MINISTERUL EDUCAȚIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA
IP COLEGIUL „IULIA HASDEU” DIN CAHUL
CATEDRA TIC

Asistență în managementul proiectelor software

Studiu de caz
Tema: ”Sisteme de control al versiunilor. GIT. GitHub”

A efectuat:	elev grupa AAW 2042	Apareci Aurica
	elev grupa AAW 2042	Babaianu Savelie
A verificat:	profesor discipline TIC	Turceac Natalia

Cahul 2023

Cuprins

Tema 1: Noțiuni generale despre Sisteme de Control al Versiunilor (SCV)

Ce este un SCV?

Un Sistem de Control al Versiunilor (SCV) este o componentă esențială a dezvoltării software, conceput pentru a urmări, gestiona și controla modificările aduse codului sursă și resurselor într-un proiect de dezvoltare software. SCV-ul oferă un cadru care permite dezvoltatorilor să înregistreze, să stocheze și să gestioneze toate versiunile de cod ale unei aplicații sau proiect, permițându-le să lucreze în mod colaborativ și să mențină istoricul modificărilor.

Explicarea conceptului de SCV și de ce este important în dezvoltarea software.

În lumea dezvoltării software, gestiunea și controlul versiunilor codului sursă reprezintă o componentă fundamentală pentru succesul și eficiența proiectelor. Acest rol esențial este îndeplinit de Sistemele de Control al Versiunilor (SCV), beneficiile căruia sunt următoarele:

Istoric și urmărire: Sistemele de Control al Versiunilor (SCV) oferă dezvoltatorilor posibilitatea de a păstra un istoric detaliat al fiecărei modificări aduse codului sursă și resurselor unui proiect software. Aceasta înseamnă că fiecare modificare este înregistrată și stocată, ceea ce facilitează urmărirea evoluției proiectului în timp. Acest istoric oferă o viziune clară asupra schimbărilor făcute și a persoanelor implicate, ceea ce este esențial pentru înțelegerea și documentarea proiectului.

Colaborare eficientă: Un alt aspect esențial al SCV-urilor este facilitarea colaborării eficiente în echipa de dezvoltare. Dezvoltatorii pot lucra simultan la proiect, fiecare pe propriul branch sau ramură, și pot integra modificările lor în mod ordonat și controlat. Acest lucru previne conflictele și ajută echipa să gestioneze dezvoltarea colaborativă. Fără un SCV, colaborarea ar fi mult mai dificilă, iar riscul de suprascriere accidentală a modificărilor celorlalți membri ai echipei ar fi crescut semnificativ.

Revenire la versiuni anterioare: Un alt beneficiu important al SCV-urilor este capacitatea de a reveni la versiuni anterioare ale codului. Dacă o modificare introduce erori sau probleme, dezvoltatorii pot utiliza SCV-ul pentru a reveni la o versiune anterioară funcțională. Aceasta oferă o modalitate sigură de a gestiona erorile și de a menține funcționalitatea proiectului în timpul dezvoltării. Revenirea la versiuni anterioare este o măsură de siguranță crucială în dezvoltarea software și reduce riscul de pierdere a muncii și de potențiale pierderi financiare.

Identificarea și soluționarea problemelor: SCV-urile facilitează identificarea rapidă și soluționarea problemelor în cod. Dacă există conflicte între modificările făcute de diferiți membri ai echipei, SCV-ul oferă instrumente pentru a le rezolva. Acest lucru contribuie la menținerea calității codului și la prevenirea introducerii de erori în proiect. De asemenea, un istoric detaliat al

modificărilor permite echipelor să investigheze și să soluționeze problemele care apar în timpul dezvoltării, îmbunătățind astfel eficiența procesului.

Împărțirea proiectului: SCV-urile permit dezvoltatorilor să lucreze pe branch-uri separate sau ramuri, ceea ce ușurează dezvoltarea de funcționalități noi fără a afecta versiunea principală a proiectului. Acest lucru este deosebit de util în proiectele mari sau complexe, unde diferite echipe sau dezvoltatori lucrează la funcționalități diferite. Împărțirea proiectului în branch-uri permite o gestionare mai eficientă și organizată a dezvoltării, precum și o izolare a schimbărilor, astfel încât acestea să nu interfereze cu partea funcțională stabilă a proiectului.

În concluzie, într-un mediu de dezvoltare modern, Sistemele de Control al Versiunilor (SCV) sunt esențiale pentru gestionarea proiectelor software. Ele oferă un cadru eficient pentru gestionarea schimbărilor, colaborare, identificarea problemelor și menținerea calității codului, contribuind semnificativ la eficiența și controlul procesului de dezvoltare. Git, ca unul dintre cele mai utilizate SCV-uri, oferă dezvoltatorilor funcționalități puternice pentru gestionarea versiunilor și este o componentă fundamentală a dezvoltării software moderne.

Tema 2: Git - Instalare și Noțiuni de bază

Terminologie și vocabular

Depozit Git (Repository): Un depozit Git este un spațiu în care Git stochează toate informațiile despre un proiect. Acesta include toate fișierele, directoarele și istoricul schimbărilor legate de proiect. De obicei, există două tipuri principale de depozite: local și remote.

Commit: Un commit reprezintă un "snapshot" sau o imagine a stării actuale a proiectului într-un anumit moment. Un commit include toate schimbările efectuate asupra fișierelor înainte de a face acel commit. Fiecare commit este identificat printr-un cod și este însoțit de o descriere care explică schimbările efectuate. Commit-urile sunt folosite pentru a urmări istoricul schimbărilor și pentru a reveni la stări anterioare ale proiectului.

Branch (Ramură): O linie separată de dezvoltare care permite lucrul asupra funcționalităților sau problemelor în izolare, fără a afecta ramura principală ("main"). Aceasta facilitează gestionarea proiectelor mai mari și colaborarea.

Clonare (Clone): Clonarea unui depozit Git înseamnă a crea o copie locală a unui depozit existent, de obicei, un depozit remote. Aceasta vă permite să lucrați pe proiect local și să obțineți o copie a întregului istoric al schimbărilor.

Pull (Tragere): Obținerea și combinarea schimbărilor dintr-o altă ramură sau depozit în ramura curentă. Acest proces vă permite să sincronizați proiectul local cu schimbările de pe un depozit remote sau să aduceți schimbări dintr-o ramură în alta.

Push (Împingere): Încărcarea schimbărilor locale într-un depozit remote pentru a le partaja cu ceilalți dezvoltatori.

Conflict de îmbinare (Merge Conflict): O situație în care două ramuri sau commit-uri conțin schimbări incompatibile în același fișier. Trebuie rezolvat manual.

Rezolvare de conflicte (Conflict Resolution): Procesul de soluționare a conflictelor de îmbinare pentru a permite finalizarea îmbinării cu succes.

.gitignore: Un fișier special în care specificați fișierele sau directoarele care nu trebuie urmărite sau incluse în depozit. Acesta este util pentru a exclude fișiere temporare, fișiere generate automat sau fișiere sensibile, astfel încât acestea să nu fie adăugate accidental în depozit.

Furcă (Fork): Crearea unei copii a unui depozit remote în contul personal pentru a lucra independent asupra proiectului. Este folosit adesea pentru a contribui la proiectele open-source sau pentru a dezvolta o versiune separată a unui proiect.

Pull Request (Cerere de tragere): O solicitare de a aduce schimbările dintr-o ramură a unui depozit forked în depozitul original. Proprietarul depozitului original poate revizui schimbările și decide dacă le acceptă sau le respinge.

