



SAPIENZA
UNIVERSITÀ DI ROMA

Programmazione per il Web

Appunti integrati con il libro
"Introduction to full stack web development with Go and Vue.js"

author Valerio Fontana

15 ottobre 2024

Indice

Prefazione	3
1 Applicazioni	4
1.1 Servizi web	5
1.2 REpresentational State Transfer	5
1.2.1 Uniform Resource Identifier	6
1.2.2 Vincoli di REST	7
1.3 Controllo delle Versioni	8
2 Global Information Tracker	11
2.1 Commit	12
2.1.1 Tag	13
2.1.2 Staging Area	13
2.2 Branch	14
2.2.1 il puntatore HEAD	15
2.3 Merge	16
2.3.1 Conflitti di Merge	18
2.4 Repository	22
2.4.1 Remote Repository	22
2.5 Clone	23
2.6 Push	24
2.7 Fetch	26
2.8 Pull	27
2.9 Hosting delle Repository Git	28
2.9.1 Issues	30
2.9.2 Fork	31
2.9.3 Richieste di Pull/Merge	31

Prefazione

Appunti personali per il corso di Programmazione per il Web, tenuto dal professore Emanuele Panizzi presso l'Università degli Studi di Roma "La Sapienza".

Per ulteriori risorse inerenti ad altri corsi <https://github.com/FeddyLix17/Computer-Science>.

Licence:

These documents are distributed under the [GNU Free Documentation License](#), a form of copy-left intended for use on a manual, textbook or other documents. Material licensed under the current version of the license can be used for any purpose, as long as the use meets certain conditions:

- All previous authors of the work must be attributed.
- All changes to the work must be logged.
- All derivative works must be licensed under the same license.
- The full text of the license, unmodified invariant sections as defined by the author if any, and any other added warranty disclaimers (such as a general disclaimer alerting readers that the document may not be accurate for example) and copyright notices from previous versions must be maintained.
- Technical measures such as DRM may not be used to control or obstruct distribution or editing of the document.

Capitolo 1

Applicazioni

Nel mondo dello sviluppo web, le applicazioni, disponibili in varie forme e dimensioni, consentono agli utenti di svolgere un'ampia gamma di attività, adattandosi alle diverse piattaforme ed esigenze.

Un'applicazione può essere:

- [Desktop](#), installata su un computer ed eseguita direttamente dal corrispondente sistema operativo (potrebbe richiedere una connessione ad internet).
- [Mobile](#), installata ed eseguita su uno smartphone, tablet oppure orologio direttamente dal corrispondente sistema operativo (potrebbe richiedere una connessione ad internet).

Può essere a sua volta nativa (sviluppata per un sistema operativo specifico) oppure ibrida (sviluppata per più sistemi operativi).

- [Web](#), eseguita all'interno di un [Browser](#), accessibile attraverso un [Unified Resource Locator](#) (richiede una connessione ad internet).

Può essere a sua volta

- [Single Page Applications](#), interagendo con gli utenti aggiornando dinamicamente la pagina web corrente con i dati forniti dal server web, senza bisogno di ricaricare l'intera pagina
- oppure [Progressive Web Apps](#), una evoluzione delle applicazioni web, può essere aggiunta alla schermata iniziale del dispositivo così da poterne sfruttare le stesse funzionalità di un'applicazione nativa, inoltre funziona anche in assenza o connessione di rete di scarsa qualità.

All'interno di un'applicazione web, alcune parti del codice come la logica locale oppure l'interfaccia utente vengono eseguite esclusivamente dal browser (frontend) mentre altre parti come la logica dell'applicazione o il sistema di gestione del database esclusivamente dal server web (backend).

Le parti di codice di frontend sono scritte utilizzando linguaggi di markup, mentre quelle di backend utilizzano linguaggi compilati o interpretati.

La realizzazione di [entrambe](#) le parti viene definita come [full-stack web development](#).

La trasmissione di dati e l'esecuzione di operazioni sono alla base della comunicazione tra il client (browser) ed il server.

1.1 Servizi web

Esistono insiemi di regole e protocolli che permettono ad una applicazione (desktop, mobile, web o addirittura tra server) di richiedere e scambiare dati o funzionalità con un server tramite Internet, indicate come Application Programming Interface.

Le API Web sono i componenti fondamentali che consentono a diversi servizi Web, server e client di comunicare efficacemente.

Esse definiscono una raccolta di [endpoint](#) o URL che gli sviluppatori possono utilizzare per inviare richieste e ricevere risposte in un formato strutturato, tipicamente JSON (JavaScript Object Notation) oppure XML (eXtensible Markup Language).

Tali endpoint sono come porte di accesso a un servizio Web dove ognuno offrirà una funzione o una risorsa dati specifica.

1.2 REpresentational State Transfer

L'informatico americano Roy Fielding propose nella sua [tesi di dottorato](#) negli anni 2000 uno stile architettonico per la progettazione di applicazioni di rete e servizi Web la cui comunicazione sarebbe altrimenti poco curata.

Tale paradigma viene attualmente utilizzato per lo sviluppo di Web APIs (attraverso il protocollo HTTP).

Questo stile non si focalizza sui dettagli dell'implementazione, piuttosto si concentra sui ruoli dei componenti all'interno di un sistema descritto con dei vincoli.

Un servizio viene considerato RESTful se implementa tutti i principi [REST](#) e rispetta tutti i suoi vincoli.

Ogni possibile oggetto (come un documento, un'immagine, etc.) viene astratto in REST come una **risorsa**.

Una risorsa può variare nel tempo, essere a sua volta un insieme di altre risorse oppure gruppi di esse possono essere mappate ad uno stesso valore.

Per tenere traccia dello stato corrente o previsto di una risorsa, in ogni momento, vengono aggiunti dei metadati che la descrivono (o **rappresentano**) nel formato media type (o MIME).

MIME identifica rispettivamente il formato dei file e dei contenuti trasmessi su Internet (definendo così univocamente come processare la rappresentazione di una risorsa).

1.2.1 Uniform Resource Identifier

Ogni risorsa (fisica o logica) viene identificata da una univoca sequenza di caratteri.

Essi non sono da confondere

- con gli URL, che forniscono in aggiunta i mezzi per individuare una risorsa su Internet e ne specificano le modalità per accedervi
- e con gli URN (Uniform Resource Name), che invece forniscono un nome univoco a livello globale per una risorsa.

Dunque è previsto che un URN rimanga persistente anche se la risorsa non dovesse essere più disponibile (per convenzione, iniziano con "urn:").

Formalmente, un URI è strutturato come

URI = scheme ":" ["/" authority] path ["?" query] ["#" fragment] authority = [userinfo "@"] host [":" port]

dove

- **scheme** identifica il protocollo di comunicazione
- **authority** identifica l'entità che controlla la risorsa (solitamente il server)
- **path** identifica il preciso percorso dove risiede la risorsa
- **query** contiene una richiesta di informazioni specifiche
- **fragment** identifica una specifica parte della risorsa
- **userinfo** contiene informazioni di autenticazione per accedere alla risorsa
- **host** identifica il server web che ospita la risorsa al livello di rete
- **port** identifica la porta del server web che ospita la risorsa

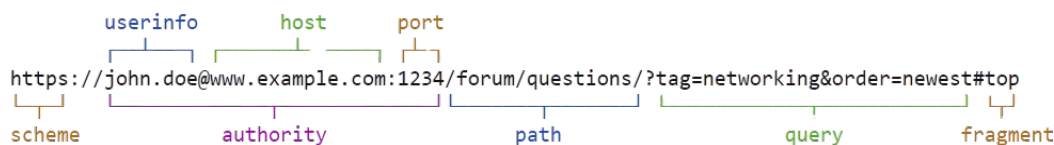


Diagram illustrating the components of the URI `https://john.doe@www.example.com:1234/forum/questions/?tag=networking&order=newest#top`:

- scheme**: `https`
- authority**: `john.doe@www.example.com` (includes **userinfo**: `john.doe@` and **host**: `www.example.com`)
- port**: `1234`
- path**: `/forum/questions/`
- query**: `?tag=networking&order=newest`
- fragment**: `#top`

REST supporta i seguenti metodi per rispettivamente creare, leggere, modificare e cancellare una risorsa dato il suo URI

- **POST** `http://example.com/managed-devices/{id}`
- **GET** `http://example.com/managed-devices/{id}`
- **PUT** `http://example.com/managed-devices/{id}`
- **DELETE** `http://example.com/managed-devices/{id}`

1.2.2 Vincoli di REST

- **stateless**, ogni richiesta da parte del client deve contenere tutte le informazioni necessarie per essere totalmente compresa dal server (aumentandone indirettamente la visibilità, l'affidabilità e la scalabilità)
- **cache**, tutti i dati di risposta devono essere contrassegnati implicitamente o esplicitamente come "cacheable".

