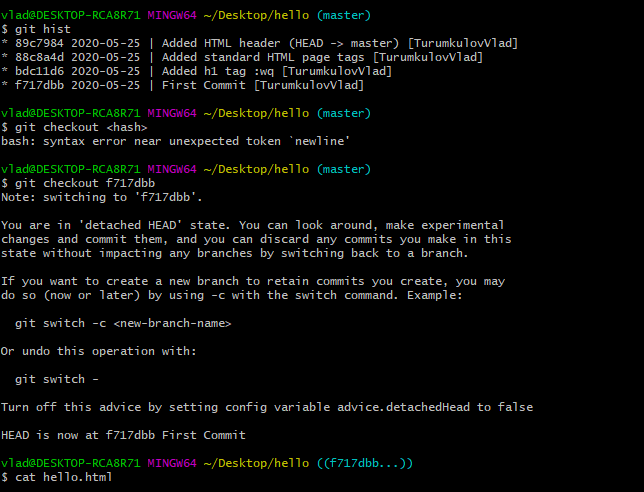
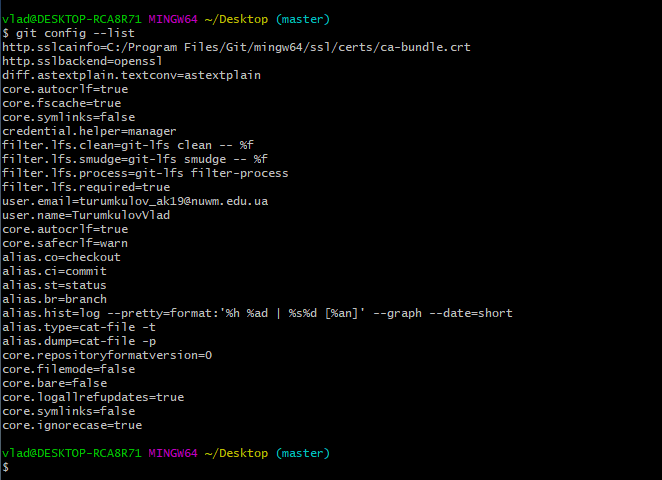
**Звіт до лабораторної роботи №4-5**

**11. Зберігання аліасу**



**12. Повернення до останньої версії**



**13. Створення тегів версій**

Створіть тег першої версії

git tag v1

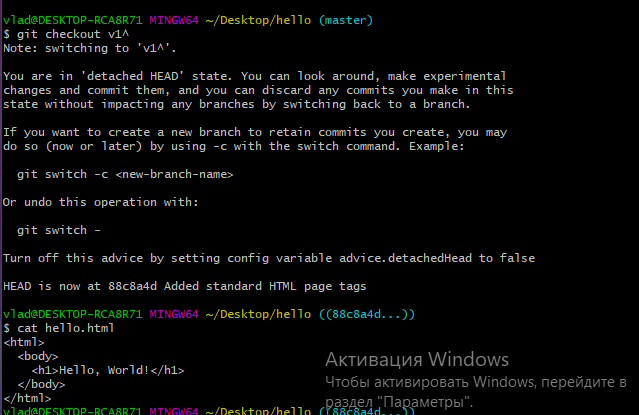
Тепер поточна версія сторінки називається v1.



Теги для попередніх версій

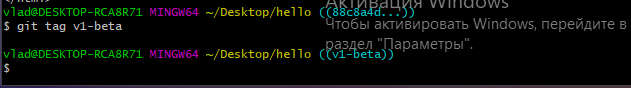
git checkout v1^

cat hello.html



Це версія з тегами <html> та <body>, але поки що без <head>. Давайте зробимо її версією v1-beta.

git tag v1-beta

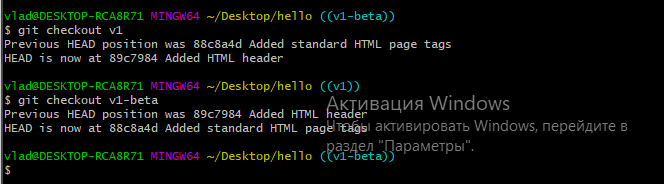


Перемикання за ім'ям тегу

Тепер спробуйте по перемикатися між двома зазначеними версіями.

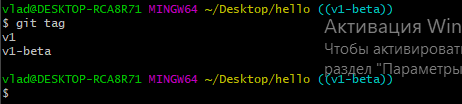
git checkout v1

git checkout v1-beta



Перегляд тегів за допомогою команди tag

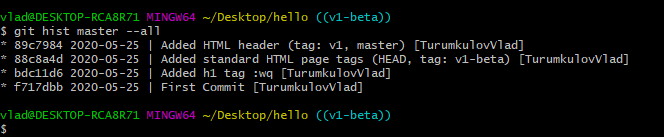
Ви можете побачити, які теги доступні, використовуючи команду git tag.



Перегляд тегів у логах

Ви також можете подивитися теги у лозі.

