08
DVCS, parte II
Input-Output e Reflection

Danilo Pianini
Giovanni Ciatto, Angelo Croatti, Mirko Viroli

Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

29 ottobre 2017

Outline

1. Decentralized Version Control Systems
   - Risolvere i merge conflict
   - Lavorare in remoto: clone, fetch, pull, push
   - Hosting e Bitbucket
   - Plugin Eclipse (eGit)
   - Features avanzate e prossimo episodio

2. Testing con JUnit

Outline

Outline

Conflicted files

In generale

Abbiamo visto nell'ultima lezione come sia possibile riunire due flussi di lavoro diversi. Abbiamo anche visto che, nel caso in cui due linee di sviluppo abbiano modificato concorrentemente un file nello stesso punto, il merge non è banale da effettuare ma occorre risolvere manualmente un *merge conflict*.

Esempio di conflict file

```
public final class HelloWorld {

    private static final String AUTHOR = "Danilo Pianini";

    public static void main(final String[] args) {
<<<<<<< HEAD
        System.out.println("This program is running in a PC with " + procNumber() + " logic processors!");
    }

    public static int procNumber() {
        return Runtime.getRuntime().availableProcessors();
=======
        System.out.println("This program has been realised by " + AUTHOR);
>>>>>>> feature
    }

}
```

Risoluzione del conflitto

- Modifica dei file che confliggono, fino a portarli allo stato desiderato.
- Aggiunta dei file alla staging area (con `git add`)
- Salvataggio delle modifiche tramite `git commit`
  - Il messaggio di commit di default nel caso di merge viene auto-generato da Git e può essere mantenuto

Errore frequente: bad tracking

Cosa succede, se per errore abbiamo messo in tracking rigenerabili che non avremmo dovuto mettere?

Un altro buon motivo per stare attenti a cosa si mette in tracking

- Se abbiamo dei file binari in tracking (ad esempio dei class files che vengono ricompilati ogni volta), ad **ogni** merge avremo dei conflitti.
- Ci sarà un conflitto per ogni risorsa modificata. Potenzialmente, centinaia ad ogni merge.
- I conflitti su binari sono difficili da risolvere: il file può essere solo ispezionato tramite editor binari, e buona fortuna a capirne il contenuto.
- Di norma si risolve cancellando tutti i file, rigenerandoli, marcando tutti i conflitti come risolti e facendo un commit.
- **ESPLODE** la dimensione del repository
- State attenti a cosa mettete in tracking!

Outline

Decentralizzazione

Decentralizzazione totale...

Nei DVCS, non esiste un punto centrale che fa repository "ufficiale": tutti i repository ricevono l'intera storia ed hanno pari importanza.
Il design dei DVCS è ideale per un mondo P2P

... nella realtà della struttura di Internet

Per via della struttura attuale di Internet, della presenza di NAT e simili strutture di rete, il modello client/server è spesso difficile da aggirare. È spesso anche sconveniente aggirarlo: tutto sommato, il cloud spesso è utile...
Esistono servizi di hosting che consentono di avere in un punto sempre accessibile dei repository in rete: chi ha bisogno di collaborare non deve preoccuparsi di problematiche di rete diverse da "andare sul web".

Clone

In generale

Occorre un meccanismo che consenta di fare copie di repository esistenti, al fine di cominciare a lavorare su qualcosa di già esistente, o di effettuare copie di sicurezza del proprio lavoro.

In Git

- `git clone URI localfolder`

Scarica l'intera storia del repository conservato in `URI` all'interno di `localfolder`, che diventa un repository Git. Esempi di `URI`:

- `/home/user/repository/` — URI locale (*nix)
- `C:\Users\Username\repository\` — URI locale (Windows)
- `ssh://username@server/repository/` — URI Secure Shell (raccomandata per chi lavora in remoto con server e/o client Unix)
- `https://user@server/repository/` — URI HTTPS, multipiattaforma

Remote repositories I

In generale

Vogliamo conservare un riferimento agli URI ai quale si trovano i repository dai quali vogliamo leggere o sui quali vogliamo scrivere.
In caso di operazioni remote, infatti, è comodo avere un nome simbolico invece di specificare sempre una URI.

Remote repositories II

In Git

Il sottocomando `git remote` consente di gestire repository remoti. Ogni repository remoto ha un nome ed una URL.

- `git remote -v`
  - Elenca i repository remoti configurati
- `git remote add name url`
  - Aggiunge un nuovo repository remoto di nome `name` che punta ad `uri`
- `git remote rm name`
  - Rimuove il repository remoto `name`

Remote repositories III

Branch upstream

È possibile configurare, per ciascun branch, un uri di un repository remoto dove si andrà a leggere e scrivere a meno di diversa specifica

- Si usa l'opzione `-u` (`--set-upstream-to`)
  - `git branch -u remoteName/remoteBranch`
  - D'ora in poi, tutte le operazioni accesso remoto lanciate dal branch corrente verranno mappate sul branch `remoteBranch` del repository remoto `remoteName`

Nel caso in cui il repository sia stato clonato e non inizializzato (ossia, il primo comando dato è stato `git clone` e non `git init`), un riferimento al repository di origine viene automaticamente inserito fra i remote e configurato come upstream per tutti i branch col nome di `origin`.

Remote repositories IV

Visualizzazione dei branch remoti

Al momento del clone, Git scaricherà solo il branch "primario" (normalmente, `master`). È possibile comunque visualizzare tutti i branch disponibili in remoto:

- `git branch -a`
  - Elenca tutti i branch, inclusi quelli remoti.

Remote repositories V

Importazione di branch remoti

Qualora si voglia importare in locale un branch remoto per lavorarci, occorre creare un "tracking branch" locale:

- `git checkout -b localBranchName remoteName/remoteBranchName`
  - Crea un nuovo branch di nome `localBranchName`, che avrà i contenuti di `remoteName/remoteBranchName`
  - Sovente si dà al branch locale lo stesso nome del branch remoto
    - `git checkout -b develop origin/develop`

Fetch e pull I

In generale

Vogliamo una operazione che ci consenta di acquisire nuovi commit da una fonte, in maniera tale da acquisire il lavoro fatto da altri. Tale operazione si chiama fetch, e *teoricamente* è l'unica operazione necessaria a rendere completamente distribuito un DVCS.

Fetch e pull II

In Git

Il sottocomando `fetch` scarica i commit da un repository remoto

- `git fetch URI branchName`
  - Scarica da `URI` tutti i commit del branch `branchName`
- `git fetch remoteName branchName`
  - Scarica dal remote `remoteName` tutti i commit del branch `branchName`
- `git fetch`
  - Scarica dal remote di default (se presente) tutti i commit

Dopo la `fetch`, è necessario utilizzare `git merge` per unire i commit scaricati al branch corrente.

Fetch e pull III

Pull

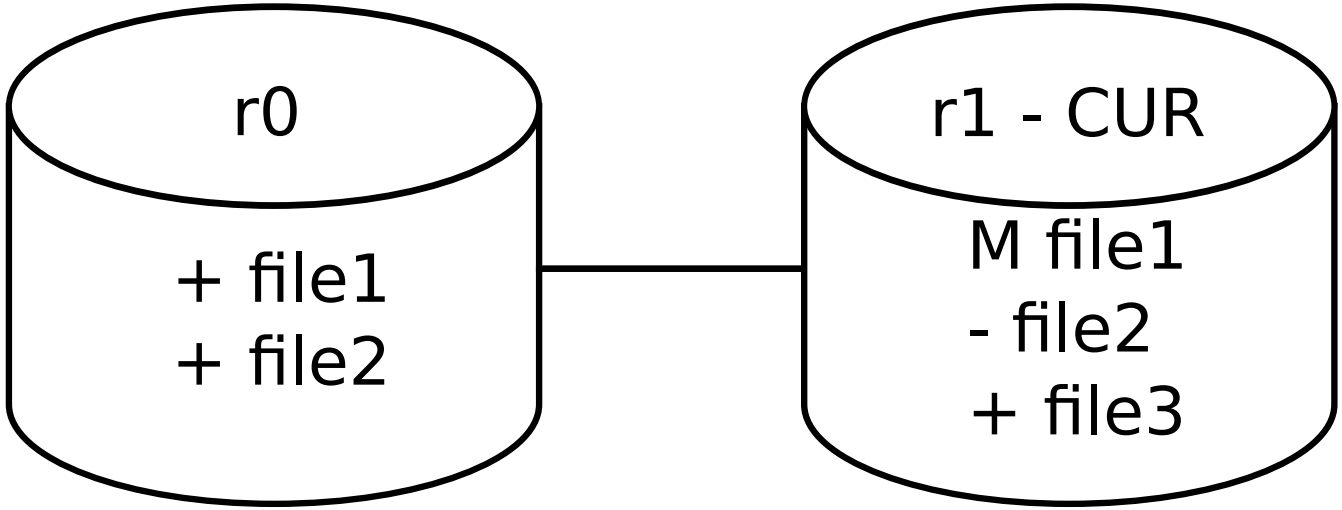
Il sottocomando `pull` (stessa sintassi di `fetch`) esegue una `fetch` seguita da un `merge`, ed è quindi conveniente da usare nel caso in cui si vogliano eseguire entrambe le operazioni.

- `git pull URI branchName`
  - Scarica da `URI` tutti i commit del branch `branchName` e prova a mergerli nel branch corrente
- `git pull remoteName branchName`
  - Scarica dal remote `remoteName` tutti i commit del branch `branchName` e prova a mergerli nel branch corrente
- `git pull`
  - Scarica dal remote di default (se presente) tutti i commit e prova a mergerli nel branch corrente

