МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Управління версіями програмних засобів проекту

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці»
спеціальності 121 Інженерія програмного забезпечення

Укладачі: В. О. Кузьміних, О. В. Коваль, Р. А. Тараненко

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2023

Рецензенти *Ланде Дмитро Володимирович, д.т.н., зав. кафедри інформаційної безпеки НН ФТІ, «КПІ ім. Ігоря Сікорського»*
Сенченко В'ячеслав Родіонович, к.т.н., с.н.с. відділу цифрових моделюючих систем, ІПРІ НАН України

Відповідальний редактор *Гагарін Олександр Олександрович, к.т.н., доц. кафедри ІПЗЕ, ННІ АТЕ, «КПІ ім. Ігоря Сікорського»*

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 6 від 30.03.2023 р.)
за поданням Вченої ради ННІ АТЕ
(протокол № 8 від 27.02.2023 р.)

Навчальний посібник присвячено розкриттю основ роботи з програмним інструментом для керування версіями програмного забезпечення – системами контроля версій (СКВ). В посібнику розгляну найбільш поширену систему контроля версій GitHub. Навчальний посібник «Управління версіями програмних засобів проекту» викладено у обсязі необхідному в підготовці здобувачів ступеня бакалавр за спеціальністю 121 «Інженерія програмного забезпечення». Навчальний посібник створено для забезпечення проведення лекційних занять. В посібнику подано загальний матеріал, словник термінів, опис існуючих програмних засобів СКВ, довідка Git. Викладений матеріал може бути корисним всім фахівцям з ІТ.

Реєстр. № НП 22/23-539. Обсяг 6,45 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

Зміст

ВСТУП

Складність управління сучасними проектами, що полягає в їх багатофакторності, досягла стану нових вимог – необхідності «всеохоплюючого представлення». Наслідки ускладнення – проблема підвищення витрат та зниження ефективності. ІТ як не від'ємна складова проекту потребує побудови ефективних та прагматичних робочих процесів, включаючи створення та супровід якісного ПЗ, що можна застосувати в будь-якому проекті. Лише в сфері ІТ існує багато вимог – ознак якісного ПЗ: надійність, стійкість, модульність, безпечність, продуктивність, масштабованість, зручність застосування, тестування та супровід, та багато інших. При цьому проект має включати: гарну документацію, включаючи журнали змін; деталізовані узгодження та стандарти; версіонування; автоматизовані тести та ін. Одним з зручних та ефективних засобів покращення робочих процесів стали системи верифікування версій, що на сьогодні надають гнучкі інструменти, які не диктують вам як потрібно реалізувати та документувати зміни, яку стратегію злиття використовувати, на яких етапах та стадіях фіксувати результати, які інструменти застосовувати, яким стандартам commit-ів слідувати та що потрібно рецензувати. Все це зорієнтовано вашими творчими підходами.

З точки зору досягнення якості, системи контролю версій (СКВ), як системи верифікування версій, стали не від'ємною складовою проектів та продовжують швидкими темпами свій розвиток, надаючи все нових можливостей підвищення ефективності та зменшення витрат. Відповідно оволодіння засобами СКВ є невід'ємною складовою оволодіння кваліфікаційними навичками фахівців у сфері інформаційних технологій та систем.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ ТА АКРОНІМІВ

ПЗ	програмне забезпечення
СКВ	система контролю версій
РСКВ	розподілена система контролю версій
ЦСКВ	централізована система контролю версій
CVS	Concurrent Versions System – централізована система управління версіями
DevOps	Development & operations — методологія автоматизації технологічних процесів збірки, налаштування та розгортання програмного забезпечення
DVCS	Distributed Version Control System - розподілена система контролю версій
FSFS	Fast Secure File System - безпечна розподілена файлова система в просторі користувача, побудована на основі FUSE і OpenSSL
IT	Information Technology – інформаційні технології
RCS	Revision Control System - система управління редакціями
SCCS	Source Code Control System – система контролю вихідного коду
SHA	Secure Hash Algorithm - алгоритм криптографічного хешування
SHA-1	Secure Hash Algorithm 1 — алгоритм криптографічного хешування - 1
SVN	Apache Subversion - сучасна система контролю версій. SVN – скорочення від Subversion
VCS	Version Control System - система контролю версій

РОЗДІЛ 1. ЗАГАЛЬНА ІНФОРМАЦІЯ ПРО СИСТЕМИ КОНТРОЛЮ ВЕРСІЙ

1.1. Що таке система контролю версій

Система контролю версій (СКВ від англ. Version Control System, VCS) - це спеціалізоване місце зберігання коду, система, що записує зміни у файл, чи набір файлів протягом часу і дозволяє повернутися пізніше до певної версії [1].

Інше формулювання звучить так:

СКВ - це програмне забезпечення, яке допомагає розробникам програм співпрацювати та вести повну історію своєї роботи. Воно може зберігати зміни файлів та модифікації вихідного коду. Щоразу, коли користувач вносить зміни в проект, СКВ приймає стан проекту та зберігає їх. Ці різні збережені стани проекту відомі як версії.

Системи **VCS** інколи називають інструментом **SCM** (source code management - управління початковим кодом) або **RCS** (revision control system - система управління редакціями).

Контроль версії - це практика відстеження змін програмного коду та управління ним [2]. Системи контролю версії — це програмні інструменти, які допомагають командам розробників керувати змінами в вихідному коді з течією часу. В умовах праці вони допомагають команді розробників працювати швидше і ефективно. Системи контролю версії найбільш корисні для команд **DevOps**, оскільки допомагають скоротити час розробки та збільшити кількість успішних розгортань.

Програмне забезпечення контролю версій відстежує всі зміни, що вносяться в код, у спеціальній базі даних. При виявленні помилки розробники можуть повернутися назад і порівняти з попередніми версіями коду для виправлення помилок, зводячи до мінімуму проблеми для всіх учасників команди.

Усі системи контролю версій по суті вирішують 4 задачі.

1. **Доступ до коду.** Вихідники коду зберігаються у віддаленому репозиторії (сховищі даних), куди звертаються розробники, щоб забрати актуальну версію файлів або внести зміни. Так вибудовується командна технологія.
2. **log-ування змін у коді.** Відстеження **commit-**ів (внесення змін до коду) допомагає знайти хто, що і коли змінював, вирішити конфлікти при модифікуванні одних і тих же файлів, відкотитися на будь-який попередній стан.

3. **Розгалуження розробки.** Програмісти паралельно ведуть розробку нового функціоналу у окремих гілках, не торкаючись працездатності старого.

4. **Підтримка версії продуктів.** При випуску оновлень програмних продуктів ми позначаємо релізні версії, наприклад, за допомогою **tag**-ів, щоб зафіксувати їх у цьому стані, для дебагу або ретроспективи.

На сьогодні існує розподіл СКВ на три головні категорії [3]:

- **Локальні;**
- **клієнт-серверні (централізовані);**
- **розподілені (децентралізовані).**

В свою чергу СКВ поділяють на **вільні (відкриті) та закриті**.

1.1.1. Локальні системи контролю версій

Кожен користувач як метод контролю версій застосовує копіювання файлів в окремі каталоги, але такий підхід є недосконалим, оскільки не дозволяє відслідковувати всі важливі зміни та відповідно уникнути різник помилок в роботі з файлами, що погіршує якість та продуктивність праці і особливо в складних проектах.

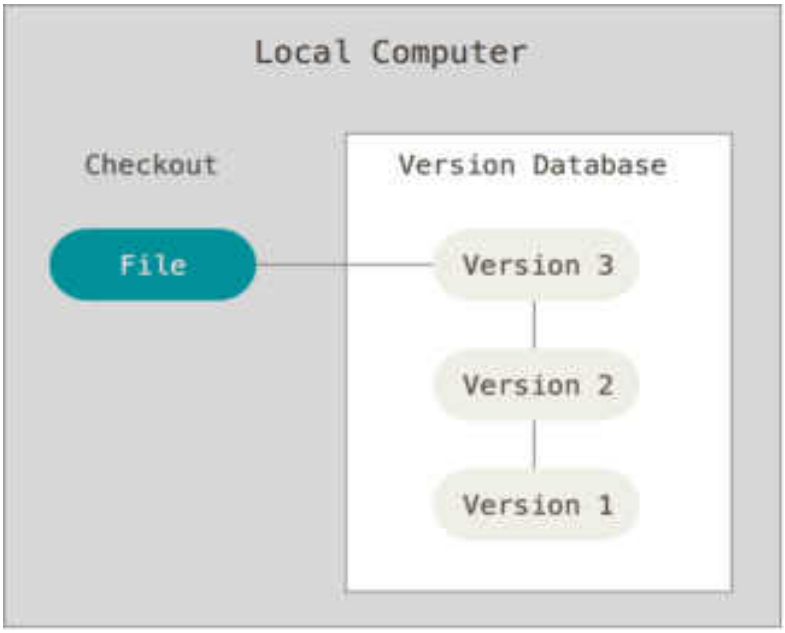


Рис. 1.1. Локальний контроль версій

Для того, щоб вирішити цю проблему, програмісти давно розробили локальні СКВ з простою базою даних, яка зберігає записи про всі зміни в файлах, здійснюючи тим самим контроль ревізій.

1.1.2. Централізовані системи контролю версій

Для вирішення проблеми необхідності тісної взаємодії з іншими розробниками проекту, були розроблені централізовані системи контролю версій (ЦСКВ). Такі системи, як **CVS, Subversion** і **Perforce**, використовують єдиний сервер, що містить всі версії файлів, і кілька клієнтів, які отримують файли з цього централізованого сховища. Застосування ЦСКВ було стандартом багато років.

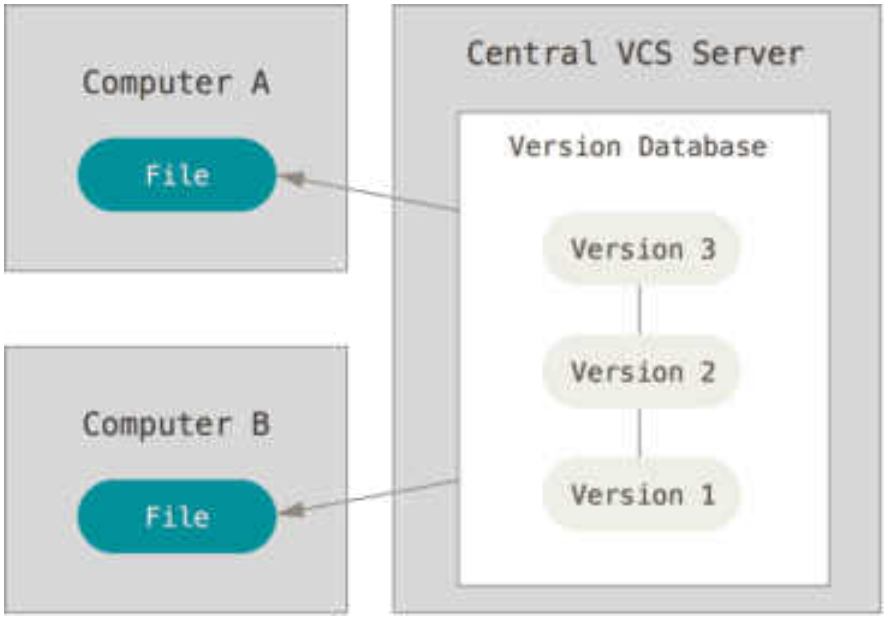


Рис. 1.2. Централізований контроль версій

Централізована СКВ має переваги над іншими СКВ, особливо над локальним. Наприклад, всі розробники проекту певною мірою знають, чим займається кожен із них. Адміністратори мають повний контроль над тим, хто і що може робити і також набагато простіше адмініструвати ЦСКВ, ніж оперувати локальними базами даних на кожному клієнті.

Мінуси – єдина точка відмови представлена централізованим сервером.

Якщо цей сервер вийде з ладу на годину, протягом цього часу ніхто не зможе використовувати контроль версій для збереження змін, над якими працює, а також ніхто не зможе обмінюватися цими змінами з іншими розробниками. Якщо жорсткий диск, на якому зберігається центральна БД, пошкоджений, а своєчасні бекапи відсутні, ви втратите весь проект разом з історією проекту, не враховуючи одиничних знімків репозиторію, які збереглися на локальних машинах розробників.

Локальні СКВ страждають від тієї самої проблеми: коли вся історія проекту зберігається в одному місці, ви ризикуєте втратити все.

1.1.3. Розподілені системи контролю версій

Розподілені СКВ (РСКВ) вже долають головні недоліки локальних та централізованих СКВ. У РСКВ (таких як **Git, Mercurial, Bazaar** або **Darcs**) клієнти не просто завантажують знімок всіх файлів (стан файлів на певний момент часу) - вони повністю копіюють репозиторій. У цьому випадку, якщо один із серверів, через який розробники обмінювалися даними, помре, будь-який репозиторій клієнта може бути скопійований на інший сервер для продовження роботи. Кожна копія репозиторію є повним бекапом всіх даних.

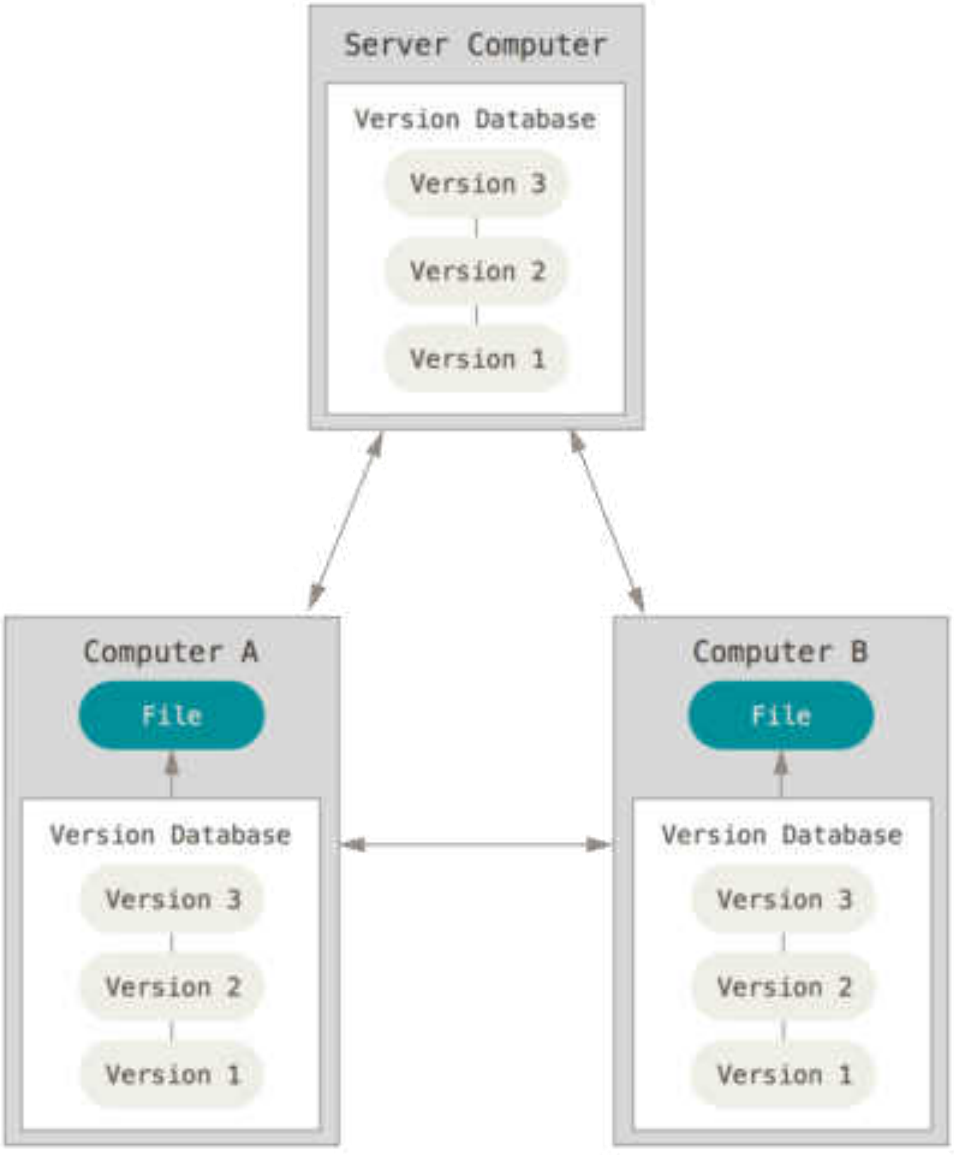


Рис. 1.3. Розподілений контроль версій

Ще однією з переваг РСКВ є те, що більшість з таких систем можуть одночасно взаємодіяти з декількома віддаленими репозиторіями. Завдяки цьому ви можете працювати з різними групами людей, застосовуючи різні підходи

одночасно в рамках одного проекту. Це дозволяє застосовувати відразу кілька підходів у розробці, наприклад, ієрархічні моделі, що не можливо в централізованих системах.

1.2. Історія розвитку систем контролю версій

Найвідоміші СКВ за технічними ознаками історично поділяють на три покоління [4].

Системи контролю версій (VCS) *першого покоління* відслідковували зміни в окремих файлах, а редагування підтримувалося лише локально та одним користувачем за один раз. Системи будувалися на припущенні, що всі користувачі заходитимуть по своїх облікових записах на той самий загальний вузол Unix.

У VCS *другого покоління* з'явилася підтримка мережі, що призвело до централізованих сховищ з офіційними версіями проектів. Це був значний прогрес, оскільки кілька користувачів могли одночасно працювати з кодом, роблячи **commit**-и в той самий центральний репозиторій. Однак для **commit**-ів був потрібний доступ до мережі.

Третє покоління складається з розподілених VCS, де копії репозиторію вважаються рівними, немає центрального репозиторію. Це відкриває шлях для **commit**-ів, гілок та злиття, які створюються локально без доступу до мережі та переміщуються в інші репозиторії за необхідності.

Перше покоління:

- SCCS (Source Code Control System)
- RCS (Revision Control System)

Друге покоління:

- CVS (Concurrent Versions System)
- SVN (Apache Subversion)

Третє покоління:

- Git
- Mercurial

Рис. 1.4. Дати появу різних поколінь СКВ

SCCS (Source Code Control System) - перше покоління

SCCS вважається однією з перших успішних систем керування версіями. Вона була розроблена в 1972 році Марком Рочкіндом з Bell Labs. Система написана на C та створена для відстеження версій вихідного файлу. Крім того, вона значно полегшила пошук джерел помилок у програмі. Базова архітектура та синтаксис SCCS дозволяють зрозуміти коріння сучасних інструментів VCS.

Архітектура Source Code Control System

Як і більшість сучасних систем, у SCCS є набір команд для роботи з версіями файлів:

- Внесення (check-in) файлів для відстеження історії у SCCS.
- Вилучення (check-out) конкретних версій файлів для ревю або компіляції.
- Вилучення конкретних версій для редагування.
- Внесення нових версій файлів разом із коментарями, які пояснюють зміни.
- Скасування змін, внесених до вилученого файлу.
- Основні розгалуження та злиття змін.
- Журнал змінення файлу.

RCS (Revision Control System) - перше покоління

RCS написана в 1982 році Уолтером Тихи мовою C як альтернатива системі SCCS, яка в той час не була відкритим програмним забезпеченням.

Архітектура Revision Control System

У RCS багато спільного зі своїм попередником, у тому числі:

- Веде версії окремо для кожного файлу.
- Зміни в кількох файлах не можна згрупувати на єдиний **commit**.
- Файли, що відстежуються, не можуть одночасно змінюватися кількома користувачами.
- Немає підтримки мережі.
- Версії кожного файлу, що відстежується, зберігаються у відповідному файлі історії.
- Розгалуження та об'єднання версій тільки для окремих файлів.

CVS (Concurrent Versions System) - друге покоління

CVS створена Діком Груном у 1986 році з метою додати до системи управління версіями підтримку мережі. Вона також написана на C і знаменує собою народження другого покоління інструментів VCS, завдяки яким географічно розосереджені команди розробників отримали можливість працювати над проектами разом.

Архітектура Concurrent Versions System

CVS - це фронтенд для RCS, у ньому з'явився новий набір команд для взаємодії з файлами у проекті, але під капотом використовується той самий формат файлу історії RCS та команди RCS. Вперше CVS дозволив кільком розробникам одночасно працювати з одними й тими самими файлами. Це реалізовано за допомогою моделі централізованого репозиторію. Перший крок – налаштування на віддаленому сервері централізованого репозиторію за допомогою CVS. Потім проекти можна імпортувати до репозиторію. Коли проект імпортується в CVS, кожен файл перетворюється на файл історії, і зберігається в центральній директорії - модулі. Репозиторій зазвичай знаходиться на віддаленому сервері, доступ до якого здійснюється через локальну мережу чи інтернет.

Розробник отримує копію модуля, який копіюється в робочий каталог на локальному комп'ютері. У цьому процесі жодні файли не блокуються, тому не має обмеження на кількість розробників, які можуть одночасно працювати з модулем. Розробники можуть змінювати свої файли і при необхідності фіксувати зміни

(робити **commit**). Якщо розробник фіксує зміни, інші розробники повинні оновити свої робочі копії за допомогою (зазвичай) автоматизованого процесу злиття перед фіксацією своїх змін. Іноді доводиться вирішувати вручну конфлікти злиття, перш ніж виконати **commit**. CVS також надає можливість створювати та об'єднувати гілки.

SVN (Subversion) - друге покоління

Subversion створена в 2000 році компанією **Collabnet Inc.,** а зараз підтримується **Apache Software Foundation**. Система написана на C та розроблена як більш надійне централізоване рішення, ніж CVS.

Архітектура Subversion

Як і **CVS, Subversion** використовує модель централізованого репозиторію. Видаленим користувачам потрібне мережне підключення для **commit** у центральний репозиторій.

Subversion представила функціональність атомарних **commit** із гарантією, що **commit** або повністю успішний, або повністю скасовується у разі проблеми. У **CVS** при неполадці через **commit** (наприклад, через збій мережі) репозиторій міг залишитися у пошкодженому та неузгодженому стані. Крім того, **commit** або версія в **Subversion** може включати кілька файлів і директорій. Це важливо, тому що дозволяє відстежувати набори пов'язаних змін разом як згрупований блок, а не окремо для кожного файлу, як у минулих системах.

В даний час, **Subversion** використовує файлову систему **FSFS (Fast Secure File System)**. Тут створюється база даних структурою файлів і каталогів, які відповідають файловій системі хосту. Унікальна особливість **FSFS** полягає в тому, що вона призначена для відстеження не лише файлів та каталогів, а й їх версій. Це файлова система із сприйняттям часу. Крім того, директорії є повноцінними об'єктами **Subversion**. У систему можна **commit**-ити порожні директорії, тоді як інші (навіть **Git**) не помічають їх.

Під час створення репозиторію **Subversion** у складі створюється (майже) порожня база даних файлів і папок. Створюється каталог ***db/revs***, де зберігається вся інформація відстеження версій для доданих (зафіксованих) файлів. Кожен **commit** (який може включати зміни в декількох файлах) зберігається в новому файлі в каталозі ***revs***, і йому надається ім'я з послідовним числовим

ідентифікатором, що починається з 1. При першому **commit** зберігається повний вміст файлу. Майбутні **commit** того самого файлу призведуть до збереження лише змін, які також називаються диффами (diff-и) або дельтами — для економії місця. Крім того, для зменшення розміру, дельти стискаються за допомогою алгоритмів стиснення **lz4** або **zlib**.

Така система працює лише до певного моменту. Хоча дельти заощаджують місце, але якщо їх дуже багато, то на операції йде чимало часу, тому що для відтворення поточного стану файлу необхідно обробити всі дельти. Тому **Subversion** зберігає до 1023 дельт на файл, а потім робить нову повну копію файлу. Це забезпечує хороший баланс зберігання та швидкості.

SVN не використовує звичайну систему розгалуження та тегів. Звичайний шаблон репозиторію **Subversion** містить три папки в корені:

trunk/

branches/

tags/

Директорія ***trunk/*** використовується для продакшн-версії проекту. Директорія ***branches/*** - для зберігання вкладених папок, що відповідають окремим гілкам. Директорія ***tags/*** - для зберігання тегів, що представляють певні (зазвичай значні) версії проекту.

Git - третє покоління

Систему **Git** розробив у 2005 році Лінус Торвальдс (творець **Linux**). Вона написана переважно на **C** у поєднанні з деякими сценаріями командного рядка. Відрізняється від **VCS** за функціями, гнучкістю та швидкістю. Торвальдс спочатку написав систему для кодової бази Linux, але згодом її сфера використання розширилася, і сьогодні це найпопулярніша у світі система управління версіями.

Архітектура Git

Git є розподіленою системою. Центрального репозиторію не існує: всі копії створюються рівними, що різко відрізняється від **VCS** другого покоління, де робота заснована на додаванні та вилученні файлів із центрального репозиторію. Це означає, що розробники можуть обмінюватись змінами один з одним безпосередньо перед об'єднанням своїх змін в офіційну гілку.

Крім того, розробники можуть вносити свої зміни до локальної копії репозиторію без відома інших репозиторіїв. Це дозволяє **commit**-и без підключення до мережі або інтернету. Розробники можуть працювати локально в автономному режимі, доки не будуть готові поділитися своєю роботою з іншими. У цей момент зміни відправляються до інших репозиторіїв для перевірки, тестування або розгортання.

Коли файл додається для відстеження **Git**, він стискається за допомогою алгоритму стиснення ***zlib***. Результат хешується за допомогою хеш-функції **SHA-1**. Це дає унікальний хеш, який відповідає конкретному вмісту в цьому файлі. **Git** зберігає його в базі об'єктів, що знаходиться в прихованій папці ***.git/objects***. Ім'я файлу – це згенерований хеш, а файл містить стислий контент. Дані файли називаються **blob-ами** і створюються щоразу при додаванні до репозиторію нового файлу (або зміненої версії існуючого файлу).

Git реалізує проміжний індекс (***staging index***), який виступає як проміжна область для змін, які готуються до **commit**. У міру підготовки нових змін на їхній стислий вміст посилаються в спеціальному індексному файлі, який набуває форми об'єкта дерево. Дерево – це об'єкт **Git**, який пов'язує **blob-и** з їхніми реальними іменами файлів, дозволами на доступ до файлів та посиланнями на інші дерева і таким чином представляє стан певного набору файлів та каталогів. Коли всі відповідні зміни підготовлені для **commit**, індексне дерево можна зафіксувати у репозиторії, який створює об'єкт **commit** у базі даних об'єктів **Git. Commit** посилається на дерево заголовків для конкретної версії, а також на автора **commit**, адресу електронної пошти, дату та повідомлення **commit**. Кожен **commit** також зберігає посилання на свій батьківський **commit**(-и), і так згодом створюється історія розвитку проекту.

Як уже згадувалося, всі об'єкти **Git** – **blob-и, tree (дерева) та commit**-и – стискуються, хешуються та зберігаються в базі даних об'єктів на основі їх хешів. Вони називаються вільними об'єктами (**Loose Objects**). Тут не використовуються ніякі **diff**-и для економії місця, що робить **Git** дуже швидким, оскільки повний вміст кожної версії файлу є вільним об'єктом. Однак деякі операції, такі як передача **commit**-ів у віддалений репозиторій, зберігання дуже великої кількості об'єктів або ручний запуск команди складання сміття **Git** викликають переупаковку об'єктів у пакетні файли. У процесі упаковки обчислюються зворотні **diff-**и, які стискаються для виключення надмірності та зменшення розміру. В результаті створюються

файли ***.pack*** з вмістом об'єкта, а для кожного з них створюється файл ***.idx*** (або індекс) з посиланням на упаковані об'єкти та їхнє розташування в пакетному файлі.

Коли гілки переміщуються у віддалені сховища або витягуються з них, мережі передаються ці пакетні файли. Під час витягування або вилучення гілок файли пакета розпаковуються для створення вільних об'єктів у репозиторії об'єктів.

Mercurial - третє покоління

Mercurial створено в 2005 році Меттом Макколлом і написано на **Python**. Він також розроблений для хостингу кодової бази **Linux**, але для цього завдання в результаті вибрали **Git**. Це друга за популярністю система управління версіями, хоча вона використовується набагато рідше.

Архітектура Mercurial

Mercurial - теж розподілена система, яка дозволяє будь-якій кількості розробників працювати зі своєю копією проекту незалежно від інших. **Mercurial** використовує багато з тих же технологій, що і **Git**, у тому числі стиснення та хешування **SHA-1**, але робить це інакше.

Коли новий файл фіксується для відстеження в **Mercurial**, для нього створюється відповідний файл **revlog** у прихованому каталозі ***.hg/store/data/.*** Можете розглядати файл **revlog** (журнал змін) як модернізовану версію файлів історії у старих **VCS**, таких як **CVS, RCS** та **SCCS**. На відміну від **Git**, який створює новий блог для кожної версії кожного підготовленого файлу, **Mercurial** просто створює новий запис у **revlog** для цього файлу. Для економії місця кожен новий запис містить лише дельту від попередньої версії. Коли досягається граничне число дельт, знову зберігається повний знімок файлу. Це скорочує час пошуку для обробки великої кількості дельт для відновлення певної версії файлу.

Кожен **revlog** називається відповідно до файлу, який він відстежує, але з розширенням **.i** або **.d**. Файли **.d** містять стислу дельту. Файли **.i** використовуються як індекси для швидкого відстеження різних версій усередині файлів **.d**. Для невеликих файлів із малою кількістю змін індекси та вміст зберігаються всередині файлів **.i**. Записи файлу **revlog** стискаються для продуктивності та хешуються для ідентифікації. Ці хеш-значення називаються **nodeid.**

При кожному **commit Mercurial** відстежує всі версії файлів цього **commit** у так званому маніфесті. Маніфест також є файлом **revlog** - у ньому зберігаються

записи, що відповідають певним станам репозиторію. Замість окремого вмісту файлу, як **revlog,** маніфест зберігає список імен файлів та **nodeids**, які визначають, які версії файлу існують у версії проекту. Ці записи маніфесту також стискуються та хешуються. Хеш-значення теж називаються **nodeid**.

Нарешті **Mercurial** використовує ще один тип **revlog**, який називається **changelog**, тобто журнал змін. Це список записів, які пов'язують кожен **commit** із такою інформацією:

- **nodeid** маніфесту: повний набір версій файлу, які існують у певний момент часу;
- один або два **nodeid** для батьківських **commit**: це дозволяє **Mercurial** будувати тимчасову шкалу або гілку історії проекту. Залежно від типу **commit** (звичайний або злиття) зберігається один або два батьківські **ID** (ідентифікатор).

Автор **commit.**

Дата **commit.**

Повідомлення **commit.**

Кожен запис **changelog** також генерує хеш, відомий як його ***nodeid***.

Питання до розділу 1

1. Що таке системи контролю версій?
2. Які задачі виконують системи контролю версій?
3. Які категорії систем контролю версій ви знаєте? Опишіть їх.
4. Які покоління систем контролю версій ви знаєте? Опишіть їх.
5. Що ви знаєте про третє покоління систем контролю версій?

РОЗДІЛ 2. ПОБУДОВА GIT

Git, що відноситься до розподілених **СКВ** (англ. **DVCS**), на відміну від традиційних, таких як **Subversion та Perforce**, не дивлячись на схожий інтерфейс користувача, зберігає та використовує інформацію зовсім інакше.

2.1. Два різні підходи в роботі систем контролю версій

1. Концептуально, більшість СКВ зберігають інформацію як список змін у файлах (рис. 2.1.). Ці системи (**CVS, Subversion, Perforce, Bazaar** і т. ін.) представляють збережену інформацію як набір файлів і змін, зроблених у кожному файлі, за часом (зазвичай це називають контролем версій, заснованим на відмінностях).

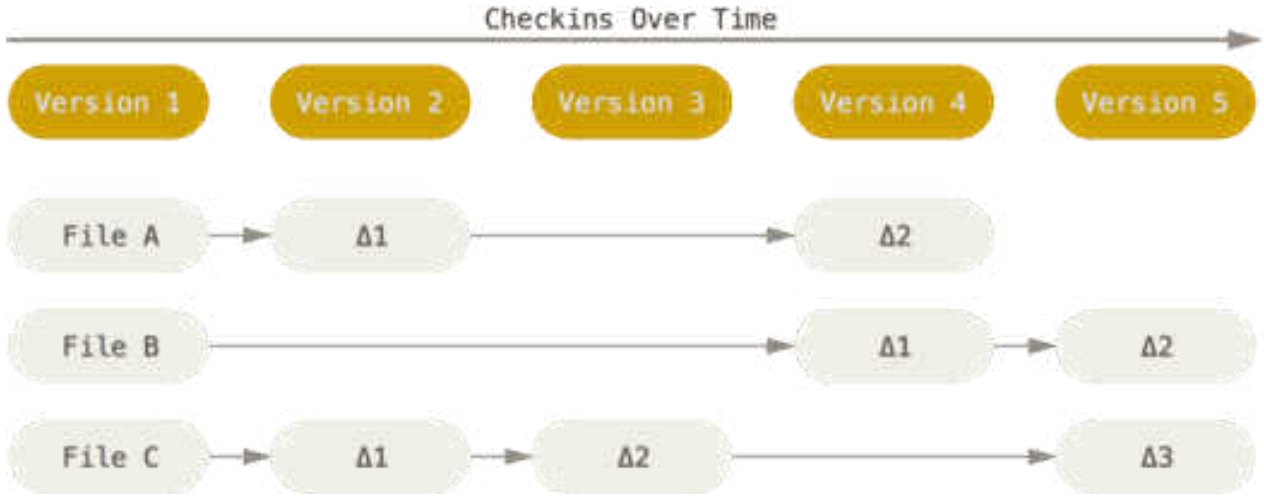


Рис. 2.1. Зберігання даних як набору змін щодо початкової версії кожного файлу

2. **Git** не зберігає та не обробляє дані вище вказаним способом. Натомість, підхід **Git** до зберігання даних більше схожий на набір знімків мініатюрної файлової системи [1]. Щоразу, коли ви виконуєте параметр «**commit**», тобто зберігаєте стан свого проекту в Git, система запам'ятовує, як виглядає кожен файл у цей момент, і зберігає посилання на цей знімок. Для збільшення ефективності, якщо файли не були змінені, **Git** не запам'ятовує ці файли знову, а лише створює посилання на попередню версію файлу, який вже збережений. Git представляє свої дані, як, скажімо, потік знімків.

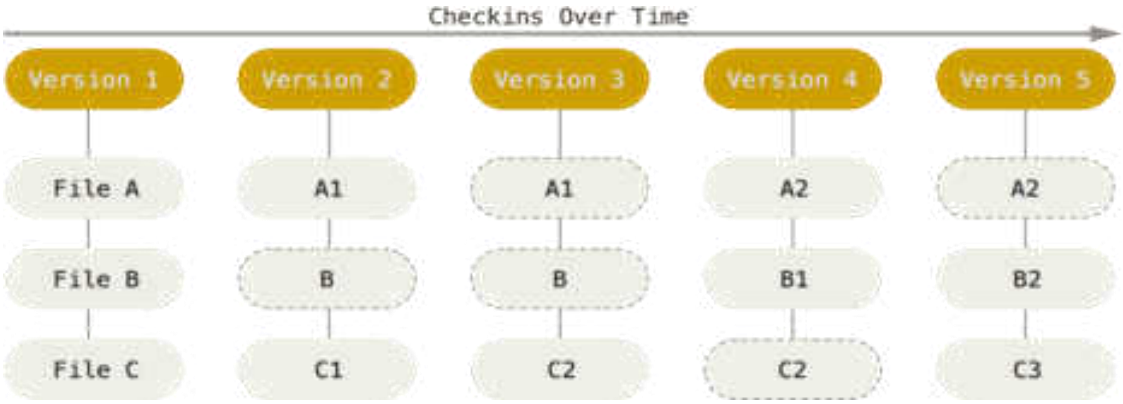


Рис. 2.2. Зберігання даних як знімків проекту у часі

Це дуже важлива відмінність між **Git** і від будь-якої іншої СКВ. **Git** переосмислює практично всі аспекти контролю версій, скопійованих з попереднього покоління більшістю інших систем. Це робить **Git** більш схожим на мініатюрну файлову систему з напрочуд потужними утилітами, надбудованими над нею, ніж просто на СКВ. Такий підхід надає суттєвих переваг підходам використаним в **Git**.