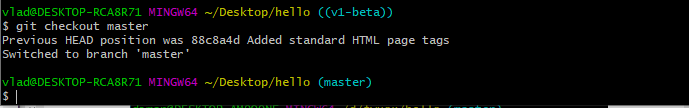
git hist master –all



**14. Скасування локальних змін (до індексації)**

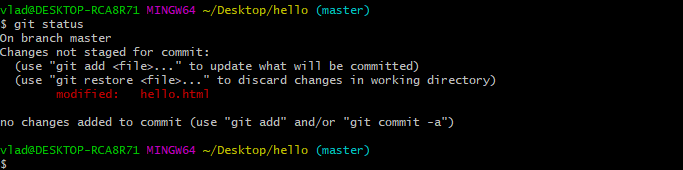
Перейдіть на гілку Master

Переконайтеся, що ви перебуваєте на останньому коміті гілки master, перш ніж продовжити роботу



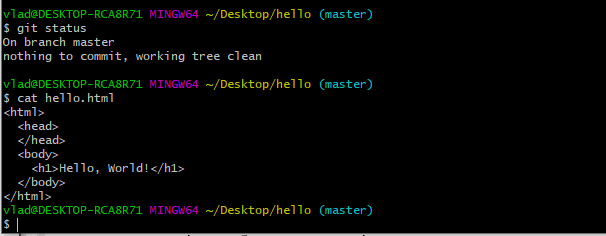
Перевірте стан

Спочатку перевірте стан робочого каталогу.



Скасування змін в робочому каталозі

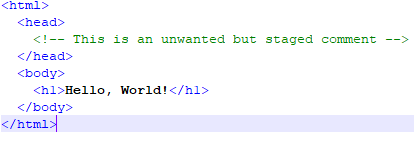
Використовуйте команду checkout для перемикання в версію файлу hello.html у репозиторію.

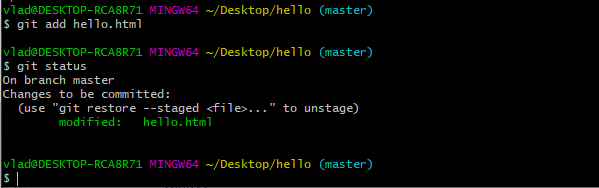


**15. Скасування проіндексованих змін (перед комітом)**

Внесіть зміни у файл і проіндексуйте їх

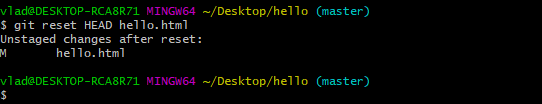
Внесіть зміни у файл hello.html у вигляді небажаного коментаря



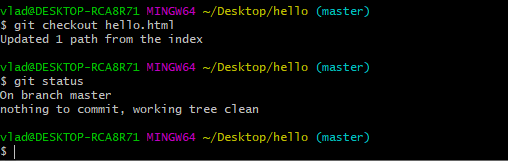


Виконайте сброс буферної зони

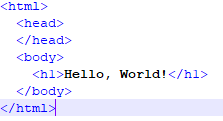
На щастя, вивід стану показує нам саме те, що ми повинні зробити для скасування індексації змін.



Перейдіть на версію коміту



Результат:

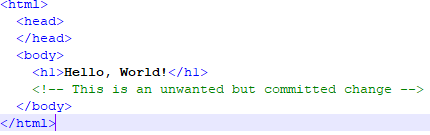


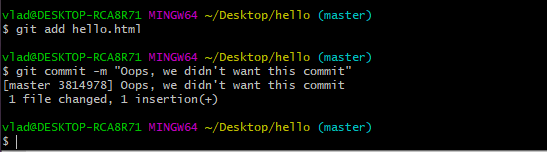
**16. Скасування комітів**

Скасування комітів

Іноді ви розумієте, що нові коміти є невірними, і хочете їх скасувати. Є кілька способів вирішення цього питання, тут ми будемо використовувати найбезпечніший.

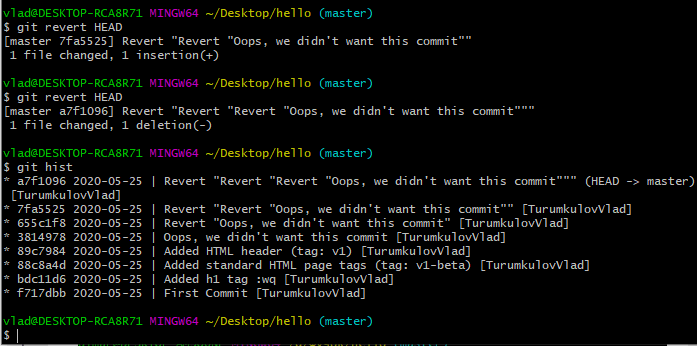
Ми скасуємо коміт шляхом створення нового коміту, що скасовує небажані зміни.





