# SOFTWARE VERSIONING

## Software configuration management

Le SCM sono un insieme di discipline che servono per gestire le evoluzione del software definendo degli standard da seguire per creare codice di qualità nei tempi previsti e con il budget disponibile. Quello che si cerca di evitare sono le **aziende di livello 1** cioè quelle aziende composte da poche persone e dove c'è una sola persone che prende le decisioni.

**Struttura del sistema software** che identifica in modo univoco i singoli componenti e li rende accessibili in una qualche forma.

- individuare gli item da tenere sotto controllo nell'SCM
- creare degli schemi, documentazioni che descrivono la gerarchia e che può essere modificata man mano che il codice viene modificato
- stabilire le configurazioni **basedline** cioè una versione formale e stabile di un prodotto software, che viene utilizzata come punto di riferimento per monitorare e gestire i cambiamenti futuri.

**Controllo dei cambiamenti di configurazione**: implica il controllo del rilascio e delle modifiche ai prodotti software durante tutto il ciclo di vita del software. Definiamo i processi di cambiamento:

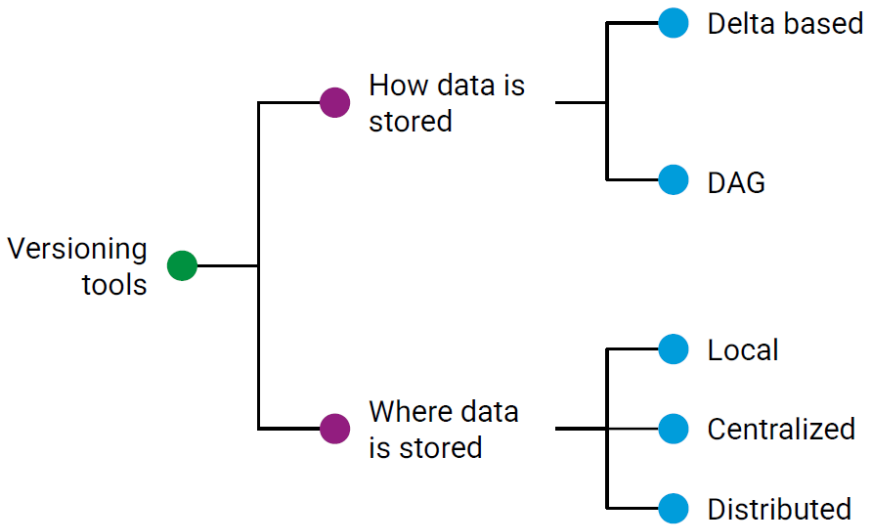
- cambio che vengono chiesti dall'utente
- valutazione dei cambiamenti sulla base degli obiettivi del progetto
- discussione dei cambiamenti. Se viene accettato vengono implementate altrimenti no.
- stabilire una **policy del controllo di cambiamenti** per motivi di **promotion** cioè notificare chi è interessato dei cambiamenti fatti e di **realise** per una nuova versione del prodotto

**Contabilità dello stato della configurazione**: implica la registrazione e la segnalazione del processo di modifica. L'obiettivo della contabilità dello stato è mantenere "un record continuo dello stato e della storia di tutti gli elementi definiti come base (baselined) e delle modifiche proposte a essi.

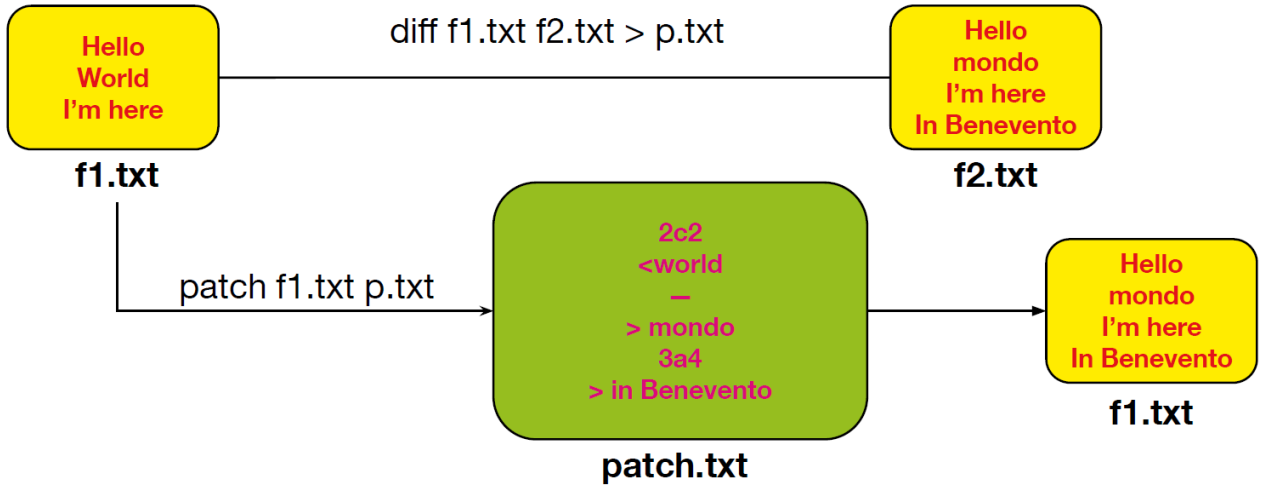
**Auditing della configurazione**: verifica che il prodotto software sia costruito secondo i requisiti,gli standard o l'accordo contrattuale

