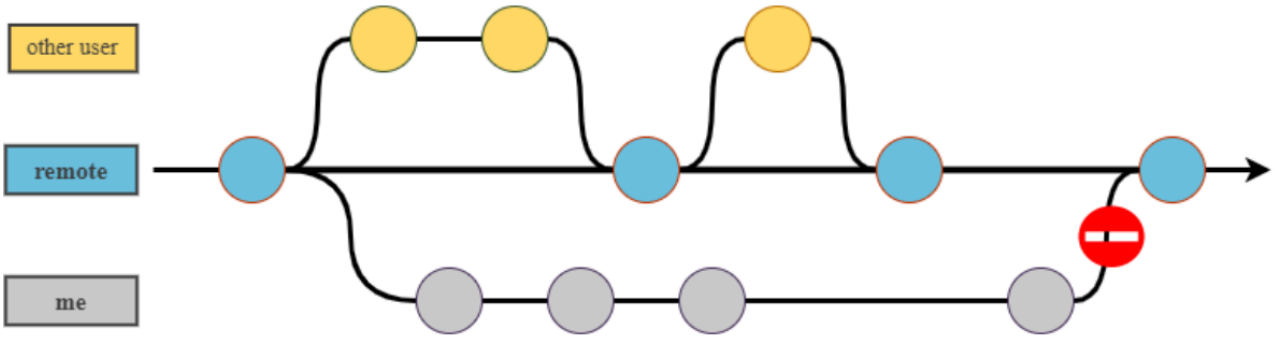
## Varie Terminologie

- *Repository*: set di commit, branch e tag
- *Working copy*: set di file tracciati nella copia locale
- *Commit*: un'istantanea del repository in un determinato momento.  Il commit ha diversi campi tra cui:
  - `data+autore, data+committer`
  - Commento obbligatorio
  - tree: hash di tutti (~) i file nel commit
    Un ***TAG*** è un etichetta per un commit, come ad esempio le versioni, ed è meglio non cancellarli.
    Nello ***STAGING***, il commit può contenere un sottoinsieme delle modifiche, oppure un sottoinsieme di modifiche ad un singolo file.
    Assumendo di fare commit solo dei file interi, è necessario aggiungere alla **staging area** i cambiamenti.

- *Branch*: una linea di sviluppo. È un insieme ordinato di commit. E' la linea di sviluppo in cui è presente un set ordinato di commit collegati in un DAG. Esso punta sempre all'ultimo commit ed inizia dall'ultimo orfano (solitamente il primo commit del repo). Il primo commit crea il primo branch.
  *Permette di lavorare in parallelo a più versioni*.



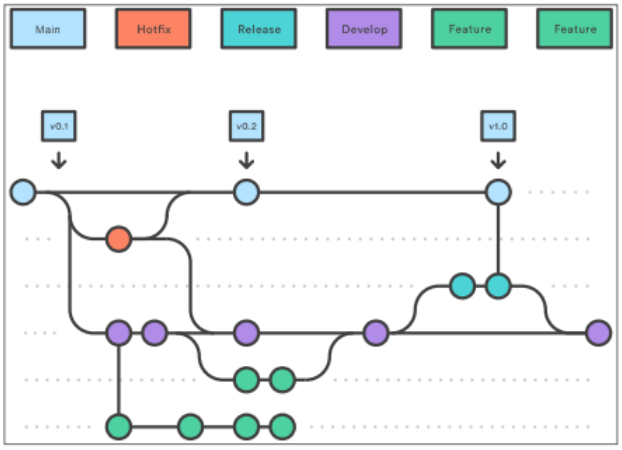

  L'***head*** può puntare ad un commit, branch o tag. Se non punta ad un branch:
  `detached head-> no commit`