Così facendo diventa possibile riutilizzare tale risposta per una richiesta identica in futuro. (migliorando la scalabilità e le prestazioni)

- **separazione delle "preoccupazioni"**, avendo client e server ruoli distinti possono "evolvere" in modo indipendente (migliorando la portabilità e la scalabilità)
- **interfaccia uniforme**, ogni interazione utilizza la stessa struttura di base (stesso protocollo, stessa struttura del messaggio, ecc.).

Ciò aiuta a semplificare il design complessivo del sistema a discapito della personalizzazione.

I vincoli che REST definisce per implementare correttamente un'interfaccia uniforme sono

- **identificazione delle risorse**
 - **manipolazione delle risorse tramite rappresentazioni**
 - **messaggi autodescrittivi**
 - **e hypermedia come motore dello stato dell'applicazione** (ad esempio, il client ha bisogno solo dell'URL iniziale).
- **sistema a livelli**, i livelli nelle trasmissioni (ad esempio, gli host intermedi) possono isolare, modificare, riprodurre e incapsulare i messaggi REST.

Ciò consente all'architettura di avere agenti intermedi che, separando i "livelli" nell'architettura, riducono la complessità complessiva e disaccoppiano i servizi in livelli diversi.

I componenti in livelli diversi non possono vedere oltre i livelli immediatamente precedenti/successivi.

Ogni componente in un livello intermedio in REST può modificare efficacemente il contenuto dell'interazione che scorre, in quanto i messaggi sono autodescrittivi e la semantica è visibile a ogni componente.

- **codice su richiesta**, il client può scaricare ed eseguire codice

1.3 Controllo delle Versioni

Sei alla fine del tuo ultimo anno all'università e stai scrivendo la tesi.

Quando pensi che sia pronta, chiami il documento **final.doc** e lo invii al tuo supervisore/-professore per la revisione (e si spera) l'approvazione.

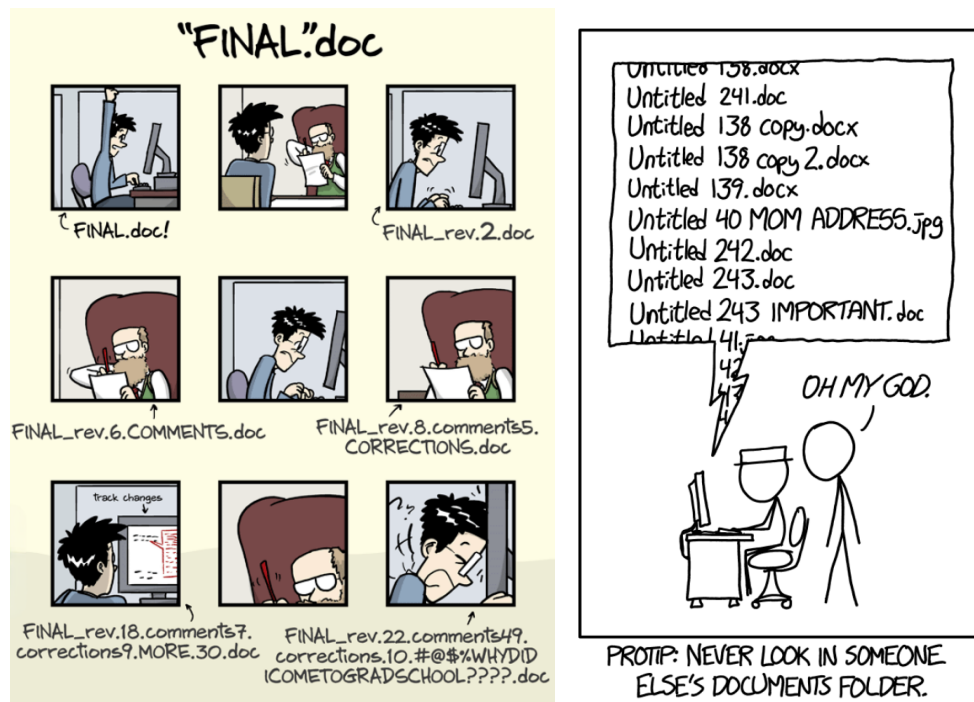
Il professore legge la tua tesi, aggiunge commenti e segna i paragrafi del testo che dovresti riscrivere, rispeditoti il documento denominato **final.comments.doc**.

Inizi a modificare con entusiasmo la tua tesi e la reinvii come **final_rev2.doc**, tuttavia il tuo professore ti rispedisce un nuovo documento nominato **final_rev2.comments2.doc**.

Ora rispeditrai **final_rev2.comments2.corrections1.doc** e lui rispeditrà nuovamente ... avete capito.

Alla fine di questa lunga conversazione, ti sarai ritrovato con molti file con nomi incoerenti nella migliore delle ipotesi, sperando che l'ultimo file creato sia anche la versione più recente e migliore.

Questo è uno dei principali problemi quando più persone lavorano allo stesso progetto o il progetto dura anni.



La prima immagine rappresenta il racconto [originale](#) scritto da [Jorge Cham](#), mentre per la seconda i crediti vanno a XKCD - Randall Munroe.

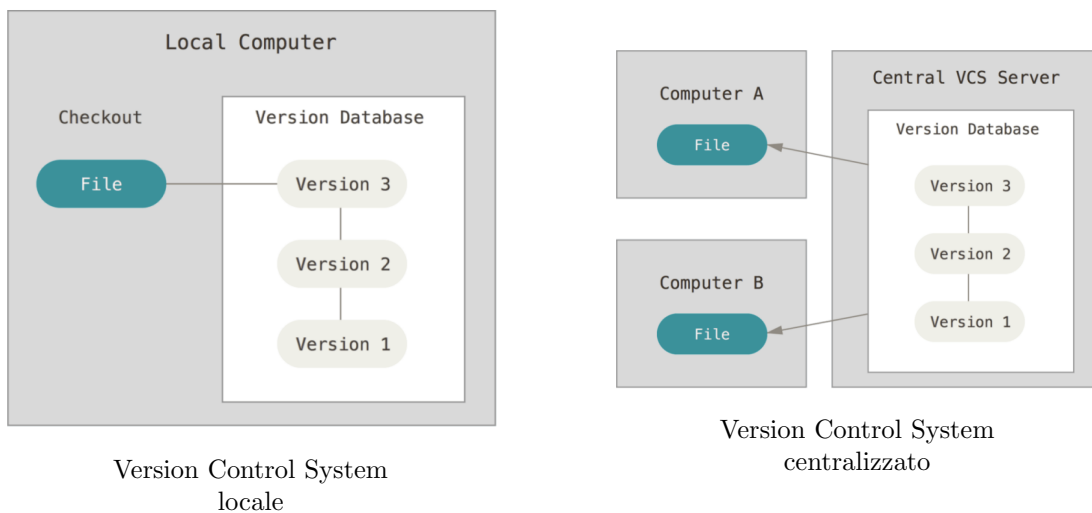
I sistemi di [controllo delle versioni](#) (Version Control System) nascono da una serie di esigenze che ogni team o singolo sviluppatore si ritrova ad affrontare dopo alcuni anni dall'inizio di un progetto:

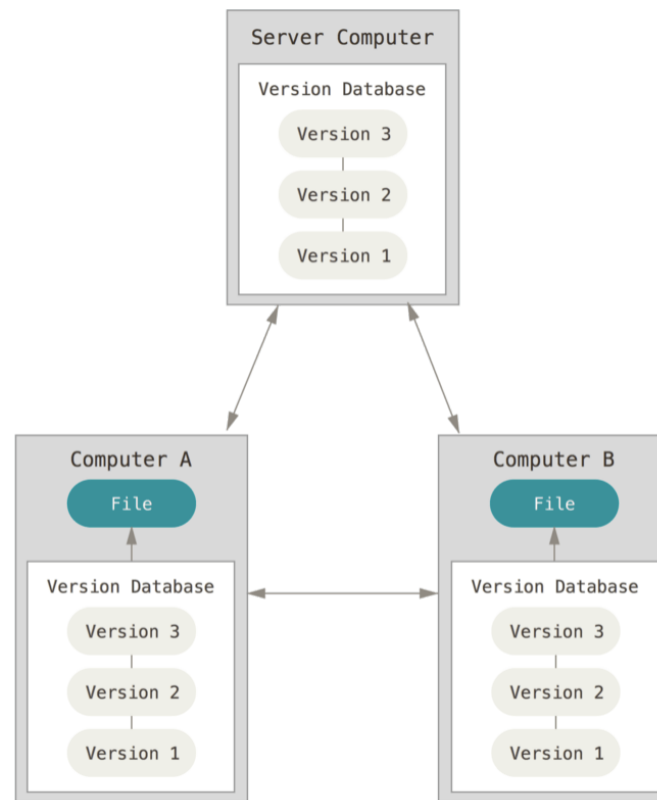
- I progetti crescono in complessità e dimensioni, diversi collaboratori/sviluppatori apportano diverse modifiche e il monitoraggio e la valutazione di tali modifiche sono necessari.
- A volte, i progetti possono avere più versioni sviluppate o testate contemporaneamente (si consideri, ad esempio, il tentativo di provare varie ottimizzazioni allo stesso algoritmo).
- E quando il progetto ha finalmente mesi/anni, dovrebbe esserci un modo per rintracciare quando una specifica funzionalità o un bug è stato introdotto nel codice.