Esempio con `clone` e `pull`

Situazione iniziale

REMOTE REPO

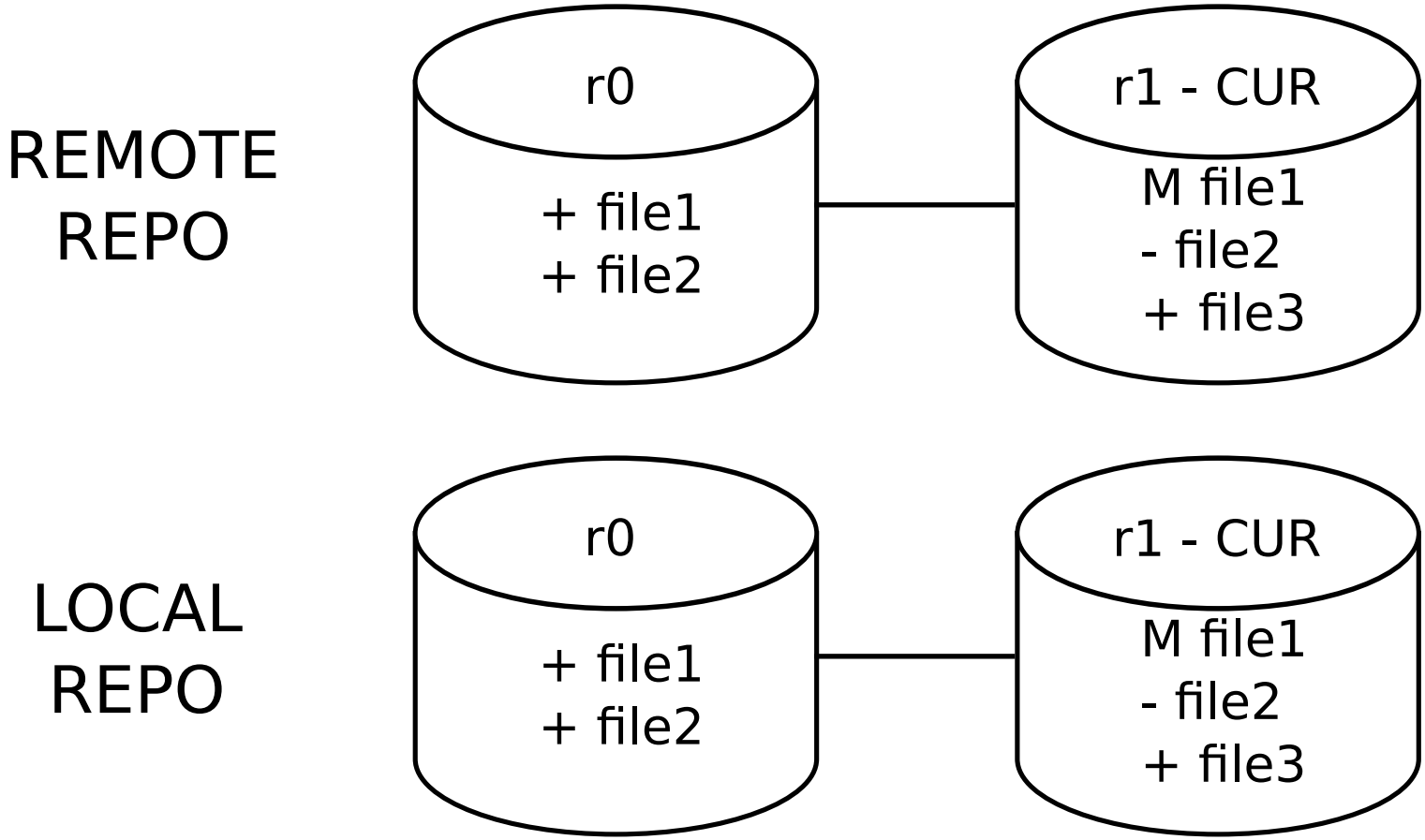


LOCAL REPO

Esempio con clone e pull

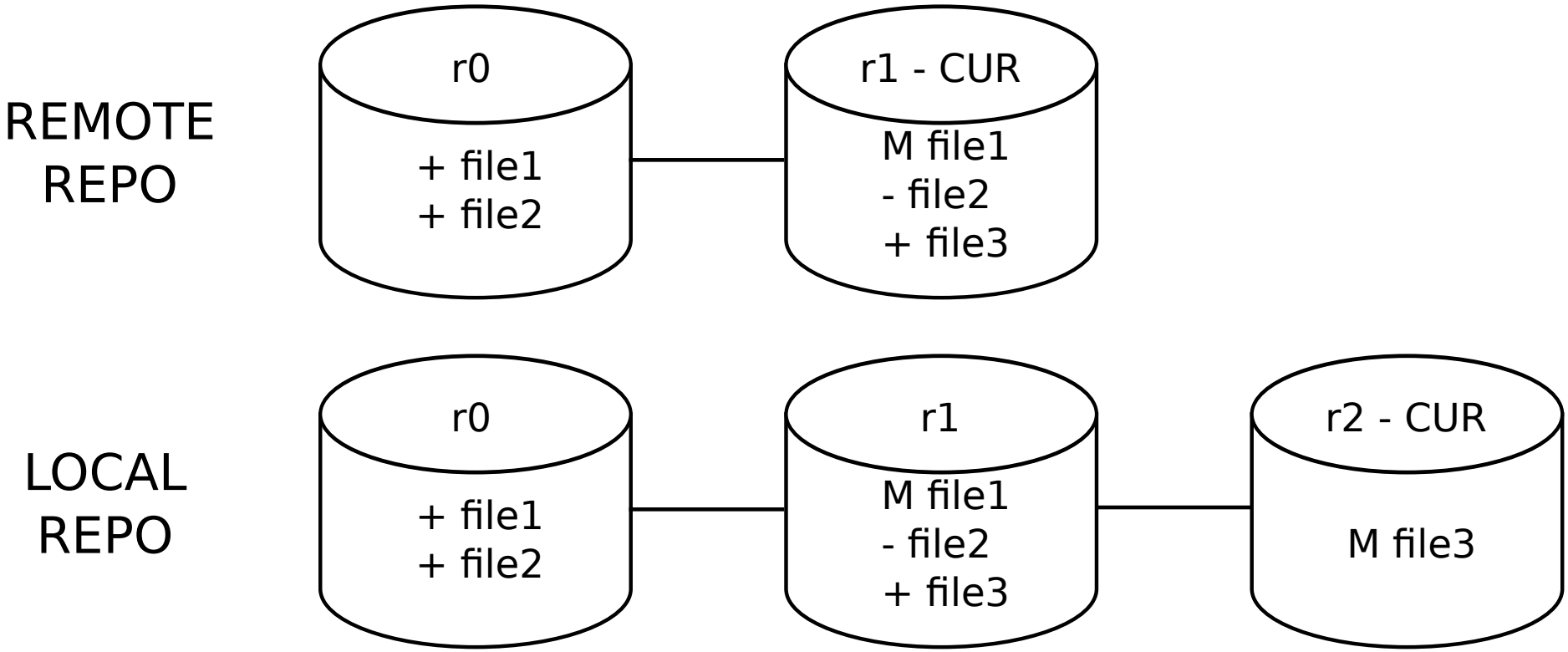
Local esegue:

```
git clone indirizzo_di_remote_repo percorso_di_local_repo
```



Esempio con `clone` e `pull`

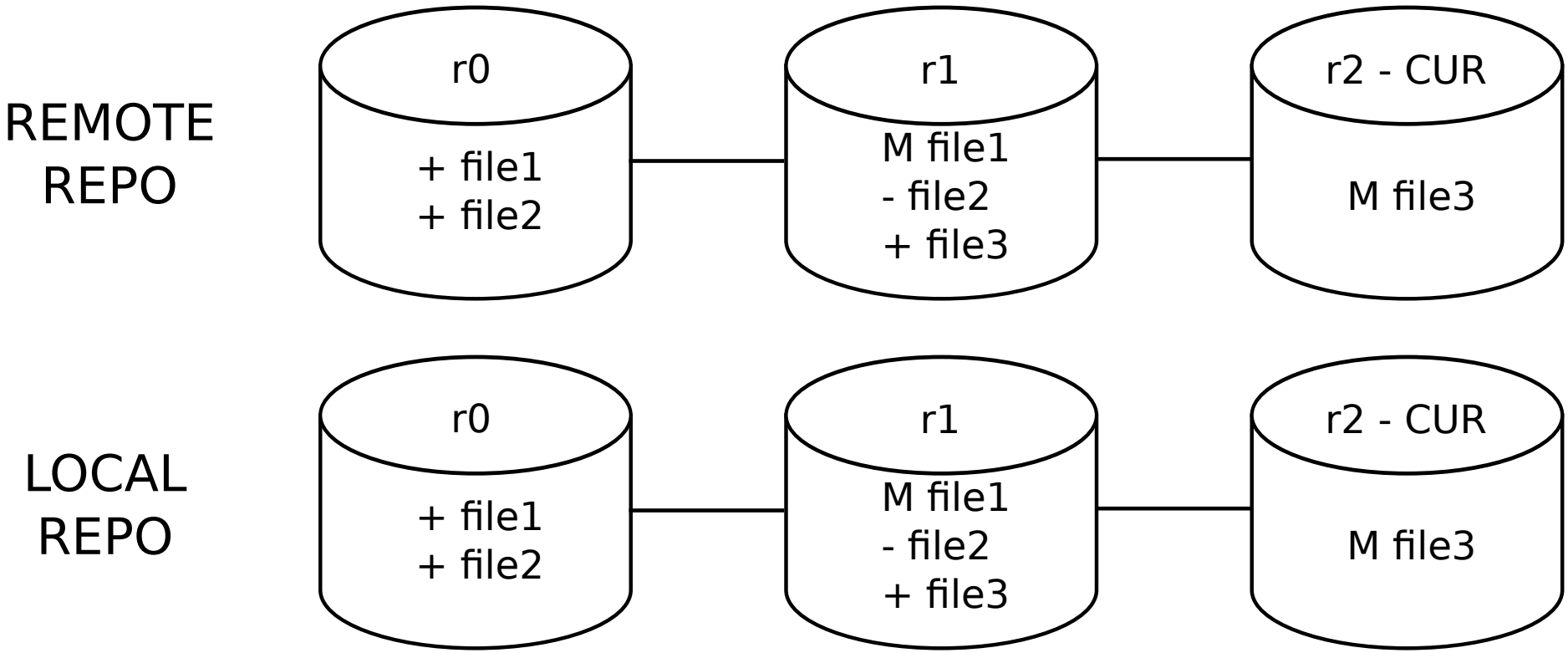
Local esegue:
modifica di `file3`
`commit`



Esempio con clone e pull

Remote esegue:

`git pull indirizzo_di_local_repo`



In generale

Nella realtà dei fatti le operazione di `fetch` e `pull` non sono sufficienti. L'operazione duale della `pull` si chiama `push`, ed è più delicata: chi la esegue deve avere i diritti di scrittura verso la destinazione.

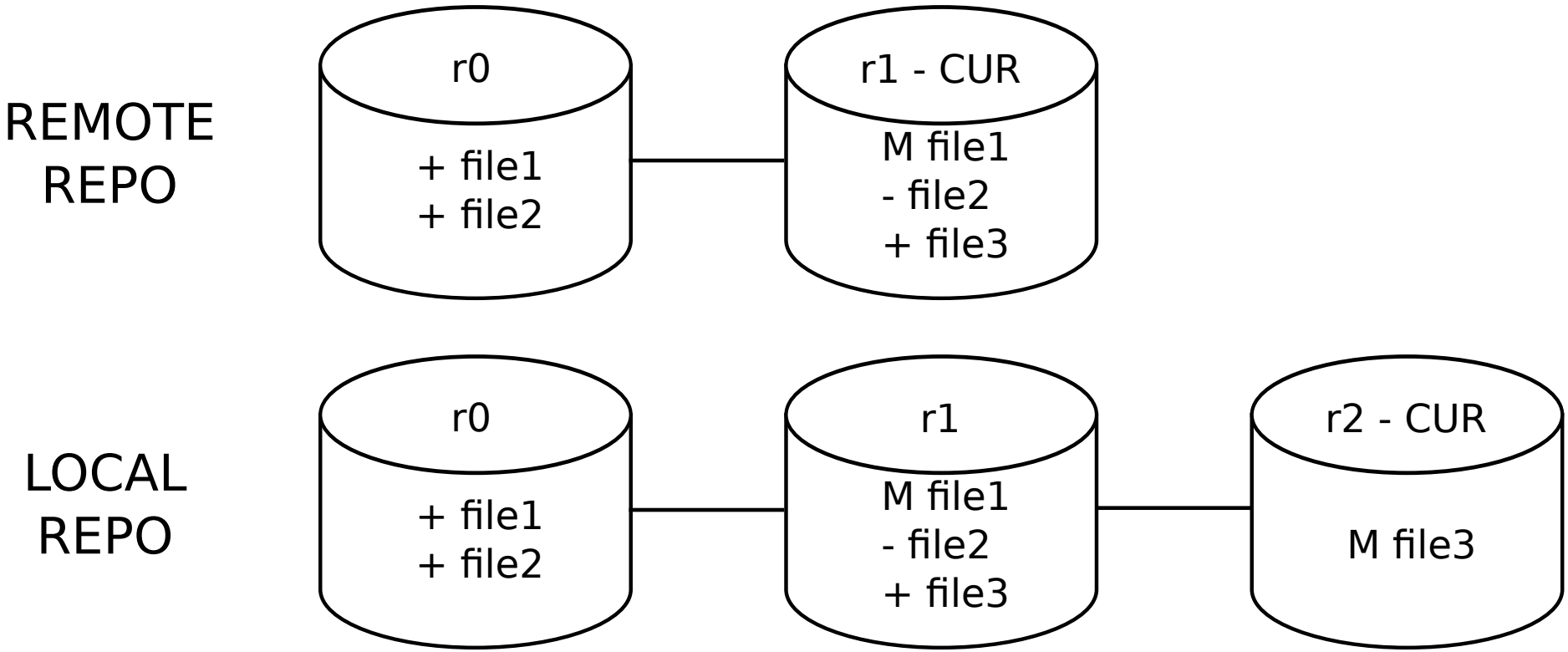
In Git

- `git push URI branchName`
  - Carica sul branch `branchName` `URI` tutti i commit del branch
- `git push remoteName branchName`
  - Carica sul remote `remoteName` tutti i commit del branch `branchName`
- `git push`
  - Carica dal remote di default (se presente) tutti i commit del branch corrente

Nel caso in cui ci fossero commit sul branch destinazione del repository remoto non presenti in quello locale, la `push` **fallirebbe**. In questo caso, infatti, è necessario effettuare prima una pull

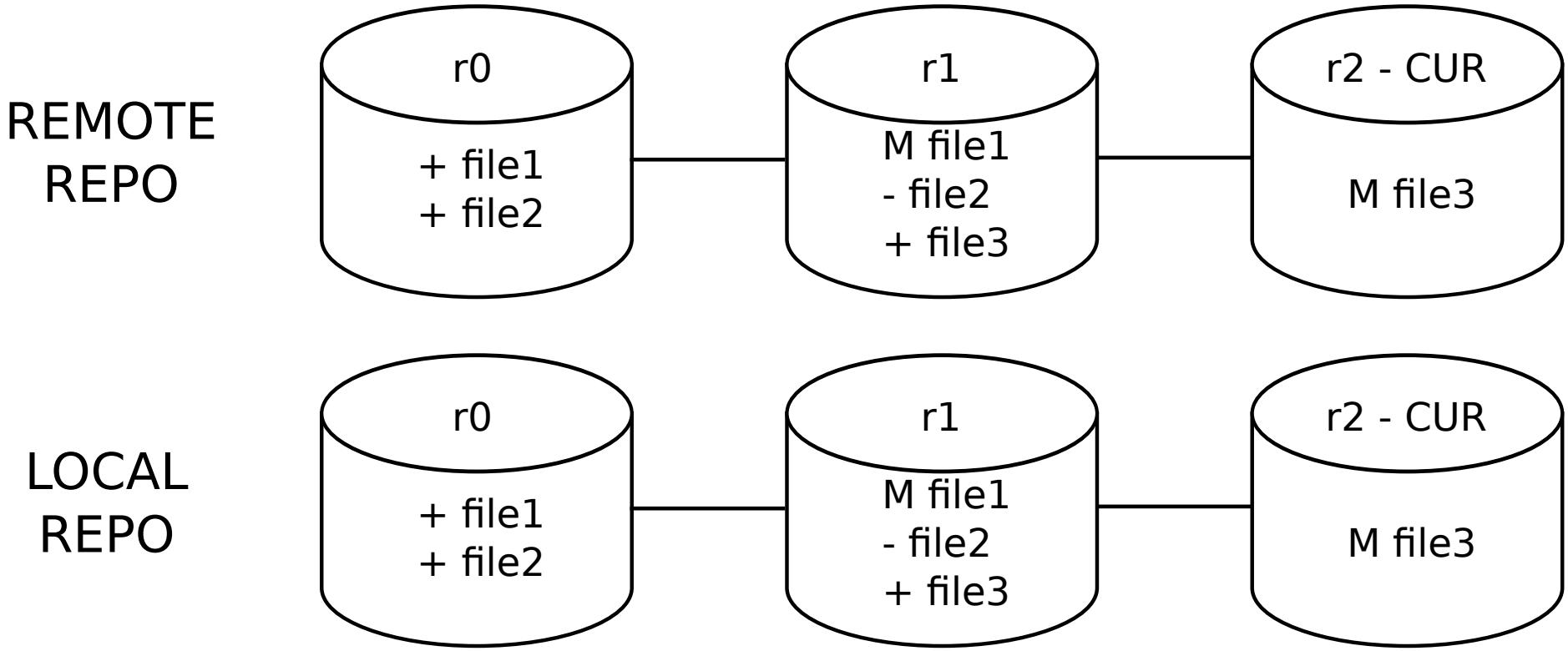
Esempio con push I

Situazione iniziale:



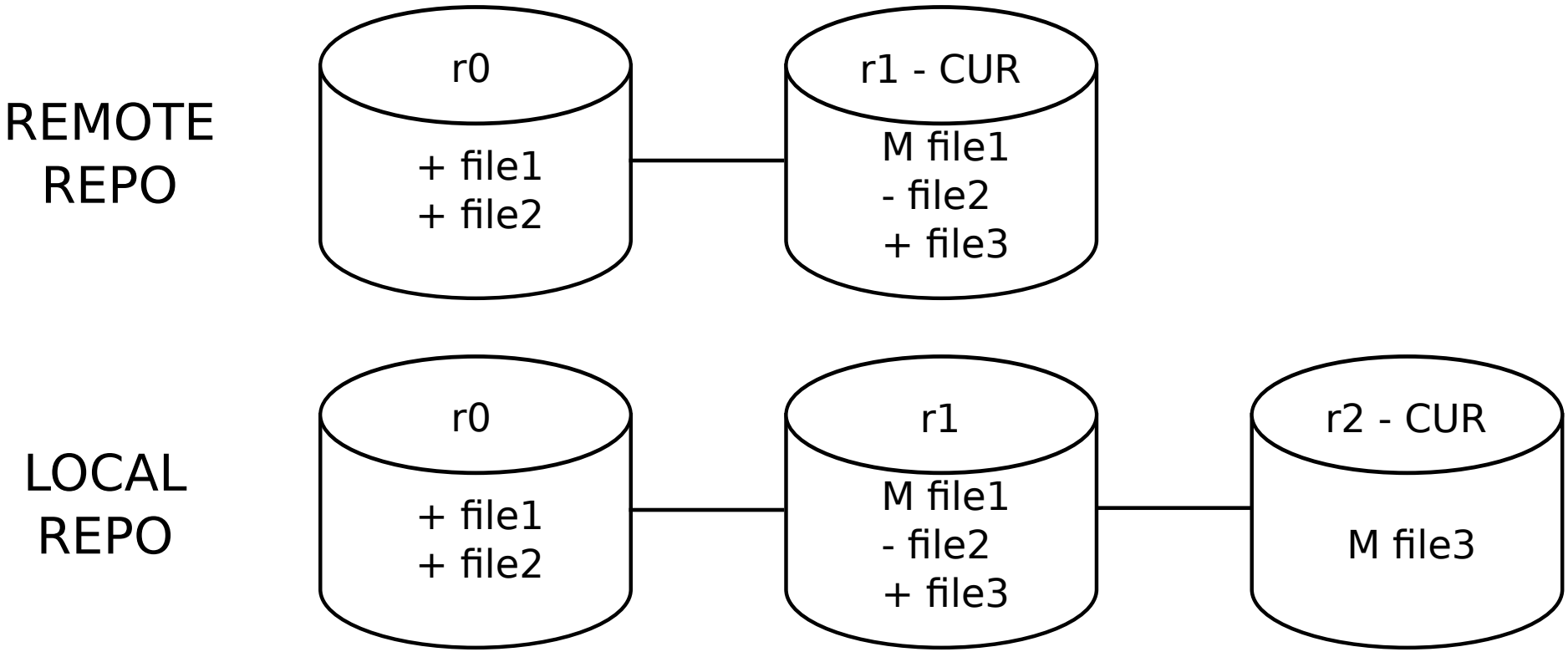
Esempio con push II

Local esegue: `git push indirizzo_di_remote_repo`



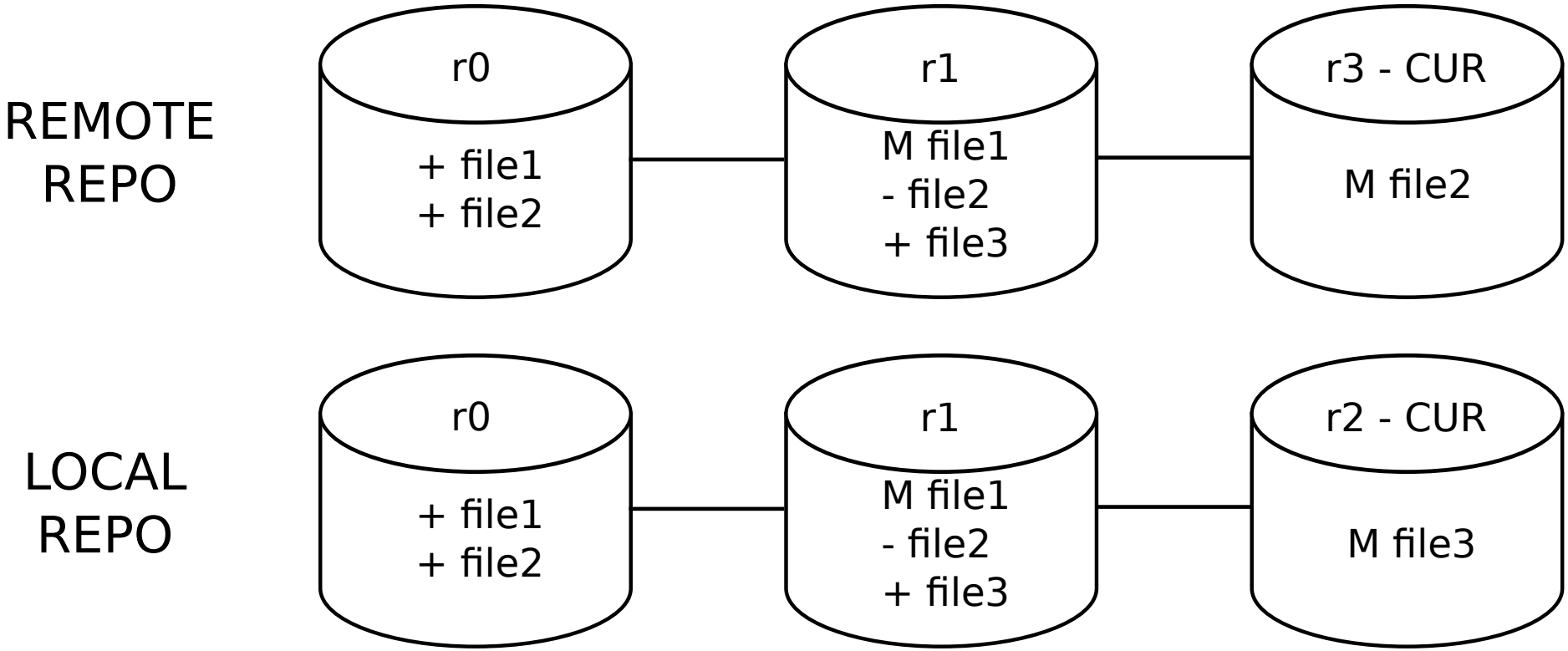
Lavorare in parallelo: esempio I

Situazione iniziale:



Lavorare in parallelo: esempio II

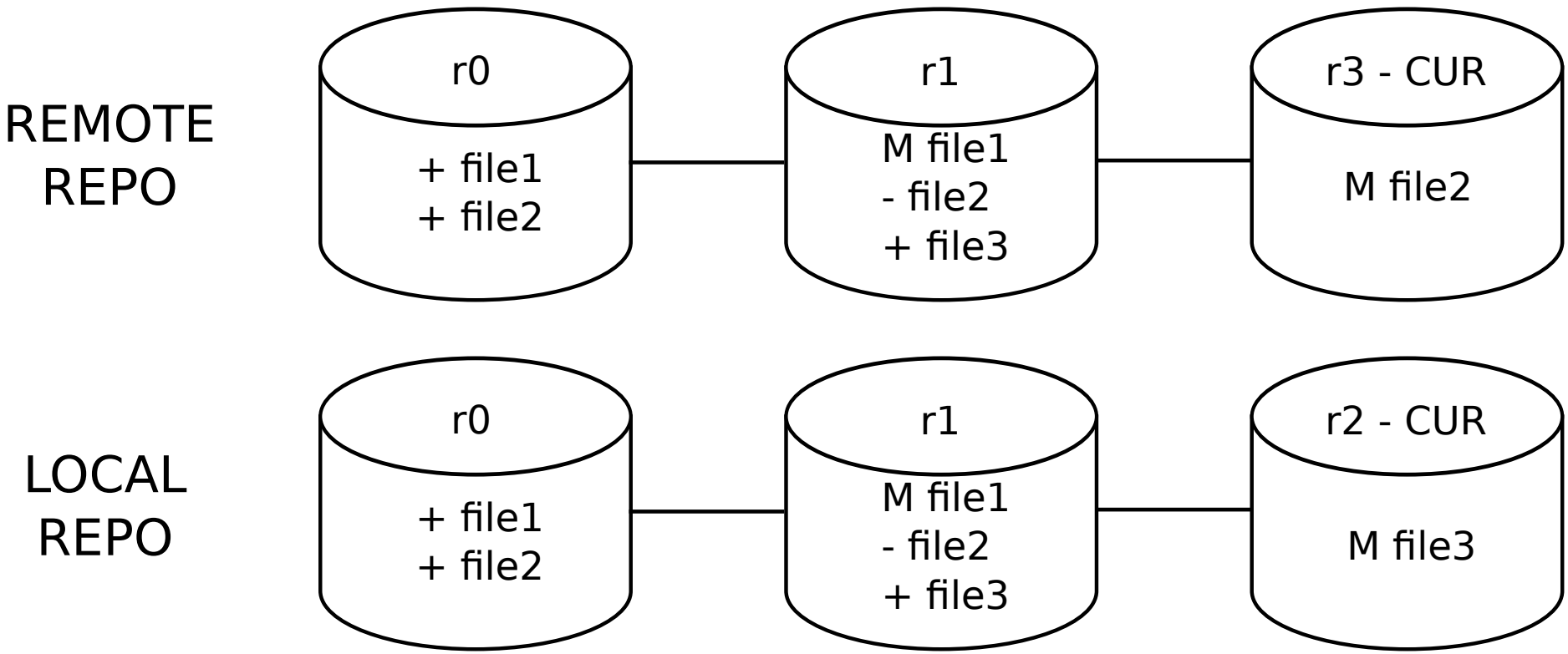
Remote esegue:
Modifica di `file2`
`git commit`



Lavorare in parallelo: esempio III

Local esegue:

`git push indirizzo_di_remote_repo`

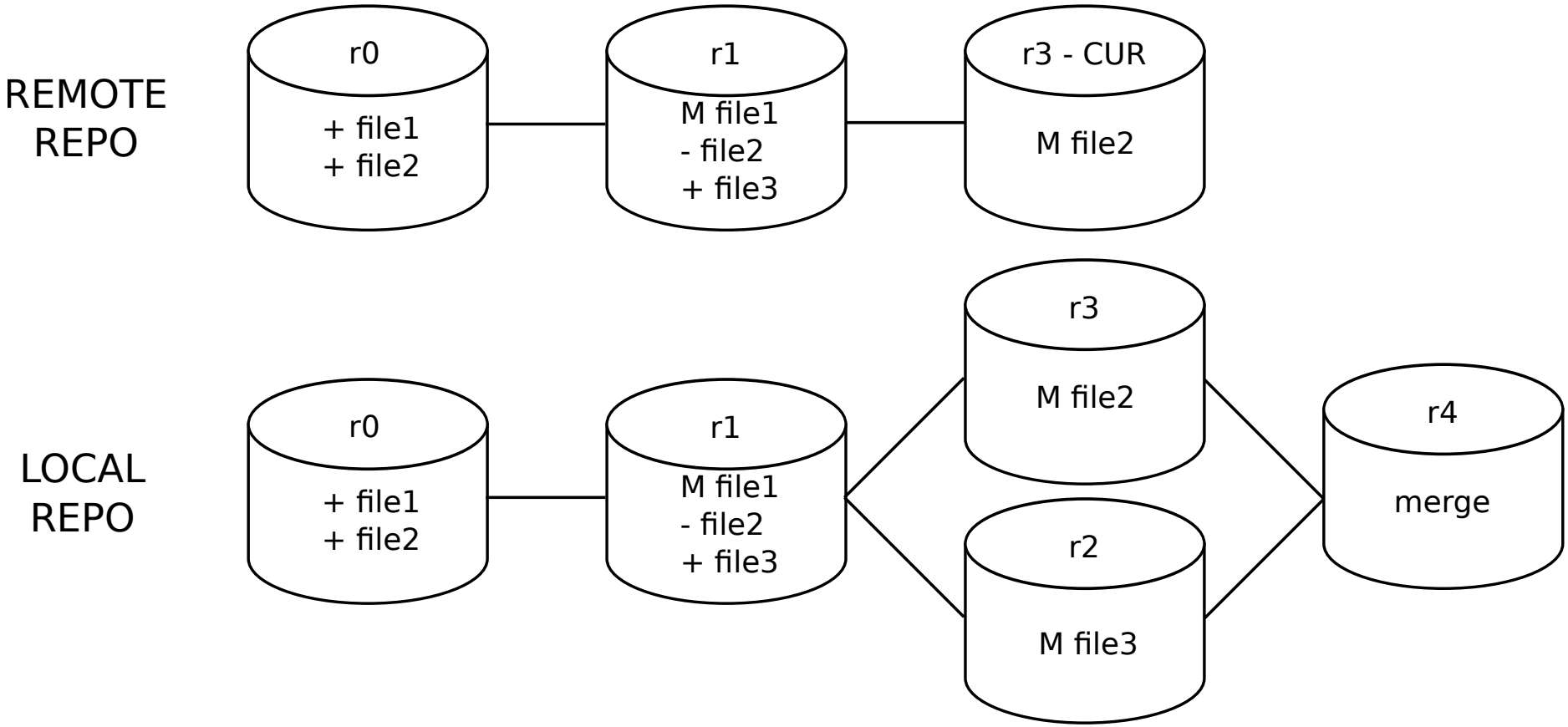


La push viene rifiutata: la radice dei due repository è diversa!

Lavorare in parallelo: esempio IV

Local esegue:

`git pull indirizzo_di_remote_repo`

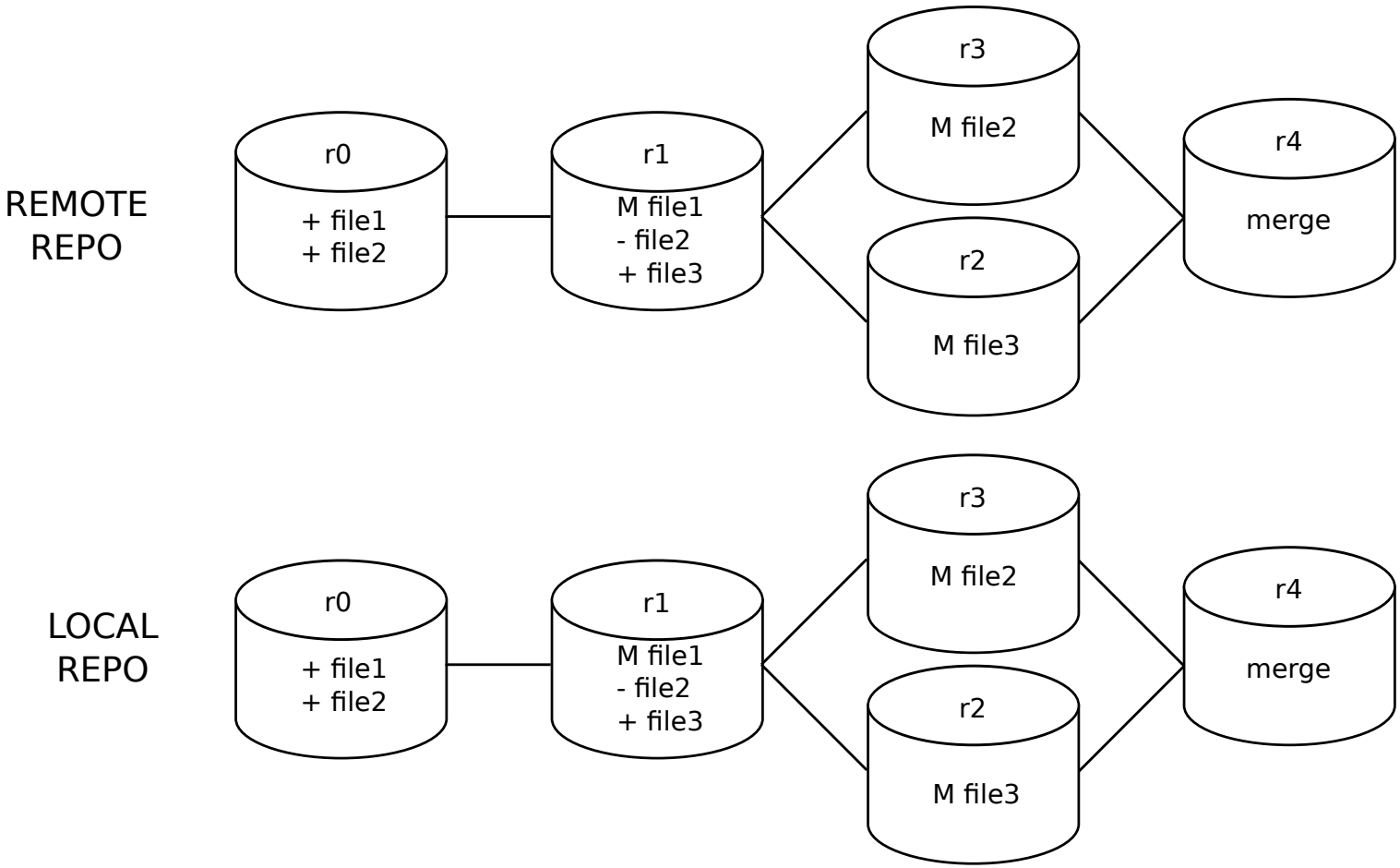


Ora è possibile effettuare la `push`!

