mantiene i file in staging), --mixed (default, toglie da staging), --hard (elimina tutto, anche i file nella working directory).
- **Conseguenze:** Riscrive la storia, potenzialmente creando conflitti per altri sviluppatori.



git revert

Sintassi:

```
git revert
```

Crea un nuovo commit che annulla uno precedente; è molto importante nell'utilizzo in team.

- **Cosa fa:** Crea un nuovo commit che applica l'inverso delle modifiche di un commit precedente.
- **Quando usarlo:** Su branch pubblici/condivisi o quando è fondamentale mantenere una traccia cronologica delle modifiche.
- **Conseguenze:** Non rimuove nulla, aggiunge solo nuovi commit, rendendolo un'operazione sicura.

git push

Sintassi:

`git push origin main`

oppure

`git push -u origin main` //invia i commit locali al repository remoto e crea una relazione di tracciamento (upstream) tra il ramo locale e quello remoto.
Questo permette di usare `git push` o `git pull` in futuro senza specificare i nomi del repository remoto o del ramo

Invia commit al repository remoto



git pull

Sintassi:

`git pull`

oppure

`git pull origin main`

che equivale a:

`git fetch + git merge`

Scarica modifiche senza integrarle

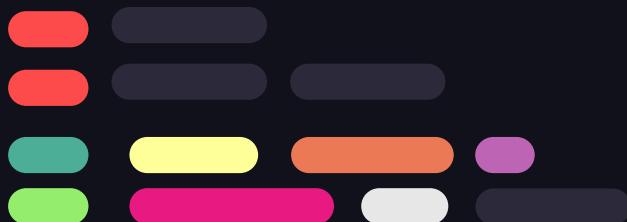
Flussi di branching a confronto

- **GitFlow:** main, develop, feature, release, hotfix
- **GitHub Flow:** main + feature branch → pull request → merge
- **Trunk Based Development:** main sempre stabile, piccoli branch brevi



06 { . .

Collaborazione con
repository remoti



. .



Cosa sono i repository remoti e come funzionano

- Un repository remoto è una versione del progetto ospitata su un server (GitHub, GitLab, Bitbucket).
- Permette la collaborazione tra più sviluppatori.
- I repository locali possono sincronizzarsi con i remoti tramite git push e git pull.

Configurazione di un repository remoto (git remote add, git fetch, git pull, git push)

- Aggiungere un repository remoto: `git remote add origin <URL>`.
- Scaricare aggiornamenti: `git fetch` e `git pull`.
- Inviare le modifiche: `git push`.



Configurazione delle credenziali SSH/HTTPS per GitHub, GitLab e Bitbucket

- **Connessione con HTTPS:** richiede autenticazione via username e token.
- **Connessione con SSH:** configurazione di chiavi SSH per accesso sicuro.

Gestione credenziali moderne in GitHub

- Dal 2021 GitHub non accetta più password per l'HTTPS
- Due modalità di accesso:
- HTTPS + Token personale (PAT)
- SSH + Coppia di chiavi pubblica/privata
- I token sostituiscono la password e garantiscono maggiore sicurezza



Esercizio pratico: configurazione credenziali GitHub

- Creare un Personal Access Token dal proprio profilo GitHub
- Usare il token al posto della password durante git push
- In alternativa, generare una chiave SSH e testarla con ssh -T git@github.com



Creazione di una Pull Request su GitHub/GitLab e codice review

- Creazione di una pull request.
- Revisione del codice prima della fusione.
- Commenti e approvazioni.



Pull Request: esempio reale

- La Pull Request è una richiesta di unione di un branch nel main
- Permette di avviare una revisione del codice
- Consente di aggiungere commenti, approvazioni, suggerimenti
- Favorisce la collaborazione e la qualità del progetto

Esercizio guidato: aprire una PR e fare code review



- Creare un branch e fare commit
- Spingerlo su GitHub
- Aprire una Pull Request
- Simulare revisione con commenti e approvazioni



Lavoro collaborativo e gestione dei permessi su repository remoti

- Assegnazione di ruoli e permessi.
- Protezione del branch main.