## Software Versionig

E' responsabile della gestione del cambio del codice sorgente assegnando un **identificativo di revisione** e un **nome** alla modifica che viene associata anche al timestamp e all'autore che ha svolto la modifica



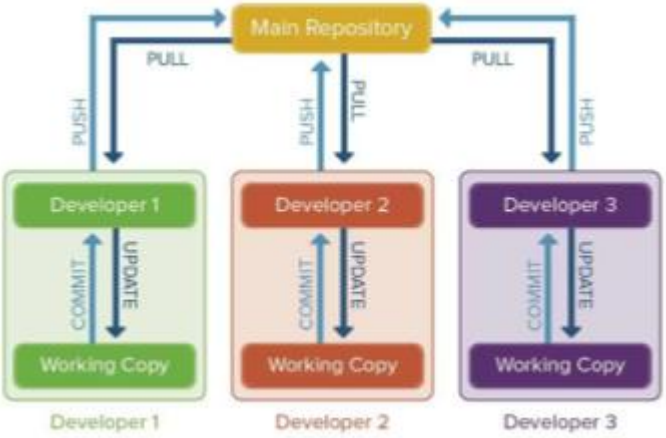

**Delta Base** Il delta-based è una tecnica di controllo della configurazione del software che si concentra sulla registrazione delle modifiche rispetto a uno stato precedente, piuttosto che conservare copie complete di ogni versione del software. Questa tecnica si basa su un approccio a snapshot, in cui viene creato uno stato iniziale del software, e poi, quando si verificano modifiche, vengono salvate solo le differenze (i delta) tra la versione corrente e quella precedente.




Utilizzando diff f1.txt f2.txt >p.txt viene creato un **file di patch** che contiene solo le differenze frai due documenti. Questo file ha una struttura simili:

2c2 -->la riga 2 del primo file è stata cambiata con la riga 2 del secondo file
<world -->per indicare cosa si è tolto
mondo -->per indicare cosa si è aggiunto 3a4 -->la riga 3 nel primo file è stata aggiunta dopo la riga 3 del primo file in Benevento
Con patch f1.txt p.txt applichiamo effettivamente le modifiche e solo quelle contenute nel file di patch

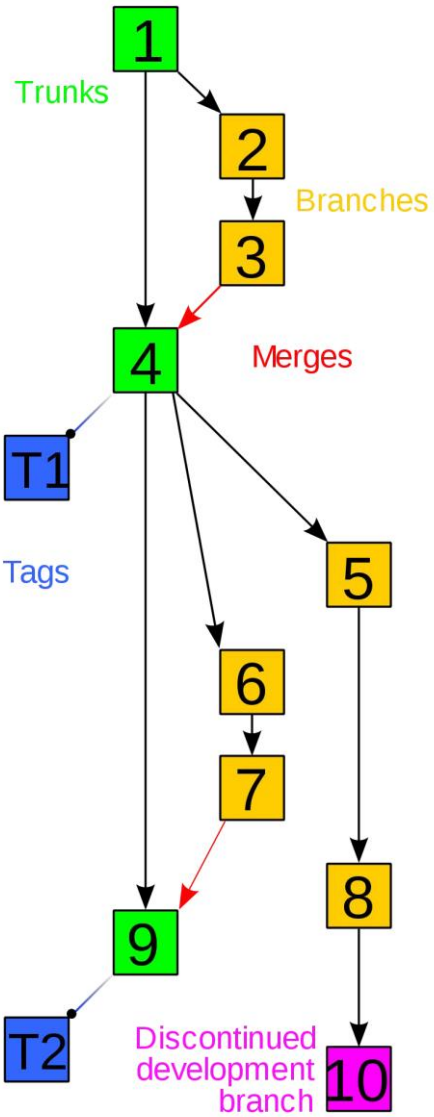
**Structure**
Più sviluppatori possono lavorare insieme allo stesso progetto software in modo **indipendentemente**, sul proprio computer,in momenti diversi o da luoghi diversi condividendo il codice in un **repository centrale** che rappresenta la **versione ufficiale e condivisa** del progetto. Ogni sviluppatore può **contribuire con il proprio lavoro**, ma anche **ricevere gli aggiornamenti degli altri**.
Ogni sviluppatore ha una **copia locale del progetto**, dove può lavorare liberamente. Man mano che apporta modifiche, può **salvarle localmente**, tenendo traccia delle versioni, senza interferire subito con il lavoro degli altri.
Quando lo sviluppatore si sente pronto a condividere ciò che ha fatto, può **inviare le modifiche al repository centrale**.
A sua volta, può anche **recuperare** le novità introdotte dagli altri membri del team, per assicurarsi che tutto il lavoro sia aggiornato e coerente.