2.2 Виконання операцій локально

Загальна проблема у роботі більшості СКВ - необхідність постійної підтримки мережевого зв’язку, що призводить до постійних затримок. З метою подолання цієї проблеми більшість операцій **Git** виконує з локальним розташуванням файлів і ресурсів – системі не потрібна жодна інформація з інших комп'ютерів у вашій мережі. Оскільки вся історія проекту зберігається прямо на вашому локальному диску, більшість операцій здаються мало не миттєвими.

Для прикладу, щоб переглянути історію проекту, **Git** не потрібно з'єднуватися з сервером для її отримання та відображення — система просто зчитує дані безпосередньо з локальної бази даних. Це означає, що ви побачите історію проекту практично миттєво. Якщо вам необхідно переглянути зміни, зроблені між поточною версією файлу та версією, створеною місяць тому, **Git** може знайти файл місячної давності та локально обчислити зміни, замість того, щоб вимагати віддалений сервер виконати цю операцію, або замість отримання старої версії файлу з сервера та виконання операції локально.

Це також означає, що є лише невелика кількість дій, які ви не зможете виконати, якщо ви знаходитесь офф-лайн або не маєте доступу до VPN на даний момент. Ви можете без будь-яких проблем працювати з вашою локальною копією:

коли буде можливість підключитися до мережі, всі зміни можна буде синхронізувати.

Домогтися такої поведінки в багатьох інших системах або дуже складно, або зовсім неможливо. У **Perforce**, наприклад, якщо ви не підключені до сервера, вам не вдасться зробити багато чого; у **Subversion** і **CVS** ви можете редагувати файли, але ви не зможете зберегти зміни до бази даних (бо ви не підключені до БД). Все це може здатися не таким вже й значущим, але ви здивуєтеся, яке велике значення це може мати на практиці.

2.3 Цілісність Git

Цілісність у **Git** забезпечується обчисленням хеш-суми, і потім відбувається збереження [1]. Всі подальші звернення до збережених об'єктів відбувається за цією хеш-сумою, що виключає будь-які випадкові зміни вмісту файлів чи каталогів. Ця функціональність вбудована в **Git** на низькому рівні і є невід'ємною частиною його філософії. Ви не втратите інформацію під час її передачі та не отримаєте пошкоджений файл без відома **Git**.

SHA-1 хеш - механізм, яким користується **Git** для обчислення хеш-сум, становить рядок довжиною 40 шістнадцяткових символів (0-9 і a-f). Він обчислюється на основі вмісту файлу або структури каталогу.

Зразок **SHA-1** хеш:

24b9da6552252987aa493b52f8696cd6d3b00373

Git зберігає всі об'єкти в свою базу даних не за ім’ям, а за хеш-сумою вмісту об'єкта.

2.4 Головна мета Git - добавляти дані

Робота **Git** практично заснована на збереженні будь-яких дій додаванням нових даних в базу **Git**. Дуже складно змусити систему видалити дані або зробити щось, що не можна згодом скасувати. Ви можете втратити або зіпсувати свої зміни, поки вони не зафіксовані, але після того, як ви зафіксуєте знімок у **Git**, буде дуже складно щось втратити, особливо, якщо ви регулярно синхронізуєте свою базу з іншим репозиторієм.

Вищевказані особливості **Git** розкривають множину переваг, суттєво полегшуючи роботу з проектами, дозволяючи серйозно експериментувати не боячись серйозних проблем.

2.5 Три стани Git

У **Git –** файли можуть бути в трьох основних станах [1]:

- **змінений (modified),**
- **індексований (staged),**
- **зафіксований (committed).**

До **змінених** відносяться файли, які змінилися, але ще не були зафіксовані.

Індексований - це змінений файл у його поточній версії, позначений для включення в наступний «**commit**».

Зафіксований означає, що файл збережений у вашій локальній базі.

2.6 Три робочі секції проекту Git.

Проект **Git** поділяється на три робочі секції [1]**:**

- **робоча копія (*working tree*),**
- **область індексування (*staging area*),**
- **каталог Git (*Git directory*).**

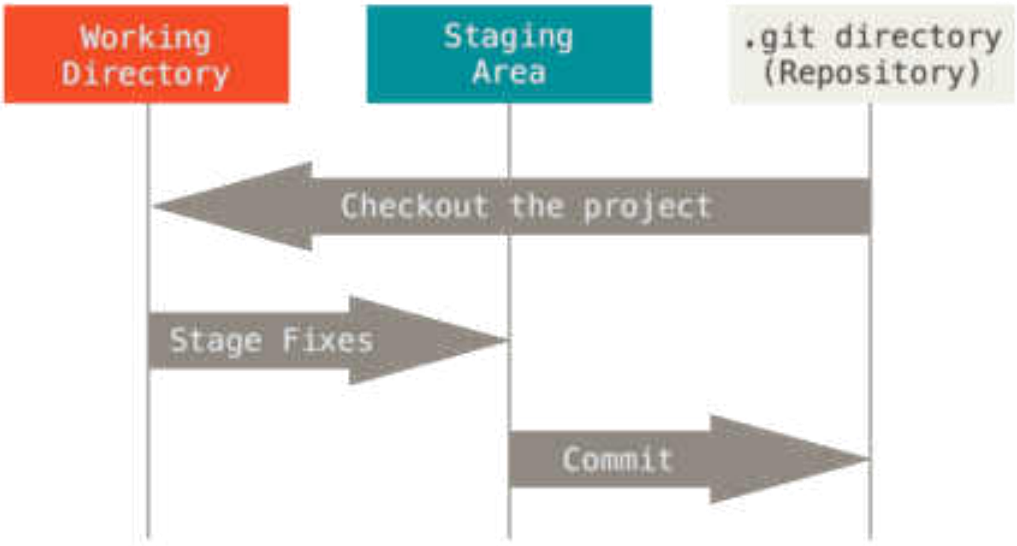


Рис. 2.3. Робоча копія, область індексування та каталог **Git**

Робоча копія є знімком однієї версії проекту. Ці файли виймаються зі стиснутої бази даних у каталозі **Git** і поміщаються на диск, щоб їх можна було використовувати чи редагувати.

Область індексування - це файл(-и), що зазвичай знаходиться в каталозі **Git**, в ньому міститься інформація про те, що потрапить в наступний «**commit**». Її технічна назва мовою **Git — індекс**, але фраза **«область індексування»** також працює.

Каталог Git - це місце, де **Git** зберігає метадані і базу об'єктів вашого проекту. Це найважливіша частина **Git** і це частина, яка копіюється при клонуванні репозиторію з іншого комп'ютера.

2.7 Базовий підхід у роботі з Git

Базовий підхід у роботі з **Git**:

1. Ви змінюєте файли вашої робочої копії.
2. Вибірково додаєте до індексу лише ті зміни, які мають потрапити до наступного «**commit**», додаючи тим самим знімки лише цих змін до індексу.
3. Коли ви робите **«commit»**, використовуються файли з індексу як є, і цей знімок зберігається у вашому каталозі **Git**.

Якщо певна версія файлу є у каталозі **Git**, то ця версія вважається **зафіксованою (committed).** Якщо файл був змінений і доданий до індексу, значить, він **індексований (staged).** І якщо файл був змінений з моменту останнього розпаковування з репозиторію, але не був доданий до індексу, він вважається **зміненим** (**modified**).

Питання до розділу 2

1. Як побудована Git?
2. Які підходи в побудові роботи СКВ ви знаєте? Яка відмінність Git від інших СКВ?
3. Як забезпечується цілісність Git?
4. Які стани закладено в роботу Git? Опишіть їх.
5. Опишіть базовий підхід у роботі Git?

РОЗДІЛ 3. ДИРЕКТОРІЯ GIT ТА РОБОЧА ДИРЕКТОРІЯ. ОБ'ЄКТНА МОДЕЛЬ GIT

3.1. Директорія Git та робоча директорія

В **“директорії git”** зберігається вся історія **Git** та мета-інформація вашого проекту - включаючи всі об'єкти (**commit**, дерева, **blob**-и, **tag-**и), всі покажчики на різні гілки та багато іншого [5].

На кожен проект є тільки одна директорія **Git** (на відміну SVN або CVS, де вона в кожній піддиректорії), і це директорія (за замовчуванням, але не обов'язково) **“.git”** в корені вашого проекту. Якщо ви подивитеся на вміст цієї директорії, то побачите всі ваші важливі файли (можуть бути і інші файли/директорії):

$>tree -L 1

.

|-- HEAD # вказівник на вашу активну гілку

|-- config # ваші персональні налаштування

|-- description # опис проекту

|-- hooks/ # pre/post action hooks (скрипти (надалі хуки) які можуть викликатися git командами)

|-- index # індексний файл

|-- logs/ # історія гілок проекту (де вони розташовані)

*|-- objects/ # ваші об’єкти (**commit**, дерева, **blob**-и, **tag-**и)*

`-- refs/ # вказівники на ваші гілки розробки

Робоча директорія це просто тимчасове місце, де ви можете модифікувати файли, а потім виконати **commit**.

3.2. Об'єктна модель Git

3.2.1. .SHA

Історія проекту зберігається у особливо організованих файлах, що посилаються один на одного за допомогою 40-значного "імені об'єкта" **SHA1** – криптографічної хеш-функції [6].

Це дозволяє:

- **Git** може швидко визначити чи ідентичні два об'єкти чи ні, просто порівнюючи їх імена.

- Так як імена об'єктів обчислюються однаково у всіх репозиторіях, то об'єкти з однаковим вмістом у двох репозиторіях завжди зберігаються під однаковими іменами.

- **Git** може знаходити помилки, коли читає об'єкт, для цього потрібно просто порівняти хеш значення вмісту об'єкта з його ім'ям.

3.2.2. Об’єкти

Основою побудови **Git** є чотири типи об’єктів - **"блоб" (blob)**, **"дерево" (tree)**, "**commit**", та **"таг" (tag)**. Кожен об’єкт має три частини – **тип, розмір та зміст**. Таку структуру порівнюють з надбудовою над традиційною файловою системою.

"blob" використовується щоб зберігати вміст файлу - зазвичай це просто файл.

"tree" це щось на зразок директорії - воно посилається на групу інших дерев та/або **blob-**ів (тобто файлів та директорій).

"Commit" вказує на окреме дерево, за суттю відзначає дерево фіксуючи в історії яким чином воно виглядає в момент виконання **commit**. Він містить мета-інформацію фіксуючи момент часу та автора змін, внесених з останнього **commit**, покажчик на попередній **commit** тощо.

"tag" це спосіб маркувати певним чином певний **commit**. Зазвичай це використовується щоб маркувати (по суті дати якесь ім'я, що легко запам'ятовується) певні **commit** є специфічними, щоб згодом було легше їх знайти.

3.2.3. Об’єкт типу «Блоб» (blob).

Blob зберігає зміст файлу.

Рис. 3.1. Представлення об’єкту типу Блоб

Об'єкт **"blob"** це порція бінарних даних. **Blob** ні на що не посилається у нього немає жодних атрибутів, не має навіть імені файлу.

Оскільки **blob** повністю визначається його власним вмістом, то якщо два файли в директорії або навіть у різних версіях репозиторію мають однаковий вміст, вони будуть розділяти той самий **blob** об'єкт. Об'єкт повністю залежить від розташування в дереві каталогів, і перейменування файлу не змінить об'єкт із яким файл пов'язаний.

Зміст **blob** можливо переглянути командою **git show**. Наприклад:

$ git show 6ff87c4664

Note that the only valid version of the GPL as far as this project is concerned is _this_ particular version of the license (ie v2, not v2.2 or v3.x or whatever), unless explicitly otherwise stated.

...

3.2.4. Об’єкт «Дерево» (Tree)

Tree це простий об'єкт який містить у собі групу покажчиків на **blob**-и та інші дерева - представляє вміст директорій чи піддиректорій.

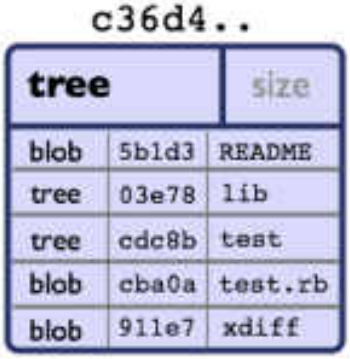


Рис. 3.2. Представлення об’єкту дерево

Щоб оглянути зміст дерева необхідно використати команду **git ls-tree (**можливо використовувати **git show**, але вона надає менший обсяг інформації про зміст дерева). Наприклад за відомим значенням **SHA** дерева:

$ git ls-tree fb3a8bdd0ce

100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c .gitignore

100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d .mailmap

100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3 COPYING

040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745 Documentation

100755 blob 3c0032cec592a765692234f1cba47dfdcc3a9200 GIT-VERSION-GEN

100644 blob 289b046a443c0647624607d471289b2c7dcd470b INSTALL

100644 blob 4eb463797adc693dc168b926b6932ff53f17d0b1 Makefile

100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52 README

...

Об'єкт **tree** містить список записів, що складаються з виду, типу об'єкту, значення **SHA1**, і імені відповідно. Записи відсортовані на ім'я. Так виглядає вміст однієї директорії **tree**.

Посилання на об'єкт у дереві може бути як **blob** (файлом по суті), так і **tree** (піддиректорією). Оскільки імена всіх об'єктів, **tree** та **blob**, збігаються з **SHA** хеш-значенням їхнього вмісту, то **SHA** значення двох **tree** будуть ідентичні тільки якщо їх вміст (включаючи, рекурсивно, вміст усіх піддиректорій) ідентичний.

3.2.5. Об’єкт commit

Об'єкт "**commit**" пов'язує фізичний стан **tree** з описом того, яким чином ми прийшли до нього і чому.

Рис. 3.3. Представлення об’єкту **commit**

Щоб дослідити **commit** можна застосувати параметр **--pretty=raw** з **git show** або **git log**. Наприклад:

$ git show -s --pretty=raw 2be7fcb476

commit 2be7fcb4764f2dbcee52635b91fedb1b3dcf7ab4

tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf

parent 257a84d9d02e90447b149af58b271c19405edb6a

author Dave Watson <dwatson@mimvista.com> 1187576872 -0400

committer Junio C Hamano <gitster@pobox.com> 1187591163 -0700

Fix misspelling of 'suppress' in docs

Signed-off-by: Junio C Hamano <gitster@pobox.com>

Commit визначається наступними головними значеннями:

- **Дерево (tree)**: ім'я **SHA1** об'єкта **tree** (як визначено нижче), що представляє вміст директорії в певний момент часу.
- **Батько(и)**: **SHA1** ім'я деякого числа **commit**, які являють собою попередній крок(и) в історії проекту. Приклад вище має одного з батьків; хоча **commit**-и злиття можуть мати більше одного батька. **Commit** без батьків називається **"root (кореневий)" commit**, і є початковим станом проекту. Кожен проект повинен мати принаймні один кореневий **commit**. Проект може також мати багато коренів, проте це не загальний випадок (і не обов'язково хороша ідея).
- **Автор:** Ім'я розробника відповідального за ці зміни, разом із датою.
- **Commit**-ер: ім'я розробника, який створив цей **commit**, разом з датою цієї події. Воно (ім'я) може відрізнятись від імені автора; наприклад у випадку, якщо автор написав патч і відправив його електронною поштою іншому розробнику який наклав патч і виконав **commit**.
- **Коментар:** описує цей **commit**.

3.2.6. Об’єктна модель

Розглянемо приклад поєднання 3х вище розглянутих головних об’єкти (**blob**, **tree** та **commit**).

Для проекту директорії зразкової структури:

```
$>tree
.
|-- README
`-- lib
    |-- inc
    |   `-- tricks.rb
    `-- mylib.rb

2 directories, 3 files
```

Рис. 3.4. Директорія зразкової структури

За умови, якщо виконано **commit** в репозиторій **Git** отримаєм наступну відображення.

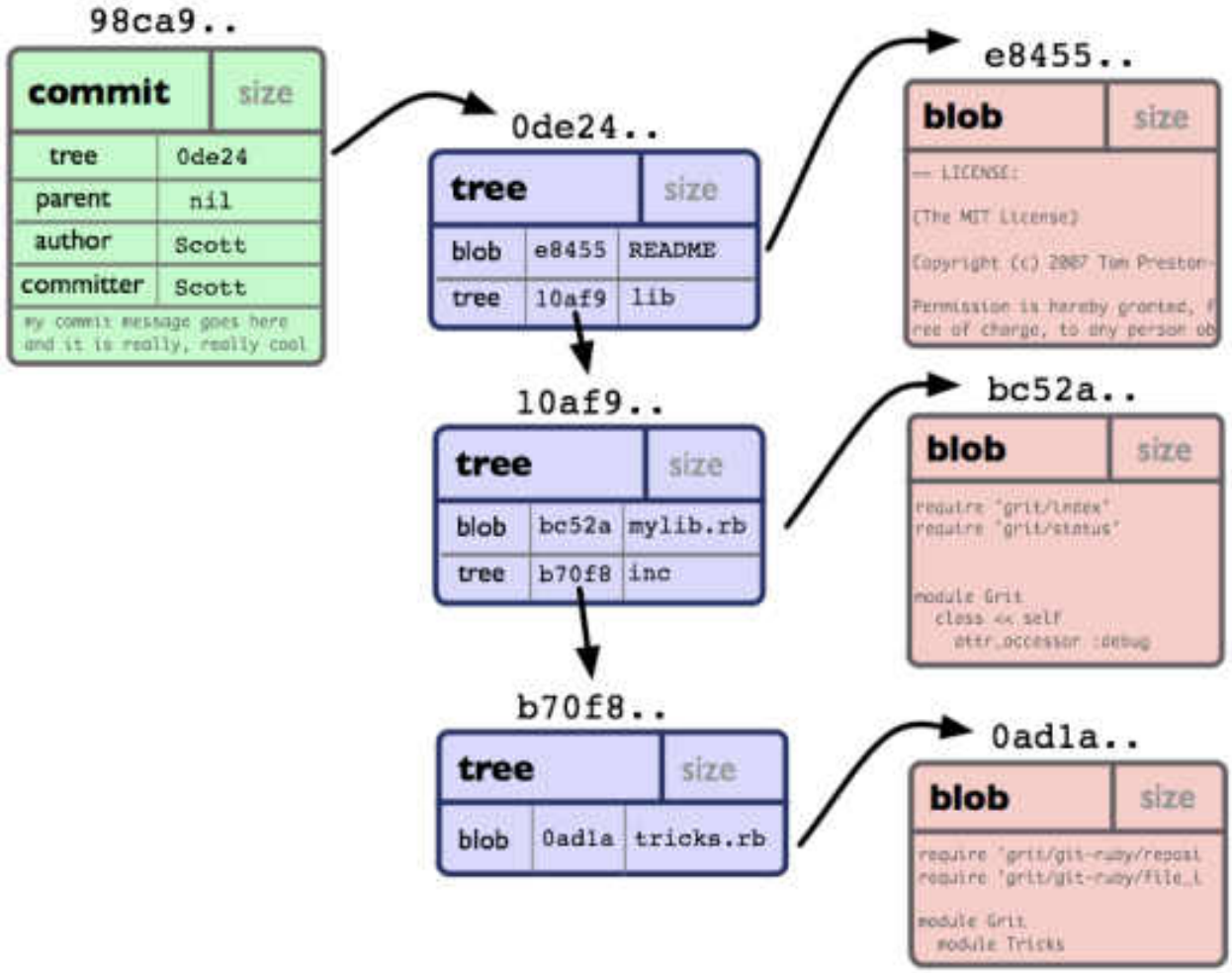


Рис. 3.5. Представлення зразка директорії після **commit**

3.2.7. Об’єкт «таг» (tag)

Рис. 3.6. Представлення об’єкту таг

Об'єкт **tag** містить **ім'я об'єкту** (називається просто - “об'єкт”), **тип об'єкту**, **ім'я tag**, або розробника ("таггер") який створив **tag**, і **повідомлення**, яке може містити підпис. Оглянути об’єкт **tag** можна командою **git cat-file**. Наприклад:

$ git cat-file tag v1.5.0
object 437b1b20df4b356c9342dac8d38849f24ef44f27
type commit
tag v1.5.0
tagger Junio C Haemano <junkio@cox.net> 1171411200 +0000

GIT 1.5.0
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBF0lGqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCd G4ui
nLE/L9aUXdWeTFPron96DLA=
=2E+0
-----END PGP SIGNATURE-----

Питання до розділу 3

1. Як організована директорія Git?
2. Об’єктна модель Git? Опишіть її.
3. Опишіть об’єкт blob.
4. Опишіть об’єкт tree.
5. Опишіть об’єкт commit.
6. Опишіть об’єкт tag.

РОЗДІЛ 4. РОЗГАЛУЖЕННЯ В GIT

Можливості проектів розширюють розгалуження, що підтримують більшість СКВ. **Git** заохочує процес роботи, при якому розгалуження та злиття виконується часто: операція створення гілки виконується майже миттєво, перемикання між гілками, як правило, також швидко. Розгалуження, як функціональність надає унікальний та потужний інструмент, який може суттєво підвищити ефективність змінити звичний процес розробки [1].

Раніш було розглянуто, що **Git** зберігає дані як послідовність знімків.

Об'єкт фіксації **Git** зберігає **вказівник на знімок змісту**, який ви додали. Цей об'єкт також містить:

- ім'я,
- поштову адресу автора,
- набране вами повідомлення,
- вказівники на фіксацію або фіксації, що були прямо до цієї фіксації (батько чи батьки): нуль для першої фіксації, одна фіксація для нормальної фіксації, та декілька фіксацій для фіксацій, що вони є результатом злиття двох чи більше гілок.

Приклад: припустимо, що у вас є тека з трьома файлами, які ви додали та зафіксували. Додання файлів обчислює контрольну суму для кожного (**SHA-1** хеш про котрий ми згадували раніш), зберігає версію файлу в сховищі **Git** (**Git** називає їх **blob-ами**), та додає їх контрольні суми до області додавання:

$ git add README test.rb LICENSE

$ git commit -m 'The initial commit of my project'

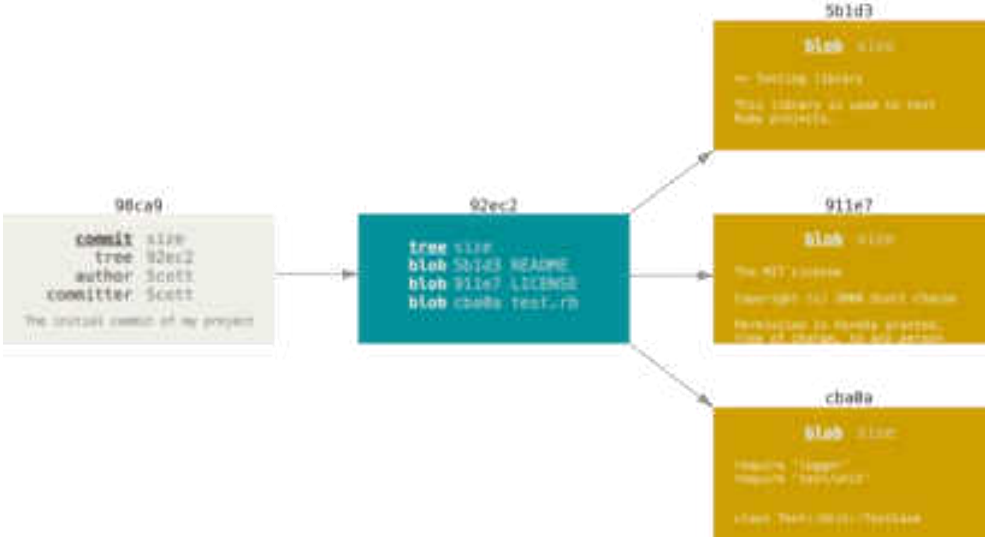


Рис. 4.1. Фіксація як дерево

Коли ви створили фіксацію за допомогою **git commit** (рис. 4.1.), **Git** обчислив контрольну суму кожної теки (у цьому випадку, тільки кореневої теки) та зберігає ці об'єкти дерева в сховищі **Git.** Потім **Git** створює об'єкт фіксації, що зберігає метадані та вказівник на корінь дерева проекту, щоб він міг відтворити цей знімок, коли потрібно.

Ваше **Git** сховище тепер зберігає п'ять об'єктів: по одному **blob** зі змістом на кожен з трьох файлів, одне **tree**, що перелічує зміст теки та вказує, які файли зберігаються у яких **blob**, та одну фіксацію, що вказує на корінь дерева, та зберігає метадані фіксації.

Якщо ви зробите якісь зміни та зафіксуєте знову, наступна фіксація буде зберігати вказівник на попередню.

Гілка в **Git** це просто легкий вказівник, що може пересуватись, на одну з цих фіксацій. Загальноприйнятим ім'ям першої гілки в **Git** є **master**. Коли ви почнете робити фіксації, вам надається гілка **master**, що вказує на останню зроблену фіксацію. Щоразу, коли ви фіксуєте, вона переміщується вперед автоматично.

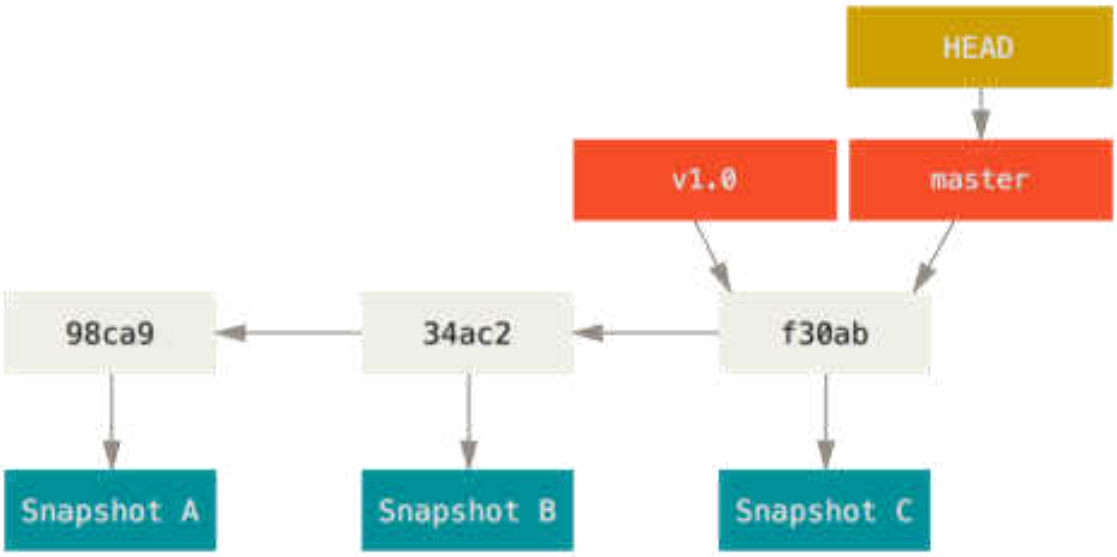


Рис. 4.2. Гілка та її історія фіксацій

4.1. Створення нової гілки

Створення нової гілки виконується командою:

$ git branch testing

Наприклад нова гілка під назвою **testing** (рис. 4.3.)

Рис. 4.3. Дві гілки вказують на одну послідовність фіксацій. HEAD вказує на поточну гілку

4.2. Переключення гілок

Перехід на існуючу гілку, виконують командою **git checkout**.

$ git checkout testing

Рис. 4.4. HEAD вказує на поточну гілку.

При фіксації поточна гілка пересувається вперед (**testing**)

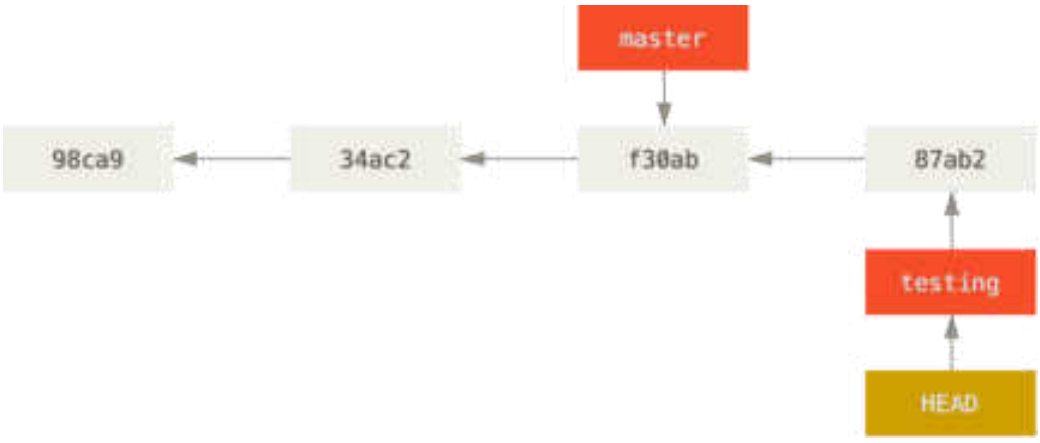


Рис. 4.5. Гілка HEAD пересувається уперед при фіксації

Переключення до гілки **master** виконують командою **git checkout:**

$ git checkout master

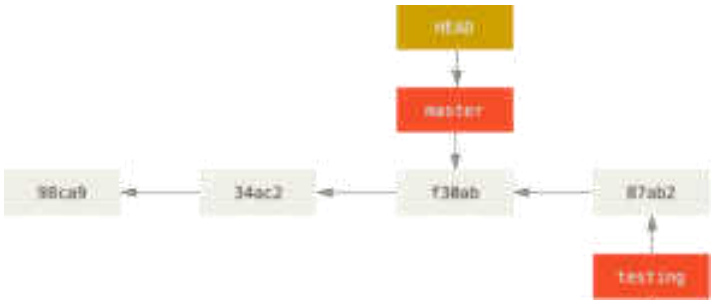


Рис. 4.6. HEAD пересувається, коли ви отримуєте (checkout)

Якщо знову зафіксувати то отримуємо розбіг (**diverged**) (рис 4.7.) історії проекту.

Рис. 4.7. Розбіг історії проекту

Перевірити історію ваших **commit** ви, можете командою **git log**.

Перевага **Git** над іншими СКВ у простій та зручній роботі з гілками. Гілка в **Git** — це насправді простий файл, що містить 50 символів контрольної суми **SHA-1 commit**, на який вказує. Гілки легко створювати та знищувати. Створити гілку так же швидко, як записати 41 байт до файлу (40 символів та символ нового рядка).

4.3. Основи зливання гілок

Ми маємо певну розгалужену структуру проекту, наприклад зображену на рис. 4.8.

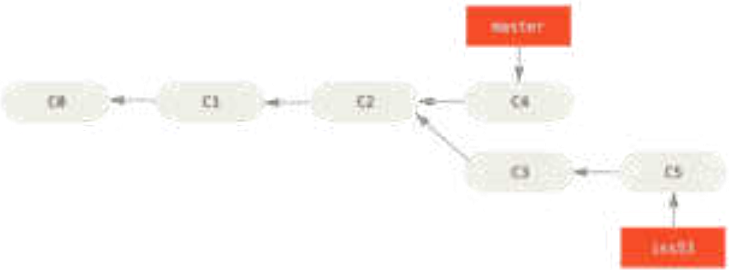


Рис. 4.8. Структура проекту з розгалуженням на гілку iss53

Якщо в процесі роботи над проектом – над гілкою **iss53**, ми вирішили що роботу завершено і можна злити з гілкою **master**. Для цього потрібно перейти на робочу гілку та виконати команду **git merge**:

$ git checkout master

Switched to branch 'master'

$ git merge iss53

Merge made by the 'recursive' strategy.

index.html | 1 +

1 file changed, 1 insertion(+)

В цьому випадку **Git** робить просте три-точкове злиття, користуючись двома знімками, що вказують на гілки та третім знімком - їх спільним предком.

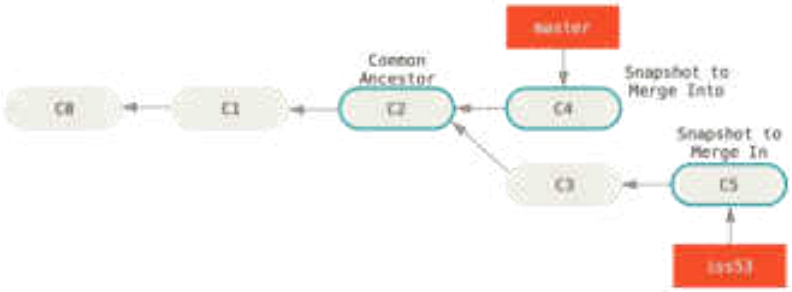


Рис. 4.9. Три відбитки типового злиття

Історія змін двох гілок почала відрізнятися в якийсь момент. Так як **commit** поточної гілки не є прямим нащадком гілки, в яку ви зливаєте зміни, **Git** мусить трохи попрацювати. В цьому випадку **Git** робить просте три-точкове злиття, користуючись двома знімками, що вказують на гілки та третім знімком - їх спільним предком. Такий **commit** називають **commit** злиття (**merge commit**). Його особливістю є те, що він має більше одного батьківського **commit**.

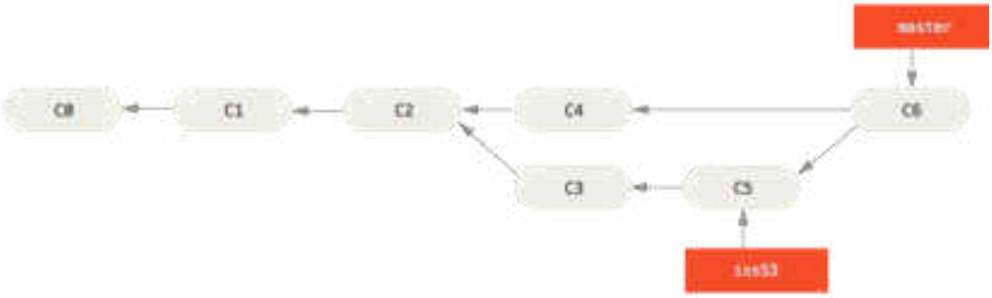


Рис. 4.10. **Commit** злиття

Тепер, коли ваші зміни зафіксовані, гілка **iss53** вам більше не потрібна. Її можна видалити:

$ git branch -d iss53

Git сам визначає найбільш підходящого спільного спадкоємця, якого брати за основу зливання. Це відрізняє **Git** від старіших систем таких як **CVS** чи **Subversion** (до версії 1.5), де розробник, що виконує зливання, сам повинен вказувати що брати за основу зливання.

Питання до розділу 4

1. Зміст об’єкту фіксації Git?
2. Опишіть фіксацію об’єктів в Git/
3. Створення нової гілки в Git?
4. Переключення гілок в Git?
5. Зливання гілок в Git?