CLI (Command Line Interface): Interfața de linie de comandă (CLI) pentru Git este o modalitate de a interacționa cu Git folosind comenzi textuale introduse într-o fereastră de terminal sau linie de comandă.

GUI (Graphical User Interface): Interfața grafică pentru Git este o interfață cu utilizatorul bazată pe grafică, care oferă o experiență mai ușor de utilizat decât linia de comandă.

Instalarea Git

Git, unul dintre cele mai influente și utilizate Sisteme de Control al Versiunilor (SCV), a fost creat de către Linus Torvalds, cunoscut pentru dezvoltarea kernelului Linux. În 2005, Linus a pornit proiectul Git ca răspuns la dezamăgirea sa față de SCV-urile existente la acea vreme. Git a fost proiectat cu accent pe performanță, putere și securitate, pentru a satisface nevoile de dezvoltare a proiectului Linux, care implica mii de dezvoltatori din întreaga lume. Git a revoluționat gestionarea versiunilor și colaborarea în dezvoltarea software datorită distribuției descentralizate, ușurinței de utilizare și capacității de a face față proiectelor de orice dimensiune. Astăzi, Git este folosit pe scară largă în industria IT, de la dezvoltarea open source la proiecte comerciale, și reprezintă o parte fundamentală a dezvoltării software moderne. Istoria Git ilustrează modul în care inovațiile tehnologice pot schimba radical modul în care dezvoltatorii lucrează și colaborează în lumea software.

Instalarea Git este primul pas crucial pentru a începe să lucrați cu această tehnologie, și este important să o realizați în mod corespunzător, indiferent de sistemul de operare pe care îl utilizați. Acest ghid vă va oferi instrucțiuni detaliate privind instalarea Git pe diferite platforme, inclusiv Windows, Linux și MacOS, astfel încât să puteți beneficia de accesibilitate pe mai multe sisteme de operare. Pe lângă aceasta, veți învăța cum să inițializați un depozit Git și să configurați identitatea, ceea ce vă va permite să urmăriți, să modificați și să colaborați asupra proiectelor software cu ușurință.

Cum să instalați Git pe diferite platforme (Windows, Linux, MacOS).

Git poate fi instalat pe diferite platforme, cum ar fi Windows, Linux și MacOS. Iată cum puteți instala Git pe fiecare dintre aceste sisteme de operare:

1. *Windows.*

Descărcați instalatorul Git de pe site-ul oficial Git (https://git-scm.com/download/win) și urmați instrucțiunile pentru instalare. Odată instalat, Git va fi disponibil în linia de comandă.

2. *Linux:*

Pentru distribuțiile bazate pe Debian, utilizați comanda: *`sudo apt-get install git`*

Pentru distribuțiile bazate pe Red Hat, utilizați comanda: *`sudo yum install git`*

Pentru alte distribuții, consultați documentația distribuției dvs. specifică.

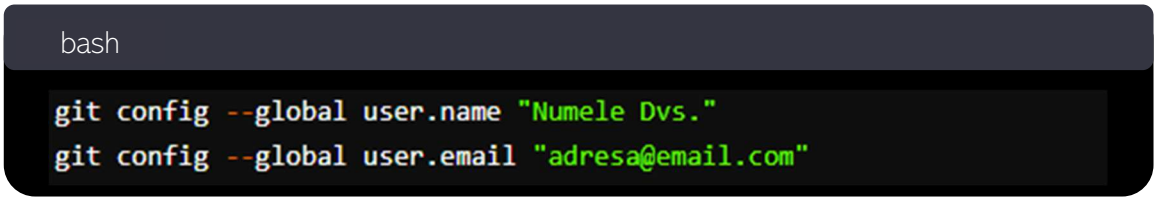
3. *MacOS:*

Pe MacOS, cel mai ușor mod de a instala Git este utilizarea Homebrew. Dacă nu aveți Homebrew, instalați-l și apoi utilizați comanda: *`brew install git`*

Configurarea Git

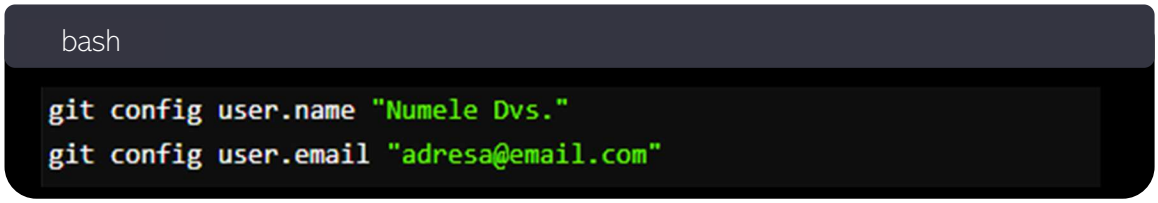
Pentru a utiliza Git, este important să configurați numele și adresa de e-mail pentru a vă identifica. Puteți configura aceste informații global sau local, în funcție de nevoile dvs. Iată cum să configurați aceste informații:

Configurare globală (valabilă pentru toate proiectele):



Figură 1 Configurare globală

Configurare locală (valabilă doar pentru proiectul curent):



Figură 2 Configurare locală

Tema 3: Inițializarea unui depozit

Comenzi utilizate *git init*

Descriere

Inițializarea unui depozit Git este primul pas în gestionarea versiunilor pentru proiectele tale. Comanda git poate fi folosită pentru a converti un proiect existent, neversionat, într-un depozit Git sau pentru a inițializa un nou depozit gol. Majoritatea celorlalte comenzi Git nu sunt disponibile în afara unui depozit inițializat, așa că aceasta este de obicei prima comandă pe care o veți rula într-un proiect nou. Prin inițializarea unui depozit git, se crează un mediu de urmărire a modificărilor proiectului. Pentru a crea un nou depozit Git, utilizezi comanda *`git init`*.

Executarea *git init* creează un un director ascuns numit *".git"* în directorul de lucru curent, care conține toate metadatele Git necesare pentru noul depozit. Aceste metadate includ subdirectoare pentru obiecte, referințe și fișiere șablon. De asemenea, este creat un fișier *HEAD* care indică la comiterea verificată în prezent.

Exemplu

```bash
mkdir new_project
cd new_project
git init
```

Figură 3 Crearea unui directoriu și inițializarea depozitului

Dacă ați rulat deja *git init* într-un director de proiect și acesta conține un subdirector *.git*, puteți rula *git init* din nou în siguranță în același director de proiect. Nu va înlocui configurația existentă.

O notă rapidă: *git init* și *git clone* pot fi ușor confundate. La un nivel înalt, ambele pot fi folosite pentru a „inițializa un nou depozit git”. Cu toate acestea, *git clone* depinde de *git init*. Comanda pentru clonare este folosită pentru a crea o copie a unui depozit existent. Intern, *git clone* apelează mai întâi *git init* pentru a crea un nou depozit. Apoi copiază datele din depozitul existent și verifică un nou set de fișiere de lucru.