Зробіть коміт з новими змінами, скасовуючими попередні

Щоб скасувати коміт, нам необхідно зробити коміт, який видаляє зміни, збережені небажаним комітом.



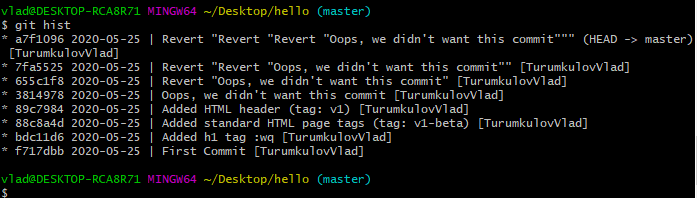
**17. Видалення комітів з гілки (revert)**

Команда reset

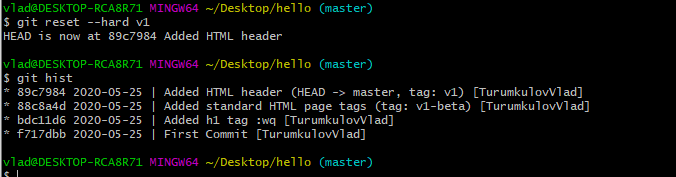
Ми вже бачили команду reset і використовували її для узгодження буферної зони і вибраного коміту (ми використовували коміт HEAD у нашому попередньому уроці).

При отриманні посилання на коміт (тобто хеш, гілка або ім'я тега), команда reset…

1. перепише поточну гілку, щоб вона вказувала на потрібний коміт
2. опціонально скине буферну зону для відповідності із зазначеним комітом
3. опціонально скине робочий каталог для відповідності із зазначеним комітом

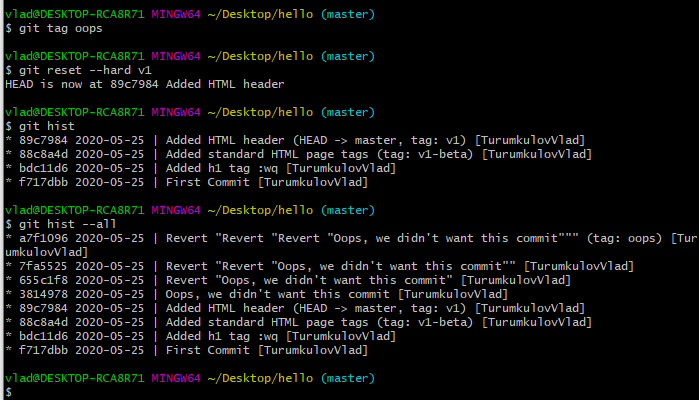


Скидання коміту до попереднього стану Oops



Нічого ніколи не губиться

Що ж трапляється з помилковими комітами? Виявляється, що коміти все ще знаходяться в репозиторію. Насправді, ми все ще можемо на них посилатися. Пам'ятаєте, на початку цього уроку ми створили для скасованого коміту тег «oops». Давайте подивимося на всі коміти.



Небезпека скидання

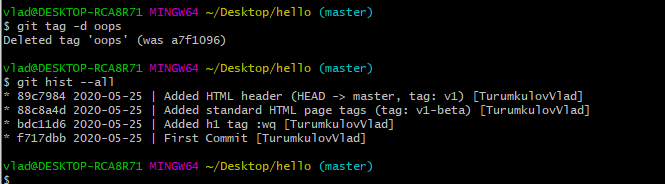
Скидання в локальних гілках, як правило, безпечний. Наслідки будь-який «аварії», як правило, можна відновити простим скиданням за допомогою потрібного коміту.

Однак, якщо гілку «розшарено» на віддалені репозиторії, скидання може збити з пантелику інших користувачів гілки.

**18. Видалення тегу oops**

Видалення тегу oops

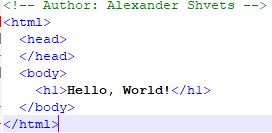
Тег oops свою функцію виконав. Давайте видалимо його і коміти, на які він посилався, сміттязбирачем.

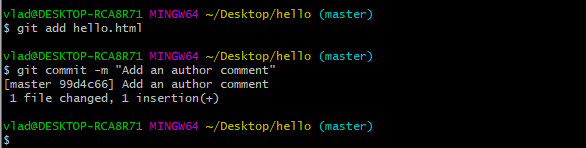


**19. Внесення змін до комітів**

Змініть сторінку, а потім зробіть коміт

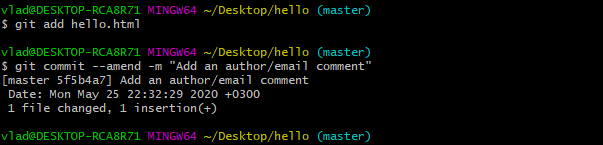
Додайте в сторінку коментар автора

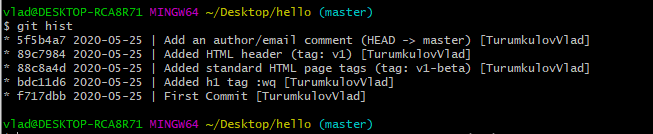




Змініть попередній коміт

Ми дійсно не хочемо створювати окремий коміт тільки заради електронної пошти. Давайте змінимо попередній коміт, включивши в нього адресу електронної пошти.