Attualmente i sistemi di [controllo delle versioni](#) sono diventati strumenti essenziali per lo sviluppo software moderno e i progetti collaborativi (anche in progetti low-code o no-code).

Ognuno fornisce:

- **Etichettatura coerente delle revisioni**, più versioni di un file vengono etichettate utilizzando un modo standardizzato (imposto dallo strumento)
- **Monitoraggio delle modifiche**, ogni differenza tra due revisioni viene monitorata e può essere richiamata in futuro (ad esempio, se è necessario esplorare una versione precedente del codice)
- **Metadati** (data, autori, ecc.) per le revisioni, per aiutare, ad esempio, a rintracciare quando e chi ha apportato una modifica specifica
- **Revisioni coerenti point-in-time**, ogni revisione viene salvata in modo esplicito e può contenere più file
- **Branch**, le revisioni possono divergere e lo stesso progetto può avere più "versioni" parallele (sempre etichettate)
- **Merge**, branch diversi possono essere "fusi" unendo le loro modifiche;
- Sincronizzazione tra più utenti e computer.





Version Control System
distribuito

I crediti per queste 3 immagini vanno agli autori del libro "[Pro Git](#)".

Dropbox, Google Drive e OneDrive sono alcuni esempi di soluzioni il cui scopo non rientra in quello di un VSC, di fatto

- le revisioni di uno o più file non vengono salvate in modo esplicito
- tracciare i cambiamenti da parte del solo utente risulta praticamente impossibile
- non è possibile creare dei branch (e conseguentemente di fonderli)
- sono presenti pochi metadati per i file/revisioni

rimandando comunque possibili scelte quando si parla della **sincronizzazione** di uno o più file (tra più utenti e computer).

Esempi di applicazioni che invece forniscono [queste](#) funzionalità possono essere CVS, Subversion, Mercurial.

All'interno di questo corso verrà esaminato Git, una soluzione moderna che è rapidamente diventata uno standard del settore alcuni anni fa (in generale, la maggior parte dei concetti, se non tutti, sono condivisi tra i diversi VCS).

Capitolo 2

Global Information Tracker

Git è un Version Control System distribuito creato nel 2005 da Linus Torvalds per gestire il codice sorgente del kernel Linux.

Utilizza grafi aciclici diretti (DAG) e strutture dati simili agli alberi di Merkle.

La **working copy** (o working directory), è una copia locale del progetto.

Rappresenta l'insieme dei file monitorati da Git, (solitamente è la directory che contiene il progetto).

Ciò include anche le sottodirectory. Ogni volta che si aggiunge, modifica o elimina un file, bisogna renderlo noto a Git (attraverso un commit).

Git gestisce i file nella working copy. Quando si passa da una revisione all'altra, Git modifica tutti i file nella working directory per allinearne il contenuto a quello che avevano al momento di tale revisione.

Per esempio, supponiamo di avere un file denominato **travels.txt** nella working copy.

Supponiamo che il contenuto durante la prima revisione fosse:

2022 Roma

Ora, supponiamo di avere un altro viaggio da aggiungere al file e di modificare il contenuto creando la seconda revisione con il seguente contenuto:

2022 Roma

2023 Parigi

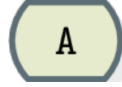
Se dovessimo tornare alla prima revisione (utilizzando il comando appropriato), git cambierà automaticamente il contenuto del file in:

2022 Roma

2.1 Commit

Un commit rappresenta uno [snapshot](#) della working copy in un dato momento.

Quando si prepara la propria working copy per un commit, è possibile decidere quali file devono essere presenti (si può decidere di includere alcuni o tutti i file presenti nel progetto).



Di solito, ogni commit ha un suo "predecessore"

- a volte, quando c'è un merge tra alcune branch, un commit può avere più predecessori.
- in rari casi, come il primo commit, può non avere alcun predecessore (viene detto "orfano").

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

crediti: [XKCD - Randall Munroe](#)

Ogni commit contiene

- un messaggio, formato un sommario obbligatorio, convenzionalmente di 72 caratteri al massimo, e un testo esplicativo dettagliato dopo una riga vuota (la lunghezza del sommario rimane comunque facoltativa)
- nome ed e-mail del committente, e la data del commit
- nome ed e-mail dell'autore (differisce dal committente nel caso l'autore del codice non sia la stessa persona che ha creato il commit nel repository)
- un albero (Tree) ovvero una sorta di copia della working directory in quel momento
- i suoi commit precedenti

Un commit viene identificato univocamente in un repository (sezione [2.4](#)) da una stringa alfanumerica (ID).

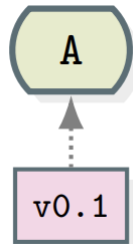
La stringa è data dalla funzione di hash [SHA1](#) del suo intero contenuto.

Ad esempio, **20ea1e7d13c1b544fe67c4a8dc3943bb1ab33e6f** è un ID di un commit.

2.1.1 Tag

A volte si può volere etichettare un commit specifico in modo da poterci tornare rapidamente in futuro.

Per esempio, rilasciando una nuova versione di un software, si potrebbe voler etichettare il commit contenente il codice da cui è stato compilato.



Una volta inserito, un tag potrebbe essere teoricamente rimosso (seppur sconsigliato, in quanto considerato "immutabile" e con possibili danneggiamenti se il repository dovesse essere condiviso).

2.1.2 Staging Area

Git implementa la possibilità di contenere solo un sottoinsieme di tutti i file modificati attraverso il concetto di "staging area" (area di "sosta").

La staging area rappresenta l'insieme di tutti i cambiamenti scelti da includere nel prossimo commit.

Prima di invocare il comando per creare un commit è necessario dunque aggiungere tutti i file aggiunti, modificati o eliminati nella staging area.

Controintuitivamente, anche i file eliminati dovrebbero essere aggiunti nella staging area, in quanto la loro stessa eliminazione è una modifica da tracciare.

Un file potrebbe essere stato modificato più volte (ad esempio in linee diverse), Git permette di aggiungere solo alcune parti di un file alla staging area (tuttavia in tutti gli esempi successivi si considererà sempre ogni file nella sua interezza).



2.2 Branch

Un branch rappresenta una possibile linea di sviluppo, un insieme ordinato di commit collegati tra loro in un grafo aciclico diretto (DAG) da una relazione padre-figlio.

Ognuno ha un nome proprio ed iniziano con un commit.

Il branch punta sempre all'ultimo commit nel branch stesso, e la sua history (una sorta di cronologia) inizia dal primo commit orfano al suo interno (che solitamente è il primo commit in assoluto del repository).

In altre parole, dato l'ultimo commit di un branch, la sua history conterrà tutti i commit precedenti fino a raggiungerne uno "orfano".

Solitamente parte della history di un branch viene condivisa con altri branch.

Un repository Git appena creato non conterrà alcun commit (e di conseguenza nessun branch).

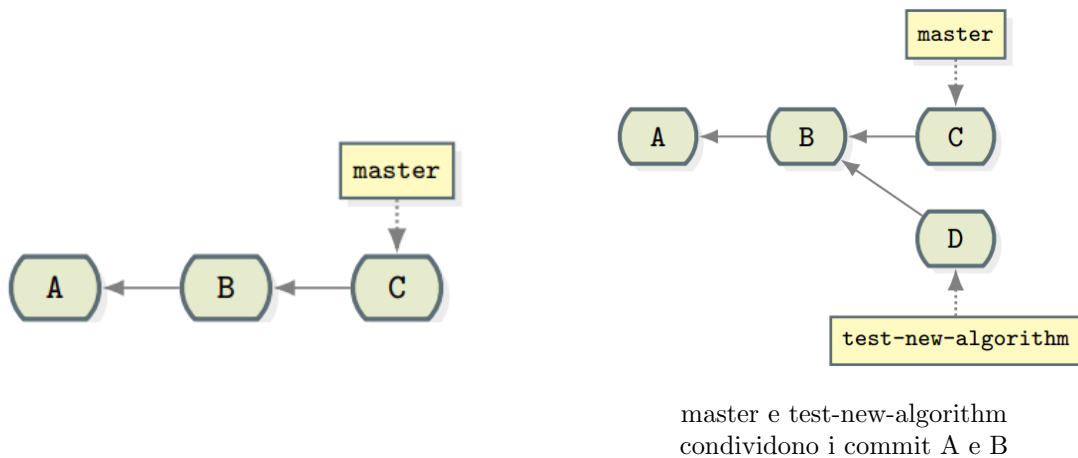
Git assegnerà automaticamente il nome "master" al primo branch che si andrà a creare una volta effettuato il primo commit.

