Dezvoltarea Aplicațiilor Web utilizând ASP.NET Core MVC
Curs 8 - GIT

Cuprins

Ce este GIT

GIT este un sistem de control al versiunilor distribuit, utilizat pentru a urmări modificările din fișierele unui proiect, coordonând munca în echipă. Este folosit de dezvoltatori pentru a colabora eficient și pentru a păstra un istoric complet al proiectelor software.

Fiecare copie a unui depozit (repository) conține întregul istoric al proiectului. GIT este rapid, iar datele sunt stocate în siguranță, reducând riscul pierderii informațiilor.

Crearea unui cont

Un cont pe o platformă precum GitHub, GitLab sau Bitbucket este esențial pentru a gestiona proiectele software, pentru colaborare în echipă și pentru păstrarea unui backup al codului. Aceste platforme oferă spațiu pentru găzduirea repository-urilor, permit controlul accesului la proiecte și includ instrumente pentru managementul proiectelor.

Un repository online este o copie centralizată a codului, accesibilă de pe orice dispozitiv conectat la internet.

Pentru a utiliza GIT se procedează astfel:

- Se accesează una dintre platformele populare:
  - GitHub – https://github.com/ (cel mai popular pentru proiecte open-source);
  - GitLab – https://gitlab.com/ (folosit pentru securitate sporită și funcții enterprise);
  - Bitbucket – https://bitbucket.org/ (folosit pentru proiecte private în echipe mici);

- Se înregistrează un cont și se confirmă e-mailul

Instalarea GIT

Windows

1. Se descarcă executabilul de pe site-ul oficial (https://git-scm.com/)
2. Se rulează installer-ul și se urmează pașii
    - Se selectează opțiunea de linie de comandă și un editor (de ex. VS Code/ Visual Studio)
3. Se deschide **Git Bash** pentru a verifica instalarea

```
git --version
```

Linux

1. Se deschide terminalul și se execută:

```
sudo apt update
sudo apt install git
```

2. Se verifică versiunea:

```
git --version
```

Mac OS

1. Se descarcă Git prin Homebrew:

```
brew install git
```

2. Se verifică versiunea instalată:

```
git --version
```

Configurarea GIT

Pentru a utiliza GIT corespunzător, este necesară configurarea utilizatorului. Configurarea se face astfel:

- Se deschide terminalul/command prompt
- Se configurează numele și e-mailul:

```
git config --global user.name "Numele Tău"
git config --global user.email "email@exemplu.com"
```

- Se verifică configurația:

```
git config --list
```

Utilizarea GIT

În continuare, se vor analiza pașii fundamentali pentru a lucra cu GIT, de la inițializarea unui repository la salvarea și sincronizarea modificărilor.

Crearea unui repository local

Un repository (repo) este un spațiu unde sunt gestionate codul sursă și istoricul modificărilor. Inițializarea unui repo se face urmând pașii:

- Se deschide terminalul (sau **Git Bash** pe Windows)
- Se navighează la directorul unde se dorește crearea repository-ului:

```
cd /cale/catre/proiect
```

- Se creează repository-ul cu comanda:

```
git init
```

Această comandă inițializează un repository GIT în folderul curent și creează un folder ascuns numit **.git**, care conține toate informațiile necesare pentru gestionarea versiunilor. Astfel, directorul **Proiect** este un repository GIT.

După inițializarea unui repository local folosind comanda **`git init`**, este important ca acesta să fie conectat la un repository remote (numit **origin**), pentru a putea sincroniza proiectul cu un spațiu online, precum GitHub, GitLab sau Bitbucket. Această conexiune permite utilizarea comenzilor precum **git push** și **git pull** pentru a urca modificările pe repo sau pentru a le prelua.