РОЗДІЛ 5. GIT ТА GITHUB

5.1. Загальні дані Git та GitHub

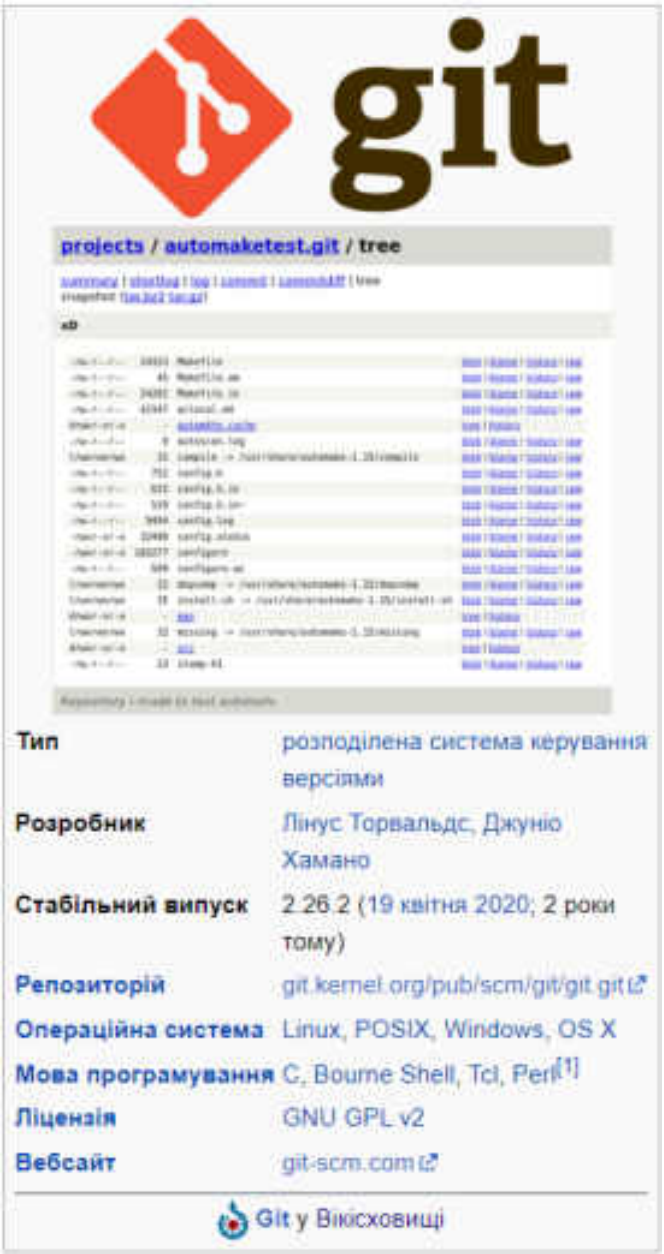


Рис. 5.1. Загальні дані **Git** та **GitHub**

Git це інструмент, що дозволяє реалізувати розподілену систему контролю версій.

GitHub - сервіс онлайн-хостингу репозиторіїв, що володіє всіма функціями розподіленого контролю версій та функціональністю управління вихідним кодом - все, що підтримує **Git** і навіть більше. Також **GitHub** може похвалитися контролем доступу, баг-трекінгом, керуванням завданнями та вікі для кожного проекту.

Git-репозиторій, завантажений на **GitHub**, доступний за допомогою інтерфейсу командного рядка **Git** та **Git**-команд. Також є й інші функції: документація, запити на прийняття змін (**pull requests**), історія **commit**, інтеграція з безліччю популярних сервісів, email-повідомлення, емодзі, графіки, вкладені списки завдань, система згадувань, схожа на ту, що в **Twitter**, і та ін.

Крім **GitHub** є інші сервіси, які використовують **Git**, наприклад, **Bitbucket** і **GitLab**. Ви можете розмістити **Git** репозиторій на будь-якому з них.

5.2. Коротко про GitHub

GitHub — найбільший веб-сервіс для хостингу IT-проектів та їх спільної розробки.

Веб-сервіс заснований на **системі контролю версії (СКВ) Git** і розроблений на **Ruby on Rails і Erlang** компанії **GitHub, Inc** (раніше **Logical Awesome**). Сервіс безкоштовний для проектів з відкритим вихідним кодом і (з 2019 року) невеликими приватними проектами, надаючи їм усі можливості (включаючи **SSL**), а для великих корпоративних проектів пропонуються різні платні тарифні плани.

Це дозволяє вам працювати разом з іншими людьми по всьому світу, планувати свої проекти та відстежувати свою роботу застосовуючи **GitHub**.

GitHub почав працювати в квітні 2008 року, а розробники називають його «соціальною мережею для розробників».

GitHub також є одним із найбільших онлайн-сховищ проектів спільної роботи по всьому світу.

5.3. Створення аккаунту GitHub

Використання можливостей **GitHub** починається з реєстрації нового користувача, тобто створення аккаунту – на сторінці https://github.com/

Рис. 5.2. Головна сторінка сайту github.com

шляхом **створення нового облікового запису**.

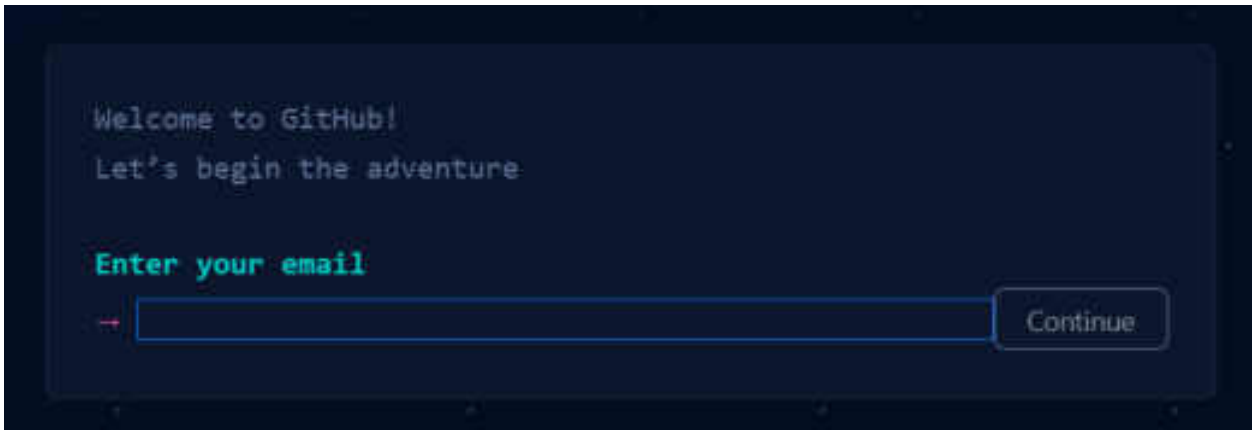


Рис. 5.3. Реєстрація email

Після введення електронної пошти покроково вводимо необхідні реєстраційні дані – пароль, ім'я користувача та обираємо можливість отримувати, чи не отримувати на електронну пошту, анонси про зміні програмного продукту.

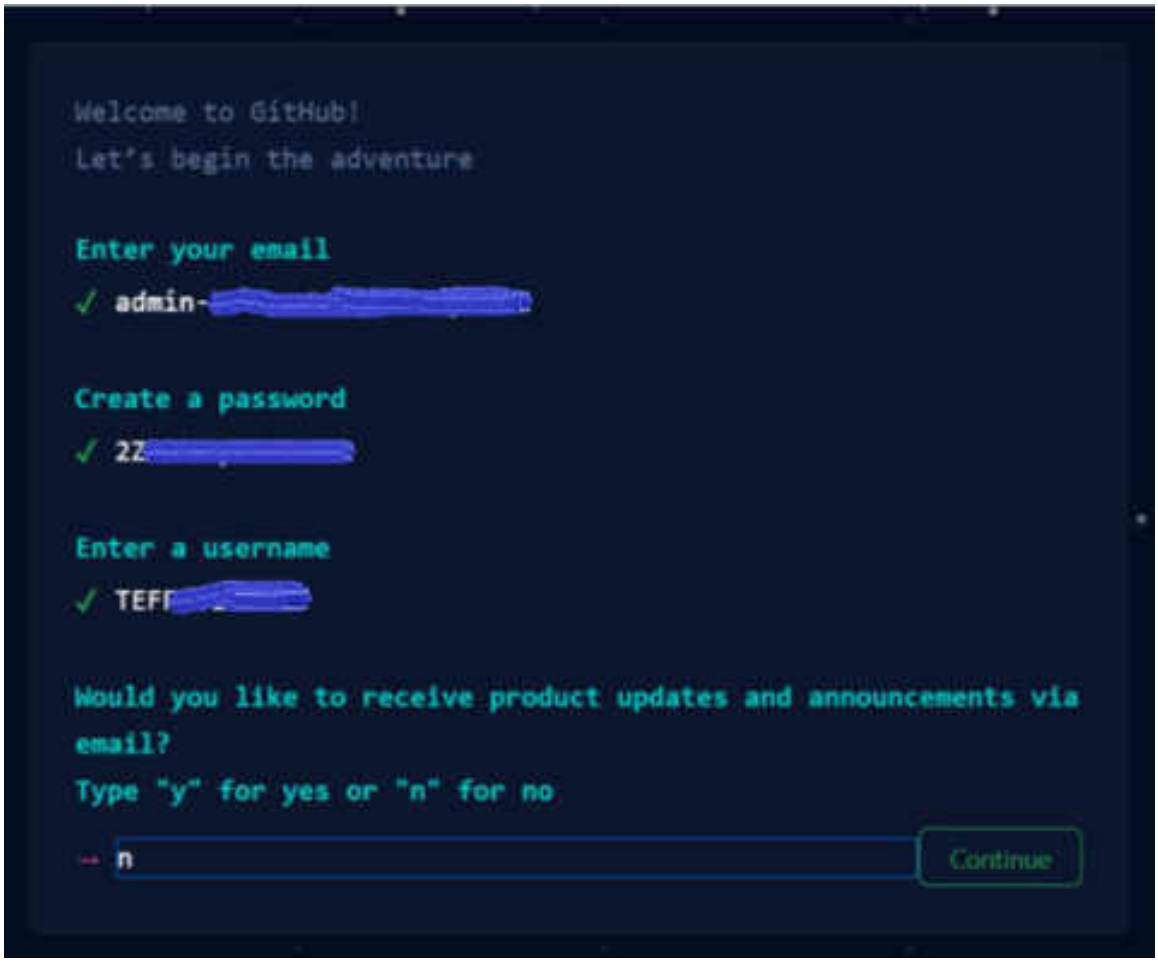


Рис. 5.4. Реєстраційні дані

Потім виконується проста верифікація.

Доступ до платформи можливий тільки після обов'язкового підтвердження електронної пошти.

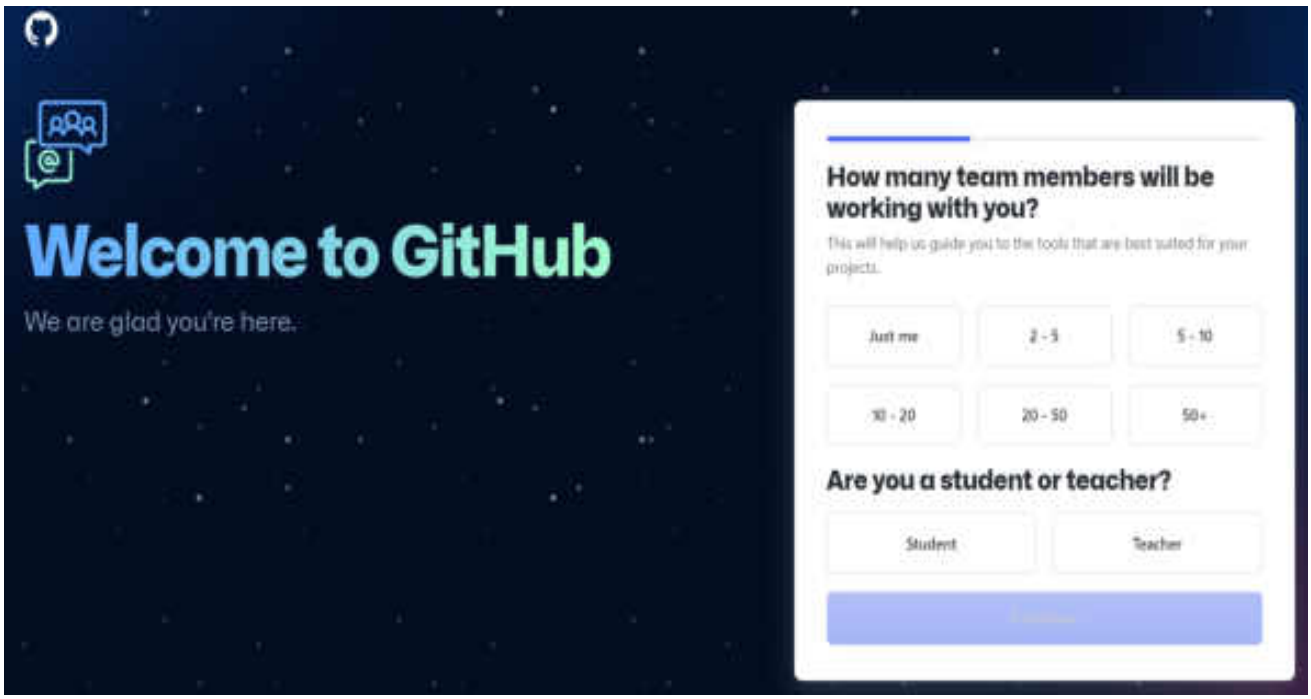


Рис. 5.5. Сторінка реєстрації “Welcome to GitHub”

Для навчання розробників надається спеціальний безкоштовний доступ до стандартних інструментів.

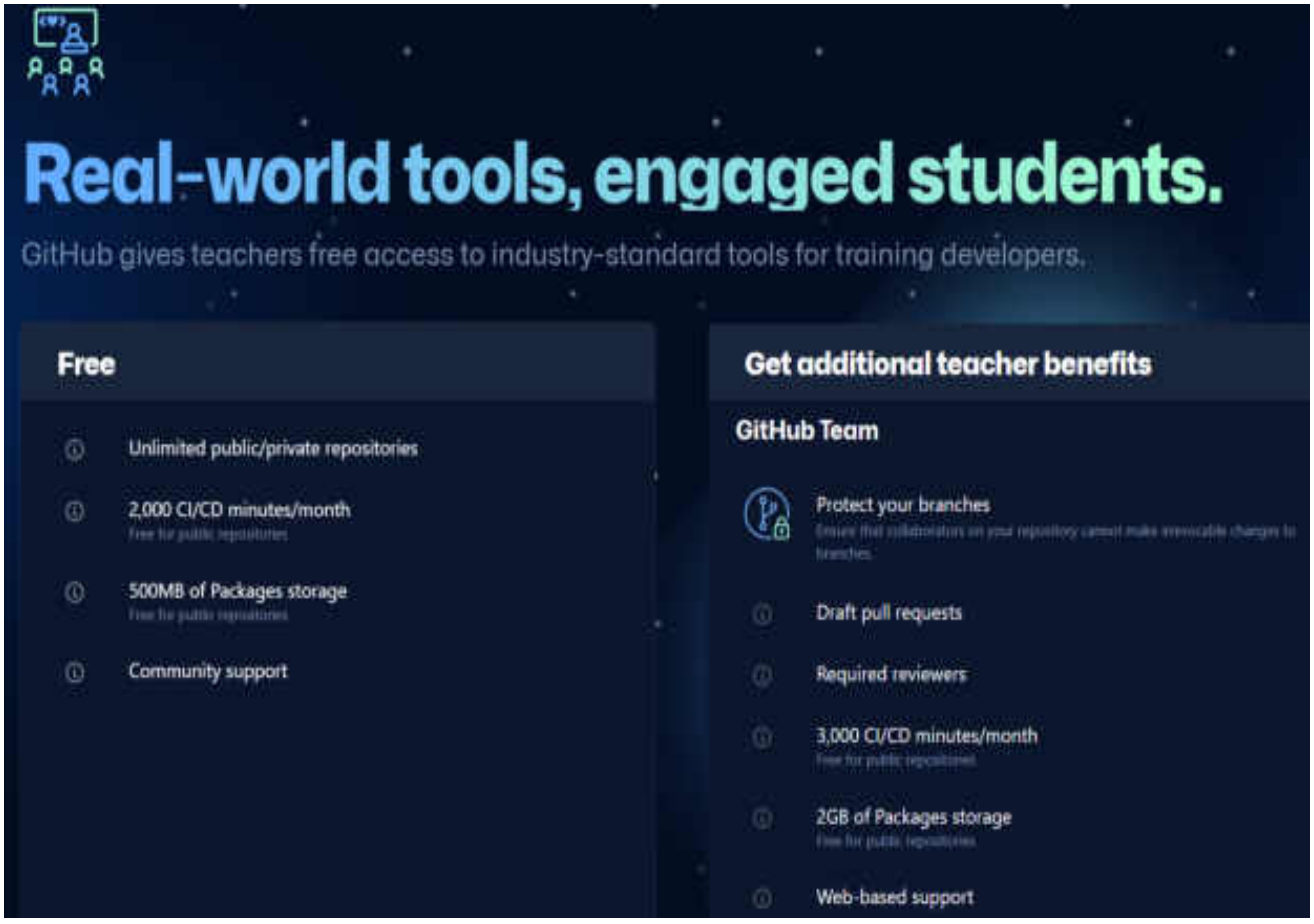


Рис. 5.6. Додаткові можливості для навчання

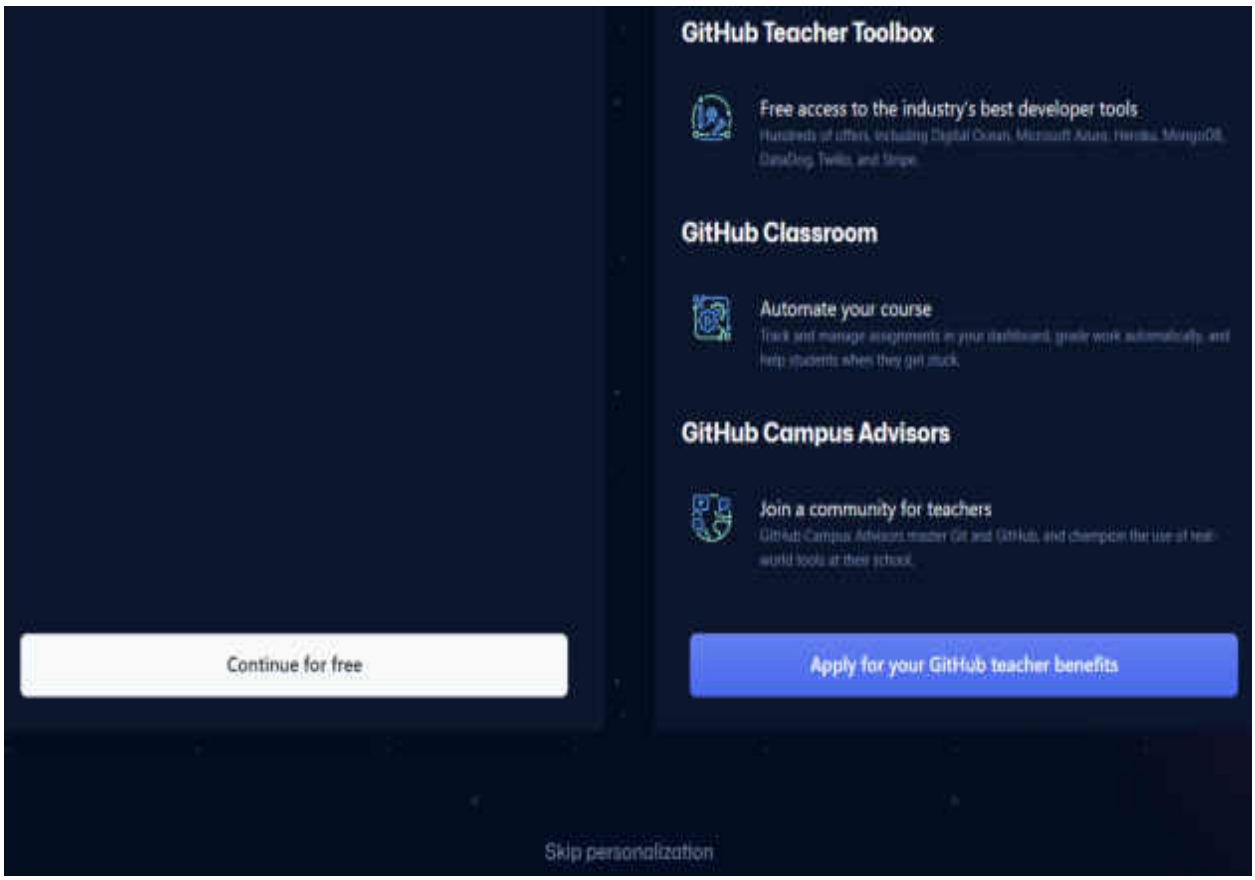


Рис. 5.7. Додаткові можливості для навчання. Продовження

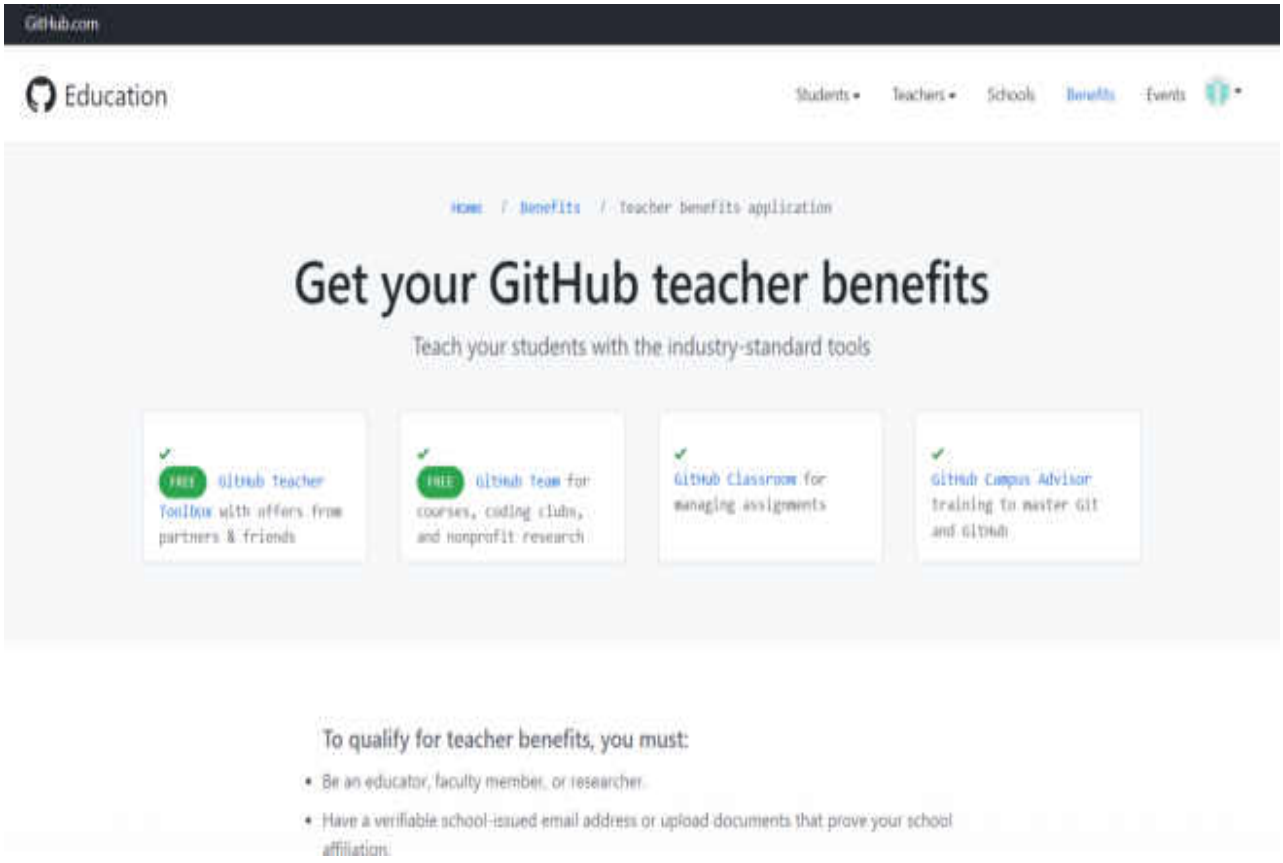


Рис. 5.8. Переваги для вчителів

5.4. Встановлення Git на локальному пристрої

На офіційній сторінці https://git-scm.com/ в розділі можна отримати необхідну вам версію.

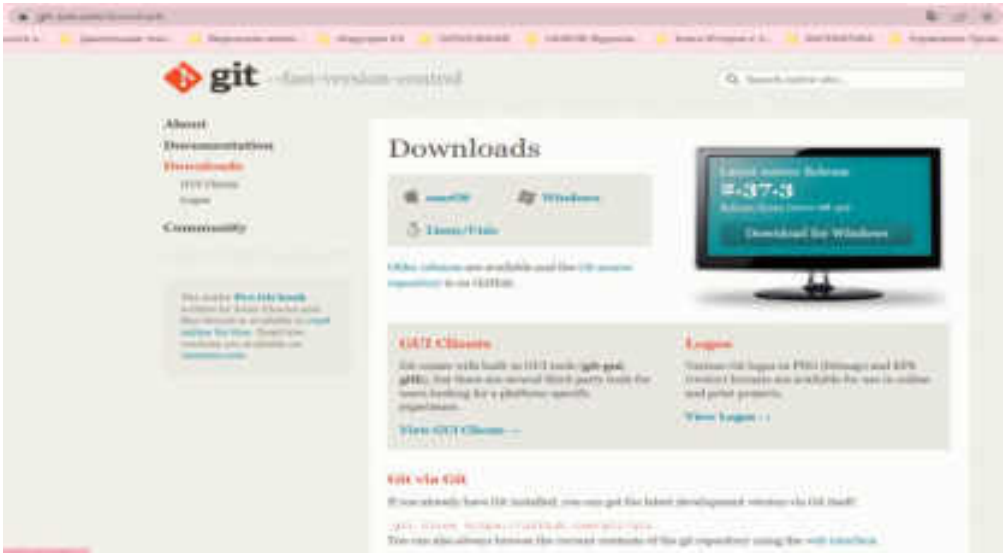


Рис. 5.9. Офіційна сторінка Downloads GitHub

Встановлення версій **Git** для **Linux**, **Mac** та **Windows** є простою процедурою, що викладено інструкціями для **Linux** та **Mac** на офіційному сайті. В даному керівництві ми розглянемо приклад встановлення **Git** інсталяційним файлом для **Windows**.

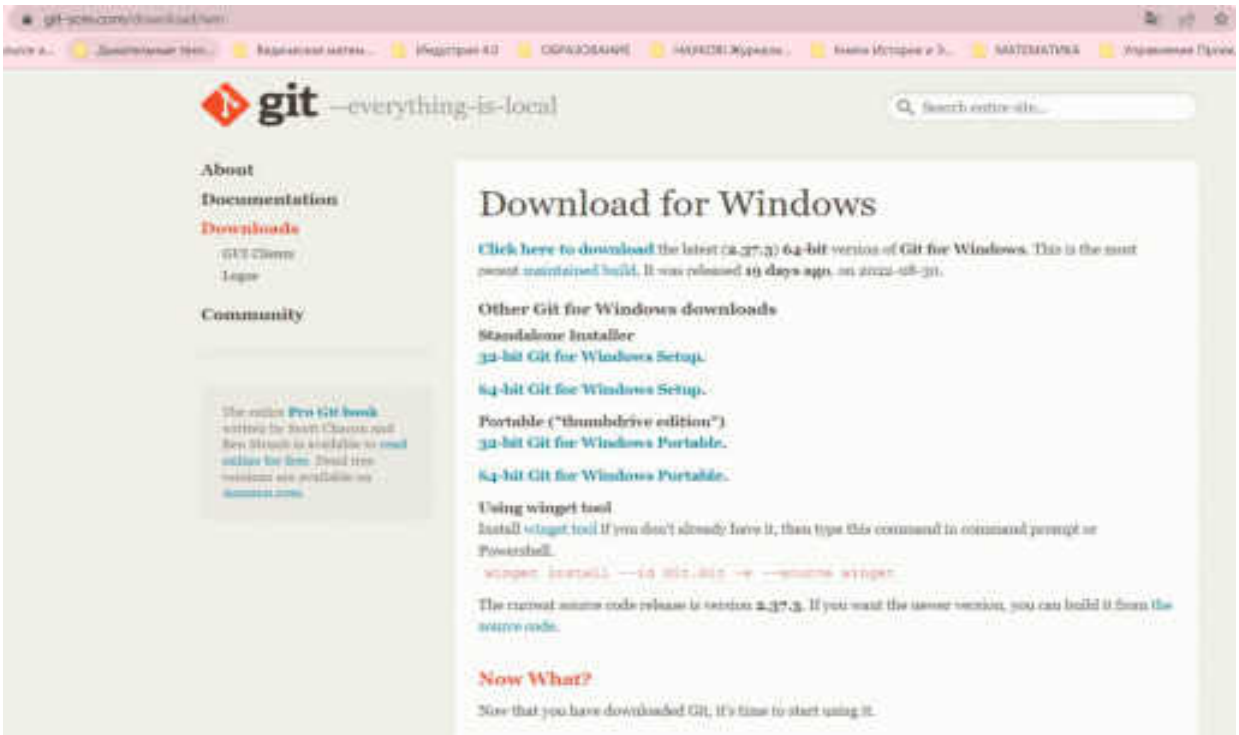


Рис. 5.10. Вибір версії Git для Windows

Встановлення **Git** виконується в наступні кроки:

1. Підтвердження ліцензійної угоди.
2. Вибір розташування установки.
3. Вибір опцій встановлення:

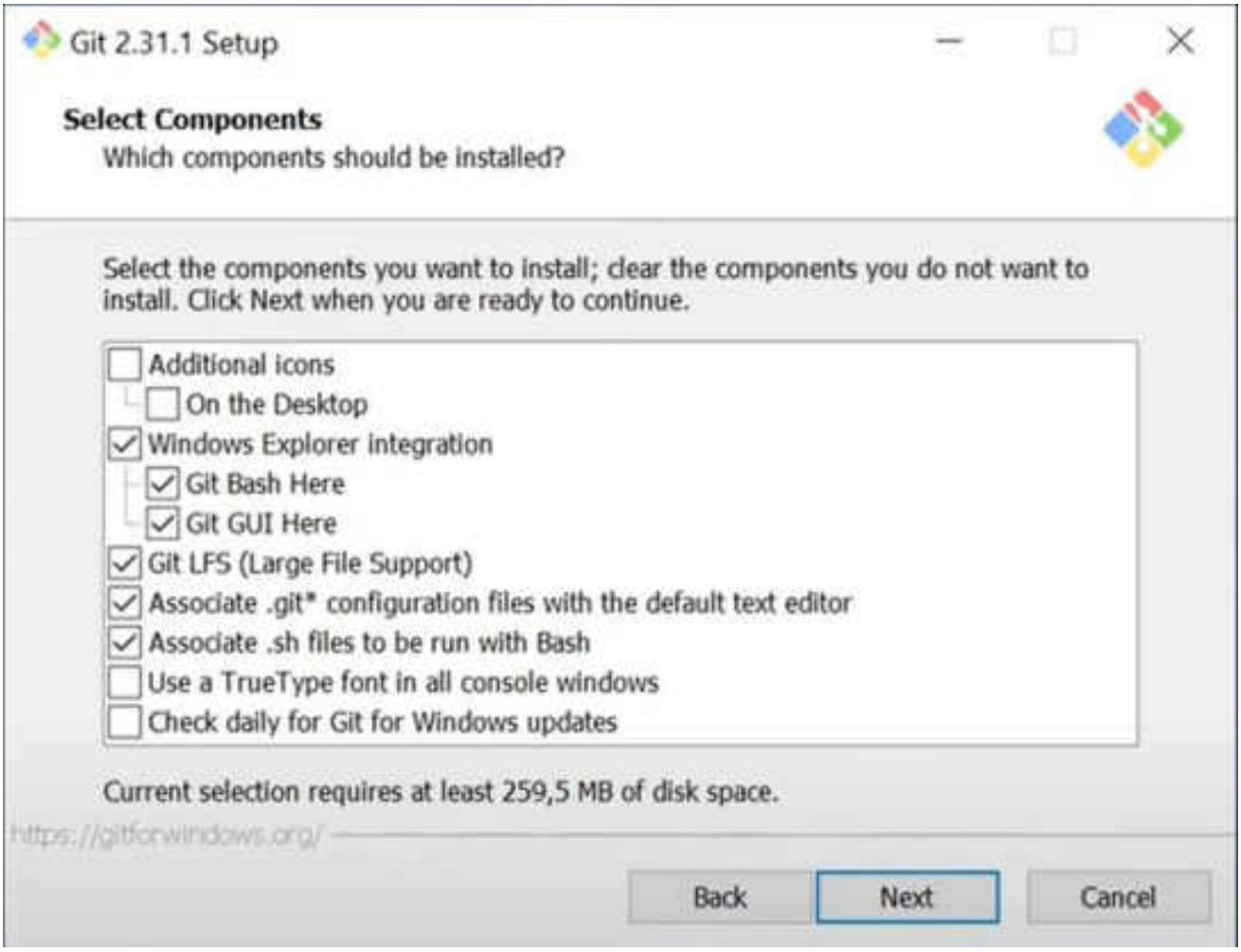
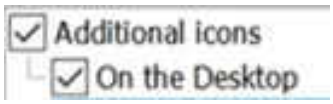


Рис. 5.11. Компоненти встановлення Git

Де:

- 

- додавання іконки на робочий стіл.
- 

- **Git Bash** - це додаток для середовищ **Microsoft Windows**, що емулює роботу командного рядка **Git. Bash** - абревіатура від **Bourne Again Shell**.
- **GUI (graphical user interface)** – графічний інтерфейс користувача. Часто користувачі використовують **GUI** - особистого середовища розробки.

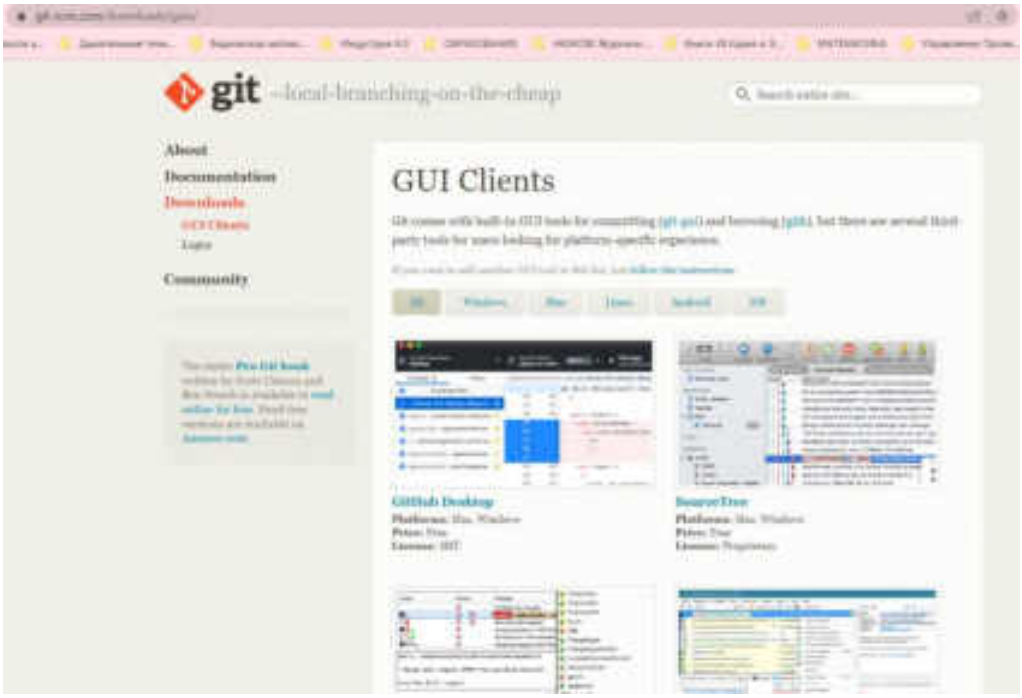


Рис. 5.12. GUI Clients

- 

- плагін для роботи з файлами великого розміру.

