Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №1
із дисципліни «*Технології Розробки Програмного Забезпечення*»
Тема: «Системи контролю версій. Git.»
Варіант-11

Виконав:
Студент групи ІА-24
Коханчук Михайло Миколайович

Перевірив:
Мягкий Михайло Юрійович

Київ-2024

ЗМІСТ

МЕТА

Метою цієї лабораторної роботи є оволодіння практичними навичками роботи з системою контролю версій Git. У межах роботи передбачається створення локального та віддаленого репозиторію, налагодження їхньої взаємодії, додавання та відстеження файлів, здійснення комітів, а також виконання операцій з передачі змін із локального репозиторію у віддалений.

Окрім того, мета включає ознайомлення з основними командами Git для ефективного управління версіями, організацію структури репозиторію, налаштування його змісту відповідно до потреб проєкту та впровадження базової роботи з пул-реквестами.

Теоретичні відомості:

Система контролю версій — це інструмент, який дозволяє відстежувати і управляти змінами в файлах та документах, що використовуються в процесі розробки програмного забезпечення або інших проектів. Система контролю версій зберігає історію змін у файлах, дозволяючи повертатися до попередніх версій проекту за потреби. Вона підтримує спільну роботу кількох користувачів, допомагаючи зберігати та зливати зміни, а також вирішувати конфлікти. Крім того, система дозволяє відстежувати, хто і коли вніс зміни, що спрощує аналіз історії проекту. Якщо файли втрачені або пошкоджені, їх можна легко відновити.

Види систем контролю версій:

1. *Локальні системи контролю версій*
   Історія змін зберігається на одному комп'ютері. Простий приклад — ручне копіювання файлів у різні каталоги, хоча це неефективний метод.

2. *Централізовані системи контролю версій*
   Усі файли і історія зберігаються на центральному сервері, і користувачі отримують доступ до цих файлів через цей сервер. Приклади: Subversion(SVN), CVS.

3. *Розподілені системи контролю версій*
   Кожен розробник має повну копію репозиторію з усією історією змін. Це дозволяє працювати офлайн і зменшує залежність від центрального сервера. Приклади: Git, Mercurial.

Git — це розподілена система контролю версій, де кожен користувач зберігає повну копію репозиторію, що дозволяє працювати автономно і зберігати всю історію змін. Git підтримує швидке створення *гілок* для розробки окремих функцій, полегшуючи паралельну роботу. Кожен *коміт* фіксує авторство та вміст змін, з можливістю автоматичного злиття і вирішення конфліктів між користувачами. Такі сервіси, як GitHub чи GitLab, забезпечують спільний доступ до *репозиторіїв*.

Основна термінологія git:

Репозиторій(repository) — це сховище, яке містить всі файли проекту та історію їх змін. Він включає всі коміти, гілки та метадані, необхідні для управління версіями проекту. Репозиторій може бути локальним(на вашій машині) або віддаленим(на серверах, таких як GitHub чи GitLab).
Робоча директорія(working directory) — це поточна папка проекту, де знаходяться файли, над якими ви працюєте. У робочій директорії відбуваються зміни файлів перед їх додаванням у стейджинг область і створенням коміту. Вона відображає стан проекту у вибраній гілці.

Стейджинг область(staging area) — це тимчасова зона для підготовки змін перед створенням коміту. Зміни спочатку додаються в робочу область(папку), а щоб включити їх у коміт, потрібно додати їх у стейджинг за допомогою команди `git add`. Це дозволяє вибірково підготувати зміни для фіксації.

Коміт(commit) — це зафіксована версія проекту, яка містить всі зроблені зміни на момент коміту. Кожен коміт включає посилання на попередню версію, повідомлення про зміни, автора, дату, час та унікальний хеш(SHA). Коміти дозволяють зберігати та відновлювати стан проекту і забезпечують історію змін.
Гілки(branches) — це незалежні лінії розробки, які дозволяють працювати над окремими функціями чи виправленнями, не змінюючи основний код. Основна гілка зазвичай називається main або master, але можна створювати нові гілки для різних завдань і зливати їх з основною після завершення роботи.
Тег(tag) — це мітка у Git, яка використовується для позначення певного коміту, зазвичай для важливих моментів у проекті, таких як випуск релізу. Теги дозволяють швидко ідентифікувати конкретну версію проекту. Є два типи тегів: анотаційні (annotated) теги, які містять додаткову інформацію, таку як ім'я автора, дата і повідомлення, та легкі (lightweight) теги, які є просто посиланням на конкретний коміт без додаткових даних.
Git-команда — це інструкція, яку ви вводите в терміналі для виконання певної дії в репозиторії. Команди в Git використовуються для виконання різних операцій, таких як створення комітів, злиття гілок, перевірка статусу проекту тощо.
Кожна команда може містити опції або аргументи, які змінюють поведінку команди або уточнюють, які саме дії потрібно виконати. Опції зазвичай починаються з одного або двох дефісів(наприклад, -m або --message) і дозволяють вам надавати додаткову інформацію або налаштування. Також, зазвичай, більшість опції можна комбінувати.