Risoluzione di conflitti durante il lavoro in team

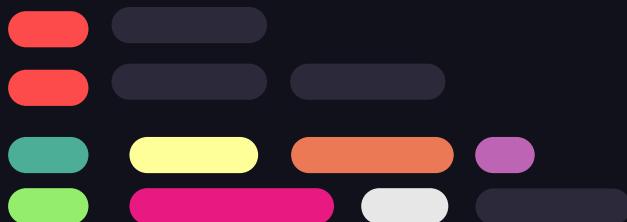


- Conflitti durante il push e il pull.
- Strategie per evitarli.



07 { ..

GIT avanzato: strumenti
e comandi utili



}

..

Gestione delle modifiche temporanee con git stash



- git stash permette di salvare temporaneamente le modifiche senza effettuare un commit.
- Recuperare le modifiche con git stash pop o git stash apply.
- Visualizzare le modifiche salvate con git stash list.
- Eliminare uno stash con git stash drop.



Annullamento delle modifiche con git reset, git revert

- `git reset` riporta i file a uno stato precedente, eliminando commit locali.
- `git revert` annulla un commit creando un nuovo commit di annullamento.
- `git reset --hard` elimina completamente le modifiche (attenzione!).
- `git reset --soft` mantiene le modifiche nello staging.



Recupero di commit cancellati con git reflog

- `git reflog` mostra l'elenco delle azioni recenti eseguite nel repository.
- È possibile recuperare commit cancellati con `git reset --hard <hash>`.
- Utile per recuperare modifiche perse accidentalmente.

Applicazione selettiva di commit con git cherry-pick



- `git cherry-pick <hash>` applica un commit specifico da un branch a un altro.
- Utile per selezionare singole modifiche senza fare un merge completo.
- Aiuta a portare solo le modifiche necessarie senza includere altri commit.

Gestire file e cartelle con `.gitignore`

`.gitignore` permette di escludere file e cartelle dal versioning

Esempi comuni:

- `node_modules/`
- `*.log`
- `.env`

Importante per evitare di salvare file temporanei, build o credenziali



Automazioni con Git Hooks

I Git Hooks sono script che si attivano su eventi specifici

Esempi:

- pre-commit: controlli prima di un commit
- pre-push: blocca il push se non passa un test

Possono automatizzare regole di team e ridurre errori



Ricerca avanzata con git grep e git log --grep

- `git grep <parola>` cerca una stringa all'interno dei file versionati.
- `git log --grep="parola"` cerca commit contenenti una determinata parola.
- `git log -S"parola"` trova commit in cui un termine è stato aggiunto o rimosso.

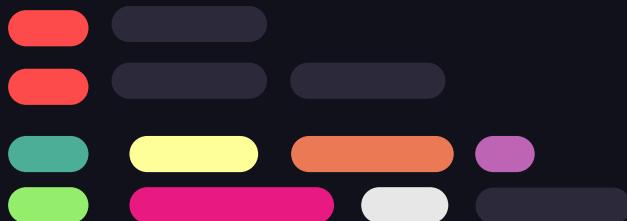
Recupero branch con git reflog

- `git reflog` registra tutte le azioni fatte nel repository
- Permette di recuperare commit o branch cancellati
- Comando tipico: `git reset --hard <hash>`



08 { ..

Simulazione di problemi
reali e risoluzione



}

..



Caso pratico: due utenti lavorano sullo stesso file e si verifica un conflitto

- Due utenti modificano la stessa riga di un file in branch separati.
- Durante il merge, GIT segnala un conflitto.
- Il conflitto deve essere risolto manualmente prima di poter completare il merge.

Debug di un repository: come recuperare un commit errato



- Verifica dello storico con `git log` e `git reflog`.
- Ripristino di un commit cancellato con `git reset` o `git checkout`.
- Uso di `git diff` per capire le modifiche tra versioni.

Workflow GitFlow: come gestire i branch di sviluppo e produzione



- Struttura del workflow GitFlow: branch main, develop, feature, hotfix.
- Branch feature per nuove funzionalità, hotfix per correzioni urgenti.
- Come gestire release stabili con il merge nei branch principali.



Strategie per evitare conflitti e migliorare la collaborazione

- Eseguire git pull prima di iniziare a lavorare su nuove modifiche.
- Creare branch specifici per ogni funzionalità o bugfix.
- Usare git rebase per mantenere una storia più pulita.