Tutti i commit successivi saranno figli dell'ultimo commit effettuato in quel branch.

È teoricamente possibile creare un commit senza associarlo ad alcun branch.

Viene fortemente sconsigliato, in quanto il commit diventerebbe difficile da ritrovare ed inoltre il Garbage Collection di Git potrebbe cancellarlo.

Con garbage collection (GC) viene intesa l'operazione periodica effettuata da Git per rimuovere vecchi elementi e ottimizzare il repository.



Creare uno o più branch è un'operazione permette agli sviluppatori di creare una linea di sviluppo parallela, ad esempio per testare un nuovo algoritmo senza interferenze sul codice principale (e creando di conseguenza una history dedicata per ques'ultimo).

Alla fine dello uno sviluppo in parallelo, tale branch potrebbe

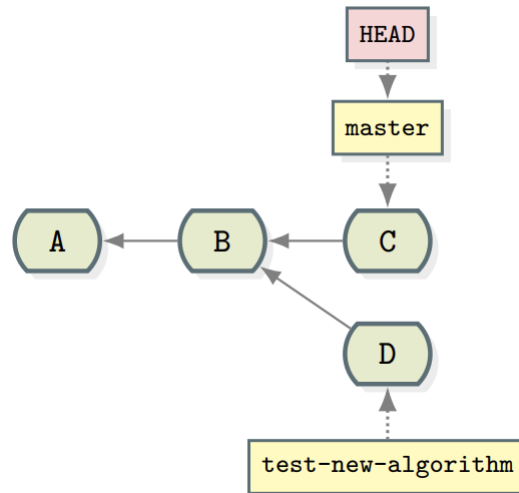
- venire abbandonato (ad esempio, se l'esperimento fallisce)
- oppure essere fuso con altri branch

2.2.1 il puntatore HEAD

Git assegna il nome HEAD al puntatore alla locazione corrente nella Git history, determinando lo stato della working copy.

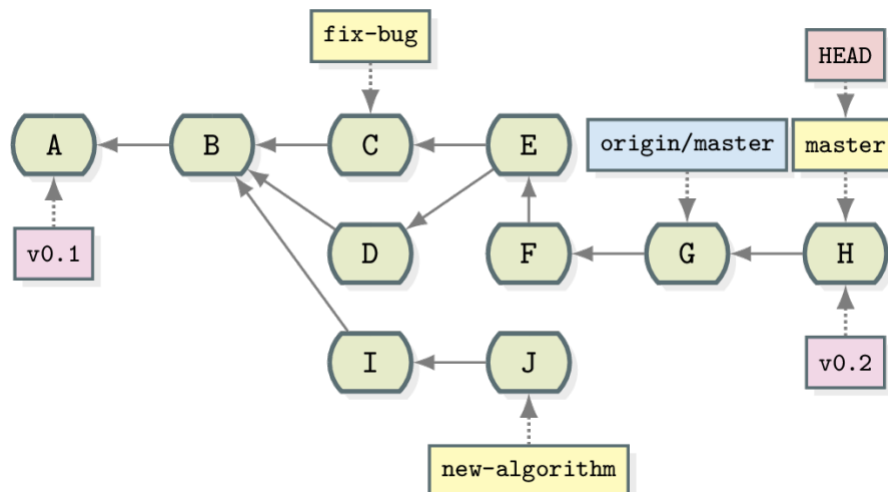
In altre parole i file nella working copy corrispondono al contenuto del commit puntato da HEAD.

HEAD potrebbe puntare ad un commit, un branch o un tag, e può essere spostato attraverso il comando "checkout" (rialineando ogni volta il contenuto della working copy).



Quando HEAD non punta ad un branch viene detto "detached" (distaccato).

In questo ulteriore caso, la creazione di nuovi commit (non appartenenti ad alcun branch) incrementerebbero la possibile perdita di dati.



- i commit A e H sono rispettivamente etichettati come "v0.1" e "v0.2"
- HEAD punta all'ultimo commit del branch master
- il commit E rappresenta la fusione tra il commit C e D
- G è l'attuale ultimo commit nella remote repository (sezione 2.4.1)

2.3 Merge

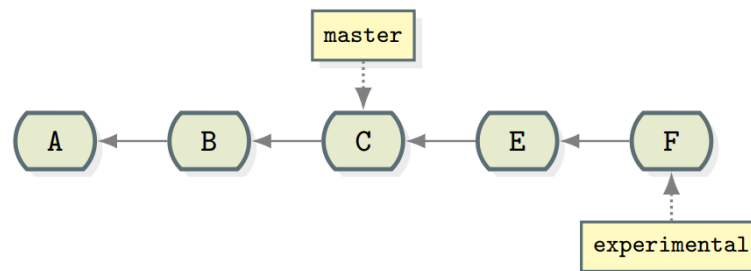
Una volta che i cambiamenti effettuati in un branch sono pronti per essere spostati in un altro (master oppure un qualsiasi altro presente), si può procedere con la loro fusione (merge).

Tale azione corrisponderà all'unione delle modifiche tra i due branch coinvolti.

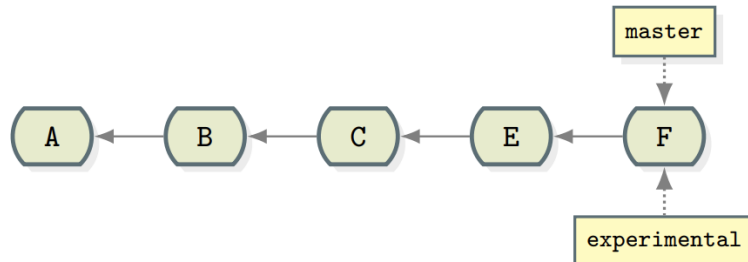
Solitamente quando ciò accade, un branch conterrà effettivamente le modifiche di entrambi, mentre l'altro rimarrà invariato.

È possibile fondere più branch attraverso diverse strategie:

- **fast-forward**, la più semplice, quando il branch da fondere è una diretta continuazione del branch di destinazione



history prima del merge fast-forward

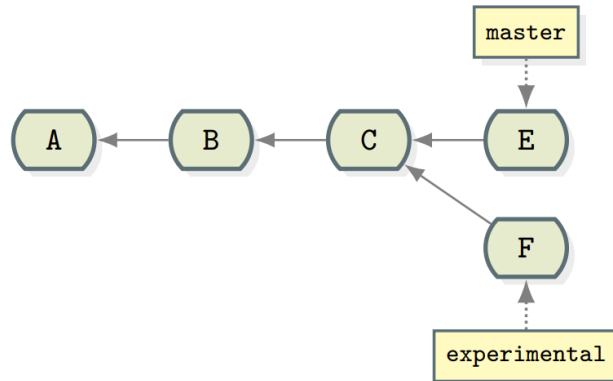


il puntatore master viene semplicemente spostato al commit F

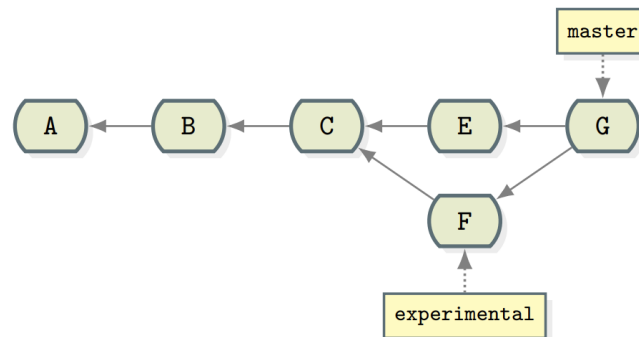
- **non-fast-forward** (o "merge commit"), più versatile del precedente, e può essere utilizzato anche quando la history non risulta lineare.

Viene creato un nuovo commit che rappresenta la fusione delle modifiche tra i due branch e il puntatore del branch di destinazione viene spostato al nuovo commit.