Зміни у Git зберігаються в прихованій папці .git, яка містить метадані репозиторію, включаючи історію комітів, гілки, посилання та індексацію. Вона зберігає об'єкти у вигляді blobs(Binary Large Objects), дерев(Trees), комітів та тегів, що дозволяє Git швидко відстежувати та відновлювати зміни в проекті.

Детальний опис вмісту папки .git:
HEAD – вказує на поточну гілку або коміт.
config – файл конфігурації репозиторію (локальні налаштування, такі як користувацькі параметри або налаштування віддалених репозиторіїв).
description – опис проєкту (використовується переважно у bare-репозиторіях).
hooks/ – містить скрипти, які можуть автоматично виконуватися на певні події (наприклад, перед комітом або перед пушем).
info/ – зберігає допоміжну інформацію, зокрема exclude – список файлів, що ігноруються (аналог .gitignore).
objects/ – зберігає всі об'єкти Git (коміти, дерева, бло́би) у вигляді SHA-1 хешів.
refs/ – містить посилання на гілки (у папці heads/) та теги (у папці tags/).
logs/ – зберігає рефлоги, які відслідковують зміни у гілках та інших посиланнях.
index – індексний файл, що зберігає інформацію про файли в індексі (staging area).
packed-refs – містить посилання, стиснені для зменшення використання місця (коли рефів стає багато).

Основні команди Git:

`git init`
Створює приховану папку .git, де зберігається вся інформація про версії проекту, включаючи коміти, гілки, логи змін. Після виконання цієї команди каталог стає Git-репозиторієм, готовим до відстеження змін.

`git status`
Показує, які файли були змінені, додані або видалені, але ще не зафіксовані(коміти). Також вказує, чи є незатрекані файли (нові файли, які не відслідковуються Git) та чи є коміти, які можна зробити.

`git add`
Команда додає зміни з робочої директорії до staging area(індексу), готуючи їх до наступного коміту. Вона не фіксує зміни в історії, а лише вказує Git, які файли або частини файлів потрібно включити в наступний коміт. Це також включає додані або змінені файли, але не видалені файли, якщо не використовувати спеціальні прапорці (наприклад, git add -A для відстеження всіх змін).

Команда має декілька корисних опцій:

`git add .`
Додати всі файли в поточній директорії.

`git add -u`
Додати тільки змінені та видалені файли.

`git add <filename>`
Додати зміни в одному конкретному файлі.

`git add -i`
Включає повноцінний інтерактивний режим додавання змін у Git, що дозволяє більш точно керувати файлами та їх частинами перед комітом

`git commit`
Команда git commit фіксує зміни, додані до staging area, в історію репозиторію, створюючи новий коміт. Коміт зберігає знімок проєкту в певний момент часу і включає метадані: автора, дату, повідомлення та хеш(SHA-1). Для створення коміту обов'язково потрібно додати описові повідомлення(git commit -m "повідомлення"), які пояснюють, що саме було змінено.

Команда має декілька корисних опцій:

`git commit -m "Your commit message"`
Дозволяє вказати повідомлення коміту прямо в командному рядку. Без цієї опції

Git відкриє текстовий редактор за замовчуванням для введення повідомлення.

`git commit -a -m "Updated all modified files"`
Автоматично додає всі змінені файли(unstaged changes) (ті, які вже були під контролем версій) у коміт, минаючи стадію додавання до індексу (git add). Ця опція не додає нові файли, які ще не були додані до індексу.

`git commit --amend -m "Updated commit message"`
Дозволяє змінити останній коміт. Перезаписує хеш коміту. Ця опція корисна, коли ти хочеш виправити помилки в попередньому коміті (наприклад, додати забутий файл або змінити повідомлення коміту).

`git commit --allow-empty -m "This is an empty commit"`
Створює "порожній" коміт, навіть якщо немає змін у файлах. Це корисно для створення комітів із повідомленням для фіксації певних подій або дій.

`git stash`
Команда git stash зберігає всі незакомічені зміни(як індексовані, так і неіндексовані) у тимчасове сховище в папці .git/refs/stash, очищаючи робочу директорію. Це дозволяє перемикатися на іншу гілку або виконувати інші дії без втрати поточних змін. При стешуванні створюються два додаткових коміти: один для індексованих змін і ще один для неіндексованих. Пізніше ці зміни можна відновити за допомогою додаткових опцій команди. Це корисно для тимчасового збереження незавершеної роботи.

Команда має декілька корисних опцій:

`git stash`
Зберігає всі незакомічені зміни, як у стейджинг-області, так і в робочій директорії. Після виконання робоча директорія повертається до чистого стану(відповідного останньому коміту)

`git stash –all`
Зберігає всі зміни, включаючи файли, які були проігноровані (.gitignore).