3.1 Duplicarea unui depozit Git

Comenzi utilizate *git clone <URL_dep>*

Descriere

Dacă un proiect a fost deja configurat într-un depozit central, comanda *`git clone`* este folosită pentru a crea o copie locală a unui depozit Git existent. Comanda este esențială atunci când dorești să începi să lucrezi la un proiect existent sau să colaborezi cu alți dezvoltatori, deoarece îți

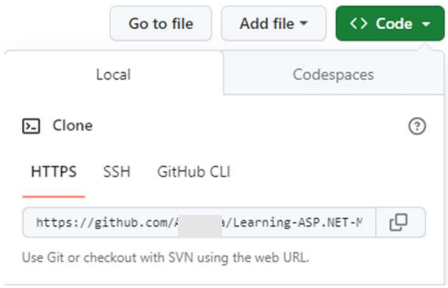
oferă o copie locală a întregului depozit și a istoricului său, permițându-ți să lucrezi la proiect în mod eficient și să gestionezi versiunile folosind Git. Odată ce un dezvoltator a obținut o copie de lucru, toate operațiunile de control al versiunilor și colaborările sunt gestionate prin intermediul depozitului local.

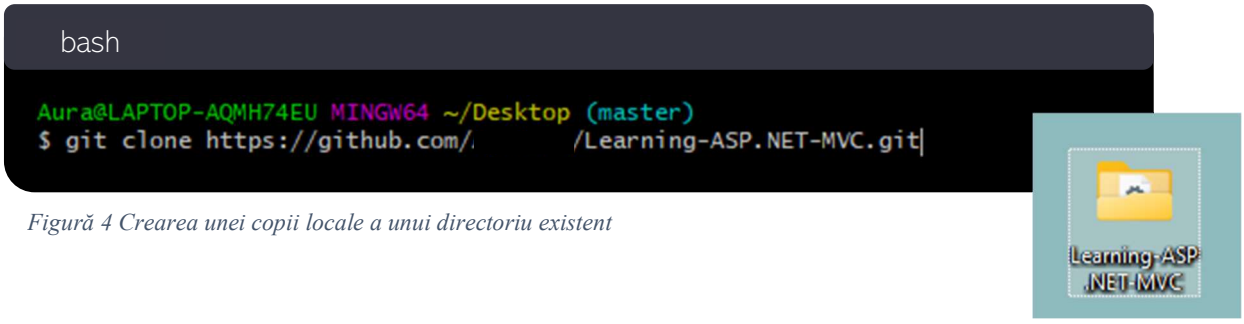
Această comandă face următoarele:

1. **Descărcare:** Clonează un depozit Git de pe un server remote (cum ar fi GitHub, GitLab, sau un alt server Git) sau din alt repositoriu local și aduce toate datele și istoricul asociat cu acel depozit pe computerul tău.
2. **Crearea unei copii de lucru:** Creează o copie a conținutului depozitului într-un director local, astfel încât poți lucra cu fișierele și conținutul din depozit pe computerul tău.
3. **Setarea unui remote:** Adaugă o referință la depozitul remote din care ai clonat, astfel încât să poți să trimiti și să primești modificări între depozitul local și cel remote folosind comenzile git push și git pull.
4. **Inițializează ramura implicită:** Creează o ramură locală numită de obicei "master" sau "main" care este o copie a ramurii implicite din depozitul remote.

Exemplu

<URL_dep> reprezintă adresa URL a depozitului Git pe care dorești să-l clonezi. Acesta poate fi un URL HTTP/HTTPS sau un URL SSH, și poate fi de pe GitHub, GitLab, sau oricare alt server Git.





Figură 4 Crearea unei copii locale a unui directoriu existent

Tema 4 : Adăugarea și Comiterea Fișierelor

Comenzi utilizate | *git add, git commit, git log, git reset, git revert*

Descriere

Procesul de adăugare și comitere a fișierelor într-un depozit Git presupune două etape esențiale. În primul rând, utilizatorul utilizează comanda *`git add`* pentru a selecta și pregăti fișierele sau modificările dorite, aducându-le în zona de pregătire (staging area) pentru următorul commit. După ce modificările sunt adăugate în staging area, utilizatorul utilizează comanda *`git commit`* pentru a crea un nou commit care înregistrează starea actuală a depozitului Git, însoțit de un mesaj de commit descriptiv. Acest mesaj de commit explică în mod clar ce modificări au fost efectuate în cadrul commit-ului respectiv, ceea ce ajută la urmărirea și gestionarea versiunilor proiectului.

Comanda *`git add`* este utilizată pentru a adăuga sau pregăti fișierele sau modificările pentru a fi incluse în următorul commit. Aceasta transferă modificările din directorul de lucru (working directory) în staging area (zona de pregătire), ceea ce înseamnă că aceste modificări sunt gata să fie comise în istoricul depozitului Git. Poate fi folosită pentru adăugarea fișierelor noi sau a modificărilor la fișierele existente.

Comanda *`git commit`* este folosită pentru a crea un nou commit care înregistrează starea curentă a staging area în istoricul depozitului Git. La fiecare commit, se adaugă un mesaj de commit care descrie modificările efectuate în cadrul acelui commit. Commit-urile sunt puncte de referință în istoricul depozitului și sunt utilizate pentru a urmări și a reveni la stări anterioare ale proiectului.

Exemplu

Metoda Command Line: Utilizarea liniei de comandă pentru adăugarea și comiterea fișierelor într-un depozit Git presupune utilizarea comenzilor Git direct. Iată cum poți face acest lucru:

1. Deschide terminalul și navighează în directorul depozitului Git folosind comanda *`cd`*.
2. Pentru a adăuga fișierele modificate în staging area (pregătirea pentru commit), folosește comanda *`git add`* pentru fiecare fișier sau director individual sau folosește *`git add .`* pentru a adăuga toate fișierele din directorul curent.

```bash
Aura@LAPTOP-AQMH74EU MINGW64 ~/Desktop/Learning-ASP.NET-MVC (main)
$ git add new_folder
```

Figură 5 Adăugarea folderului în staging area

```bash
Aura@LAPTOP-AQMH74EU MINGW64 ~/Desktop/Learning-ASP.NET-MVC (main)
$ git add .
```

Figură 5.1 Adăugarea tuturor modificărilor în staging area

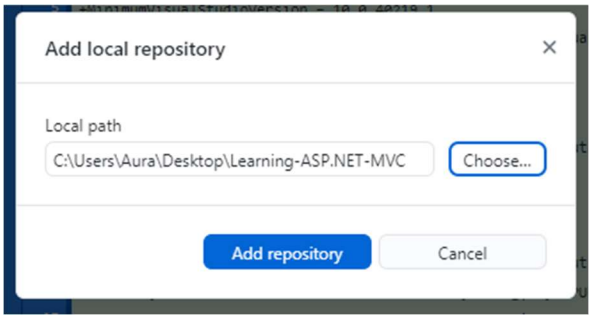
3. Pentru a face un commit cu modificările adăugate, utilizează comanda `*git commit -m*` urmată de un mesaj de commit care descrie modificările efectuate.

