GIT, Gitea e Software Version Control

F. Durastante (fabio.durastante@unipi.it)
S. Massei (stefano.massei@unipi.it)
S. Gazzola (silvia.gazzola@unipi.it)

30 Ottobre

Laboratorio di Introduzione alla Matematica Computazionale – A.A. 2026/2027

Software Version Control

Software Version Control

Nell'**ingegneria del software**, il controllo di versione (in inglese: Software Version Control) è una classe di sistemi responsabili della gestione delle modifiche a programmi per computer, documenti, siti web di grandi dimensioni o altre raccolte di informazioni.





Software Version Control

Nell'**ingegneria del software**, il controllo di versione (in inglese: Software Version Control) è una classe di sistemi responsabili della gestione delle modifiche a programmi per computer, documenti, siti web di grandi dimensioni o altre raccolte di informazioni.





Noi faremo uso di git che è un sistema di controllo della versione distribuito *gratuito* e **open source** progettato per gestire qualsiasi cosa, dai progetti piccoli a quelli molto grandi, con velocità ed efficienza.

Software Version Control

Nell'**ingegneria del software**, il controllo di versione (in inglese: Software Version Control) è una classe di sistemi responsabili della gestione delle modifiche a programmi per computer, documenti, siti web di grandi dimensioni o altre raccolte di informazioni.





Noi faremo uso di git che è un sistema di controllo della versione distribuito *gratuito* e **open source** progettato per gestire qualsiasi cosa, dai progetti piccoli a quelli molto grandi, con velocità ed efficienza.

In una modalità leggermente diversa dalle altre lezioni, questa sarà in forma di *tutorial* in cui cercheremo di fare le differenti operazioni passo-passo.

Git

Server Git

In **primo luogo** abbiamo bisogno di un server che gestisca un servizio **git**.

Nel migliore dei mondi possibili, avremmo un server sotto il nostro controllo assoluto in cui questo servizio è installato.

Quello che faremo invece sarà di usare un servizio gestito dal PHC del Dipartimento di Matematica

`https://git.phc.dm.unipi.it/`

Altre opzioni (più o meno commerciali) sono **GitHub, Inc.**: un provider di hosting Internet per lo sviluppo di software e il controllo della versione tramite Git. Offre le funzionalità di gestione del codice sorgente di Git, oltre ad alcune altre funzionalità proprietarie. Ha sede in California, ed è una **filiale di Microsoft dal 2018**.

Server Git

In **primo luogo** abbiamo bisogno di un server che gestisca un servizio **git**.

Nel migliore dei mondi possibili, avremmo un server sotto il nostro controllo assoluto in cui questo servizio è installato.

Quello che faremo invece sarà di usare un servizio gestito dal PHC del Dipartimento di Matematica

`https://git.phc.dm.unipi.it/`







Esistono delle **alternative** come: about.gitlab.com/ o la stessa gitea.com, Forgejo (forgejo.com) che si possono installare anche su un vostro server! O altri prodotti commerciali come Bitbucket: bitbucket.org/.

Login su Gitea

La **prima operazione** da compiere è quella di fare login su Gitea.



Login su Gitea

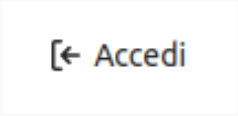
La **prima operazione** da compiere è quella di fare login su Gitea.

- Fare *click* su Accedi: 

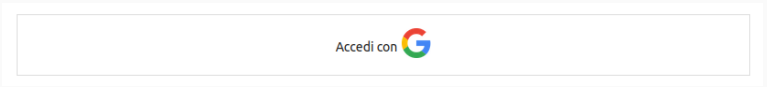
Login su Gitea

La **prima operazione** da compiere è quella di fare login su Gitea.

- Fare *click* su Accedi:



- Il sistema utilizza l'autenticazione OAuth2 tramite Google con le **credenziali di ateneo**.



OAuth
OAuth (abbreviazione di "Open Authorization") è uno standard aperto per la delega dell'accesso, comunemente utilizzato come modo per gli utenti di Internet di concedere a siti Web o applicazioni l'accesso alle proprie informazioni su altri siti Web ma senza fornire loro le password.

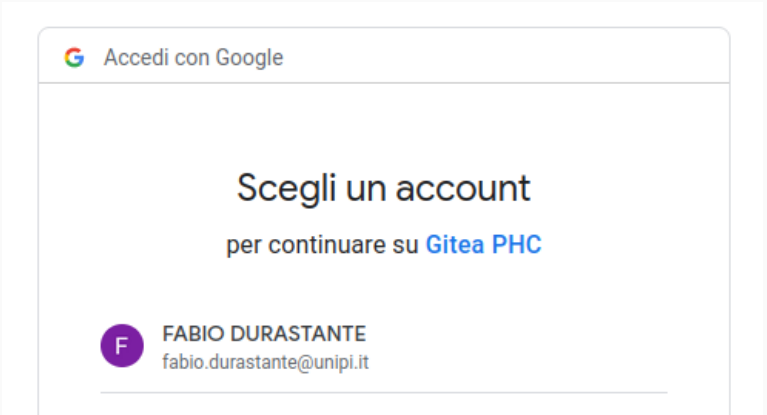
Login su Gitea

La **prima operazione** da compiere è quella di fare login su Gitea.

- Fare *click* su Accedi:



- Il sistema utilizza l'autenticazione OAuth2 tramite Google con le **credenziali di ateneo**.
- Si compila l'autenticazione con le proprie credenziali di ateneo.



– ora abbiamo il nostro account pronto ed operativo –

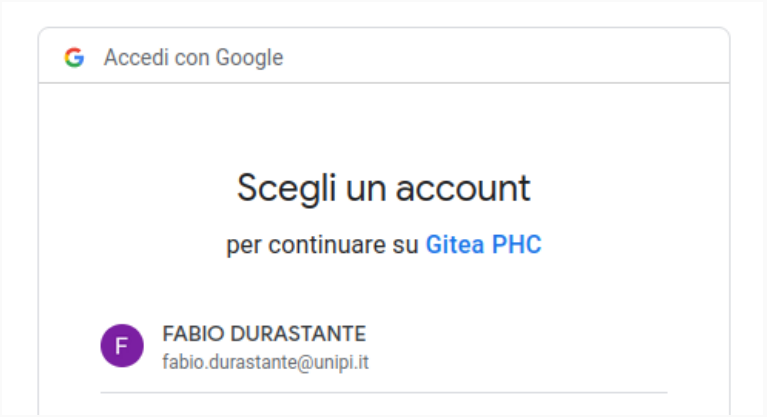
Login su Gitea

La **prima operazione** da compiere è quella di fare login su Gitea.

- Fare *click* su Accedi:



- Il sistema utilizza l'autenticazione OAuth2 tramite Google con le **credenziali di ateneo**.
- Si compila l'autenticazione con le proprie credenziali di ateneo.



– ora abbiamo il nostro account pronto ed operativo –

- possiamo adesso **configurare il nostro profilo**,

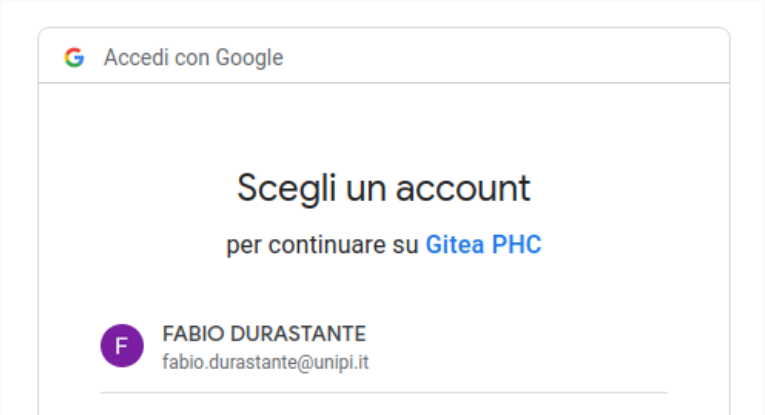
Login su Gitea

La **prima operazione** da compiere è quella di fare login su Gitea.

- Fare *click* su Accedi:



- Il sistema utilizza l'autenticazione OAuth2 tramite Google con le **credenziali di ateneo**.
- Si compila l'autenticazione con le proprie credenziali di ateneo.



– ora abbiamo il nostro account pronto ed operativo –

- possiamo adesso **configurare il nostro profilo**,
- impostare le **chiavi di sicurezza** (SSH),

Git su Windows e MAC

Il resto delle slide suppone l'utilizzo di una delle macchine dei laboratori in ambiente Linux. Tuttavia è **possibile utilizzare Git anche da Windows**:

https://git-scm.com/download/win

Un modo semplice è quello di installa Winget (se non lo avete già) e digitare questo comando nel prompt dei comandi o in PowerShell:

```
winget install --id Git.Git -e --source winget
```

Analogamente si può avere **Git anche su MAC**

https://git-scm.com/download/mac

Ovvero utilizzando *homebrew* da una `shell`:

```
brew install git
```

Git su Windows e MAC

Il resto delle slide suppone l'utilizzo di una delle macchine dei laboratori in ambiente Linux. Tuttavia è **possibile utilizzare Git anche da Windows**:

https://git-scm.com/download/win

Un modo semplice è quello di installa Winget (se non lo avete già) e digitare questo comando nel prompt dei comandi o in PowerShell:

```
winget install --id Git.Git -e --source winget
```

Analogamente si può avere **Git anche su MAC**

https://git-scm.com/download/mac

Ovvero utilizzando *homebrew* da una `shell`:

```
brew install git
```

Avendo installato Git è **possibile** poi *mutatis mutandis* **adattare** il resto delle istruzioni al proprio ambiente.

Profilo e chiavi SSH

Dal menù in alto a destra possiamo accedere a diverse funzionalità:

- Il nostro **profilo** che mostra una pagina riassuntiva delle nostre informazioni, ovvero quella che contiene i dati pubblici a cui può accedere chi cerca il nostro utente.
- La pagina delle **impostazioni** da cui possiamo configurare il nostro account.



Creiamo una chiave SSH

Per fare sì che la coppia di chiavi sia conservata, la genereremo su una macchina dell'Aula 4/oppure su una macchina mathsgalore.

1. Connettiamoci via `ssh` a `a4-labYY.polo2.ad.unipi.it`:
   `ssh n.cognomeXX@a4-labY.polo2.ad.unipi.it`

Creiamo una chiave SSH

Per fare sì che la coppia di chiavi sia conservata, la genereremo su una macchina dell'Aula 4/oppure su una macchina mathsgalore.

1. Connettiamoci via `ssh` a `a4-labYY.polo2.ad.unipi.it`:
   `ssh n.cognomeXX@a4-labY.polo2.ad.unipi.it`
2. Usiamo il comando `ssh-keygen`:
   `ssh-keygen -t ed25519 -C "n.cognomeXX@studenti.unipi.it"`

```
Generating public/private ed25519 key pair.
Enter file in which to save the key
↪  (/home/unipi/n.cognomeX/.ssh/id_ed25519):
```

Creiamo una chiave SSH

Per fare sì che la coppia di chiavi sia conservata, la genereremo su una macchina dell'Aula 4/oppure su una macchina mathsgalore.

1. Connettiamoci via `ssh` a `a4-labYY.polo2.ad.unipi.it`:
   `ssh n.cognomeXX@a4-labY.polo2.ad.unipi.it`
2. Usiamo il comando `ssh-keygen`:
   `ssh-keygen -t ed25519 -C "n.cognomeXX@studenti.unipi.it"`
   ```
   Generating public/private ed25519 key pair.
   Enter file in which to save the key
   ↪ (/home/unipi/n.cognomeX/.ssh/id_ed25519):
   ```
   2.1 dobbiamo scegliere un *path* in cui salvare i file delle chiavi, oppure premere invio ed accettare quello di default, scegliamo ad esempio:
   `/home/unipi/n.cognomeXX/.ssh/id_gitea_ed25519`

Creiamo una chiave SSH

Per fare sì che la coppia di chiavi sia conservata, la genereremo su una macchina dell'Aula 4/oppure su una macchina mathsgalore.