`git stash -u`
Команда зберігає не тільки індексовані та неіндексовані зміни, але й невідстежувані файли(untracked files) у сховище. Це дозволяє тимчасово прибрати всі зміни з робочої директорії, включно з новими файлами, які ще не були додані до індексу.

`git stash apply stash@{n}`
Команда відновлює зміни з конкретного сховища, вказаного індексом n. У цьому випадку n – це номер стешу у списку, який можна переглянути за допомогою команди git stash list. Використання apply дозволяє застосувати зміни до робочої директорії без видалення відповідного стешу з сховища, що дає можливість знову їх застосувати в майбутньому, якщо це потрібно.

git stash pop
Застосовує останній збережений стеш і видаляє його зі стеку стешів

git stash list
Виводить список усіх стешів у вигляді журналу збережених змін.
Кожен стеш має унікальний індекс, який можна використовувати для доступу до нього.

git stash show
Показує зміни, збережені в останньому стеші, в короткій формі.

git stash clear
Очищає всі збережені стеші.

git checkout
Команда git checkout використовується для перемикання між гілками, на певні коміти або для відновлення файлів до певного стану.

Команда має декілька корисних опцій:

git checkout <commit>
Це від'єднає HEAD від поточної гілки та перемістить його на зазначений коміт.

git checkout <branch>
Це перенесе HEAD на зазначену гілку.

git checkout HEAD^
git checkout master^
Вказує git що потрібно знайти батька посилання до якого застосована каретка

git checkout -b <branch>
Створює та переміщається на гілку

git checkout -
Переміщається на попередню гілку

git checkout -- <filename>
Ця команда відновлює файл з останнього коміту у вашій поточній гілці. Іншими словами, вона скидає зміни в цьому файлі до того стану, який був у поточному коміті. Аналог git restore --source=<commit> <file>

git checkout HEAD <filename>
Це подібно до попередньої команди, але конкретно вказує на те, що файл має бути відновлений до стану з останнього коміту на поточній гілці (тобто стану HEAD).

git switch

Команда використовується для перемикання між гілками в Git, а також для створення нових гілок. Вона була введена в Git 2.23 як частина зусиль щодо спрощення інтерфейсу, щоб уникнути плутанини, пов'язаної з командою git checkout

Команда має декілька корисних опцій:

git switch --detach <commit>
Це від'єднає HEAD від поточної гілки та перемістить його на зазначений коміт.

git switch <branch>
Це перенесе HEAD на зазначену гілку.

git switch HEAD^
git switch master^
Вказує git що потрібно знайти батька посилання до якого застосована каретка

git switch -c <branch>
Створює та переміщається на гілку

git switch -
Переміщається на попередню гілку

git branch

Команда git branch використовується для управління гілками в Git. Вона дозволяє створювати, видаляти та переглядати гілки. Без опцій команда показує список усіх локальних гілок у вашому репозиторії. Виділяє поточну гілку за допомогою зірочки (*).

Команда має декілька корисних опцій:

git branch <branch>
Створення гілки із назвою branch

git branch <branch> <commit>
Створення гілки на вказаному коміті.

git branch -r
Відображає віддалені гілки, що доступні у віддаленому репозиторії.

git branch -f <branch> HEAD~n
Це примусово пересуне гілку feature на n коміти назад від поточного положення HEAD. Ця операція перезапише історію гілки, тому варто бути обережним, щоб не втратити незбережені зміни або коміти.

git branch -f <branch> <commit>
Це примусово пересуне гілку feature на вказаний коміт.

`git branch -d <branch>`
Щоб видалити гілку, яка більше не потрібна на локальному репозиторії

`git branch -m <new_branch_name>`
Перейменування гілки

`git merge`
Команда git merge використовується для об'єднання змін з однієї гілки в іншу. Це дозволяє інтегрувати різні гілки, об'єднуючи їх історії.

Fast-forward злиття (за замовчуванням)
Якщо поточна гілка повністю “відстає” від цільової гілки, Git просто перемотає вказівник поточної гілки вперед до цільової без створення нового коміту злиття. Fast-forward злиття відбувається, коли немає інших, відмінних комітів між поточною та цільовою гілками.

Команда має декілька корисних опцій:

`git merge <branch>`
Ця команда зливає гілку <branch> в поточну гілку.

`git merge --no-ff <branch>`
Якщо ви хочете, щоб навіть при можливому fast-forward Git створив окремий коміт злиття, ви можете використовувати опцію **`--no-ff`**. Це корисно для збереження явної історії злиття.

`git merge --squash <branch>`
Опція --squash дозволяє об'єднати всі коміти з цільової гілки в один коміт у поточній гілці, але не створює коміт злиття. Замість цього зміни додаються в індекс (стейджинг-область), і користувач самостійно виконує коміт після цього

`git merge --abort`
Якщо виникли конфлікти, ви можете скасувати процес злиття за допомогою цієї команди, повернувши репозиторій у стан до початку злиття.