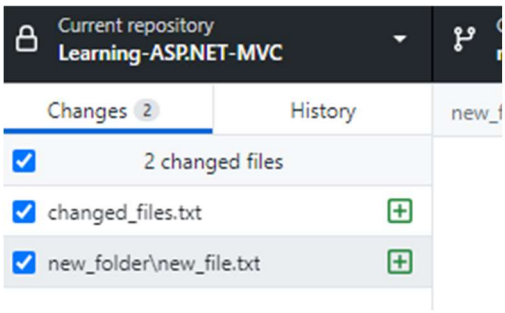
```bash
Aura@LAPTOP-AQMH74EU MINGW64 ~/Desktop/Learning-ASP.NET-MVC (main)
$ git commit -m "Added new folder"
```

Figură 6 Crearea unui nou commit

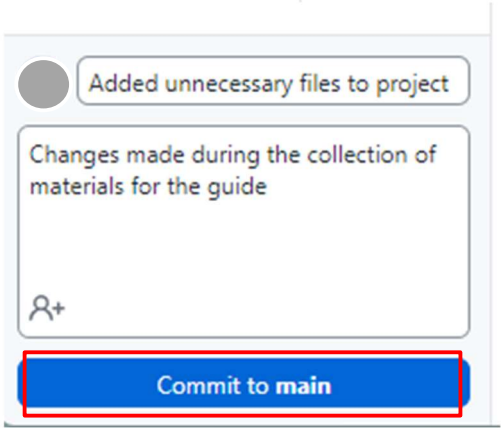
Metoda Grafică: Una dintre cele mai populare interfețe grafice pentru a adăuga și comite fișiere într-un depozit Git este folosirea unei aplicații de gestionare a versiunilor care oferă suport pentru Git, cum ar fi GitHub Desktop. Iată cum poți efectua adăugarea și comiterea fișierelor folosind GitHub Desktop:

1. Deschide GitHub Desktop și selectează depozitul tău Git.
2. În interfața grafică, vei vedea o listă a fișierelor modificate sau necomise.



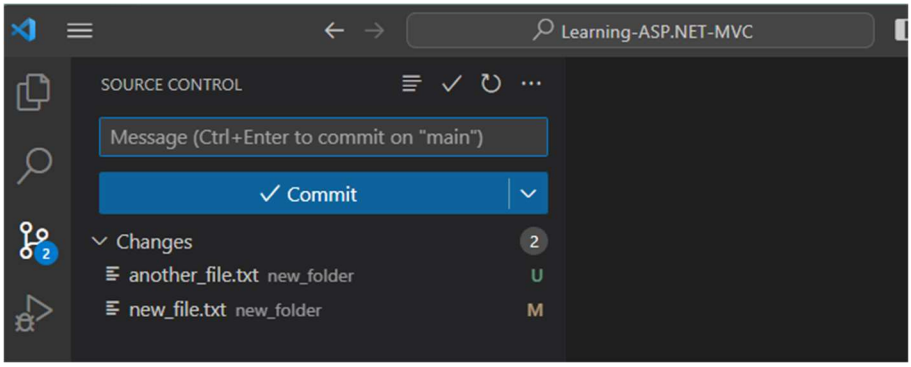


3. Selectează fișierele pe care dorești să le adaugi și comiți într-un commit. De obicei, aceste fișiere vor fi trecute din starea "Uncommitted Changes" în starea "Changes to be committed"
4. Introdu un mesaj de commit care descrie modificările
5. Apasă butonul "Commit" pentru a face commit la fișierele selectate.

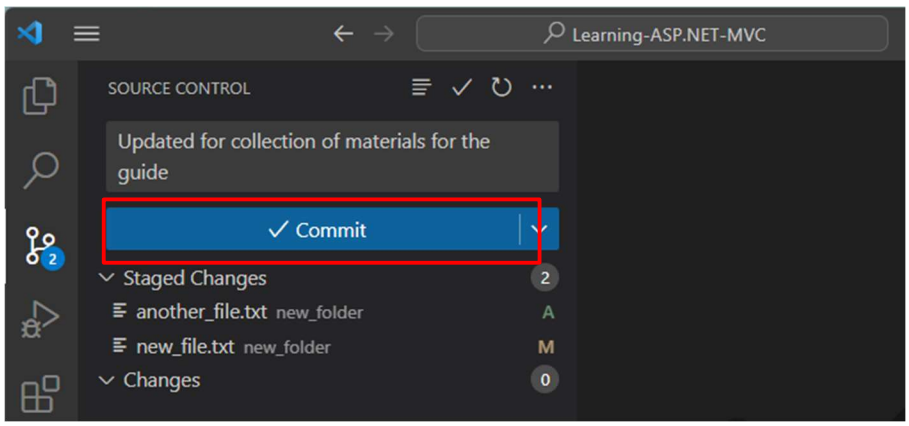


Metoda VS Code: Folosind Visual Studio Code (VS Code) poți să adaugi și să comiți modificări într-un depozit Git folosind o interfață grafică integrată care simplifică gestionarea versiunilor. Iată cum poți să adaugi și să comiți modificări folosind VS Code:

1. Deschide Proiectul în VS Code.
2. În partea stânga a interfeței VS Code, vei vedea o secțiune numită "Source Control". Aici, poți vizualiza toate fișierele care au fost modificate sau adăugate în depozitul Git.



3. Adăugarea Fișierelor Modificate: Pentru a adăuga fișierele modificate în staging area, fă clic pe fiecare fișier în secțiunea "Source Control" sau selectează-le folosind comanda "Stage All Changes" din meniu. Acest lucru va transfera fișierele în staging area.
4. Comiterea Modificărilor: După ce ai adăugat modificările în staging area, poți comite aceste modificări. Fă clic pe butonul "Checkmark" (✓) din partea de jos a secțiunii "Source Control" sau utilizează comanda "Commit" din meniu. Acest lucru va deschide o fereastră unde poți introduce un mesaj de commit pentru a descrie modificările făcute. Introdu mesajul de commit și apasă "Checkmark" (✓) pentru a finaliza comiterea.



Acestea sunt câteva modalități principale de adăugare și comitere a fișierelor într-un depozit Git, una utilizând interfața grafică și cealaltă folosind linia de comandă. Toate metodele sunt eficiente și depind de preferințele tale personale sau de contextul în care lucrezi.

Descriere

Comenzile *git log, git reset* și *git revert* sunt instrumente esențiale în gestionarea istoricului și a modificărilor într-un depozit Git.

`git log` este folosit pentru a afișa istoricul commit-urilor dintr-un depozit GIT, în ordine cronologică inversă, cele mai recente fiind afișate primele. Aceasta arată informații precum autorul, data și mesajul fiecărui commit.

`git reset` este folosit pentru a modifica starea staging și istoricul commit-urilor.

`git revert` este folosită pentru a crea un nou commit care anulează efectele unui commit anterior. În loc să șterge istoricul, git revert adaugă un nou commit care face "reversul" modificărilor introduse de commit-ul specificat.

Aceste trei comenzi sunt utile pentru gestionarea istoricului și modificărilor într-un depozit Git, fie pentru a afișa istoricul, a modifica starea staging și istoricul, sau pentru a anula modificările introduse de commit-uri anterioare. Este important să fie folosite cu atenție, mai ales *`git reset --hard`*, deoarece pot cauza pierderi ireversibile de date.

Exemplu

```bash
Aura@LAPTOP-AQMH74EU MINGW64 ~/Desktop/Learning-ASP.NET-MVC (main)
$ git log
commit 0497c4daa05885cf7946726125112782517 2d56f (HEAD -> main)
Author: 
Date:   Sat Oct 28 01:47:33 2023 +0300

    Updated for collection of materials for the guide

commit 363ae153650b503781de83ad0629e2a2c4c20ccd
Author: 
Date:   Sat Oct 28 01:36:28 2023 +0300

    Added unnecessary files to project

    Changes made during the collection of materials for the guide
```

Figură 9 Istoricul commit-urilor

```bash
git reset --soft name_commit
```

Figură 10.1 `git reset –soft`

```bash
git reset name_commit
```

Figură 10.2 `git reset`

```bash
git reset --hard name_commit
```

Figură 10.3 `git reset --hard`

`git reset –soft`: resetează doar starea staging, păstrând modificările locale necomise în directorul.

`git reset –mixed` sau *`git reset`:* resetează starea staging și anulează commit-urile, dar păstrează modificările locale în directorul de lucru.