**Terminologia**
**Repository**: Nei sistemi di controllo versione, un repository è una struttura dati che memorizza i metadati di un insieme di file o di una struttura di directory. I metadati di un repository includono un record storico delle modifiche nel repository, un insieme di oggetti di commit e un insieme di riferimenti agli oggetti di commit, chiamati heads.
**Branch**: Un insieme di file sotto controllo versione può essere "ramificato" o "forkato" in un determinato momento. Da quel momento in poi, due copie di quei file possono evolversi in modo indipendente, a velocità o in modi differenti.
**Trunk**: La linea unica di sviluppo che non è una ramificazione (nota anche come Baseline, Mainline o Master).
**Commit** (sostantivo): Un insieme di modifiche raggruppate insieme (chiamato changeset),insieme a informazioni meta (come le informazioni sull'autore, un messaggio di commit che descrive la modifica) riguardo alle modifiche. Un

commit descrive le differenze esatte tra due versioni successive. Viene trattato come un'unità atomica.

**Commit** (verbo): Scrivere o unire le modifiche apportate alla copia di lavoro nel repository.




**Commit message**: Una breve nota che registra una descrizione dell'effetto o dello scopo della modifica.

**Head**: Si riferisce al commit più recente, sia nel trunk che in una branch.

**Tag**: Un tag o etichetta si riferisce a uno snapshot importante nel tempo, denominato con un nome amichevole o un numero di revisione.

**Cloning**: Creare un repository che contenga le revisioni da un altro repository.

**Checkout**: Creare una copia locale di lavoro dal repository. Un utente può specificare un are visione specifica o ottenere l'ultima versione.

**Pull, push**: Copiare le revisioni da un repository a un altro. Pulling si riferisce al recupero delle modifiche da un repository remoto e alla loro fusione nel tuo repository locale. Questo aggiorna il tuo repository locale con le modifiche più recenti dal remoto. Pushing implica l'invio delle modifiche che hai effettuato nel tuo repository locale a un repository remoto. Questo aggiorna il repository remoto con le tue modifiche locali.

**Merge**: È il processo di combinazione delle modifiche provenienti da rami diversi in un unico ramo. Permette di integrare le modifiche fatte in rami separati in un ramo comune, risolvendo eventuali conflitti che potrebbero sorgere durante il processo.

**Conflict**: Un conflitto si verifica quando parti diverse fanno modifiche allo stesso documento e il sistema non è in grado di riconciliare le modifiche. Un utente deve risolvere il conflitto combinando le modifiche o selezionando una modifica.

**Blame**: Una ricerca per identificare l'autore e la revisione che ha modificato per ultima una particolare riga.

**Semantica**
Un numero di versione normale assume la forma **X.Y.Z**, dove **X**, **Y** e **Z** sono interi non negativi.
**X** è la versione principale (major),
**Y** è la versione secondaria (minor),
**Z** è la versione di correzione (patch). Ogni elemento deve aumentare numericamente. Ad esempio: 1.9.0 -> 1.10.0 -> 1.11.0.

Una volta che un pacchetto versione è stato rilasciato, il contenuto di quella versione **NON DEVE** essere modificato. Ogni modifica **DEVE** essere rilasciata come una nuova versione. La versione principale zero (0.y.z) è per lo sviluppo iniziale. Qualsiasi cosa può cambiare in qualsiasi momento, infatti è usato per indicare un codice instabile e insicuro. Dalla versione 1.0.0 si definisce l'API pubblica. Il modo in cui il numero di versione viene incrementato dopo questo rilascio dipende dall'API pubblica e da come essa cambia.

**Versione di correzione (Patch version) Z**
**(x.y.Z | x > 0)**: deve essere incrementata se vengono introdotte solo correzioni di bug compatibilicon la versione precedente. Una correzione di bug è definita come un cambiamento interno cherisolve un comportamento errato.

**Versione secondaria (Minor version) Y**
**(x.Y.z | x > 0)**: deve essere incrementata se viene introdotta una nuova funzionalità compatibilecon la versione precedente nell'API pubblica. Può includere modifiche di livello patch. Laversione di correzione **DEVE** essere azzerata a 0 quando la versione secondaria vieneincrementata.

**Versione principale (Major version) X**
**(X.y.z | X > 0)**: essere incrementata se vengono introdotte modifiche incompatibili con laversione precedente nell'API pubblica. Può includere anche modifiche di versione secondaria edi correzione. Le versioni di correzione e secondarie **DEVE** essere azzerate a 0 quando laversione principale viene incrementata.

**Pull/Merge Request**
Chiunque voglia modificare il codice non lo modifica direttamente nel branch principale(main/master), ma crea un branch separato in cui effettua le modifiche. Queste modifiche non diventano effettive immediatamente, perché è necessario effettuare una commit.

Tuttavia, la commit da sola non è sufficiente: per proporre le modifiche, bisogna creare una pull request (o merge request), alla quale devono essere allegati tutti i test effettuati per verificare la correttezza del codice.
Se le modifiche vengono accettate dal maintainer, allora è possibile effettuare il merge nel branch principale (main/master), ma solo dopo che il software versioning ha verificato che non ci siano conflitti con la versione esistente.
I conflitti nel version control si verificano quando due o più sviluppatori modificano la stessa parte di un file in branch diversi e il sistema non è in grado di decidere automaticamente quale versione mantenere.
I conflitti avvengono quando si hanno modifiche concorrenti sulla stessa riga di un file tra due commit diversi; eliminazione di un file in un branch mentre viene modificato in un altro; modifiche incompatibili tra due versioni dello stesso file.