`git merge --continue`
Після вирішення конфліктів і додавання файлів до індексу можна продовжити злиття.

`git merge --strategy=recursive <branch>`
Дозволяє обрати стратегію об'єднання. Наприклад, recursive — стандартна стратегія, але можна використовувати і ours чи theirs, щоб при конфлікті автоматично вибирати зміни однієї з гілок.

`git rebase`
Команда git rebase використовується для зміни історії комітів шляхом переміщення або "перезапису" комітів на нову базу – певну гілку або коміт.

Команда має декілька корисних опцій:

`git rebase <branch>`
Використовується для ребейзу поточної гілки на вказану гілку.

`git rebase -i HEAD~5`
`git rebase -i <commit>`
За допомогою опції **`-i`** або ж **`--interactive`** можливо запустити інтерактивний ребейз починаючи з певного коміту до найновішого на певній гілці; Git відкриє редактор, у якому можна буде змінити порядок або дії над комітами (наприклад, squash, reword, fixup, drop тощо).

`git rebase --no-ff <branch>`
Використовується для примусового збереження нових комітів під час ребейзу, навіть якщо вони можуть бути об'єднані за допомогою fast-forward. Це дозволяє уникнути fast-forward під час процесу ребейзу.

`git rebase --continue`
Повертає вас до поточного процесу ребейзу з того моменту, де ви зупинилися.
Якщо всі конфлікти вирішені, процес ребейзу продовжиться автоматично.
Якщо є ще невирішені конфлікти, Git знову попросить вас їх вирішити.

`git rebase --abort`
Скасовує процес ребейзу і повертає гілку в її початковий стан. Використовується, коли виникають проблеми під час ребейзу або якщо ти вирішив не продовжувати операцію.

`git rebase --quit`
Процес ребейзу тимчасово припиняється.
Ваш прогрес не скасовується: зміни, які ви вже вирішили та додали до стейджингу, збережені.
Конфліктні файли не будуть перезаписані.

`git rebase --skip`
Пропускає поточний коміт і продовжує процес ребейзу. Використовується, коли ти хочеш пропустити коміт, який викликає конфлікти, або якщо вирішив, що коміт більше не потрібен.

`git cherry-pick`
Це команда, яка дозволяє вибірково перенести один або кілька комітів з однієї гілки в іншу, зберігаючи їх вміст. Вона застосовується, коли вам потрібно перенести конкретні зміни (коміти) з однієї гілки в іншу, не зливаючи всі зміни між гілками.

Команда має декілька корисних опцій:

`git cherry-pick <commit>`
Опція дозволяє побачити історію для всіх гілок у репозиторії.

`git cherry-pick <start_commit>^..<end_commit>`
Виводить перші 7 символів хешу кожного коміту конкретної гілки в одному рядку.

`git cherry-pick -n <commit>`
Відображає коміти у вигляді символічного дерева, що показує злиття та розгалуження гілок.

`git cherry-pick --abort`
Дозволяє порівняти дві гілки та вивести коміти, які є в одній гілці, але відсутні в іншій.

`git cherry-pick --quit`
Опція дозволяє вивести коміти зазначеного автора.

`git reset`
Команда яка дозволяє "відкотити" стан репозиторію до попередньої точки. Вона може змінювати стан робочої директорії, стейджинг області або повертати на певний коміт без видалення історії комітів. Команда має кілька режимів, які визначають, як саме буде скасовано зміни.

Основні режими **`git reset`**:

`--soft`:

Відкочує тільки коміти, залишаючи всі зміни в стейджинг області. Файли залишаться підготовленими для коміту, але Git повернеться до попереднього коміту.

`--mixed` (режим за замовчуванням):

Скидає коміти і стейджинг область, залишаючи зміни у робочій директорії. Після виконання команди зміни залишаться у робочій директорії, але будуть вилучені зі стейджинг області.

`--hard`:

Повністю скидає коміти, стейджинг область і робочу директорію до вказаного коміту. Всі зміни, що були зроблені після вказаного коміту, будуть втрачені назавжди.

Команда має декілька інших корисних опцій:

git reset <commit>
git reset HEAD~1
Змінює стан поточної гілки, переміщуючи її HEAD на вказаний коміт. Це дозволяє "відкотитися" до попереднього коміту або змінити місце поточної гілки.

git reset HEAD filename
Ця команда скасовує додавання файлу до індексу (staged changes), але не скидає локальні зміни. Іншими словами, зміни в файлі залишаються, але цей файл більше не буде додано до коміту.

git restore
Дозволяє відновлювати файли до певного стану або скасовувати зміни в робочій директорії або стейджинг області. Вона менш "груба" порівняно з git reset і надає більш контрольований спосіб керування файлами та їх змінами, без зміни комітів

Команда має декілька корисних опцій:

git restore <file>
Відновлення файлів із індексу. Це означає, що після виконання скасуються локальні зміни, повертаючи файли до стану, який збережений в індексі.

git restore --source=<commit> <file>
Це скасує всі зміни у файлі, які відбулись після зазначеного коміту, і відновить його до того стану, який був на момент цього коміту.

git restore --staged <file>
Це видалить файл із індексу, але збереже зміни в робочій директорії.

git restore --ignore-unmerged <file>
Використовується для пропуску файлів, що знаходяться у стані конфлікту під час мерджу.

git revert
Дозволяє скасувати зміни, внесені певним комітом, шляхом створення нового коміту, який робить протилежні зміни.