După autentificarea pe platforma GIT, se parcurg pașii următori:

- Se accesează secțiunea **Repositories**;
- Se apasă pe **New Repository**;
- Se completează detaliile (numele proiectului, descriere, setările de vizibilitate);
- Se creează repository-ul apăsând **Create Repository**;

După creare, platforma va genera un URL pentru repository-ul remote, de exemplu:

```
https://github.com/utilizator/ProiectNou.git
```

În continuare se conectează repository-ul local la cel remote, folosind comanda de adăugare a originii. Această comandă se rulează în folderul proiectului, unde s-a inițializat repo-ul anterior.

```
git remote add origin <URL>
```

Se inlocuiește **<URL>** cu adresa URL a repository-ului remote:

```
git remote add origin
https://github.com/utilizator/ProiectNou.git
```

Clonarea unui repository local

Ca alternativă de inițializare, se poate utiliza comanda "**`git clone`**" pentru a inițializa un repo local sau pentru a clona un repo deja existent.

Clonarea permite copierea unui repository online pe computerul local. Aceasta comanda copiază întregul repository (inclusiv istoricul modificărilor) pe computerul personal si creează un folder local cu același nume ca repository-ul.

Pașii pentru clonare sunt:

- Se obține URL-ul repository-ului de pe platforma online (GitHub, GitLab etc.);
- În terminal, în folderul în care se dorește copierea proiectului, se folosește comanda:

```
git clone <URL>
```

Exemplu de comandă:

```
git clone
https://github.com/utilizator/ProiectNou.git
```

În acest moment a fost creată o copie locală a repository-ului **ProiectNou** într-un folder numit **ProiectNou**.

Adăugarea fișierelor în repository (repo)

Un pas esențial în utilizarea GIT este adăugarea fișierelor pentru urmărire. Aceasta presupune mutarea fișierelor în **staging area**, pregătindu-le pentru a fi incluse într-un **commit**. Procesul implică selectarea fișierelor care se doresc a fi incluse în istoricul proiectului.

Staging area este un spațiu virtual intermediar între fișierele din directorul de lucru și repository-ul local. Fișierele adăugate în staging area sunt cele care vor fi incluse în următorul **commit**. Acest mecanism permite controlul modificărilor care vor fi salvate în istoricul proiectului.

Pentru adăugarea unui fișier în staging area, se efectuează următorii pași:

- Se creează sau se modifică fișiere în proiect – se poate crea un fișier nou sau se poate edita un fișier existent în directorul local al proiectului.

- Se verifică starea repository-ului
  - Înainte de a adăuga fișiere, se verifică starea repository-ului pentru a vedea ce fișiere sunt neurmărite, modificate sau gata de commit. Acest lucru se face folosind comanda:

```
git status
```

Răspunsul acestei comenzi arată care este starea fișierelor în repo:

```
On branch main
Untracked files:
  (use "git add <file>..." to include in what
will be committed)
    fisier_test.txt
```

Această comandă arată două tipuri de fișiere:

- **Untracked files**: Fișierele nu sunt încă urmărite de GIT;
- **Modified files**: Fișierele existente au fost modificate;

În continuare:

- Se adaugă fișiere pentru urmărire, folosind comanda **`git add`**, pentru a muta fișierele în **staging area**. Adăugarea se poate face pentru unul sau mai multe fișiere.

Pentru a adăuga un fișier specific:

```
git add nume_fișier.txt
```

Pentru a adăuga toate fișierele noi sau modificate din proiect:

```
git add .
```

. indică faptul că toate modificările din directorul curent vor fi adăugate.

Pentru a adăuga un tip specific de fișiere:

```
git add *.txt
```

Această comandă adaugă toate fișierele cu extensia **`.txt`**

- Se verifică starea după adăugare

După ce fișierele sunt adăugate în staging area, se verifică din nou starea repository-ului:

```
git status
```

Exemplu de răspuns pentru această comandă după adăugare:

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to
unstage)
    new file:   fisier_test.txt
```

Changes to be committed reprezintă fișiere care sunt în staging area și vor fi incluse în următorul commit.

Eliminarea unui fișier din staging area

Dacă se adaugă un fișier greșit sau se dorește eliminarea din staging area înainte de a face un commit, se folosește comanda:

```
git restore --staged <nume_fisier>
```

Exemplu:

```
git restore --staged fisier_test.txt
```

Acum fișierul revine la starea de untracked, adică nu va fi inclus în commit-ul curent.

Salvarea modificărilor (commit)

Un commit este un moment esențial în utilizarea GIT. Practic, un commit salvează o "fotografie" (snapshot) a stării fișierelor din **staging area** în istoricul repository-ului. Acest lucru permite păstrarea unui istoric clar al modificărilor, fiind posibilă revenirea la o versiune anterioară, dacă acest lucru este necesar.

Un commit este o unitate de salvare în GIT. Fiecare commit conține:

- Modificările aduse fișierelor;
- Un mesaj descriptiv care explică modificările;
- Identificator unic (hash) pentru urmărirea commit-urilor;
- Informații despre autor (nume, e-mail, dată și oră);

Pașii pentru a face un commit:

- Se adaugă fișierele în staging area. Folosește **`git add`** pentru a include fișierele dorite;
- Se efectuează commit-ul. Modificările din staging area se salvează folosind comanda:

```
git commit -m "Mesaj descriptiv al modificărilor"
```

Exemplu:

```
git commit -m "Adăugat fișierul de configurare inițială"
```

În acest moment, commit-ul a fost creat la nivel local. Pentru ca acesta să poată fi stocat și accesat de toti userii care fac parte din proiect, este necesar ca modificările să fie sincronizate online. Pentru a sincroniza modificările locale cu repository-ul online, se folosește comanda **`git push.`**

Repository-ul local trebuie să fie conectat la un remote (**VEZI secțiunea despre inițializare sau clonare**).

Se duc modificările către branch-ul de lucru (de exemplu: main sau master):

```
git push origin main
```

Această comandă trimite toate commit-urile locale către repository-ul remote. **Origin** este numele implicit al remote-ului, iar **main** este branch-ul implicit. Dacă se utilizează **git push** pentru prima dată, este posibil să fie necesară autentificarea folosind un token personal sau un utilizator/parolă (pe GitHub, de exemplu).

După acest pas, modificările se trimit și se sincronizează online. Acum, toti utilizatorii care fac parte din echipă pot prelua informațiile sincronizate din repo, folosind comenzile **`git clone`** sau **`git pull`**.

Comanda **`git pull`** este utilizată pentru a sincroniza branch-ul curent al repository-ului local cu branch-ul corespunzător din repository-ul remote. Practic, această comandă combină două operațiuni:

- **Fetch** – comandă care preia toate commit-urile noi și actualizările din repository-ul remote;
- **Merge** – care integrează aceste actualizări în branch-ul curent;

Aceasta este o comandă frecvent utilizată. Ea ne asigură că există cea mai recentă versiune a codului în repository-ul remote. Comanda se execută după cum urmează:

```
git pull [<remote>] [<branch>]
```

Exemplu:

```
git pull origin master
```

Comanda preia modificările din online și le aplică local.

Atenție!

Această comandă modifică fișierele locale și poate produce pierderea informației nesalvate inițial (rescrierea fișierelor locale cu cele sincronizate online).

Branch-uri și utilizarea lor

Branch-urile sunt "ramuri" ale proiectului, permițând lucrul paralel fără a afecta codul principal. Branch-urile permit echipelor să lucreze simultan la diferite funcționalități, fără riscul de a altera codul principal înainte ca modificările să fie testate și integrate.

Un branch este o linie paralelă de dezvoltare într-un repository GIT. Este o copie a codului sursă care poate fi modificată fără a afecta direct linia principală de dezvoltare (branch-ul principal este numit main sau master).

Importanța branch-urilor

- **Colaborare eficientă** – fiecare membru al echipei poate lucra pe propriul branch, evitând conflictele directe;
- **Organizare** - branch-urile izolează funcționalități sau sarcini;
- **Siguranță** – codul principal rămâne stabil, iar modificările sunt integrate numai după verificare și aprobare;
- **Flexibilitate** – permite lucrul simultan la funcționalități noi;

Comportamentul branch-urilor în GIT?

Branch-ul implicit:

Când se creează un repository, GIT inițializează un branch implicit numit **main** (sau **master** în versiunile mai vechi).

Branch-urile suplimentare:

Acestea sunt create pentru modificări specifice și pot fi fuzionate ulterior cu branch-ul principal.

Fiecare branch este o referință la un anumit commit. Pe măsură ce se adaugă commit-uri, branch-ul "se mișcă" pentru a indica cel mai recent commit din linia sa.

Comenzi pentru gestionarea branch-urilor

- Listarea branch-urilor existente

  `git branch`

  Această comandă afișează toate branch-urile locale. Branch-ul curent este marcat cu *.

- Crearea unui branch nou

  `git branch <nume_branch>`

  Această comandă creează un branch nou pornind de la branch-ul curent, dar nu comută automat pe el.

- Comutarea pe un branch

  `git checkout <nume_branch>`

  Această comandă comută pe branch-ul specificat.

- Crearea și comutarea pe un branch nou simultan

  `git checkout -b <nume_branch>`

- Ștergerea unui branch

  `git branch -d <nume_branch>`

  Această comandă șterge branch-ul specificat. Utilizați opțiunea **`-d`** pentru a forța ștergerea unui branch necompletat.

Exemplu de adăugare în branch

1. Crearea unui branch pentru o nouă funcționalitate:

   `git branch feature-login`

   `git checkout feature-login`

   Sau într-o singură comandă:

   `git checkout -b feature-login`

2. Implementarea pe branch-ul **feature-login**, făcând modificări și commit-uri:

   ...

   `modificarea fisierului autentificare.cs`

   ...

   `git add autentificare.cs`

   `git commit -m "Implementat funcția de autentificare"`

3. Revenirea la branch-ul principal:

   `git checkout main`

4. Integrarea modificărilor din **feature-login** în main:

   `git merge feature-login`

Tipuri de branch-uri utilizate în proiecte

- **Branch principal (main sau master):**
  - Conține codul stabil, gata de livrare sau utilizare;
  - Modificările trebuie să fie testate înainte de integrarea în acest branch;

- **Branch-uri de funcționalitate (feature branches):**
  - Sunt create pentru dezvoltarea unor funcționalități noi;
  - După finalizare, sunt fuzionate în branch-ul principal sau în branch-ul de dezvoltare;

- **Branch-uri de corecție (hotfix branches):**
  - Sunt utilizate pentru a remedia probleme critice apărute în producție;
  - După aplicarea modificărilor, sunt fuzionate atât în branch-ul principal, cât și în branch-ul de dezvoltare;

- **Branch-uri de dezvoltare (develop branches):**
  - Utilizate pentru a pregăti funcționalități înainte de livrarea în branch-ul principal;

- **Branch-uri de lansare (release branches):**
  - Folosite pentru pregătirea unei versiuni de producție;
  - Permit ajustări finale fără a afecta branch-ul principal;

Fluxul de lucru recomandat în lucrul cu branch-uri

- Se începe întotdeauna de pe branch-ul principal (main);
- Se creează un branch nou pentru fiecare sarcină (feature-login, bugfix-articles, etc);
- Se lucrează pe branch-ul creat, făcând modificări și commit-uri regulate;
- Se testează modificările pe branch;
- Se integrează branch-ul în branch-ul principal, folosind un merge (sau pull request dacă se lucrează în echipă);

Merge request