- *Merge*: l'azione di fondere due o più branch. Fonde i cambiamenti tra due branch, se si fa merge verso un branch la destinazione contiene tutti e due i cambiamenti e l'origine rimane immutata.
  Esistono varie strategie di merge:
  - ***fast-forward:*** Assumiamo di fare merge di B in A, E' valido nel caso in cui B è una continuazione di A.
    - *Condizione*: Branch unito è diretto discendente di HEAD. (Storia lineare)
    - *Azione*: Git sposta solo il puntatore di HEAD.
    - *Risultato*: Nessun nuovo merge commit.
      **ESEMPIO:** `git checkout master  git merge hotfix # Fast-Forward`
  - ***merge commit:*** crea un nuovo commit con due genitori ed A punta al nuovo commit
  - ***rebase:*** revisionismo storico, ossia modifica la storia in modo che sia lineare ed applica il fast-forward. Ricrea ogni commit non in comune tra A e B dopo l'ultimo in A. I commit originali nel branch, ora spostato, rimangono appesi (dangling).

- ***three-way merge***:
  - *Condizione*: Storie divergenti (commit unici su entrambi i branch).
  - *Punti di Confronto*:
    1. Base Comune (Ancestor).
    2. Versione Locale (HEAD).
    3. Versione Remota (Branch).
  - *Azione*: Git crea un nuovo snapshot combinando le modifiche.
  - *Risultato*: Viene creato un Merge Commit (con due genitori).
    Modifiche in parti diverse del file --> Git le combina automaticamente.
    **Conflitto**: modifiche sulle stesse righe o se un file è stato modificato in un branch ed eliminato nell'altro --> dichiara un conflitto.
- *Tag*: un'etichetta personalizzata allegata a un commit.
- *Fork*: un nuovo repository che è una copia di uno esistente.
- *Pull/Merge request*: una richiesta di unire il codice da un fork o branch al repository/branch padre.

## COMANDI ESSENZIALI

- `git init`: Inizializza un repository.
- `git status`: Visualizza lo stato.
- `git add`: Aggiunge al staging.
- `git commit` -m: Crea un commit.
- `git log`: Visualizza la storia.
- `git branch <nome>`: crea un nuovo puntatore branch.
- `git checkout <nome>`: passa al branch specificato (Sposta HEAD)
- `git checkout -b <nome>`: crea e passa immediatamente al nuovo branch
- `git checkout`: crea un nuovo branch locale con tracciamento automatico del remoto.
- `git fetch`: scarica i dati (commit, oggetti) dal repository remoto **senza** modificare il branch locale.
- `git pull`: scarica il contenuto dal remoto e lo unisce immediatamente nel branch locale corrente. E' l'unione di `git fetch` seguito da `git merge`.
- `git push` : carica i commit locali sul branch remoto e richiede che il branch locale tracci il branch remoto.

| | |
|---|---|
| ```<br>git config --global user.email "you@example.com"<br>git config --global user.name "Your Name"<br>git init<br>git status<br>touch test.txt<br>git add test.txt<br>git status<br>mkdir thedir<br>mv test.txt thedir/<br>git add thedir/<br>git commit -m 'made a dir'<br>``` |  |
| Esempio di comandi Git | Esempio di flow |

## GIT FLOW

Modello di branching rigido per la gestione di rilasci e cicli di sviluppo definiti.
*Vantaggi*: Separazione tra ambiente di produzione, sviluppo, funzionalità e correzioni.

- **MAIN**
  - ***Contenuto***: Solo codice stabile, testato e rilasciato.
  - ***Merge*** in: Solo da `release-* o hotfix-*`.
  - ***Tag***: Ogni merge su `main` riceve un *tag di versione* (es. v1.0).

- **DEVELOP**
  - ***Contenuto***: Cronologia completa delle *funzionalità in sviluppo* (il prossimo rilascio).
  - ***Merge da***: Riceve le nuove *feature* da `feature-*`.

- **FEATURE-**
  - ***Scopo***: Lavoro isolato su una nuova funzionalità.
  - Ramifica da: *develop*.

```
git checkout -b feature-nuova-ui develop
```

  - Unisce in: ***develop***.
  - ***Regola***: non interagisce mai con il main.