Команда має декілька корисних опцій:

git revert <commit>

Використовується для скасування одного або декількох комітів, але не видаляє коміти з історії, а створює новий коміт, який зворотньо змінює (ревертує) внесені раніше зміни.

`git revert <commit> --no-commit`
Ця опція застосовує реверс змін, але не робить автоматичний коміт. Це дає можливість внести додаткові зміни перед комітом.

`git revert <merge_commit> -m 1`
Використовується для реверсування комітів злиття (merge commit). Оскільки злиття має два батьківські коміти, потрібно вказати, який із них вважати основною лінією розвитку(батьківським комітом). В даній команді -m 1 означає, що основним батьком є перший батько.

`git log`
Показує список комітів із детальною інформацією про кожен коміт у поточній гілці, включаючи його унікальний хеш(SHA), автора, дату та повідомлення про коміт. Ви можете переглядати попередні версії змін і аналізувати, хто і коли зробив певні правки.

Команда має декілька корисних опцій:

`git log --all`
Опція дозволяє побачити історію для всіх гілок у репозиторії.

`git log --oneline`
Виводить перші 7 символів хешу кожного коміту конкретної гілки в одному рядку.

`git log --graph`
Відображає коміти у вигляді символічного дерева, що показує злиття та розгалуження гілок.

`git log <branch1>..<branch2>`
Дозволяє порівняти дві гілки та вивести коміти, які є в одній гілці, але відсутні в іншій.

`git log --author="John Doe"`
Опція дозволяє вивести коміти зазначеного автора.

`git log --since="2 weeks ago"`
`git log --until="2023-01-01"`
Дані опції дозволяють вивести коміти зазначеної певним чином дати.

`git log --grep="bug fix"`
Опція дозволяє вивести коміти за ключовим словом у повідомленні коміту.

git reflog

Команда показує історію всіх змін, які відбувалися з посиланнями(refs), такими як HEAD, включно з тими, що не зберігаються у звичайному історичному логу комітів. Це може включати перемикання гілок, ребейзи, ресети та інші дії, що впливають на поточне посилання. На відміну від `git log`, яка відображає історію комітів в репозиторії, reflog показує повну історію дій, включаючи ті, що були змінені чи видалені, і дозволяє відновити коміти навіть після "втрат".

Сама команда без додаткових опції дозволяє переглянути список всіх змін позиції HEAD у локальному репозиторії.

Команда має декілька корисних опцій:

git reflog <branch>

Перегляд reflog для конкретної гілки.

git reflog <filename>

Перегляд reflog для конкретного файлу.

git reflog HEAD@{2.days.ago}
git reflog master@{2.hours.ago}

Ці команди показують запис рефлогу для HEAD/master.

git diff

Команда може показати, які зміни були внесені у робочій папці з моменту останнього коміту.

Команда має декілька корисних опцій:

git diff --cached

Порівняння змін між стейджинг областю та останнім комітом.

git diff <commit1> <commit2>

Порівняння двох комітів щоб побачити, які зміни були внесені між ними.

git diff <branch1> <branch2>

Також можна порівняти зміни між двома гілками

git tag

Команда git tag використовується для створення та управління мітками(tags) у Git, які слугують як фіксовані точки в історії комітів, зазвичай для позначення релізів чи важливих версій.

Сама команда без додаткових опції дозволяє переглянути список усіх тегів

Команда має декілька корисних опцій:

`git tag <tag_name> <commit>`
Створення тега на конкретному коміті

`git tag <tag_name>`
Це створить тег на поточному коміті.

`git tag -a <tag_name> -m "<message>"`
Анотований тег(annotated tag) містить додаткову інформацію: ім'я автора, дату створення і повідомлення.

`git remote`
Використовується для керування віддаленими репозиторіями. Відображає підключені віддалені репозиторії або дозволяє додавати/видаляти їх.

Команда має декілька корисних опцій:

`git remote -v`
Показує URL-адреси для читання та запису віддалених репозиторіїв.

`git remote add <name> <url>`
Додає новий віддалений репозиторій із певним ім'ям.

`git remote remove <name>`
Видаляє віддалений репозиторій.

`git fetch`
Завантажує зміни з віддаленого репозиторію, але не інтегрує їх у локальну гілку. Ви можете побачити, які зміни були зроблені у віддаленому репозиторії, і вирішити, що з ними робити.

Команда має декілька корисних опцій:

`git fetch <remote>`
Завантажує всі зміни з вказаного віддаленого репозиторію.

