5. Operazioni di Base di Git e GitHub

Questo capitolo fornisce una guida dettagliata all'utilizzo pratico di Git, il sistema di controllo di versione distribuito più diffuso, e introduce GitHub come piattaforma di hosting per i repository Git e per la collaborazione allo sviluppo software.

5.1 Concetti Fondamentali di Git

Prima di addentrarci nelle operazioni, è utile ripassare i concetti chiave di Git che sono stati introdotti nel Capitolo 1. Git gestisce il progetto attraverso **tre stati** principali per i file:

1. **Working Directory (Working Tree)**: La copia dei file del progetto sul tuo sistema **locale**, dove apporti le modifiche.
2. **Staging Area (Index)**: Un'area intermedia dove **prepari le modifiche che vuoi includere** nel prossimo commit. Non tutte le modifiche nel working directory devono essere committate; la staging area ti permette di selezionare quali includere.
3. **Repository (Local Repository)**: Il **database locale** di Git che contiene **tutti i commit**, la cronologia del progetto e i metadati.

Questo flusso di lavoro permette un controllo granulare su quali modifiche vengono salvate nella cronologia del progetto.

5.2 Configurazione Iniziale di Git

Prima di iniziare a usare Git, è necessario configurare alcune informazioni di base che verranno associate ai tuoi commit.

5.2.1 Comando `git config`

Il comando `git config` è usato per **impostare** le opzioni di **configurazione** di Git. Queste configurazioni possono essere a diversi livelli:

- **Sistema (`-system`)**: Applicabile a tutti gli utenti del sistema.
- **Globale (`-global`)**: Applicabile a tutti i repository creati dall'utente corrente. Questa è la configurazione più comune per nome utente ed email.
- **Locale (`-local`)**: Specifico per il repository corrente.

Configurazione dell'utente (globale):
È fondamentale impostare il tuo nome utente e la tua email, poiché queste informazioni verranno incluse in ogni commit che effettuerai.

```
git config --global user.name "Il Tuo Nome"
git config --global user.email "tua.email@example.com"
```

Puoi anche configurare l'editor di testo predefinito che Git userà per i messaggi di commit e altre operazioni:

```
git config --global core.editor "vim" # o "nano", "code --wait", ecc.
```

Per visualizzare tutte le configurazioni globali:

```
git config --global --list
```

5.3 Creazione e Gestione del Repository Locale

5.3.1 Inizializzazione di un Nuovo Repository (`git init`)

Per iniziare a tracciare un nuovo progetto con Git, devi inizializzare un **repository** nella cartella del progetto.

```
cd /percorso/alla/tua/cartella/progetto
git init
```

```
```

Questo comando crea una sottocartella `.git` nascosta all'interno della directory corrente. Questa cartella contiene tutti i file necessari per il repository Git, inclusa la cronologia dei commit, i branch e le configurazioni.

5.3.2 Aggiunta di File alla Staging Area (`git add`)

Dopo aver creato o modificato dei file nel tuo working directory, devi indicare a Git quali di queste modifiche vuoi includere nel prossimo commit. Questo si fa usando il comando `git add`.

```
git add nome_file.txt        # Aggiunge un singolo file
git add cartella/            # Aggiunge tutti i file in una cartella
git add .                    # Aggiunge tutti i file modificati/nuovi nella directory corrente e sottodirectory
```

L'operazione `git add` **sposta** le modifiche **dal working directory alla staging area**. I file nella staging area sono pronti per essere committati.

5.3.3 Salvataggio delle Modifiche (`git commit`)

Una volta che le modifiche desiderate sono nella staging area, puoi registrarle nella cronologia del repository locale come un nuovo commit.

```
git commit -m "Messaggio descrittivo del commit"
```

Il flag `-m` permette di specificare il **messaggio di commit** direttamente dalla riga di comando. Un buon messaggio di commit è conciso ma descrittivo, spiegando *cosa* è stato fatto e *perché*.

Se ometti `-m`, Git aprirà l'editor di testo configurato (es. Vim, Nano) per permetterti di scrivere un messaggio più lungo.