Errori comuni in GIT e come risolverli

- Messaggio di errore "fatal: not a git repository"
→ **Soluzione:** git init o verifica della posizione.
- Commit pushato nel branch sbagliato
→ **Soluzione:** git reset e git push --force.
- Merge non riuscito
→ **Soluzione:** risoluzione manuale del conflitto.



Best practices in azienda

- Scrivere messaggi di commit chiari e coerenti
- Dare ai branch nomi descrittivi (feature/..., bugfix/...)
- Non lavorare mai direttamente su main
- Usare Pull Request e code review come passaggio obbligato

Best practices per un utilizzo efficace di GIT

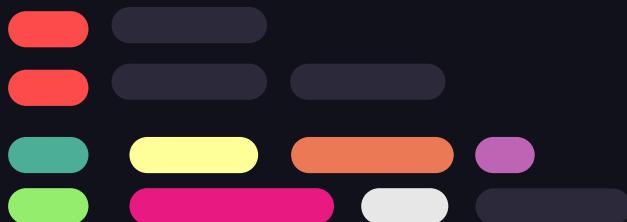


- Scrivere messaggi di commit chiari e descrittivi.
- Usare branch per organizzare il lavoro.
- Non pushare direttamente nel branch main.
- Eseguire pull request e code review per mantenere alta la qualità del codice.



00 { . .

Guida pratica



. .



Installare Git Bash

- Se non hai ancora Git Bash installato, scaricalo e installalo dal sito ufficiale:
 [Download Git](#)
- Dopo l'installazione, apri **Git Bash**.



Creare un repository su GitHub

1. Accedere a [GitHub](#) e fare il login.
2. Cliccare su **New Repository**.
3. Inserire un **nome per il repository**.
4. Selezionare **Public** o **Private** a seconda delle esigenze.
5. **NON** selezionare l'opzione "Initialize this repository with a README" (lo inizializzeremo con Git Bash).
6. Cliccare su **Create repository**.
7. Copiare l'**URL del repository** che verrà mostrato nella pagina successiva.

Configurare Git Bash

Aprire Git Bash e configurare il tuo nome utente e la tua email (necessario per firmare i commit):

```
git config --global user.name "IlTuoNome"  
git config --global user.email iltuoemail@example.com
```

Verifica la configurazione con:

```
git config --list
```

Creare una cartella e inizializzarla con Git

- Aprire Git Bash e spostarsi nella cartella in cui vuoi creare il repository:

```
cd PercorsoDellaCartella
```

(Puoi creare una nuova cartella con `mkdir nome-cartella` e poi entrarci con `cd nome-cartella`).

- Inizializza il repository locale:

```
git init
```

Questo comando creerà una cartella nascosta `.git` all'interno della cartella, trasformandola in un repository Git.

Creare una cartella e inizializzarla con Git

- Crea un file README.md (opzionale, ma consigliato):

```
echo "# Nome del Progetto" > README.md  
git status
```

- Aggiungi tutti i file della cartella alla **staging area**:

```
git add .
```

- Crea il primo commit:

```
git commit -m "Primo commit"
```



Collegare il repository locale a GitHub

Ora devi collegare il repository locale a quello remoto su GitHub. Aggiungi il repository remoto copiando l'URL ottenuto nel Passo 2:

```
git remote add origin https://github.com/TUONOME/NOMEREPOSITORY.git
```

Verifica che il repository remoto sia stato aggiunto correttamente con:

```
git remote -v
```



Collegare il repository locale a GitHub

Carica il repository locale su GitHub con:

```
git push -u origin main
```

Se il branch principale si chiama master, usa:

```
git push -u origin master
```

Ti verrà chiesto di inserire username e password. Se hai attivato l'autenticazione a due fattori, dovrai usare un **token di accesso personale** invece della password.

Recuperare e aggiornare il repository

Se vuoi aggiornare il repository locale con le modifiche fatte su GitHub, usa:

```
git pull origin main
```

Se vuoi caricare nuove modifiche fatte in locale:

```
git add .  
git commit -m "Descrizione delle modifiche"  
git push origin main
```

Clonare un repository GitHub esistente

Se invece vuoi clonare un repository esistente da GitHub al tuo computer, usa:

```
git clone https://github.com/tuo-utente/nome-repository.git
```

Dopo la clonazione, entra nella cartella del progetto con:

```
cd nome-repository
```