Lavorare in parallelo: esempio V

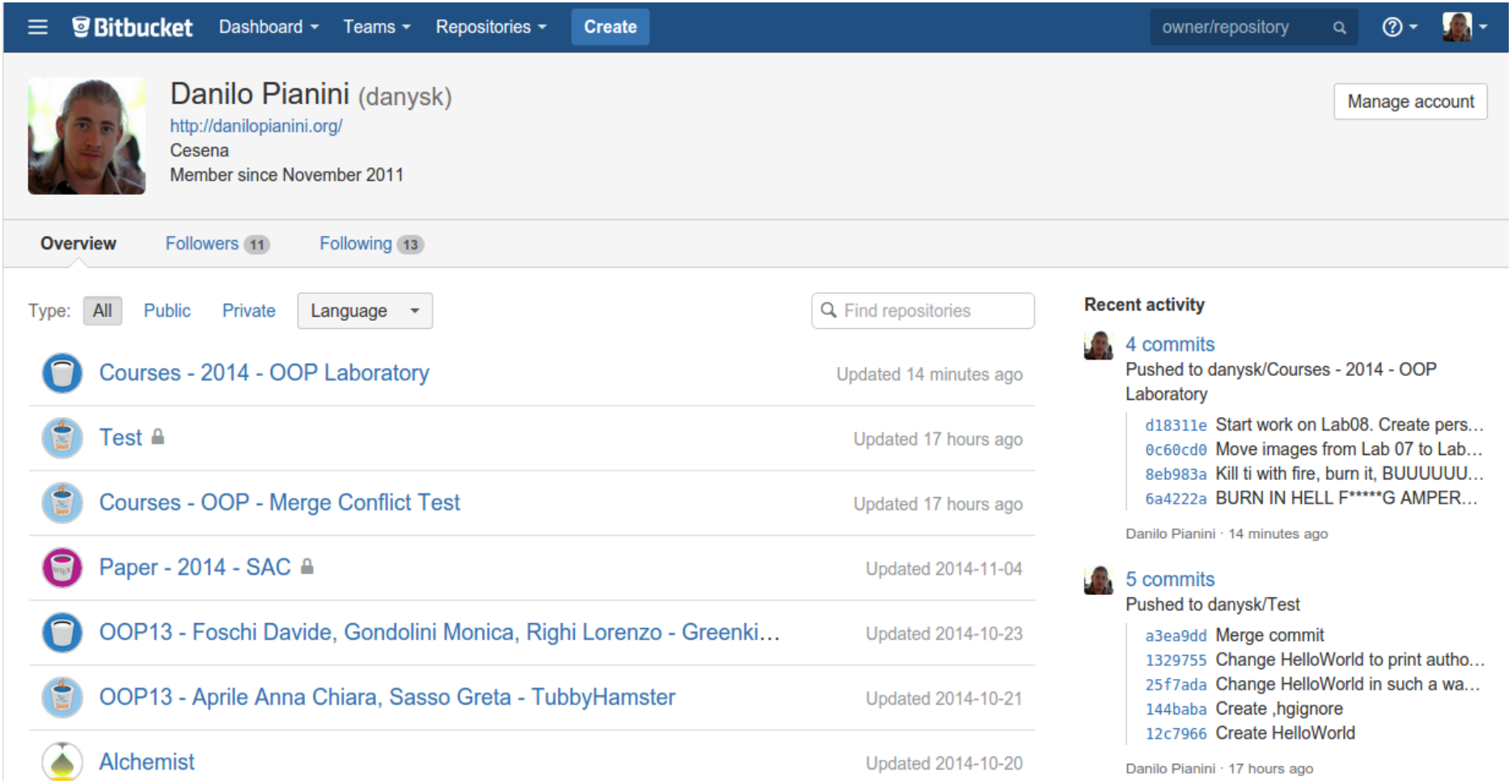
Local esegue:

`git push indirizzo_di_remote_repo`

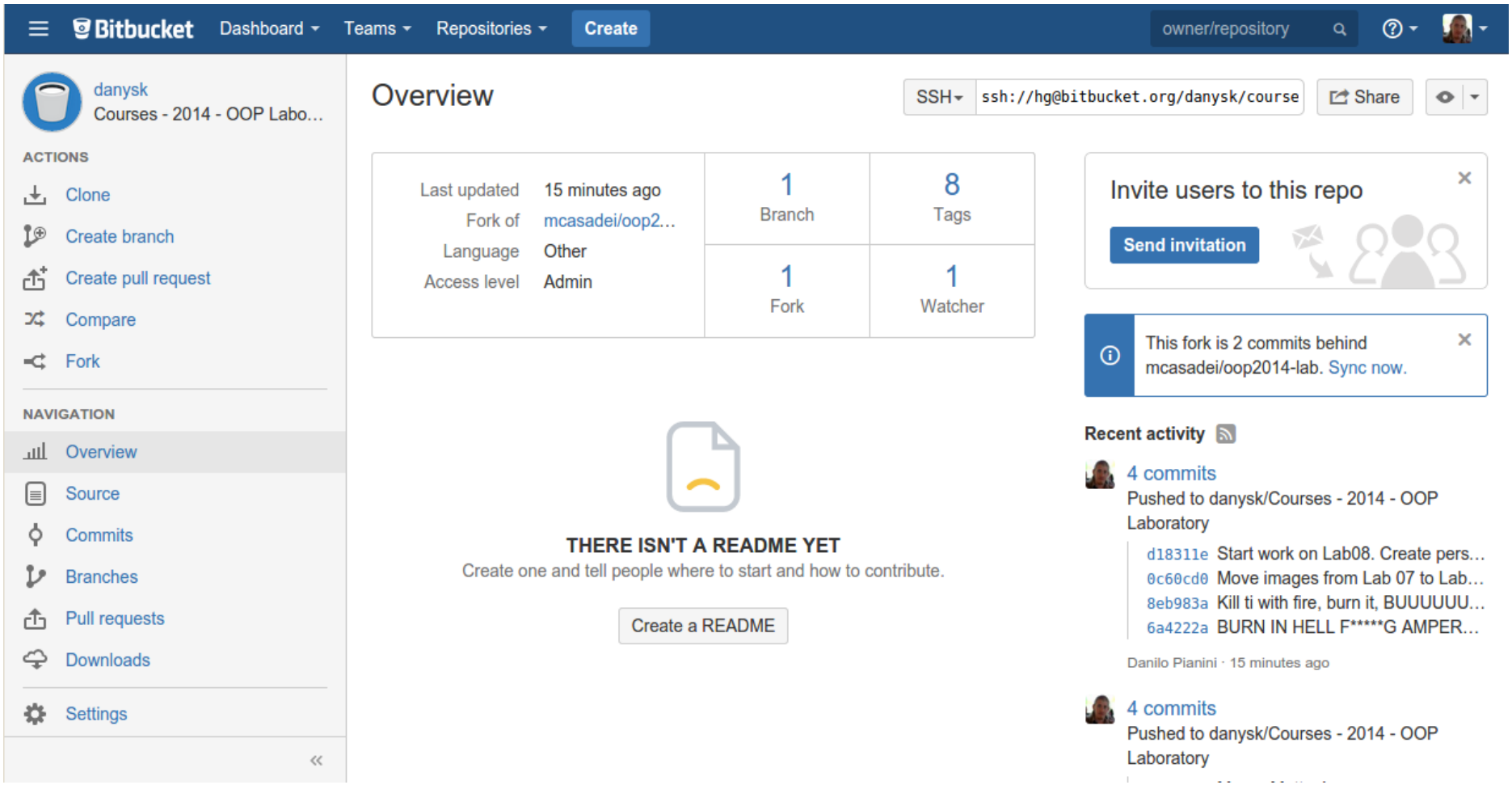


Outline

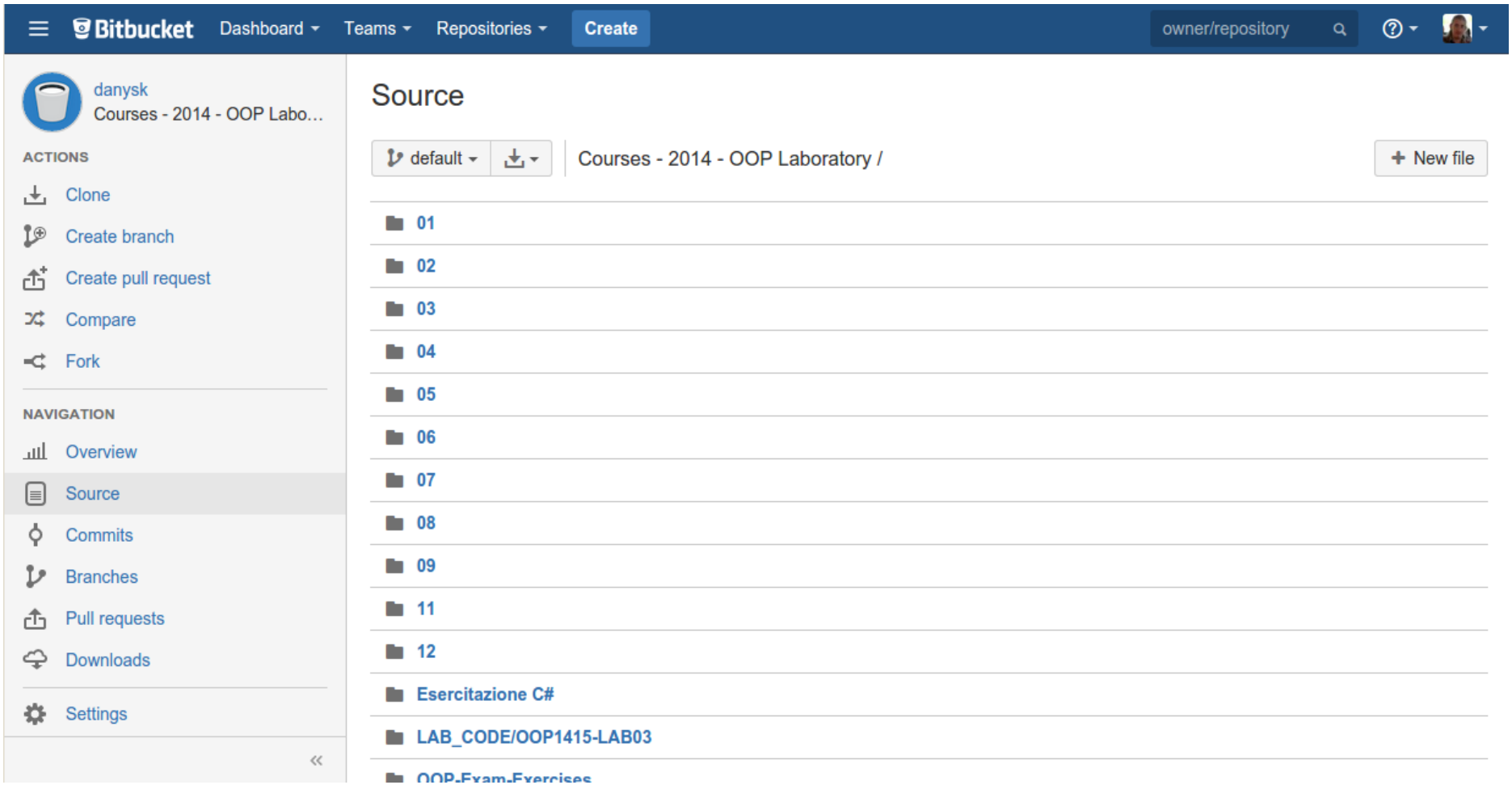
Bitbucket Overview



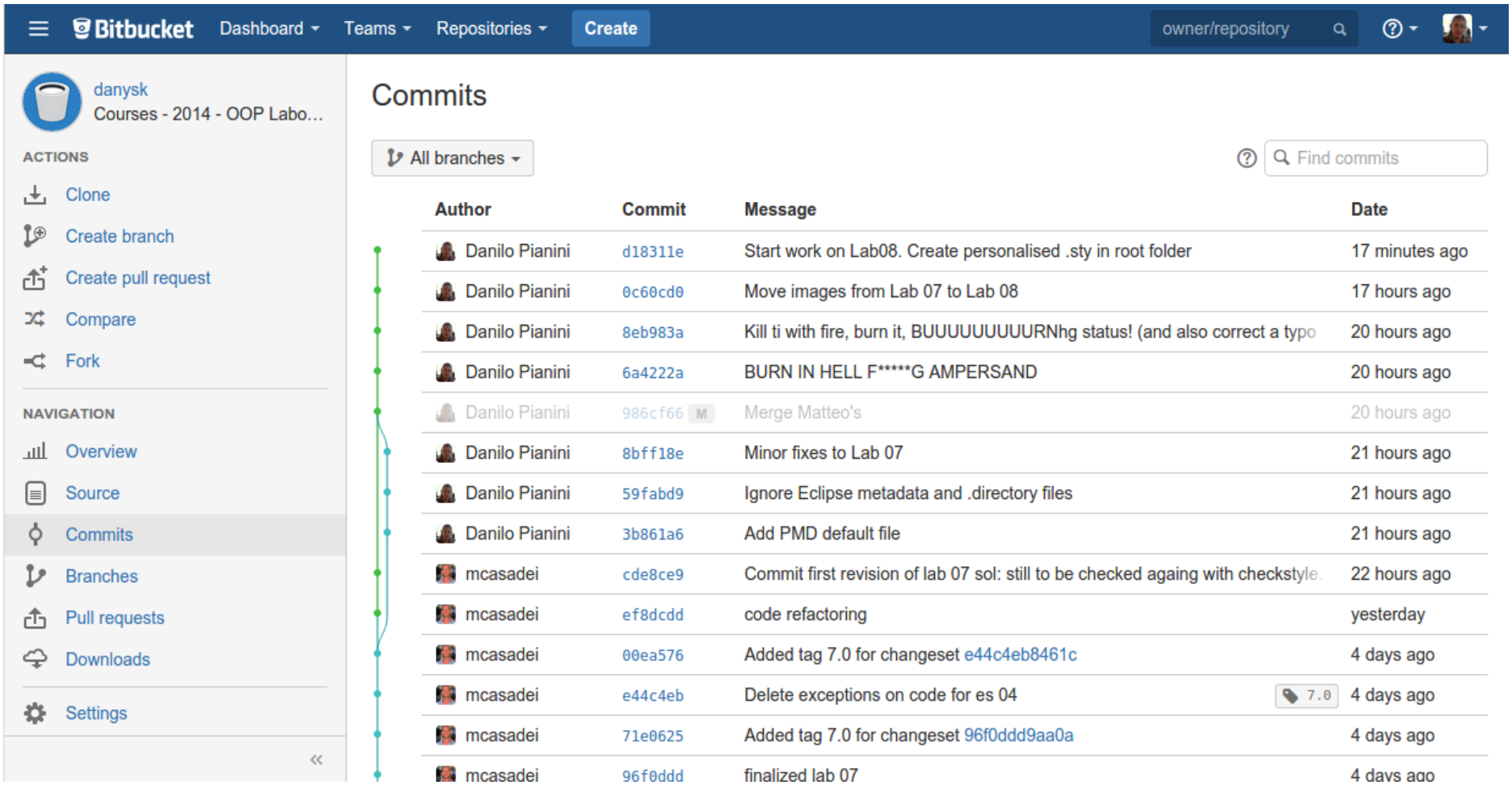
Bitbucket Overview



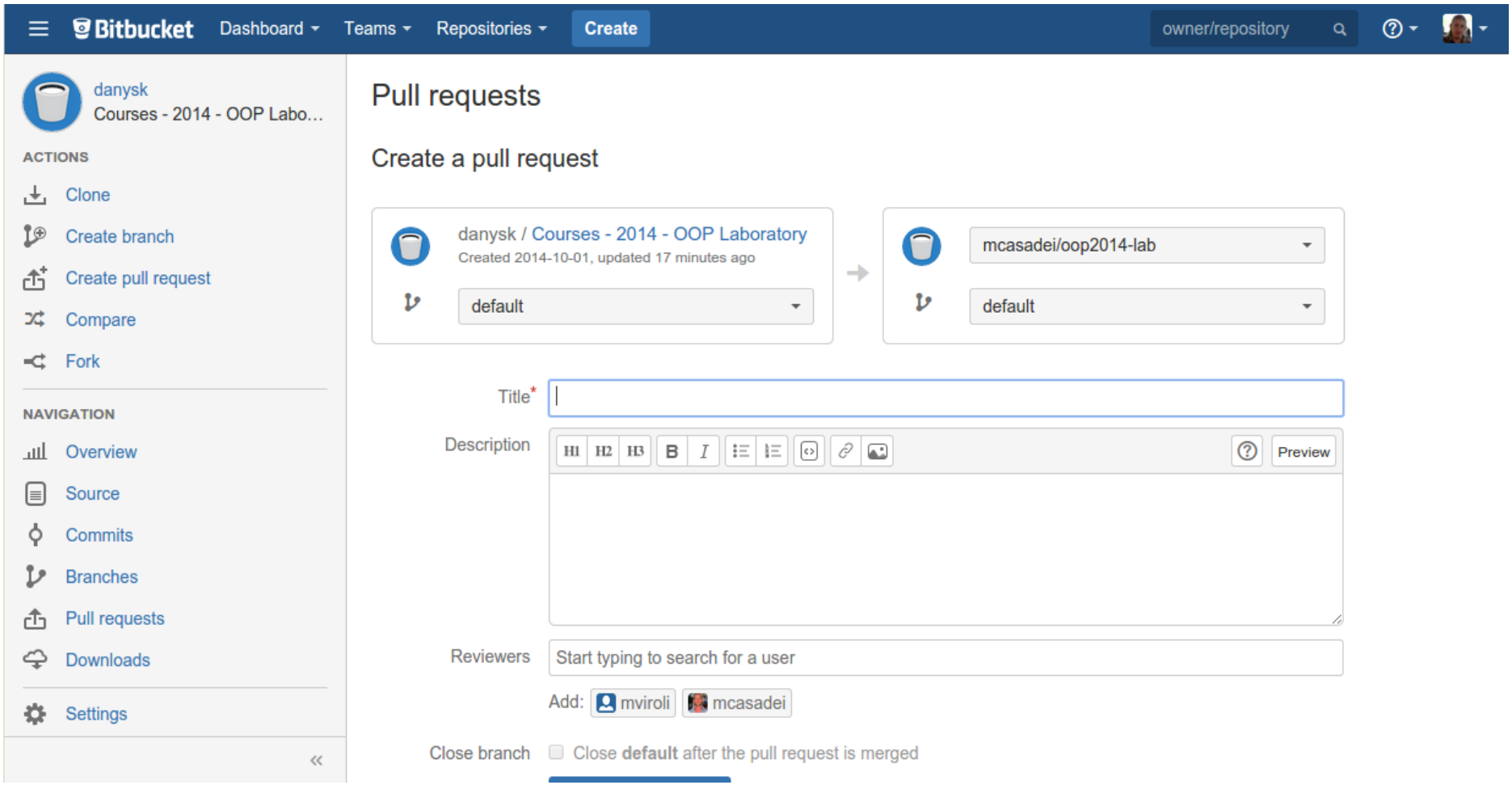
Bitbucket Overview



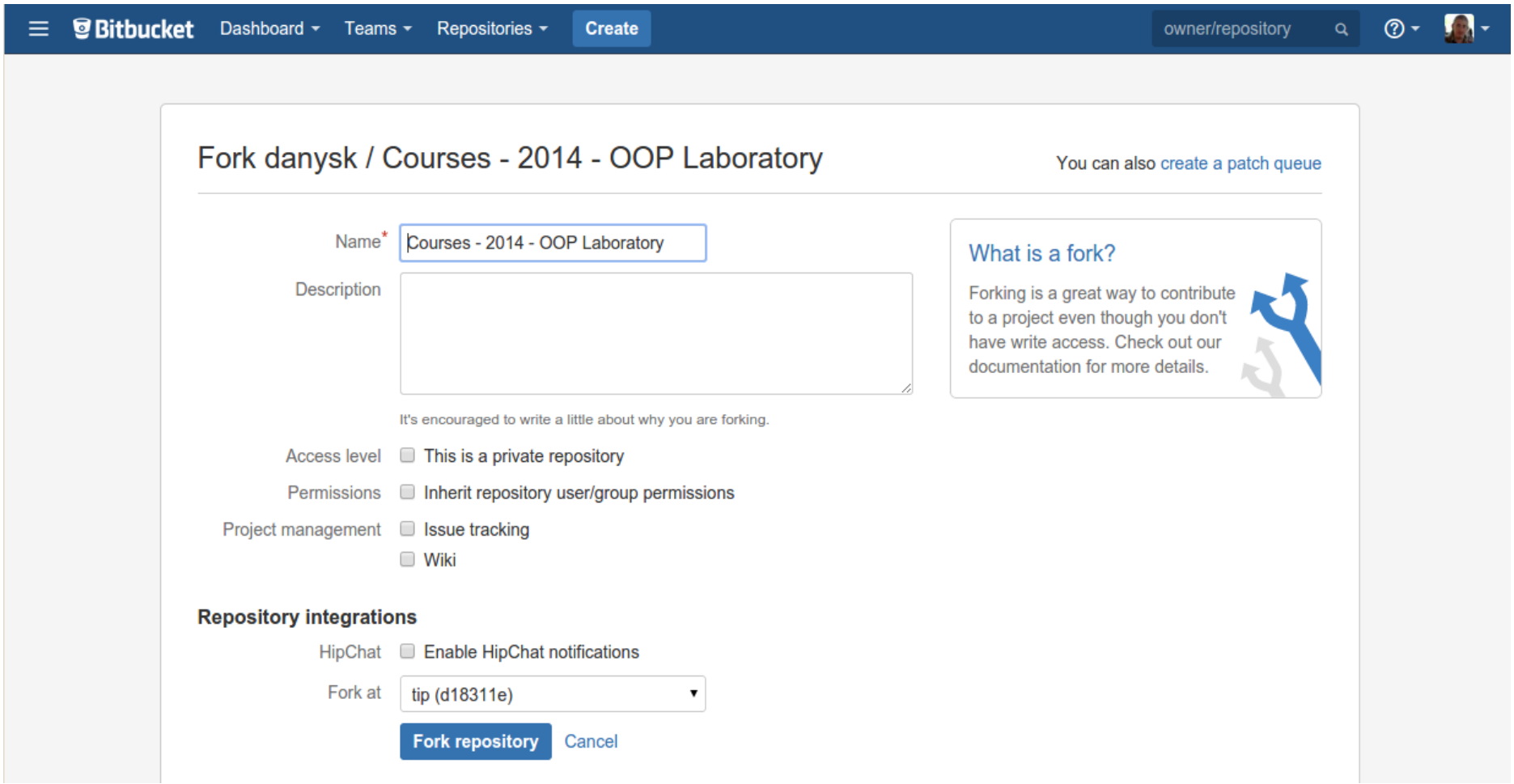
Bitbucket Overview



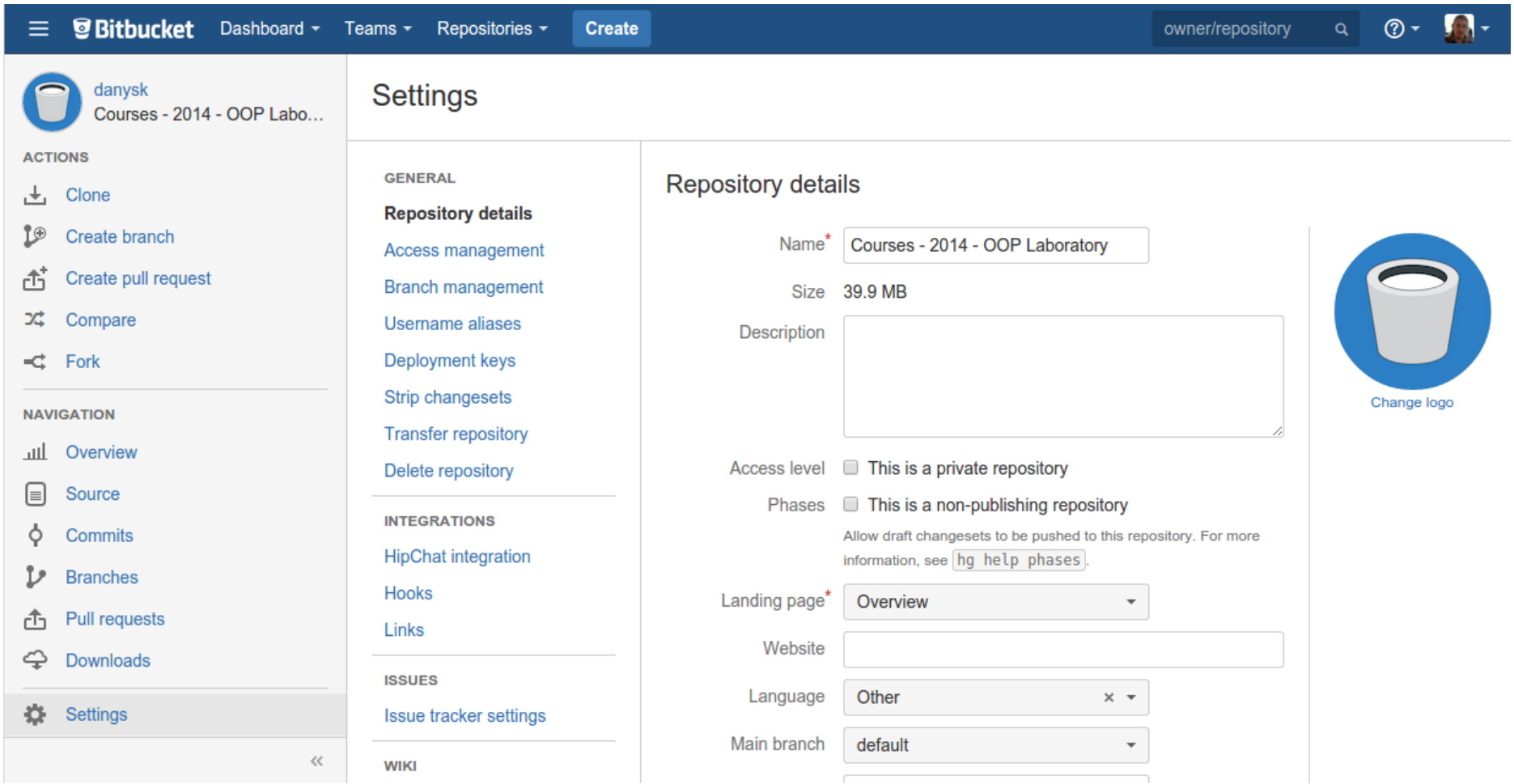
Bitbucket Overview



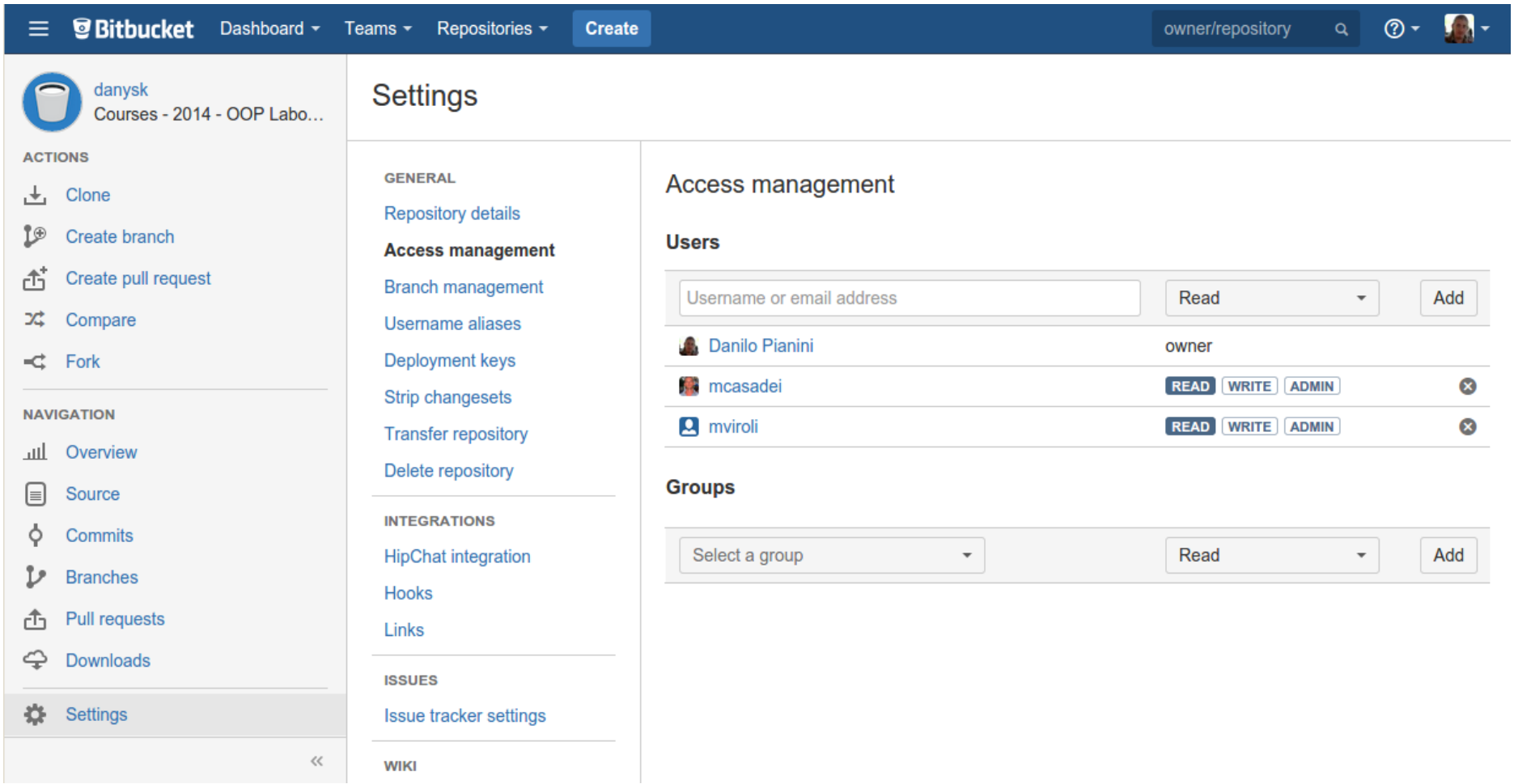
Bitbucket Overview



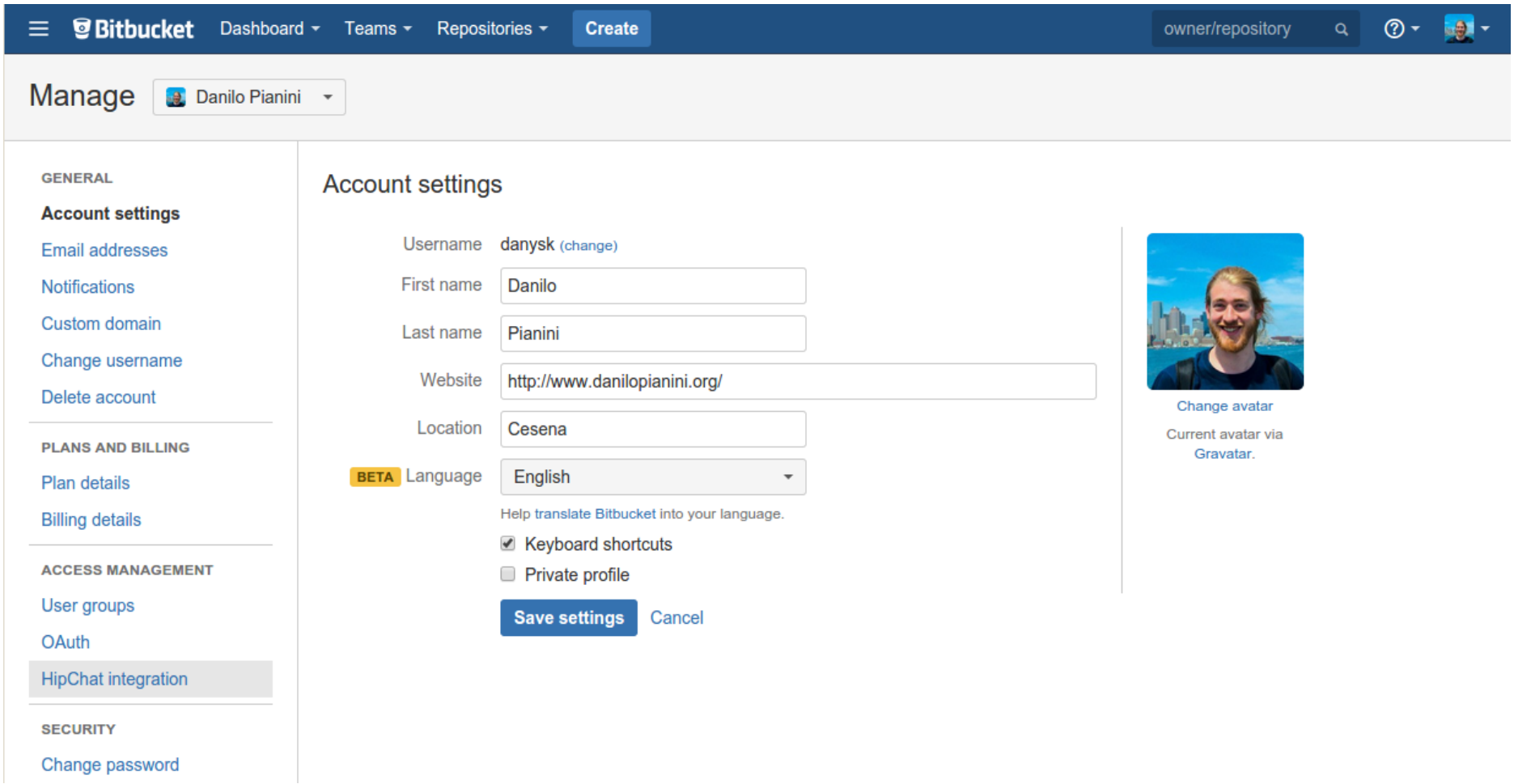
Bitbucket Overview



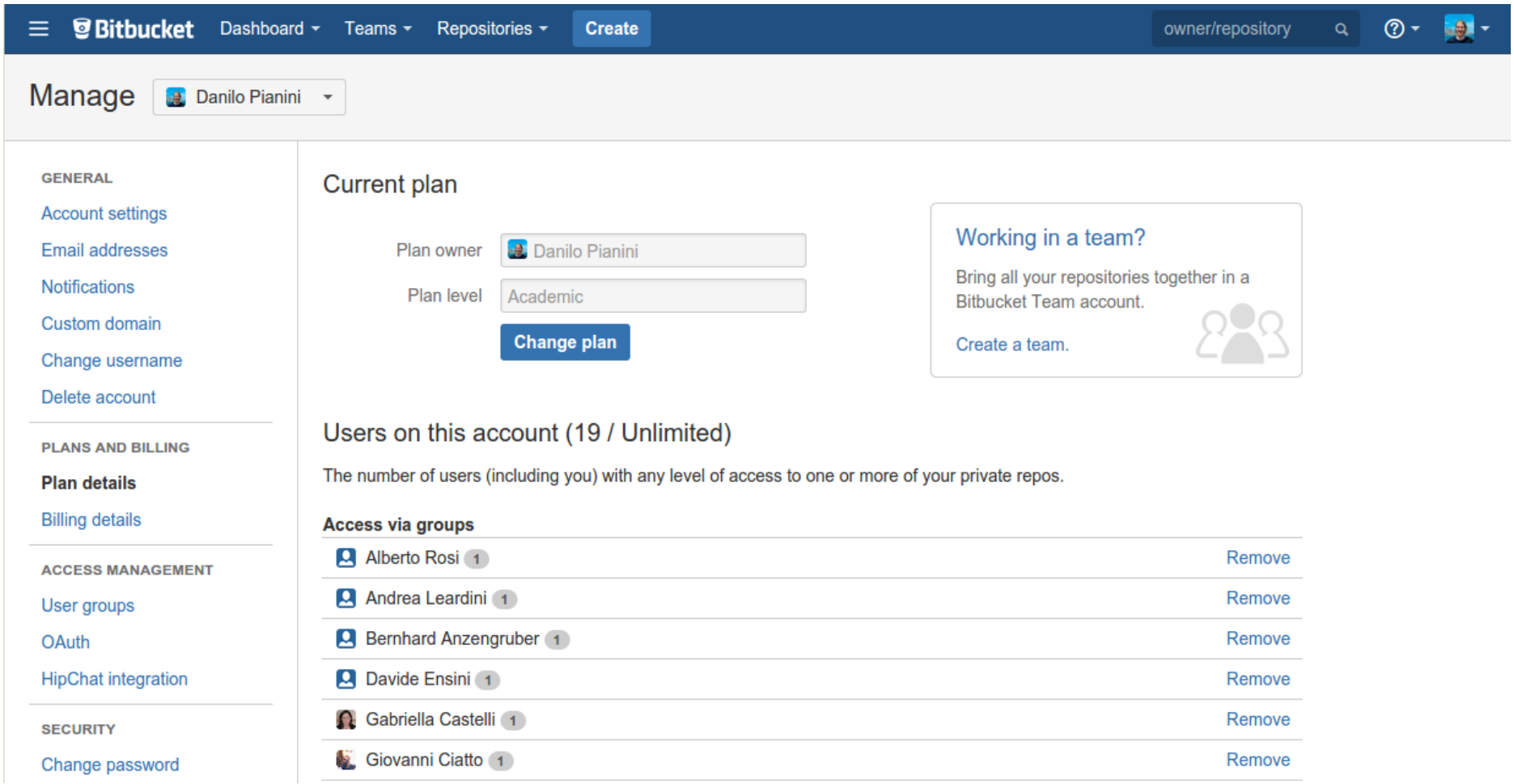
Bitbucket Overview



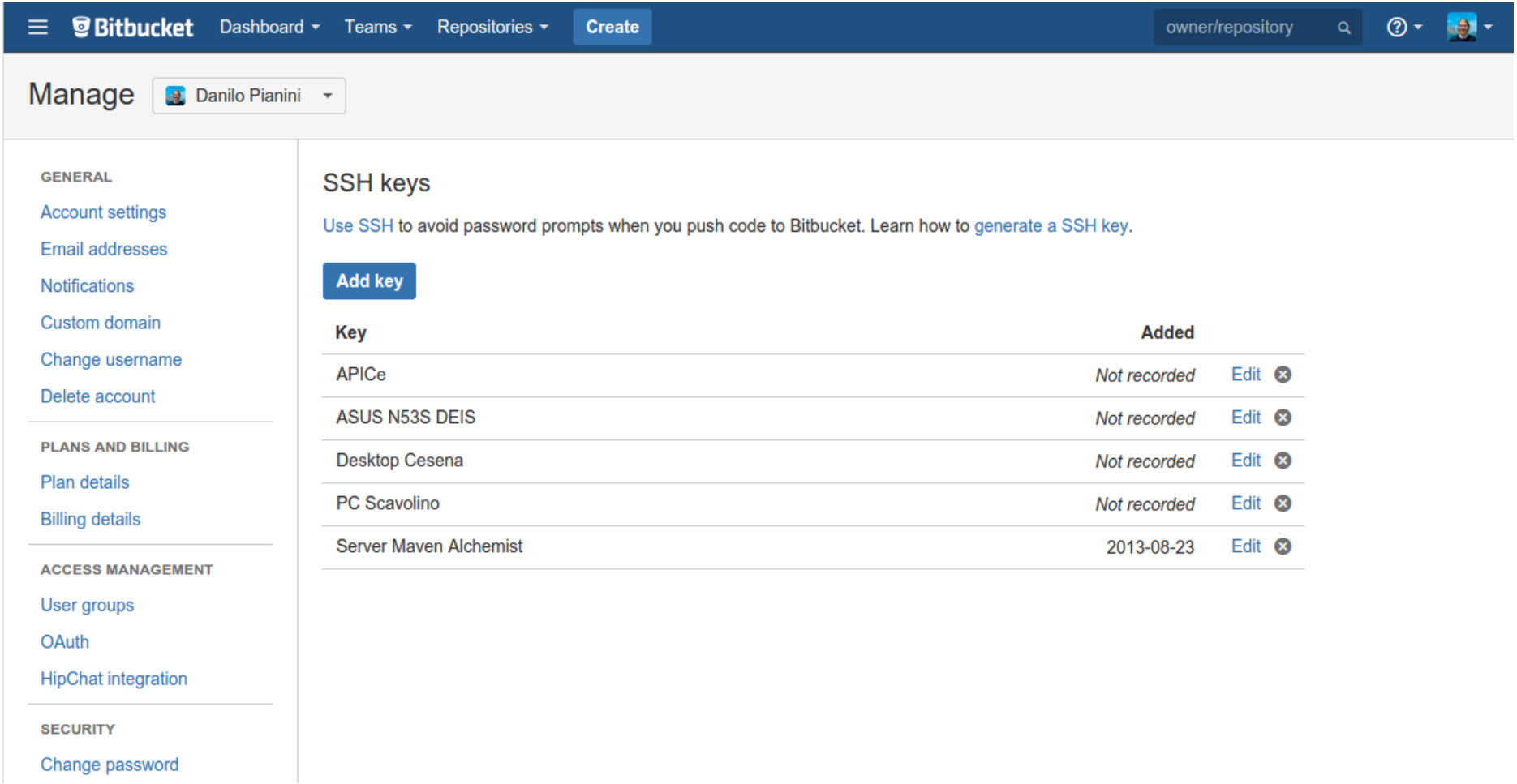
Bitbucket Overview



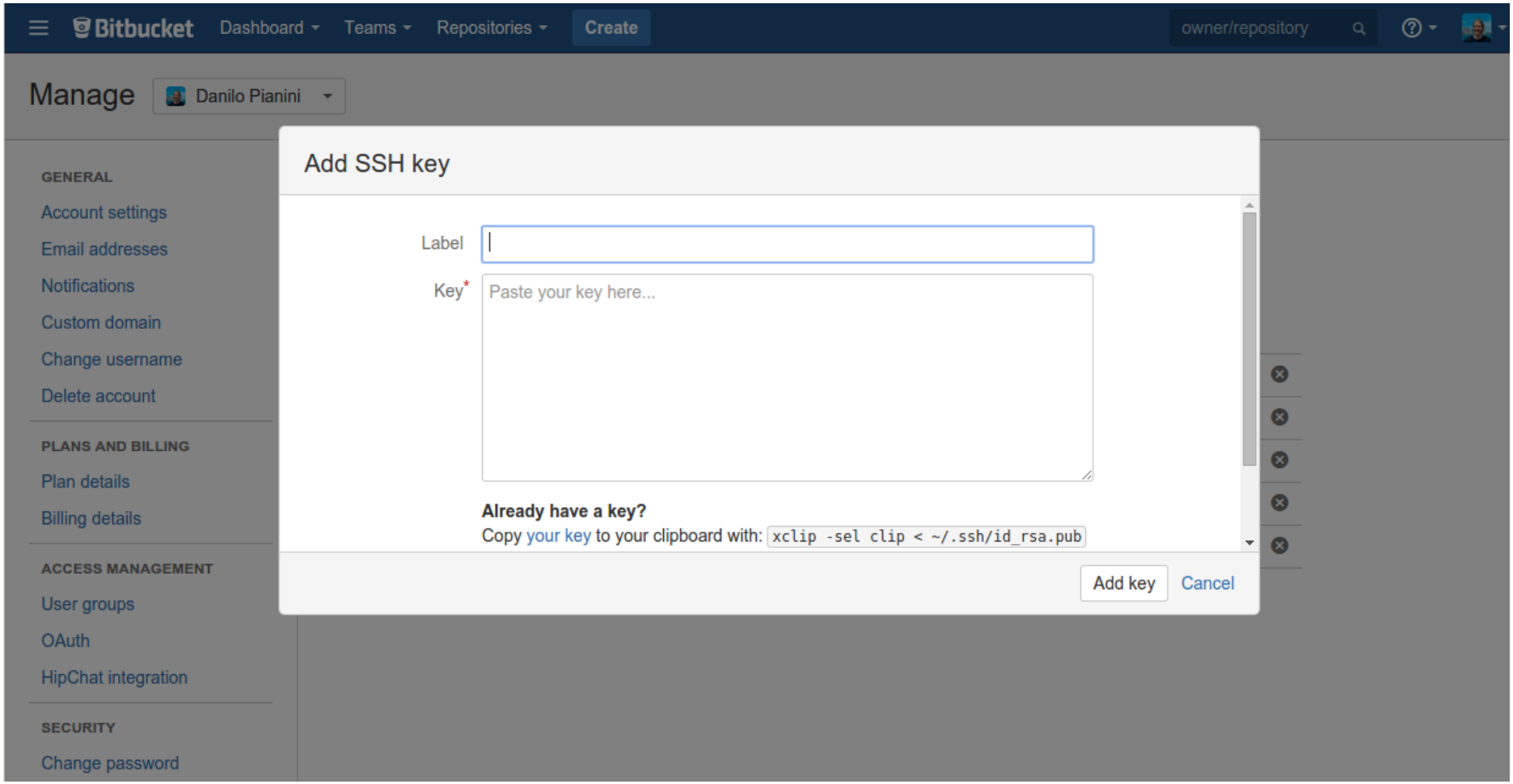
Bitbucket Overview



Bitbucket Overview



Bitbucket Overview



Su sistemi *nix, si utilizzi il comando `ssh-keygen -t rsa` per generare una coppia di chiavi pubblica e privata. Non si inserisca alcuna password e si mantengano le opzioni suggerite dal sistema. A questo punto, all'interno della cartella `~/.ssh/`, si troveranno i file delle chiavi. Si incolli nel campo key il contenuto del file `~/.ssh/id_rsa.pub`

Outline

Plugin eclipse

Installazione

Il plugin è preinstallato dentro la distribuzione di Eclipse.

Funzionalità

Il plugin mostrerà le proprie funzionalità dal menu "Team", ed esporrà le stesse funzionalità del tool a linea di comando, mostrando anche quelle che non abbiamo trattato, e che non dovete utilizzare se non avete **cristallino** il loro funzionamento da terminale.
Appare anche una nuova opzione per l'import di un progetto tramite clone di un repository Git.

È bene (come per tutte le utility con interfaccia grafica) fare un utilizzo **consapevole** del plugin Eclipse.
Vi consigliamo di preferire l'uso di Git da terminale!

Outline

1. Decentralized Version Control Systems
   - Risolvere i merge conflict
   - Lavorare in remoto: clone, fetch, pull, push
   - Hosting e Bitbucket
   - Plugin Eclipse (eGit)
   - Features avanzate e prossimo episodio
2. Testing con JUnit

Aspetti avanzati (cenni)

Rebasing

Procedura alternativa al merge, in cui i due commit, invece di essere fusi, vengono messi in sequenza. Simula una storia "lineare" anche dove è in realtà parallela.