Inoltre, questo nuovo commit avrà due predecessori (rispettivamente i 2 ultimi commit dei branch coinvolti).



history prima del merge non fast-forward



il puntatore master viene spostato al nuovo commit G

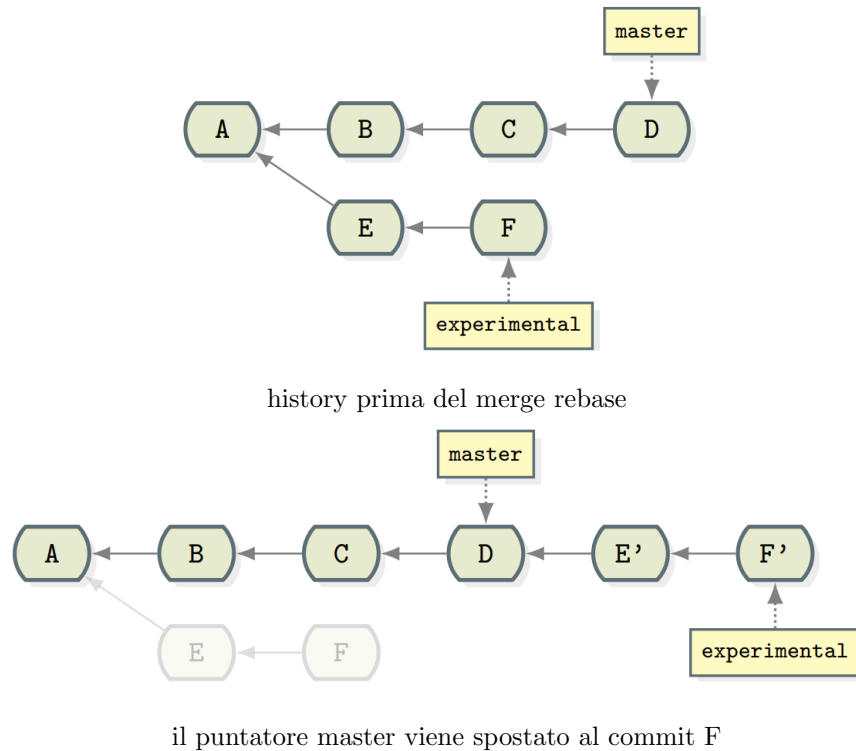
Questa strategia è soggetta a possibili conflitti (sezione 2.3.1).

- **rebase**, la cui idea è quella di modificare la history prima della fusione in modo tale che risulti lineare (potendoci applicare il fast-forward).

Per farlo, vengono creati una nuova copia di tutti i commit nella branch da fondere che non sono in comune con il branch di destinazione.

Il primo commit copiato non in comune con il branch di destinazione avrà come parent il commit più recente del branch di destinazione (e tutti gli altri seguiranno di conseguenza).

È come se si andasse a creare un nuovo branch a partire dall'ultimo commit del branch di destinazione, riapplicando tutti i cambiamenti del branch da fondere.



- i commit E' e F', per loro stessa natura, saranno comunque diversi rispettivamente da E e F (anche se rappresentano gli stessi cambiamenti) ma ciò permetterà di avere linearità nella history e di poter spostare il puntatore del branch da fondere al commit F'
- dopo la fusione, E e F risulteranno dei "dangling commit" (dei commit non associati a nessun branch o tag)

In linea teorica, è come se fossero stati "rimossi" durante il rebase, nella pratica, rimarranno nel repository fino al prossimo garbage collection.

Anche questa strategia, come il non fast-forward, è soggetta a possibili conflitti.

2.3.1 Conflitti di Merge

Durante un merge ci si potrebbe trovare in una situazione nella quale un file sia stato modificato in tutti i branch coinvolti.

Naturalmente Git proverà comunque a fondere tali cambiamenti applicandoli da entrambi i branch.

Le modifiche applicate in punti "distanti" del file non andranno a creare problemi, mentre si avrà un conflitto nel caso in cui le modifiche si sovrappongano (magari essendo sulla stessa riga, oppure quando il file viene solamente modificato in un branch ma totalmente rimosso in un altro).

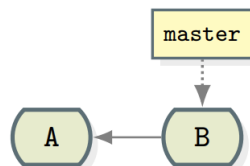
Supponiamo di avere il seguente file **example.go**

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("")
7 }

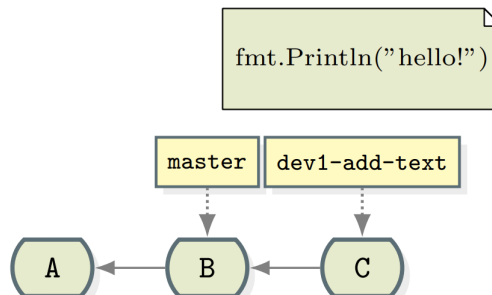
```

e la seguente history

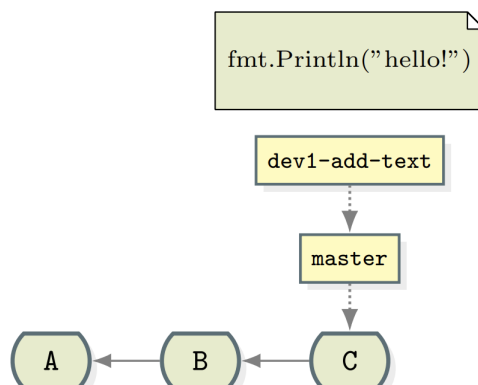


Supponiamo che 2 sviluppatori modifichino parallelamente il messaggio in due testi differenti

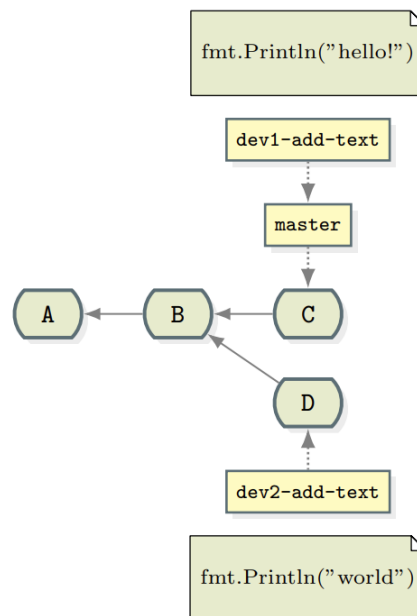
Il primo sviluppatore (**dev1**) crea un nuovo commit in un nuovo branch **dev1-add-text**, modificando **fmt.Println("")** con **fmt.Println("hello!")**



successivamente **dev1-add-text** viene fuso con il branch **master** usando la strategia fast-forward ottenendo

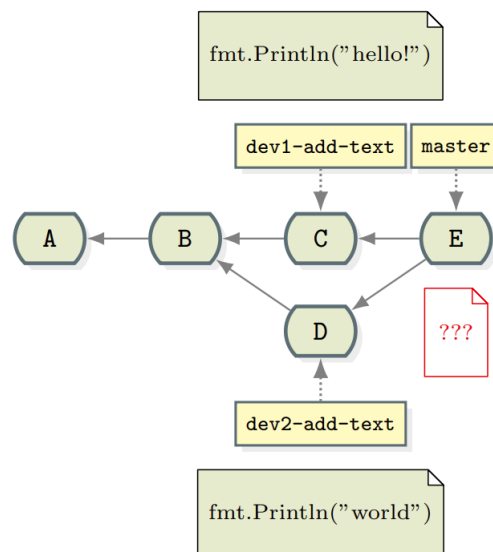


Supponiamo che, mentre **dev1** stava lavorando al suo commit, oppure mentre si stava fondendo la sua branch, un secondo sviluppatore **dev2** abbia creato un nuovo commit in un'altra nuova branch **dev2-add-text**, modificando `fmt.Println("")` con `fmt.Println("world")`



all'interno di questa history dunque lo stesso file **example.go** è stato modificato nello stesso punto simultaneamente in due branch diversi.

Se provassimo a fondere **dev2-add-text** con **master** attraverso un non fast-forward, Git non sarebbe in grado di risolvere automaticamente il conflitto emerso



Quando si verifica una situazione del genere, Git interromperà la fusione segnalando il conflitto riscontrato tramite una descrizione dello stato corrente.

Tale descrizione sarà direttamente presente all'interno del file coinvolto