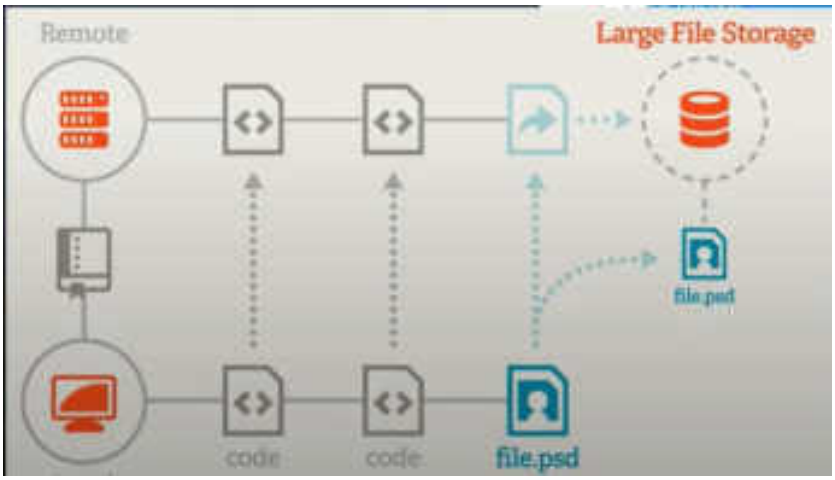
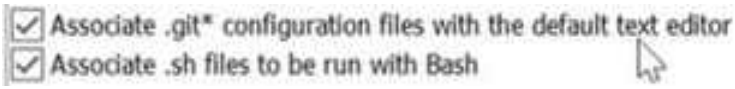


Рис. 5.13. Схема роботи з файлами великого розміру

- 

- асоціація зі стандартним текстовим редактором та можливість запускати скриптові - **.sh** фали в **Git Bash**.

- 

- можливість використовувати TrueType шрифти та щоденна перевірка оновлень.

4. Вибір папки для стартового меню:

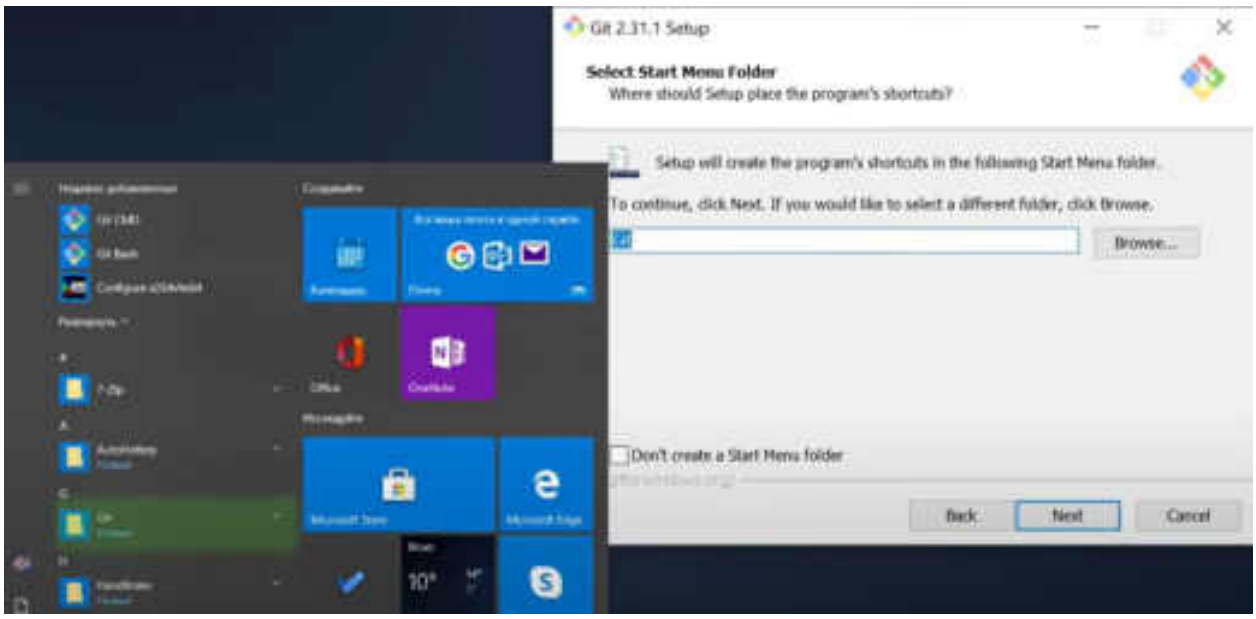


Рис. 5.14. Вибір назви папки в стартовому меню

5. Вибір стандартного редактору для напису коментарів.
6. Вибір назви гілки за замовчуванням при створенні нового репозиторію командою **git init**.

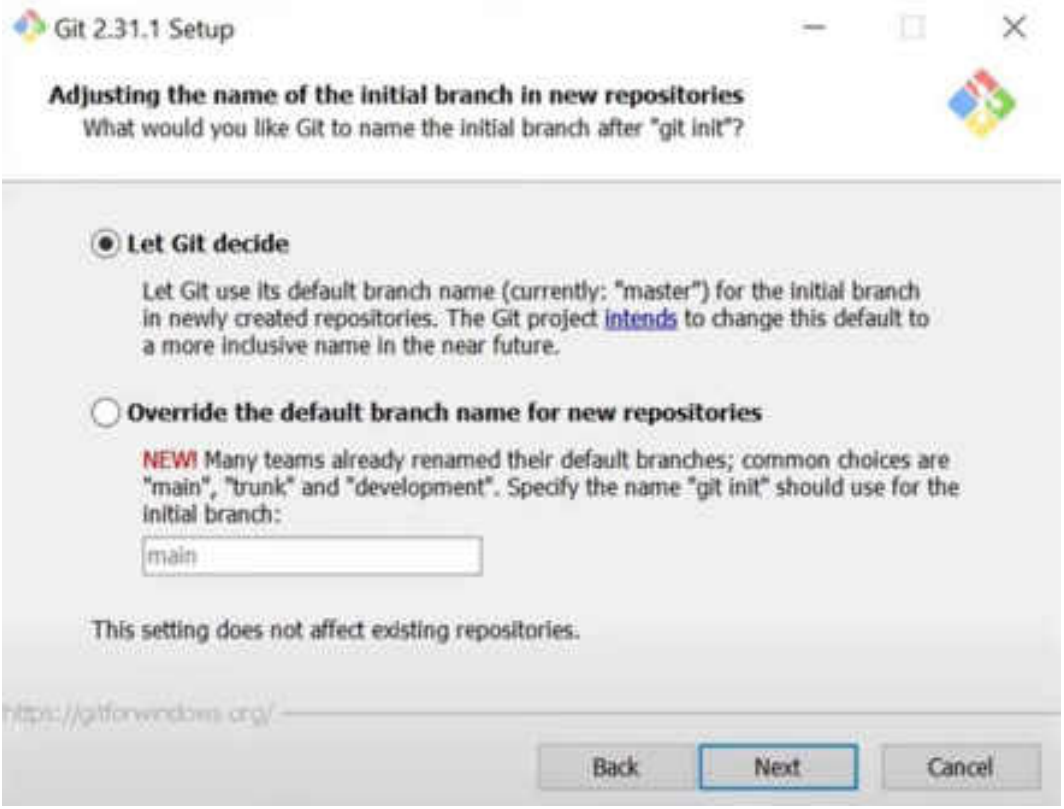


Рис. 5.15. Вибір назви гілки за замовчуванням при створенні нового репозиторію командою **git init**

7. Вибір змінних оточення **PATH**.
8. Вибір бібліотеки для **HTTPS** з’єднання.

9. Вибір стилю переносу рядків.

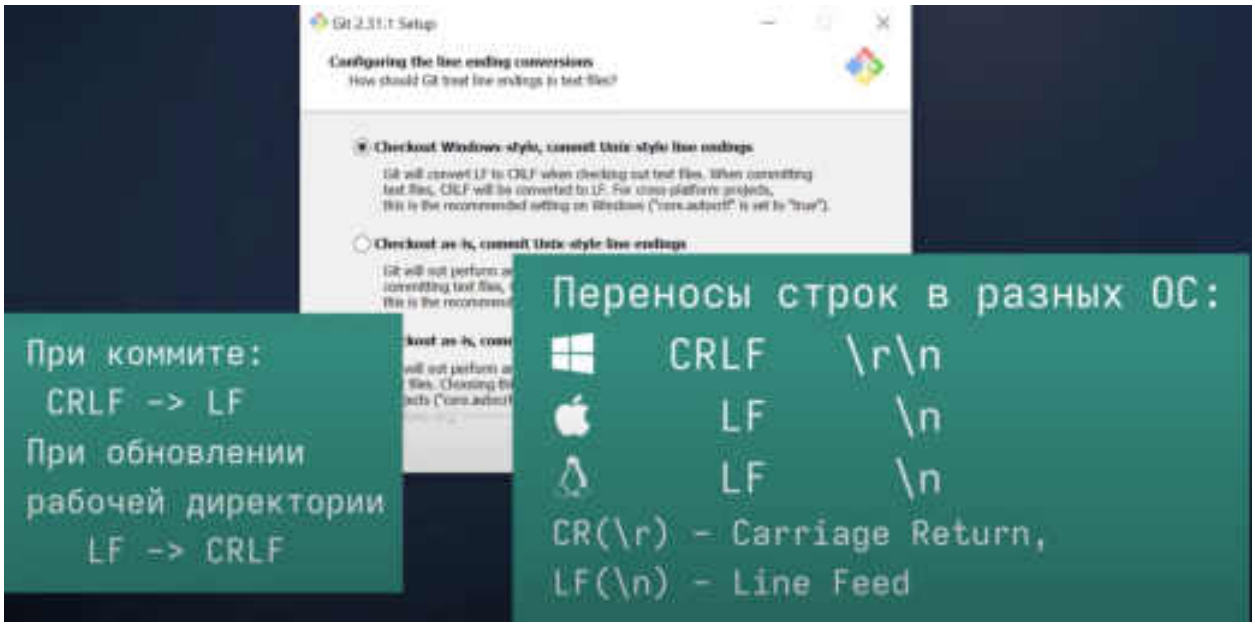


Рис. 5.16. Вибір стилю переносу рядків

10. Вибір консолі, що буде використовуватись для **GIT**.
11. Вибір стратегії роботи команди **git pull.**

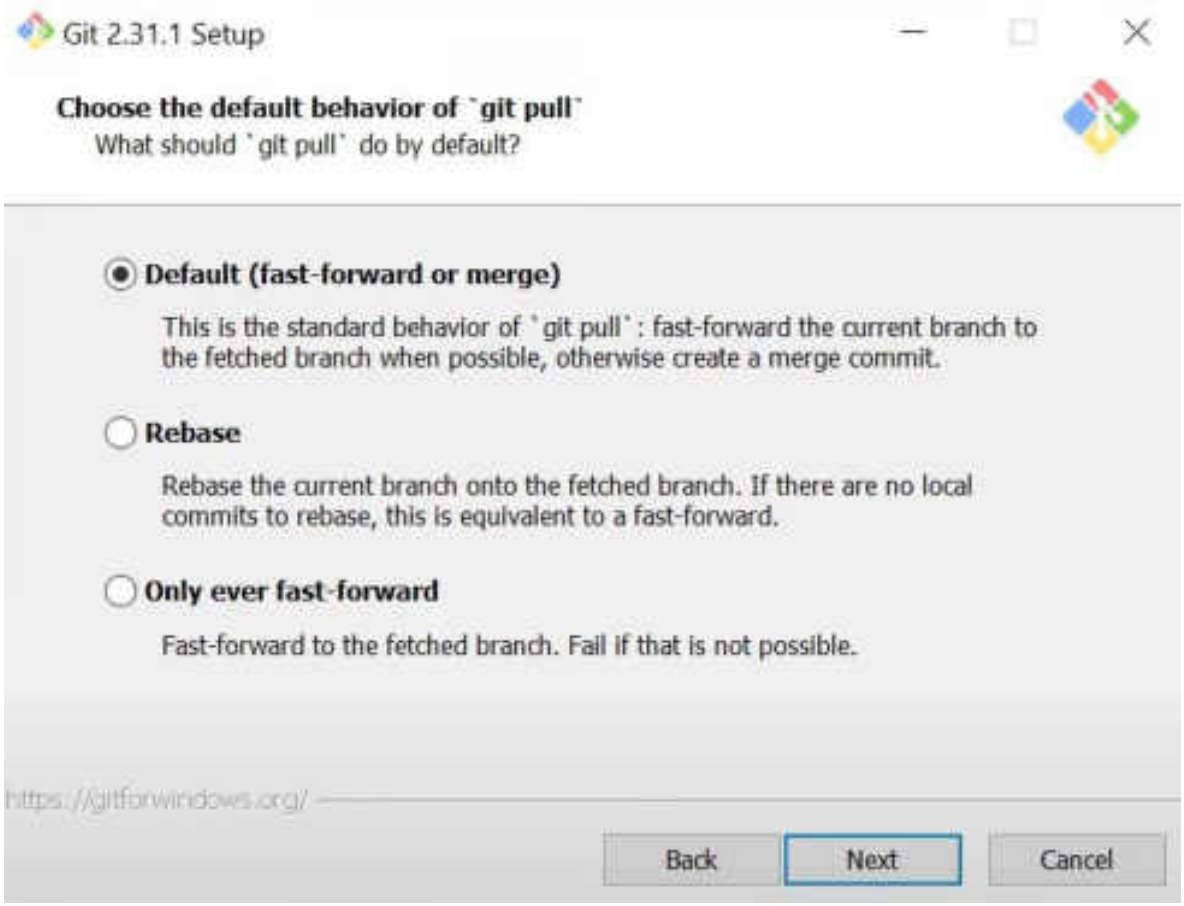


Рис. 5.17. Вибір стратегії роботи команди **git pull**

12. Налаштування **HTTPS** доступу до віддалених репозиторіїв.

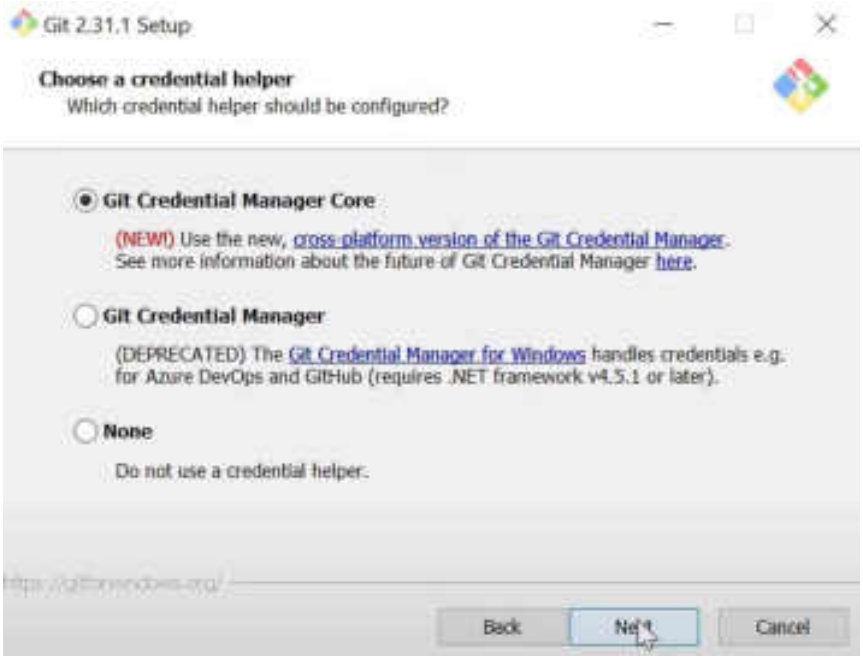


Рис. 5.18. Налаштування **HTTPS** доступу до віддалених репозиторіїв

Git Credential Manager Core – дозволяє вводити логін та пароль лише один раз, після чого облікові дані будуть збережені в сховищі та підключення буде виконуватись автоматично. Цей метод не обмежує застосування **SHA** ключів.

13. Вибір спеціальних налаштувань – прискорення роботи **Git** та використання символьних посилань.

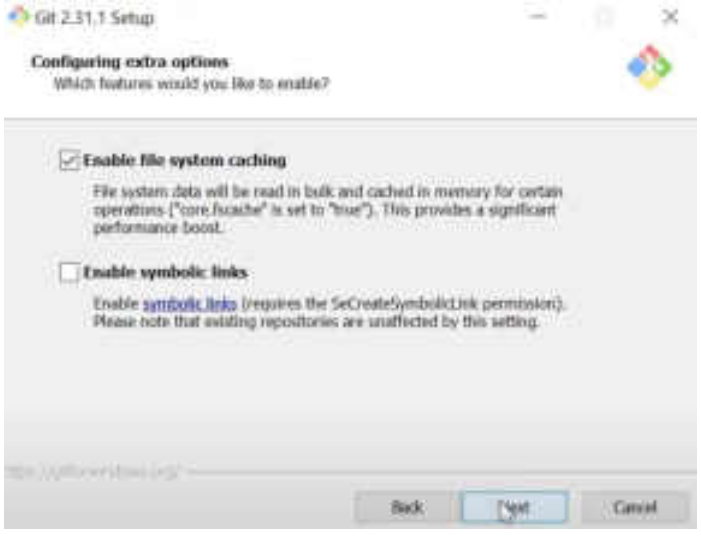


Рис. 5.19. Вибір спеціальних налаштувань

14. Вибір експериментальних налаштувань.
15. Процес інсталяції.
16. Вибір – **Git Bash**

Рис. 5.20. Вибір Git Bash

17. Перевірка установки **Git.**

Перевірка установки **Git** виконується командою:

$ git --version

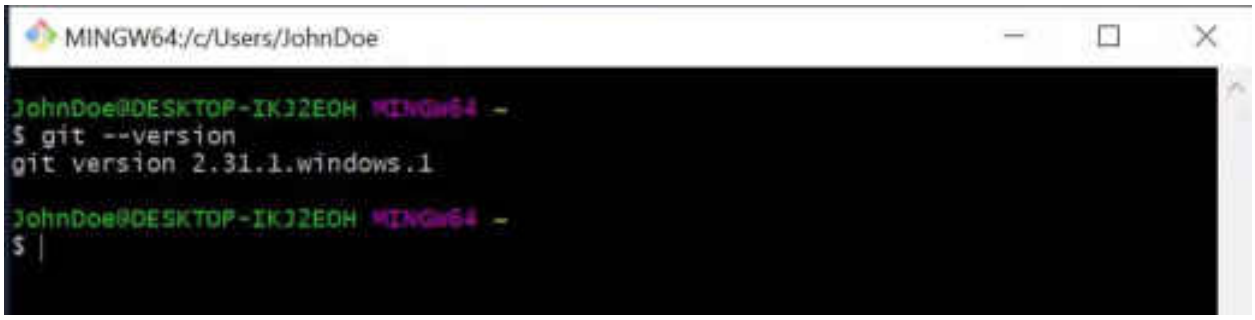


Рис. 5.21. Перевірка установки **Git**

5.5. Початкові налаштування Git на локальному пристрої

До складу **Git** входить утиліта ***git config***, яка дозволяє переглядати та налаштовувати параметри, що контролюють усі аспекти роботи **Git**, а також його зовнішній вигляд. Ці параметри можуть бути збережені у трьох місцях [1]:

1. Файл ***[path]/etc/gitconfig*** містить значення, загальні для всіх користувачів системи та всіх їх репозиторіїв. Якщо при запуску ***git config*** вказати параметр ***--system***, параметри читаються і зберігатися саме в цей файл. Так як цей файл є системним, то вам потрібні права супер-користувача для внесення змін до нього.

2. Файл ***~/.gitconfig*** або ***~/.config/git/config*** зберігає установки конкретного користувача. Цей файл використовується при вказівці параметра ***--global*** і застосовується до всіх репозиторіїв, з якими ви працюєте у поточній системі.

3. Файл ***config*** у каталозі **Git** (тобто ***.git/config***) репозиторію, який ви використовуєте в даний момент, зберігає налаштування конкретного репозиторію. Ви можете змусити **Git** читати і писати в цей файл за допомогою ***--local***, але насправді це значення за замовчуванням. Не дивно, що вам потрібно бути десь у репозиторії **Git**, щоб ця опція працювала правильно.

Налаштування на кожному наступному рівні підміняють налаштування з попередніх рівнів, тобто значення ***.git/config*** перекривають відповідні значення ***[path]/etc/gitconfig***.

У системах сімейства **Windows Git** шукає файл ***.gitconfig*** у каталозі **$HOME** (**C:\Users\$USER** для більшості користувачів). Крім того, **Git** шукає файл ***[path]/etc/gitconfig***, але вже щодо кореневого каталогу **MSys**, який знаходиться там, куди ви вирішили встановити **Git** під час запуску інсталятора.

Якщо ви використовуєте **Git для Windows версії 2.x або новіше**, то також обробляється файл конфігурації рівня системи, який має шлях ***C:\Documents and Settings\All Users\Application Data\Git\config*** у **Windows XP** або ***C:\ProgramData\Git\config*** у **Windows Vista** та новіший. Цей файл може бути змінений лише за допомогою команди ***git config -f <file>***, запущеної з правами адміністратора.

Щоб переглянути всі встановлені налаштування та дізнатися, де саме вони задані, використовуйте команду:

$ git config --list --show-origin

Дані користувача:

Для того, щоб приступити до роботи необхідно ідентифікувати користувача (опція *–global* застосовує налаштування для всіх операцій в системі):

*$ git config –global user.name «**John Doe**»*
*$ git config –global user.email «**johndoe@example.com**»*

Вибір редактору:

*$ git config –global core.editor **emacs***

Налаштування гілки за замовчуванням:

$ git config --global init.defaultBranch main

Перевірка налаштувань:

git config --list

Питання до розділу 5

1. Що таке Git та GitHub?
2. Процедура реєстрації GitHub та встановлення Git?
3. Початкові налаштування Git на локальному пристрої?
4. Налаштування за замовчуванням?

РОЗДІЛ 6. ОРГАНІЗАЦІЯ. КОМАНДА. РЕПОЗИТОРІЙ. РОЛІ

Організаційною основою роботи з **GitHub** є “Організації” в просторі яких знаходяться їх проекти. Облікові записи організацій представляють групи людей з сумісним правом власності проектів та різними інструментами для керування групами.

6.1. Створення організації

Щоб створити організацію необхідно обрати іконку “+” нагорі праворуч на будь-якій сторінці **GitHub** та виберіть у меню **“New organization”** (нова організація).

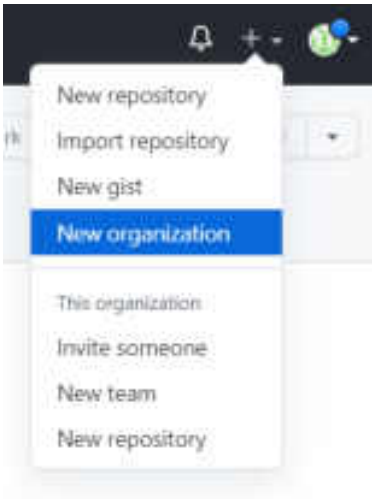


Рис. 6.1. Створення нової організації

Переходимо до сторінки вибору.

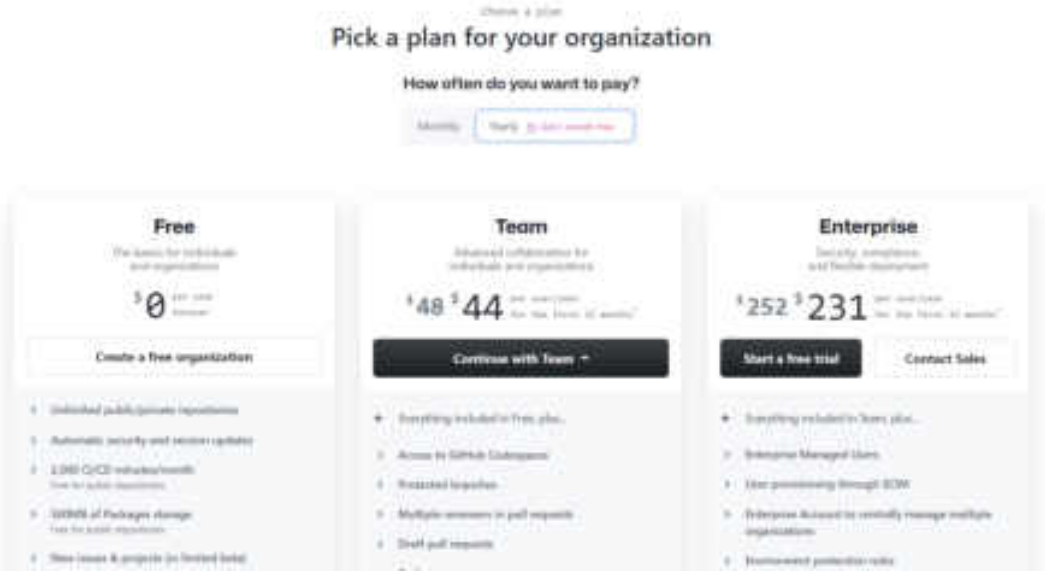


Рис. 6.2. Вибір плану для вашої організації

Де обираємо план “**Free**” і потім обираємо назву організації та контактну адресу електронної пошти для організації

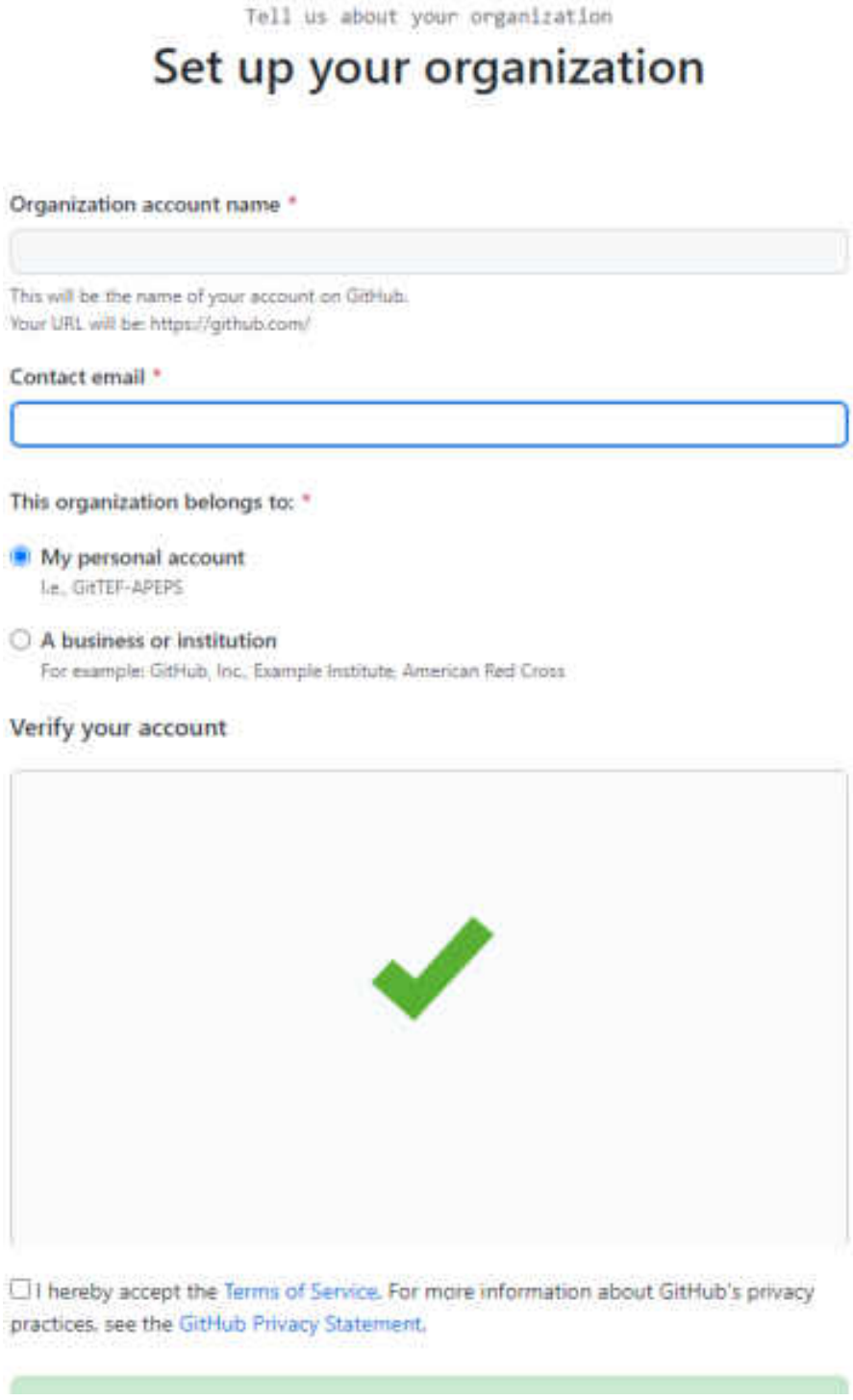


Рис. 6.3. Сторінка початкових налаштувань організації

Далі обираєте варіант вашого аккаунта – чи персональний, чи бізнес організації. Проводите верифікацію та підтверджуєте згоду і обираєте продовжити (Next).

Наступною сторінкою буде сторінка завершення створення організації, що пропонує пошуком додати учасників до організації шляхом пошуку ім’я користувача, повного ім’я або адреси електронної пошти.

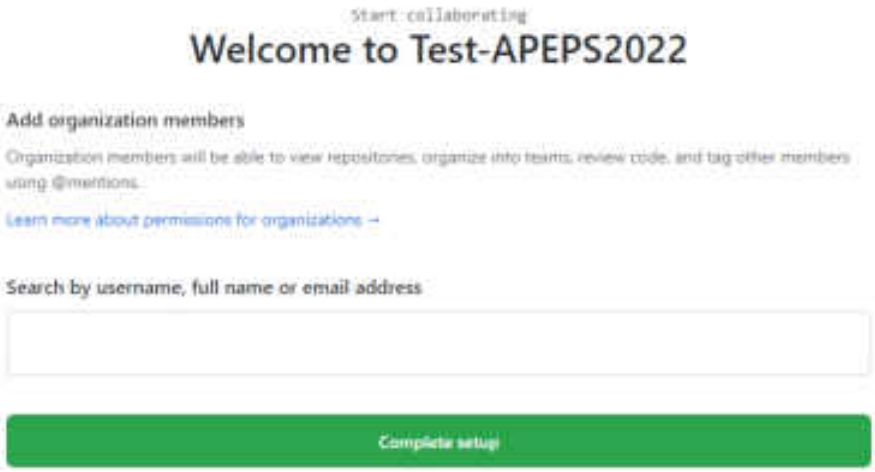


Рис. 6.4. Сторінка додання членів до організації

Наступна сторінка є опитувальною.

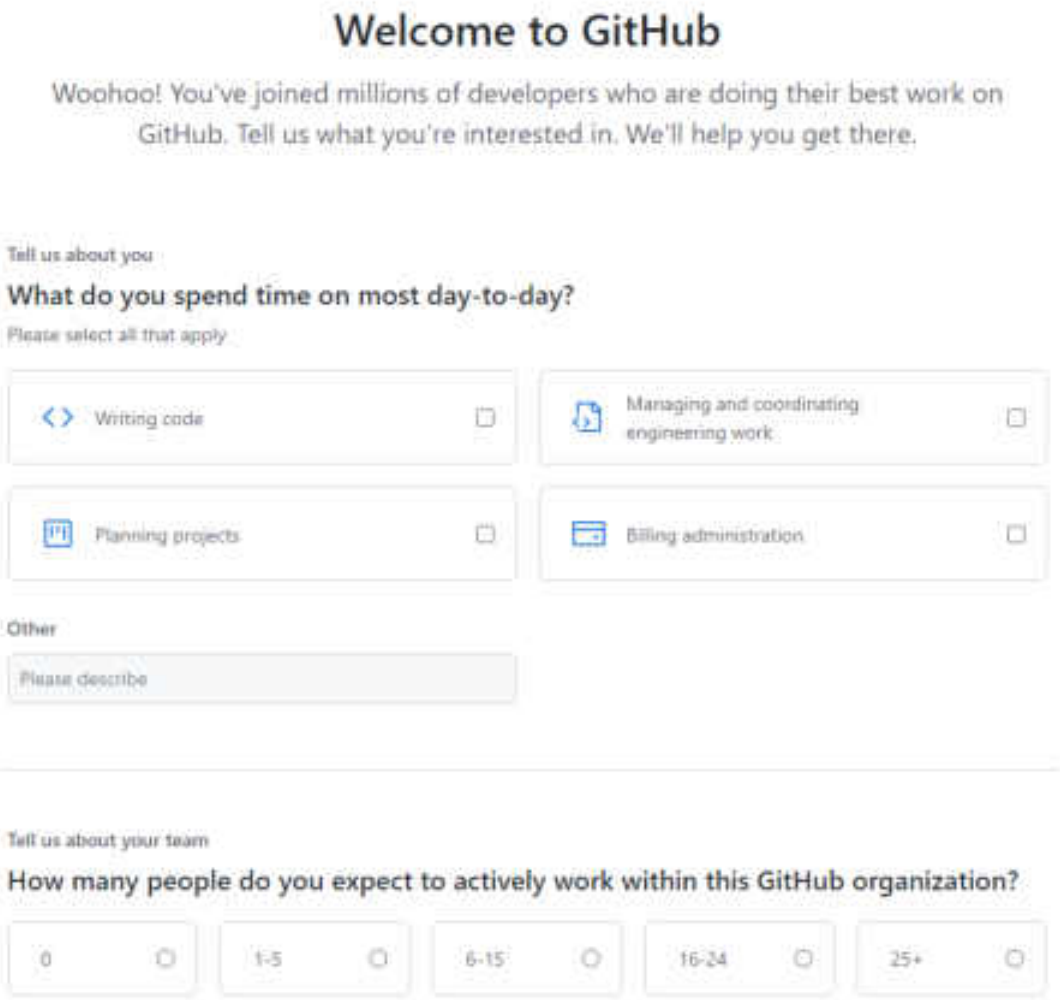


Рис. 6.5. Сторінка “Welcome to GitHub”

Використовуючи в правому верхньому куті значок вашого профілю ви можете переглянути інформацію за вашою організацією. Де ви отримаєте доступ до різних даних вашої організації.

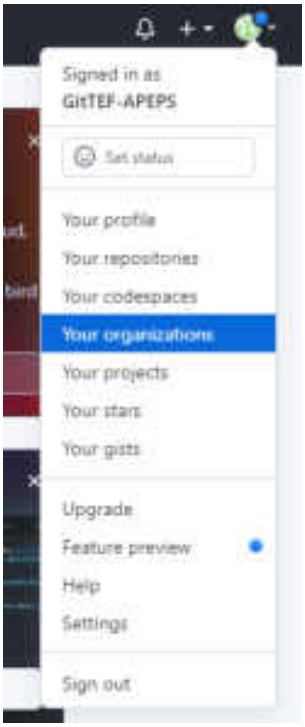


Рис. 6.6. Перехід до сторінки інформації про організацію

Також доступ до можливості управляти організаціями розташовано в лівому верхньому куті.