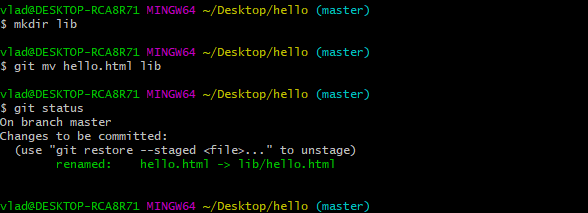




**20. Переміщення файлів**

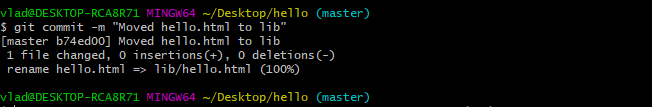
Перемістіть файл hello.html в каталог lib

Зараз ми збираємося змінити структуру нашого репозиторію. Давайте перенесемо сторінку в каталог lib.



Коміт в новий каталог

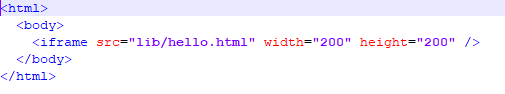
Давайте зробимо коміт цього переміщення.

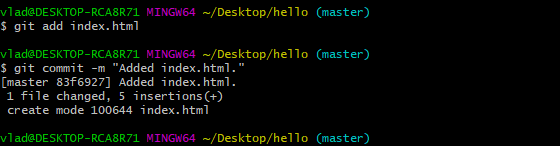


**21. Детальніше про структуру**

Додавання index.html

Давайте додамо файл index.html до нашого репозиторію. Наступний файл відмінно підійде для цієї мети.





**22. Git всередині: Каталог .git**

Каталог .git

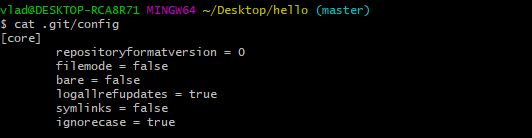
Настав час провести невелике дослідження. Для початку, з кореневого каталогу вашого проекту…



База даних об'єктів



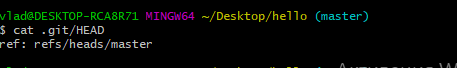
Config File



Гілки і теги



Файл HEAD

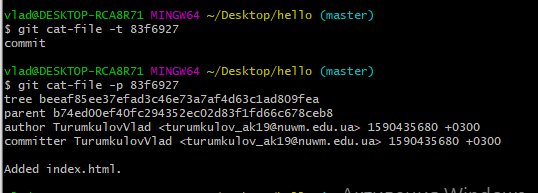


**23. Git всередині: Робота безпосередньо з об'єктами git**

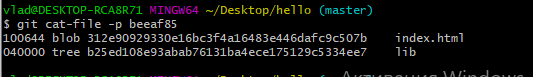
Пошук останнього коміту



Вивід останнього коміту



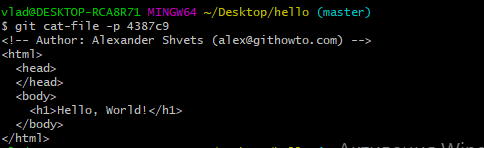
Пошук дерева



Вивід каталогу lib



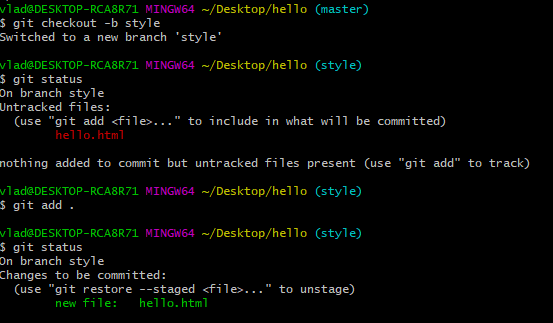
Вивід файлу hello.html



**24. Створення гілки**

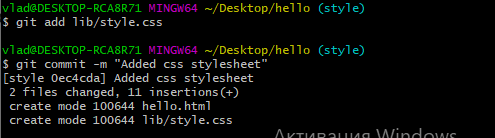
Створіть гілку

Давайте назвемо нашу нову гілку «style».



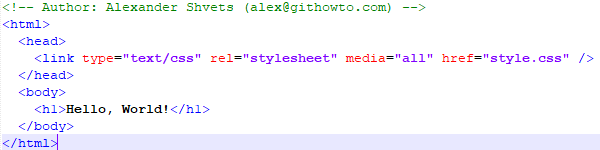
Додайте файл стилів style.css



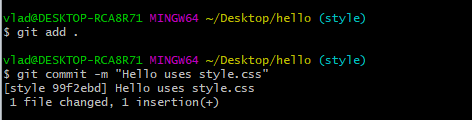


Змініть основну сторінку

Оновіть файл hello.html, щоб використовувати стилі style.css.