1. Connettiamoci via `ssh` a `a4-labYY.polo2.ad.unipi.it`:
   `ssh n.cognomeXX@a4-labY.polo2.ad.unipi.it`
2. Usiamo il comando `ssh-keygen`:
   `ssh-keygen -t ed25519 -C "n.cognomeXX@studenti.unipi.it"`
   ```
   Generating public/private ed25519 key pair.
   Enter file in which to save the key
   ↪ (/home/unipi/n.cognomeX/.ssh/id_ed25519):
   ```
   2.1 dobbiamo scegliere un *path* in cui salvare i file delle chiavi, oppure premere invio ed accettare quello di default, scegliamo ad esempio:
   `/home/unipi/n.cognomeXX/.ssh/id_gitea_ed25519`
   2.2 Scegliamo una *passphrase* che **dobbiamo ricordare**:
   ```
   Enter passphrase (empty for no passphrase):
   Enter same passphrase again:
   ```

Creiamo una chiave SSH

3. Al termine questo ci restituirà la conferma che la chiave è stata creata:

```
Your identification has been saved in
↪  /home/unipi/n.cognomeXX/.ssh/id_gitea_ed25519
Your public key has been saved in
↪  /home/unipi/n.cognomeXX/.ssh/id_gitea_ed25519.pub
The key fingerprint is:
SHA256:zlhv22FK51g972UUysGfHdz7QhBm4niidpn0odPd8rQ
↪  n.cognomeXX@studenti.unipi.it
The key's randomart image is:
+--[ED25519 256]--+
|          . +    |
|         o +... .|
|        + + .o +.|
|       o O o.o+ *|
|      o S o oo++o|
|     . * o   * o.|
|      . o + = E +|
|         o X . =.|
|          + o  .o|
+----[SHA256]-----+
```

Creiamo una chiave SSH

3. Al termine questo ci restituirà la conferma che la chiave è stata creata.

4. Possiamo verificare la presenza della chiave con il comando `ls` sul path in cui abbiamo salvato la chiave, e.g.,

   ```
   ls/home/unipi/n.cognomeXX/.ssh/
   authorized_keys  id_gitea_ed25519  id_gitea_ed25519.pub
   ↪  known_hosts
   ```

Creiamo una chiave SSH

3. Al termine questo ci restituirà la conferma che la chiave è stata creata.

4. Possiamo verificare la presenza della chiave con il comando `ls` sul path in cui abbiamo salvato la chiave, e.g.,

   ```
   ls/home/unipi/n.cognomeXX/.ssh/
   authorized_keys  id_gitea_ed25519  id_gitea_ed25519.pub
   ↪  known_hosts
   ```

5. Aggiungiamo la chiave appena generata all'OpenSSH authentication agent

   ```
   eval `ssh-agent`
   Agent pid 1606958
   ssh-add/home/unipi/n.cognomeXX/.ssh/id_gitea_ed25519
   Enter passphrase for /home/unipi/n.cognomeXX/.ssh/id_gitea_ed25519:
   Identity added: /home/unipi/n.cognomeXX/.ssh/id_gitea_ed25519
   ↪  (n.cognomeXX@studenti.unipi.it)
   ```

Creiamo una chiave SSH

3. Al termine questo ci restituirà la conferma che la chiave è stata creata.

4. Possiamo verificare la presenza della chiave con il comando `ls` sul path in cui abbiamo salvato la chiave, e.g.,

   ```
   ls/home/unipi/n.cognomeXX/.ssh/
   authorized_keys  id_gitea_ed25519  id_gitea_ed25519.pub
   ↪  known_hosts
   ```

5. Aggiungiamo la chiave appena generata all'**OpenSSH authentication agent**

   ```
   eval `ssh-agent`
   Agent pid 1606958
   ssh-add/home/unipi/n.cognomeXX/.ssh/id_gitea_ed25519
   Enter passphrase for /home/unipi/n.cognomeXX/.ssh/id_gitea_ed25519:
   Identity added: /home/unipi/n.cognomeXX/.ssh/id_gitea_ed25519
   ↪  (n.cognomeXX@studenti.unipi.it)
   ```

6. Possiamo ora aggiungere la chiave Gitea.

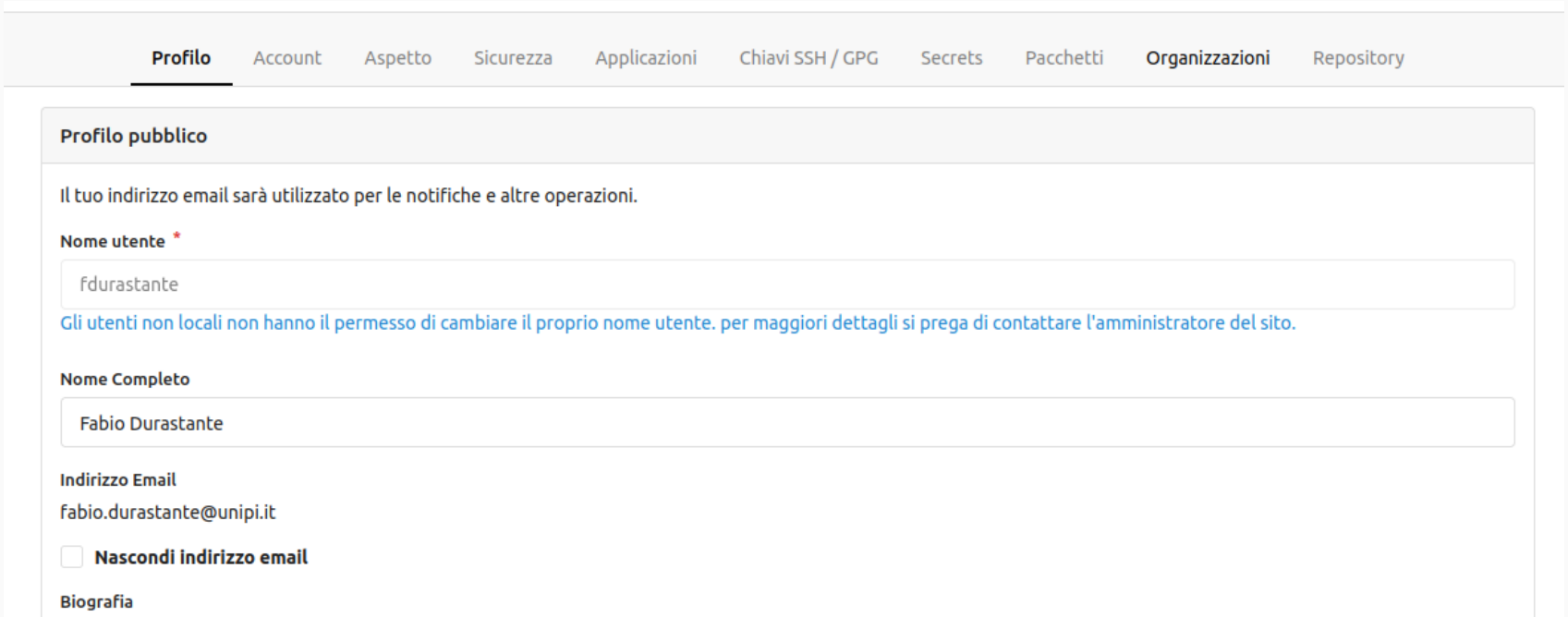
Aggiungere la chiave a Gitea

- Dal menù in alto a destra su Gitea selezioniamo le impostazioni



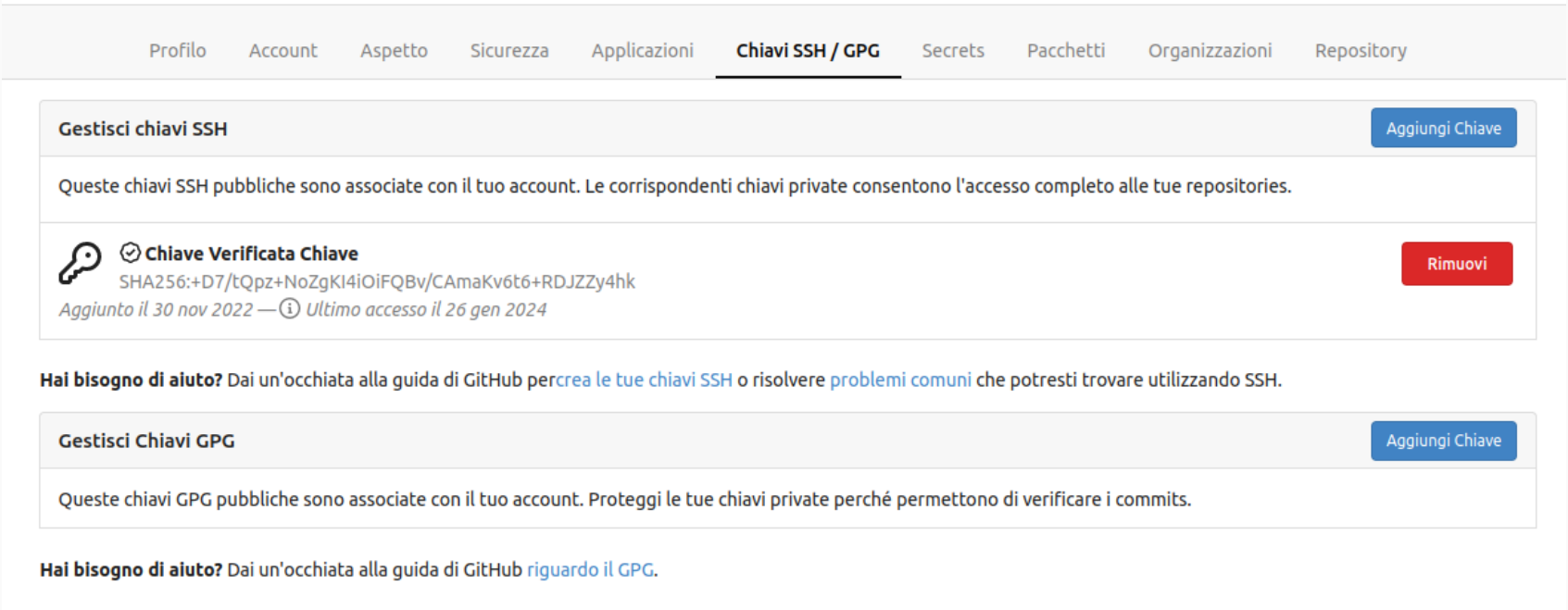
Aggiungere la chiave a Gitea

Da questa pagina possiamo **configurare il nostro account** e **impostare le chiavi `ssh`**.



Aggiungere la chiave a Gitea

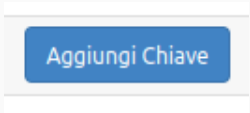
Da questa pagina possiamo **configurare il nostro account** e **impostare le chiavi `ssh`**.



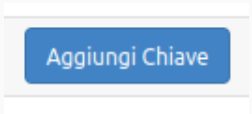
Facciamo click sulla voce di menù: **Chavi SSH/GPG**.

Nel caso a schermo c'è già una chiave configurata, nel vostro-se è la prima volta che usate il servizio-non ci sarà nessuna chiave inserita.

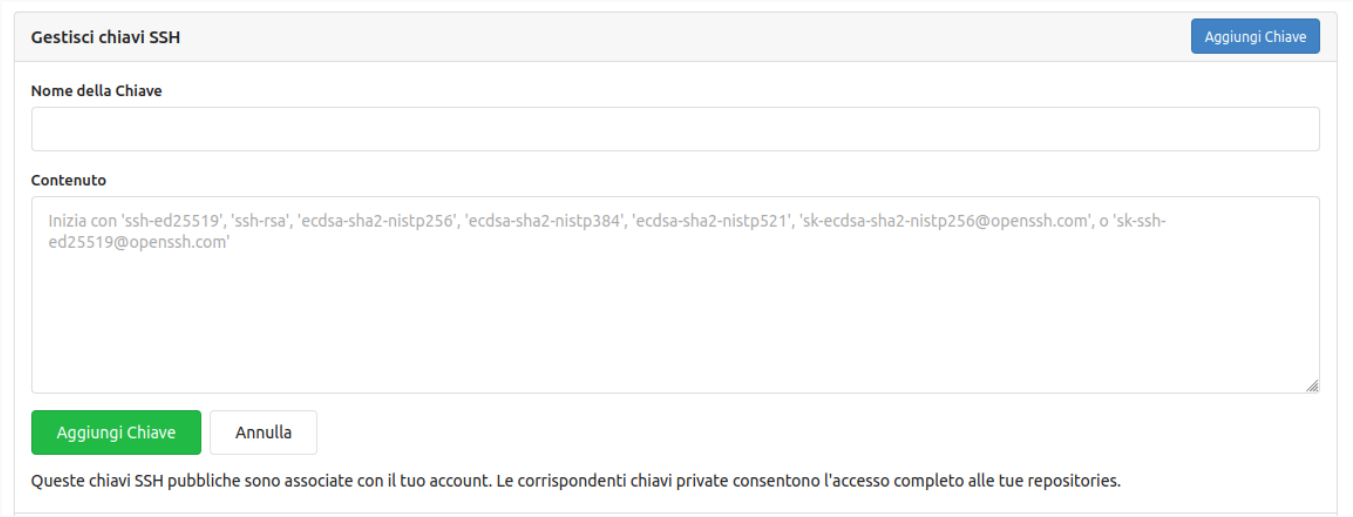
Aggiungere la chiave a Gitea

- Facciamo click su: 

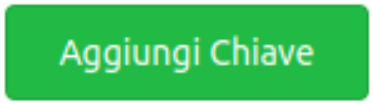
Aggiungere la chiave a Gitea

- Facciamo click su: 

- Dobbiamo ora inserire qui la nostra **chiave pubblica**




```
cat ~/.ssh/id_gitea_ed25519.pub
ssh-ed25519
↪ ABBAC3NzaC11YUI1LRS5CADAIFaeS1vfCL/DYtMN6Q761CTqn5wc8ZGXRD6wDDGqMOI7
↪ n.cognomeXX@studenti.unipi.it
```

copiamo il contenuto e **incolliamolo** nella maschera e facciamo click su 

.

Il nostro primo *repository*

Un **repository** viene solitamente utilizzato per organizzare un **singolo progetto**. I *repository* possono contenere cartelle e file, immagini, video, fogli di calcolo e set di dati, ed in generale tutto ciò di cui il vostro progetto ha bisogno.

È buona norma per repository includere un file `README`, questo contiene **informazioni sul progetto**. GitHub semplifica l'aggiunta di uno nello stesso momento in cui create il vostro nuovo repository. Offre anche altre opzioni comuni come un file di licenza.