**Costi**
**Tempo e sforzo**: Nel caso più semplice, gli sviluppatori potrebbero conservaremanualmente più copie delle diverse versioni del codice, etichettandole di conseguenza.

**Apprendimento e manutenzione**: Il team deve imparare i concetti alla base del versioncontrol e seguire le migliori pratiche per ottenere benefici concreti.

**Benefici**
**Possibilità di annullare le modifiche**: Il version control semplifica l'annullamento delle modifiche, permettendo agli sviluppatori di sperimentare senza il timore di compromettere il codice esistente.

**Semplificazione dello sviluppo**: Grazie a funzionalità come **branching, merging e labeling**,è più facile gestire lo sviluppo in diverse fasi: sviluppo, test, staging, produzione,deployment, ecc.

**Gestione dei branch**: Tipicamente, il branch principale (**main/master**) contiene il codiceconsolidato, mentre gli altri branch contengono nuove funzionalità in fase di sviluppo.

**Migliora la collaborazione e la comunicazione**: Il version control permette di individuareconflitti tra le modifiche (modifiche incompatibili sulle stesse righe di codice), riducendo lanecessità di coordinazione tra gli sviluppatori.

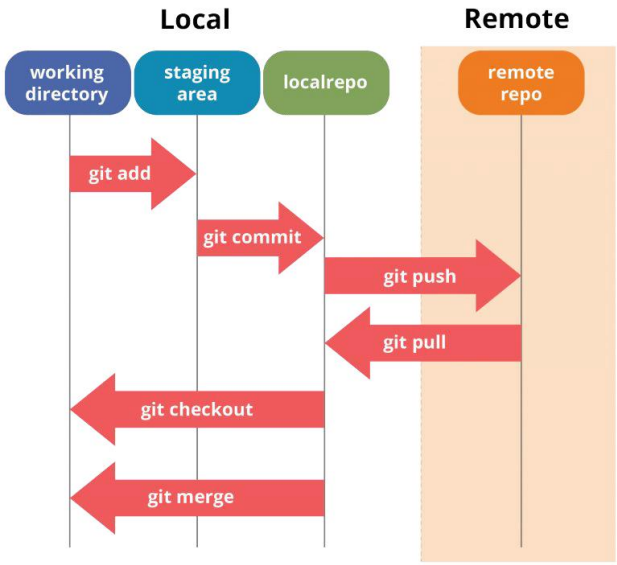
**Mitigazione dei danni**: Monitorare la cronologia del codice aiuta a comprendere cosa èstato fatto, da quanto tempo esiste un bug e come risolverlo. È inoltre possibile installaree testare versioni precedenti del software.

**Semplifica il debugging**: Applicando un test case a più versioni del codice, si puòidentificare rapidamente quale modifica ha introdotto un bug.

## Git

Git è un software per il controllo di versione distribuito utilizzabile da interfaccia a riga dicomando, creato da Linus Torvalds nel 2005, basata su versioning di tipo DAG.

### Workflow




Per poter modificare il codice presente in un **repository remoto**, è necessario seguire alcuni passaggi fondamentali. Il primo passo consiste nel **clonare** il repository remoto sulla propria macchina locale. Questo permette di ottenere una copia del codice sorgente nella propria **working directory**, ovvero la cartella locale dove si lavorerà sulle modifiche. Una volta effettuatele modifiche ai file, questi devono essere aggiunti alla **staging area**, uno spazio temporaneo che raccoglie i file modificati prima di essere confermati. Questo passaggio avviene tramite il comando git add, che può essere utilizzato per aggiungere singoli file o tutti i file modificati. Dopo aver preparato i file nella staging area, è necessario creare un **commit**, che rappresenta un'istantanea delle modifiche apportate. Il commit viene effettuato con un messaggio descrittivo che spiega le modifiche effettuate. Tuttavia, il commit rimane ancora nella repository locale e non è stato inviato al repository remoto. Per rendere effettive le modifiche nel repository remoto, è necessario eseguire un **push**, che aggiorna il branch su cui si sta lavorando con i commit effettuati localmente. Nel caso in cui altri sviluppatori abbiano modificato il codice nel repository remoto, è importante aggiornare la propria copia locale prima di procedere con nuove modifiche. Questo si fa con un pull, che permette di scaricare e integrare le modifiche più recenti. Infine, quando si lavora su più branch, è possibile passare da un branch all'altro e unire le modifiche tramite le operazioni di **checkout** e **merge**. Il merge permette di combinare le modifiche provenienti da diversi branch in un unico flusso di sviluppo.

| COSA GESTIRE | COSA ESCLUDERE |
| --- | --- |
| **Source Code** | **File temporanei o generati** |
| **Configuration Files** | **File di log** (.log, .tmp) |
| **Build & Deployment Scripts** | **File compilati** (.class, .exe, dist/, node_modules/) |
| **Dependencies** (Librerie esterne e package manager) | **File di cache** (__pycache__, .cache) |
| | **File di configurazione specifici per l'utente** |
| | **Impostazioni locali di IDE/editor** (.vscode/, .idea/, .DS_Store) |
| | **File personali** (a meno che non siano critici per il deployment) |
| | **Credenziali e segreti hardcoded** |

Il **file .gitignore** viene utilizzato per escludere specifici file e directory dal versionamento in Git. Questo è utile per evitare di includere file temporanei, di log, compilati, di configurazione locale odi cache.

