GIT a supporto dei tecnici UNIV

18 Novembre 2025

```
const whoami = {
  firstName: 'Marco',
  lastName: 'Spasiano',
  age: 52,
  city: 'Napoli',
  company: {
    acronym: 'CNR',
    description: 'Consiglio Nazionale delle Ricerche',
    office: 'UFFICIO AGENDA DIGITALE E PROCESSI',
    profile: 'Primo Tecnologo'
  },
  contacts: {
    email: 'marco.spasiano@cnr.it',
    github: 'https://github.com/mspasiano'
  }
}
```

Trasferimento e Condivisione della conoscenza

La storia dell’umanità è, prima di tutto, la storia del trasferimento della conoscenza. Dalla parola orale alla scrittura, dalla stampa fino al mondo digitale, ogni progresso è nato dalla capacità di trasmettere, condividere e trasformare ciò che sappiamo.

Nel Medioevo la conoscenza passava dai maestri agli apprendisti; nel Rinascimento circolava tra le accademie e le corti; oggi viaggia in tempo reale attraverso reti globali, università e imprese.

Nel corso dei secoli, anche i supporti utilizzati per trasferire la conoscenza si sono evoluti profondamente. All’inizio erano la tradizione orale e la memoria collettiva, strumenti fragili ma potenti, grazie ai quali miti, riti e tecniche venivano tramandati di generazione in generazione.

Poi arrivò la scrittura, che rese possibile fissare il sapere nel tempo: dalle tavolette d’argilla alle pergamene, fino ai manoscritti custoditi nei monasteri.

Con l’invenzione della stampa a caratteri mobili, la conoscenza uscì dalle biblioteche e divenne patrimonio condiviso, contribuendo in modo decisivo alla diffusione delle idee scientifiche e umanistiche del Rinascimento.

Questo flusso continuo di saperi ha portato tre vantaggi fondamentali:

- **INNOVAZIONE .... CI PERMETTE DI COSTRUIRE SU ESPERIENZE GIÀ ACQUISITE, ACCELERANDO IL PROGRESSO SCIENTIFICO E TECNOLOGICO.**
- **EFFICIENZA .... CONDIVIDERE BUONE PRATICHE RIDUCE GLI ERRORI E MIGLIORA LA PRODUTTIVITÀ DELLE ORGANIZZAZIONI.**
- **CRESCITA .... COLLETTIVA DIFFONDERE CONOSCENZA SIGNIFICA DIFFONDERE OPPORTUNITÀ: FAVORISCE SVILUPPO ECONOMICO, DIALOGO CULTURALE E COESIONE SOCIALE.**

Oggi, nel pieno dell’era digitale, il trasferimento della conoscenza è ancora più strategico. Non basta accumulare informazioni: serve la capacità di trasformarle in valore, di adattarle e di farle circolare tra persone, istituzioni e generazioni.

Linus Torvalds

La nascita di GIT (2005)

I motivi che hanno spinto Linus Torvalds a creare Git nel 2005 sono stati principalmente legati alla crisi del sistema di controllo versione che il kernel Linux stava utilizzando.

Il Problema con BitKeeper (2005)

BitKeeper Crisis: Fino al 2005, il progetto del kernel Linux utilizzava BitKeeper, un sistema di controllo versione distribuito proprietario.

La società BitMover aveva concesso una licenza gratuita per progetti open source, ma nell'aprile 2005 **revocò questa licenza gratuita** dopo dispute riguardanti il reverse engineering del protocollo da parte di alcuni sviluppatori della community Linux.