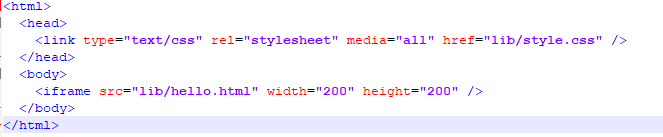


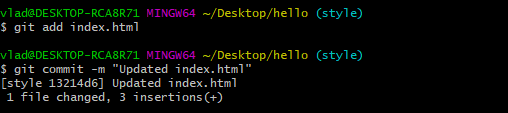




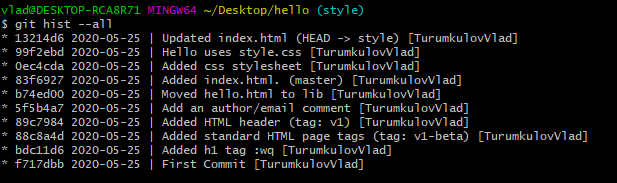
Змініть index.html

Оновіть файл index.html, щоб він теж використовував style.css





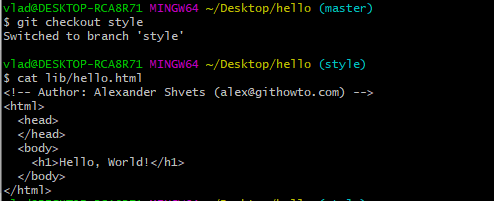
**25. Навігація по гілках**



Перемикання на гілку "master".



Повернемося до гілки «style».

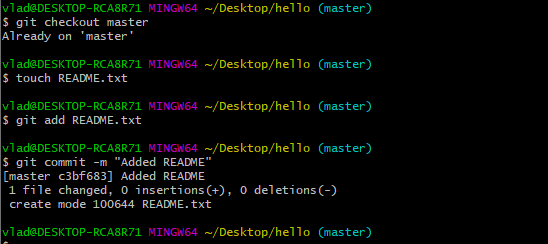


**26. Зміни в гілці master**

Цілі

Навчитися працювати з декількома гілками з різними (і, можливо, конфліктуючими) змінами.

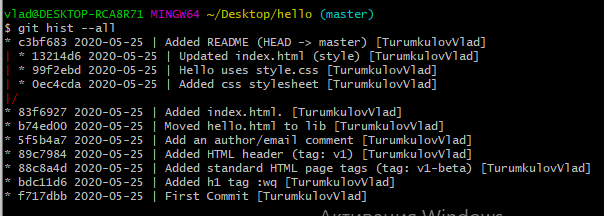
Поки ви міняли гілку «style», хтось вирішив оновити гілку master. Вони додали README.



**27. Перегляд розбіжних гілок**

Перегляньте поточні гілки

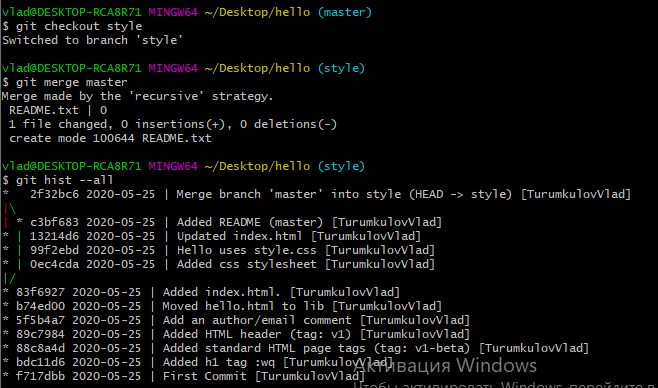
Тепер у нас в репозиторію є дві відмінні гілки. Використовуйте наступну лог-команду для перегляду гілок і їх відмінностей.



**28. Злиття**

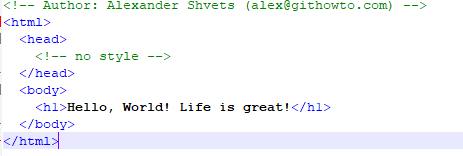
Злиття гілок

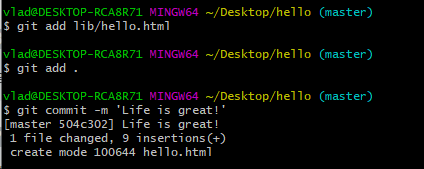
Злиття переносить зміни з двох гілок в одну. Давайте повернемося до гілки style і зіл'ємо master із style.