**Come usare .gitignore**

Creare un file .gitignore nella root del repository.
Aggiungere le regole per escludere i file non necessari.

**1.Clonare il repository remoto**

- Comando: git clone <URL-del-repository>
- Scarica una **copia completa** del repository sulla propria macchina (codice + cronologia).
- Crea una **working directory locale** dove poter lavorare.

**2.Modificare i file nella working directory**

- Apporta le modifiche desiderate ai file del progetto.
- Queste modifiche sono **non tracciate** finché non vengono aggiunte esplicitamente a Git.

**3.Aggiungere i file modificati alla staging area**

- Comando: git add <file> oppure git add . (per tutti i file modificati)
- La staging area è una **zona intermedia** prima del commit.
- Permette di **selezionare** quali modifiche includere nel prossimo commit.

**4.Creare un commit**

- Comando: git commit -m "Messaggio descrittivo"
- Crea un'**istantanea delle modifiche** nella repository locale.
- Il messaggio dovrebbe spiegare chiaramente **cosa è stato modificato**.

**5.Inviare le modifiche al repository remoto (push)**

- Comando: git push origin <nome-branch>
- **Condivide i commit** fatti localmente con il repository remoto.
- Rende visibili le modifiche agli altri collaboratori.

**6.Aggiornare la propria copia locale (pull)**

- Comando: git pull origin <nome-branch>
- Scarica e integra le modifiche più recenti dal repository remoto.
- Aiuta a **lavorare su una base aggiornata** ed evitare conflitti.

**7.Gestire i branch (opzionale, ma comune)**

- Cambiare branch: git checkout <nome-branch>
- Creare un nuovo branch: git checkout -b <nuovo-branch>
- Unire branch: git merge <branch-da-unire>
- Utile per **organizzare il lavoro**, ad esempio separando sviluppo, test e produzione.

**CONFLITTO**

Auto-merging messaggio.txt
CONFLICT (content): Merge conflict in messaggio.txt
Automatic merge failed; fix conflicts and then commit the result.

^Git ha trovato modifiche incompatibili nella stessa riga del file. Non può decidere da solo quale versione tenere.

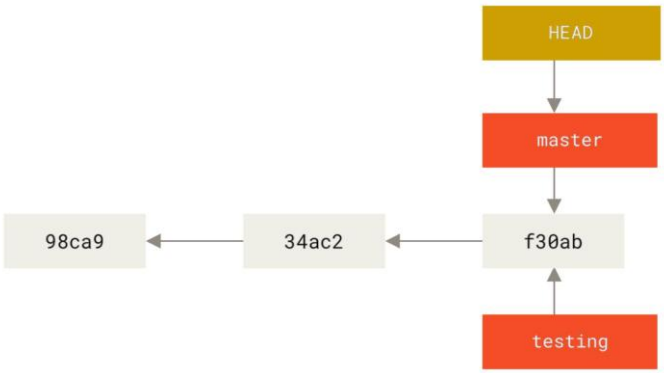
```
<<<<<<< HEAD
Ciao, questo è un messaggio modificato da Main.
=======
Ciao, questo è un messaggio modificato dal nuovo branch.
>>>>>>> nuovo-branch
```

- <<<<<<< HEAD → indica la versione attuale nel branch main
- ======= → separa le due versioni
- >>>>>>> nuovo-branch → indica la versione del branch che stai cercando di unire

**BRANCH**

In Git, ogni **branch** è come un **segnalibro** che punta a un certo punto della storia del progetto, cioè a un **commit** identificato da un identificativo **hash SHA-1** di 40 caratteri

Il **puntatore HEAD** non è un branch vero e proprio, ma serve a Git per capire **dove ti trovi attualmente**.




```
git checkout f30ab
```

Riporta “indietro nel tempo”, quindi alle modifiche fatte in quel commit.

```
git log

commit 38056a08efafb205ce8fc9a3bd20b203e7d15829 (HEAD -> main)
Author: cla <claudio@DESKTOP-GVEABB2.>
Date:   Fri Mar 21 11:25:25 2025 +0100

   addedd book.java

commit 2741a4d5496f9c78530c01914559774e49ce62ae (origin/main,
origin/HEAD)
Author: Claudio Mininno <c.mininno@studenti.unisannio.it>
Date:   Fri Mar 21 10:09:51 2025 +0000

   Initial commit
```

(HEAD -> main) significa che il tuo HEAD sta tracciando il branch main e che sei attualmente su quel branch. Questo vale per qualunque ramo in cui ci troviamo.

```
git status

On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

      modified:   file1.txt
      new file:   file2.txt
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)

        modified:   file3.txt
        deleted:    file4.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        file5.txt
```

- **Changes to be committed**: I file che sono stati aggiunti all'area di staging (pronti per essere committati).
- **Changes not staged for commit**: I file che sono stati modificati, ma non ancora aggiunti all'area di staging.
- **Untracked files**: I file che non sono ancora sotto il controllo di versione Git (ad esempio, file nuovi che non sono stati aggiunti).