Le Frustrazioni di Linus

Sistemi Esistenti Inadeguati

- CVS: Troppo lento e centralizzato, non gestiva bene i merge
- Subversion: Ancora centralizzato, non abbastanza veloce per un progetto delle dimensioni del kernel Linux
- Altri sistemi distribuiti: Non esistevano o erano troppo complessi/lenti

Le Critiche ai Sistemi Esistenti

Linus era particolarmente critico verso:

CVS/Subversion:

- "CVS è l'esempio di cosa NON fare"
- Troppo lento per operazioni su larga scala
- Merge problematici
- Modello centralizzato inadeguato

Altri DVCS(Distributed Version Control System) dell'epoca:

- **Monotone**: Troppo lento, design over-engineered
- **Darcs**: Problemi di performance con repository grandi
- **Bazaar**: Non esisteva ancora in forma utilizzabile

Requisiti Specifici del Kernel Linux

- Scale massive: Migliaia di sviluppatori, milioni di righe di codice
- Merge frequenti: Centinaia di patch al giorno da integrare
- Velocità: Operazioni che dovevano completarsi in secondi, non minuti
- Integrità: Garanzia assoluta che i dati non fossero corrotti
- Workflow distribuito: Sviluppatori in tutto il mondo senza server centrale

La "Crisi" del Weekend

Timeline critica:

```
- 3 Aprile 2005: BitMover annuncia la fine della licenza gratuita
- 6-7 Aprile 2005: Linus inizia a sviluppare Git
- 7 Aprile 2005: Primo commit di Git
- 16 Aprile 2005: Git si auto-ospita (Git gestisce il proprio codice sorgente)
- 26 Luglio 2005: Primo kernel Linux gestito con Git
```

Filosofia di Design di Git

Linus aveva requisiti molto specifici:

Performance

> *"Git is designed to be very fast. All operations should complete in a few seconds at most"*

Integrità dei Dati

- Ogni oggetto è identificato dal suo hash SHA-1
- Impossibile corrompere file senza accorgersene
- Verifica automatica dell'integrità

Semplicità Concettuale

- Poche operazioni primitive ma potenti
- Modello di dati semplice (blob, tree, commit, tag)

Supporto per Workflow Non-lineari

- Branch rapidi e economici
- Merge intelligente e automatico
- Supporto per migliaia di branch paralleli

Il Fattore Personalità

Pragmatismo Estremo

Linus voleva qualcosa che "funzionasse semplicemente" senza filosofie complicate:

> "*I'm an engineer. I see a problem and I fix it*"

Controllo Totale

Dopo l'esperienza con BitKeeper, Linus voleva:

- Nessuna dipendenza da software proprietario
- Controllo completo degli algoritmi e delle decisioni di design
- Garanzia che non si ripetesse mai più una "crisi BitKeeper"

L'Urgenza della Situazione

La community Linux aveva bisogno di una soluzione **immediata**:

- Il successivo rilascio del kernel (2.6.12) era imminente
- Migliaia di patch in attesa di essere integrate
- Impossibilità di tornare a sistemi primitivi come patch e tar

Risultato

Git fu sviluppato in tempo record:

- **2 settimane** per la prima versione funzionante
- **3 mesi** per diventare il sistema ufficiale del kernel Linux
- **Meno di 1 anno** per diventare lo standard de facto per progetti open source

La Filosofia "Do One Thing Well"

Linus applicò la filosofia Unix anche a Git:

- Ogni comando fa una cosa specifica molto bene
- Componibilità: i comandi si combinano per operazioni complesse
- Efficienza: ottimizzato per le operazioni più comuni

Citazione famosa di Linus:

> *"I really never wanted to do source control management at all and felt that it was just about the least interesting thing in the computing world, but somebody had to do it."*

IN CONCLUSIONE

La creazione di Git fu quindi una **necessità pratica urgente** più che una passione per i sistemi di controllo versione, ma il risultato fu rivoluzionario per l'intero mondo dello sviluppo software.

Problemi risolti

- Gestione versioni distribuite
- Collaborazione in team
- Tracciabilità delle modifiche

Git - Concetti Fondamentali

Concetti Base

- **Repository**: cartella "intelligente" con cronologia
- **Commit**: snapshot del lavoro
- **Staging Area**: area di preparazione
- **Working Directory**: cartella di lavoro
- **Branch**: linee di sviluppo parallele

Git - Repository

Un repository Git (o repo) è lo spazio in cui viene memorizzato e gestito il codice sorgente di un progetto, insieme alla storia completa delle modifiche.