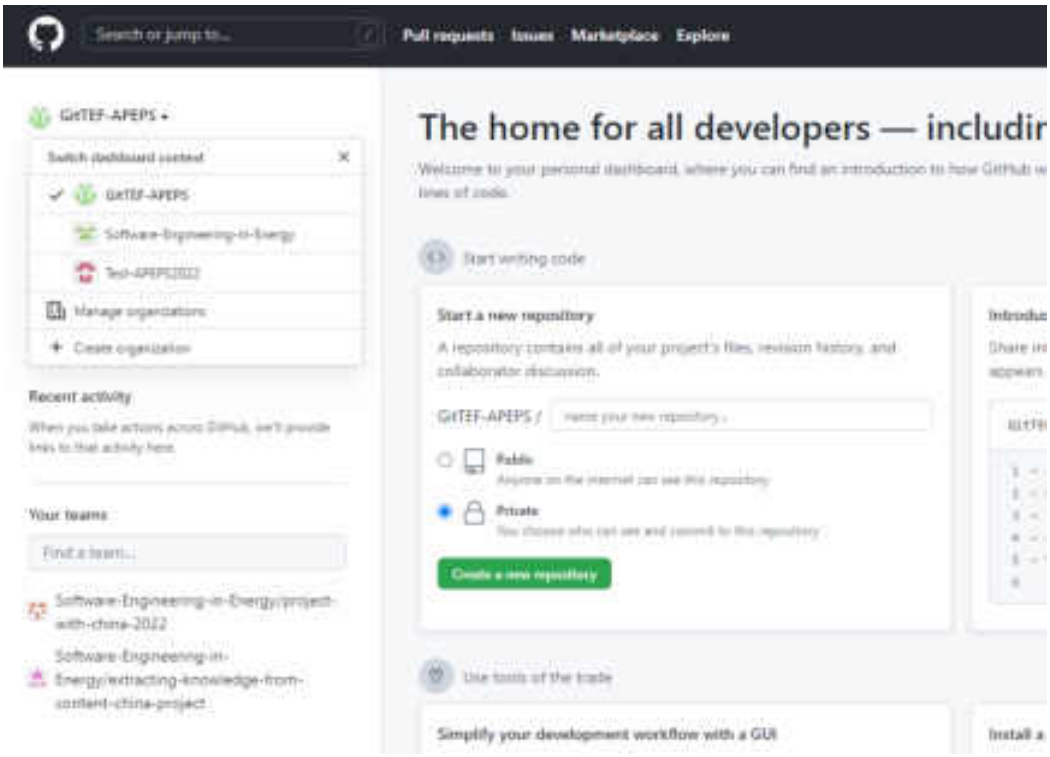


Рис. 6.7. Управління організаціями

Як і особисті облікові записи, організації безкоштовні, якщо все, що ви будете в них зберігати буде відкритим кодом.

Як власник організації, коли ви робите **fork** сховища, у вас буде вибір: робити **fork** у вашому власному просторі імен, чи у просторі імен організації. Коли ви створюєте нові сховища, ви можете створити їх або під особистим обліковим записом, або під будь-якою організацією, що її власником є ви. Також ви автоматично будете "слідкувати" (**watch**) за всіма сховищами, що ви створили для цих організацій.

Ви маєте головну сторінку організації, на якій є список усіх ваших сховищ — її можуть бачити й інші люди.

6.2. Команди

Організації можуть працювати, як з окремими людьми так і з окремими людьми через команди.

Щоб керувати командами потрібно на сторінці організації обрати і перейти за "**Teams**".

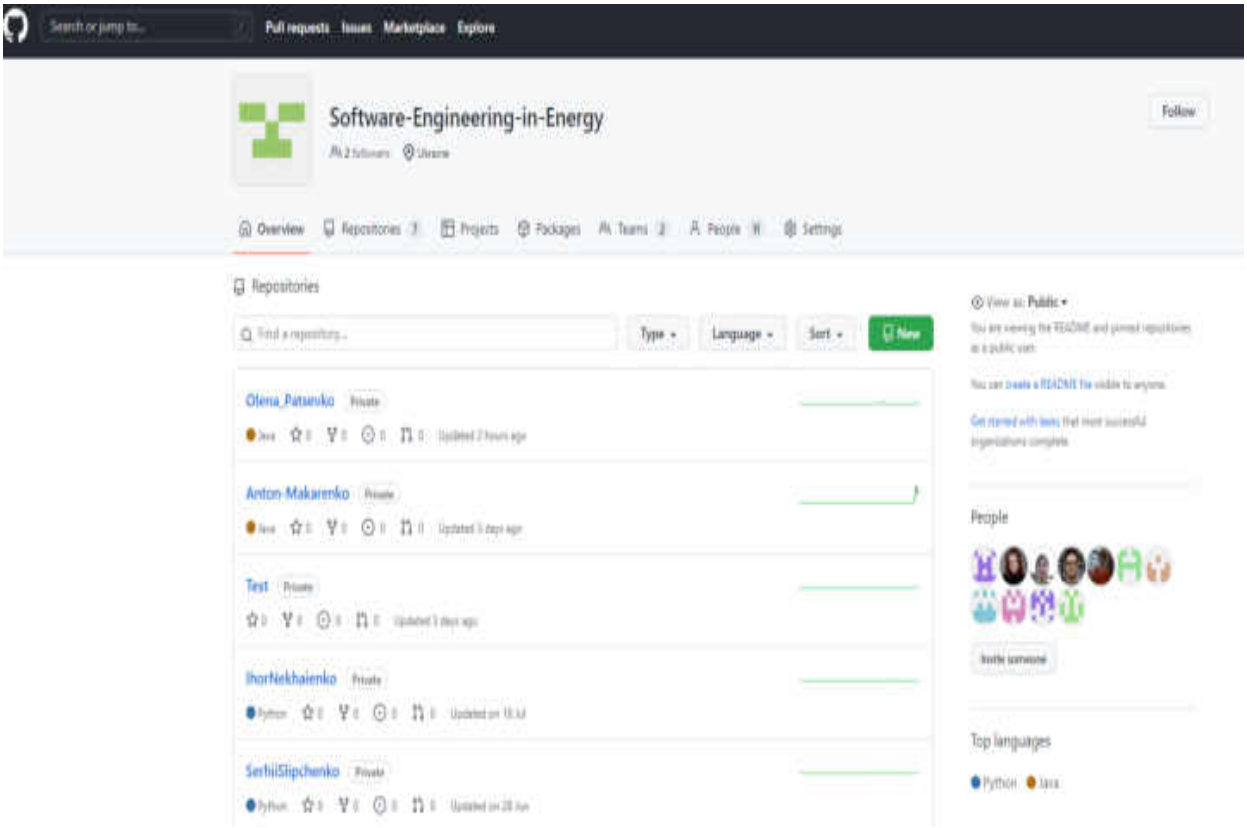


Рис. 6.8. Керування командами

Ви опинитесь на сторінці, що буде відображати команди організації та можливість створити нові – **"New team"**.

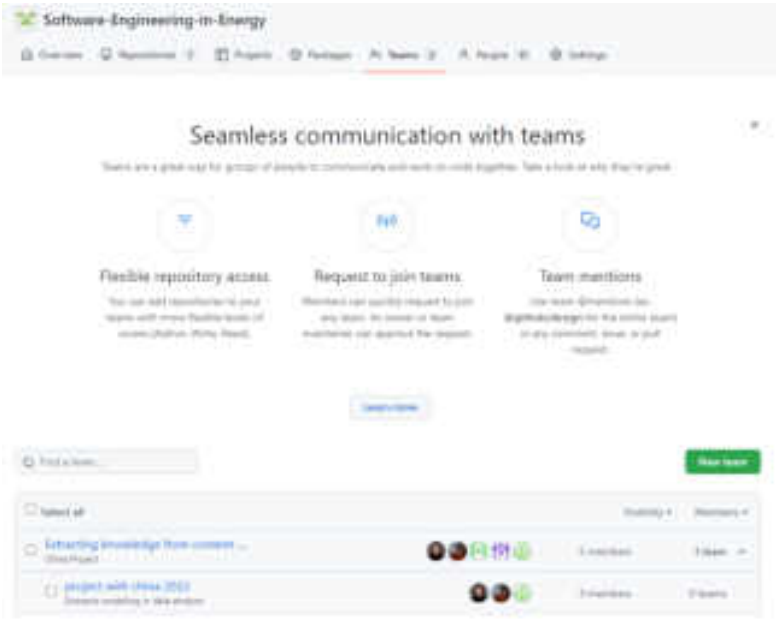


Рис. 6.9. Сторінка “Teams”

З’являється можливість приєднатися до дискусії стосовно команди

Рис. 6.10. Перехід до дискусій команд

або переглянути профілі членів команди, або створити команду – **“New team”**

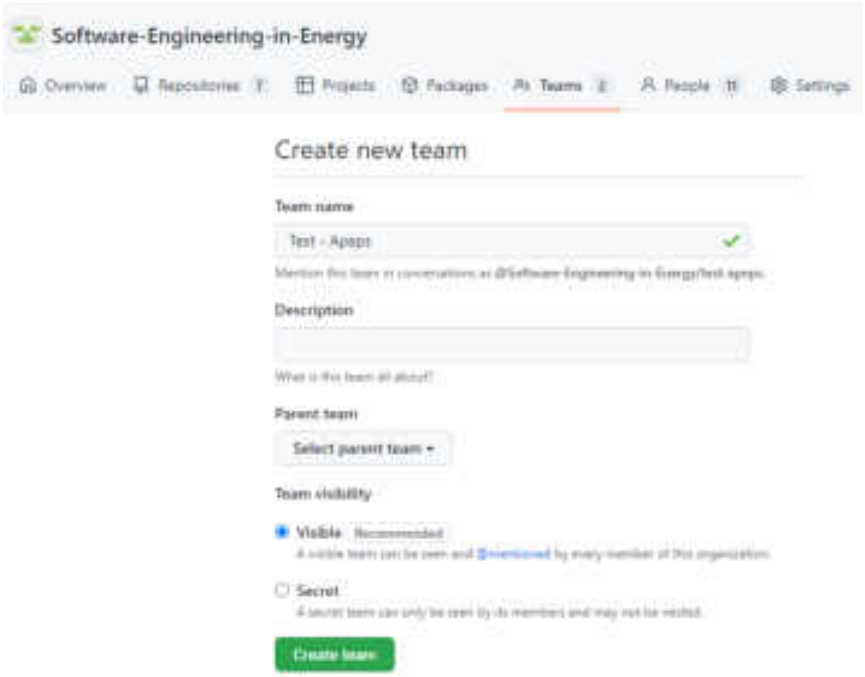


Рис. 6.11. Створення команди

Наступна сторінка після реєстрації команди відкриває можливість вести дискусію та з лівої сторони має можливість додавати “+” користувачів

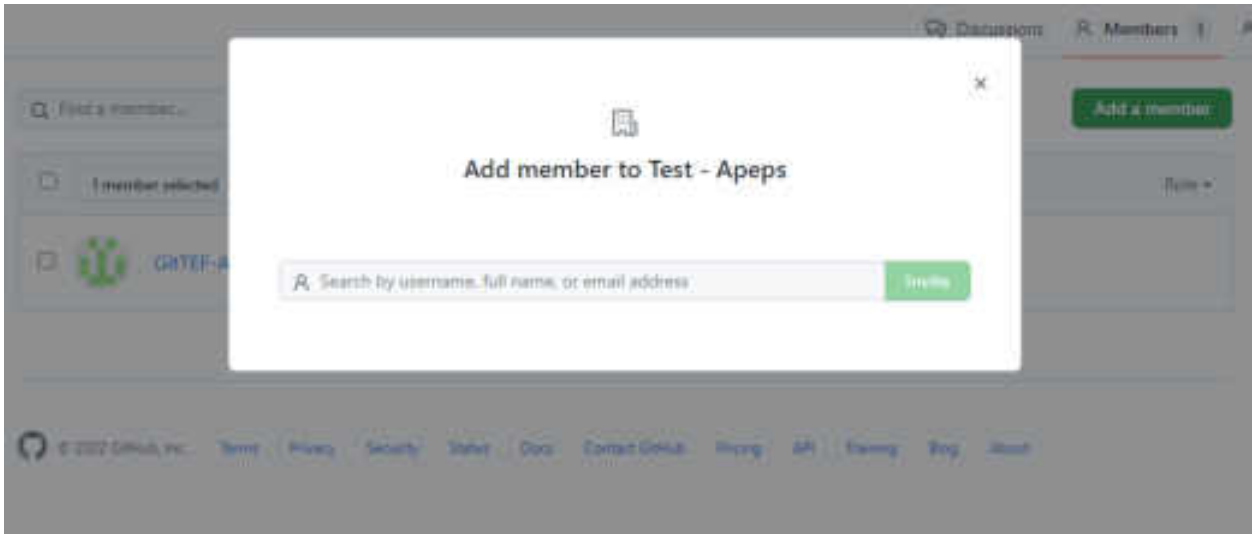


Рис. 6.12. Додавання користувачів в команди

Коли ви когось запрошуєте до команди, вони отримають листа, що повідомить їм про запрошення.

6.3. Створення репозиторію

На головній сторінці в лівій частині є можливість обрати репозиторій, або створити новий обравши 

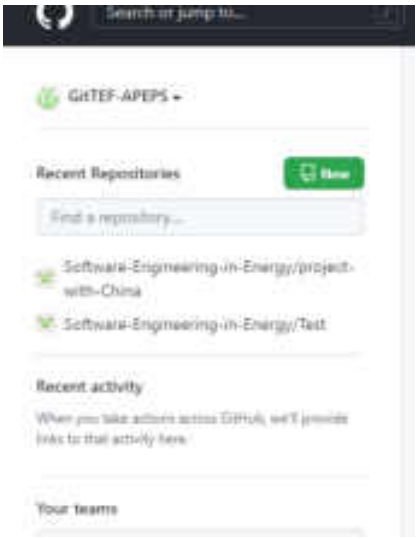


Рис. 6.13. Створення репозиторію

На новій сторінці відкривається можливість обрати показники нового репозиторію – назву, опис (не обов’язковий), публічний чи приватний, READMI file, **.gitignore** (які файли будуть проігноровані при **commit**), чи є ліцензія та створити репозиторій.

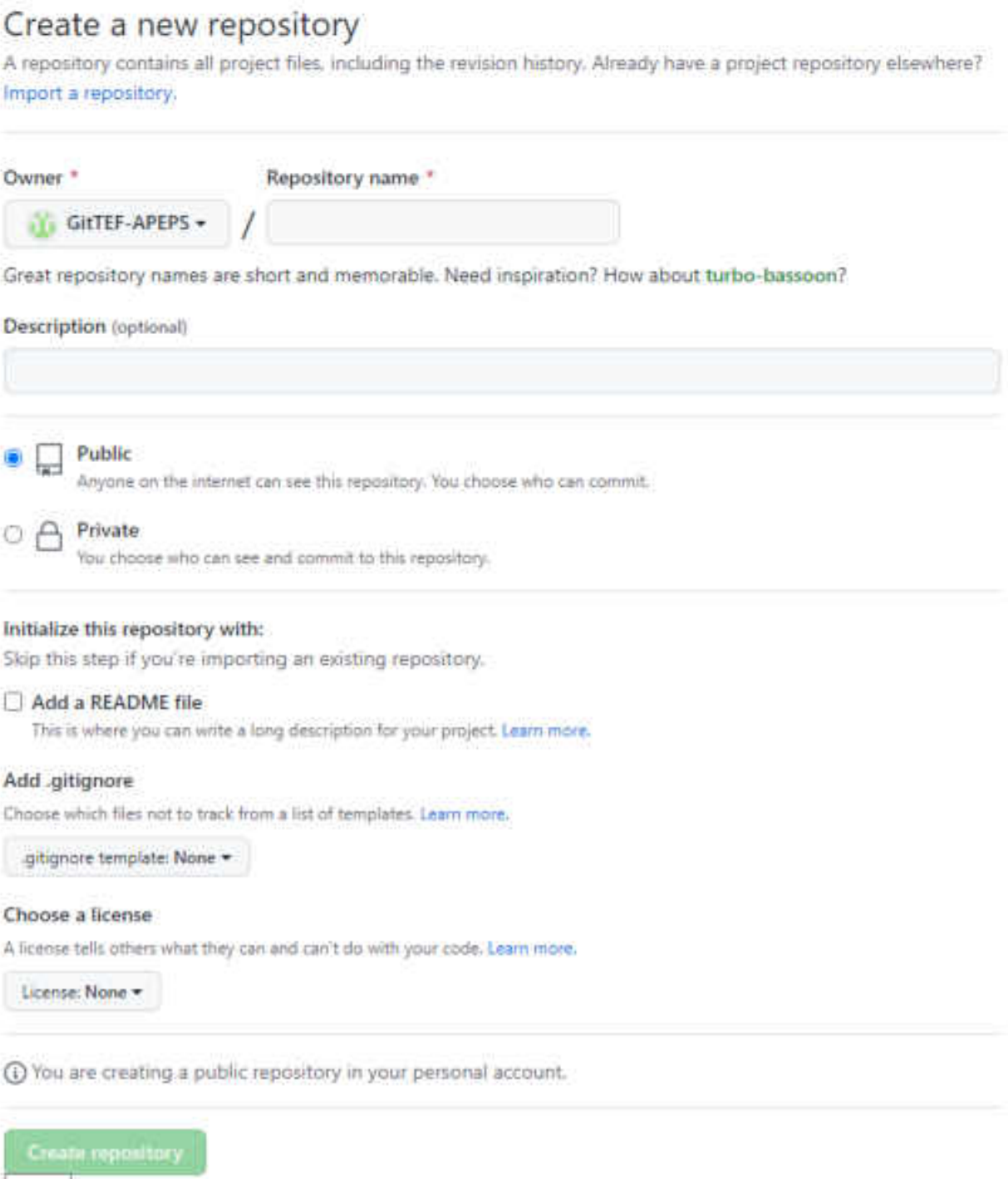


Рис. 6.14. Сторінка створення репозиторію

6.4. Ролі

Якщо увійти до організації то налаштування - Settings, дають змогу перейти до сторінки де ви можете обрати Repository roles (ролі в репозиторії) і ознайомитись з ролями, що призначені за замовчуванням чи створити нові ролі за вашими вимогами.

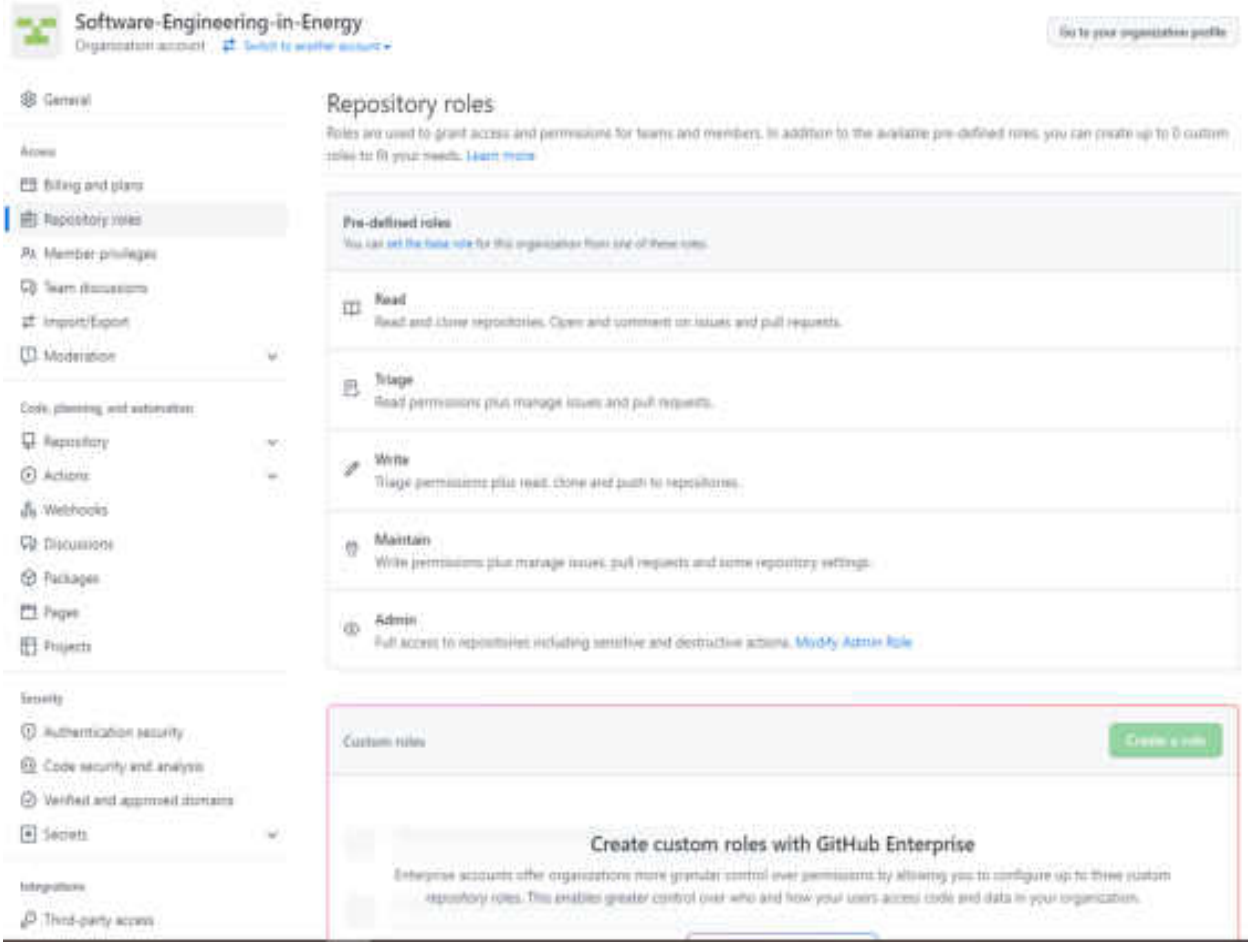


Рис. 6.15. Ролі в репозиторії

Ролі в організації:

Read - Ви можете читати та клонувати сховища, відкривати та коментувати проблеми і запити на вилучення.

Triage – Ви маєте дозвіл на читання та керувати проблемами та запитами на отримання.

Write – Ви можете сортувати дозволи, а також читати, клонувати та надсилати до репозиторіїв.

Maintain – Ви маєте дозволи на запис, а також керування проблемами, запити на отримання та деякі налаштування сховища.

Admin – Ви маєте повний доступ до репозиторіїв включаючи конфіденційні та деструктивні дії. Також ви маєте можливість відредагувати роль **Admin.**

Якщо ви зайдете на сторінку певного репозиторію в організації то шляхом вибору налаштувань

Рис. 6.16. Налаштування

ви маєте можливість переглянути учасників обравши

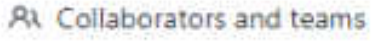


Рис. 6.17. Співробітники та команди

і опинитесь на сторінці, де можливо обрати ролі для кожного учасника.

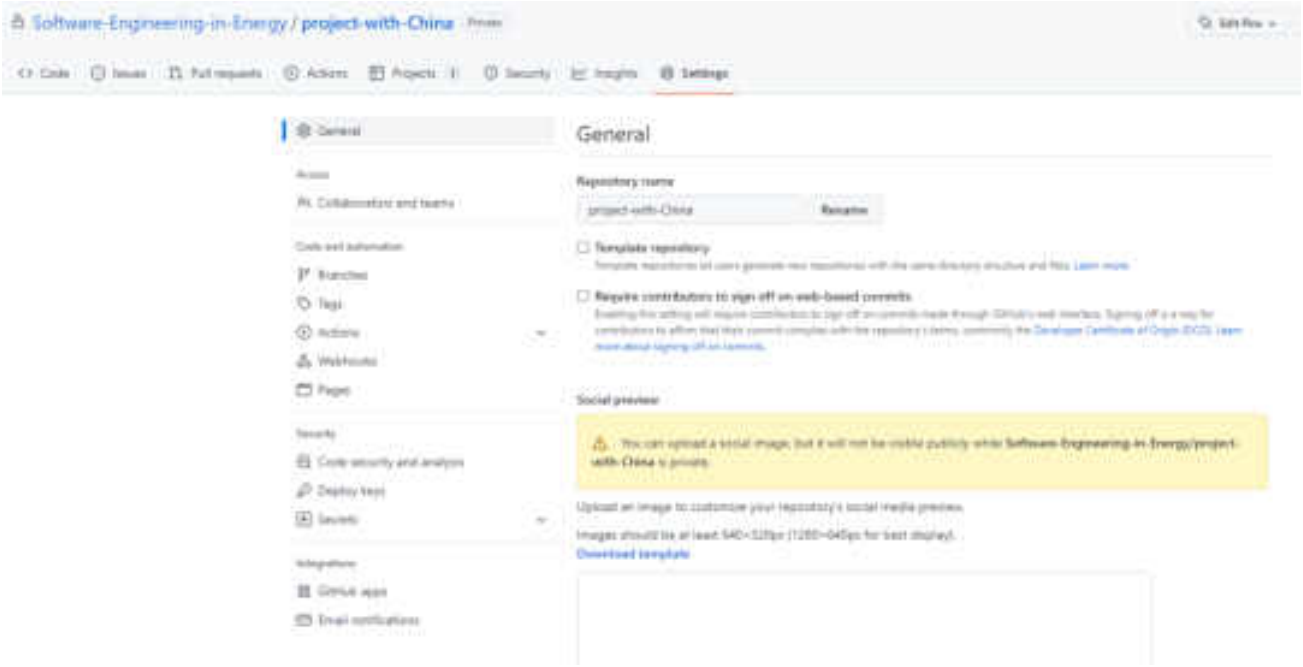


Рис. 6.18. Перехід до перегляду учасників репозиторію

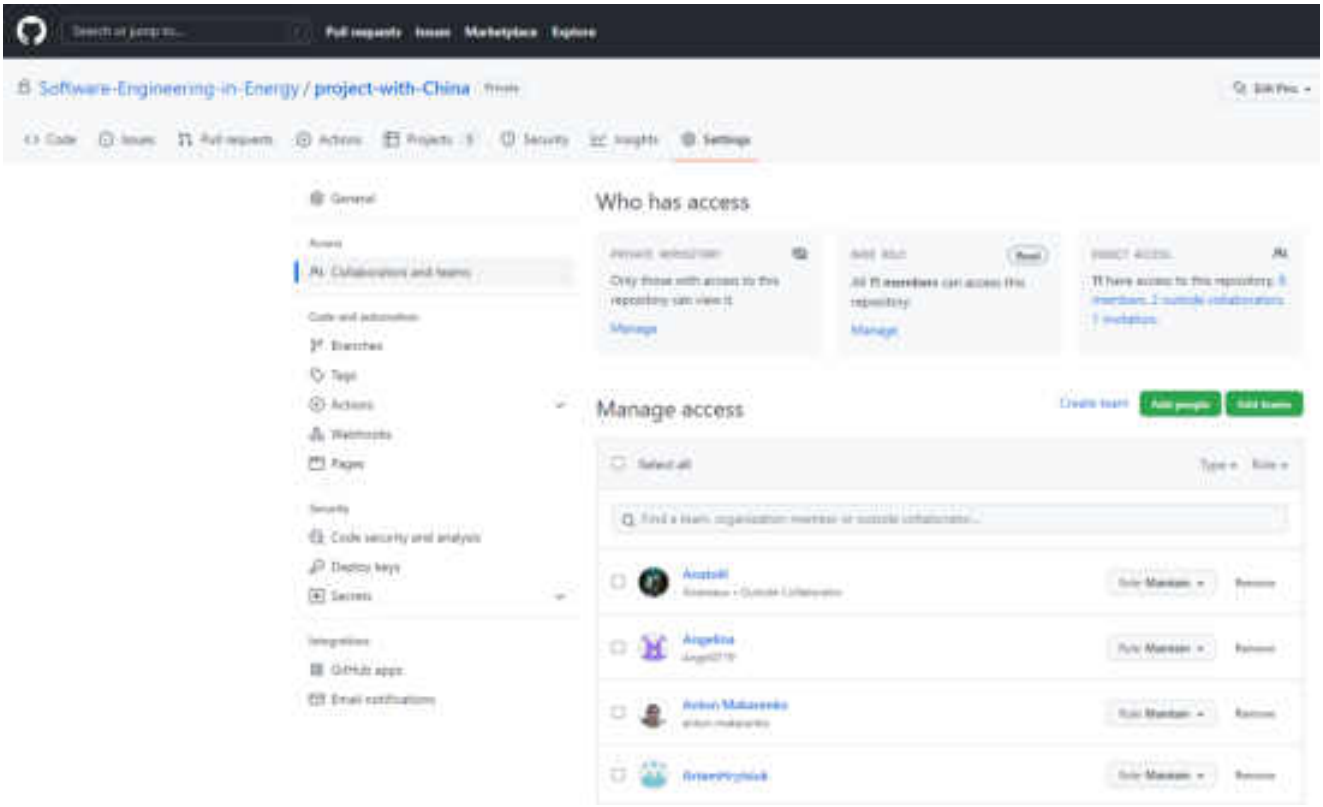


Рис. 6.19. Управління доступом учасникам репозиторію

Ролі в репозиторії:

Read - Ви можете читати та клонувати це сховище, відкривати та коментувати проблеми і запити на вилучення.

Triage (Сортування) – Ви можете читати та клонувати це сховище. Також ви можете керувати проблемами та витягувати запити.

Write – Ви можете читати, клонувати та надсилати в це сховище. Також можете керувати проблемами та витягувати запити.

Maintain – Ви можете читати, клонувати та надсилати в це сховище. Також ви можете керувати проблемами, запитами на отримання та деякими налаштуваннями сховища.

Admin – Ви можете читати, клонувати та надсилати в це сховище. Також можете керувати проблемами, запитами на отримання та налаштування сховища, зокрема додавати співавтора.

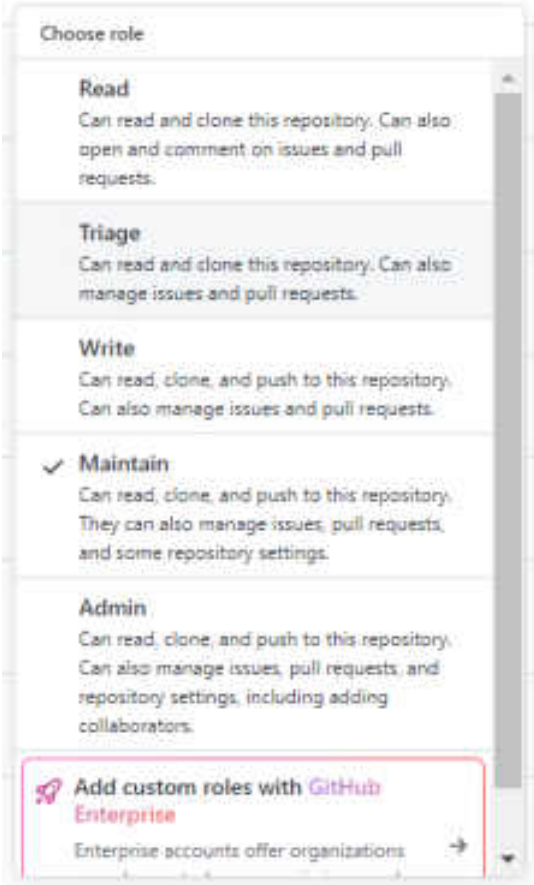


Рис. 6.20. Ролі в репозиторії

Більш детально як власники організацій можуть назначати ролі окремим особам та групам, надаючи їм різні набори дозволів в організації та як налаштувати доступ до кожного репозиторію в вашій організації, призначити детальовані ролі,

надаючи людям доступ до необхідних їм функцій та задачам можна ознайомитись в наступних джерелах [8, 9].

Питання до розділу 6

1. Організаційна основа GitHub?
2. GitHub. Опишіть організації, команди, репозиторії?
3. GitHub. Ролі в організації?
4. GitHub. Ролі в репозиторії?

РОЗДІЛ 7. СКВ В ІНТЕГРОВАНИХ СЕРЕДОВИЩАХ РОЗРОБКИ

7.1 . GitHub в Visual Studio

7.1.1. Клонування репозиторію

Сучасні інтегровані середовища розробки передбачають взаємодію з СКВ. В **Visual Studio** це можна зробити або при запускі або безпосередньо в інтегрованому середовищі розробки. Для цього виконується наступний порядок дій:

1. При запуску потрібно обрати розділ «**клонувати репозиторій**» (**Clone a repository**)

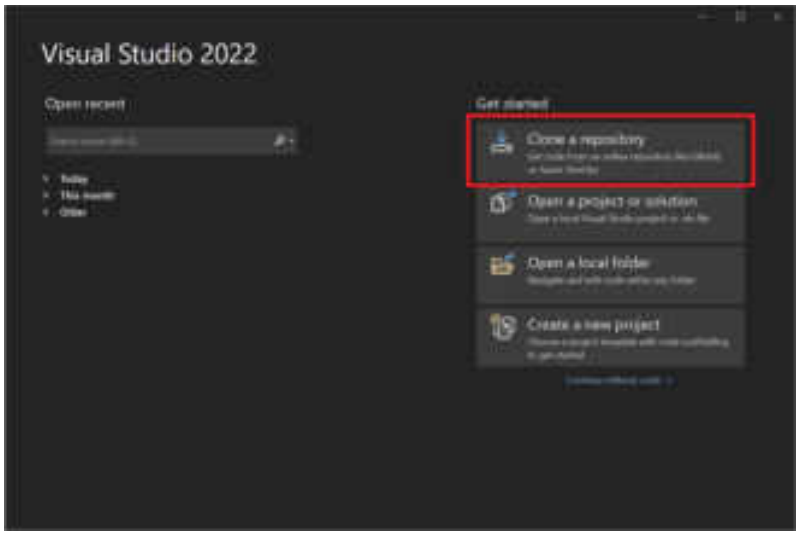


Рис. 7.1. Clone a repository

2. Ввести чи вказати розташування репозиторію, а потім виконати «**Клонувати**» (**Clone**).

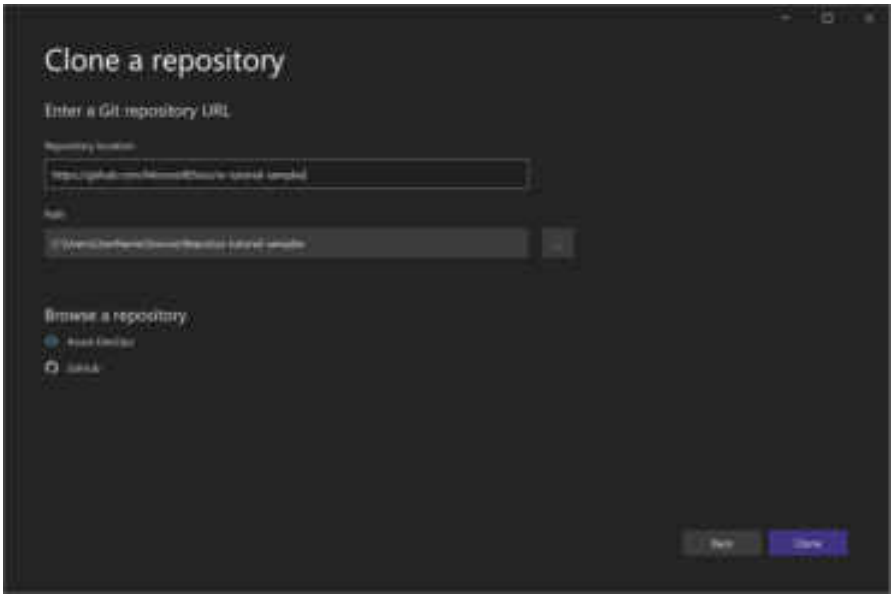


Рис. 7.2. Clone

3. Якщо з’являється діалогове вікно «**Відомості про користувача Git**» з запитом на вхідну інформацію можливо або оновити дані за замовчуванням або добавити нові.

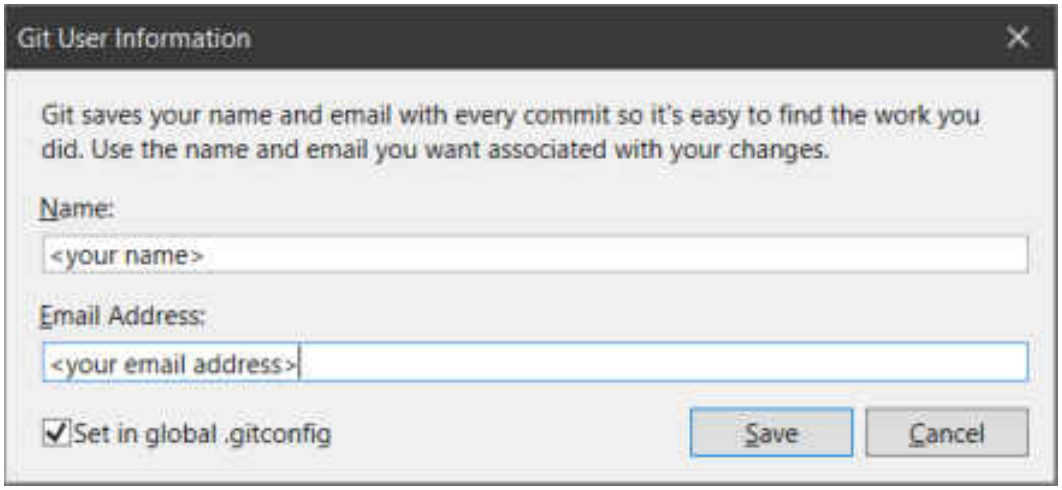


Рис. 7.3. Відомості про користувача Git

При виборі «**Зберегти**»(**Save**) інформація поновлюється в **GITCONFIG**-файл.