Se avete deciso di non usare Gitea del PHC...
Potete comunque seguire il resto della lezione "poggiando" i vostri *repository* di prova sulle macchine *mathsgalore* come remoto:

```
n.cognomeXX@login:$ ssh utente@mathsgalore<-2-3-4>.unipi.it
n.cognomeXX@mathsgalore<-2-3-4>:$ mkdir mygitserver; cd mygitserver;
n.cognomeXX@mathsgalore<-2-3-4>:$ git init --bare hello-world.git
```

Quale licenza scegliere? GPL e BSD

GPL (**G**eneral **P**ublic **L**icense)

- La GPL è una licenza copyleft che impone che le opere derivate siano anch'esse distribuite con licenza GPL.
- È progettata per garantire che il software rimanga sempre libero e accessibile a tutti.
- Tutti gli utilizzatori del software devono rispettare le condizioni della GPL.

Caratteristiche della **GPL**

- La condivisione delle modifiche è obbligatoria.
- È permesso l'uso commerciale del software, ma è necessario rispettare le condizioni della GPL.
- La GPL offre una maggiore protezione delle libertà degli utenti rispetto ad altre licenze.

Quale licenza scegliere? GPL e BSD

Caratteristiche della **BSD** License

- Non impone restrizioni sul software derivato.
- Permette l'uso commerciale senza vincoli stringenti.
- È più permissiva rispetto alla GPL e offre maggiore flessibilità.

Quale licenza scegliere? GPL e BSD

Caratteristiche della **BSD** License

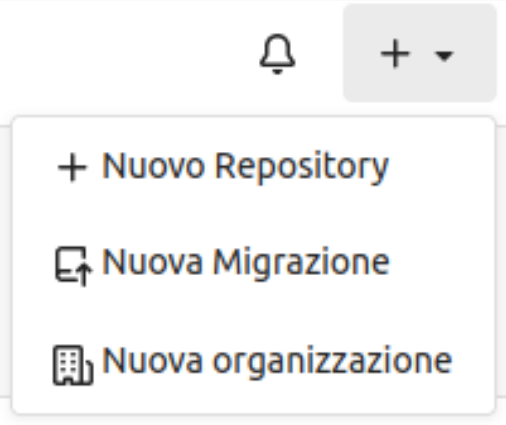
- Non impone restrizioni sul software derivato.
- Permette l'uso commerciale senza vincoli stringenti.
- È più permissiva rispetto alla GPL e offre maggiore flessibilità.

Confronto tra GPL e BSD

- GPL: Garantisce la libertà del software e richiede che le opere derivate siano distribuite con la stessa licenza.
- BSD: Permette maggiore libertà agli sviluppatori, consentendo l'uso del software in progetti proprietari.
- Scegliere la licenza giusta è importante per garantire la libertà e la protezione del software.

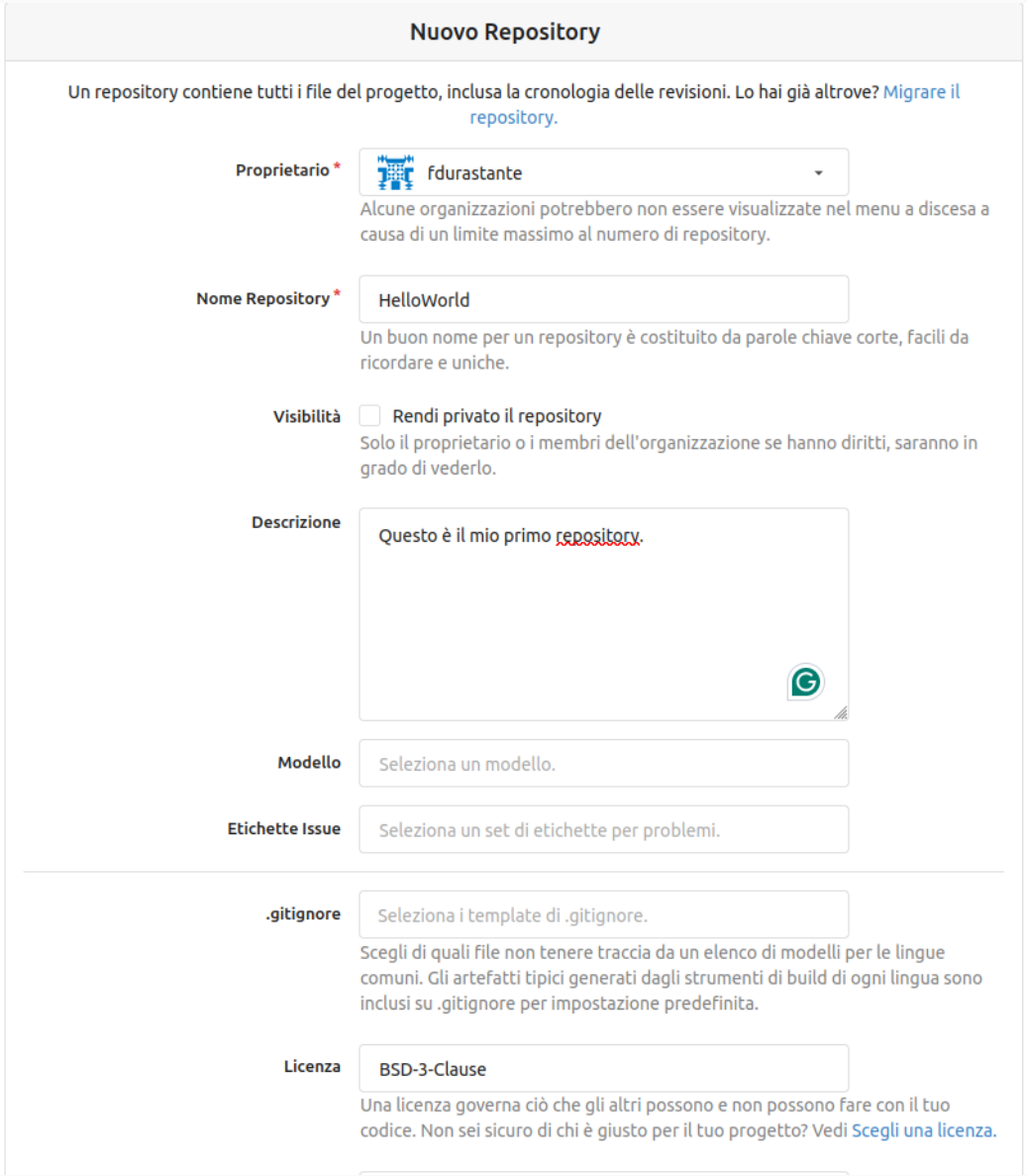
Hello-world repository

Costruiamolo passo-passo.



1. Nell'angolo in alto a destra di qualsiasi pagina, si scelga il menu `+` a discesa e seleziona **Nuovo repository**.
2. Nella casella **Nome Repository**, si inserisca `hello-world`.
3. Nella casella **Descrizione**, si inserisca una breve descrizione: "Il mio primo *repository*"
4. Dalle opzioni **Visibilità** si può scegliere se `Rendi privato il repository`
5. Si faccia click su Crea Repository

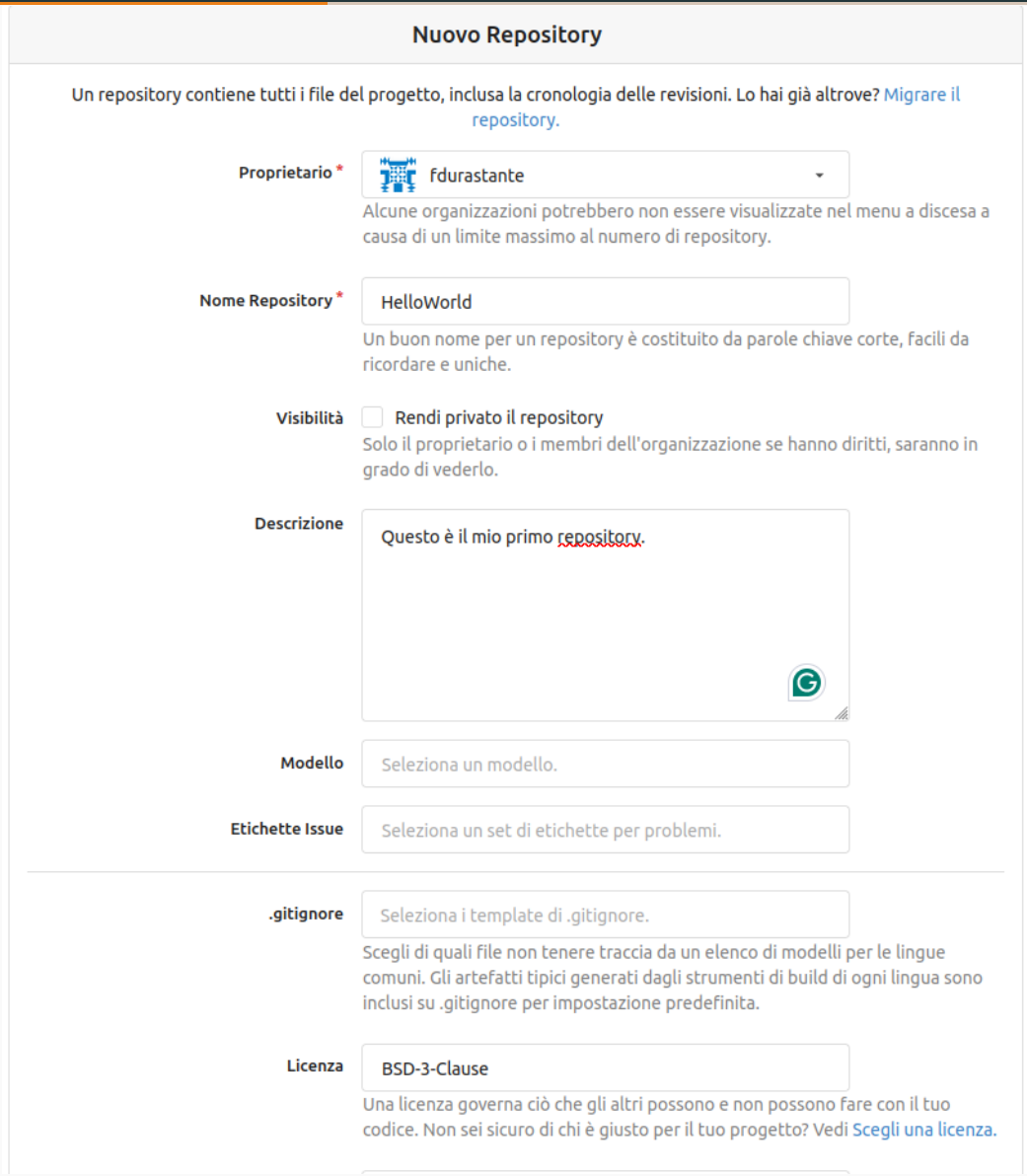
Hello-world repository



In questo esempio abbiamo scelto la licenza BSD.

- **Libertà di redistribuzione**: La licenza consente la redistribuzione in forma binaria o sorgente, purché vengano mantenute le clausole di copyright e la dichiarazione di non responsabilità.

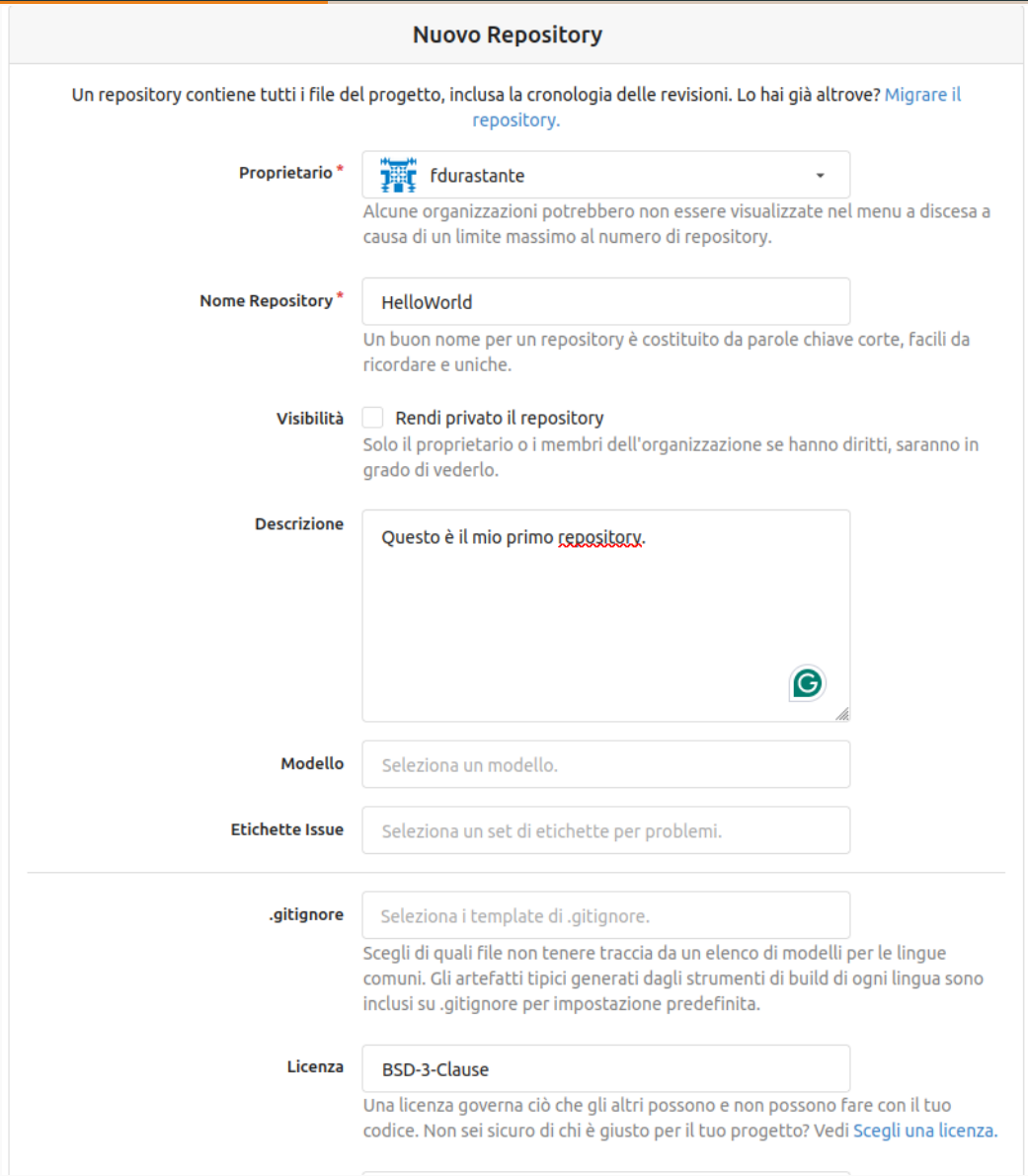
Hello-world repository



In questo esempio abbiamo scelto la licenza BSD.