In sintesi:

- È un archivio che contiene file, cartelle e versioni del progetto nel tempo
- Permette a più sviluppatori di collaborare, condividere modifiche, ripristinare versioni precedenti e gestire rami di sviluppo (branch)

Un repository può essere:

- Locale → sul computer dello sviluppatore (git init)
- Remoto → su una piattaforma come GitHub, GitLab o Bitbucket, per la collaborazione online.
- 💡 In breve un repository Git è la memoria storica e collaborativa del codice di un progetto.

Git - Commit

Un git commit è un’istantanea (snapshot) dello stato del progetto in un determinato momento.

In pratica:

- Registra le modifiche apportate ai file nel repository
- Include un messaggio descrittivo che spiega cosa è stato cambiato
- Diventa parte della cronologia del progetto, permettendo di tornare indietro o confrontare versioni
- 💡 In breve: Un commit è come un “salvataggio” ufficiale del progetto nel tempo, con autore, data e descrizione delle modifiche

Git - Staging Area

La staging area (o area di preparazione) è una zona intermedia di Git dove vengono raccolte le modifiche prima di confermarle con un commit.

In pratica:

- Permette di scegliere quali file o modifiche includere nel prossimo commit
- Funziona come un “piano di lavoro temporaneo” tra il working directory e il repository
- 💡 In breve: La staging area è l’area in cui prepari con precisione ciò che verrà salvato nel prossimo commit

Git - Working Directory

La working directory è la cartella del progetto sul tuo computer dove lavori sui file tracciati da Git.

In pratica:

- Contiene la versione attuale dei file del repository
- È l’area in cui modifichi, aggiungi o elimini file prima di metterli in staging o fare un commit
- 💡 In breve: La working directory è lo spazio di lavoro locale in cui apporti le modifiche al progetto gestito da Git

Git - Branch

Un branch in Git è un ramo di sviluppo indipendente che consente di lavorare su nuove funzionalità o correzioni senza modificare il codice principale.

In pratica:

- Ogni branch rappresenta una linea separata di sviluppo
- Puoi creare, unire o eliminare branch per gestire versioni o funzionalità diverse del progetto
- 💡 In breve: Un branch è una copia del codice su cui puoi lavorare in parallelo, senza influenzare il ramo principale (main o master).

Setup dell'Ambiente

🛠️ **HANDS-ON**: Preparazione workstation

- Installazione GIT sui notebook dei partecipanti

Windows

1. Vai su https://git-scm.com/download/win
2. Scarica il file Git-<version>.exe
3. Esegui l’installer e segui la procedura guidata:
    - Mantieni le opzioni predefinite consigliate.
    - Seleziona *“Git Bash Here”* per aggiungere Git al menu contestuale.
4. Al termine, apri **Git Bash** e verifica:

```
git --version
```

macOS 

Installa Homebrew se non lo hai già, e poi installa Git:

```
brew install git
```

Metodo 2 – Tramite Xcode Command Line Tools

```
xcode-select --install
```

Verifica l’installazione:

```
git --version
```

Linux 

Ubuntu / Debian

```
sudo apt update
sudo apt install git -y
```

Fedora

```
sudo dnf install git -y
```

CentOS / RHEL

```
sudo yum install git -y
```

Arch Linux

```
sudo pacman -S git
```

Verifica l’installazione:

```
git --version
```

Configurazione iniziale:

```
git config --global user.name "Nome Cognome"
git config --global user.email "email@unich.it"
git config --global init.defaultBranch main
```

- Setup editor preferito (VS Code, nano, vim)

```
git config core.editor "vim"
```

Il *--global* fa si che imposta **Nano** come editor predefinito per tutti i repository Git

```
git config --global core.editor "nano"
```

Per VS Code invece:

```
git config --global core.editor "code --wait"
# Se si vuole usare VS Code anche per risolvere conflitti
git config --global merge.tool vscode
git config --global mergetool.vscode.cmd "code --wait $MERGED"
git config --global diff.tool vscode
git config --global difftool.vscode.cmd "code --wait --diff $LOCAL $REMOTE"
```