5.3.4 Visualizzazione dello Stato del Repository (`git status`)

Il comando `git status` è fondamentale per capire lo **stato attuale del tuo working directory** e della staging area. Ti mostra **quali file sono stati modificati**, quali sono nella **staging area** e quali **non sono tracciati**.

```
git status
```

Output tipico di `git status`:

- `On branch master`: Indica il branch corrente.
- `No commits yet`: Se è un nuovo repository senza commit.
- `Untracked files:`: File che Git non sta tracciando.
- `Changes to be committed:`: File nella staging area, pronti per il commit.
- `Changes not staged for commit:`: File modificati ma non ancora aggiunti alla staging area.

5.3.5 Visualizzazione della Cronologia (`git log`)

Il comando `git log` mostra la cronologia dei commit nel repository.

```
git log
```

Output di `git log`:
Ogni commit viene visualizzato con:

- Un hash SHA-1 univoco (es. `commit b82f7567961ba13b1794566dde97dda1e501cf88`)
- L'autore del commit (`Author: Il Tuo Nome <tua.email@example.com>`)
- La data e l'ora (`Date: ...`)
- Il messaggio di commit

Opzioni utili per `git log`:

- `git log --oneline`: Mostra un **riassunto compatto** di ogni commit su una singola riga.
- `git log --graph --oneline --decorate`: Mostra la **cronologia** come un **grafo ASCII**, utile per visualizzare i **branch e i merge**.
- `git log -p`: Mostra le **differenze** (**patch**) introdotte da ogni commit.

5.3.6 Visualizzazione delle Differenze (`git diff`)

Il comando `git diff` mostra le **differenze** tra diversi **stati** dei file.

```
git diff                # Mostra le differenze tra working directory e staging area
git diff --staged       # Mostra le differenze tra staging area e l'ultimo commit
git diff HEAD           # Mostra le differenze tra working directory e l'ultimo commit
git diff <commit1> <commit2> # Mostra le differenze tra due commit specifici
```

`git diff` è uno strumento potente per **rivedere le modifiche prima di committarle** o per capire cosa è cambiato tra diverse versioni.

5.3.7 Annullamento delle Modifiche (`git reset` e `git restore`)

Git offre diversi modi per annullare le modifiche, a seconda dello stato in cui si trovano i file.

- `git reset`:
  - `git reset --hard <commit-hash>`: **Pericoloso!** Riporta il working directory e la staging area allo stato di un commit specifico, **eliminando tutte le modifiche non committate**. Usare con estrema cautela.
  - `git reset <commit-hash>`: Sposta `HEAD` e il branch al commit specificato, ma **mantiene le modifiche nel working directory e nella staging area** (le modifiche sono "un-staged").
  - `git reset --soft <commit-hash>`: Sposta `HEAD` e il branch, ma **lascia le modifiche nella staging area** (le modifiche sono "staged").
- `git restore`: (Introdotto in Git 2.23, più intuitivo di `git reset` per alcune operazioni)
  - `git restore <file>`: Annulla le modifiche non committate **in un file**, riportandolo allo stato dell'**ultimo commit** o della staging area (se il file è staged).
  - `git restore --staged <file>`: **Rimuove** un file dalla **staging area** (lo "un-stages"), **mantenendo** le modifiche nel **working directory**.

5.4 Branching e Merging

Il branching è una delle funzionalità più potenti di Git, che permette agli sviluppatori di lavorare su **diverse linee di sviluppo in parallelo**.

5.4.1 Creazione di un Branch (`git branch`)

Un branch è un **puntatore mobile a un commit**. Il branch `master` (o `main`) è il branch predefinito.

```
git branch nuova_funzionalita       # Crea un nuovo branch chiamato "nuova_funzionalita"
git branch                          # Elenca tutti i branch locali
```

5.4.2 Spostamento tra Branch (`git checkout`)

Per passare a un altro branch, si usa `git checkout`. Questo a**ggiorna il working directory e la staging area** per riflettere lo stato del **branch di destinazione**.

```
git checkout nuova_funzionalita     # Sposta HEAD al branch "nuova_funzionalita"
git checkout -b nuovo_branch        # Crea un nuovo branch E ci si sposta immediatamente
```

Come menzionato nel Capitolo 1, `git checkout <commit-hash>` ti porta in uno stato di "detached HEAD"