```

1 package main
2
3 import "fmt"
4
5 func main() {
6 <<<<<< HEAD
7 fmt.Println("hello!")
8 =====
9 fmt.Println("world")
10 >>>>>> dev2-add-text
11 }

```

dove

- "======" rappresenta il "centro" del conflitto
- le righe comprese tra "<<<<<< HEAD" e "======" rappresentano il contenuto presente nel branch puntato da HEAD (in questo caso **master**)
- mentre tutte le righe comprese tra "======" e ">>>>>> dev2-add-text" rappresentano il contenuto presente nel branch che si vuole fondere (in questo caso il branch **dev2-add-text**)

I conflitti possono essere risolti in questo caso (e allo stesso modo in generale)

- mantenendo solamente il file fuso precedentemente dal branch **dev1-add-text** oppure il nuovo file presente nel branch **dev2-add-text**.

Entrambi i casi sono implementati con comandi Git specifici.

- modificando il file manualmente (secondo le specifiche esigenze)

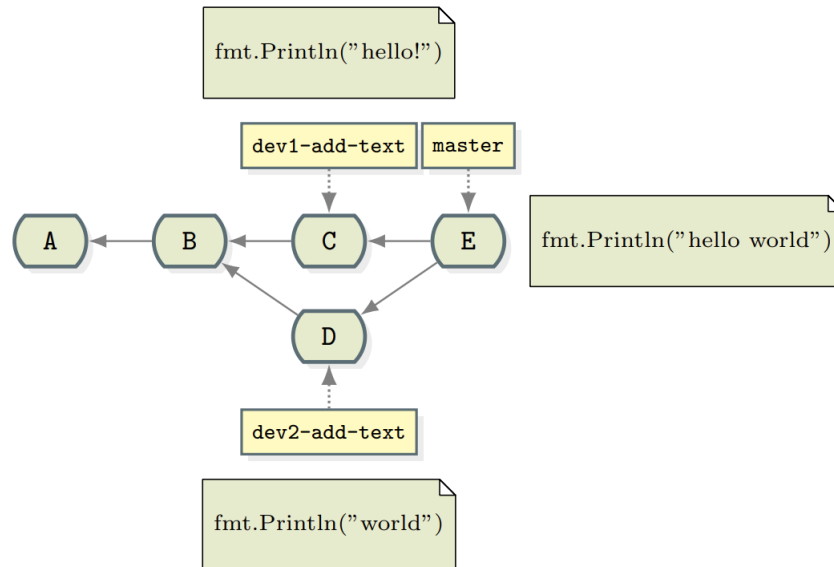
Nel caso volessimo fondere manualmente le 2 modifiche in modo da stampare il messaggio **"hello world!"** potremmo nuovamente aprire il file **example.go**, rimuovere i marcatori **"======"**, **"<<<<<< HEAD"** e **">>>>>> dev2-add-text"**, e rimpiazzare le 2 righe interessate con **fmt.Println("hello world!")**.