Il Primo Repository

🛠️ HANDS-ON: Creazione repository

```
# Creiamo una cartella per documentazione IT
mkdir doc-procedure-it
cd doc-procedure-it
git init

# Primo file: procedura backup
echo "# Procedure di Backup Server" > backup-procedure.md
echo "## Backup giornaliero" >> backup-procedure.md
echo "1. Verifica spazio disco" >> backup-procedure.md

git add backup-procedure.md
git commit -m "Prima versione procedura backup"
```

Lavorare con i File

🛠️ HANDS-ON: Modifiche e commit

```
# Aggiungiamo un file binario (simuliamo un PDF)
cp /path/to/sample.pdf checklist-backup.pdf
git add checklist-backup.pdf

# Modifichiamo il file esistente
echo "2. Esecuzione script backup.sh" >> backup-procedure.md
echo "3. Verifica log errori" >> backup-procedure.md

git add backup-procedure.md
git commit -m "Aggiunta checklist e nuovi passi procedura"

# Vediamo la storia
git log --oneline
git diff HEAD~1 HEAD
```

Gestione Avanzata Locale

Navigare nella Storia

🛠️ HANDS-ON: Esplorare i commit

```
# Vediamo cosa è cambiato
git log --stat
git log --graph --oneline

# Torniamo indietro per vedere una versione precedente
git checkout HEAD~1
cat backup-procedure.md

# Torniamo al presente
git checkout main

# Correggiamo l'ultimo commit
echo "4. Notifica completamento backup" >> backup-procedure.md
git add backup-procedure.md
git commit --amend -m "Procedura backup completa con notifiche"
```

Gestione dei Branch

🛠️ HANDS-ON: Creazione e uso branch

```
# Creiamo un branch per una nuova procedura
git checkout -b procedura-restore

# Nuovo file per restore
echo "# Procedure di Restore Server" > restore-procedure.md
echo "1. Identificazione backup da ripristinare" >> restore-procedure.md
git add restore-procedure.md
git commit -m "Inizio procedura restore"

# Torniamo su main e vediamo la differenza
git checkout main
ls  # restore-procedure.md non c'è
git checkout procedura-restore
ls  # restore-procedure.md c'è

# Merge del branch
git checkout main
git merge procedura-restore
```

Collaborazione e Strumenti Avanzati

Repository Remoti

🛠️ **HANDS-ON**: Personal Access Token su GitHub

- Vai su https://github.com/settings/tokens
- Clicca su “Generate new token (classic)”
- Dai un nome al token (es. git-training)
- Imposta la durata (es. 7 giorni)
- Seleziona i permessi minimi:

✅ repo → per gestire repository privati/pubblici

Clicca Generate token

Copia il token una sola volta – non potrai rivederlo!

Lavoro Distribuito

🛠️ HANDS-ON: Cloniamo repository su GitHub

```
# Cloniamo un repository esistente su GitHub
git clone https://github.com/training-it-unich/esempio-documentazione.git
cd esempio-documentazione

# Esploriamo il repository
git status
git log --oneline
git remote -v

# Creiamo un nuovo repo sulla nostra organizzazione GitHub
# Che chiamiamo doc-procedure-it (tramite interfaccia web)

# Aggiungiamo il nostro repo locale come remoto
cd ../doc-procedure-it
git remote add origin https://<TOKEN>@github.com/training-it-unich/doc-procedure-it.git

# Push del nostro lavoro
git push -u origin main
```

Collaborazione Base

🛠️ HANDS-ON: Push e Pull

```
# Simuliamo lavoro di un collega (dal docente)
# Modifica da interfaccia web GitHub

# I partecipanti fanno pull
git pull origin main

# Ogni partecipante fa una modifica locale
echo "## Procedura di Monitoring" >> backup-procedure.md
echo "- Controllo stato servizi ogni 30 min" >> backup-procedure.md

git add backup-procedure.md
git commit -m "Aggiunta sezione monitoring"

# Push delle modifiche
git push origin main
```