`git reset –hard`: resetează complet starea staging și istoricul, ștergând și modificările locale necomise. Această comandă trebuie folosită cu mare atenție, deoarece poate cauza pierderea ireversibilă a datelor.

`name-commit` se referă la commit-ul anterior celui curent.

```bash
git revert <SHA_commit>
```

Figură 11 `git revert`

`SHA_commit` reprezintă codul unic al commit-ului pe care dorești să-l revoci. Acesta va crea un nou commit care elimină modificările introduse de commit-ul specificat, menținând totuși istoricul neschimbat.

Tema 5: Crearea și schimbarea branch-urilor

Comenzi utilizate *git branch, git checkout*

Descriere

Branch-urile sunt linii de dezvoltare paralele care permit dezvoltatorilor să lucreze la diferite caracteristici ale proiectului în paralel, fără să interfere unul cu celălalt. De asemenea, ele sunt utile pentru organizarea și gestionarea dezvoltării într-un mod structurat prin separarea funcționalităților. Crearea unui branch nou se face cu comanda *`git branch nume_branch`*, unde *`nume_branch`* este numele noului branch. Branch-ul nou creat va fi o copie a branch-ului curent sau a branch-ului specificat în acel moment. Pentru a te schimba la un branch existent, poți utiliza comanda *` git checkout nume_branch`*.

Exemplu

Presupunând că depozitul în care lucrați conține ramuri preexistente, puteți comuta între aceste ramuri folosind *git checkout*. Pentru a afla ce ramuri sunt disponibile și care este numele actual al ramurilor, executați *git branch*.

```bash
git branch
main
another_branch
feature_inprogress_branch
git checkout feature_inprogress_branch
```

Exemplul de mai sus demonstrează cum să vizualizați o listă de ramuri disponibile executând comanda *git branch* și să comutați la o ramură specificată, în acest caz, *feature_inprogress_branch.*

```bash
git branch new_branch
```

Figură 12.1 `git branch`

```bash
git checkout new_branch
```

Figură 12.2 `git checkout`

Comanda *git branch* poate fi folosită pentru a crea o nouă ramură. Când doriți să începeți o nouă funcție, creați o nouă ramificare folosind *git branch new_branch.* Odată creată, puteți folosi *git*

checkout new_branch pentru a comuta la acea ramură. În plus, *git checkout* comanda acceptă un argument care acționează ca o metodă convenabilă care va crea noua ramură și va trece imediat la ea. Puteți lucra la mai multe funcții într-un singur depozit, comutând între ele cu *-b git checkout.*

```bash
Aura@LAPTOP-AQMH74EU MINGW64 ~/Desktop/Learning-ASP.NET-MVC (main)
$ git checkout -b new-branch
Switched to a new branch 'new-branch'

Aura@LAPTOP-AQMH74EU MINGW64 ~/Desktop/Learning-ASP.NET-MVC (new-branch)
$
```

Exemplul de mai sus creează și verifică simultan . Opțiunea este un indicator de confort care îi spune lui Git să ruleze *git branch new_branch* înainte de a rula *git checkout new_branch*.

```bash
git branch -d new-branch
```

Rulează comanda *`git branch -d <branch>`,* unde *`<branch>`* reprezintă numele branch-ului pe care dorești să-l ștergi. Dacă branch-ul poate fi șters în siguranță (adică nu conține modificări necomise), Git va șterge branch-ul și îți va afișa un mesaj de confirmare.

Când colaborați cu o echipă, este obișnuit să utilizați depozite de la distanță. Aceste depozite pot fi găzduite și partajate sau pot fi copia locală a altui coleg. Fiecare depozit la distanță va conține propriul său set de ramuri. Pentru a verifica o ramură la distanță, trebuie mai întâi să preluați conținutul ramurii.

```bash
git fetch --all
```

În versiunile moderne de Git, puteți verifica apoi ramura la distanță ca o sucursală locală.

```bash
git checkout <remotebranch>
```

Versiunile mai vechi de Git necesită crearea unei noi ramuri bazate pe remote.

```bash
git checkout -b <remotebranch> origin/<remotebranch>
```

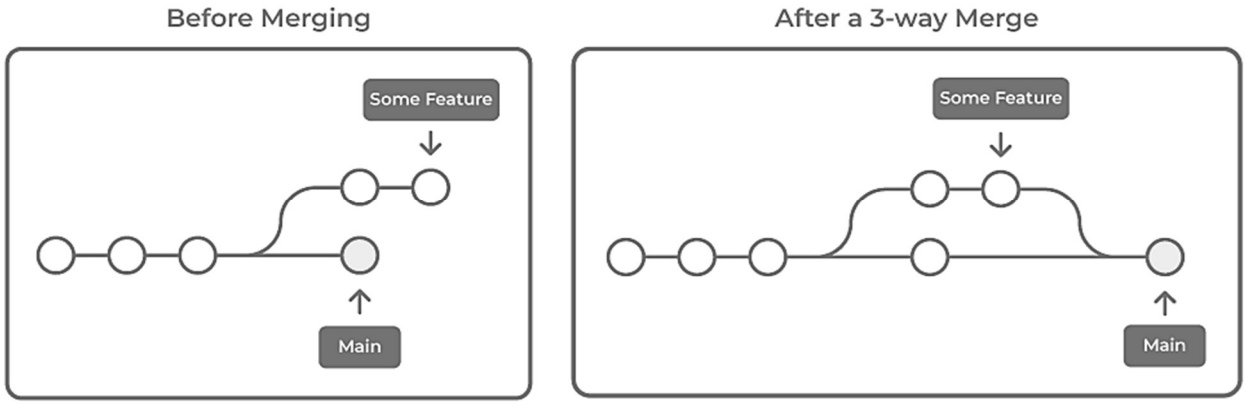
Tema 6: Unirea branch-urilor (merge)

Comenzi utilizate | *git merge* |

Descriere

Unirea branch-urilor (merge) în Git este procesul prin care modificările dintr-un branch sunt integrate în alt branch. Acest proces este utilizat pentru a combina dezvoltarea realizată pe branch-uri separate, permițând astfel adăugarea noilor funcționalități sau corectarea problemelor în branch-urile principale sau în alte branch-uri. Comanda utilizată pentru a realiza un merge este *`git merge`,* urmată de numele branch-ului din care dorești să faci merge. Merge-ul poate fi făcut într-un branch curent sau în alt branch specificat.

În scenariul de combinare a două ramuri, *git merge* ia doi pointeri de comitere, de obicei vârfurile de ramuri, și va găsi o comitare de bază comună între ei. Odată ce Git găsește o comitere de bază comună, va crea o nouă *„comitare de îmbinare”* care combină modificările fiecărei secvențe de comitere de îmbinare din coadă.



Să presupunem că avem o nouă caracteristică de ramură care se bazează pe ramura *main*. Acum dorim să îmbinăm această ramura *Some Feature* în *main*. Invocarea acestei comenzi va îmbina *Some Feature* specificată în ramura curentă, vom presupune *main*. Git va determina automat algoritmul de îmbinare. Merge commit-urile sunt unice față de alte commit-uri prin faptul că au două commit-uri părinte. Când creați o comitere de îmbinare, Git va încerca să îmbine automat istoricele separate pentru dvs. Dacă Git întâlnește o bucată de date care este modificată în ambele istorii, nu va putea să le combine automat. Acest scenariu este un conflict de control al versiunilor și Git va avea nevoie de intervenția utilizatorului pentru a continua.