```

1 package main
2
3 import "fmt"
4
5 func main() {
6 fmt.Println("hello world!")
7 }

```

A questo punto, aggiungendo nuovamente il file alla staging area è possibile riprendere (e concludere) il merge.



2.4 Repository

L'insieme di tutti i commit, branch, tag e tutti gli altri elementi correlati a Git per un dato progetto sono "contenuti" all'interno di un repository.

Un progetto (ad esempio, un'applicazione mobile) potrebbe venire diviso in più repository per motivi organizzativi (ad esempio a seconda del ciclo di vita di alcune parti).

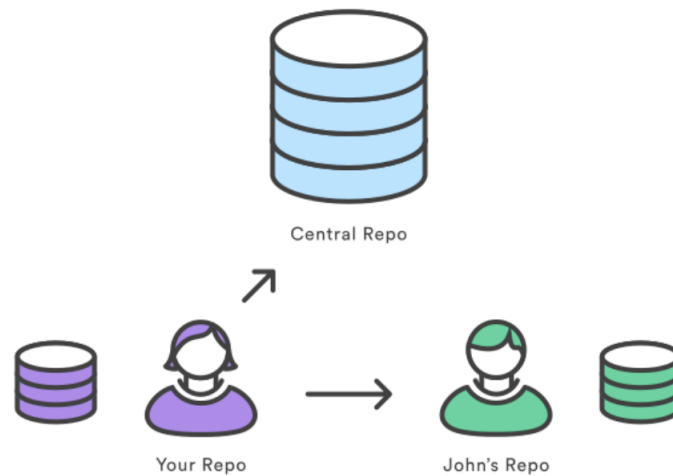
Fino adesso è stato trattato il concetto di repository da un punto di vista locale, e nonostante Git permetta di farne un utilizzo anche in questo senso, rimane comunque un Version Control System distribuito.

2.4.1 Remote Repository

Git può acquisire e inviare commit, tag e branch a repository remoti ("ospitati" su host sempre remoti, come un server, identificati da un nome e un URL, e generalmente chiamati "remotes")

Come ulteriori tipi di "remotes", Git permette di usare

- altri Personal Computer, ad esempio, dato un gruppo di tre sviluppatori, ognuno potrebbe impostare due remotes (uno per ciascun collega).
- [directory](#) presenti, ad esempio, in una chiavetta USB. (altre opzioni sono presenti nel manuale ufficiale)



Quando ad un remote non viene assegnato un nome, Git utilizza come predefinito "origin" (supportando comunque l'utilizzo di nomi differenti rispettivamente per ognuno).

Quando Git scarica commit e tag dai remote, vengono copiati "così come sono" nel repository locale.

Per tenere traccia dello stato dei remotes (nel caso debbano essere sincronizzati, sezione ???), i nomi dei branch remoti sono prefissati con il nome della remote corrispondente (ad esempio, il branch master nel remote origin sarà origin/master).

Non è dunque possibile creare un commit direttamente in un branch prefissato da un remote, per loro stessa natura vengono semplicemente utilizzati per tenere traccia dei branch remoti.

Per creare comunque un commit in un branch remoto, bisognerà dunque creare prima un branch locale (che sarà automaticamente collegato da Git a quello remoto).

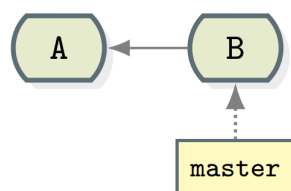
2.5 Clone

Scaricare per la prima volta il contenuto di una repository remota equivale a clonarla.

In questo modo si andrà a creare una copia completa della repository remota come working directory (il cui URL fornito per il clone rappresenterà il remote origin).

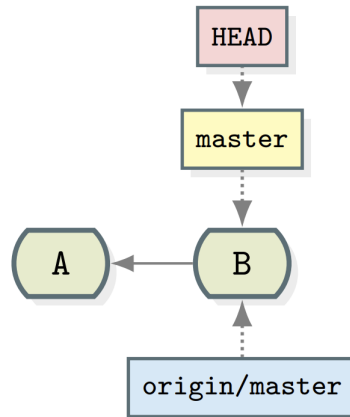
Quando ciò avviene, la working directory sarà "allineata" al commit remoto a cui fa riferimento il puntatore HEAD remoto.

Data la seguente history della repository contenuta in un host remoto



Quando verrà effettuata la sua clonazione, Git andrà a creare una repository locale scaricando tutti i commits (A e B) e tutti i branch (in questo caso, solo master, che diventerà origin/master).

Git andrà comunque a creare una branch locale master (corrispondente a origin/master) e imposterà il puntatore HEAD locale al corrispettivo puntatore HEAD remoto.



2.6 Push

L'aggiornamento dei branch o tag remoti avviene attraverso l'azione di push.

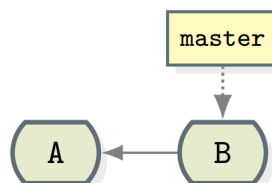
In questo caso Git invierà il branch o tag specificato al rispettivo remote.

In particolare

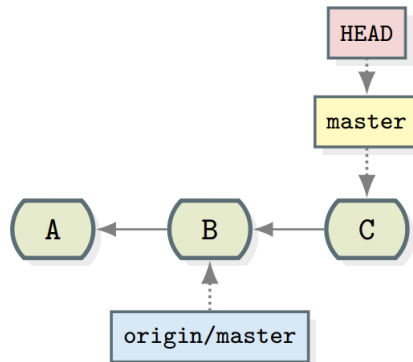
- se il branch locale sta tracciando un branch remoto, il branch remoto verrà "allineato" alla stessa posizione del branch locale.
- se il branch o tag remoto non esistono, verranno automaticamente creati.

I nuovi commit (ad esempio quelli creati per uno specifico branch/tag che esiste localmente ma non nel host remoto) vengono inviati automaticamente.

Per esempio, data la history della seguente repository remota origin

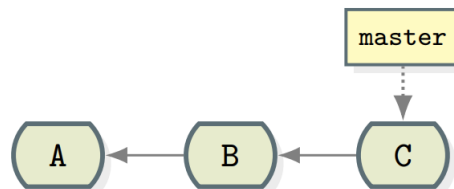


Supponiamo anche di avere una repository locale (creata tramite clone) dove abbiamo creato un nuovo commit C nel branch master



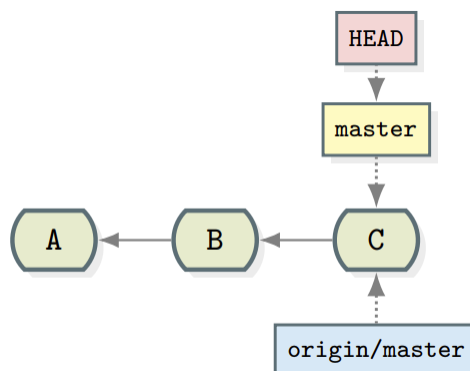
Quando ciò avviene, verrà aggiornata la posizione del branch master locale mentre il branch remoto origin/master rimarrà invariato (continuando a puntare al commit B).

Attraverso l'operazione di push, Git invierà il commit C e aggiornerà la posizione del branch remoto origin/master.



history remote origin dopo il push

Inoltre, verrà nuovamente aggiornata anche la repository locale (la branch origin/master sarà allineata con la branch master remota, che ora punta al commit C).

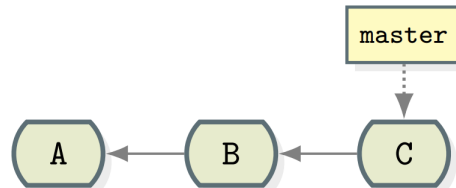


2.7 Fetch

L'operazione di fetch sincronizza i branch e i tag remoti nella repository locale (scaricando tutti i commits mancanti).

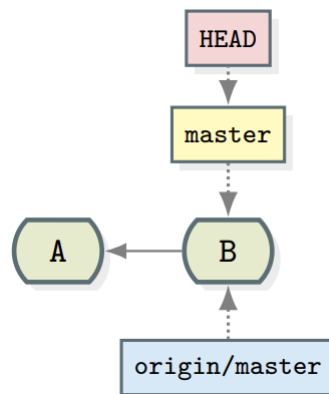
Verranno dunque aggiornati solo i branch con il nome prefissato (data una remote origin, e 2 branch master e origin/master, verrà aggiornato solo il secondo).

Per esempio, data la seguente history della repository remota origin

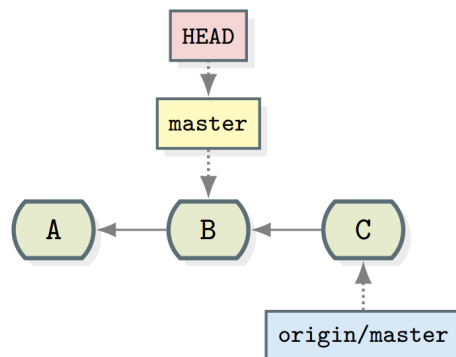


Supponiamo che la repository locale non sia aggiornata (manca il commit C, dunque il branch master locale attualmente punta al commit B).

Dunque prima di effettuare il fetch, sia il branch master locale che origin/master punteranno al commit B (in quanto Git non sa ancora che la repository remota origin è stata aggiornata).

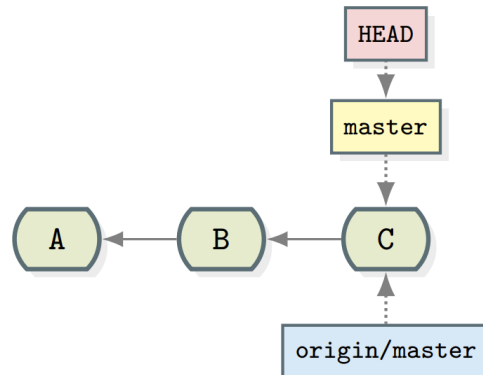


Durante l'esecuzione del fetch, la repository locale verrà aggiornata in modo tale che origin/master punti al commit C (scaricato durante parte di quest'ultima).



Nonostante ora Git sia in grado di interagire anche con la repository aggiornata anche in locale, la working directory, così come il branch master locale, rimarranno invariati (in quanto il fetch non modifica il contenuto della working directory).

Diventa ora possibile riallineare la working directory con il branch origin/master attraverso l'operazione di merge fast forward.



Rimane comunque possibile utilizzare anche le altre strategie di merge.

Ad esempio, quando viene creato un nuovo commit localmente prima di sincronizzarsi nuovamente con la repository remota, oppure quando la branch corrispondente remota viene aggiornata dopo il proprio ultimo fetch, in entrambi i casi oltre a divergere le rispettive branch locali e remote si potrebbe incorrere in uno o più conflitti di merge.

In questi casi, viene consigliato

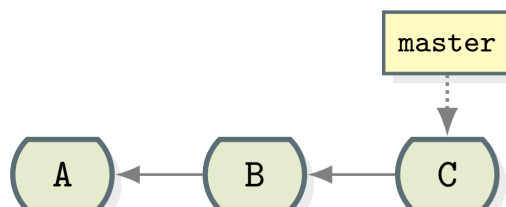
- in generale di effettuare l'operazione di fetch affiancata alla strategia di merge preferita regolarmente (ma con meno frequenza quando si sta lavorando sulla propria working copy)
- di avere un singolo sviluppatore per branch, oppure di affidarsi a strumenti di sincronizzazione del proprio IDE come ["Code With Me"](#) per [JetBrains](#) oppure ["Live Share"](#) per [Visual Studio Code](#)

Considerata la frequenza con la quale viene eseguita l'operazione di fetch + merge, è stato implementato in Git un singolo comando che le svolgesse entrambe.

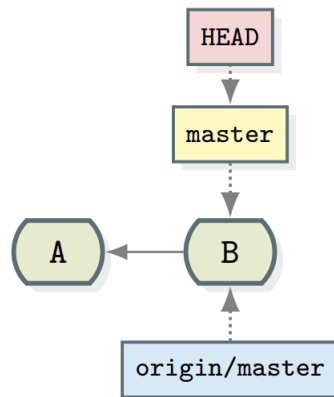
2.8 Pull

Tramite l'utilizzo del comando pull, Git esegue automaticamente un fetch delle informazioni della repository remota e successivamente un merge dal branch remoto al branch corrente (quello a cui punta HEAD).

Ad esempio, data la seguente history della repository remota origin

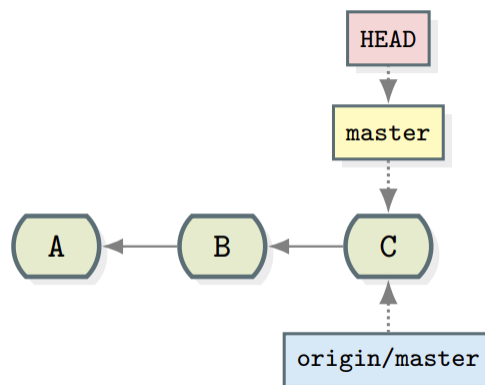


Supponiamo di avere la seguente history della repository locale (la quale è attualmente una vecchia copia di quella remota)



Eseguendo l'operazione di pull, Git

- sincronizzerà automaticamente le informazioni della repository remota origin, scaricando il commit C e aggiornando il puntatore origin/master di conseguenza
- e successivamente fonderà le branch master locale e origin/master (aggiornando anche la posizione del puntatore HEAD locale)



2.9 Hosting delle Repository Git