Gestione Conflitti

Generazione e Risoluzione Conflitti

🛠️ HANDS-ON: Conflitti reali

```
# Il docente modifica lo stesso file dalla web interface
# I partecipanti modificano localmente la stessa riga

echo "## Backup Schedulati ore 02:00" >> backup-procedure.md
git add backup-procedure.md
git commit -m "Orario backup specificato"

# Tentativo di push (fallirà)
git push origin main

# Pull per scaricare le modifiche remote
git pull origin main
# CONFLITTO!

# Risoluzione conflitto con editor
# Spiegazione dei marker <<<<<<< ======= >>>>>>>
# Risoluzione manuale e commit

git add backup-procedure.md
git commit -m "Risolto conflitto orario backup"
git push origin main
```

Generazione e Risoluzione Conflitti

git config pull.rebase false - Merge

Quando eseguiamo un git pull, Git deve integrare le modifiche remote (del server) con le modifiche locali.

I tre comandi che abbiamo visto servono a dire a Git come deve fare questa integrazione.

🧩 1. **git config pull.rebase false** → Merge (comportamento predefinito) 👉 Dice a Git di unire le modifiche remote usando un merge commit.

Crea un nuovo commit di fusione ed è la scelta più “sicura” e mantiene la storia completa e ramificata.

✅ Vantaggio: preserva la cronologia completa.

⚠️ Svantaggio: la storia può diventare più “ramificata” e difficile da leggere.

git config pull.rebase true → Rebase

👉 Dice a Git di spostare i commit locali sopra a quelli remoti, come se fossero stati fatti dopo, in pratica riscrive la storia.

✅ Vantaggio: storia lineare e pulita

⚠️ Svantaggio: riscrive i commit → attenzione se il branch è condiviso con altri.

git config pull.ff only → Fast-forward only

👉 Dice a Git di aggiornare il branch **solo se può avanzare senza conflitti**, cioè senza creare merge né rebase. Funziona solo se non ci sono commit locali.

✅ Vantaggio: storia perfettamente lineare e sicura.

⚠️ Svantaggio: fallisce se hai commit locali non presenti nel remoto.

Buone Pratiche nei Messaggi di Commit

🛠️ HANDS-ON: Analisi repository reali

```
# Analizziamo commit ben scritti
git log --oneline -10

# Esempio di commit messages efficaci:
# "Fix: risolto bug calcolo spazio disco nelle procedure backup"
# "Add: nuova procedura restore database PostgreSQL"
# "Update: aggiornate credenziali accesso server backup"
# "Doc: completata documentazione troubleshooting restore"

# Comando git blame per vedere chi ha modificato cosa
git blame backup-procedure.md

# Colleghiamoci a un repository complesso (es. PostgreSQL)
git clone https://github.com/postgres/postgres.git
cd postgres
git log --oneline -20
git log --grep="backup"
```

Piattaforme Git Web-Based GitLab vs GitHub vs Bitbucket

- **GitHub**
  - Piattaforma più popolare e diffusa per ospitare codice.
  - Ottima integrazione con open source, GitHub Actions, e una grande community.
  - Ideale per progetti pubblici o collaborazioni open.
  - 💡 Punto di forza: community e integrazione con strumenti di sviluppo (CI/CD, issues, code review).

Piattaforme Git Web-Based GitLab vs GitHub vs Bitbucket

- **GitLab**
  - Alternativa open source e self-hosted a GitHub.
  - Include in un’unica piattaforma tutto il ciclo DevOps: repository, CI/CD, sicurezza, monitoraggio.
  - Ideale per aziende o organizzazioni che vogliono controllo completo dei dati.
  - 💡 Punto di forza: pipeline CI/CD integrate e completa automazione DevOps.

Piattaforme Git Web-Based GitLab vs GitHub vs Bitbucket

- **Bitbucket**
  - Piattaforma Atlassian integrata con Jira e Confluence.
  - Supporta sia Git che Mercurial (storicamente).
  - Ottima per team aziendali già nell’ecosistema Atlassian.
  - 💡 Punto di forza: integrazione stretta con strumenti di project management (Jira, Trello).