7.1.2. Перегляд файлів в оглядачі рішень

Оглядач рішень дозволяє загрузити рішення проекту з репозиторію з допомогою вікна «**Представлення папок**».

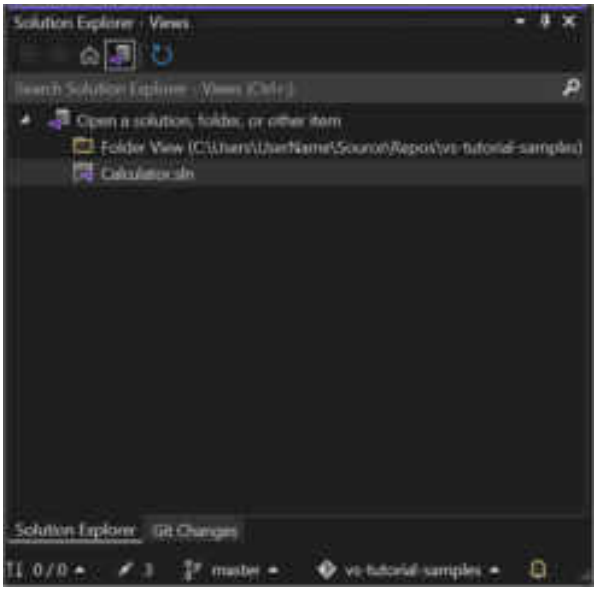


Рис. 7.4. Представлення папок

Двійне клацання файлу **SLN** у вікні «**Подання рішення**» дозволяє переглянути рішення проекту. Також кнопка «**Переключити представлення**» і потім вибір **program.cs** дозволяє переглянути код рішення.

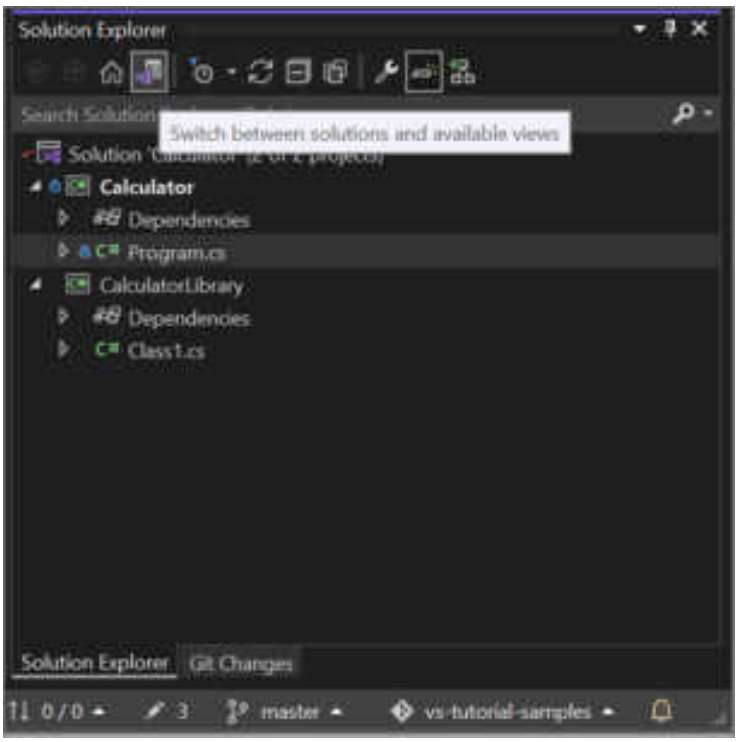


Рис. 7.5. Вибір перегляду коду рішень

Відкриття локального проекту з репозиторію **GitHub,** клонованого раніш.

1. При запуску **Visual Studio** в початковому вікні при виборі «**Відкрити проект чи рішення**» (**Open a project or solution**) відкривається провідник, що дозволяє пошуком вибрати рішення або проект та відкрити його.

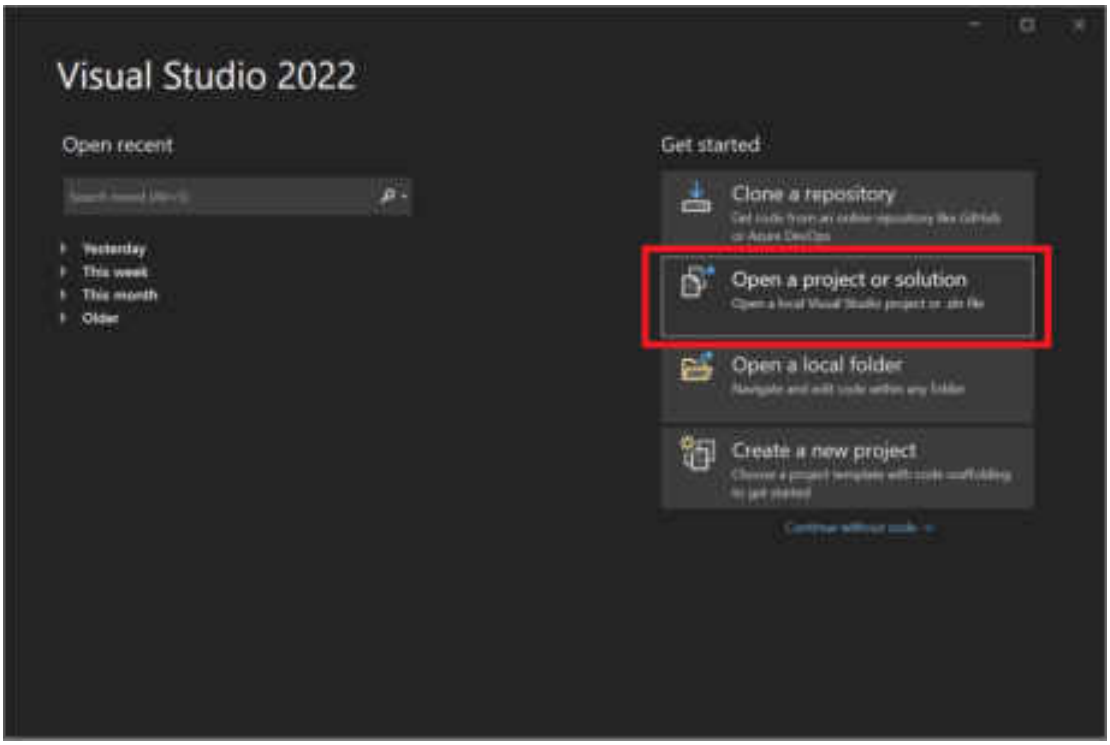


Рис. 7.6. Відкриття проекту чи рішення

7.1.3. Використання інтегрованого середовища розробки Visual Studio

В інтегрованому середовищу розробки **Visual Studio** для взаємодії з папками та файлами можна застосовувати також або меню **Git** або елемент керування «**Вибір репозиторію**»

7.1.4. Клонування репозиторію та відкриття проекту

1. В інтегрованому середовищі розробки **Visual Studio** відкрийте меню **Git** та виберіть «**Клонувати репозиторій**» (**Clone Repository**).

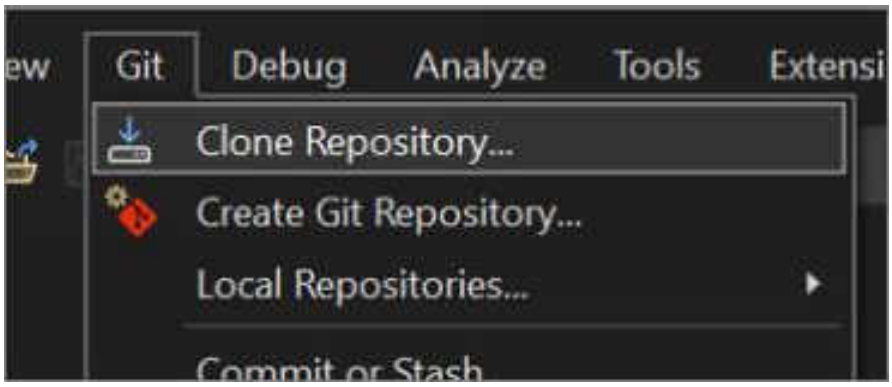


Рис. 7.7. Клонування репозиторію

2. Далі, щоб підключитися до репозиторію необхідно слідувати інструкції на екрані.

7.1.5. Відкриття локальних папок та файлів

1. В інтегрованому середовищу розробки **Visual Studio** відкрийте меню Git та оберіть «**Локальні репозиторії**» (**Local Repositories**) і потім «**Відкрити локальний репозиторій**» (**Open Local Repository**).

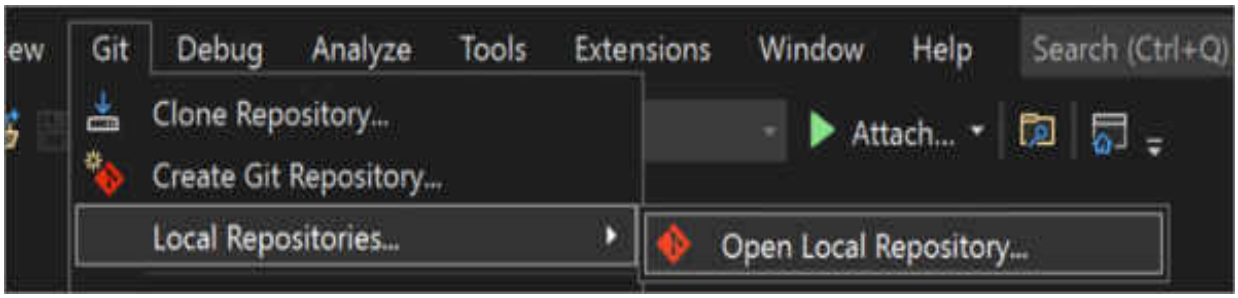


Рис. 7.8. Відкриття локального репозиторію

Ту ж задачу можна виконати в «**Оглядачі рішень**». Для цього необхідно обрати елемент керування «**Обрати репозиторій**», та клацанням знак з багато

крапкою поряд з полем «**Фільтрувати репозиторій**» і потім обрати «**Відкрити локальний репозиторій**»

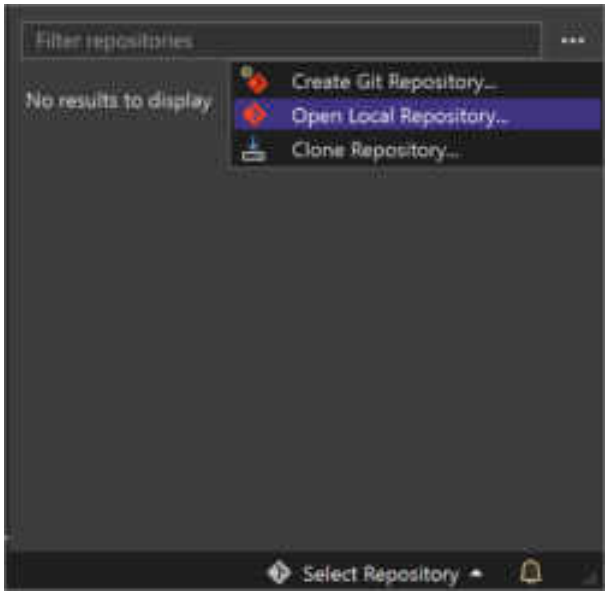


Рис. 7.9. Вибір “Open Local Repository…”

2. Далі, щоб відкрити файли та папки слідуйте за інструкціями на екрані.

7.2. GitHub в PyCharm

Щоб настроїти роботу **GitHub** в **PyCharm** необхідно зайти через меню «**Файл**» в настройки «**Settings…**» і далі в розділ «**Version Control**» на сторінці «**GitHub**»:

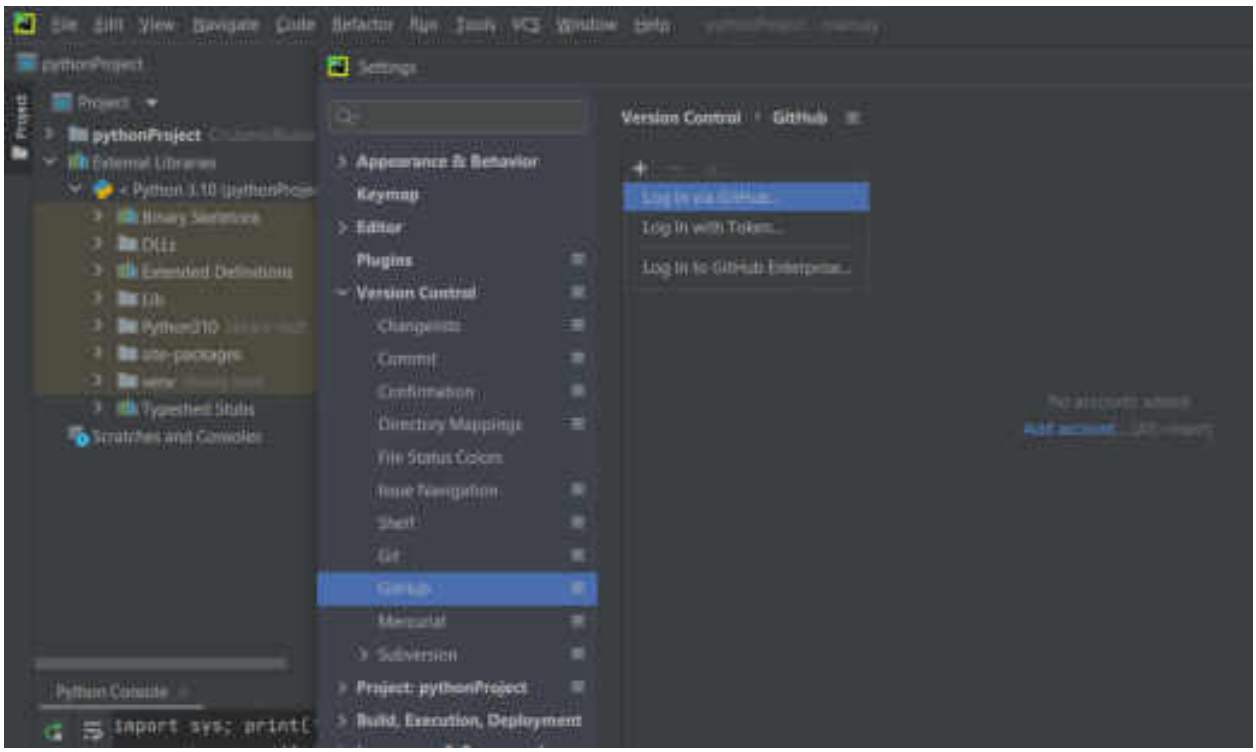
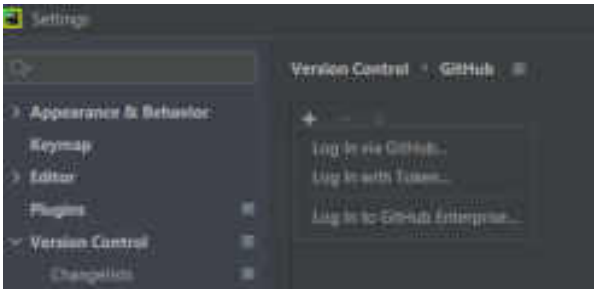


Рис. 7.10. GitHub в PyCharm

Налаштування можна виконати двома шляхами меню…

1. підменю «+»



2. підменю «Add account…»

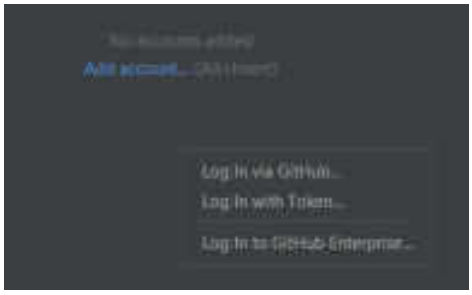


Рис. 7.11. Налаштування

Налаштування з’єднання шляхом

Рис. 7.12. Логування до GitHub

потребує авторизації

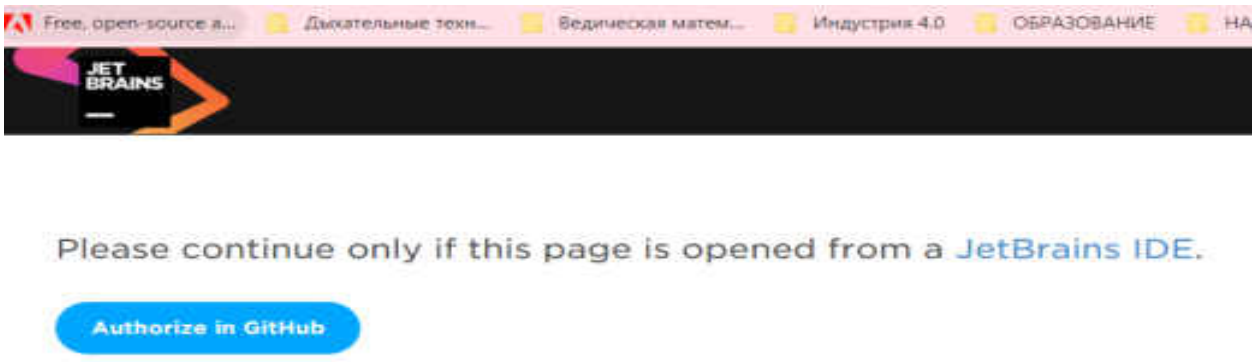


Рис. 7.13. Запит на авторизацію.

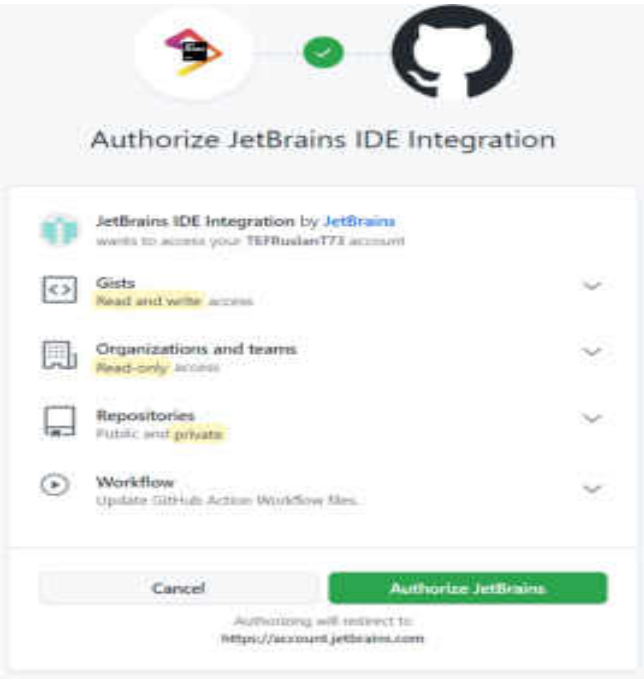


Рис. 7.14. Авторизація

Далі згідно інструкції на екрані.

Налаштування за токеном

Рис. 7.15. Логування за токеном

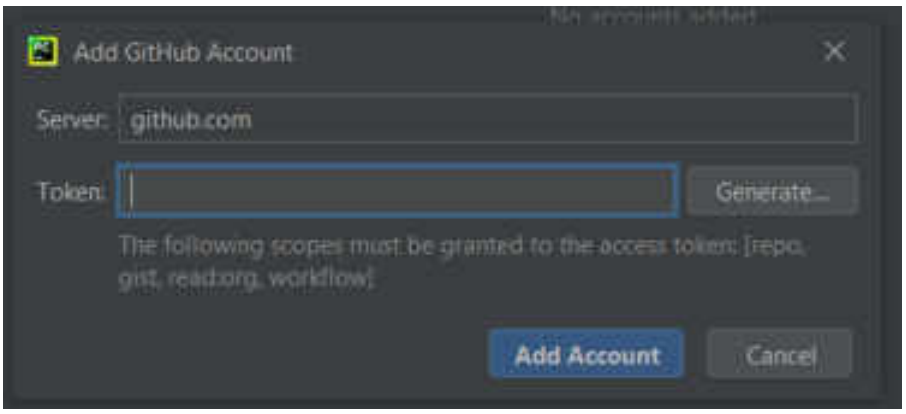


Рис. 7.16. Налаштування з'єднання токеном

Налаштування "Log In to GitHub Enterprise…" подібне "Log In with Token…".

Для праці з різними СКВ передбачено окреме меню робота з яким виконується згідно інструкцій на екрані.

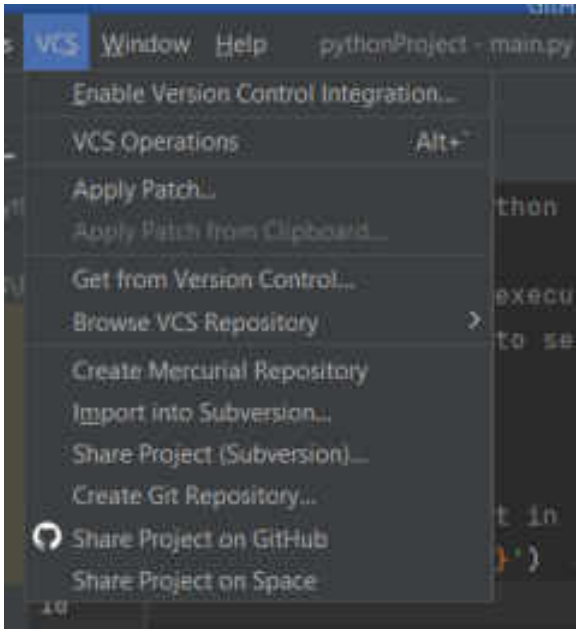


Рис. 7.17. Вибір СКВ

Питання до розділу 7

1. GitHub в Visual Studio? Опишіть.
2. GitHub в PyCharm? Опишіть.

РОЗДІЛ 8. ДОДАТКОВІ МОЖЛИВОСТІ GitHub

Хоча більшість з фахівців IT користуються сервісом практично щодня, не всі знають, що він має багато фішок, які допомагають полегшити роботу або рутинні операції. Наприклад, отримання публічного ключа з **URL**; відстеження того, з яких сайтів користувачі приходять до репозиторію; правильний шаринг посилань на файли, що живуть у репозиторіях **Github**; гарячі клавіші тощо.

8.1. Можливості URL

Філософія **Github** – бути простим, але водночас дуже гнучким інструментом. Саме з цієї причини багато функцій недоступні з інтерфейсу користувача, але доступні через URL-параметри.

8.1.1. Доступ до публічних ключів

Якщо ви налаштовуєте **ssh-**доступ колегам, які мають обліковий запис на **Github**, найпростіше зробити це за допомогою публічних ключів, отриманих прямо з **Github** через **URL**: **https://github.com/<user_name>.keys** (наприклад, **https://github.com/defunkt.keys**). Якщо у вас налаштовані процеси CI з використанням цієї можливості, у вас завжди будуть актуальні публічні ключі.

8.1.2. Доступ к diff'ам та patch'ам

Якщо додати **.diff** або **.patch** до кінця **URL-**сторінки з **commit** або **pull request**, можна отримати цей висновок у форматі **unix-**утиліт **diff** або **patch** (наприклад, за допомогою **URL** у такому форматі **https://github.com/tars/tars/commit/07902a956da92e6a616a69d3b3f0f9276f0c13fe.diff** - зверніть увагу на **".diff"** в кінці). Іноді зручно таким чином отримати набір змін не відходячи від каси та відправити його до **Slack** або по **email**.

8.1.3. Шаринг URL-посилань на файли у репозиторіях

Через динамічну природу проектів **commit**-и в майстер додаються постійно, вміст файлу в майбутньому може змінитися або файл може бути зовсім видалений, тому звичайне копіювання адресного рядка браузера тут не підходить. Щоб отримати постійне посилання на поточну версію файлу (**permanent link** - так

називає їх **Github**), треба замість назви гілки **URL** використовувати хеш **commit**-а. Так, це досить незручно, тож фахівці з **Github** зробили гарячу клавішу. Натисніть "**y**" під час перегляду файлу - URL у браузері буде змінено на **permanent link**.

8.1.4. Виключення пробільних символів під час перегляду diff

Додайте **?w=1** в URL при перегляді різниці файлів, і якщо відмінності були тільки в пробілах, це більше не відволікатиме увагу:

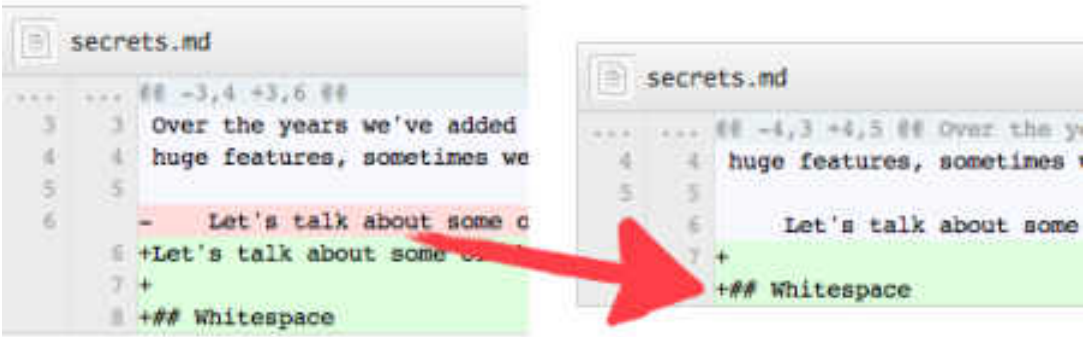


Рис. 8.1. Виключення пробільних символів під час перегляду **diff**

8.1.5. Підсвічування певного блоку коду

Під час перегляду файлу можна натиснути на будь-який його рядок і надіслати кому-небудь посилання на вказане місце. Можна виділити набір рядків, додавши в URL після номера рядка останній номер рядка діапазону через знак мінус, ось так (зверніть увагу на L30-L32 в кінці URL): **https://github.com/torvalds/linux/blob/9256d5a308c95a50c6e85d68249a /include/uapi/linux/netfilter_bridge/ebt_among.h#L30-L32**

8.1.6. Порівняння ревізій гілок у репозиторії

Під час створення **pull request** відразу видно, які зміни потраплять у цільову гілку внаслідок злиття. Але є неочевидна можливість сторінки порівняння гілок: ми можемо переглянути всі зміни у гілці за певний час, наприклад, за два тижні - **https://github.com/github/linguist/compare/master@%7B2week%7D...master.** Іноді корисно зробити закладку в браузері на сторінку з порівнянням і якщо щось йде не так, оперативно перевіряти, що було змінено в коді, наприклад, за останню добу (**master@{1day}...master**). Під капотом цієї фічі використовують стандартний **git diff**, тому можна використовувати будь-який формат часу, який сприймає **git**. Більше інформації про порівняння гілок, **commit**-ів тощо дивіться в документації.

8.2. Гарячі клавіші

У **Github** багато гарячих клавіш, розглянемо найкорисніші.

8.2.1. Порівняння ревізій гілок у репозиторії

Якщо натиснути «**t**» під час перегляду репозиторію, з'явиться рядок пошуку файлів, в який можна ввести частину шляху та вибрати необхідний файл, заощадивши хвилини блукання репозиторію в пошуках.

8.2.2. Швидкий перехід до певного рядка у файлі

Натисніть «**l**» під час перегляду файлу – з'явиться маленьке вікно, щоб ввести номер рядка.

8.2.3. Швидкі переходи до розділів Github

Ці комбінації легко запам'ятати: вони починаються з префікса «**g**» (**go**), і літера, що наступає за ним, вказує на місце призначення. **gp** – перехід до списку **pull request**-ів, **gi** – список **issues, gn** – сторінка нотифікацій тощо.

Цей список гарячих клавіш далеко не сповнений. Щоб переглянути всі доступні хот-кеї, натисніть "**?**".

8.3. Тикети (issues) та пулл-реквести (pull request-и)

Тикети (**issues**) і пулл-реквести (**pull request-и)** - постійні помічники під час проекту. **Github** надає багато засобів для роботи з ними. Можна використовувати систему міток і кожному **issue** або **pull request** проставляти відповідні їм ознаки, наприклад, **feature, bug, documentation**. Також можна створити **milestones** і вказувати, над яким **issue** у якій версії буде виконуватись робота.

8.3.1. Автоматичне закриття issues за допомогою commit-ів

Якщо текст **commit**, що попадає в основну гілку репозиторію, містить слова **fix/resolve/close** у різних варіаціях та номер **issue** після символу «#», тоді **issue** із відповідним номером буде закритий.

Тобто **commit** із таким повідомленням:

```
$ git commit -m "Fix screwup, fixes #12"
```

Рис. 8.2. Закриття **issue commit**-ом

Приведе до наступного результату

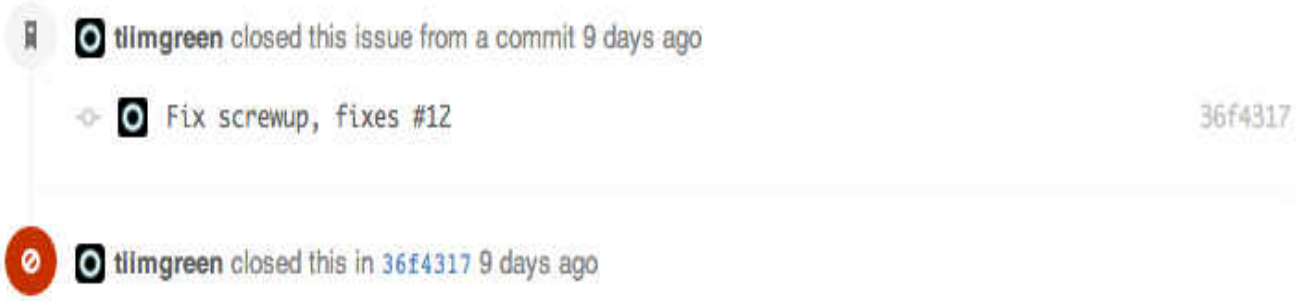


Рис. 8.3. Результат закриття **issue commit**-ом

8.3.2. Пошук найулюбленіших pull requests та issues

Github додав реакцію на коментарі. Тому сортування може враховувати кількість **emoji** у реакціях. Відповідно, тип **emoji** для сортування можна вибрати в меню "**Sort**":

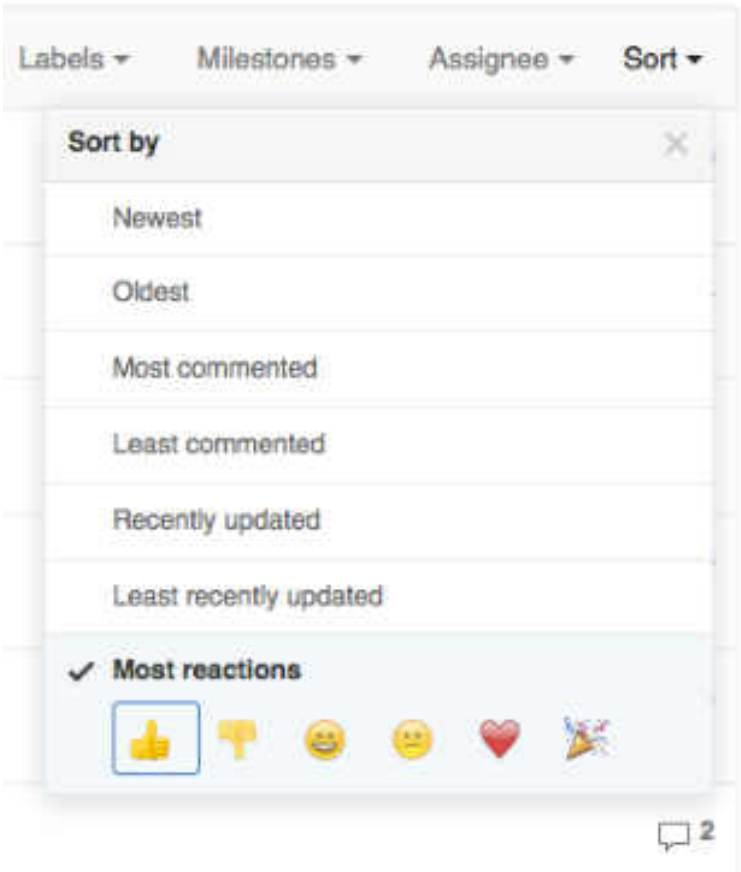


Рис. 8.4. Вибір додаткової можливості у сортуванні коментарів за типом **emoji**

8.4. Github markdown

У **Github** використовується надмножина **markdown-розмітки**, яка використовується в коментарях та **markdown-файлах**. З його допомогою можна робити крос-посилання між репозиторіями або **issues**, створювати **to-do** списки, робити підсвічування коду.

8.4.1. Крос-посилання

Якщо під час створення **issue** треба згадати інший **issue**, можна ввести знак «#» — з'явиться меню, з якого можна вибрати необхідний **issue**; також можна написати номер **issue** самостійно. При збереженні коментаря з посиланням, відповідне повідомлення з'явиться на сторінці згаданого **issue**. Якщо ви згадали інший **issue** з приватного репозиторію, згадку буде видно лише тим, хто має доступ до цього репозиторію. При цьому можна зробити посилання не тільки на **issue**, але і на конкретний **commit**, просто вказавши його хеш - **Github** автоматично перетворить його на правильне посилання.

8.4.2. Підсвічування синтаксису

Для того щоб підсвітити шматок коду у вашій **markdown-розмітці**, введіть назву мови після потрійної зворотної лапки перед блоком коду та потрійну зворотну лапку в кінці блоку:

Рис. 8.5. Підсвічування синтаксису коду

8.4.3. To-do списки

Синтаксис у списку **to-do** має такий вигляд:

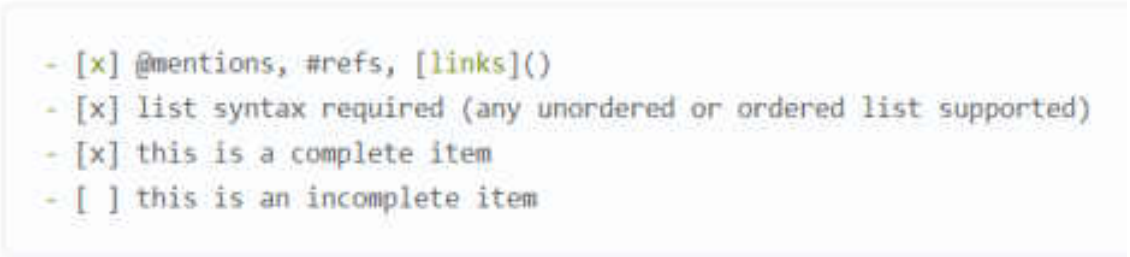


Рис. 8.6. Вигляд синтаксису у списку **to-do**

При збереженні коментаря зі списком він перетворюється на повноцінний список завдань із чек-боксами, які можна відзначати (при цьому відповідний [x] з'являється у **markdown-розмітці** автоматично). Ось гарний приклад роботи зі списками: **https://github.com/neovim/neovim/pull/243**

Інші можливості у довідці **Github** – Github flavored markdown.

8.5. Акаунт

8.5.1. Двофакторна автентифікація та безпека

Ця опція знаходиться в **Settings -> Security**. До речі, там можна подивитися всю історію дій, пов'язаних з безпекою акаунта (список минулих сесій, **ip-**адреси і та інші).