Fino a questo momento si è dato per assunto che una repository remota Git venisse "ospitata" da qualche parte in un host remoto raggiungibile tramite un URL.

Git potrebbe ad esempio utilizzare una macchina Linux raggiungibile attraverso una connessione SSH per memorizzare la repository remota, ma molti sviluppatori preferiscono invece utilizzare applicazioni di terze parti per ospitare le loro repository Git, in quanto forniscono strumenti aggiuntivi per la gestione dei progetti e la collaborazione.

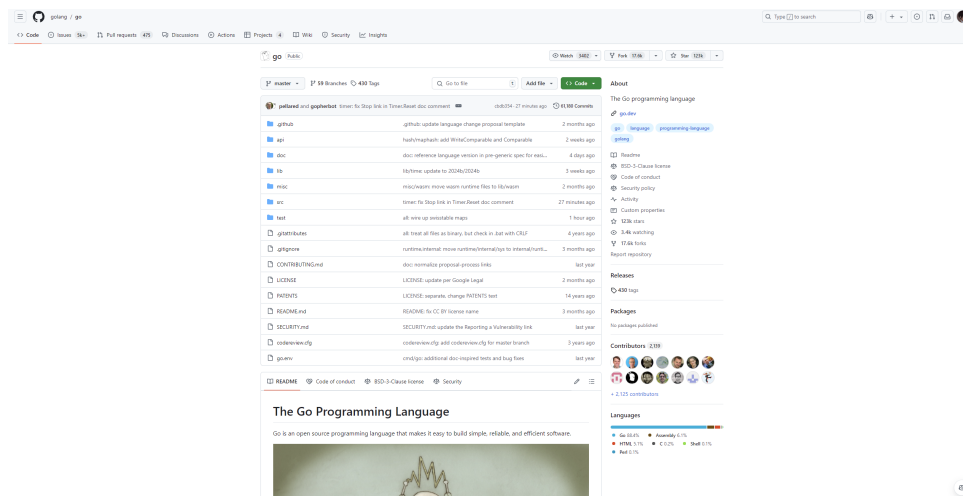
Questo tipo di applicazioni vengono generalmente chiamate "forges".

Utilizzare una "forge" fornisce diversi vantaggi:

- disponibilità online continua, così che ogni collaboratore possa sincronizzare la propria repository senza restrizioni di tempo
- soluzioni integrate per il tracciamento dei bug e delle richieste di Feature (sezione 2.9.1)
- soluzioni integrate per la documentazione come fossero delle wikis
- strumenti per la gestione dei fork e delle richieste di pull/merge (sezione 2.9.2 e 2.9.3)
- altri strumenti come pipelines per la continua integrazione di modifiche al codice e distribuzione del software, IDE web, integrazioni con altri strumenti, ecc.



Alcuni forge vengono "ospitati" sul proprio Personal Computer, mentre altri sono offerti come servizi di terze parti.



2.9.1 Issues

Ricordando come uno dei bisogni principali durante lo sviluppo di un progetto (soprattutto quando si collabora con altre persone) sia quello di tracciare e pianificare il lavoro sul codice, le forges forniscono una funzionalità per poter tracciare

- problemi con la propria applicazione (bug, comportamenti inaspettati, ecc.)
- richieste di feature da parte degli utenti o degli stakeholder
- miglioramenti futuri per l'applicazione stessa (come ad esempio migliorare la velocità di un algoritmo)
- oppure compiti periodici da svolgere sul codice stesso

astruendo ognuna di queste categorie in delle "issues".

Quando viene creata una issue, la forge corrispondente se ne aspetta un riassunto/titolo (dovrebbe contenere la vera essenza del problema).

Viene fortemente consigliato di aggiungere una lunga descrizione per fornire più contesto possibile, e, qualora si trattasse di un bug, possibilmente anche ogni informazione su come riprodurre il problema.

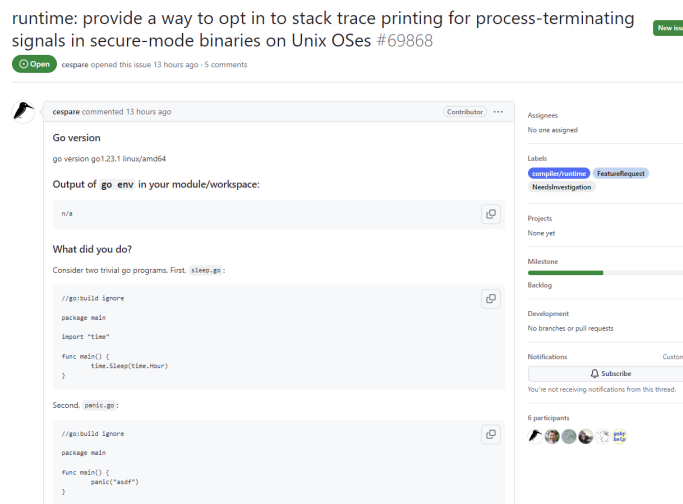
Solitamente all'interno di ogni progetto viene regolamentato il contenuto di questi campi.

Gli stati assumibili da un'issue sono

- **open**, nel momento in cui viene creata
- **closed**, nel momento in cui viene risolto un problema oppure viene implementata una feature (oppure sapendo che il problema/feature non verrà risolto/implementato)

Alcune piattaforme potrebbero fornire stati più dettagliati, come "completed", "won't fix", "invalid", ecc.

Un'issue può essere assegnata ad uno sviluppatore per tenere traccia "di chi sta facendo cosa", viene etichettata in base all'importanza o alla parte del progetto a cui si riferisce, e può inoltre avere delle scadenze e dei milestone (ad esempio, una futura versione dell'applicazione, anche se interna).



2.9.2 Fork

Può capitare a volte di non avere i permessi per inviare commit o branch a un particolare repository, tuttavia rimane possibile "inviare" comunque il proprio codice tramite richieste di pull o merge (sezione [2.9.3](#)).

In ulteriori situazioni, si potrebbe voler creare una propria versione del codice per applicare alcune modifiche che non sono accettate dagli autori originali (creando di fatto un nuovo progetto basato su quello originale).

Questo tipo di repository viene chiamato "fork".

È bene fare attenzione alle licenze di ogni progetto, in quanto alcune potrebbero proibire di creare fork, modificare o ridistribuire il codice (anche se il codice è già pubblico), e nel caso chiedere ulteriori chiarezze/informazioni agli autori originali.

Ricapitolando, uno sviluppatore esegue l'operazione di fork andando a creare una copia del codice sorgente di un progetto esistente (di solito copiando l'intero repository Git) e ne prosegue indipendentemente lo sviluppo, andandosi a creare quindi software distinti e separati.

In termini di funzionalità, un progetto "forkato" è identico a quello esistente (verrà sempre fornito un URL dove sarà possibile eseguire il clone, il push e il pull del repository).

2.9.3 Richieste di Pull/Merge

Qualora si abbia un fork o un semplice branch da fondere in una linea principale di sviluppo, si potrebbero non avere i permessi per eseguire il merge. In questo caso, le forge permettono di creare una "pull request" (o "merge request" in alcune piattaforme).

Essenzialmente, una "pull/merge request" è una richiesta (da parte di uno sviluppatore) di fondere il proprio codice in un altro branch (solitamente quello principale).

Oltre ad avere gli stessi attributi delle issues, le richieste di pull/merge mostrano anche le modifiche apportate utilizzando i diff, uno strumento che permette agli sviluppatori di "revisare" il codice aggiungendo commenti, ponendo domande o suggerendo modifiche.

Quando una richiesta di pull/merge viene completata, la piattaforma web può automaticamente fondere i 2 branch utilizzando una strategia configurata.

Seguendo lo stesso ragionamento delle issue, una richiesta di pull/merge può anche essere chiusa senza "completarla" (ovvero senza fondere i 2 branch coinvolti).

2.9. Hosting delle Repository Git

cmd/go: clarify that -coverpkg uses import paths #69655

Open

mattprout wants to merge 1 commit into `golang:master` from `mattprout:docs/coverpkg-clarification`

Conversation 35

Commits 1

Checks 1

Files changed 4

+38 -18

mattprout commented 3 weeks ago • edited

Contributor

This change amends the long-form help output for 'go help build' and 'go help testflag' to specify that the `-coverpkg` flag operates explicitly on import paths as well as package names. Import paths are fundamental for precise specification of packages versus unqualified package names, and the naming of the flag `-coverpkg` and its original documentation leads a user to assume that it only operates on the simple, unqualified package name form. The situation warrants clarification.

Fixes #69653

gopherbot commented 3 weeks ago

Contributor

This PR (HEAD: `bbf6766`) has been imported to Gerrit for code review.

Please visit Gerrit at <https://go-review.googlesource.com/c/go/+616257>.

Important tips:

- Don't comment on this PR. All discussion takes place in Gerrit.
- You need a Gmail or other Google account to [log in to Gerrit](#).
- To change your code in response to feedback:
 - Push a new commit to the branch used by your GitHub PR.
 - A new "patch set" will then appear in Gerrit.
 - Respond to each comment by marking as **Done** in Gerrit if implemented as suggested. You can alternatively write a reply.
 - Critical:** you must click the [blue Reply button](#) near the top to publish your Gerrit responses.
 - Multiple commits in the PR will be squashed by GerritBot.
- The title and description of the GitHub PR are used to construct the final commit message.
 - Edit these as needed via the GitHub web interface (not via Gerrit or git).
 - You should word wrap the PR description at ~76 characters unless you need longer lines (e.g., for tables or URLs).
- See the [Sending a change via GitHub](#) and [Reviews](#) sections of the Contribution Guide as well as the [FAQ](#) for details.

Reviews

No reviews

Still in progress? [Learn about draft PRs](#)

Assignees

No one assigned

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

Successfully merging this pull request may close these issues.

cmd/go: -coverpkg documentation is too obnoxious...

Notifications

Subscribe

You're not receiving notifications from this thread.

2 participants

[illegible]