Exemplu

Înainte de a efectua o îmbinare, trebuie să urmați câțiva pași de pregătire pentru a vă asigura că îmbinarea se desfășoară fără probleme.

Executați *git status* pentru a vă asigura că *HEAD* indică către ramura corectă de îmbinare-recepție. Dacă este necesar, executați *git checkout* pentru a comuta la ramura de primire. În cazul nostru vom executa *git checkout main*. Asigurați-vă că ramura care primește și ramura care fuzionează sunt actualizate cu cele mai recente modificări de la distanță. Executați *git fetch* pentru a extrage cele mai recente comiteri de la distanță. Odată ce preluarea este finalizată, asigurați-vă că ramura *main* are cele mai recente actualizări prin executare *git pull*. Odată ce pașii discutați anterior de *„pregătire pentru fuzionare”* au fost parcurși, o îmbinare poate fi inițiată prin executarea *git merge* unde este indicat numele sucursalei care va fi fuzionată în ramura destinatară.

```bash
git merge new-branch
```

Figură 13 `git merge`

<*nume_branch*> reprezintă branch-ul din care dorești să faci merge în branch-ul curent.

```bash
# Start a new feature
git checkout -b new-feature main
# Edit some files
git add <file>
git commit -m "Start a feature"
# Edit some files
git add <file>
git commit -m "Finish a feature"
# Merge in the new-feature branch
git checkout main
git merge new-feature
git branch -d new-feature
```

Acesta este un flux de lucru obișnuit pentru ramurile tematice de scurtă durată, care sunt folosite mai mult ca o dezvoltare izolată decât un instrument organizațional pentru funcții cu funcționare mai lungă. Aceasta comandă efectuează următoarele:

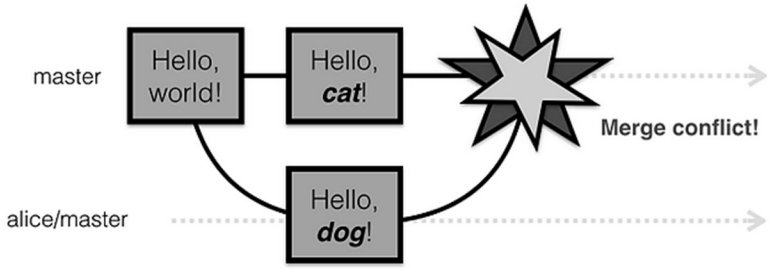
1. Identifică commit-ul comun cel mai recent dintre branch-ul curent și branch-ul specificat.
2. Creează un nou commit care unifică schimbările din branch-ul specificat în branch-ul curent.
3. Actualizează referința branch-ului curent pentru a reflecta noul commit de merge.

În esență, git merge încorporează schimbările dintr-un branch în altul, astfel încât să poți lucra cu cea mai recentă versiune a codului. Dacă nu există conflicte, merge-ul se va efectua fără probleme. Cu toate acestea, dacă există conflicte între branch-uri (adică modificări conflictuale în aceleași zone ale aceluiași fișier), va trebui să rezolvi aceste conflicte manual.

Tema 7: Rezolvarea conflictelor de unire (merge conflicts)

Descriere

Conflictul de unire (merge conflict) apare atunci când două branch-uri sau două versiuni diferite ale aceluiași fișier au modificări incompatibile. Acest lucru se întâmplă în timpul procesului de unire (merge) atunci când Git nu poate decide automat cum să combine aceste modificări. Pentru a rezolva conflictul de unire, dezvoltatorul trebuie să intervină manual și să decidă cum să combine aceste modificări incompatibile. Procesul implică analiza fișierelor cu conflicte, selectarea modificărilor corecte și apoi realizarea unui nou commit pentru a finaliza unirea.



O îmbinare poate intra într-o stare conflictuală în două puncte separate. La pornirea și în timpul unui proces de îmbinare.

1. **Git nu reușește să înceapă îmbinarea**

O îmbinare nu va începe când Git vede că există modificări fie în directorul de lucru, fie în zona de pregătire a proiectului curent. Git nu reușește să pornească îmbinarea, deoarece aceste modificări în așteptare ar putea fi scrise de commit-urile în care sunt îmbinate. Când se întâmplă acest lucru, nu este din cauza unor conflicte cu cele ale altor dezvoltatori, ci din cauza unor conflicte cu modificările locale în așteptare. Statul local va trebui să fie stabilizat folosind *git stash*, sau *git checkout*. Un eșec de îmbinare la pornire va afișa următorul mesaj de eroare: *git commitgit reset*

```
error: Entry '<fileName>' not uptodate. Cannot merge.
            (Changes in working directory)
```

2. **Git eșuează în timpul îmbinării**

Un eșec ÎN TIMPUL unei îmbinări indică un conflict între ramura locală curentă și ramura care este fuzionată. Aceasta indică un conflict cu un alt cod de dezvoltator. Git va face tot posibilul pentru a îmbina fișierele, dar vă va lăsa lucrurile pe care să le rezolvați manual în fișierele aflate în conflict. Un eșec la mijloc de îmbinare va afișa următorul mesaj de eroare:

```
error: Entry '<fileName>' would be overwritten by merge.
          Cannot merge. (Changes in staging area)
```

Exemplu

Ne imaginăm că două branch-uri separate, *"branch-A"* și *"branch-B",* au avut modificări în același fișier *"fisier.txt"*. Când încerci să faci merge a branch-ului *"branch-A"* în *"branch-B"* sau invers, Git detectează un conflict de unire în *"fisier.txt"*. În acest moment, fișierul *"fisier.txt"* conține marcaje speciale care indică zonele conflictuale, cum ar fi *"<<< HEAD"* pentru branch-ul curent *("branch-B")* și "<<< *branch-A"* pentru branch-ul *"branch-A"*.

Dezvoltatorul trebuie să editeze manual *"fisier.txt"* și să aleagă care modificări să păstreze și care să elimine. Aceste decizii trebuie luate în funcție de cerințele proiectului.

După ce modificările sunt făcute și conflictul este rezolvat, se adaugă fișierul la staging area cu *`git add fisier.txt`* și se face commit cu un mesaj care indică rezolvarea conflictului. Conflictul de unire a fost rezolvat cu succes și procesul de merge poate continua.

Rezolvarea conflictelor de unire este o parte importantă a gestionării versiunilor cu Git și necesită atenție și comunicare între membrii echipei pentru a asigura că modificările sunt integrate corect.

Crearea unui conflict de fuziune

Pentru a vă familiariza cu adevărat cu conflictele de îmbinare, secțiunea următoare va simula un conflict pentru a-l examina și rezolva ulterior. Exemplul va folosi linia de comandă Git.

```bash
mkdir git-merge-test
cd git-merge-test
git init .
echo "this is some content to mess with" > merge.txt
git add merge.txt
git commit -am"we are commiting the inital content"
   [main (root-commit) d48e74c] we are commiting the inital content
   1 file changed, 1 insertion(+)
   create mode 100644 merge.txt
```

Acest exemplu de cod execută o secvență de comenzi care realizează următoarele.

1. Crează un director nou numit *git-merge-test*, modifică acest director și îl inițializă ca un nou depozit Git.
2. Crează un fișier text nou *merge.txt* cu conținut în el.
3. Adaugă *merge.txt* la repositoriu și îl comite.