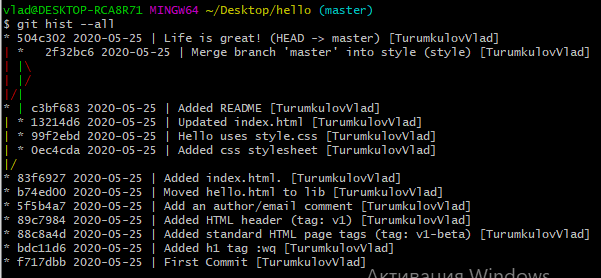
**29. Створення конфлікту**

Поверніться у master і створіть конфлікт

Поверніться у гілку master і внесіть такі зміни: 



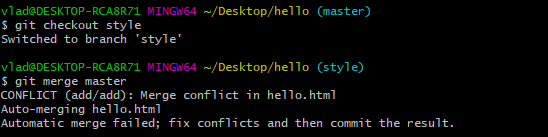
Перегляд гілок



**30. Вирішення конфліктів**

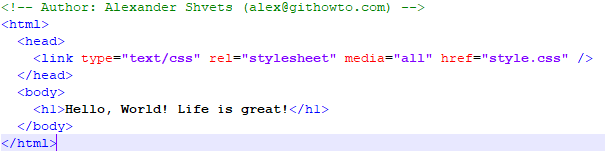
Злиття master з гілкою style

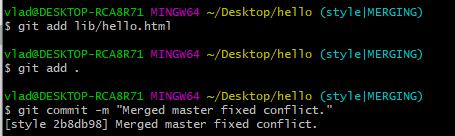
Тепер повернемося до гілки style і спробуємо об'єднати її з новою гілкою master.



Рішення конфлікту

Вам необхідно вручну вирішити конфлікт. Внесіть зміни в lib/hello.html для досягнення наступного результату.





**31. Перебазування як альтернатива злиттю**

Цілі

Дізнатися відмінності між перебазуванням і злиттям.

Обговорення

Давайте розглянемо відмінності між злиттям і перебазуванням. Для того, щоб це зробити, нам потрібно повернутися в репозиторій в момент до першого злиття, а потім повторити ті ж дії, але з використанням перебазування замість злиття.

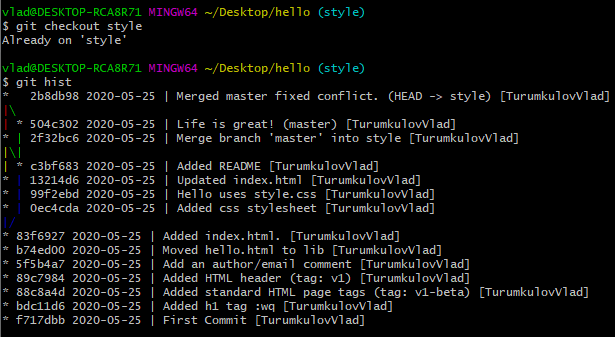
Ми будемо використовувати команду reset для повернення гілок до попереднього стану.

**32. Скидання гілки style**

Скидання гілки style

Давайте повернемося в часі у гілці style до точки перед тим, як ми злили її з гілкою master. Ми можемо скинути гілку до будь-якого коміту. По суті, це зміна покажчика гілки на будь-яку точку дерева комітів.

У цьому випадку ми хочемо повернутися в гілці style в точку перед злиттям з master. Нам необхідно знайти останній коміт перед злиттям.

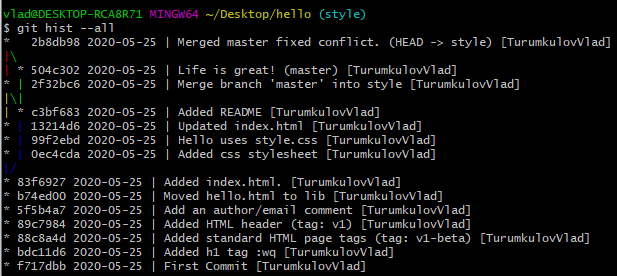


Це трохи важко читати, але, дивлячись на дані, ми бачимо, що коміт «Updated index.html» був останнім у гілці style перед злиттям. Давайте скинемо гілку style до цього коміту.



Перевірте гілку.

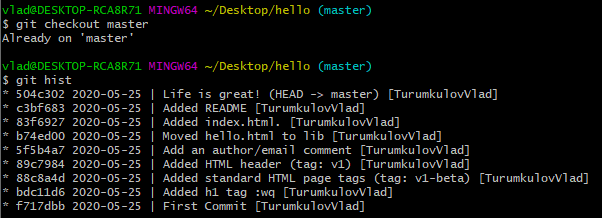
Пошукайте лог гілки style. У нас в історії більше немає комітів злиття.