Sicurezza e Best Practices

Cosa NON fare - Gestione Segreti

🛠️ **HANDS-ON:** Simulazione errore comune

```
# ERRORE: Commit accidentale di password
echo "DB_PASSWORD=super_secret_123" > config.txt
git add config.txt
git commit -m "Aggiunta configurazione database"

# SCOPRIAMO L'ERRORE!
# Come rimuovere definitivamente il segreto
git reset --soft HEAD~1
git reset HEAD config.txt
rm config.txt

# Versione corretta
echo "DB_PASSWORD=\${DB_PASSWORD}" > config.txt
echo "# Variabile d'ambiente DB_PASSWORD richiesta" >> config.txt
git add config.txt
git commit -m "Template configurazione database (senza credenziali)"
```

File .gitignore

```
# Creiamo .gitignore per file sensibili
echo "*.log" > .gitignore
echo "config.local" >> .gitignore
echo ".env" >> .gitignore
echo "backup/*.sql" >> .gitignore

git add .gitignore
git commit -m "Add: gitignore per file sensibili"
```

Documentazione con Git, Sphinx e RST

Documentazione Tecnica: un workflow moderno

Git – Versionamento della conoscenza

- Permette di tracciare l’evoluzione della documentazione come il codice
- Supporta branching, pull request e code review anche per i documenti
- Garantisce collaborazione controllata e storico completo

Sphinx – Generatore di documentazione professionale

- Converte file RST in HTML, PDF, ePub e altro
- Supporta temi, estensioni, API docs auto-generate
- Ideale per documentare progetti software, procedure e manuali tecnici

RST (reStructuredText) – Linguaggio semplice e potente

- Sintassi leggibile, minimale e altamente estensibile
- Perfetto per guide, manuali, specifiche tecniche
- Integrato nativamente con Sphinx

Automazione con GitLab CI/CD (Sessione Avanzata)

Introduzione alle Pipeline

🛠️ **HANDS-ON**: Prima pipeline GitLab CI

Creiamo `.gitlab-ci.yml`:

```yaml
# Pipeline per generazione documentazione automatica
stages:
  - validate
  - build
  - deploy

validate_markdown:
  stage: validate
  script:
    - echo "Validazione sintassi Markdown..."
    - find . -name "*.md" -exec echo "Checking {}" \;

build_docs:
  stage: build
  script:
    - echo "Generazione HTML da Markdown..."
    - echo "Creazione PDF delle procedure..."
  artifacts:
    paths:
      - docs/
    expire_in: 1 week

deploy_docs:
  stage: deploy
  script:
    - echo "Deploy su server documentazione intranet..."
  only:
    - main
```

Principali sistemi di Continuous Integration / Continuous Deployment (CI/CD)

CircleCI

- Servizio cloud-based molto flessibile e veloce.
- Supporta pipeline complesse con configurazione in YAML.
- Buone prestazioni e caching intelligente.
- 💡 Ideale per progetti multi-linguaggio e team che vogliono scalabilità rapida.
- config.yml

Travis CI

- Uno dei primi servizi CI popolari per GitHub.
- Semplice da configurare (.travis.yml), ma oggi meno usato.
- Ottimo per progetti open source, con build gratuite.
- 💡 Facile da usare, ma meno integrato rispetto a soluzioni più moderne.
- .travis.yml

Principali sistemi di Continuous Integration / Continuous Deployment (CI/CD)

GitLab CI/CD

- Integrato nativamente in GitLab, non richiede servizi esterni.
- Gestisce tutto il ciclo DevOps: build, test, deploy, sicurezza.
- Supporta sia cloud che self-hosted runners.
- 💡 Ideale per chi vuole controllo e automazione completa nel proprio ambiente.
- .gitlab-ci.yml

GitHub Actions

- Integrato direttamente in GitHub.
- Usa workflow YAML (.github/workflows) per automatizzare test, build e release.
- Ampia libreria di azioni predefinite della community.
- 💡 Perfetto per progetti già su GitHub e integrazione continua semplice.
- docs.yml
- maven.yml
- release.yml

> *"Nessun ladro, per quanto scaltro, potrà mai rubarti la conoscenza"*

Grazie!

Qualche riferimento utile

- marco.spasiano@cnr.it
- https://github.com/mspasiano
- https://github.com/consiglionazionaledellericerche
- https://github.com/trasparenzAI
- Questa presentazione in pdf