Acum avem un nou depozit cu o ramură *main* și un fișier *merge.txt* cu conținut în el. În continuare, vom crea o nouă ramură pe care să o folosim ca îmbinare în conflict.

```bash
git checkout -b new_branch_to_merge_later
echo "totally different content to merge later" > merge.txt
git commit -am"edited the content of merge.txt to cause a conflict"
   [new_branch_to_merge_later 6282319] edited the content of merge.txt to cause a conflict
   1 file changed, 1 insertion(+), 1 deletion(-)
```

Secvența de comandă care procedează realizează următoarele:

1. Crează o nouă ramură numită *new_branch_to_merge_later.*
2. Suprascrie conținutul în *merge.txt.* Comite noul conținut.

Noua ramură: *new_branch_to_merge_later* crează un commit care anulează conținutul *merge.txt*

```bash
git checkout main
Switched to branch 'main'
echo "content to append" >> merge.txt
git commit -am"appended content to merge.txt"
  [main 24fbe3c] appended content to merge.tx
  1 file changed, 1 insertion(+)
```

Acest lanț de comenzi verifică ramura *main*, adaugă conținut la *merge.txt*, și îl commite. Acest lucru pune acum repositoriul nostru într-o stare în care avem 2 noi comiteri. Unul în ramura *main* și unul în ramura *new_branch_to_merge_later*. Utilizând *git merge* observăm că Git ne informează despre apariția unui conflict.

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

După cum am experimentat din exemplul de mai sus, Git va produce o ieșire descriptivă care să ne arate că a avut loc un CONFLICT. Putem obține mai multe informații prin rularea comenzii *git status*. Ieșirea de la *git status* indică faptul că există căi necombinate din cauza unui conflict. Fișierul merge.txt apare acum într-o stare modificată.

```bash
git status
On branch main
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified:   merge.txt
```

Cel mai direct mod de a rezolva un conflict de îmbinare este editarea fișierului aflat în conflict. Deschideți *merge.txt* fișierul în editorul dvs. preferat. Pentru exemplul nostru, să eliminăm pur și simplu toți divizorii de conflict. Conținutul modificat *merge.txt* ar trebui să arate astfel:

```
this is some content to mess with
content to append
totally different content to merge later
```

Odată ce fișierul a fost editat, utilizați *git add merge.txt* pentru a pune în scenă noul conținut îmbinat. Pentru a finaliza îmbinarea, creați un nou commit executând:

```bash
git commit -m "merged and resolved the conflict in merge.txt"
```

Git va vedea că conflictul a fost rezolvat și creează o nouă comitere de îmbinare pentru a finaliza fuziunea.

Comenzi care pot ajuta la rezolvarea conflictelor

1. **Instrumente generale**

```bash
git status
```

Comanda de stare este utilizată frecvent atunci când se lucrează cu Git și în timpul unei îmbinări va ajuta la identificarea fișierelor aflate în conflict.

```bash
git log --merge
```

Trecerea *--merge* argumentului la comandă va produce un jurnal cu o listă de comiteri care sunt în conflict între ramurile care fuzionează *.git log*

```bash
git diff
```

diff ajută la găsirea diferențelor între stările unui depozit/fișiere. Acest lucru este util în prezicerea și prevenirea conflictelor de îmbinare.

2. **Instrumente pentru cazul în care git nu reușește să înceapă o combinare**

```bash
git checkout
```

checkout poate fi folosit pentru a anula modificările aduse fișierelor sau pentru a schimba ramurile

```bash
git reset --mixed
```

reset poate fi folosit pentru a anula modificările aduse directorului de lucru și zonei de pregătire.

3. Instrumente pentru rezolvarea conflictelor în timpul unei îmbinări

```bash
git merge --abort
```

Executarea *git merge* cu *–abort* opțiunea va ieși din procesul de îmbinare și va întoarce ramura la starea de dinainte de începerea fuziunii.

```bash
git reset
```

Git reset poate fi folosit în timpul unui conflict de îmbinare pentru a reseta fișierele aflate în conflict la o stare bună.

Tema 8: Refacerea istoricului (Git history rewrite)

Comenzi utilizate | *git rebase, git commit –amend*

Descriere

Refacerea istoricului (Git History Rewrite) se referă la procesul de modificare a istoricului commit-urilor într-un depozit Git. Acest proces poate include schimbarea sau eliminarea commit-urilor existente, restrângerea sau extinderea istoricului, și alte manipulări ale istoricului commit-urilor pentru a satisface cerințele sau pentru a face istoricul mai curat și mai organizat.

Comanda *`git commit --amend`* este folosită pentru a face modificări suplimentare sau pentru a adăuga schimbări la commit-ul anterior. Aceasta este utilă atunci când dorești să adaugi sau să corectezi ceva în cel mai recent commit fără a crea un commit nou.

1. După ce ai efectuat modificările dorite în fișiere sau ai adăugat fișiere noi în staging area, rulează comanda *`git commit --amend`*. Această comandă va deschide editorul de text implicit, care afișează o listă a commit-urilor, permițându-ți să editezi mesajul de commit sau să adaugi modificări suplimentare.

   Fiecare commit este listat cu un prefix care indică acțiunile pe care le poți efectua:

   pick: poate fi folosit pentru a păstra commit-ul neschimbat.

   reword: poți modifica mesajul de commit. După selectarea acestei opțiuni, vei putea să editezi mesajul de commit în modul interactiv.

   edit: poți modifica commit-ul. După selectarea acestei opțiuni, rebase-ul se va opri înainte de commit-ul respectiv, permițându-ți să faci modificări la fișierele din acel commit.

   squash sau fixup: aceste opțiuni permit combinarea unui commit cu commit-ul precedent. Folosind squash, commit-ul curent va fi combinat cu commit-ul precedent, iar mesajul acestuia va fi adăugat la mesajul precedent. Cu fixup, commit-ul curent va fi combinat cu commit-ul precedent, dar mesajul său va fi ignorat.

   drop: acesta este folosit pentru a elimina complet un commit.
2. În editorul de text, poți să faci orice modificări necesare, inclusiv schimbarea mesajului de commit sau adăugarea altor modificări.
3. Salvează și închide editorul de text.
4. Comanda *`git commit --amend`* va aplica aceste modificări asupra commit-ului anterior, creând astfel un nou commit care îl înlocuiește pe cel anterior.

Este important să știi că utilizarea *`git commit --amend`* modifică istoricul și face ca commit-ul anterior să fie înlocuit. Din acest motiv, ar trebui să fie folosită cu precauție, mai ales în cazul în care

ai deja commit-uri în istoricul depozitului tău sau dacă depozitul tău este partajat cu alți dezvoltatori. Această comandă este mai potrivită pentru ajustările rapide ale ultimului commit.

Comanda *`git rebase`* este folosită în Git pentru a restrânge sau reorganiza istoricul commit-urilor. Acest proces implică mutarea sau regăzduirea commit-urilor pe o altă ramură sau pe o altă bază.

Există două scenarii principale în care *`git rebase`* este adesea folosit:

1. Rebasing pentru a integra modificări dintr-un branch în altul: poți face rebase pentru a integra schimbările dintr-un branch într-un alt branch într-un mod liniștit și organizat.
2. Rebasing pentru a curăța istoricul commit-urilor: poți face rebase pentru a elimina commit-uri inutile sau pentru a combina commit-uri mici într-unul singur pentru a menține istoricul mai clar și mai ușor de citit.

Exemplu

```bash
git commit --amend
```

Figură 14 `git commit --amend`

```bash
git rebase bază
```

Figură 15 `git rebase`

Tema 9: Lucrul cu depozitele remote

Comenzi utilizate | *git remote, git push, git pull*

Descriere

Lucrul cu depozitele remote se referă la interacțiunea cu depozite Git aflate pe servere sau în cloud, care pot fi utilizate pentru colaborare sau pentru a gestiona copii ale proiectelor tale. Comenzile menționate sunt folosite pentru a efectua diferite operațiuni cu depozitele remote.