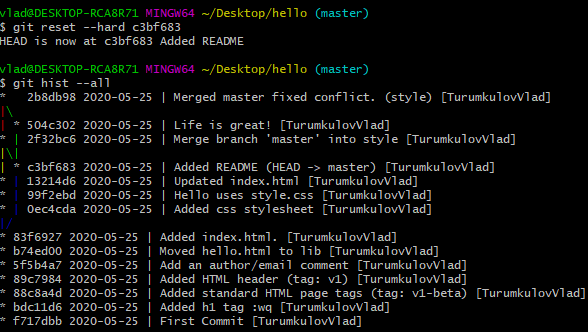
**33. Скидання гілки master**

Скидання гілки master

Додавши інтерактивний режим в гілку master, ми внесли зміни, конфліктуючі зі змінами в гілці style. Давайте повернемося в гілці master в точку перед внесенням конфліктуючих змін. Це дозволяє нам продемонструвати роботу команди rebase, не турбуючись про конфлікти.



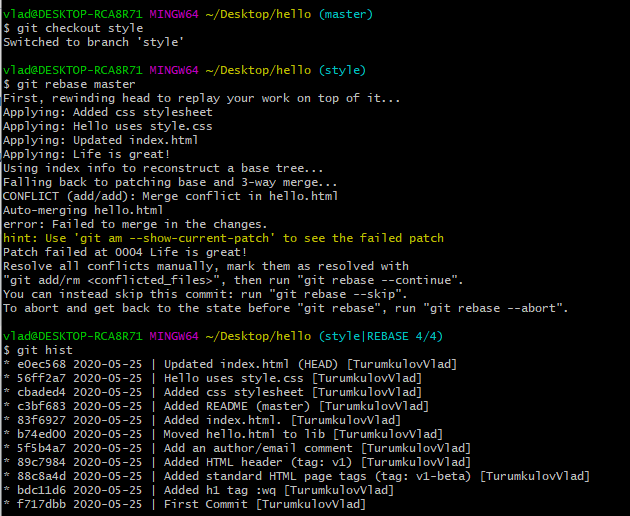
Коміт «Added README» йде безпосередньо перед комітом конфліктуючого інтерактивного режиму. Ми скинемо гілку master до коміту «Added README».



**34. Перебазування**

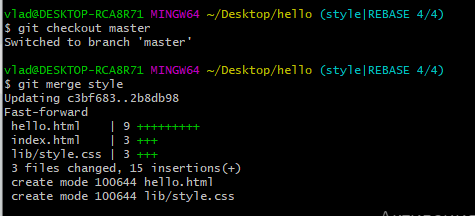
Отже, ми повернулися в точку до першого злиття і хочемо перенести зміни із master у нашу гілку style.

Цього разу для перенесення змін з гілки master ми будемо використовувати команду rebase замість merge.

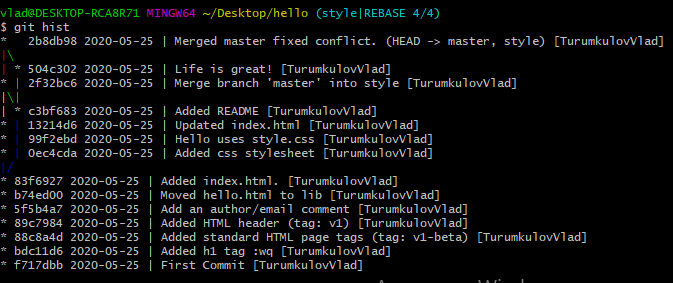


**35. Злиття в гілку master**

Злиття style в master



Перегляньте логи

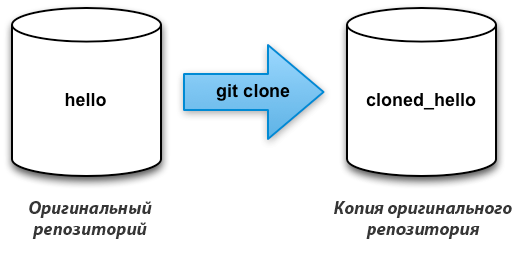


Тепер гілки style і master ідентичні.

**36. Кілька репозиторіїв**

До цих пір ми працювали з одним git репозиторієм. Однак, git вдається відмінно працювати з декількома репозиторіями. Ці додаткові репозиторії можуть зберігатися локально, або доступ до них може здійснюватися через мережеве підключення.

У наступному розділі ми створимо новий репозиторій з ім'ям «cloned\_hello». Ми покажемо, як переміщувати зміни з одного репозиторію в інший і як вирішувати конфлікти, що виникають в результаті роботи з двома репозиторіями.



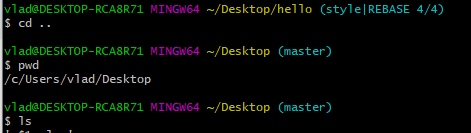
А поки що попрацюємо з локальними репозиторіями (тобто репозиторіями, що зберігаються на вашому локальному жорсткому диску), однак практично все, що ви дізнаєтесь у цьому розділі, застосовуватиметься до будь-яких репозиторіїв, незважаючи на те, чи зберігаються вони локально або є публічними.

Примітка: Ми будемо вносити зміни в обидві копії наших репозиторіїв. Звертайте увагу на те, в якому репозиторію ви знаходитесь на кожному кроці наступних уроків.