Cherry picking

Pull di un singolo commit o di un piccolo gruppo di commit. Spesso utilizzato quando si desidera avere un bugfix che si trova in un altro branch, ma non tutto il resto. Molto comune nel backporting delle patch di sicurezza.

Bisection

Strategia per scoprire bug in maniera automatizzata, testando il software a varie versioni (ricerca dicotomica). Una volta trovata la prima versione dove il bug si verifica, si controllano commit message e le differenze.

Nel prossimo episodio (verso fine corso)

- Abbiamo in mano uno strumento formidabile per la gestione di progetti
- Non ci resta che trovare una strategia efficace per massimizzarne l'efficacia
- Vedremo una strategia di lavoro che promuoverà e faciliterà il lavoro in parallelo

Outline

1. Decentralized Version Control Systems
   - Risolvere i merge conflict
   - Lavorare in remoto: clone, fetch, pull, push
   - Hosting e Bitbucket
   - Plugin Eclipse (eGit)
   - Features avanzate e prossimo episodio

2. Testing con JUnit

Importanza del testing

(Tutte cose che discuterete meglio a lezione)

- Il testing è importante quanto il sistema stesso
- Si dovrebbero scrivere **prima** i test, e solo dopo realizzare il sistema
  - Consente di dare una specifica formale del comportamento delle entità che si stanno modellando
  - Consente di intercettare immediatamente eventuali problemi in fase di sviluppo
  - Fornisce una base per effettuare "regression testing", ossia per evitare che nuove funzionalità introducano dei bug
  - È vero per tutte le branche dell'ingegneria: prima ancora di realizzare un motore, si prepara il suo banco prova
- Esistono degli strumenti appositi che facilitano il testing!

Unit vs integration I

Unit testing

- Scopo ristretto: testa "piccole parti di codice"
  - idealmente, una sola "unità", ad esempio una sola implementazione di una sola interfaccia
- Servono a supportare lo sviluppatore
  - L'utente ne beneficia indirettamente, ottenendo software con meno difetti
- Non hanno dipendenze da sistemi esterni a quello in test
- Testano la consistenza interna