- **RELEASE-**
  **Scopo**: Preparazione per il prossimo rilascio (es. release-1.2).
  - ***Ramifica da***: develop.
  - ***Attività***: Solo bug fixing minori e aggiornamento metadata (numero di versione).
  - ***Doppio merge***:
    1. **main**: Per il rilascio in produzione e l'applicazione del tag.
    2. **develop**: Per preservare le correzioni fatte nel branch di rilascio.

- **HOTFIX-**
  - ***Scopo***: Correzione immediata di bug critici trovati in produzione (main).
  - ***Ramifica da***: main (l'ultima versione stabile).

```
git checkout -b hotfix-bug-critico main
```

  - ***Doppio merge***:
    1. **main**: Per deployare subito la correzione.
    2. **develop**: Per garantire che il bug non riappaia nel prossimo rilascio.

## Remotes

- Riferimenti ai branch in repository remoti (DUH!).
- `origin` è il nome remoto predefinito.
- Visualizzazione nel formato: `<remote>/<branch>` (es. origin/main).
- ***Tracking branch***: Branch locali con relazione diretta a un remoto.
  - Facilita l'uso dei comandi `git pull` o `git push`.

***FORGES:*** Un Git remote può essere ospitato da una qualsiasi macchina con sshd.

## STATO DEL REPOSITORY

Il merge viene messo in pausa e Git marca i file in conflitto.

- *Verifica*: git status mostra i file non uniti (Unmerged paths).
- *Contenuto del File (Marcatori)*:

```
<<<<<<< HEAD

// Codice sul branch master

// Codice sul branch iss53
>>>>>>> iss53
```

### CONFLITTI

- Azione Git: Merge in pausa; file marcati come non uniti.
- Risoluzione Manuale:
  1. Aprire il file e rimuovere i ***marcatori di conflitto*** `(<<<<<<<, = = = =, >>>>>>>)`.
  2. `git add <file-risolto>`: Marca il file come risolto.
  3. `git commit`: Completa il merge commit.
- Tool: git mergetool per interfaccia grafica.

## UNDO

### Annullare in Staging

- **Problema**: Hai eseguito `git add` su un file, ma non vuoi committarlo.
- **Soluzione**: Rimuovi il file dall'area di staging. Le modifiche restano nella directory di lavoro.

```
git reset HEAD <file>
```

- **doppio trattino (--)**: segue il nome del file, non un'opzione o un branch.

### Annullare le modifiche locali

- *Problema*: Hai modificato un file e vuoi ripristinare la versione dell'ultimo commit
- *Soluzione*: Scarta le modifiche locali non committate.

```
git checkout -- <file>
```

### Annullare un commit pubblicato

- ***Goal***: Annullare un commit che è già stato *pubblicato* e condiviso.
- ***Motivo***: Crea un nuovo *commit opposto* che annulla le modifiche. La cronologia *non viene riscritta*, rendendolo sicuro per i repository condivisi.

```
git revert <hash-commit>
```

### Annullare un commit locale

- Goal: Annullare un commit che non è stato ancora pubblicato.
- Motivo: Riscrive la storia locale spostando il puntatore HEAD e il branch.

```
git reset --soft HEAD~1
```

### Opzioni e riferimenti

- ***HEAD~1***: Indica il commit direttamente precedente a HEAD. (HEAD~3 indica il 30 etc).
- **-soft**: Rimuove l'ultimo commit. Mantiene le modifiche nell'area di staging.
- - **-hard**: Rimuove l'ultimo commit e cancella completamente le modifiche.
  - *Uso*: Ripulire completamente il branch. ATTENZIONE: Modifiche perse!

## RISCRIVERE L'ULTIMO COMMIT

- ***Azione***: Rimpiazza l'ultimo commit con uno nuovo. La storia viene riscritta.

```
git commit --amend
```