Un merge request (numit și **pull request pe platforma GitHub**) este o cerere de integrare a modificărilor realizate pe un branch într-un alt branch (de obicei, branch-ul principal, cum ar fi main). Este o etapă esențială în colaborarea cu GIT, mai ales în echipe, deoarece permite revizuirea și aprobarea modificărilor înainte de a le integra în codul principal.

Fluxul de lucru pentru un merge request

- Se creează un branch pentru modificări;
- Se adaugă codul nou sau modificările pe branch-ul creat;
- Se fac commit-uri pentru a salva modificările local;
- Se împinge branch-ul pe repository-ul remote;
- Se creează un merge request de pe platforma de găzduire GIT;
- Alți membri ai echipei pot sa revizuiască și să aprobe cererea;
- Modificările sunt fuzionate în branch-ul țintă;

Cum se adaugă un merge request - Pe GitHub

- După ce branch-ul a fost încărcat, GitHub afișează o notificare pentru a crea un pull request;
- Se apasa pe butonul ***Compare & pull request***;
- Se completează următoarele câmpuri:
    - Titlu (de exemplu: „S-a adaugat sistemul pentru managementul userilor si rolurilor”);
- Descriere – se explică ce modificări au fost făcute, care este rolul lor, etc;
- Se selectează branch-ul sursă și branch-ul țintă (de obicei ***main***);
- Se apasă ***Create Pull Request***;

Cum se adaugă un merge request - Pe GitLab

- Se accesează repository-ul pe GitLab;
- Se navighează la Merge Requests și se apasă New Merge Request;
- Se selectează branch-ul sursă și branch-ul țintă;
- Se completează titlul și descrierea;
- Se apasă Create Merge Request;

Cum se adaugă un merge request - Pe Bitbucket

- Se accesează secțiunea Pull Requests și se apasă pe Create Pull Request;
- Se selectează branch-urile implicate și se completează detaliile cererii;
- Se apasă pe Create;

Best Practices pentru Merge Request-uri

- Este indicat să se realizeze modificări mici și bine definite. Nu este indicat să se implementeze prea multe funcționalități într-un singur merge request.
- Este indicat să se scrie mesaje cât mai clare si detaliate. Se explică modificările și utilitatea lor.
- Testarea este esențială. Este foarte important ca modificările să fie testate înainte de a avea loc un merge request.

Introducere în Modele de Branching

Git Flow este un model de lucru structurat și bine definit pentru gestionarea branch-urilor în GIT, propus de **Vincent Driessen**. Este foarte popular pentru proiecte complexe, deoarece oferă o modalitate clară de organizare a branch-urilor, separând lucrările în funcție de funcționalități, remedieri de bug-uri și lansări de versiuni.

Git Flow definește un set de reguli pentru crearea și gestionarea branch-urilor într-un repository GIT. Se bazează pe utilizarea a cinci tipuri principale de branch-uri:

- **Main** – branch-ul principal, cel care conține întotdeauna cod stabil, gata pentru producție.
- **Develop** – branch-ul de dezvoltare, unde se integrează toate funcționalitățile înainte de a fi lansate.
- **Feature** – branch-uri temporare pentru dezvoltarea noilor funcționalități.
- **Release** – branch-uri dedicate pregătirii unei noi versiuni de producție.
- **Hotfix** – branch-uri temporare pentru remedieri rapide ale problemelor din producție.

În Git Flow, funcționalitățile noi sunt dezvoltate pe branch-uri de **feature**, pornind din **develop**, testate și apoi integrate. Versiunile stabile sunt pregătite pe branch-uri de release, în timp ce problemele critice sunt remediate prin branch-uri de hotfix, plecând din main. Acest model asigură o separare clară a activităților și stabilitatea branch-ului principal. Cu toate acestea, Git Flow poate fi prea complex pentru proiectele mici.

Cum funcționează Git Flow?

Branch-ul main

- Reprezintă linia de producție (live) a proiectului.
- Orice commit pe acest branch este o versiune oficială stabilă.
- Nu se lucrează direct pe acest branch, ci doar se fuzionează branch-uri pregătite și testate.

Branch-ul develop

- Toate funcționalitățile și modificările sunt integrate aici înainte de a fi gata pentru producție.
- Este punctul de plecare pentru branch-urile de feature și release.
- După testare completă, modificările din develop sunt integrate în main.

Branch-uri de feature

- Create pentru fiecare funcționalitate nouă, pornind de la develop.
- După finalizarea dezvoltării și testării, sunt fuzionate înapoi în develop.

Exemplu de creare și utilizare:

```
git checkout -b feature/login develop
# Dezvoltare cod
git add .
```

```
git commit -m "Implementat componenta de
autentificare"
git checkout develop
git merge feature/login
git branch -d feature/login
```

Branch-uri de release

- Create din develop pentru pregătirea unei noi versiuni stabile.
- Permit ajustări finale, teste și corectări minore, fără a afecta dezvoltarea pe develop.
- După finalizare, sunt fuzionate atât în main, cât și în develop.

Exemplu:

```
git checkout -b release/1.0 develop
# Ajustări finale
git commit -m "Actualizat documentația pentru
versiunea 1.0"
git checkout main
git merge release/1.0
git tag -a v1.0 -m "Versiunea 1.0"
git checkout develop
git merge release/1.0
git branch -d release/1.0
```

Branch-uri de hotfix

- Create din main pentru remedierea urgentă a problemelor critice din producție.
- După corectare, modificările sunt fuzionate în main și în develop.

Exemplu:

```
git checkout -b hotfix/urgent-bug main
# Reparare problemă
git add .
git commit -m "Rezolvat bug-ul critic"
git checkout main
git merge hotfix/urgent-bug
git checkout develop
git merge hotfix/urgent-bug
git branch -d hotfix/urgent-bug
```

Fluxul de lucru cu Git Flow

- **Dezvoltarea unei funcționalități:**
  - Se creează un branch de feature din develop.
  - După finalizare, se fuzionează branch-ul în develop.

- **Pregătirea unei versiuni noi:**
  - Se creează un branch de release din develop.
  - Se rezolvă eventualele probleme și se finalizează documentația.
  - Se fizionează branch-ul de release în main și develop, apoi se marchează versiunea (tag).

- **Rezolvarea unui bug critic:**
  - Se creează un branch de hotfix din main.
  - Se corectează problema, se testează și se fuzionează branch-ul în main și develop.

Exemplu Concret

Scenariul: Ana și Mihai lucrează pe la un proiect web. Ana adaugă pagina Home, Mihai adaugă pagina About. Ambii modifică același fișier Program.cs → o să apară un conflict.

1. Setup inițial (o singură dată)

Pe GitHub se creează repository "MyWebApp"

Ana – inițializează proiectul

```
dotnet new webapp -n MyWebApp
cd MyWebApp
git init
git remote add origin https://github.com/team/MyWebApp.git
git add .
git commit -m "Initial commit - ASP.NET Core 9 webapp"
git push -u origin main
```

Mihai – clonează proiectul

```
git clone https://github.com/team/MyWebApp.git
```

2. Ana implementeaza un nou feature – Home Page

```
git checkout -b feature/home-page
```

Ana editează **Index.cshtml**

```
<h1>Bine ai venit!</h1>
<p>Aceasta este pagina principală</p>
```

Editează **Program.cs** – adaugă logging

```
builder.Logging.AddConsole(); // Adăugat de Ana
git add .
git commit -m "Adaugă conținut pagina Home + logging"
git push origin feature/home-page
```

Creează **pull-request** pe GitHub și așteaptă review

3. Mihai adaugă un nou feature – About Page

```
git checkout -b feature/about-page
```

Mihai creează **About.cshtml**

```
<h1>Despre Noi</h1>
```

Editează **Program.cs** – adaugă HTTPS

```
builder.Services.AddHttpsRedirection(o => o.HttpsPort =
443); // Adăugat de Bogdan
git add .
git commit -m "Adaugă pagina About + HTTPS config"
git push origin feature/about-page
```

4. Pull-requestul Anei este aprobat și merge-uit

Pe GitHub: **Merge pull request**.
Main (master) conține acum codul Anei.

5. Mihai: Sincronizare + Rezolvare Conflict

```
git checkout main
git pull origin main
git checkout feature/about-page
git merge main
```

O să apară conflict în Program.cs!

Mihai deschide Program.cs și vede:

```
<<<<<<< HEAD
builder.Services.AddHttpsRedirection(...);
=======
builder.Logging.AddConsole();
>>>>>>> main
```

Rezolvare: Mihai păstrează AMBELE linii, după care șterge marcajele:

```
builder.Logging.AddConsole();
builder.Services.AddHttpsRedirection(...);
git add Program.cs
git commit -m "Rezolvă conflict - păstrează logging + HTTPS"
git push origin feature/about-page
```

6. Mihai face Pull-request și Merge

Pe GitHub Ana face review, după care aprobă. O să facă un **merge pull request**

În final, ambii sincronizează:

```
git checkout main && git pull origin main
```