Unit vs integration II

Integration testing

- Scopo generale: verifica che diverse parti del sistema lavorino bene insieme
- Hanno frequentemente dipendenze da sistemi esterni
- Molto più impegnativi da realizzare
  - Possono richiedere anche risorse esterne
    - Database
    - Hardware apposito
    - Macchine virtuali
- Testano il funzionamento globale del sistema
  - Tanto che a volte si fanno dei veri e propri dimostratori...

JUnit I

Generalità

JUnit è il framework per lo unit testing più usato nel mondo Java

- Probabilmente, la libreria Java (escluso il JDK) più usata in assoluto
- Sfrutta alcune funzionalità di Java che conoscerete solo a fine corso
  - Reflection e annotazioni

JUnit II

Funzionamento

- Si crea una classe di test
  - Normalissima classe Java, come tutte le altre
- Si creano uno o più metodi con:
  - Modificatore di visibilità `public`
  - Tipo di ritorno `void`
  - Nome a piacere (formato suggerito: "`testThingYouAreTesting`")
  - Nessun argomento
  - Annotazione `@Test`
    - Ossia, si scrive `@Test` prima di `public`
    - L'annotazione va importata
- Nel corpo dei metodi:
  - Si creano e usano le entità che si vogliono testare
  - Si verificano le proprietà usando le *asserzioni* fornite da JUnit
- I metodi annotati saranno eseguiti senza necessità di alcun `main`

JUnit III

Asserzioni

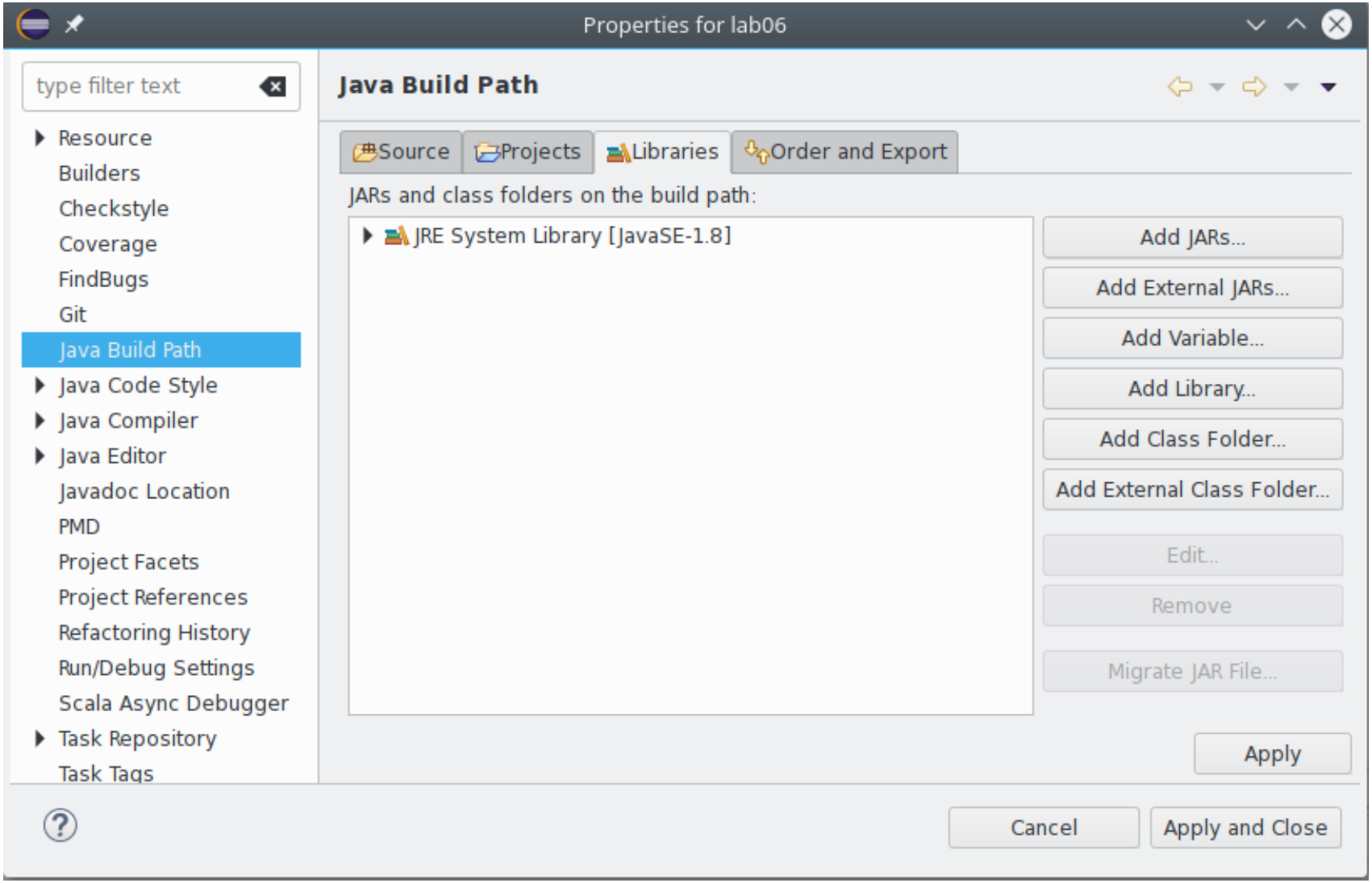
Sono particolari metodi che verificano una certa condizione, e fanno fallire il test se non è rispettata

- Metodi statici della classe `org.junit.Assert`
- `assertTrue(boolean)` — verifica che la condizione passata sia `true`
- `assertFalse(boolean)` — verifica che la condizione passata sia `false`
- `assertEquals(Object, Object)` — verifica che i due argomenti passati siano uguali (chiama la `equals`)
- `assertEquals(double, double, double)` — verifica che il modulo della differenza fra i primi due argomenti sia inferiore al terzo
  - Non si deve mai fare == fra numeri in virgola mobile
- `assertNotEquals` — reciproco di `assertEquals`
- `fail()` — Fa fallire il test immediatamente
  - Ad esempio se non viene generata una eccezione che ci si sarebbe attesa