Comanda *git remote* este o parte a sistemului mai larg care este responsabilă pentru sincronizarea modificărilor. Înregistrările înregistrate prin comanda *git remote* sunt utilizate împreună cu comenzile *git fetch, git push*, și *git pull.*

git remote -v afișează lista depozitelor remote legate de depozitul tău local.

git remote add <nume> <url> adaugă un nou depozit remote la depozitul tău local și îi atribuie un nume (de obicei, "origin" este folosit pentru depozitul principal).

Comanda *git fetch* descarcă comiteri, fișiere și referințe dintr-un depozit de la distanță în depozitul dvs. local. Preluarea este ceea ce faci atunci când vrei să vezi la ce au lucrat toți ceilalți. Acesta va descărca conținutul de la distanță, dar nu va actualiza starea de lucru a depozitului dvs. local, lăsând intactă munca dvs. curentă. *git pull* este alternativa mai agresivă; va descărca conținutul de la distanță pentru ramura locală activă și va executa imediat git mergepentru a crea un commit de îmbinare pentru noul conținut la distanță. Dacă aveți modificări în așteptare, acest lucru va provoca conflicte și va demara fluxul de soluționare a conflictelor de îmbinare.

Comanda *git push* este utilizată pentru a încărca conținutul depozitului local într-un depozit la distanță. Împingeți ramura specificată împreună cu toate commit-urile și obiectele interne necesare. Aceasta creează o ramură locală în depozitul de destinație. Pentru a vă împiedica să suprascrieți commit-urile, Git nu vă va lăsa să împingeți atunci când are ca rezultat o îmbinare fără redirecționare rapidă în depozitul de destinație.

Comanda *git pull* este folosită pentru a prelua și descărca conținut dintr-un depozit de la distanță și pentru a actualiza imediat depozitul local pentru a se potrivi cu acel conținut. Îmbinarea modificărilor de la distanță în depozitul dvs. local este o sarcină comună în fluxurile de lucru de colaborare bazate pe Git. Comanda *git pull* este de fapt o combinație a altor două comenzi, *git fetch* urmată de *git merge*. În prima etapă de operare *git pull* se va executa un *git fetch* scoped către ramura locală către care este indicată *HEAD*. Odată ce conținutul este descărcat, git pull va intra într-un flux de lucru de îmbinare. O nouă comitere de îmbinare va fi creată și *HEAD* actualizată pentru a indica noul commit.

Exemplu

```bash
git push <remote> <branch>
```

git push <*remote*> <*branch*> încarcă (trimite) modificările din branch-ul tău local în depozitul remote specificat. Acesta este utilizat pentru a partaja modificările tale cu ceilalți colaboratori sau pentru a actualiza depozitul remote.

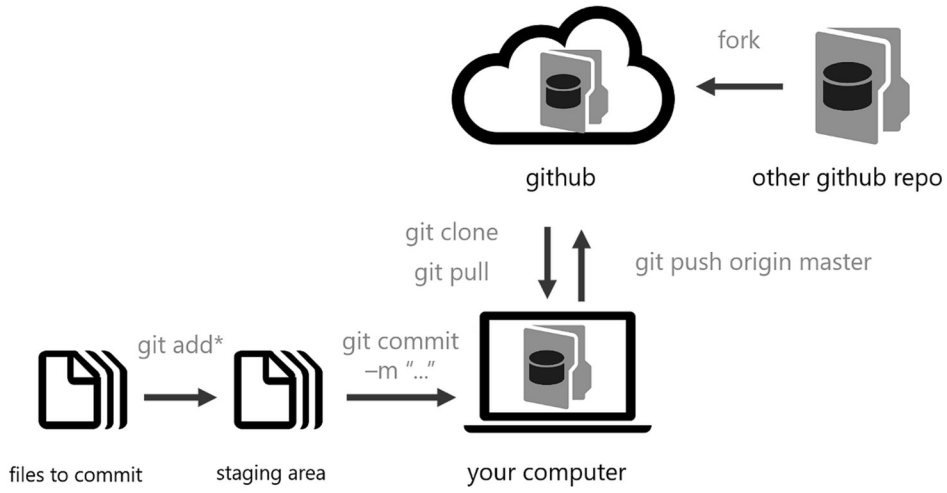
```bash
git fetch origin
```

git fetch <*remote*> obține actualizări de la depozitul remote specificat fără a le integra în branch-ul tău local. Aceasta actualizează referințele la depozitul remote.

```bash
git pull origin main
```

git pull <*remote*> <*branch*> descarcă date de la depozitul remote și le integrează în branch-ul curent. Acesta este utilizat pentru a aduce modificările din depozitul remote în branch-ul tău local.

Aceste exemple arată utilizarea practică a comenzilor pentru a clona, gestiona și sincroniza depozitele Git cu depozitele remote. Comenzile sunt utile pentru colaborarea în echipă și pentru păstrarea versiunilor sincronizate între depozitele locale și cele remote.



Tema 10: Crearea și aplicarea etichetelor (tag)

Comenzi utilizate | *git tag* |

Descriere

Etichetele (tags) în Git sunt puncte de referință fixe în istoricul depozitului care sunt utilizate pentru a marca puncte semnificative în dezvoltarea proiectului. Acestea sunt utile pentru a identifica versiuni specifice, lansări sau momente cheie în istoricul commit-urilor. Comenzile menționate sunt folosite pentru a crea, gestiona și aplica etichete în Git.

git tag <nume_etichetă> permite crearea unei etichete simple cu un nume dat. Aceasta marchează punctul curent în istoricul depozitului cu numele specificat.

git tag -a <nume_etichetă> -m "mesaj_etichetă" creează o etichetă cu un mesaj asociat, care poate oferi detalii despre lansare sau versiunea marcată.

git tag -l afișează o listă a tuturor etichetelor din depozit.

git show <nume_etichetă> afișează informații despre o etichetă specifică, inclusiv commit-ul la care este ancorată.

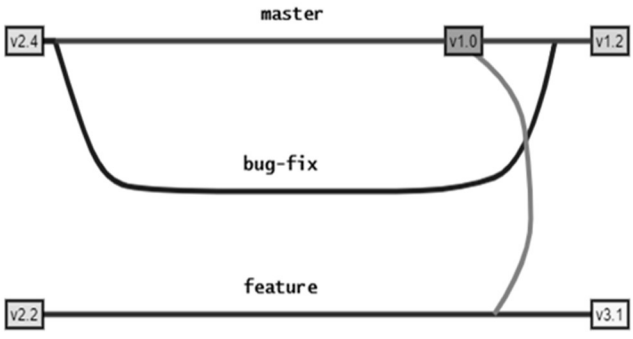
git push origin <nume_etichetă> este folosit pentru a încărca o etichetă la depozitul remote, permițând astfel altor colaboratori să acceseze eticheta.

git checkout <nume_etichetă> permite să treci la punctul specific marcat de etichetă, trecând la acel commit și creând o ramură detasată. Acest lucru poate fi util pentru a examina sau a face modificări într-o versiune specifică a proiectului.

Exemplu

bash

```
git tag <eticheta>
git tag -a <eticheta> -m "text"
git tag -1
git show <eticheta>
git push origin <eticheta>
git checkout <eticheta>
```



Figură 21 git tag