Un **merge fast-forward** accade quando il branch di destinazione non ha commit propri e può semplicemente avanzare per includere i commit del branch da unire. In pratica, Git può spostare il puntatore del branch di destinazione avanti nel tempo, senza creare un commit di merge separato.
Un "fast-forward" avviene quando:

1. Il branch di destinazione è direttamente alla base del branch che stai cercando di unire.
2. Non ci sono divergenze tra i due branch (cioè, il branch di destinazione non ha modifiche che non siano già nel branch di origine).

```
git checkout main
git merge dev1

Updating 38056a0..e20ed48
Fast-forward
 file1.txt |  2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Un **merge non fast-forward** avviene quando i due branch hanno cambiamenti divergenti e Git non può semplicemente "spostare" il puntatore del branch di destinazione avanti. In questo caso, Git crea un **commit di merge** che unisce i cambiamenti dei due branch.
Un **merge non fast-forward** accade quando:

- Hai dei commit sia sul branch di origine che su quello di destinazione.
- Git non può semplicemente spostare il puntatore del branch di destinazione per includere i commit del branch di origine, perché entrambi i branch hanno fatto modifiche in parallelo.

```
git checkout main
git merge --no-ff dev1

Updating 38056a0..e20ed48
Merge made by the 'recursive' strategy.
 file1.txt |  2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Un **rebase** è un'alternativa al merge che riscrive la cronologia dei commit. Con il rebase, si sposta un intero set di commit (un branch) sopra un altro commit, come se i tuoi cambiamenti fossero stati fatti a partire da una base più recente.
Immagina che tu abbia il branch main e il branch feature:

1. **main** ha un commit A.
2. **feature** ha i commit B e C.

Se fai un merge di feature su main, avrai qualcosa di simile:

```
A---B---C (feature)
 \     /
  M----- (main)
```

Un **rebase** cambia la cronologia in modo che i commit di feature vengano riposizionati sopra main. Il risultato sarà:

```
A---B---C (main, feature)
```

Per effettuare il rebase, è necessario avere tutti i commit che si vogliono portare nel rebase nel tuo branch master.

**Maven** è un progetto open source sviluppato da **Apache** che facilita l'organizzazione efficiente di un progetto Java. Rende più facile gestire e mantenere grandi progetti fornendo una struttura coerente e una serie di convenzioni su come organizzare il progetto, aiutando gli sviluppatori ad automatizzare il processo di **build**, test e distribuzione del software.
Una delle caratteristiche principali di Maven è la sua capacità di gestire le dipendenze. Maven tiene traccia di tutte le librerie e altre dipendenze necessarie per un progetto e le scarica automaticamente quando sono richieste. Questo rende facile per gli sviluppatori utilizzare librerie esterne nei loro progetti senza doverle scaricare e gestire manualmente.

Maven **obbliga** ad avere una **struttura fissa** delle directory

- Il **POM** è alla **radice** del **progetto**
- Poi ci sono due directory:
    - **src**: contiene il sorgente
    - **target**: contiene i file generati alla fine del processo di compilazione

Avendo a disposizione il **POM** e la directory **src**, chiunque può essere in grado di ricostruire la directory **target**

**POM (Project Object Model)**
Maven utilizza un approccio dichiarativo per specificare la build e le dipendenze del progetto. Il file centrale che Maven utilizza è il **pom.xml** (Project Object Model). Questo file gestisce le dipendenze del progetto, i plugin e la configurazione della build. Inoltre, Maven offre una serie di plugin integrati per attività comuni e può essere esteso con plugin personalizzati.

**Repository Maven**
Un **repository Maven** è una directory che contiene i file compilati insieme ai relativi metadati. I metadati si riferiscono ai file POM associati a ciascun progetto. Questi metadati consentono a Maven di scaricare le dipendenze necessarie per il progetto.
Maven dispone di tre tipi di repository:

1. **Repository Locale**:
   Il repository locale si trova sulla macchina dello sviluppatore e contiene tutte le dipendenze, come i file JAR. Ogni sviluppatore ha il proprio repository locale.
2. **Repository Remoto**:
   Il repository remoto è situato su un server web e viene utilizzato quando

Maven deve scaricare le dipendenze. Quando un artefatto non è presente nel repository locale, Maven lo scarica dal repository remoto e lo memorizza nel repository locale.

3. **Repository Centrale**:
   Il repository centrale è gestito dalla comunità Maven ed è la fonte principale da cui Maven scarica le dipendenze, qualora non siano presenti nei repository locali o remoti.