8.5.2. Прив'язка кількох поштових адрес до одного облікового запису

Якщо ви використовуєте кілька адрес для роботи та особистого листування, вам може бути зручно розмежовувати їх і на рівні **git**, роблячи **commit**-и в робочі проекти з одним **email**, а в домашні - з іншим. Можна завести спеціальну скриньку для всіх комунікацій на **Github**. У всіх випадках виникає проблема: метрики сервісу не сприйматимуть **commit**-и з поштовою скринькою, відмінною від того, з якої відбулася реєстрація. Щоб уникнути цього, вкажіть усі поштові адреси, які будуть асоційовані з вашим профілем (**Settings - > Emails**). Майте на увазі, що актуалізація інформації займе деякий час.

Якщо ви підтвердите ці email-и, то за їх допомогою можна буде отримати доступ до вашого облікового запису (відновлення пароля). Якщо ж ви хочете зробити просто асоціацію email-у з вашим обліковим записом без фічі відновлення пароля - просто не підтверджуйте цей ящик.

8.5.3. Збережені відповіді (Saved replies)

Ця функція є особливо корисною для тих, хто часто пише однотипні коментарі до **pull requests** або **issues**. Наприклад, про те, що **pull request** повинен відповідати правилам, визначеним у **contributing.md**. Можна зберегти набір відповідей та швидко вставляти їх через меню редактора **Github**:

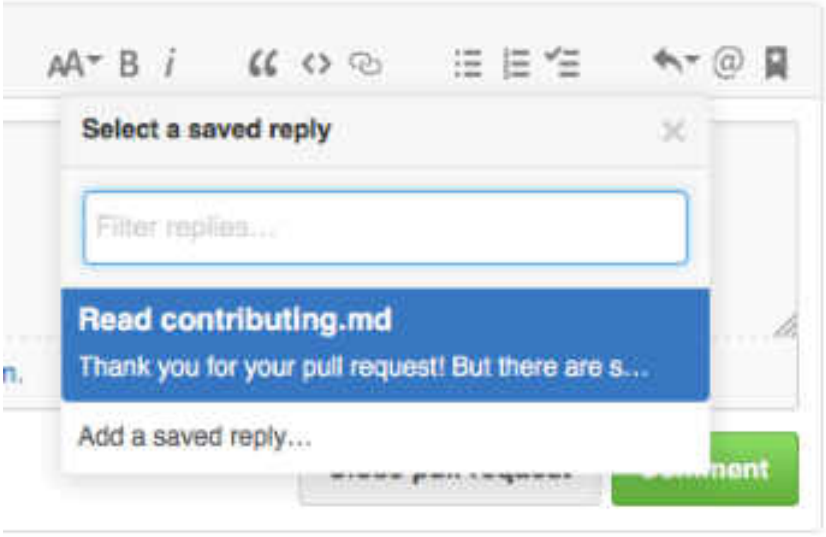


Рис. 8.7. Збереження відповіді коментаря

Сторінка додавання відповідей: **Settings -> Saved replies**.

8.5.4. Згадки (mentions)

Ця популярна функція з механіки схожа на згадки на «Хабрхабрі» або «Твітері». При створенні коментаря введіть символ «@» і відразу після нього нікнейм користувача. У згаданого користувача на сторінці оповіщень з'явиться відповідне посилання на **issue** або **pull request**, в якому його згадали.

Згадки доступні не тільки для конкретного користувача, але й для групи користувачів організації. У цьому випадку після "**@**" треба ввести назву організації і через слєш назву команди - наприклад **@2gis/mamonts**. Додавання нової команди доступне за допомогою вкладки **Teams** на головній сторінці організації.

8.5.5. Відповіді на email-повідомлення Github

Якщо у вас у налаштуваннях включені **email-**повідомлення, ви можете брати участь у дискусіях на **Github**, не виходячи із поштового клієнта. Якщо відповісти на **email**, то від імені вашого користувача на **Github** буде створено коментар у відповідному **issue** або **pull request** з вмістом вашого відправленого листа.

8.5.6. Підписка публічної активності користувачів

Якщо вам цікавий певний користувач на **Github**, його можна зафоловити (кнопка **Follow** на сторінці профілю користувача) і потім бачити його публічну активність (**commit**-и, коментарі тощо) в загальному списку на головній сторінці сайту. Головна сторінка може показати тільки зріз за останні три дні, тому, якщо ви не хочете пропускати всю активність користувачів, вам знадобляться **rss-стрічки**, на які можна підписатися за допомогою посилання виду **https://github.com/<user_name>.atom**. Так само можна передплатити всю публічну активність організації (наприклад, **https://github.com/2gis.atom**).

8.6. Робота з репозиторіями

8.6.1. Службові директорії та файли Github

Якщо у корені вашого проекту є **readme-файл**, його вміст відображатиметься під списком файлів. Це знає багато хто. Але не всі знають, що якщо створити файл **contributing.md** з рекомендаціями щодо покращення проекту, посилання на нього буде доступне на сторінці створення **pull request** або відкриття **issue**:

Рис. 8.8. Перегляд посилання на файл **contributing.md**

Github має можливості створення шаблонів при відкритті тикетів (**issue)963**.

Створіть всередині кореня проекту файл з назвою **ISSUE_TEMPLATE**, і його вміст автоматично вставлятиметься в полі для введення при відкритті нового **issue**. Те саме працює і для **pull request**, у цьому випадку файл має називатися **PULL_REQUEST_TEMPLATE**. Файли можуть бути у форматі **markdown**.

Ось приклад шаблону з репозиторію **React** - **https://github.com/facebook/react/blob/20bcabb1ea4cf492ade240bd6915b4bd44f04895/.github/ISSUE_TEMPLATE.md**. Якщо спробувати відкрити **issue** у їхній репозиторії, поле для введення тексту вже буде заповнене корисною інформацією:

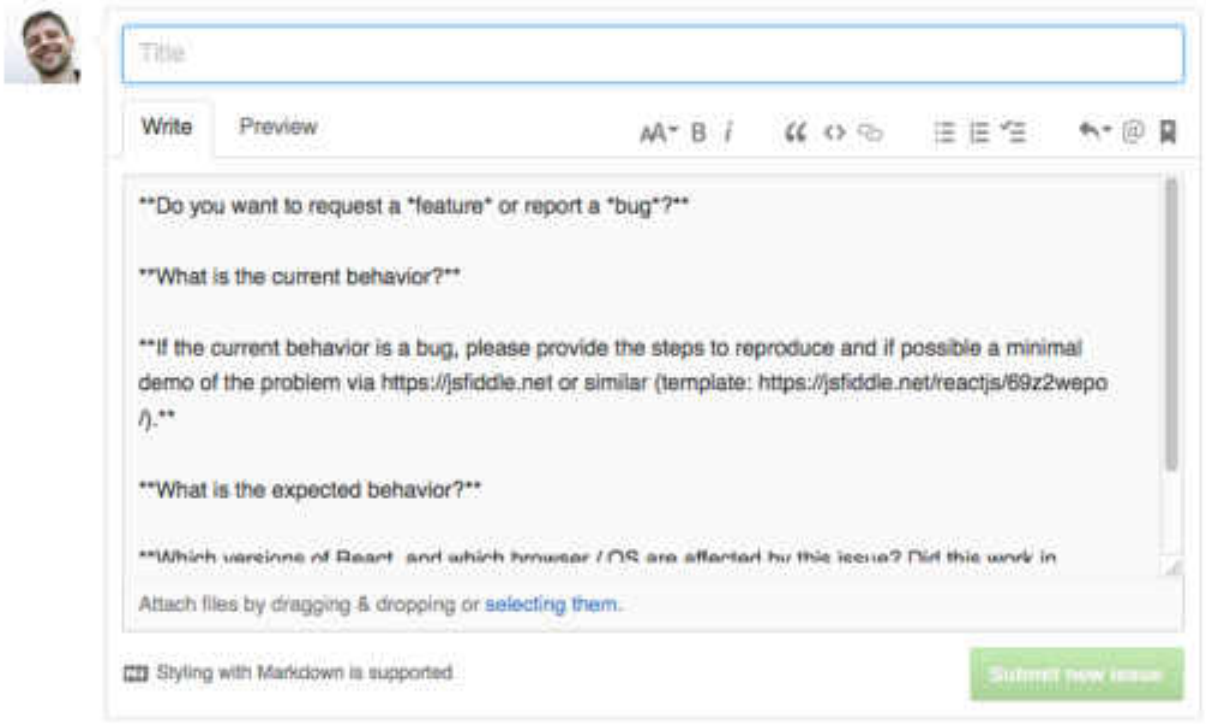


Рис. 8.9. Поле для введення тексту заповнене корисною інформацією при відкритті **issue**

Щоб не засмічувати корінь проекту службовими файлами, можна створити директорію **.github/** і помістити всі файли, пов'язані з **Github**, туди. Всі функції, пов'язані з цими файлами, працюватимуть так само, як раніше.

8.6.2. Статистика мов програмування

На головній сторінці репозиторію у вигляді кольорової смуги відображається статистика мов, що використовуються в даному репозиторії. Якщо натиснути на ній, відобразяться частки у відсотках:

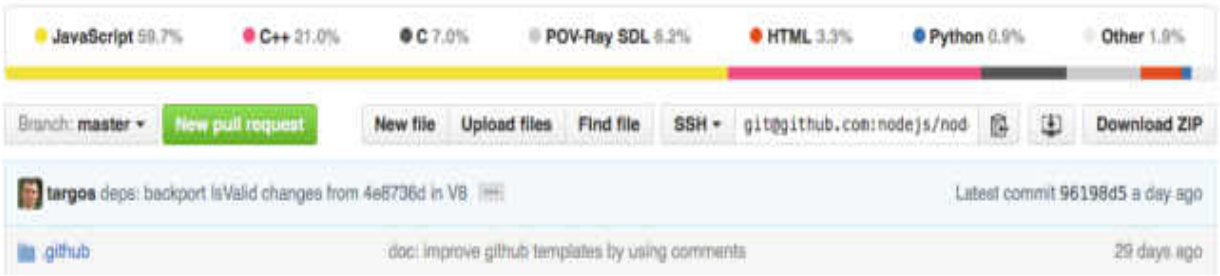


Рис. 8.10. Кольорова полоса статистики мов програмування репозиторію

Якщо ви використовуєте у своєму проекті якийсь великий **js**-фреймворк і не хочете, щоб він від'їв 70% у вашого улюбленого **Go/Python/Java/Ruby** у статистичних даних репозиторію, створіть директорію **vendors** і покладіть туди всі залежності, які не повинні враховуватися системою статистики **Github**. Або

перевірте наявність того файлу або директорії, якого ви хочете позбутися статистистичних даних тут — цілком можливо, що вам уже нічого не треба робити.

Так, якщо мови у проекті визначаються не так, як треба, то в цьому випадку варто подивитися у **linguist. Linguist** - це **ruby**-бібліотека, за допомогою якої **Github** збирає статистику з мов, що використовуються. У **readme** проекту описані різні способи перевизначення файлу, що визначається.

8.6.3. Метрики репозиторію

Github надає велику кількість метрик для відстеження роботи, що відбувається в репозиторії. Відповідні інструменти моніторингу знаходяться на вкладках **Pulse** та **Graph. Pulse** показує, що відбувалося у репозиторії у певний період часу. У розділі **Graph** різні показники відображені у вигляді графіків. У власників репозиторіїв у вкладці **Graph** також утворюється підпункт **Traffic**. За великим рахунком, це міні **google analytics** для репозиторію: в ньому можна відстежувати, скільки користувачів було у вашому репозиторії і звідки вони прийшли.

8.6.4. Створення нового репозиторію

При створенні репозиторію можна відразу вибрати, який **gitignore-**файл необхідний, яка ліцензія буде у проекту і чи потрібна заготовка для **readme-**файлу. Так ви заощадите трохи часу на початковій стадії підготовки проекту:

Рис. 8.11. Швидке налаштування при створенні репозиторію

Якщо вашого типу проекту немає в списку **gitignore**, тоді слід цю ситуацію покращити та запропонувати **pull request** у репозиторій **gitignore Github**.

8.7. Пошук коду

Ще одна неочевидна можливість **Github** — пошук коду по всіх репозиторіях: **https://github.com/search**:

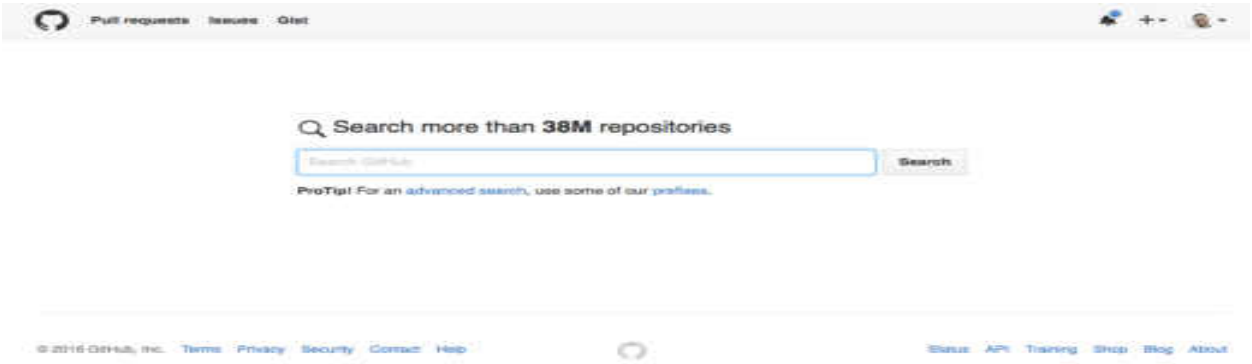


Рис. 8.12. Пошук коду по всіх репозиторіях

Як і в будь-якому серйозному пошуковому сервісі, можна перейти на сторінку розширеного пошуку та вказати уточнюючі параметри: наприклад, ім'я користувача, дату створення, мову, якою написаний код.

При пошуку діють обмеження, які варто враховувати, якщо ви ведете проект на **Github** і хочете, щоб сервіс міг проіндексувати ваш код (або навпаки, якщо не хочете): індексується лише головна гілка репозиторію (зазвичай це **master**); індексуються лише ті файли, які менші за 384 KB; індексуються ті репозиторії, у яких менше 500 000 файлів.

8.8. Командний рядок та Github

8.8.1. Hub

Hub - це консольна утиліта від творців **Github** (**git + hub = github**), мета якої - полегшити використання сервісу з шеллу. По суті, **hub** охоплює стандартний **git** та надає додаткові команди для роботи з репозиторіями, **pull requests** та **issues**.

Ось так можна клонувати репозиторій:

```
$ hub clone github/hub
```

Рис. 8.13. Клонування репозиторію

Зробити **fork**:

```
$ hub fork
```

Рис. 8.14. Fork

Відкрити **pull request**:

```
$ hub pull-request
```

Рис. 8.15. Pull request

Автори утиліти радять після установки зробити **alias** на **hub** при виклику **git (alias git=hub)**, оскільки команди **hub** не конфліктують зі стандартними командами **git**.

Більше інформації щодо роботи з утилітою можна знайти в **man** або на сайті проекту.

8.8.2. Pull requests вже у вашому репозиторії

Цікавий момент, про який мало хто знає. При створенні **pull request** всі зміни, що містяться в ньому, автоматично потрапляють у ваш репозиторій, так як **pull request** за великим рахунком - особлива гілка. Завдяки цій можливості **Github** може показувати зміни у **pull request**, навіть якщо вихідний форк був вилучений. Таким чином, у вас завжди є до них доступ. Щоб зміни з **pull request** потрапили у нову гілку (**new_branch**), треба виконати команду:

```
$ git fetch origin pull/<pr_num>/head:new_branch
```

Рис. 8.16. Pull request, для занесення зміни у нову гілку

де **<pr_num>** – номер **pull request** з **URL**.

8.9. User scripts

User script — це **JavaScript** користувача, який змінює певний сайт/веб-додаток, змінюючи його зовнішній вигляд і/або додаючи нові функції. Є багато скриптів, призначених для роботи з **Github**. У каталозі багато застарілих скриптів, але серед них трапляються досить сучасні та корисні.

8.9.1. Github Commit Whitespace

Github Commit Whitespace просто додає на сторінку порівнянь посилання, за допомогою якого можна швидко виключити із **diff** змінені пробільні символи:

Рис. 8.17. Виключення пробільних символів додаванням посилань на сторінку порівнянь

8.9.2. Github News Feed Filter

Github News Feed Filter буде особливо корисним для тих, хто моніторить (**watch**) багато активних репозиторіїв/користувачів. Цей скрипт додає на головну сторінку зі стрічкою подій фільтр, що дозволяє відображати тільки цікаву активність:

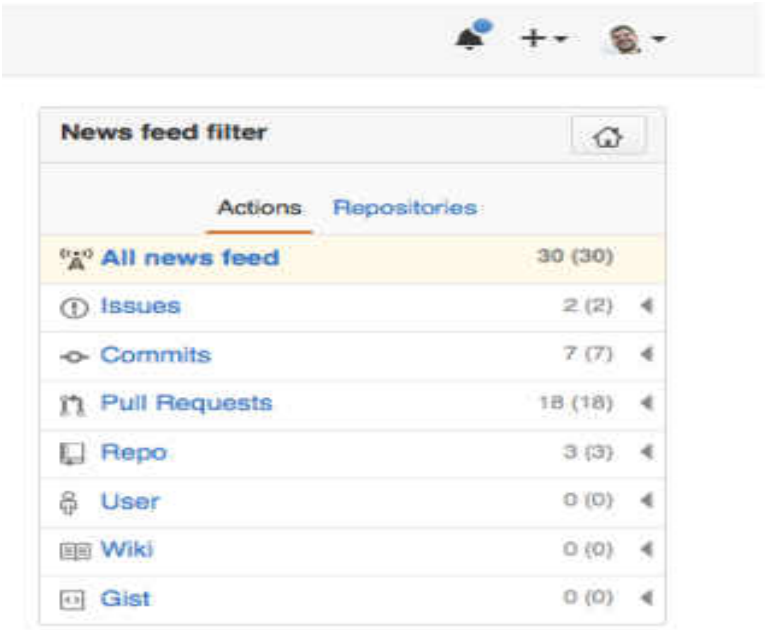


Рис. 8.18. Додавання фільтру відображення цікавих подій

8.10. Другорядні можливості

8.10.1. Gitfiti

gitfiti - генератор діаграм, в якому є встановлені шаблони зображень і можливість використовувати власні. **contribution graph** - умовне позначення квадратиками, відображує ваш вклад протягом року:

Рис. 8.19. Діаграма вкладу протягом року

Маніпулюючи git-репозиторієм, додаючи **commit**-и у певні дати у певній кількості, можна трохи урізноманітнити діаграму.

8.10.2. Заміна автора commit

У **git** є можливість виправлення автора **commit** через

```
$ git commit --amend --author="Linus Torvalds <torvalds@linux-foundation.org>"
```

Рис. 8.20. Зміна автора **commit**

В результаті **commit** у **Github** буде відображено з новим автором та його аватаркою:

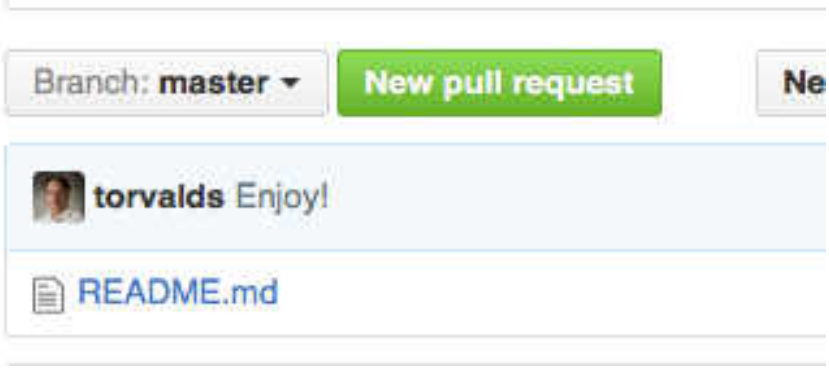


Рис. 8.21. Аватар нового автора **commit**

Github додав перевірку справжності автора **commit** за **GPG**-підписом. Якщо ви бачите слово «**verified**» поруч із **commit**, значить автор **commit** саме ця людина, а не хтось інший:

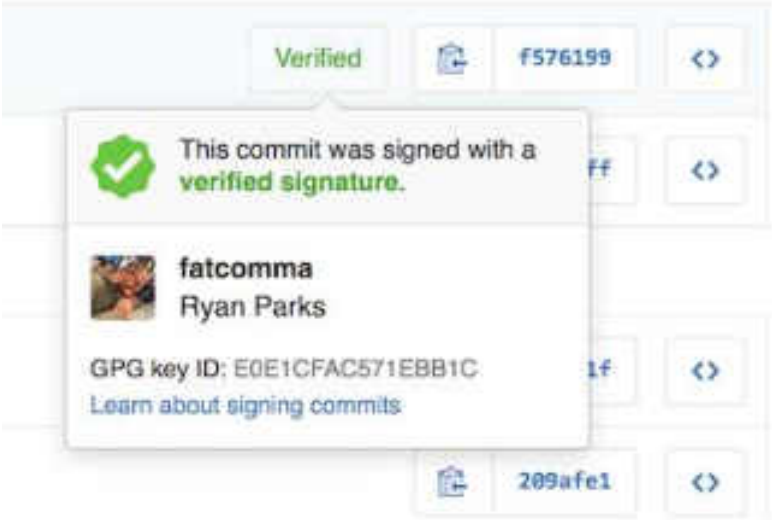


Рис. 8.22. Підтвердження автора за **commit** за **GPG**-підписом

8.11. Інші додаткові Github-ресурси

8.11.1. Рейтинги репозиторіїв

Джерело нових інструментів та бібліотек - сторінка **https://github.com/explore**. Тут розміщені різні добірки репозиторіїв, репозиторії,

які були додані до обраного тими людьми, на яких ви підписані, і, звичайно, найпопулярніші репозиторії останнім часом.

8.11.2. Статус сервісу

Досить рідко, але буває, що **Github** працює нестабільно. Основний ресурс, де можна дізнатися, що пішло не так - **https://status.github.com/.** Там можна знайти різні метрики, що відображають стан здоров'я **Github** і всі сповіщення про перебої в роботі сервісу.

8.11.3. Github pages

Github вміє хостити статичні сайти. Це дуже зручно, якщо вам потрібно зробити веб-документацію для вашого проекту або промо-сайт. Багато хто використовує **Github** для ведення особистих блогів. У найпростішому випадку достатньо створити у вашому **Github**-репозиторії гілку **gh-pages** з **index.html** усередині. Сторінка буде доступна за адресою у такому форматі: **http(s)://.github.io/<project_name>** - наприклад, **http://2gis.github.io/makeup/.** Більше інформації можна знайти у документації.

8.11.4. Gist

Gist – це **git**-репозиторій без підтримки директорій. Зазвичай його використовують для зберігання шматків коду та чернеток; там також можна знайти повноцінні туторіали та статті. Можна сказати, що це така лайт-версія **Github** для ваших нотаток будь-якого характеру, з коментуванням, версіонуванням та можливістю створювати необмежену кількість секретних записів, які будуть доступні для інших користувачів лише за прямим посиланням.

Gist можна використовувати на сторонніх ресурсах. Багато хто використовує його для підсвічування синтаксису шматків коду у статичних блогах або на **Medium**. Для отримання коду скрипта **gist**, який можна вбудувати на сторінку, треба вибрати **Embed** із меню вибору виду **URL** на репозиторій:

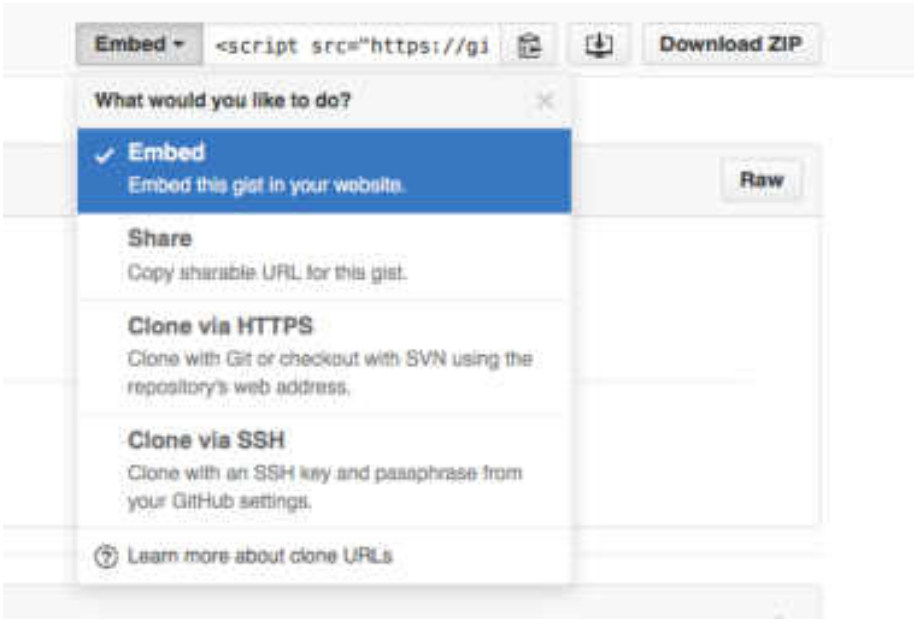


Рис. 8.23. Отримання коду скрипта **gist** на сторінку

8.11.5. Dotfiles

На **http://dotfiles.github.io/** ви можете знайти якісні вибірки налаштувань **Unix**-систем на будь-який смак.

Питання до розділу 8

1. Що ви знаєте про додаткові можливості, що розширюють можливості GitHub?
2. GitHub. Додаткові можливості URL?
3. GitHub. Додаткові можливості гарячих клавіш?
4. GitHub. Додаткові можливості issues та pull request?
5. GitHub. Додаткові можливості Github markdown?
6. GitHub. Додаткові можливості аккаунта?
7. GitHub. Додаткові можливості в роботі з репозиторіями?
8. GitHub. Додаткові можливості в роботі з кодом?
9. GitHub. Додаткові можливості в роботі з командним рядком?
10. GitHub. Додаткові можливості в роботі з User scripts?

ПЕРЕЛІК ПОСИЛАНЬ

1. Skott Chacon, Ben Straub. Pro Git. 2nd Edition : Apress, 2014. URL: https://git-scm.com/book/uk/v2 (Дата звернення 12.12.2022).
2. Что такое контроль версий? URL: https://cutt.us/AeHN8 (Дата звернення 12.12.2022).
3. Вступ – про систему контроля версій. URL: https://cutt.us/LERkK (Дата звернення 12.12.2022).
4. Якоб Стопак. История систем контроля версий. URL: https://cutt.us/76mPT (Дата звернення 12.12.2022).
5. Директория Git и рабочая директория. URL: https://cutt.us/ah2VE (Дата звернення 12.12.2022).
6. Объектная модель Git. URL: https://cutt.us/Sg23V (Дата звернення 12.12.2022).
7. Налаштування Git – Конфігурація Git. URL: https://cutt.us/GomEX (Дата звернення 12.12.2022).
8. Roles in an organization. URL: https://docs.github.com/en/organizations/managing-peoples-access-to-your-organization-with-roles/roles-in-an-organization (Дата звернення 12.12.2022).
9. Repository roles for an organization. URL: https://docs.github.com/en/organizations/managing-access-to-your-organizations-repositories/repository-roles-for-an-organization (Дата звернення 12.12.2022).

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Изучите Git с помощью Bitbucket Cloud. URL: https://cutt.us/UgohW (Дата звернення 12.12.2022).
2. Skott Chacon, Ben Straub. Pro Git. 2nd Edition : Apress, 2014. URL: https://git-scm.com/book/uk/v2 (Дата звернення 12.12.2022).
3. The Git Community Book. URL: https://uleming.github.io/gitbook/index.html (Дата звернення 12.12.2022).
4. Cheat-sheet. URL: https://github.com/tiimgreen/github-cheat-sheet (Дата звернення 12.12.2022).
5. Awesome Github. URL: https://github.com/Kikobeats/awesome-github (Дата звернення 12.12.2022).
6. Zach Holman. Git and GitHub Secrets. URL: https://speakerdeck.com/holman/git-and-github-secrets (Дата звернення 12.12.2022).
7. Zach Holman. More Git and GitHub Secrets. URL: https://vimeo.com/72955426 (Дата звернення 12.12.2022).
8. LearnGitBranching. URL: https://cutt.us/NeFbU (Дата звернення 12.12.2022).

Додаток 1.

Словник термінів Git та GitHub

Апстрим (upstream), коротка назва посилання на майстер-репозиторій.

Віддалений репозиторій - репозиторій, що знаходиться на віддаленому сервері. Це загальний репозиторій, до якого приходять усі зміни і з якого забираються усі оновлення.

Гілка (branch) – це паралельна версія репозиторію. Вона включена в цей репозиторій, але не впливає на головну версію, тим самим дозволяючи вільно працювати в паралельній. Коли ви внесли потрібні зміни, ви можете об'єднати їх з головною версією.

Гіт або Git – система контролю та керування версіями файлів.

Гітхаб або GitHub– веб-сервіс для розміщення репозиторіїв та спільної розробки проектів.

Ішью (Issue), (*ср. р., не схиляється*) запит до організаторів репозиторію, що містить пропозицію щодо покращення, вказівку на помилку, завдання чи питання, пов'язані з репозиторієм, напр. *to open a new issue* – відкрити нове ішью.

Клонування (**clone**) — завантаження репозиторію з віддаленого сервера на локальний комп'ютер у певний каталог для подальшої роботи з цим каталогом як з репозиторієм.

Кодревью (code review), - процес перевірки коду на відповідність певним вимогам, завданням і зовнішньому вигляду.

Коміт (Commit) — фіксація змін або запис змін до репозиторію. **Commit** відбувається на локальній машині.

Контрибьютор (contributor), учасник проекту з відкритим вихідним кодом, що допомагає його розвитку виправленням помилок, написанням коду та документації.

Локальний репозиторій – репозиторій, розташований на локальному комп'ютері розробника у каталозі. Саме в ньому відбувається розробка та фіксація змін, що вирушають на віддалений репозиторій.

Майстер (Master) - головна або основна гілка репозиторію.

Майстер-репозиторій, головний репозиторій, від нього починаються форки.

Мердж (Merge) - злиття змін з будь-якої гілки репозиторію з будь-якою гілкою цього ж репозиторію. Найчастіше злиття змін із гілки репозиторію з основною гілкою репозиторію.

Мердж-реквест (merge request), синонім **пул-реквесту**, використовується в **GitLab**.

Мейнтейнер (**maintainer**) учасник проекту з відкритим вихідним кодом, який приймає рішення щодо його розвитку та спрямовує хід розробки проекту.

Оновитись з апстриму – оновити свою локальну версію форка до останньої версії основного репозиторію, від якого зроблено форк.

Оновитись з ориджину – оновити свою локальну версію репозиторію до останньої віддаленої версії цього репозиторію.

Ориджин (origin), коротка назва посилання на віддалений репозиторій.

Пул (Pull) – отримання останніх змін із віддаленого сервера репозиторію.

Пул-реквест (Pull Request) - запит на злиття форки (див. форк-репозиторій) репозиторію з основним репозиторієм. Пул-реквест може бути прийнятий або відхилений вами як власником репозиторію.

Пуш (Push) — надсилання всіх ненаправлених **commit**-ів на віддалений сервер репозиторію.

Репозиторій Git - каталог файлової системи, в якому знаходяться: файли конфігурації, файли журналів операцій, що виконуються над репозиторієм, індекс розташування файлів та сховище, що містить контрольовані файли.

Сквошити або **склеювати (squash),** об'єднувати кілька **commit**-ів в один для чистішої історії змін: або вручну, або автоматично при мержі пул-реквеста.

Форк (Fork) – копія репозиторію. Його також можна як зовнішню гілку для поточного репозиторію. Копія вашого відкритого репозиторію на **Гітхабі** може бути зроблена будь-яким користувачем, після чого він може надіслати зміни до вашого репозиторію через пул-реквест.

Форк-репозиторій, форк майстер-репозиторію.

Додаток 2.

Перелік найбільш відомих програм, що забезпечують контроль версій.

Локальні.

- **Вільні:**

RCS (Revision Control System) — із особливостей роботи слід відмітити зберігання різниць, завдяки чому забезпечується більш швидкий доступ до стовбура гілок у порівнянні з **SCCS**. Також є поліпшений інтерфейс користувача. Мінуси полягають в повільному доступі до гілок і відсутності підтримки включених/виключених різниць.

PRCS (Project Revision Control System) — робота системи зосереджена на атомних операціях і простоті самої системи; за основу було взято **RCS**, і, як вважається, переписана з ефективнішою моделлю зберігання даних.

SCCS (Source Code Control System) — є частиною **UNIX**; основана на чергованих (**interleaved**) різницях, може створювати версії як довільні множини.

- **Власницькі:**

History Explorer — створена шведською компанією **Exendo**.

MKS Implementer— створена **MKS Inc**. Для IBM і **(System i / iSeries / AS400**).

Розподілені системи

- **Вільні:**

Aegis — стара система, орієнтована на роботу з файлами, слабка підтримка мережі.

ArX — почалась як відгалуження **GNU Arch**, але була пізніше повністю переписаною.

Bazaar — написана на **Python**; децентралізована, і прагне бути швидким і легким у використанні; може без втрат імпортувати **Arch** архіви.

Codeville — написана на **Python**; використовує інноваційний алгоритм злиття.

Darcs — написана на **Haskell**; може відслідковувати міжлаткові залежності.

DCVS — децентралізована та заснована на **CVS**.

Fossil — використовується для розподіленого контролю версій, вікі, і спостереження за помилками.

Git — розроблена Лінусом Торвальдсом для потреб проекту ядра **Linux**; децентралізована, і прагне бути швидкою, гнучкою та надійною.

GNU arch — являє собою розподілену систему контролю версій, яка є частиною проекту **GNU.**

LibreSource — використовується в управлінні конфігурацією.

Mercurial — написана на **Python**; децентралізована і прагне бути швидкою, легкою, портованою і простою у використанні.

Monotone — децентралізована **peer-to-peer** способом.

SVK — написана на **Perl**, побудована поверх **Subversion**, і дозволяє працювати розподілено.

TCL Database-centric Revision control system (tcldbrcs) — побудована на основі баз даних **RCS** на **PostgreSQL**.

- **Власницькі:**

BitKeeper — використовувався в розробці ядра **Linux** (2002 — квітень 2005).

Code Co-op — peer-to-peer система керування версіями (може використовувати e-mail для синхронізації).

Plastic SCM — вільна для навчання та для проектів з відкритим вихідним кодом.

TeamWare — розроблена **Larry McVoy**, створювачем **BitKeeper**.

Клієнт-серверна (централізована) модель

- **Вільні:**

Codendi — веб-платформи для спільної розробки програмного забезпечення.

CVS — спочатку була побудована на **RCS.**

CVSNT — крос-платформний порт **CVS**, який дозволяє використовувати нечутливі до регістру імена файлів.

OpenCVS — сумісна з **CVS**, з упором на безпеку і коректну роботу з вихідним кодом.

Subversion — була випущена в 2000 р.

Vesta — система збирання коду з системою контролю версій файлів і підтримкою розподілених сховищ.

CS-RCS — система збирання коду з системою контролю версій файлів і підтримкою розподілених сховищ. Працює на **Windows** та **UNIX**. Ліцензія на одного користувача є **GPL**.

- **Власницькі:**

AccuRev — входить в інструмент управління з інтегрованою системою трекінгу, базованої на потоках; доступний сервер реплікацій.

Aldon — процесно-орієнтований **Application Lifecycle Management** інструмент.

Alienbrain — частина інструменту управління конфігурацією компанії **Avid Technology.**

AllChange — інструмент управління змінами і конфігурацією компанії **Intasoft.**

AllFusion Harvest Change Manager — інструмент управління змінами і конфігурацією компанії **Computer Associates.**

Autodesk Vault — система контролю версій, спеціально призначена для додатків **Autodesk** таких, як **AutoCAD** і **Autodesk Inventor**.