- **Libertà di modificare**: Gli utenti possono modificare il codice sorgente e utilizzarlo nei propri progetti, senza restrizioni significative.

Hello-world repository

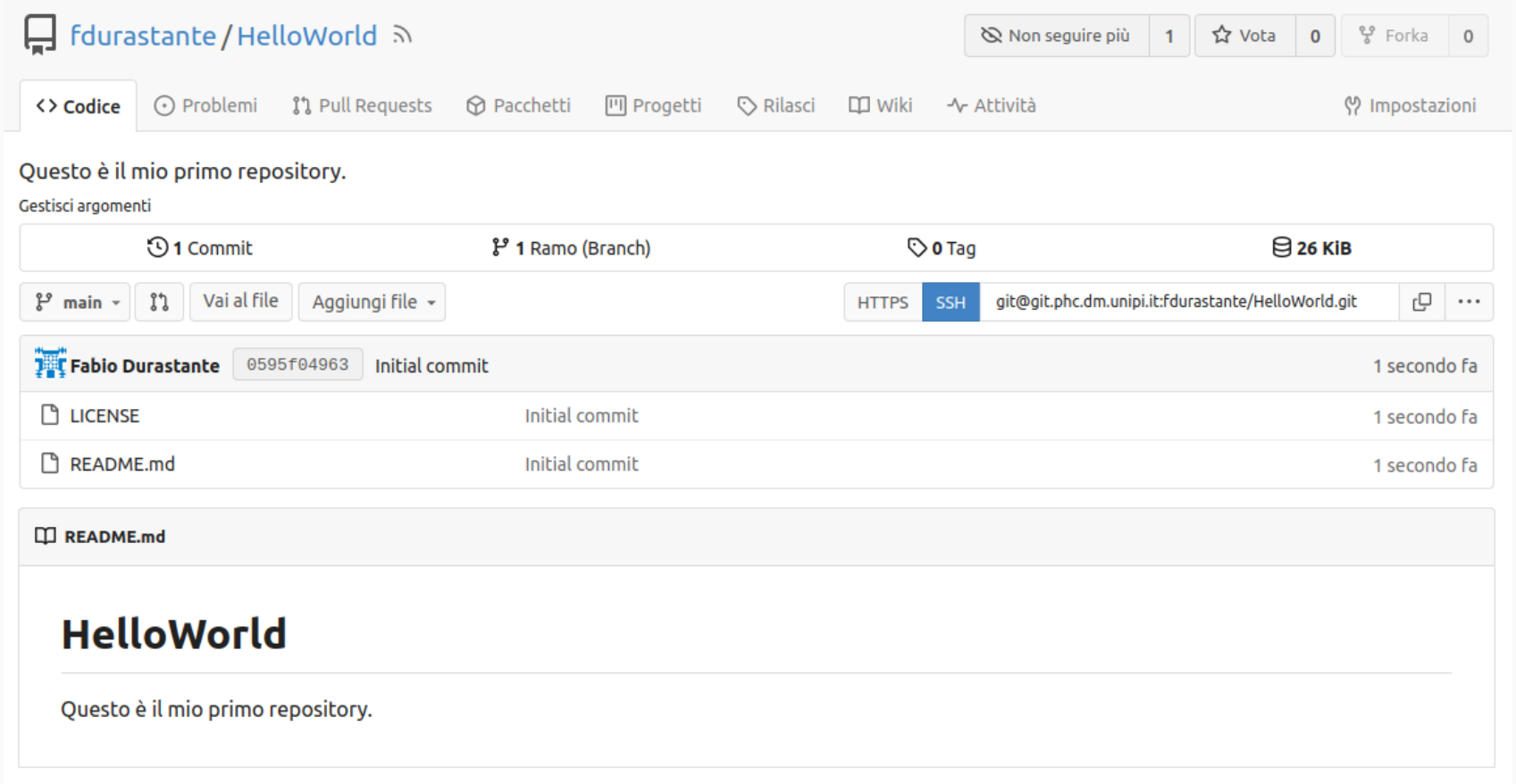


In questo esempio abbiamo scelto la licenza BSD.

- **Dichiarazione di non responsabilità**: La licenza include una dichiarazione di non responsabilità, che esonera gli sviluppatori da qualsiasi responsabilità legale derivante dall'uso del software.

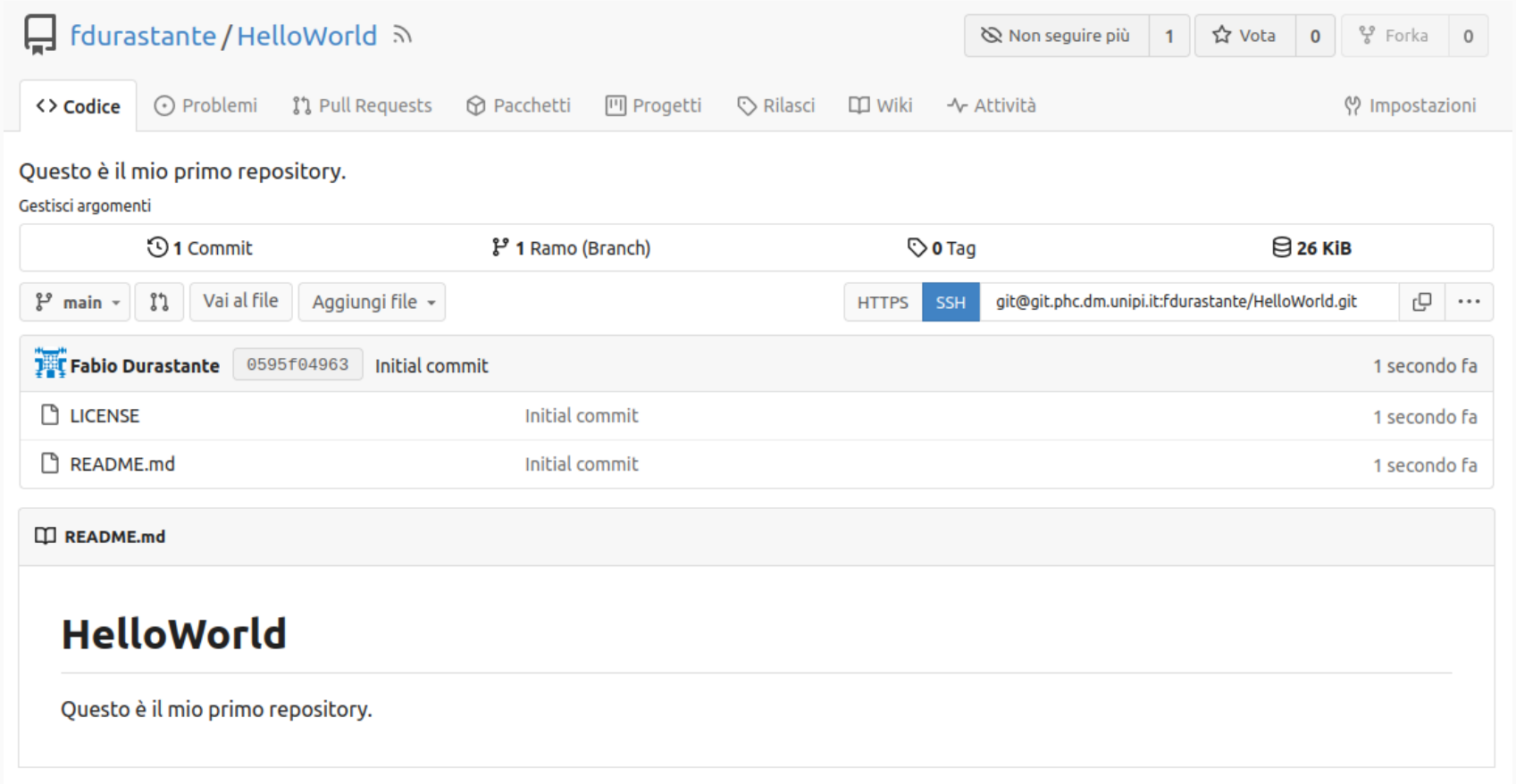
Hello-world repository

Dopo aver dato la conferma, vedremo il nostro nuovo *repository* che conterrà al suo interno solamente il file `README` e la `LICENSE`.



Hello-world repository

Dopo aver dato la conferma, vedremo il nostro nuovo *repository* che conterrà al suo interno solamente il file `README` e la `LICENSE`.



- Adesso abbiamo un luogo per i nostri file sul *server remoto*. Dobbiamo imparare ad interagire con esso per **recuperare i file**,

Clone

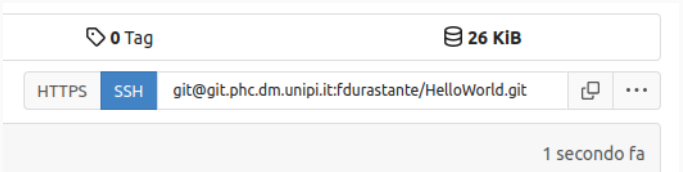
Per **prima cosa**, dobbiamo creare un **punto di sincronizzazione** per il materiale sulla nostra **macchina locale**. Questa operazione è detta operazione di `clone` ed è composta di due parti:

Clone

Per **prima cosa**, dobbiamo creare un **punto di sincronizzazione** per il materiale sulla nostra **macchina locale**. Questa operazione è detta operazione di `clone` ed è composta di due parti:

1. Per prima cosa recuperiamo l'**indirizzo presso cui il nostro *repository* risiede**.

   Se avete deciso di **non usare** Gitea, questo è:

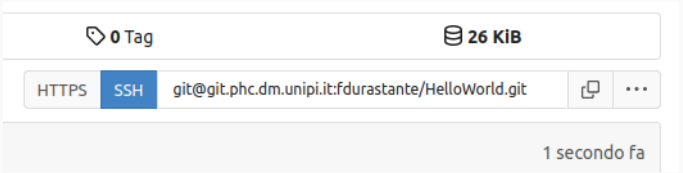



```
utente@mathsgalore<-2-3-4>.unipi.it:mygitserver/hello-world.git
```

Clone

Per **prima cosa**, dobbiamo creare un **punto di sincronizzazione** per il materiale sulla nostra **macchina locale**. Questa operazione è detta operazione di `clone` ed è composta di due parti:

1. Per prima cosa recuperiamo l'**indirizzo presso cui il nostro *repository* risiede**.



2. In una shell sulla macchina *mathsgalore* su cui abbiamo generato la chiave SSH eseguiamo il comando:
   `git clone git@git.phc.dm.unipi.it:fdurastante/HelloWorld.git`
   sostituendo all'indirizzo quello ottenuto allo **step 1**.

```
a037726@login:~/Documents$ git clone git@git.phc.dm.unipi.it:fdurastante/HelloWorld.git
Cloning into 'HelloWorld'...
The authenticity of host 'git.phc.dm.unipi.it (131.114.51.97)' can't be established.
ECDSA key fingerprint is SHA256:6tQeeoZXJrvH8nub2EbPLujeMIZfafFfdQju7gJGcaY.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'git.phc.dm.unipi.it,131.114.51.97' (ECDSA) to the list of known hosts.
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
a037726@login:~/Documents$
```

Clone

Se eseguiamo il comando `ls` osserviamo di aver creato una cartella di nome `HelloWorld`, o, più in generale, chiamata come il *repository*.

Se facciamo `cd HelloWorld` seguito da `ls`, osserviamo che al suo interno troviamo il file `README` e la `LICENSE`.

```
a037726@login:~/Documents$ cd HelloWorld/
a037726@login:~/Documents/HelloWorld$ ls
LICENSE  README.md
a037726@login:~/Documents/HelloWorld$
```

Adesso **abbiamo una copia** di tutto quello che è contenuto nel **repository** `HelloWorld` anche nella nostra **macchina locale**.

Glossario: clone

Un *clone* è sia la copia di un *repository* che risiede sul tuo computer invece che sul server di un sito web da qualche parte, sia l'atto di fare quella copia. Quando crei un *clone*, puoi modificare i file nel tuo editor preferito e **usare Git per tenere traccia delle tue modifiche** senza dover essere online. Il *repository* che hai clonato è ancora connesso alla versione remota in modo da poter inviare le modifiche locali al remoto per mantenerle sincronizzate quando sei online.