Junit in Eclipse I

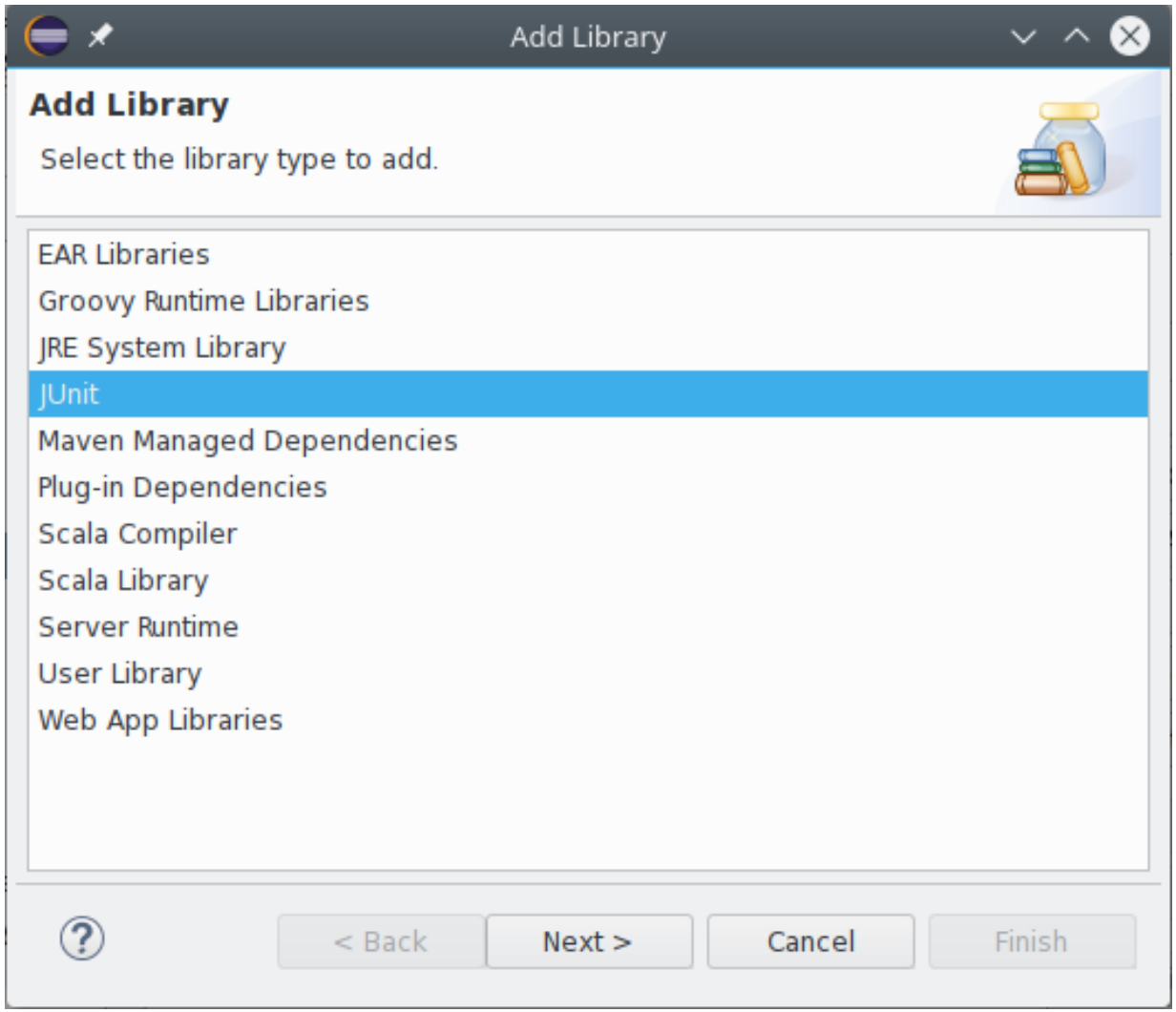
Per usare una libreria in Eclipse, occorre segnalare all'IDE che vogliamo inserirla nel classpath. Per JUnit, esiste un percorso "di favore" (viene fornita con Eclipse, non occorre installare alcunché).

Junit in Eclipse II



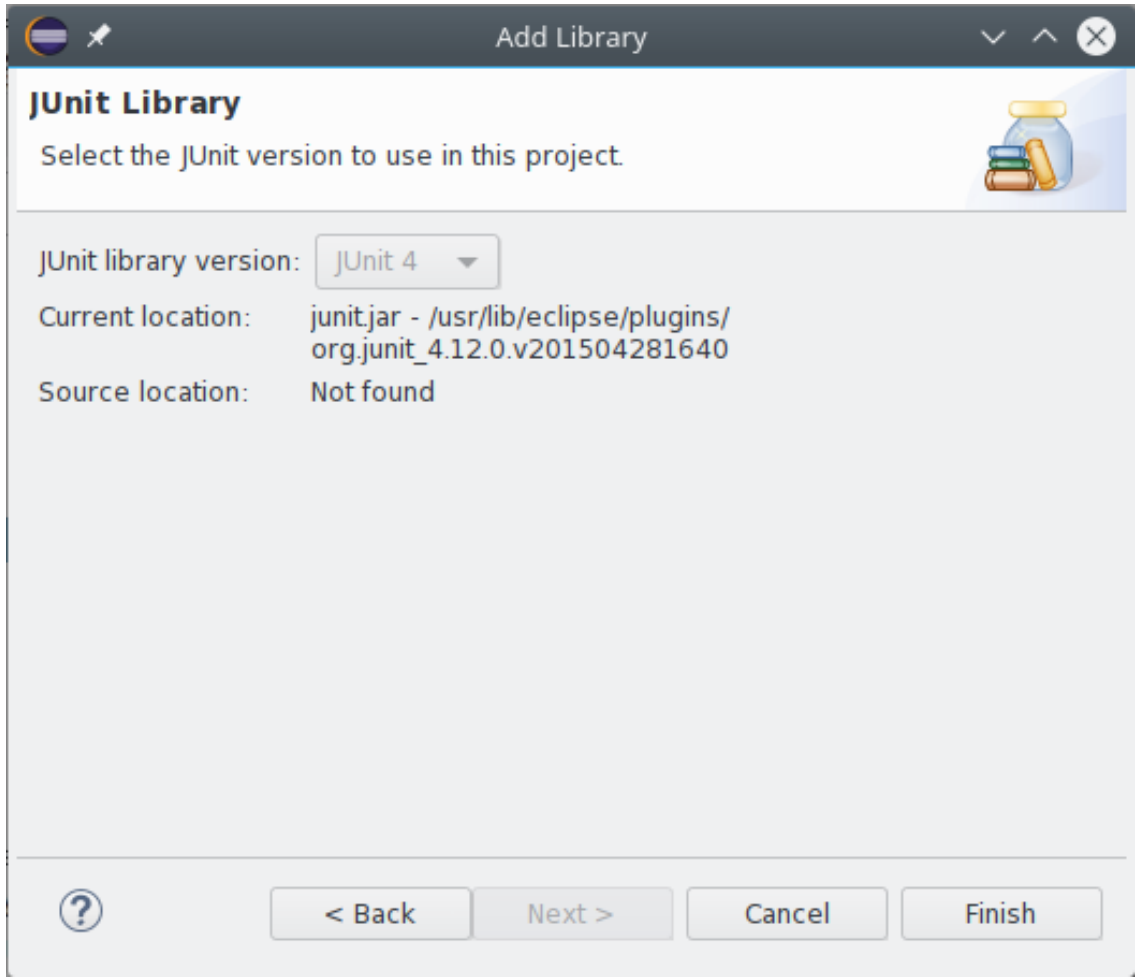
click su “Add library”

Junit in Eclipse III



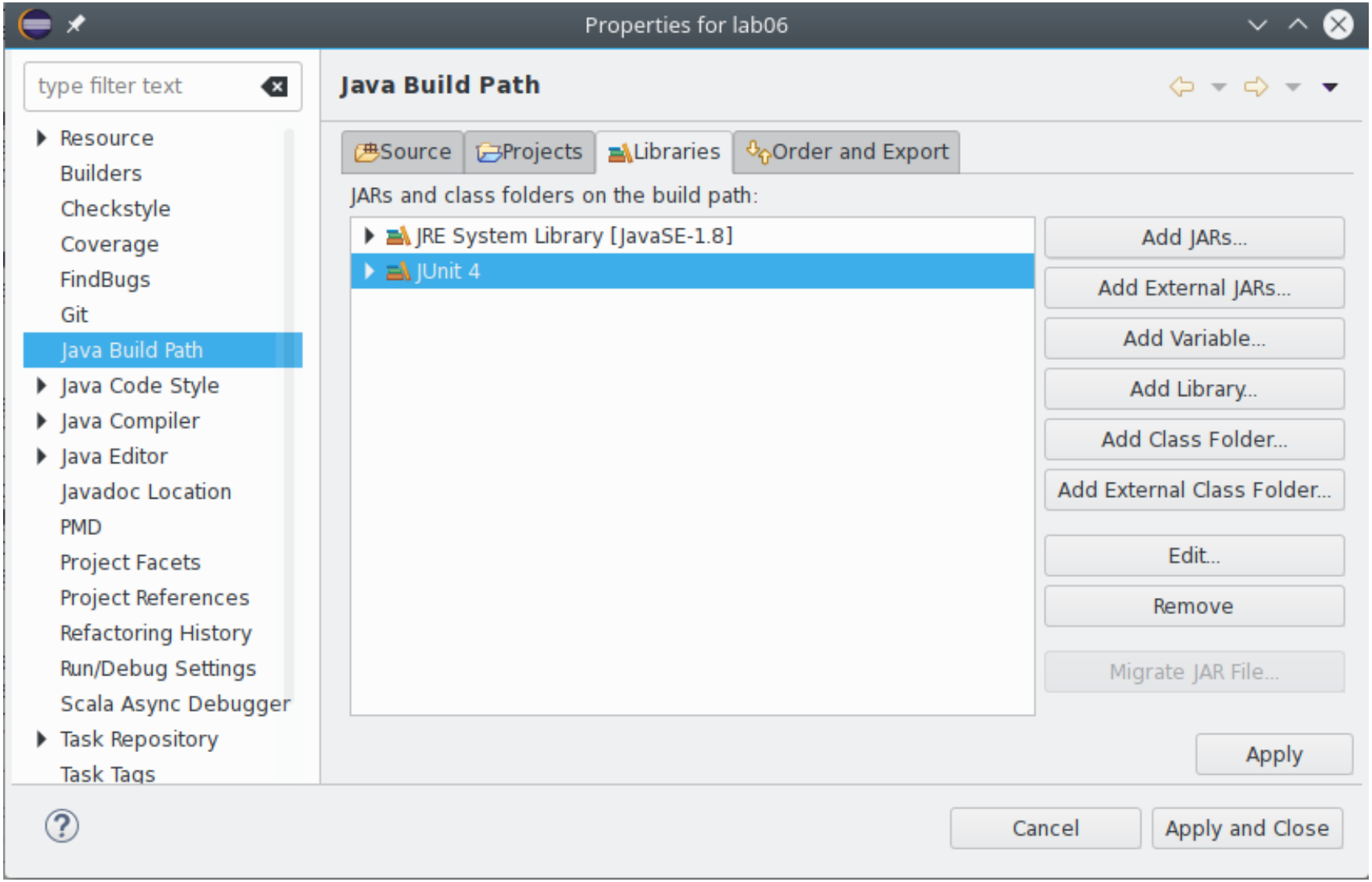
click su "Next"

Junit in Eclipse IV



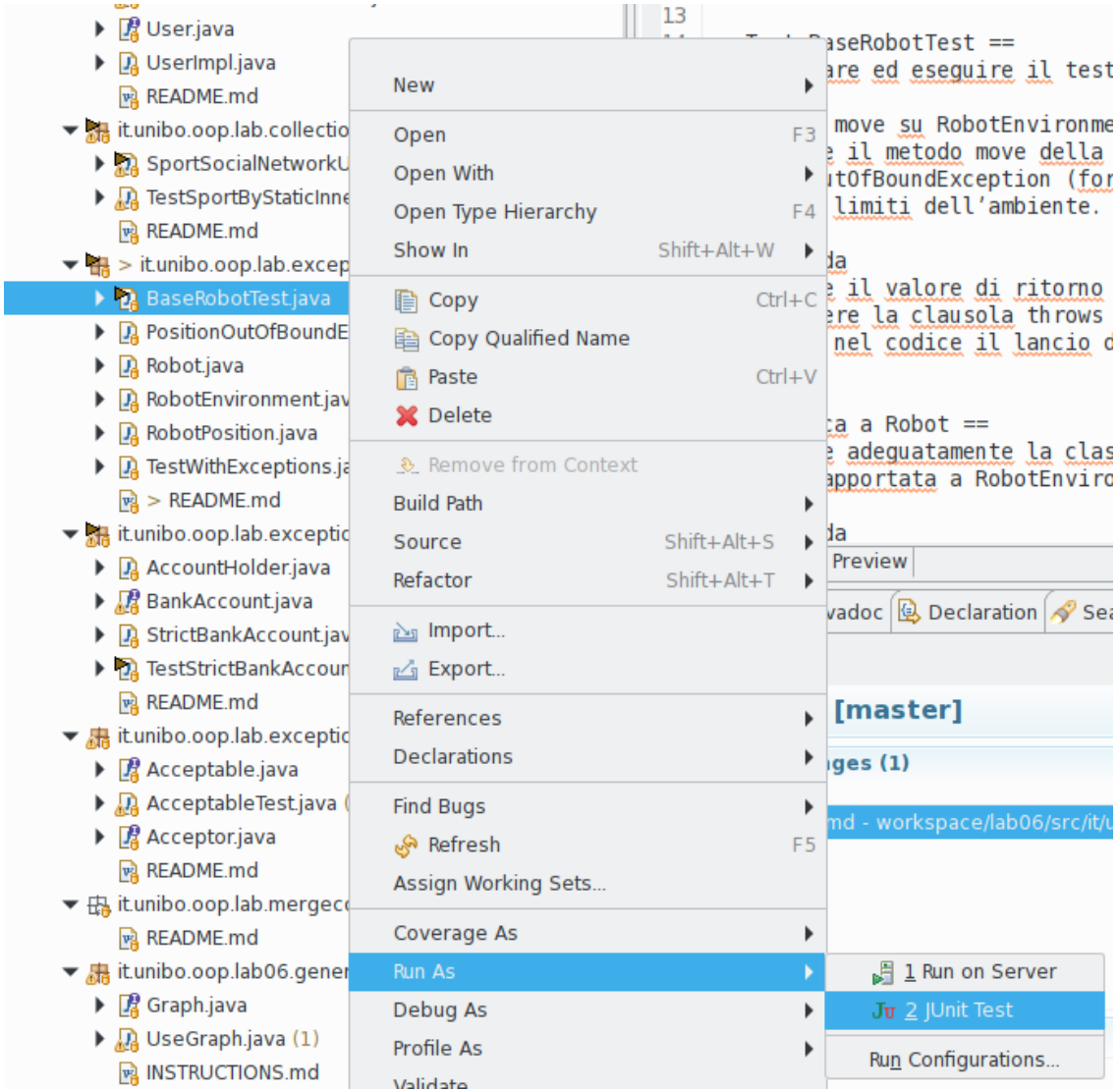
click su "Finish"

Junit in Eclipse V



La libreria appare nel classpath. Click su "Apply and Close"

Junit in Eclipse VI



Per eseguire, si usa "Run As" – "Junit Test"