5.4.3 Unione dei Branch (`git merge`)

Dopo aver completato il lavoro su un branch (es. `nuova_funzionalita`), puoi integrarlo nel branch principale (es. `master` o `main`) usando `git merge`.

```
git checkout master              # Spostati sul branch di destinazione
git merge nuova_funzionalita     # Unisci il branch "nuova_funzionalita" in "master"
```

Tipi di Merge:

- **Fast-Forward Merge**: Se il branch di destinazione non ha avuto nuovi commit da quando è stato creato il branch da unire, Git semplicemente sposta il puntatore del branch di destinazione in avanti
- **Three-Way Merge**: Se **entrambi** i branch hanno avuto **nuovi commit indipendenti**, Git crea un **nuovo commit** di merge che **combina** le modifiche di entrambi i rami

5.4.4 Risoluzione dei Conflitti di Merge

I conflitti di merge si verificano quando Git non riesce a unire automaticamente le modifiche perché due branch hanno **modificato la stessa parte dello stesso file in modi diversi**.

Quando si verifica un conflitto:

1. Git **interrompe** l'operazione di merge.
2. I file in conflitto contengono marcatori speciali (`<<<<<<<`, `=======`, `>>>>>>>`) che indicano le sezioni in conflitto.
3. Devi **modificare manualmente** il file per risolvere il conflitto, scegliendo quali modifiche mantenere.
4. Dopo aver risolto il conflitto, aggiungi il file alla staging area (`git add <file_in_conflitto>`).
5. Completa il merge con un nuovo commit (`git commit`).

5.5 Lavorare con i Repository Remoti

I repository remoti sono versioni del tuo progetto ospitate su server esterni (es. GitHub, GitLab, Bitbucket). Permettono la collaborazione e servono come backup centralizzato.

5.5.1 Clonazione di un Repository Remoto (`git clone`)

Per ottenere una copia locale di un repository remoto esistente, si usa `git clone`. Questo comando non solo scarica tutti i file, ma anche l'intera cronologia dei commit.

```
git clone https://github.com/utente/repo.git
```

Questo creerà una nuova cartella con il nome del repository e al suo interno un repository Git locale, già configurato per comunicare con il remoto.

5.5.2 Aggiunta di un Remote (`git remote add`)

Se hai un repository locale esistente e vuoi collegarlo a un remoto, puoi usare `git remote add`.

```
git remote add origin https://github.com/utente/repo.git
```
```origin` è il nome convenzionale per il repository remoto principale.

5.5.3 Caricamento delle Modifiche (`git push`)

Per inviare i tuoi commit locali al repository remoto, si usa `git push`.

```bash
git push origin master          # Invia i commit del branch master al remote "origin"
```

La prima volta che si esegue il push di un nuovo branch, potrebbe essere necessario usare il flag `-u` (o `--set-upstream`):

```
git push -u origin nome_branch   # Imposta il branch locale per tracciare il branch remoto
```

5.5.4 Scaricamento delle Modifiche (`git fetch` e `git pull`)
```

Per ottenere le modifiche dal repository remoto:

- `git fetch`: Scarica i commit dal repository remoto nel tuo repository locale, ma **non li applica** al tuo working directory o branch corrente. Ti permette di **vedere le modifiche remote prima di integrarle**.

  ```
  git fetch origin
  ```

- `git pull`: È una scorciatoia che esegue `git fetch` seguito da `git merge`. Scarica le modifiche dal remoto e le unisce immediatamente nel tuo branch corrente.

  ```
  git pull origin master
  ```

5.6 File `.gitignore`

Il file `.gitignore` è un file di testo che **elenca i file e le cartelle che Git deve ignorare**, cioè non tracciare nei commit. È essenziale per evitare di committare file temporanei, file di configurazione specifici dell'ambiente locale, dipendenze di build, o output generati.

- Ogni riga nel `.gitignore` specifica un pattern per i file o le cartelle da ignorare.
- Esempi comuni:
  - `.log`: Ignora tutti i file `.log`.
  - `/temp/`: Ignora la cartella `temp` nella radice del repository.
  - `build/`: Ignora la cartella `build` e il suo contenuto.
  - `!important.txt`: Eccezione, non ignorare `important.txt` anche se un pattern precedente lo includerebbe.