Facciamo una modifica:

Supponiamo ora di voler **modificare il nostro file README**.

Eseguiamo: `nano README.md` e scriviamo qualcosa nell'editor:



facciamo CTRL + O per salvare (confermando il nome del file con un ENTER) e poi CTRL + X per chiudere.

Ora c'è **una differenza** tra la **versione locale** e la **versione remota**!

git status

se eseguiamo il comando `git status`:



ci vengono comunicate alcune informazioni:

- ci troviamo sul **branch** `main` (torneremo a discuterne tra poco),
- tutto quello che c'è sul repository online coincide con quello che abbiamo,
- abbiamo delle **differenze locali** e possiamo decidere tra due cose:
  - aggiungere le nostre modifiche: `git add README.md`,
  - riportare il file allo stato originale: `git checkout README.md`,

add e commit

Abbiamo deciso che **le nostre modifiche sono da preservare** quindi procediamo a fare:

```
git add README.md
```

e ripetiamo di nuovo `git status`:

```
a037726@login:~/Documents/HelloWorld$ git add README.md
a037726@login:~/Documents/HelloWorld$ 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:   README.md
```

che ci dice che il file è pronto per essere inserito in un `commit`.

Dobbiamo tuttavia prima identificarci, per cui eseguiamo i comandi:

```
git config user.email "fabio.durastante@unipi.it"
git config user.name "Fabio Durastante"
```

add e commit

Siamo pronti per **eseguire il commit**:

`git commit -m "Aggiunte informazioni al README"`

che assegna alla nostra modifica un messaggio descrittivo dopo

`-m " messaggio di commit "`,

che deve descrivere **brevemente** l'argomento della **modifica** fatta.

```
a037726@login:~/Documents/HelloWorld$ git commit -m "Aggiunte informazioni al README"
[main dd4bdcc] Aggiunte informazioni al README
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Possiamo considerare di nuovo cosa succede se eseguiamo `git status`:

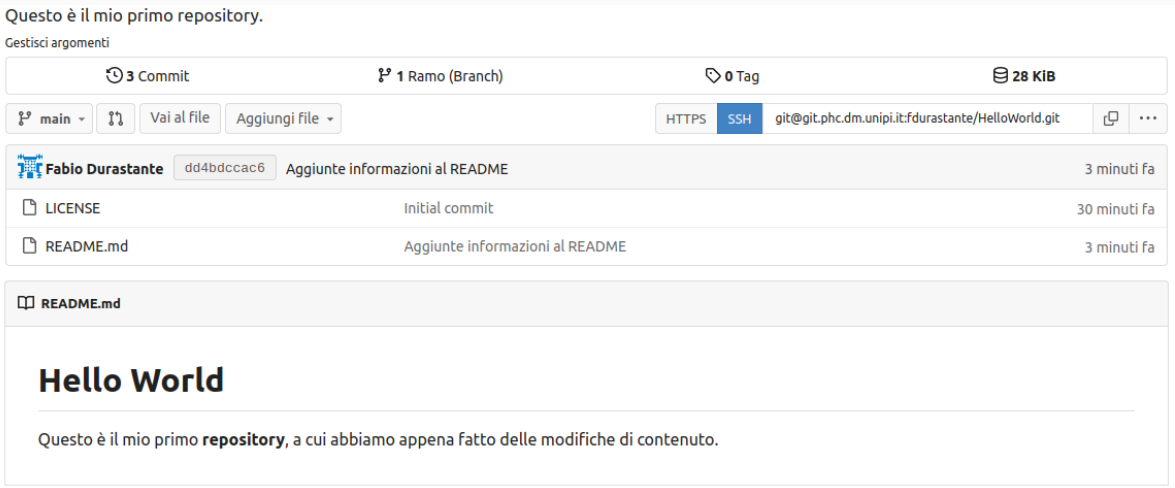
```
a037726@login:~/Documents/HelloWorld$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

git push

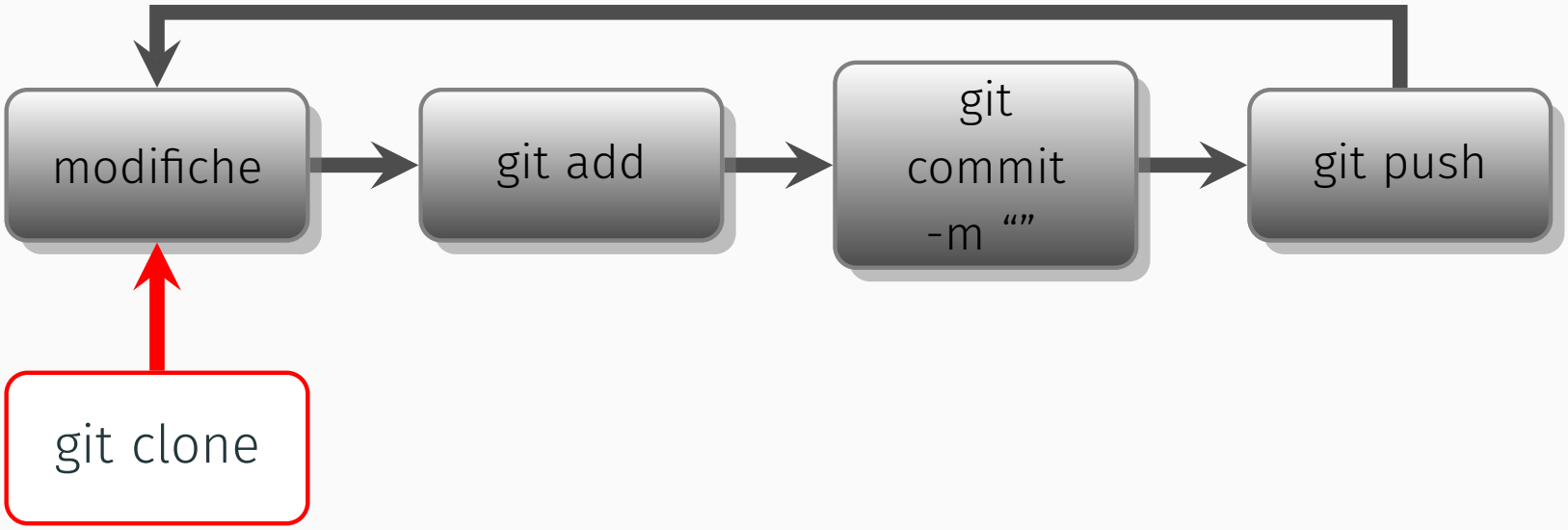
La modifica è pronta per essere “spinta” sul server remoto. Questa operazione è detta `push`:

```
a037726@login:~/Documents/HelloWorld$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 338 bytes | 338.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: . Processing 1 references
remote: Processed 1 references in total
To git.phc.dm.unipi.it:fdurastante/HelloWorld.git
   a9b2141..dd4bdcc  main -> main
```

Così facendo abbiamo ripristinato la sincronia tra locale e remoto!
Torniamo a vedere cosa è cambiato sull’interfaccia web:



Riassumiamo



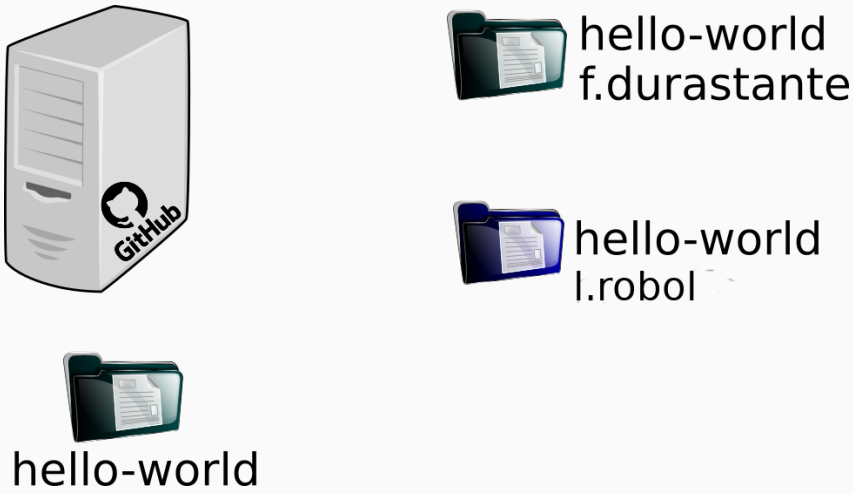
1. Creazione di un repository su GitHub,
2. Clone su una macchina locale (`git clone`),
3. Modifiche ai file locali fino alla soddisfazione (codice C, Matlab, sage, LaTeX, pagine web),
4. Preparazione di un *commit* (`git add ...`, `git commit -m " "`),
5. Sincronizzare il remoto (`git push`).

Markdown per i file README

- **Cos'è Markdown?** Markdown è un linguaggio di formattazione leggero e facile da usare per la scrittura di documenti, progettato per essere facilmente convertito in HTML.
- **Perché usare Markdown sui README?** Gitea (e GitHub) supportano il Markdown come linguaggio di formattazione per la creazione di contenuti nelle issue, nei commenti, nei README e nelle pagine wiki.
- **Sintassi di base:**
  - Testo in grassetto: `**grassetto**` o `__grassetto__`
  - Testo in corsivo: `*corsivo*` o `_corsivo_`
  - Elenco puntato: Utilizzare asterischi, trattini o numeri
  - Link: `[testo del link](URL)`
  - Immagini: `![testo alternativo](URL)`
  - Evidenziare codice: Utilizzare gli accenti gravi (backticks)
- **Estensioni di Markdown su GitHub:** GitHub supporta anche alcune estensioni di Markdown come la sintassi per le tabelle, le caselle di controllo delle liste di attività, ecc.

git pull

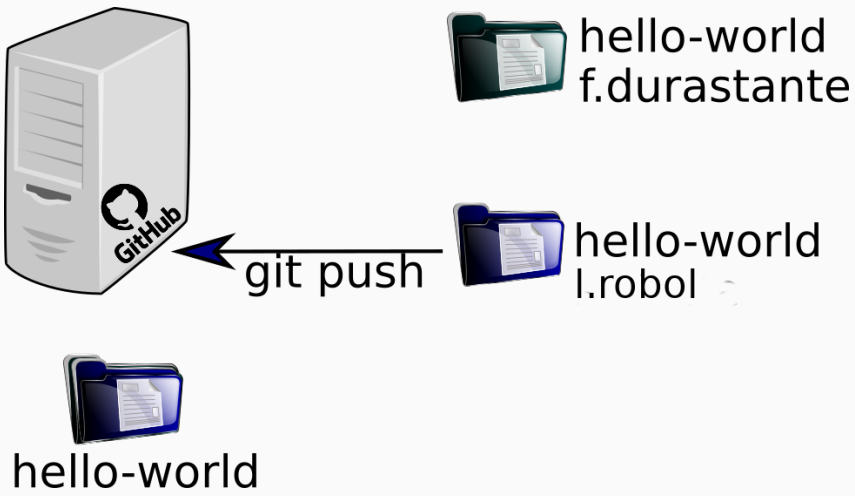
Immaginiamo ora di avere due (o più collaboratori) o macchine con cui lavoriamo sullo stesso *repository*.



L'utente `l.robol` ha fatto una sua modifica in locale, `f.durastante` e l'online non ne sanno nulla.

git pull

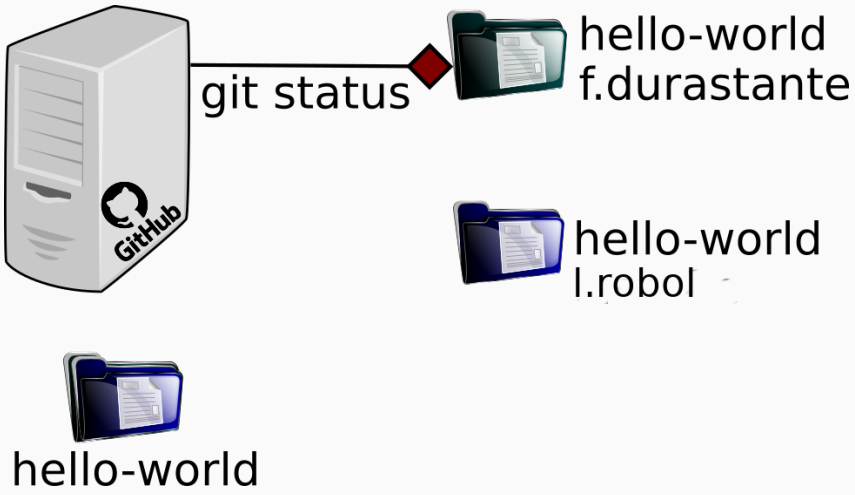
Immaginiamo ora di avere due (o più collaboratori) o macchine con cui lavoriamo sullo stesso *repository*.