**Le Coordinate Maven**

Il file **pom.xml** definisce le coordinate Maven per ciascun artefatto. Le coordinate sono composte da tre parti obbligatorie: **groupId**, **artifactId** e **version**. Questi elementi identificano in modo univoco un artefatto all'interno del repository Maven, agendo come un sistema di coordinate per i progetti.

1. **groupId**:
   Questo identificatore è generalmente univoco all'interno di un'organizzazione o di un progetto. Non è necessario che il **groupId** corrisponda alla struttura del pacchetto del progetto, ma è una buona pratica seguire la convenzione.
2. **artifactId**:
   L'**artifactId** è generalmente il nome del progetto. Insieme al **groupId**, definisce univocamente un progetto nel repository. È raro che venga menzionato separatamente, poiché il **groupId** è spesso condiviso tra più progetti all'interno della stessa organizzazione.
3. **version**:
   La **version** specifica la versione dell'artefatto. Questo campo è fondamentale per mantenere il versionamento del progetto e gestire modifiche e aggiornamenti.

Il **Build Lifecycle** di Maven è un concetto centrale che definisce il flusso di esecuzione delle fasi di costruzione di un progetto, dal momento in cui avvii la build fino alla sua conclusione. Il ciclo di vita di Maven è suddiviso in una serie di fasi, e ogni fase rappresenta una parte specifica del processo di compilazione, test e distribuzione del progetto.

Maven prevede tre cicli di vita principali:

1. **default**: gestisce il processo di build del progetto.
2. **clean**: gestisce la rimozione dei file generati durante la build.
3. **site**: gestisce la creazione del sito web del progetto (documentazione e altre risorse).

**Default Build Lifecycle (Ciclo di Vita Predefinito)**
Il ciclo di vita predefinito è il più importante e definisce le fasi che vengono eseguite durante la costruzione del progetto. Esso include fasi come la compilazione, l'esecuzione dei test, la creazione del pacchetto e il deploy.
Le fasi principali del ciclo di vita **default** sono:

1. **validate**: Verifica che il progetto sia corretto e che tutte le informazioni necessarie siano disponibili.
2. **compile**: Compila il codice sorgente del progetto.
3. **test**: Esegue i test unitari sul codice compilato per verificare che funzioni correttamente.
4. **package**: Crea il pacchetto del progetto (ad esempio, un file JAR, WAR o EAR) a partire dal codice compilato e dai file di configurazione.
5. **verify**: Verifica che il pacchetto sia valido e che i test siano passati.
6. **install**: Installa il pacchetto nel repository locale di Maven, rendendolo disponibile per altri progetti.
7. **deploy**: Distribuisce il pacchetto nel repository remoto per renderlo accessibile ad altri sviluppatori o progetti.

**GOAL**
Un **goal** appresenta un'azione specifica da eseguire durante un ciclo di vita di build. I goal sono l'unità più piccola e specifica di un processo Maven e sono generalmente associati ai **plugin**. Ogni plugin può avere uno o più goal, e ogni goal è progettato per eseguire una particolare attività, come la compilazione del codice, l'esecuzione dei test, la creazione del pacchetto o la distribuzione. Questi goal possono essere eseguiti direttamente dalla linea di comando o come parte di un ciclo di vita di Maven.

**PLUG-IN**
I **plugin** sono la funzionalità centrale di Maven che consentono il riutilizzo della logica di build comune tra più progetti. Questo avviene eseguendo un'**azione** (ad esempio, creare un file WAR o compilare i test unitari) nel contesto del POM.
Il comportamento dei plugin può essere personalizzato tramite un set di parametri unici che vengono esposti nella descrizione di ciascun **goal** del plugin (o **Mojo**).
Un **Mojo** è un goal in Maven, e i plugin consistono in uno o più goal (Mojos).
I Mojos possono essere definiti come classi Java annotate. Un Mojo specifica i metadati su un goal: il nome del goal, a quale fase del ciclo di vita appartiene, e i parametri che si aspetta.
I goal dei plugin sono legati a fasi specifiche del ciclo di vita.

**ARCHETIPI**

Gli **archetipi** in Maven sono modelli predefiniti utilizzati per creare nuovi progetti cioè è un template che fornisce una struttura base per un progetto, compreso il file pom.xml, la struttura delle directory, e talvolta anche codice di esempio. Gli archetipi semplificano la creazione di nuovi progetti, standardizzando la configurazione iniziale e riducendo il tempo necessario per configurare un nuovo progetto Maven.

Quando crei un progetto utilizzando un archetipo, Maven esegue una serie di operazioni:

1. **Creazione di una struttura di directory predefinita**: Ogni archetipo contiene una struttura di directory predefinita
2. **Generazione di un pom.xml di base**: Ogni archetipo fornisce un file pom.xml preconfigurato che definisce le dipendenze, i plugin, e le configurazioni di base per il progetto.
3. **Codice di esempio (facoltativo)**: Alcuni archetipi possono includere codice di esempio per un tipo di applicazione specifico.

```
mvn archetype:generate -DgroupId=com.example -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

- -DgroupId=com.example definisce l'ID del gruppo del progetto.
- -DartifactId=my-app definisce l'ID dell'artefatto (nome del progetto).
- -DarchetypeArtifactId=maven-archetype-quickstart specifica l'archetipo da usare (in questo caso, un archetipo di base per un'applicazione Java).
- -DinteractiveMode=false evita di dover rispondere a delle domande durante la creazione del progetto.