5.7 Gestione dei Caratteri di "Fine Linea" (Line Endings)

I diversi sistemi operativi utilizzano caratteri diversi per indicare la fine di una riga:

- Windows: Carriage Return + Line Feed (`CRLF`, `\r\n`)
- Unix/Linux/macOS: Line Feed (`LF`, `\n`)

Questo può causare problemi di compatibilità quando sviluppatori su sistemi diversi collaborano. Git ha delle configurazioni per gestire automaticamente queste differenze:

- `git config --global core.autocrlf true`: (Consigliato per Windows) Git convertirà `CRLF` in `LF` quando si committa e `LF` in `CRLF` quando si fa il checkout.
- `git config --global core.autocrlf input`: (Consigliato per Linux/macOS) Git convertirà `CRLF` in `LF` quando si committa, ma non farà conversioni al checkout.
- `git config --global core.autocrlf false`: Nessuna conversione automatica.

(Informazioni basate su conoscenza generale di Git, non direttamente dal PDF ma rilevanti per il contesto)

5.8 GitHub: Hosting per Repository Git e Servizi Collaborativi

GitHub è una piattaforma web che fornisce **hosting per repository Git** e una suite di strumenti per lo sviluppo collaborativo. È diventato il servizio di hosting di repository più popolare al mondo.

5.8.1 Funzionalità Principali di GitHub

- **Hosting di Repository**: Permette di **archiviare** i tuoi repository Git in cloud, rendendoli accessibili da qualsiasi luogo e fungendo da backup.
- **Collaborazione Facile**: Facilita il lavoro di squadra attraverso funzionalità come:
  - **Pull Requests (PRs)**: Un meccanismo per proporre modifiche a un repository. Permette ad altri sviluppatori di rivedere il codice, commentare e discutere prima che le modifiche vengano unite nel branch principale.
  - **Issues**: Un sistema di tracciamento dei bug, delle richieste di funzionalità e delle attività.
  - **Wiki**: Per la documentazione del progetto.
  - **Progetti (Projects)**: Strumenti Kanban-style per la gestione delle attività.

- **Forking**: La possibilità di creare una **copia personale** di un **repository di qualcun altro**. Questo ti permette di sperimentare e apportare modifiche senza influenzare il repository originale. Se le tue modifiche sono valide, puoi proporle al repository originale tramite una Pull Request.
- **Integrazione Continua/Deployment Continuo (CI/CD)**: Con GitHub Actions, è possibile automatizzare **test, build e deployment** del codice direttamente dalla piattaforma.
- **Autenticazione**: GitHub supporta diversi metodi di autenticazione per l'accesso ai repository remoti:
  - **HTTPS con Personal Access Token (PAT)**: Un token generato su GitHub che sostituisce la password per le operazioni Git via HTTPS. Consigliato per utenti Windows senza shell Unix.
  - **SSH (Secure Shell)**: Richiede l'autenticazione tramite una **coppia di chiavi** pubblica/privata. La chiave pubblica viene caricata su GitHub, e la chiave privata (segreta) viene usata localmente per l'autenticazione. Consigliato per utenti Linux/macOS e per chi ha un'installazione SSH funzionante.

5.8.2 Configurazione di OpenSSH per GitHub

Per utilizzare l'autenticazione SSH con GitHub:

1. **Genera una nuova coppia di chiavi (se non ne hai una)**:

   ```
   ssh-keygen
   ```

   Puoi confermare le opzioni predefinite. Se scegli una password vuota, la chiave privata sarà archiviata senza crittografia (questo comporta problemi di sicurezza, quindi è consigliato usare una password)

2. **Ottieni la chiave pubblica**:

   ```
   cat ~/.ssh/id_rsa.pub
   ```

   Questo comando stamperà la tua chiave pubblica, che assomiglia a `ssh-rsa AAAAB3Nza...`

3. **Aggiungi la chiave pubblica a GitHub**:
   - Accedi al tuo account GitHub.
   - Vai su `Settings` → `SSH and GPG keys`.
   - Clicca su `New SSH key`.
   - Fornisci un titolo che ti permetta di identificare la chiave (es. "My Laptop Key").
   - Incolla la chiave pubblica che hai ottenuto nel campo "Key".
   - Salva la chiave.

Dopo questi passaggi, potrai utilizzare l'URL SSH del tuo repository (es. `git@github.com:owner/repo.git`) per le operazioni Git, e l'autenticazione avverrà automaticamente tramite la tua chiave SSH.