`l.robol` è soddisfato dei suoi cambi, fa `git add`, `git commit` e `git push`: ora la versione sul *repository* è stata aggiornata.

git pull

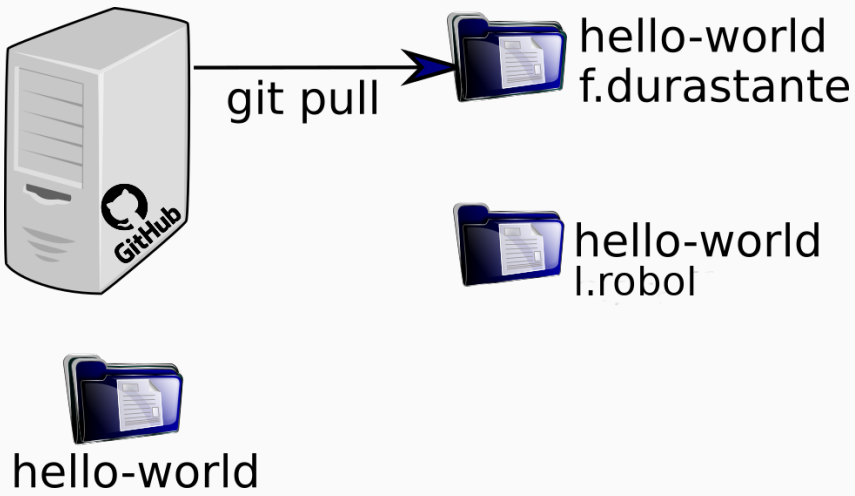
Immaginiamo ora di avere due (o più collaboratori) o macchine con cui lavoriamo sullo stesso *repository*.



Prima di mettersi a lavorare `f.durastante` si domanda se la sua versione è aggiornata, fa `git status` e scopre di essere indietro di un `commit`.

git pull

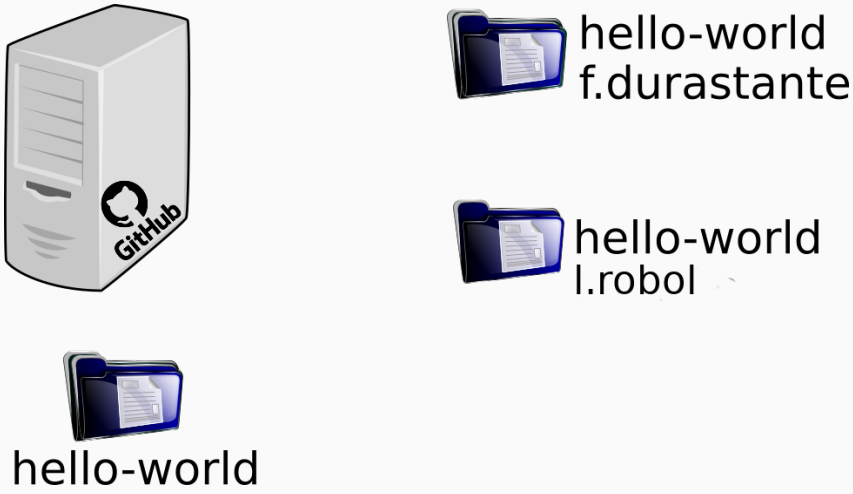
Immaginiamo ora di avere due (o più collaboratori) o macchine con cui lavoriamo sullo stesso *repository*.



Vuole mettersi in pari con le modifiche e chiama quindi `git pull` per “tirare” giù dal *repository* le modifiche.

git pull

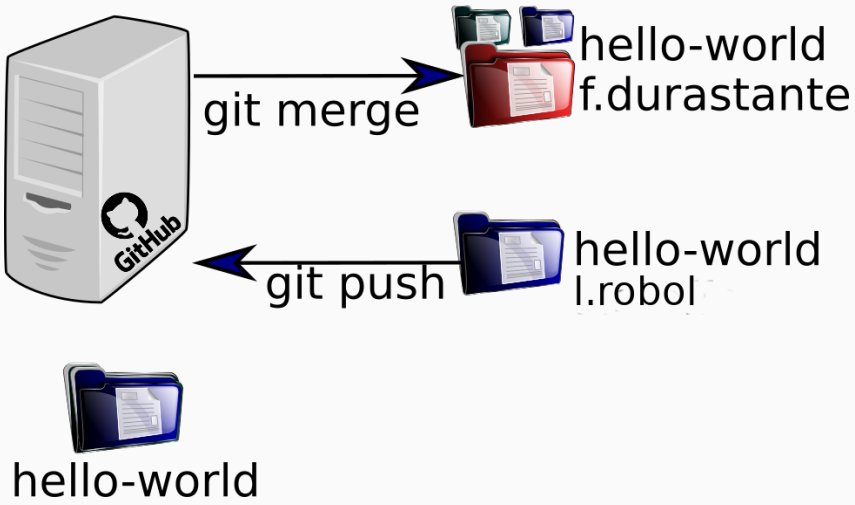
Immaginiamo ora di avere due (o più collaboratori) o macchine con cui lavoriamo sullo stesso *repository*.



Adesso tutte le versioni locali e i *repository* coincidono.

git pull

Immaginiamo ora di avere due (o più collaboratori) o macchine con cui lavoriamo sullo stesso *repository*.

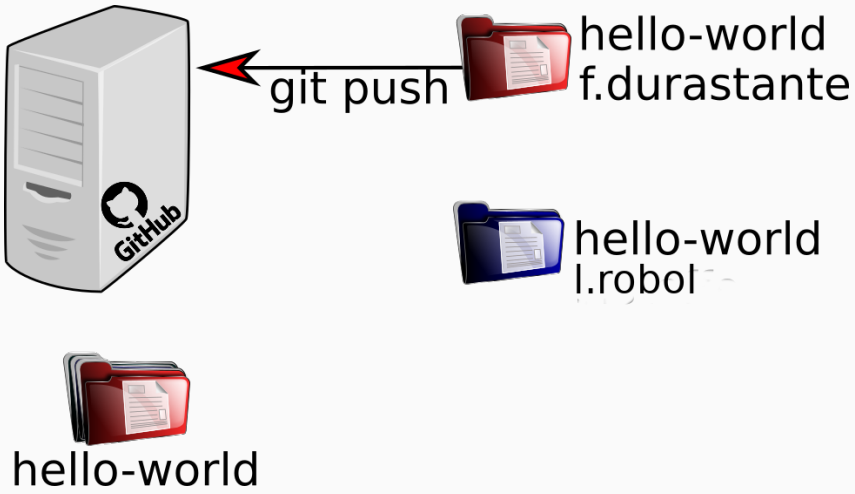


Risolvere conflitti

È facile immaginare situazioni in cui si creano dei conflitti tra le diverse versioni di diversi utenti, ad esempio `l.robol` e `f.durastante` fanno entrambi delle modifiche e tentano di farne `push`. Il primo ci riesce, il secondo dovrà fare un'operazione di `git merge` per risolvere i conflitti sulla sua copia locale prima di fare `commit` e `push`.

git pull

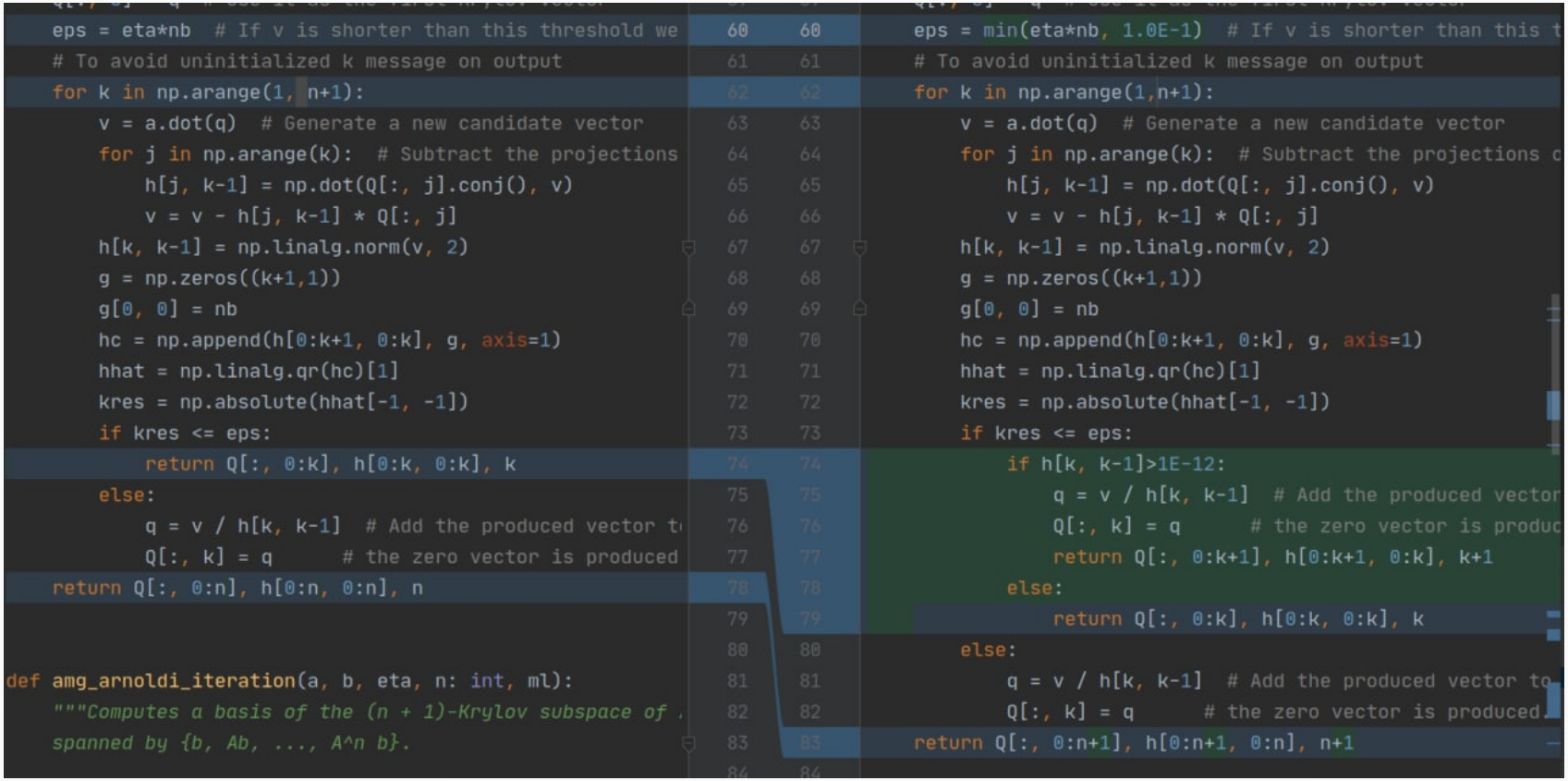
Immaginiamo ora di avere due (o più collaboratori) o macchine con cui lavoriamo sullo stesso *repository*.



Risolvere conflitti

È facile immaginare situazioni in cui si creano dei conflitti tra le diverse versioni di diversi utenti, ad esempio `l.robol` e `f.durastante` fanno entrambi delle modifiche e tentano di farne `push`. Il primo ci riesce, il secondo dovrà fare un'operazione di `git merge` per risolvere i conflitti sulla sua copia locale prima di fare `commit` e `push`.

Un esempio visivo di un’operazione di *merge*

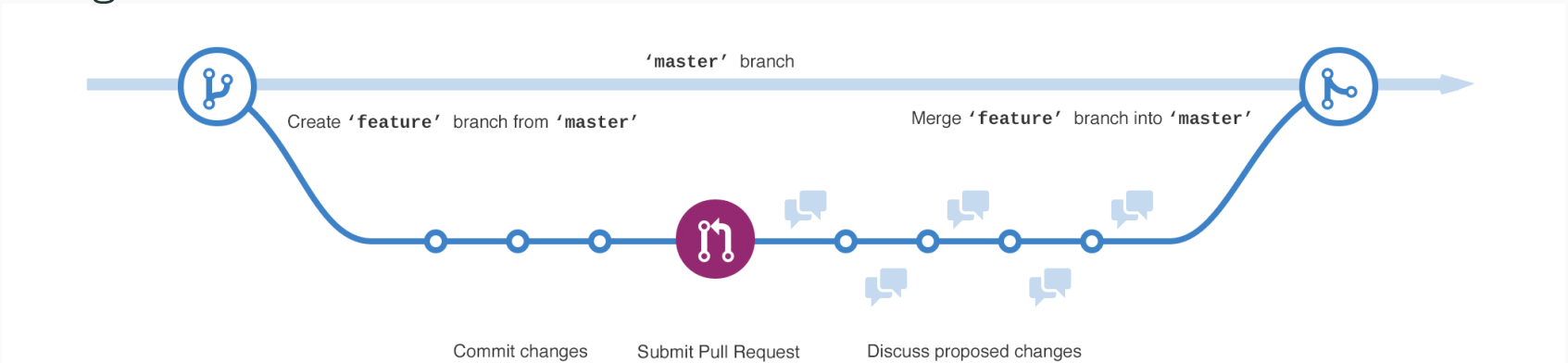


Branch

Il **branching** (ramificazione) consente di avere diverse versioni di un repository contemporaneamente.

Per impostazione predefinita, ogni nuovo *repository* su GitHub ha un **branch chiamato main** che è considerato il ramo delle versioni "definitive". L'idea dei branch è quella di usarli per sperimentare e apportare modifiche **prima** di renderle definitive in *main*.

Quando si crea un *branch* dal *main*, si sta facendo una copia, o un'**istantanea**, così com'è in quel momento. Se qualcun altro ha apportato modifiche al ramo principale mentre stavi lavorando sul tuo ramo, puoi importare quelle modifiche tramite un'operazione di merge.