`git fetch <remote> <branch>`
Завантажує лише певну гілку з віддаленого репозиторію, не торкаючись інших гілок

`git fetch --all`
Завантажує всі оновлення з усіх віддалених репозиторіїв, підключених до проекту

`git pull`

Завантажує зміни з віддаленого репозиторію і автоматично інтегрує їх у поточну гілку (це фактично комбінація git fetch та git merge)

Команда має декілька корисних опцій:

`git pull <remote> <branch>`
Отримує зміни з конкретної гілки віддаленого репозиторію і зливає їх з поточною гілкою.

`git pull --rebase`
Оновлює локальні зміни з віддаленого репозиторію, використовуючи ребейз замість злиття(merge). Це дозволяє інтегрувати зміни без створення додаткових комітів злиття.

`git push`
Команда може показати, які зміни були внесені у робочій папці з моменту останнього коміту.

Команда має декілька корисних опцій:

`git push <remote> <branch>`
Відправляє зміни з вашої локальної гілки до певної гілки на віддаленому репозиторії.

`git push --set-upstream <remote> <branch>`
Порівняння двох комітів щоб побачити, які зміни були внесені між ними.

`git push <branch1> <branch2>`
Також можна порівняти зміни між двома гілками

`git clone`
Копіює весь вміст віддаленого репозиторію на ваш комп'ютер. Ця команда створює новий локальний репозиторій із усіма файлами, історією та гілками.

Команда має декілька корисних опцій:

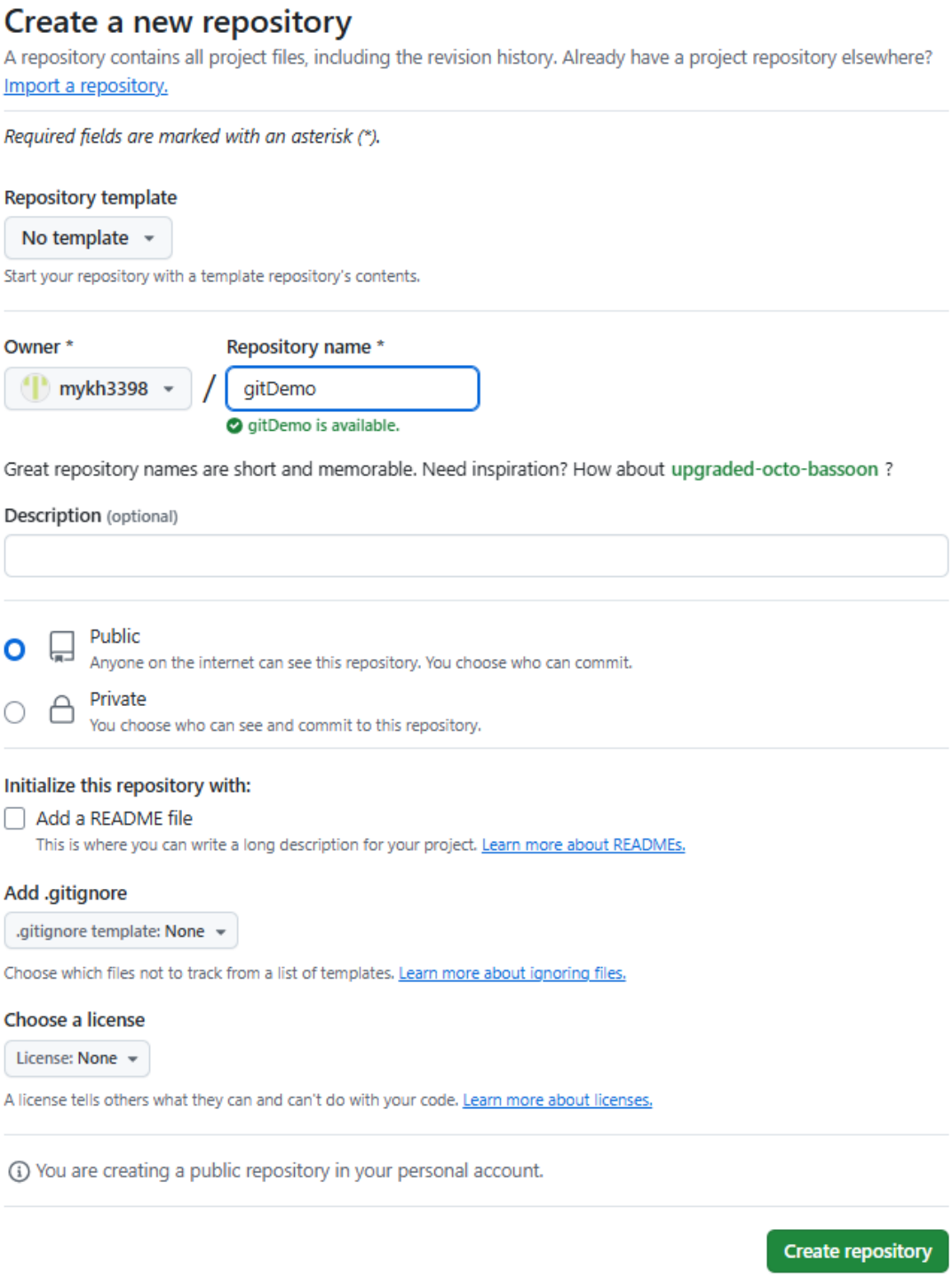
`git clone <url>`
Копіює репозиторій із зазначеної URL-адреси.