BrightStar Partners — комплексний контроль версій та управління змінами від **BrightStar BSP Partners** та **BSP Software**.

ClearCase — **SCC** сумісні системи управління конфігураціями **IBM Rational Software.**

Configuration Management Version Control — система контролю версій IBM, більше не доступна.

codeBeamer — платформа керування співпрацею та життєвим циклом програм.

DesignSync — системи управління конфігураціями від **MatrixOne.**

Evolution — управління версіями від **ionForge**; є віддалений доступ, моделі розгалуження, настроюваний робочий процес, інтеграція в розробку, графічні інструменти та інструменти моделювання.

FirePublish — багато платформна; надає функціонал контролю версій і видавничих програм для веб-додатків.

FtpVC — використовує стандартні FTP сервери.

IC Manage — інструмент управління розробкою для апаратного та програмного забезпечення.

MKS Integrity — процес-орієнтоване програмне забезпечення управління циклом життя від **MKS Inc**.

MOG — система контролю версій і підтримки розробки для відеоігор від **MOGware.**

MotioCI — система контролю версій і безперервної інтеграції для інструментів від **Motio**.

PDMWorks — управління даними від **SolidWorks**, з підтримкою **ERP** та інтегрований у **Windows Explorer** інтерфейс.

Perforce — вільна для використання в проектах з відкритими кодами.

Polarion ALM — програма веб-порталу, що використовує **Subversion** для керування версіями артефактів (документів, завдань, запитів на зміни, **Wiki** сторінок тощо) та вихідного коду.

Project Overlord Asset/Project Management Software — розроблений спеціально для комп'ютерної анімації та **VFX** студій.

PureCM — інструмент контролю версій, який підтримує паралельну та розподілену розробку; використовує підхід базований на потоках для розгалуження і злиття.

Polytron Version Control System (PVCS) — спочатку розроблена Дон Кінзером (**Don Kinzer**) з **Polytron**, вперше випущена в 1985 році.

Quma Version Control System.

Randolph архівовано 21 січня 2010 у **Wayback Machine**. — система контролю версій базована на SQL, відстежує зміни структури баз даних та зміни даних з часом, дає повний огляд історії баз даних.

Serena Dimensions — наступник **PVCS**.

SourceAnywhere Hosted — серверне рішення контролю вихідних кодів від **Dynamsoft**.

SourceAnywhere Standalone — рішення контролю вихідних кодів, базоване на **SQL.**

SourceHaven — спочатку була зроблена на основі **Subversion**; є вбудовані можливості роботи з базою даних **Oracle**, та управління за допомогою веб-додатків.

StarTeam — координація та керування процесом доставки програмного забезпечення **Borland**; централізоване управління цифровими активами та діяльністю.

Store — система управління вихідним кодом і версіями від **Cincom** для свого середовища **VisualWorks** для **Smalltalk**.

Surround SCM — крос-платформний інструмент управління вихідних кодів; помітна особливість є можливість процесу відслідковувати, в якому стані були внесені зміни.

Team Coherence — інтегрована система контролю версій та відстеження помилок.

TeamWork — система управління конфігурацією і контролю версій для схем і даних баз даних, написана **dbMaestro**.

Telelogic Synergy — інтегрована, **SCC** сумісна система управління змінами і основана на задачах система управління конфігураціями; власність **IBM**.

TrackWare — система контролю версій і управління конфігурацією від **GlobalWare**.

Vault — інструмент контролю версій від **Source Gear** (перше встановлення може бути використане безкоштовно).

VC/m — керування версіями, управління процесами, впровадження та аудит від **George James Software.**

Version Manager — оснований на даних інструмент контролю версій від **ebiexperts**; може порівнювати **Microsoft Office, XML, PDF** та інші файли.

Visual SourceSafe — інструмент контролю версій від **Microsoft**; орієнтований на малі групи.

Visual Studio Team System — процесно орієнтований клієнт-серверний набір інструментів від **Microsoft** для великих організацій розробників; об'єднує елементи контролю, звітності, автоматизації збирання, тестування та інтеграції з **Microsoft Office.**

Додаток 3.

Довідка Git. Консольні команди.

Загальне

Git – система контролю версій (файлів). Щось подібне до можливості зберігатися в комп'ютерних іграх (у **Git** еквівалент ігрового збереження — **commit**). Важливо: додавання файлів до збереження двоступінчасте: спочатку додаємо файл в індекс (***git add***), потім зберігаємо (***git commit***).

Будь-який файл у директорії існуючого репозиторію може перебувати або перебувати під версійним контролем (відслідковуються і не відслідковуванні).

Файли, що відстежуються, можуть бути в 3-х станах: незмінені, змінені, проіндексовані (готові до **commit**).

Ключ до розуміння

Ключ до розуміння концепції **git** — знання про «три дерева»:

• **Робоча директорія** – файлова система проекту (ті файли, з якими ви працюєте).

• **Індекс** — список файлів і директорій, що відстежуються **git**-ом, проміжне сховище змін (редагування, видалення файлів, що відстежуються).

• **Директорія *.git/*** — всі дані контролю версій цього проекту (вся історія розробки: **commit**-и, гілки, **tags** тощо).

Commit — «збереження» (зберігає набір змін, зроблений робочої директорії з попереднього **commit**-у). **Commit** незмінний, його не можна редагувати.

У всіх **commit**-ів (крім найпершого) є один або більше батьківських **commit**-ів, оскільки **commit**-и зберігають зміни від попередніх станів.

Найпростіший цикл робіт

• Редагування, додавання, видалення файлів (власне робота).

• Індексація/додавання файлів до індексу (вказівка для **git** які зміни потрібно буде **commit**-увати).

• **Commit** (фіксація змін).

• Повернення до кроку 1 або сну.

Вказівники

• ***HEAD*** — вказівник на поточний **commit** або поточну гілку (тобто, у будь-якому випадку, на **commit**). Вказує на батька **commit**-у, який буде створено наступним.

• ***ORIG_HEAD*** — вказівник на **commit**, з якого ви перемістили ***HEAD*** (командою ***git reset*** ..., наприклад).

• **Гілка (*master, develop etc.*)** – вказівник на **commit**. При додаванні **commit**-у вказівник гілки переміщається з батьківського **commit**-у на новий.

• **Теги (tags)** — прості вказівники на **commit**-и. Чи не переміщаються.

Налаштування

Перед початком роботи потрібно виконати деякі налаштування:

git config --global user.name "Your Name" # *вказати ім'я, яким будуть підписані* ***commit****-и*

git config --global user.email "e@w.com" # *вказати електронну пошту, яка буде в описі* **commit-у**

Якщо ви у Windows:

git config --global core.autocrlf true # *включити перетворення закінчених рядків з* ***CRLF*** *на* ***LF.***

Вказування файлів, що не відстежуються.

Файли та директорії, які не потрібно включати до репозиторій, вказуються у файлі ***.gitignore***. Зазвичай це залежні (***node_modules/, bower_components/***), готове складання ***build/*** або ***dist/*** і подібні, створювані при установці або запуску. Кожен файл або директорія вказуються з нового рядка, можливе використання шаблонів.

Консоль

Як використовувати консоль **Bash** у **Windows**, основні команди - **https://github.com/cyberspacedk/BASH-Commands**

Довгий висновок у консолі: Vim

Виклик деяких консольних команд призводить до необхідності дуже довгого виведення (приклад: виведення історії всіх змін у файлі командою ***git log -p***

fileName.txt). При цьому у консолі запускається редактор **Vim**. Він працює у кількох режимах, з яких Вас зацікавлять режим вставки (редагування тексту) та нормальний (командний) режим. Щоб потрапити із **Vim** назад у консоль, потрібно в командному режимі ввести ***:q***. Перехід до командного режиму з іншого: ***Esc***.

Якщо потрібно щось написати, натисніть ***i*** – це перехід до режиму вставки тексту. Якщо потрібно зберегти зміни, перейдіть до командного режиму та наберіть ***:w***.

Vim (деякі команди)

 # Натискання кнопок
***ESC** – перехід у командний режим*
***i** — перехід до режиму редагування тексту*
***ZQ** (затиснутий **Shift,** почергове натискання) - вихід без збереження*
***ZZ** (затиснутий **Shift**, почергове натискання) — зберегти та вийти*
```***bash***
*# Натискання кнопок*
***ESC** – перехід у командний режим*
***i** — перехід до режиму редагування тексту*
***ZQ** (затиснутий **Shift**, почергове натискання) - вихід без збереження*
***ZZ** (затиснутий **Shift**, почергове натискання) — зберегти та вийти*
*# Введення у командному режимі*
***:q!** - вийти без збереження*
***:wq** — зберегти файл та вийти*
***:w filename.txt** — зберегти файл як **filename.txt***

## Консольні команди

### Створити новий репозиторій

***git init** # створити новий проект у поточній директорії*
***git init folder-name** # створити новий проект у вказаній директорії*

### Клонування репозиторію

*# клонувати віддалений репозиторій до однойменної директорії*
***git clone https://github.com/cyberspacedk/Git-commands.git***
```

 # клонувати віддалений репозиторій у директорію «FolderName»

git clone https://github.com/cyberspacedk/Git-commands.git FolderName

 # клонувати репозиторій у поточну директорію

git clone https://github.com:nicothin/web-design.git .

Перегляд змін

git status *# показати стан репозиторію (відслідковуванні, змінені, нові файли та ін.)*

git diff *# порівняти робочу директорію та індекс (не відслідковуванні файли ІГНОРУЮТЬСЯ)*

git diff --color-words *# порівняти робочу директорію та індекс, показати відмінності в словах (не відстежуванні файли ІГНОРУЮТЬСЯ)*

git diff index.html *# порівняти файл з робочої директорії та індекс*

git diff HEAD *# порівняти робочу директорію та* ***commit****, на який вказує* ***HEAD*** *(не відстежуванні файли ІГНОРУЮТЬСЯ)*

git diff --staged *# порівняти індекс та* ***commit*** *з* ***HEAD***

git diff master feature *# подивитися що зроблено у гілці* ***feature*** *у порівнянні з гілкою* ***master***

git diff --name-only master feature *# подивитися що зроблено у гілці* ***feature*** *в порівнянні з гілкою* ***master****, показати тільки імена файлів*

git diff master...feature *# подивитися що зроблено у гілці* ***feature*** *з моменту (****commit****-у) розбіжності з* ***master***

Додавання змін до індексу

git add. *# додати до індексу всі нові, змінені, видалені файли з поточної директорії та її піддиректорій*

git add text.txt *# додати до індексу вказаний файл (було змінено, було видалено або це новий файл)*

git add -i *# запустити інтерактивну оболонку для додавання до індексу тільки вибраних файлів*

git add -p *# показати нові/змінені файли по черзі із зазначенням їх змін та питанням про відстеження/індексування*

Видалення змін з індексу

git reset *# прибрати з індексу всі додані до нього зміни (у робочій директорії всі зміни збережуться), антипод* ***git add***

git reset readme.txt *# прибрати з індексу зміни вказаного файлу (у робочій директорії зміни збережуться)*

Скасування змін

git checkout text.txt *# НЕБЕЗПЕЧНО: скасувати зміни у файлі, повернути стан файлу в індексі*

git reset --hard *# НЕБЕЗПЕЧНО: скасувати зміни; повернути те, що в* ***commit****, на який вказує HEAD (не помічені зміни видалені з індексу та з робочої директорії, файли, що не відстежуються, залишаться на місці)*

git clean -df *# видалити файли та каталоги, що не відслідковуються.*

Commit-и (Commits)

git commit -m "Name of commit" *# зафіксувати в* ***commit*** *проіндексовані зміни (****commit****-ити), додати повідомлення*

git commit -a -m "Name of commit" *# проіндексувати файли, що відстежуються (ТІЛЬКИ відстежуванні, але НЕ нові файли) і* ***commit****-ити, додати повідомлення*

Скасування commit-ів та переміщення з історії

Всі **commit**-и, які вже були відправлені до віддаленого репозиторію, повинні скасовуватися новими ***commit*** (***git revert***), щоб уникнути проблем з історією розробки в інших учасників проекту.

git revert HEAD --no-edit *# створити новий* ***commit****, який скасовує зміни останнього* ***commit*** *без запуску редактора повідомлення*

git revert b9533bb --no-edit *# те ж, але скасовуються зміни, внесені* ***commit*** *із зазначеним хешем (****b9533bb****)*

Усі команди, наведені нижче, можна виконувати ТІЛЬКИ, якщо commit-и ще не були відправлені у віддалений репозиторій.

 # УВАГА! Небезпечні команди - можна втратити не закомічені зміни

git commit --amend -m "Назва" *# «пере-**commit**-ити» зміни останнього **commit**, замінити його новим **commit** з іншим повідомленням (зрушити поточну гілку на один **commit** назад, зберігши робочу директорію та індекс «як є», створити новий **commit** з даними з «відмінюваного» **commit**, але новим повідомленням)*

git reset --hard @~ *# пересунути HEAD (і гілку) на попередній **commit**, робочу директорію та індекс зробити такими, якими вони були в момент попереднього **commit***

git reset --hard 75e2d51 *# пересунути HEAD (і гілку) на **commit** із зазначеним хешем, робочу директорію та індекс зробити такими, якими вони були в момент зазначеного **commit***

git reset --soft @~ *# пересунути HEAD (і гілку) на попередній **commit**, але в робочій директорії та індексі залишити всі зміни*

git reset --soft @~2 *# те ж, але пересунути HEAD (і гілку) на 2 **commit**-и назад*

git reset @~ *# пересунути HEAD (і гілку) на попередній **commit**, робочу директорію залишити як є, індекс зробити таким, яким він був у момент попереднього **commit** (зручніше, ніж git reset --soft @~, якщо індекс потрібно задати заново)*

*# Майже як **git reset --hard**, але безпечніше: не вдасться втратити зміни у робочій директорії*

git reset --keep @~ *# пересунути HEAD (і гілку) на попередній **commit**, скинути індекс, але в робочій директорії залишити зміни, якщо можливо (якщо файл зі змінами між **commit**-ами змінювався, буде видана помилка та перемикання не відбудеться)*

Тимчасово перейти на інший *commit*

git checkout b9533bb *# перейти на **commit** із зазначеним хешем (перемістити HEAD на вказаний **commit**, робочу директорію повернути до стану, на момент цього **commit**)*

git checkout master *# перейти на **commit**, на який вказує **master** (перемістити HEAD на **commit**, на який вказує **master**, робочу директорію повернути до стану на момент цього **commit**)*

Перейти на інший commit і продовжити роботу з нього

Потрібне створення нової гілки, що починається із зазначеного ***commit***.

git checkout -b new-branch 5589877 *# створити гілку* ***new-branch****, що починається з* ***commit*** *с хешем* ***5589877*** *(перемістити HEAD на вказаний* ***commit****, робочу директорію повернути до стану, на момент цього* ***commit****, створити вказівник на цей* ***commit****))*

Відновлення змін

git checkout 5589877 index.html *# відновити в робочій директорії вказаний файл на момент вказаного* ***commit*** *(і додати цю зміну в індекс) (****git reset index.html*** *для видалення з індексу, але збереження змін у файлі)*

Копіювання commit (перенесення commit-ів)

git cherry-pick 5589877 *# скопіювати на активну гілку зміни із зазначеного* **commit***,* ***commit****-ити ці зміни*

git cherry-pick master~2..master *# скопіювати на активну гілку зміни з master (2 останніх* ***commit****-и)*

git cherry-pick -n 5589877 *# скопіювати на активну гілку зміни із зазначеного* **commit***, але НЕ* ***COMMIT****-ИТИ (маю на увазі, що ми самі потім* ***commit****-имо)*

git cherry-pick master..feature *# скопіювати на активну гілку зміни з усіх* ***commit****-ів гілки* ***feature*** *з моменту її розбіжності з* ***master*** *(схоже на злиття гілок, але це копіювання змін, а не злиття),* ***commit****-ити ці зміни; це може викликати конфлікт*

git cherry-pick --abort *# перервати конфліктне перенесення* ***commit****-ів*

git cherry-pick --continue *# продовжити конфліктне перенесення* ***commit****-ів (спрацює тільки після вирішення конфлікту)*

Видалення файлу

git rm text.txt *# видалити незмінений файл, що відстежується, і проіндексувати цю зміну*

git rm -f text.txt *# видалити відстежуваний змінений файл і проіндексувати цю зміну*

git rm -r log/ *# видалити весь вміст відстежуваної директорії* ***log/*** *і проіндексувати цю зміну*

git rm ind* *# видалити всі відстежуванні файли з ім'ям, що починається на «ind» у поточній директорії і проіндексувати цю зміну*

git rm --cached readme.txt *# видалити з відстежуваних індексований файл (ФАЙЛ ЗАЛИШЕТЬСЯ НА МІСЦІ) (часто використовується для ненароком доданих у відстежувані файли)*

Переміщення/перейменування файлів

Для **git** немає перейменування. Перейменування сприймається як видалення старого файлу та створення нового. Факт перейменування можна визначити лише після індексації зміни.

git mv text.txt test_new.txt *# перейменувати файл «**text.txt**» на «**test_new.txt**» і проіндексувати цю зміну*

git mv readme_new.md folder/ *# перемістити файл* ***readme_new.md*** *в директорію* ***folder/*** *(має існувати) і проіндексувати цю зміну*

Історія commit-ів

Вихід із довгого логу виведення: ***q***.

git log master *# показати* ***commit****-и у вказаній гілці*

git log -2 *# показати останні 2* ***commit-****и в активній гілці*

git log -2 --stat *# показати останні 2* ***commit****-и та статистику внесених ними змін*

git log -p -22 *# показати останні 22* ***commit*** *та внесену ними різницю на рівні рядків*

git log --graph -10 *# показати останні 10* ***commit*** *з ASCII-поданням розгалуження*

git log --since=2.weeks *# показати* ***commit****-и за останні 2 тижні*

git log --after '2018-06-30' *# показати* ***commit-****и, зроблені після вказаної дати*

git log index.html *# показати історію змін файлу index.html (тільки* ***commit****-и)*

git log -5 index.html *# показати історію змін файлу index.html, останні 5* **commit** *(тільки* ***commit****-и)*

git log -p index.html *# показати історію змін файлу index.html (****commit****-и та зміни)*

git log -G'myFunction' -p *# показати всі* ***commit****-и, в яких змінювалися рядки з* ***myFunction*** *(у лапках регулярний вираз)*

git log -L '/<head>/','/<\/head>/':index.html *# показати зміни від вказаного до вказаного регулярних виразів у вказаному файлі*

git log --grep fix *# показати* ***commit****-и, в описі яких є буквосполучення fix (реєстр-залежно, тільки* ***commit****-и поточної гілки)*

git log --grep fix -i *# показати* ***commit****-и, в описі яких є буквосполучення fix (реєстр-незалежно, тільки* ***commit****-и поточної гілки)*

git log --grep 'fix(ing|me)' -P *# показати* ***commit****-и, в описі яких є збіги для регулярного вираження (тільки* ***commit****-и поточної гілки)*

git log --pretty=format:"%h - %an, %ar : %s" -4 *# показати останні 4* ***commit****-и з форматуванням даних, що виводяться*

git log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short *# мій формат виведення, що висить на аліасі оболонки*

git log master..branch_99 *# показати* ***commit****-*и *з гілки* ***branch_99****, які не влиті в* ***master***

git log branch_99..master *# показати* ***commit****-и з гілки* ***master****, які не влиті в* ***branch_99***

git log master...branch_99 --boundary -- graph *# показати* ***commit****-и із зазначених гілок, починаючи з їхнього розходження (****commit*** *розбіжності буде показаний)*

git show 60d6582 *# показати зміни з* ***commit*** *із зазначеним хешем*

git show HEAD~ *# показати дані про попередній коментар в активній гілці*

git show @~ *# аналогічно попередньому*

git show HEAD~3 *# показати дані про* ***commit****, який був 3* ***commit****-и тому*

git show my_branch~2 *# показати дані про коментарі, який був 2* ***commit****-и тому у вказаній гілці*

git show @~:index.html *# показати контент вказаного файлу на момент попереднього (від HEAD* ***commit****)*

git show :/"підвал" *# показати найновіший* ***commit****, в описі якого є вказане слово (з будь-якої гілки)*

Хто написав рядок

git blame README.md --date=short -L 5,8 *# показати рядки 5-8 вказаного файлу та* ***commit****-и, в яких рядки були додані*

Історія змін вказівників (гілок, HEAD)

git reflog -20 *# показати останні 20 змін положення вказівника HEAD*

git reflog --format='%C(auto)%h%<|(20)%gd%C(blue)%cr%C(reset)%gs(%s)' -20 *# те ж, але із зазначенням давності дій*

Гілки

git branch *# показати список гілок*

git branch -v *# показати список гілок та останній* ***commit*** *у кожній*

git branch new_branch *# створити нову гілку із вказаним ім'ям на поточному* ***commit***

git branch new_branch 5589877 *# створити нову гілку з вказаним ім'ям на вказаному коментарі*

git branch -f master 5589877 *# перемістити гілку* ***master*** *на вказаний* ***commit***

git branch -f master master~2 *# перемістити гілку* ***master*** *на 2* ***commit****-и назад*

git checkout new_branch *# перейти у вказану гілку*

git checkout -b new_branch *# створити нову гілку з вказаним ім'ям і перейти до неї*

git checkout -B master 5589877 *# перемістити гілку з вказаним ім'ям на вказаний* ***commit*** *і перейти до неї*

git merge hotfix *# влити у гілку, в якій знаходимося, дані з гілки* ***hotfix***

git merge hotfix -m "Гаряча правка" *# влити у гілку, в якій знаходимося, дані з гілки* ***hotfix*** *(вказано повідомлення* ***commit*** *злиття)*

git merge hotfix --log *# влити у гілку, в якій знаходимося, дані з гілки* ***hotfix****, показати редактор опису* ***commit****, додати в нього повідомлення* ***commit****-ів, що вливаються*

git merge hotfix --no-ff *# влити у гілку, в якій знаходимося, дані з гілки* ***hotfix****, заборонити простий зсув вказівника, зміни з* ***hotfix*** *«залишаться» в ній, а в активній гілці з'явиться тільки* ***commit*** *злиття*

git branch -d hotfix *# видалити гілку* ***hotfix*** *(використовується, якщо її зміни вже влиті в головну гілку)*

git branch --merged *# показати гілки, вже злиті з активної*

git branch --no-merged *# показати гілки, не злиті з активної*

git branch -a *# показати всі гілки (в т.ч. на віддалених репозиторіях)*

git branch -m old_branch_name new_branch_name *# локально перейменувати гілку* ***old_branch_name в new_branch_name***

git branch -m new_branch_name *# перейменувати локально ПОТОЧНУ гілку в* ***new_branch_name***

git push origin :old_branch_name new_branch_name *# застосувати перейменування у віддаленому репозиторії*

git branch --unset-upstream *# завершити процес перейменування*

Теги

git tag v1.0.0 *# створити* ***tag*** *із вказаним ім'ям на* ***commit****, на який вказує HEAD*

git tag -a -m 'У продакшені!' v1.0.1 master *# створити* ***tag*** *з описом на тому* ***commit****, на який дивиться гілка master*

git tag -d v1.0.0 *# видалити* ***tag*** *із зазначеним ім'ям(ами)*

git tag -n *# показати всі* ***tags*** *та по 1 рядку повідомлення* ***commit****-ів, на які вони вказують*

git tag -n -l 'v1.*' *# показати всі* ***tags****, що починаються з* ***'v1.*'***

Тимчасове збереження змін без commit

git stash *# тимчасово зберегти незакомічені зміни та прибрати їх з робочої директорії*

git stash pop *# повернути збережені командою* ***git stash*** *зміни до робочої директорії*

Віддалені репозиторії

Є два поширені способи прив'язати віддалений репозиторій до локального: **HTTPS і SSH**. Якщо **SSH** у вас не налаштований (або ви не знаєте що це), прив'язуйте віддалений репозиторій за **HTTPS** (адреса репозиторію, що прив'язується, повинен починатися з **https://**).

git remote -v *# показати список віддалених репозиторіїв, пов'язаних з локальним*

git branch -r *# показати видалені гілки*

git branch -a *# показати всі гілки (локальні та віддалені)*

git remote remove origin *# прибрати прив'язку віддаленого репозиторію з скор. ім'ям* ***origin***

git remote add origin https://github.com:nicothin/test.git *# додати віддалений репозиторій (з скор. ім'ям* ***origin)*** *із зазначеним* ***URL***

git remote rm origin *# видалити прив'язку віддаленого репозиторію*

git remote show origin *# отримати дані про віддалену репозиторію зі скороченим ім'ям* ***origin****.*

git fetch origin *# скачати всі гілки з віддаленого репозиторію (з скор. ім'ям* ***origin****), але не зливати зі своїми гілками*

git fetch origin master *# те ж, але скачується тільки вказана гілка*

git checkout --track origin/github_branch *# створити локальну гілку* ***github_branch*** *(дані взяти з віддаленого репозиторію з скор. ім'ям* ***origin****, гілка* ***github_branch****) і переключитися на неї*

git push origin master *# відправити у віддалений репозиторій (з скор. ім'ям* ***origin****) дані своєї гілки* ***master***

git pull origin *# влити зміни з віддаленого репозиторію (всі гілки)*

git pull origin master *# влити зміни з віддаленого репозиторію (тільки вказана гілка)*

Конфлікт злиття

Передбачається ситуація: є гілка ***master*** і є гілка ***feature***. В обох гілках є **commit-и**, зроблені після розбіжності гілок. У гілку ***master*** намагаємось влити гілку ***feature (git merge feature***), отримуємо конфлікт, тому що в обох гілках є зміни одного і того ж рядка у файлі ***index.html***.

У разі конфлікту, репозиторій перебуває у стані перерваного злиття. Потрібно залишити у конфліктуючих місцях файлів лише потрібний код, проіндексувати зміни та **commit**-увати.

git merge feature *# влити в активну гілку зміни з гілки* ***feature***

git merge-base master feature *# показати хеш останнього загального* ***commit*** *для двох зазначених гілок*

git checkout --ours index.html *# залишити в конфліктному файлі (**index.html**) стан гілки, В ЯКУ ми вливаємо (у прикладі - з гілки **master**)*

git checkout --theirs index.html *# залишити в конфліктному файлі (**index.html**) стан гілки, З ЯКОЇ ми вливаємо (у прикладі - з гілки **feature**)*

git checkout --merge index.html *# показати в конфліктному файлі (**index.html**) порівняння вмісту гілок, що зливаються (для ручного редагування)*

git checkout --conflict=diff3 --merge index.html *# показати в конфліктному файлі (**index.html**) порівняння вмісту гілок, що зливаються, плюс те, що було в місці конфлікту в **commit**, на якому розійшлися гілки, що зливаються.*

git reset --hard *# припинити це перерване злиття, повернути робочу директорію та індекс як було в момент **commit**, на який вказує **HEAD***

git reset --merge *# припинити це перерване злиття, але залишити зміни, не **commit** до злиття (для випадку, коли злиття робиться не на чистому статусі)*

git reset --abort *# те ж, що і рядком вище*

«Перенесення» гілки

Можна «перемістити» відгалуження будь-якої гілки від основної на довільний **commit**. Це потрібно для того, щоб в «гілці, що переноситься» з'явилися які-небудь зміни, внесені в основній гілці (вже після відгалуження переноситься).

Не можна «переносити» гілку, якщо вже відправлено на віддалений репозиторій.

git rebase master *# перенести всі **commit**-и (створити їх копії) активної гілки так, ніби активна гілка відповіла від **master** на нинішній вершині **master** (часто викликає конфлікти)*

git rebase --onto master feature *# перенести **commit**-и активної гілки на **master**, починаючи з того місця, в якому активна гілка відокремилася від гілки **feature***

git rebase --abort *# перервати конфліктний **rebase**, повернути робочу директорію та індекс до стану до початку **rebase***

git rebase --continue *# продовжити конфліктний **rebase** (спрацює тільки після вирішення конфлікту та індексації такого вирішення)*

Як скасувати rebase

git reflog feature -2 *# дивимося* ***лог*** *переміщень гілки, якою робили* ***rebase*** *(у цьому прикладі —* ***feature)****, бачимо останній* ***commit*** *ПЕРЕД* ***rebase****, на нього і потрібно перенести вказівник гілки*

git reset --hard feature@{1} *# перемістити вказівник гілки* ***feature*** *на один* ***commit*** *назад, оновити робочу директорію та індекс*

Різне

git archive -o ./project.zip HEAD *# створити архів з файловою структурою проекту за вказаним шляхом (стан репозиторію, що відповідає вказівнику HEAD)*

Початок роботи

Створення нового репозиторію, перший **commit**, прив'язка віддаленого репозиторію з ***gthub.com***, відправка змін до віддаленого репозиторію.

 # вказано послідовність дій:

 # створено директорію проекту, ми в ній

git init *# створюємо репозиторій у цій директорії*

touch readme.md *# створюємо файл readme.md*

git add readme.md *# додаємо файл в індекс*

git commit -m "Старт" *# створюємо* ***commit***

git remote add origin https://github.com:nicothin/test.git *# додаємо попередньо створений порожній віддалений репозиторій*

git push -u origin master *# відправляємо дані з локального репозиторію у віддалений (у гілку* ***master****)*

«Внесення змін» до commit

Тільки якщо ***commit*** ще не був відправлений до віддалених репозиторій.

 # вказано послідовність дій:

subl inc/header.html *# редагуємо та зберігаємо розмітку «шапки»*

git add inc/header.html *# індексуємо змінений файл*

git commit -m *"Прибрав телефон із шапки" # робимо* ***commit***

 # УВАГА: ***commit*** *поки не був відправлений у віддалений репозиторій*

 # усвідомлюємо, що треба було ще щось зробити в цьому коментарі.

subl inc/header.html *# вносимо зміни*

git add inc/header.html *# індексуємо змінений файл (можна git add .)*

git commit --amend -m "«Шапка»: виконано завдання №34" *# наново робимо* ***commit***

Робота з гілками

Є ***master*** (публічна версія сайту), виконуємо масштабне завдання (переверстати «шапку»), але в процесі роботи виникає необхідність підправити критичний баг (неправильно вказаний контакт у «підвалі»).

 # вказано послідовність дій:

git checkout -b new-page-header *# створимо нову гілку для завдання зміни «шапки» і перейдемо до неї*

subl inc/header.html *# редагуємо розмітку «шапки»*

git commit -a -m "Нова шапка: зміна логотипу" *# робимо* ***commit*** *(робота ще не завершена)*

 # Тут з'ясовується, що є баг із контактом у «підвалі»

git checkout master *# повертаємось до гілки* ***master***

subl inc/footer.html *# усуваємо баг і зберігаємо розмітку «підвалу»*

git commit -a -m "Виправлення контакту у підвалі" *# робимо* ***commit***

git push *# відправляємо* ***commit*** *зі швидкою критичною зміною в master у віддаленому репозиторії*

git checkout new-page-header *# перемикаємося назад у гілку* ***new-page-header*** *для продовження робіт над «шапкою»*

subl inc/header.html *# редагуємо та зберігаємо розмітку «шапки»*

git commit -a -m "Нова шапка: зміна навігації" *# робимо* ***commit*** *(робота над «шапкою» завершена)*

git checkout master *# перемикаємось у гілку* ***master***

git merge new-page-header *# вливаємо в* ***master*** *зміни з гілки* ***new-page-header***

git branch -d new-page-header *# видаляємо гілку* ***new_page_header***

Робота з гілками, злиття та відкат до стану до злиття

Була гілка ***fix***, у якій виправляли баг. Виправили, влили ***fix*** у ***master***. Але тут з'ясувалося, що це виправлення ламає якусь функціональність, Потрібно відкотити

master до стану без злиття (наявність бага менш критично, ніж псування функціональності).

*# перебуваємо у гілці **fix**, баг уже «виправлений»*

git checkout master *# переключаємось на **master***

git merge fix *# вливаємо зміни з **fix** до **master***

бачимо проблему: частина функціональності зламалася

git checkout fix *# перемикаємось на **fix** (поки ми в **master, git** не дасть її рухати)*

git branch -f master ORIG_HEAD *# пересуваємо гілку **master** на **commit**, вказаний у **ORIG_HEAD** (той, на який вказувала **master** до вливання **fix**)*

Робота з гілками, конфлікт злиття

Є гілка ***master*** (публічна версія сайту), у двох паралельних гілках (***branch-1*** і ***branch-2***) було відредаговано те саме місце одного і того ж файлу, першу гілку (***branch-1***) влили в ***master***, спроба влити другу викликає конфлікт.

 # вказано послідовність дій:

git checkout master *# перемикаємося на гілку **master***

git checkout -b branch-1 *# створюємо гілку **branch-1**, засновану на гілці **master***

subl. *# редагуємо та зберігаємо файли*

git commit -a -m "Правка 1" *# **commit**-имо*

git checkout master *# повертаємось до гілки **master***

git checkout -b branch-2 *# створюємо гілку **branch-2**, засновану на гілці **master***

subl. *# редагуємо та зберігаємо файли*

git commit -a -m "Правка 2" *# **commit**-имо*

git checkout master *# повертаємось до гілки **master***

git merge branch-1 *# вливаємо зміни з гілки **branch-1** у поточну гілку (**master**), удача (автозлиття)*

git merge branch-2 *# вливаємо зміни з гілки **branch-2** в поточну гілку (**master**), КОНФЛІКТ автозлиття*

*# **Automatic merge failed; fix conflicts and then commit the result.***

subl. *# вибираємо у конфліктних файлах ті ділянки, які потрібно залишити, зберігаємо*

git commit -a -m "Усунення конфлікту" *# **commit**-имо результат усунення конфлікту*

Синхронізація репозиторію-fork з майстер-репозиторієм

Є якийсь репозиторій на **github.com**, ми зробили **fork**, додані якісь зміни. Оригінальний (майстер) репозиторій був якось оновлений. Завдання: стягнути з майстер-репозиторію зміни (які там внесено вже після того, як ми його форкнули).

 # вказано послідовність дій:

git remote add upstream https://github.com:address.git *# додаємо віддалений репозиторій: скор. ім'я -* ***upstream, URL*** *майстер-репозиторію*

git fetch upstream *# стягуємо всі гілки майстер-репозиторію, але поки не зливаємо зі своїми*

git checkout master *# перемикаємось на гілку* ***master*** *свого репозиторію*

git merge upstream/master *# вливаємо стягнуту гілку* ***master*** *віддаленого репозиторію* ***upstream*** *у свою гілку* ***master***

Помилка в роботі: commit-ити в майстер, але зрозуміли, що треба було commit-ити в нову гілку

ВАЖЛИВО: це спрацює лише якщо ***commit*** ще не відправлений у віддалений репозиторій.

 # вказано послідовність дій:

 # Зробили зміни, проіндексували їх, ***commit****-или в* ***master****, але ЩЕ НЕ ВІДПРАВИЛИ (не робили* ***git push****)*

git checkout -b new-branch *# створюємо нову гілку з* ***master***

git checkout master *# переключаємось на* ***master***

git reset HEAD~ --hard *# зсуваємо вказівник (гілку)* ***master*** *на 1* ***commit*** *тому*

git checkout new-branch *# переключаємось назад на нову гілку для продовження роботи*

Потрібно повернути вміст файлу до стану, що був у якомусь commit-і (відомий хеш commit)

 # вказано послідовність дій:

git checkout f26ed88 -- index.html *# відновити в робочій директорії стан зазначеного файлу на момент вказаного* ***commit****, додати цю зміну до індексу*

git commit -am "Navigation fixs" *# зробити* ***commit***

При будь-якій дії з github (або іншим віддаленим сервісом) запитується логін та пароль

Йдеться саме про запит пари логін + пароль, а не ключової фрази. Відбувається це тому, що **git** за замовчуванням не збереже пароль для доступу до репозиторію **HTTPS**.

Просте рішення: вказати **git** кешувати ваш пароль.

.gitattributes
**** text=auto***

****.html diff=html***
****.css diff=css***
****.scss diff=css***