Creare e muoversi tra i branch

Per **creare un nuovo** *branch* si usa il comando:

```
git checkout -b nomedelnuovobranch
```

```
a037726@login:~/Documents/HelloWorld$ git checkout -b nuovofile
Switched to a new branch 'nuovofile'
a037726@login:~/Documents/HelloWorld$ git status
On branch nuovofile
nothing to commit, working tree clean
```

per **muoversi tra i *branch*** si usa invece il comando:

```
git checkout nomedelbranch
```

```
a037726@login:~/Documents/HelloWorld$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

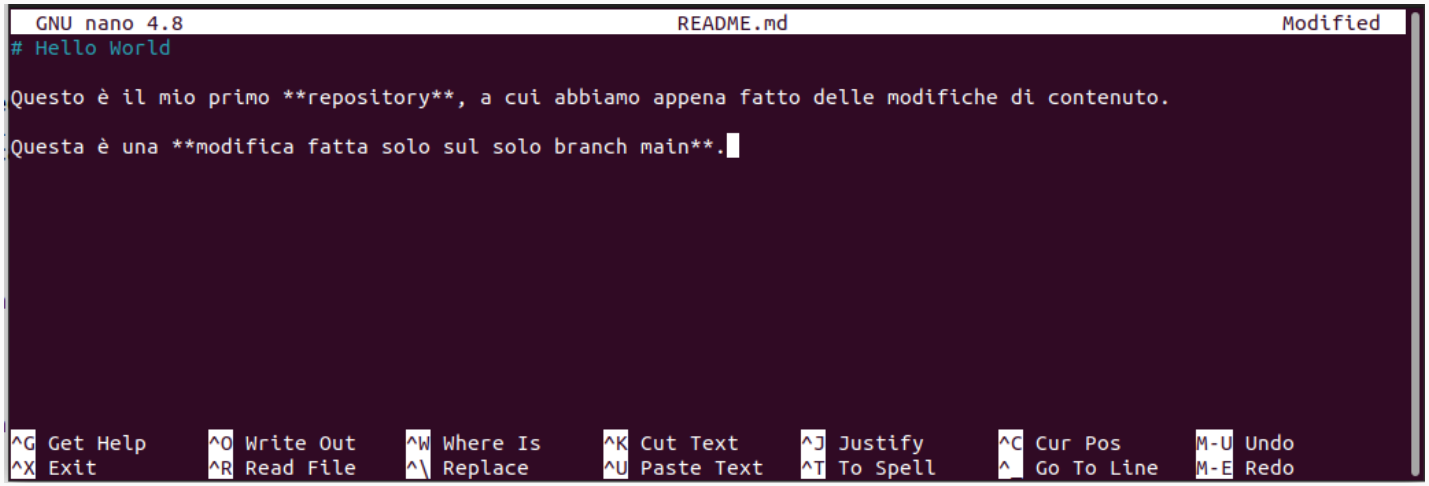
Adesso dobbiamo rendere partecipe il *repository* del nuovo branch, facendo:

```
git push --set-upstream origin nuovofile
```

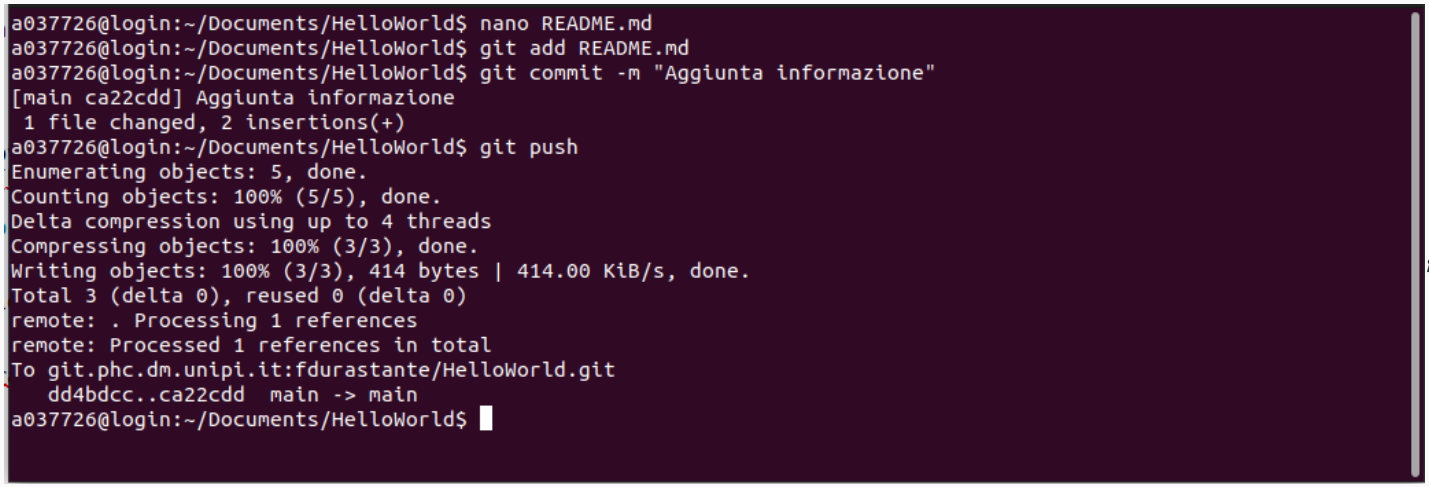
Un esercizio di branching e merging

Ora che siamo tornati in `main`, facciamo una nuova modifica al file `README.md`, ne facciamo `add`, `commit` e `push`.

1.



2.



,

Un esercizio di branching e merging

Ora che siamo tornati in `main`, facciamo una nuova modifica al file `README.md`, ne facciamo `add`, `commit` e `push`.

1. `nano README.md`
2. `git add README.md; git commit -m "Aggiunta informazione"; git push`,

Torniamo nel *branch* che avevamo appena creato con `git checkout nuovofile`, possiamo chiedere ora quali sono le differenze rispetto al branch *main*: `git diff main`:

```
a037726@login:~/Documents/HelloWorld$ git diff main
diff --git a/README.md b/README.md
index fddb58f..8a11444 100644
--- a/README.md
+++ b/README.md
@@ -1,5 +1,3 @@
 # Hello World

 Questo è il mio primo **repository**, a cui abbiamo appena fatto delle modifiche di contenuto.
-
-Questa è una **modifica fatta solo sul solo branch main.**
```

Un esercizio di branching e merging

Decidiamo che questa modifica è rilevante per quello che vogliamo fare e la importiamo nel nostro branch: `git merge main`.

```
a037726@login:~/Documents/HelloWorld$ git merge main
Updating dd4bdcc..ca22cdd
Fast-forward
 README.md | 2 ++
 1 file changed, 2 insertions(+)
a037726@login:~/Documents/HelloWorld$
```

In questo caso `git` si accorge che c'è solo un commit di differenza e l'effetto è analogo a quello di aver fatto un *git pull*.

Adesso che abbiamo **sistemato la nostra versione locale**, possiamo concludere l'operazione con un `git push`.

Aggiungere collaboratori

Per provare le funzionalità di `merge` e risoluzione dei conflitti può essere utile avere un `repository` con più di un collaboratore. I **collaboratori** possono essere **aggiunti** tramite l'**interfaccia web**.

Accedi a Gitea e Vai alle Impostazioni del Repository

1. Accedi al tuo account su Gitea.
2. Naviga al repository a cui desideri aggiungere collaboratori.
3. Fai clic su "Impostazioni" nel menu.
4. Seleziona "Collaboratori" nelle opzioni di configurazione del repository.

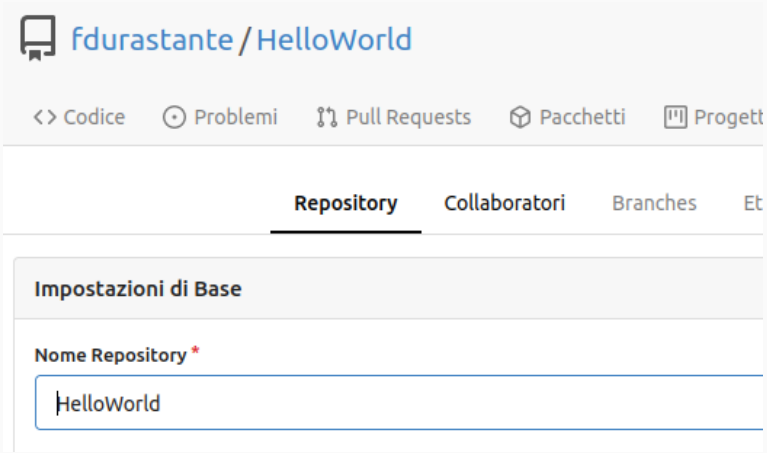


Aggiungere collaboratori

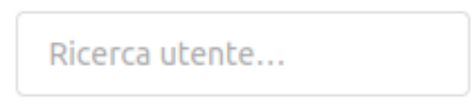
Per provare le funzionalità di `merge` e risoluzione dei conflitti può essere utile avere un `repository` con più di un collaboratore. I **collaboratori** possono essere **aggiunti** tramite l'**interfaccia web**.

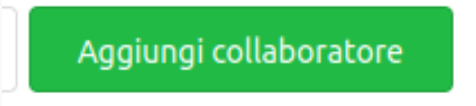
Accedi a Gitea e Vai alle Impostazioni del Repository

1. Accedi al tuo account su Gitea.
2. Naviga al repository a cui desideri aggiungere collaboratori.
3. Fai clic su "Impostazioni" nel menu.
4. Seleziona "Collaboratori" nelle opzioni di configurazione del repository.



Aggiungere collaboratori

1. Nella sezione ”Collaboratori”, cerca il campo per aggiungere i collaboratori: 

2. Digita il nome utente o l’indirizzo email del collaboratore che desideri aggiungere.
3. Premi il pulsante per confermare l’aggiunta del collaboratore: 

4. Dopo aver aggiunto il collaboratore, **verrà inviata loro una notifica**.
5. Il **collaboratore deve accettare l’invito** per diventare un collaboratore effettivo.

Esercizio:
Provate ad aggiungere un collaboratore e fate qualche modifica sullo stesso `repository`.

Come faccio a sapere quali sono le ultime modifiche?

Immaginiamo di avere un `repository` con diversi collaboratori e di voler scoprire quali modifiche sono state fatte negli ultimi `commit`.

- Il comando `git log` è utilizzato per visualizzare la cronologia dei commit in un repository Git.
- Fornisce informazioni dettagliate su chi ha committato, quando e quali modifiche sono state apportate.
- Vediamo come utilizzare questo comando per esplorare la cronologia del nostro progetto.

Come faccio a sapere quali sono le ultime modifiche?

Immaginiamo di avere un `repository` con diversi collaboratori e di voler scoprire quali modifiche sono state fatte negli ultimi `commit`.

- Il comando `git log` è utilizzato per visualizzare la cronologia dei commit in un repository Git.
- Fornisce informazioni dettagliate su chi ha committato, quando e quali modifiche sono state apportate.
- Vediamo come utilizzare questo comando per esplorare la cronologia del nostro progetto.

La **sintassi di base** del comando `git log` è:

```
git log
```

Questo comando mostra l'elenco dei commit nel repository, partendo dal commit più recente.

Come faccio a sapere quali sono le ultime modifiche?

```
commit ca22cdd8bd8a4f82d1cb47a5879d2def7289f359 (HEAD -> nuovofile, origin/nuovofile, origin/main, origin/HEAD, m
ain)
Author: Fabio Durastante <fabio.durastante@unipi.it>
Date:   Tue Mar 12 16:52:01 2024 +0000

    Aggiunta informazione

commit dd4bdccac63e7f1ffa7f000c1ea6e98871a2df32
Author: Fabio Durastante <fabio.durastante@unipi.it>
Date:   Sun Mar 10 21:56:50 2024 +0000

    Aggiunte informazioni al README

commit a9b2141710b92f991a6acadde3d0f941adfc8d5e
Author: Fabio Durastante <a037726@login.polo2.ad.unipi.it>
Date:   Sun Mar 10 21:51:24 2024 +0000

    README Modificato

commit 0595f049639c30dcb6c68f594a1065cda08d967e
:
```

Come faccio a sapere quali sono le ultime modifiche?