```
mvn archetype:catalog
```

per la ricerca di archetipi

**DIPENDENZE**

Le **dipendenze** sono librerie o altri progetti di cui il tuo progetto ha bisogno per essere compilato e funzionare correttamente. Maven gestisce le dipendenze in modo centralizzato, scaricando automaticamente le librerie richieste da repository remoti o locali.

Le dipendenze in Maven sono definite nel file pom.xml del progetto, utilizzando l'elemento **<dependencies>**. Ogni dipendenza è identificata da tre informazioni principali:

1. **groupId**: Identifica il gruppo o l'organizzazione che mantiene la libreria.
2. **artifactId**: Il nome della libreria o dell'artefatto.
3. **version**: La versione specifica della libreria.

1. **Dipendenze di compilazione (Compile Dependencies)**:
   Sono necessarie per compilare il codice. Queste dipendenze vengono incluse automaticamente in fase di compilazione e in fase di esecuzione. Sono definite nel blocco <dependencies> nel pom.xml.
2. **Dipendenze di test (Test Dependencies)**:
   Sono necessarie solo durante la fase di test e non vengono incluse nel pacchetto finale. Queste dipendenze sono definite utilizzando l'elemento <scope>test</scope>.
3. **Dipendenze runtime (Runtime Dependencies)**:
   Sono necessarie per l'esecuzione del programma, ma non per la compilazione. Vengono incluse solo in fase di esecuzione.
4. **Dipendenze di sistema (System Dependencies)**:
   Sono librerie che devono essere fornite manualmente, ad esempio librerie che non sono disponibili in un repository pubblico. Queste dipendenze sono identificate con un percorso locale.

   **Dipendenze di provided (Provided Dependencies)**:
   Sono necessarie solo durante la fase di compilazione e test, ma sono fornite in fase di esecuzione dall'ambiente in cui il progetto verrà eseguito.

Il **scope** di una dipendenza definisce in quale fase del ciclo di vita del progetto la dipendenza è necessaria.

1. **compile**: È il default e indica che la dipendenza è necessaria per la compilazione, il test, l'esecuzione e la distribuzione.
2. **provided**: Indica che la dipendenza è necessaria per la compilazione e i test, ma sarà fornita dal contenitore o dall'ambiente di esecuzione (ad esempio, un server web come Tomcat).
3. **runtime**: La dipendenza è necessaria solo durante l'esecuzione del programma, ma non durante la compilazione.
4. **test**: La dipendenza è necessaria solo durante la fase di test e non per la compilazione o l'esecuzione.
5. **system**: La dipendenza è necessaria per la compilazione e l'esecuzione, ma deve essere fornita manualmente con un percorso specificato nel pom.xml.

Le dipendenze in Maven possono essere **transitive**, cioè che se il tuo progetto dipende da una libreria A e la libreria A dipende a sua volta da una libreria B, Maven gestirà automaticamente anche la dipendenza di B. Questo ti evita di dover gestire manualmente ogni dipendenza indiretta.

**MULTI-MODUL PROJECT**

Un **progetto multi-modulo** è un progetto che contiene più moduli (o sotto-progetti) gestiti sotto un'unica configurazione. Ogni modulo può essere un progetto Maven indipendente, ma sono tutti gestiti insieme in modo centralizzato da un **progetto padre**.

Maven esegue la build dei moduli nell'ordine in cui sono elencati nel pom.xml del progetto padre. Ogni modulo dipende dalle configurazioni definite nel progetto padre, ma può anche avere configurazioni specifiche se necessario.

# SERVICE E MICROSERVICE

SOAP-> verbi
REST->stato

Se abbiamo bisogno di fare il login lo scriviamo una sola volta esponiamo la funzionalità del login come servizio lo dispieghiamo una sola volta e tutti possono usarlo altrimenti torniamo alle librerie dovete implementare la libreria per fare il login e la dovete dare a tutti e tutti la integrano invece col servizio è più leggero potete chiamare con la rete una funzionalità esterna e quando diciamo se la convibrazione andto bene se l'avete fatto la richiesta in maniera opportuna voi ritornerò OKE quindi siete autorizzate a fare qualcosa

Microservizi →sfumatura della architttura al servizio perché i servizi può essere qualsiasi cosa mentre il microservizio è un servizio specializzato con un solo compito o un solo dominio, cioè solo per un determinato problema →poliglottismo, cioè con più linguaggi diversi i servizi e unire tutto con un'unica interfaccia

| **Compilazione del progetto** | **Esecuzione del progetto** |
|---|---|
| **Maven:** mvn clean install → jar nella cartella target/ | java -jar target/nomefile.jar (spring-boot:run) |
| **Gradle:** ./gradlew build → jar in build/libs/ | java -jar build/libs/nomefile.jar |

## MultiTier Architecture

Presentation tier
Business logic tier
Data tier

→modularità dell'app e semplicià nella gestione