Завдання:

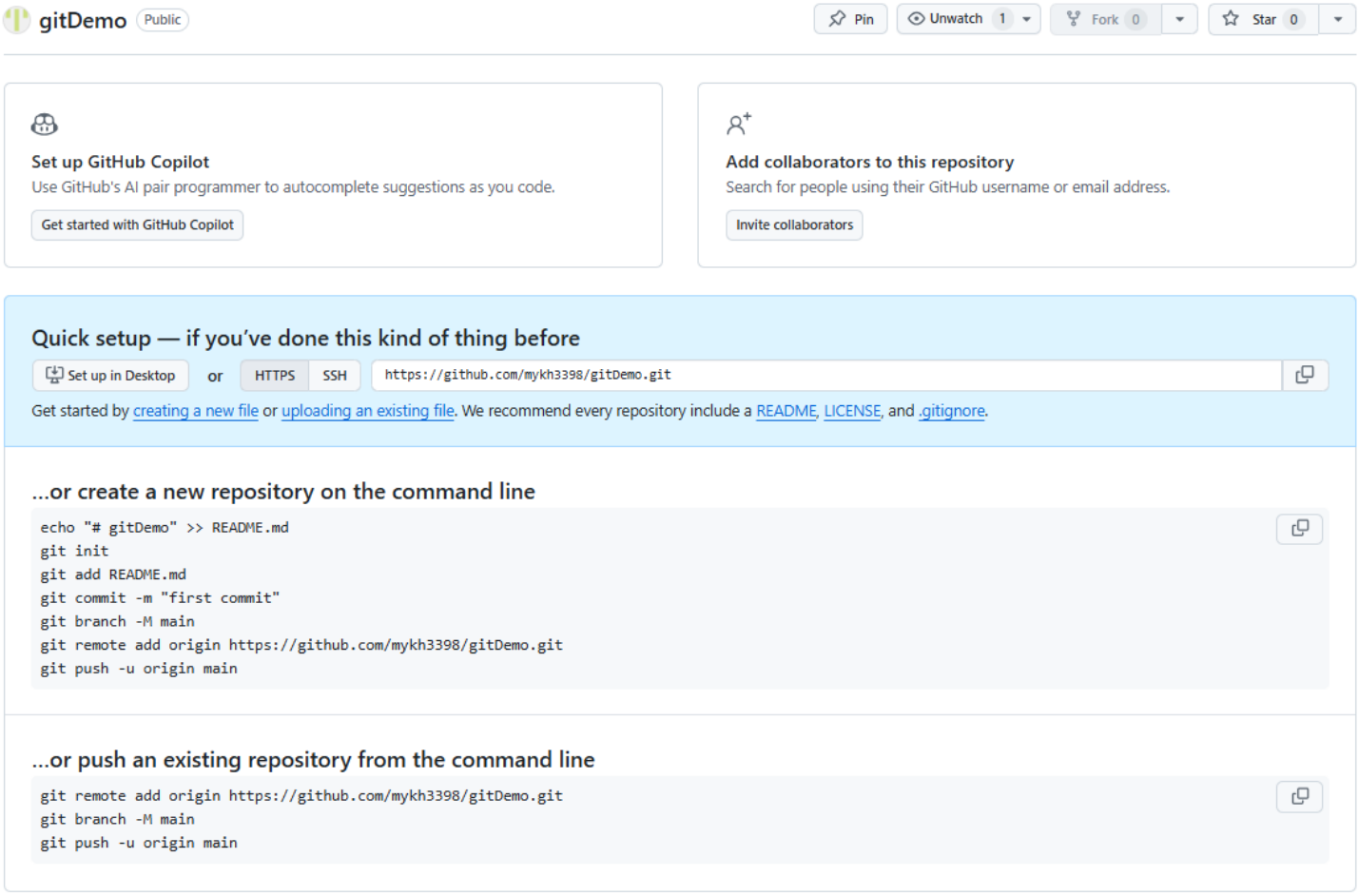
1. Створити віддалений репозиторій на GitHub
2. Клонувати віддалений репозиторій на локальну машину
3. Створити та перемкнутись на нову гілку з назвою feature
4. Створити, додати та закомітити в гілку файл з певним вмістом
5. Перемкнутись на гілку main та виконати cherry-pick коміту на гілці feature
6. Надіслати зміни на віддалений репозиторій

Порядок виконання лабораторної роботи:

Створення віддаленого репозиторію на GitHub:



Після натискання кнопки Create repository на обліковому записі Git створюється посилання на новий репозиторій після чого відкривається сторінка репозиторію:



Клонування та ініціалізація віддаленого репозиторію на локальну машину:

```
C:\Users\mickle\Desktop\practice>git clone https://github.com/mykh3398/gitDemo.git
Cloning into 'gitDemo'...
warning: You appear to have cloned an empty repository.

C:\Users\mickle\Desktop\practice>ls
gitDemo

C:\Users\mickle\Desktop\practice>cd gitDemo

C:\Users\mickle\Desktop\practice\gitDemo>echo INIT > init.txt

C:\Users\mickle\Desktop\practice\gitDemo>git add .

C:\Users\mickle\Desktop\practice\gitDemo>git commit -m "init commit"
[main (root-commit) 7fc1625] init commit
 1 file changed, 1 insertion(+)
 create mode 100644 init.txt
```

Створення гілки feature та файлу в ній:

```
C:\Users\mickle\Desktop\practice\gitDemo>git switch -c feature
Switched to a new branch 'feature'

C:\Users\mickle\Desktop\practice\gitDemo>echo FEATURE > feature.txt

C:\Users\mickle\Desktop\practice\gitDemo>git add .

C:\Users\mickle\Desktop\practice\gitDemo>git commit -m "added first feature on feature branch"
[feature 715acf6] added first feature on feature branch
 1 file changed, 1 insertion(+)
 create mode 100644 feature.txt

C:\Users\mickle\Desktop\practice\gitDemo>git log --all --oneline --graph
* 715acf6 (HEAD -> feature) added first feature on feature branch
* 7fc1625 (main) init commit
```

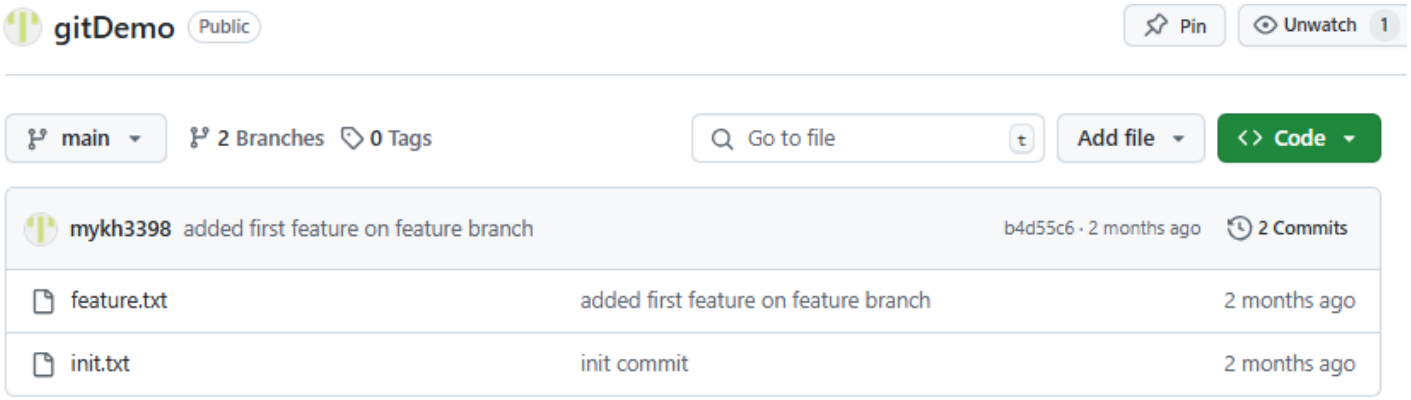
Перемикання на гілку main та cherry-pick коміту з гілки feature:

```
C:\Users\mickle\Desktop\practice\gitDemo>git switch -
Switched to branch 'main'
Your branch is based on 'origin/main', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)

C:\Users\mickle\Desktop\practice\gitDemo>git cherry-pick 715acf6
[main b4d55c6] added first feature on feature branch
 Date: Wed Oct 23 19:39:53 2024 +0300
 1 file changed, 1 insertion(+)
 create mode 100644 feature.txt
```

Надіслати зміни на віддалений репозиторій:

```
C:\Users\mickle\Desktop\practice\gitDemo>git push origin main
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), 492 bytes | 492.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/mykh3398/gitDemo.git
 * [new branch]      main -> main

C:\Users\mickle\Desktop\practice\gitDemo>git push origin feature
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 301 bytes | 301.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'feature' on GitHub by visiting:
remote:      https://github.com/mykh3398/gitDemo/pull/new/feature
remote:
To https://github.com/mykh3398/gitDemo.git
 * [new branch]      feature -> feature
```

Стан віддаленого репозиторію після пушу:



ВИСНОВОК

В результаті виконання лабораторної роботи я ознайомився з базовими командами та операціями в Git, такими як клонування віддаленого репозиторію, створення гілок, додавання файлів, коміти та перемикання між гілками.