```
commit ca22cdd8bd8a4f82d1cb47a5879d2def7289f359 (HEAD -> nuovofile, origin/nuovofile, origin/main, origin/HEAD, m
ain)
Author: Fabio Durastante <fabio.durastante@unipi.it>
Date:   Tue Mar 12 16:52:01 2024 +0000

    Aggiunta informazione

commit dd4bdccac63e7f1ffa7f000c1ea6e98871a2df32
Author: Fabio Durastante <fabio.durastante@unipi.it>
Date:   Sun Mar 10 21:56:50 2024 +0000

    Aggiunte informazioni al README

commit a9b2141710b92f991a6acadde3d0f941adfc8d5e
Author: Fabio Durastante <a037726@login.polo2.ad.unipi.it>
Date:   Sun Mar 10 21:51:24 2024 +0000

    README Modificato

commit 0595f049639c30dcb6c68f594a1065cda08d967e
:
```

Ci sono molte opzioni che possiamo utilizzare con `git log`:

- `-n`: Limita il numero di commit mostrati.
- `--author`: Filtra i commit per autore.
- `--since` e `--until`: Filtra i commit per intervallo di tempo.
- `--grep`: Filtra i commit per messaggio di commit.
- `--oneline`: Mostra ciascun commit in una sola riga.

Esempi con `git log`

Vediamo alcuni esempi di come utilizzare `git log` con alcune opzioni comuni:

- `git log -n 5`: Mostra gli ultimi 5 commit.
- `git log --author="NomeAutore"`: Mostra i commit effettuati da un autore specifico.
- `git log --since="2023-01-01"`: Mostra i commit successivi al 1 gennaio 2023.
- `git log --grep="Fix"`: Mostra i commit con il messaggio contenente ”Fix”.

Esempi con `git log`

Vediamo alcuni esempi di come utilizzare `git log` con alcune opzioni comuni:

- `git log -n 5`: Mostra gli ultimi 5 commit.
- `git log --author="NomeAutore"`: Mostra i commit effettuati da un autore specifico.
- `git log --since="2023-01-01"`: Mostra i commit successivi al 1 gennaio 2023.
- `git log --grep="Fix"`: Mostra i commit con il messaggio contenente ”Fix”.

- Il comando `git log` è un potente strumento per esplorare la cronologia dei `commit` in un repository Git.
- Con le sue varie opzioni, è possibile filtrare e visualizzare esattamente le informazioni desiderate.
- Utilizzate `git log` regolarmente per comprendere meglio lo sviluppo del vostro progetto.

Documentazione e Informazioni

Queste sono le **istruzioni basilari** per utilizzare Git e Gitea per gestire un piccolo progetto in una o poche persone. È possibile utilizzarlo in modo **più sofisticato**, ma questo sfugge agli interessi limitati che abbiamo qui.

Alcune referenze utili sono:

training.github.com/downloads/it/github-git-cheat-sheet/

e

git-scm.com/docs

con in particolare la pagina:
https://ndpsoftware.com/git-cheatsheet.html.

Introduzione a Markdown

Che cos'è Markdown?

- **Markdown** è un linguaggio di markup leggero per formattare testo utilizzando una sintassi molto semplice.
- I documenti rimangono facilmente leggibili anche nel loro formato sorgente.
- È molto utilizzato per:
  - file `README.md`;
  - documentazione di progetti software;
  - pagine Web e blog;
  - repository Git.
- I file Markdown hanno normalmente estensione `.md`.

Idea fondamentale
Markdown permette di descrivere la struttura di un documento utilizzando pochi simboli inseriti direttamente nel testo.

Titoli

I titoli vengono creati utilizzando il carattere `#`.

```
# Titolo principale

## Titolo di secondo livello

### Titolo di terzo livello

#### Titolo di quarto livello
```

Il numero di caratteri `#` indica il **livello del titolo**.

Esempio
Un file `README.md` inizia spesso con un titolo di primo livello contenente il nome del progetto.

Grassetto e corsivo

Markdown permette di evidenziare facilmente alcune parti del testo.

```
Questo testo è **in grassetto**.

Questo testo è *in corsivo*.

Questo testo è ***grassetto e corsivo***.
```

Il risultato sarà:

- Questo testo è **in grassetto**.
- Questo testo è *in corsivo*.
- Questo testo è ***grassetto e corsivo***.

Elenchi puntati

Gli elenchi puntati possono essere creati utilizzando un trattino davanti a ogni elemento.

```
- HTML
- CSS
- JavaScript
- Markdown
```

È possibile creare anche elenchi annidati:

```
- Linguaggi Web
- HTML
- CSS
- JavaScript

- Strumenti
- Git
- Codeberg
```

Elenchi numerati

Per creare un elenco numerato si utilizzano numeri seguiti da un punto.

```
1. Creare un repository
2. Aggiungere i file
3. Creare un commit
4. Eseguire il push
```

Markdown può gestire automaticamente la numerazione.

```
1. Primo elemento
1. Secondo elemento
1. Terzo elemento
```

Risultato

Il documento visualizzato mostrerà comunque gli elementi numerati correttamente.

Collegamenti ipertestuali

Un collegamento viene scritto indicando il **testo visualizzato** e l'**indirizzo**.

```
[Codeberg](https://codeberg.org/)
```

```
[Documentazione Codeberg](https://docs.codeberg.org/)
```

La struttura generale è:

```
[testo visualizzato](URL)
```

Esempio
Il testo `Codeberg` diventerà cliccabile e porterà all'indirizzo specificato tra parentesi tonde.

Inserire immagini

La sintassi per inserire un'immagine è simile a quella utilizzata per i link.

```
![Logo Codeberg](codeberg-logo.png)
```

La struttura generale è:

```
![testo alternativo](percorso-immagine)
```

- Il **testo alternativo** descrive l'immagine.
- Il percorso può indicare un file del repository.
- Può anche essere utilizzato un URL.

Codice all’interno del testo

Per rappresentare un comando o una breve porzione di codice si utilizzano i **backtick**.

```
Usa il comando `git status`
per controllare lo stato del repository.
```

Il risultato sarà simile a:

> Usa il comando `git status` per controllare lo stato del repository.

Quando usarlo
Il codice inline è utile per comandi, nomi di file, variabili e brevi frammenti di codice.

Blocchi di codice

Per inserire più righe di codice si utilizzano **tre backtick** prima e dopo il codice.

````
```bash
git add .
git commit -m "Primo commit"
git push
```
````

Dopo i primi tre backtick è possibile indicare il linguaggio.

````
```python
print("Hello!")
```
````

Questo permette alle piattaforme compatibili di applicare la **colorazione della sintassi**.

Citazioni

Le citazioni vengono indicate utilizzando il simbolo > all'inizio della riga.

```
> Markdown permette di scrivere
> documenti in modo semplice e leggibile.
```

È possibile anche creare citazioni su più livelli:

```
> Citazione principale
>
>> Citazione annidata
```

Linee di separazione

Una linea orizzontale può essere creata utilizzando tre trattini.

```
---
```

Ad esempio:

```
## Introduzione

Testo introduttivo.

---

## Installazione

Istruzioni per installare il programma.
```

Le linee orizzontali permettono di separare visivamente sezioni differenti del documento.

Checklist

Markdown può essere utilizzato anche per creare **liste di attività**.

```
- [x] Creare il repository
- [x] Aggiungere il README
- [ ] Scrivere la documentazione
- [ ] Pubblicare la prima release
```

- `[x]` indica un'attività completata.
- `[ ]` indica un'attività da completare.

Su Codeberg
Le checklist sono particolarmente utili nei file README, nelle Issues e nelle Pull Request.

Tabelle

Markdown permette anche di creare semplici tabelle.

```
| Linguaggio | Utilizzo |
|------------|----------|
| HTML       | Struttura |
| CSS        | Stile |
| JavaScript | Interattivita |
```

La seconda riga separa l'intestazione dal contenuto.

Nota
Le tabelle fanno parte delle estensioni Markdown supportate da molte piattaforme per repository Git.

Link e immagini nel repository

Nei file Markdown di un repository è spesso conveniente utilizzare **percorsi relativi**.

```
[Documentazione](docs/installazione.md)

![Screenshot](images/screenshot.png)
```

In questo modo il collegamento continua a funzionare anche quando il repository viene clonato su un altro computer.

Esempio di struttura

```
progetto/
|-- README.md
|-- docs/
|   `-- installazione.md
`-- images/
`-- screenshot.png
```

Un esempio di README.md

````
# Il mio progetto

Un semplice progetto pubblicato su Codeberg.

## Installazione

Clonare il repository:

```bash
git clone https://codeberg.org/utente/progetto
```

## Funzionalita

- Semplice
- Open source
- Facile da modificare

## Documentazione

Consulta la [documentazione](docs/README.md).

## Licenza

Il progetto e distribuito con licenza libera.
````

Un README più completo

Un buon README può contenere diverse sezioni:

```
# Nome del progetto

Breve descrizione del progetto.

## Installazione

Istruzioni per installare il software.

## Utilizzo

Esempi di utilizzo.

## Contribuire

Indicazioni per chi vuole contribuire.

## Autori

Informazioni sugli autori.

## Licenza

Informazioni sulla licenza del progetto.
```

Markdown: riepilogo

Titoli e testo

```
# Titolo

**grassetto**

*corsivo*

`codice`
```

Elenchi

```
- elemento
- elemento

1. primo
2. secondo
```

Link e immagini

```
[testo](URL)

![testo](immagine.png)
```

Citazioni e attività

```
> citazione

- [x] completato
- [ ] da fare
```

Perché usare Markdown?

- È **semplice** da imparare.
- È leggibile anche senza essere convertito.
- È ideale per essere gestito con **Git**.
- Permette di documentare facilmente un progetto.
- È supportato direttamente da Codeberg.

Nel nostro repository
Il file `README.md` sarà la pagina principale con cui presenteremo e documenteremo il progetto.

Codeberg e Pagine Web

Codeberg: codeberg.org

- **Alternativa etica e open source** a piattaforme centralizzate come GitHub e GitLab.
- Basato su **Forgejo**, un fork comunitario e indipendente di Gitea.
- Gestito da **Codeberg e.V.**, associazione senza scopo di lucro con sede in Germania.
- **Privacy-oriented**: nessun tracciamento pubblicitario e attenzione alla protezione dei dati.
- Finanziato principalmente tramite **donazioni della comunità**.



Creare un repository su Codeberg

Obiettivo
Creare un repository su Codeberg e collegarlo a una copia locale gestita con Git.

1. Creare il repository dall'interfaccia Web.
2. Configurare nome, visibilità e file iniziali.
3. Collegare il repository al proprio computer.
4. Creare il primo commit.
5. Pubblicare le modifiche con `git push`.

Importante
Su Codeberg la funzione *Push To Create* è disabilitata: il repository deve essere creato prima dall'interfaccia Web.

Passo 1 – Creare un nuovo repository

1. Accedi al tuo account su `codeberg.org`.
2. Apri il menu **Create** (+).
3. Seleziona **New Repository**.
4. Si apre il modulo per la creazione del repository.

Repository
Un repository è lo spazio che contiene il progetto e la sua cronologia Git. Codeberg può inoltre associargli Issues, Pull Request, Wiki e Releases.

Passo 2 – Configurare il repository

I campi principali del modulo sono:

Owner Utente o organizzazione proprietaria del repository.

Repository name Nome del progetto, ad esempio `hello-codeberg`.

Visibility Repository **pubblico** oppure **privato**.

Description Breve descrizione del progetto.

Default branch Nome del branch principale. In genere è possibile mantenere il valore predefinito.

Ad esempio, il repository sarà disponibile all'indirizzo:

`https://codeberg.org/USERNAME/hello-codeberg`

Passo 3 – File iniziali

Codeberg può inizializzare automaticamente il repository con alcuni file utili:

- **`README.md`**: presenta e documenta il progetto;
- **`LICENSE`**: specifica la licenza del progetto;
- **`.gitignore`**: indica a Git quali file non devono essere tracciati.

Per aggiungerli automaticamente è necessario selezionare:

Initialize repository

Attenzione
Se sul computer hai già un progetto Git che vuoi caricare su Codeberg, **non inizializzare il repository remoto** con README, LICENSE o **`.gitignore`**.

Passo 4 – Creare il repository

Dopo aver configurato le opzioni:

1. Controlla il nome del repository.
2. Controlla la visibilità.
3. Scegli se inizializzarlo con i file standard.
4. Premi il pulsante **Create Repository**.

Una volta creato, la pagina del repository permette di accedere a:

- codice e cronologia Git;
- Issues e Pull Request;
- Releases e Wiki;
- impostazioni del repository;
- URL per clonare il progetto tramite HTTPS o SSH.

Passo 5A – Iniziare un nuovo progetto

Se il progetto è nuovo, il modo più semplice è **clonare il repository** appena creato.

```
git clone https://codeberg.org/USERNAME/NOME-REPO
cd NOME-REPO
```

Cosa fa `git clone`?

`git clone` crea una copia locale del repository e configura automaticamente Codeberg come repository remoto con il nome `origin`.

A questo punto possiamo creare o modificare i file del progetto.

Passo 5B – Pubblicare un progetto esistente

Se il progetto esiste già sul computer, inizializziamo Git localmente:

```
cd mio-progetto

git init
git checkout -b main

git add .
git commit -m "First commit"
```

Poi colleghiamo il repository locale a Codeberg:

```
git remote add origin \
https://codeberg.org/USERNAME/NOME-REPO

git push -u origin main
```

Ricorda
In questo caso il repository creato su Codeberg deve essere **vuoto**, per evitare storie Git differenti.

Passo 6 – Modificare, fare commit e push

Dopo aver collegato il repository, il normale ciclo di lavoro è:

1. Modificare o creare un file.
2. Aggiungerlo alla staging area.
3. Creare un commit.
4. Inviare il commit a Codeberg.

```
echo "Hello Codeberg!" >> README.md
```

```
git status
git add README.md
git commit -m "Update README"
git push
```

Commit

Un commit rappresenta uno “snapshot” delle modifiche al progetto e diventa parte della sua cronologia Git.

Repository creato!

Il progetto è ora versionato e pubblicato su Codeberg.

Il flusso di lavoro quotidiano diventa:

$$\texttt{modifica} \longrightarrow \texttt{git add} \longrightarrow \texttt{git commit} \longrightarrow \texttt{git push}$$

Prossimi passi
Da qui possiamo imparare a usare **branch**, **Issues**, **Pull Request** e la collaborazione con altri utenti.

Creare una pagina web con Codeberg Pages

1. **Prepara il repository**

   Crea un repository pubblico chiamato `pages` oppure crea un branch dedicato `pages`:

   ```
   git switch --orphan pages
   git rm --cached -r .
   ```

2. **Configura il deployment**

   In Codeberg vai in

   Settings → *Webhooks* → *Add Webhook* → **Forgejo**

   Imposta:

   - **Target URL:** `https://USERNAME.codeberg.page/REPOSITORY`
   - **Branch filter:** `pages`

3. **Pubblica**

   Esegui il push dei contenuti statici (`HTML`, `CSS`, immagini, ecc.) sul branch `pages`.