**37. Клонування репозиторіїв**

Перейдіть в робочий каталог

Перейдіть в робочий каталог і зробіть клон вашого сховища hello.



Створіть клон репозиторію hello

Давайте створимо клон репозиторію.





У вашому робочому каталозі тепер має бути два репозиторія: оригінальний репозиторій «hello» і клонований репозиторій «cloned\_hello»

**38. Перегляд клонованого репозиторію**

Подивіться на клонований репозиторій

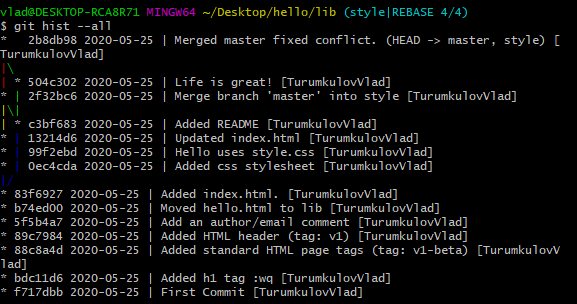
Давайте поглянемо на клонований репозиторій.







Перегляньте історію репозиторію

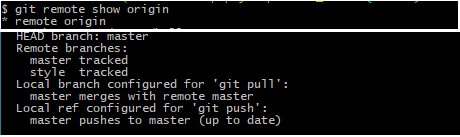


**39. Що таке origin?**

Дізнатися про імена віддалених репозиторіїв.



Ми бачимо, що клонований репозиторій знає про ім'я за замовчуванням віддаленого репозиторію. Давайте подивимося, чи можемо ми отримати більш детальну інформацію про ім'я за замовчуванням:



Ми бачимо, що «ім'я за замовчуванням» віддаленого репозиторію - оригінальне hello. Віддалені репозиторії зазвичай розміщуються на окремій машині, можливо, централізованому сервері. Однак, як ми бачимо тут, вони можуть з тим самим успіхом вказувати на репозиторій на тій самій машині. Немає нічого особливого в «імені за замовчуванням», проте існує традиція використовувати «ім'я за замовчуванням» на первинному централізованому репозиторію (якщо такий є).



**41. Зміна оригінального репозиторію**

Внесіть зміни в оригінальний репозиторій hello





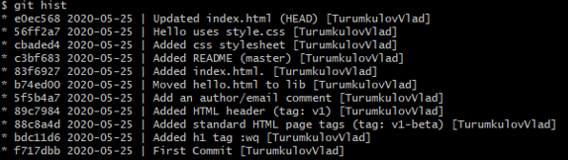
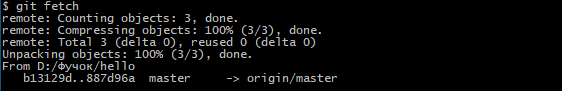




**42. Витяг змін**

Навчитися витягувати зміни з віддаленого репозиторію.





Перевірте README

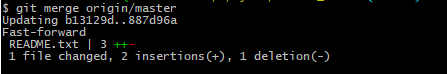
Ми можемо продемонструвати, що клонований файл README не змінився.



Як бачите, ніяких змін.

**43. Злиття витягнутих змін**

Злийте витягнуті зміни в локальну гілку master



Ще раз перевірте файл README



Ось і зміни. Хоча команда «git fetch» не зливає зміни, ми можемо вручну злити зміни з віддаленого репозиторію.

**44. Витяг і злиття змін**

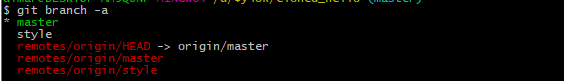
Ми не збираємося знову проходити весь процес створення нових змін та їх вилучення, але ми хочемо, щоб ви знали, що виконання:



**45. Додавання гілки відстеження**

Додайте локальну гілку, котра відстежує віддалену гілку.





Тепер ми можемо бачити гілку style у списку гілок і у лозі.

**46. Чисті репозиторії**

Чисті репозиторії (без робочих каталогів) зазвичай використовуються для розшарювання.

Невелике пояснення, що ж все-таки означає «чистий репозиторій». Звичайний git-репозиторій має на увазі, що ви будете використовувати його як робочу директорію, тому разом з файлами проекту в актуальній версії, git зберігає всі службові, «чисто-репозиторівскі» файли в піддиректорії .git. У віддалених репозиторіях немає сенсу зберігати робочі файли на диску (як це робиться в робочих копіях), а все що їм дійсно потрібно - це дельти змін та інші бінарні дані репозиторію. Ось це і є «чистий репозиторій».







Як правило, репозиторії, що закінчуються на «.git» є чистими репозиторіями. Ми бачимо, що в репозиторії hello.git немає робочого каталогу. По суті, це є не що інше, як каталог .git нечистого репозиторію

**47. Додавання віддаленого репозиторію**

Давайте додамо репозиторій hello.git до нашого оригінального репозиторію.





**48. Відправка змін**

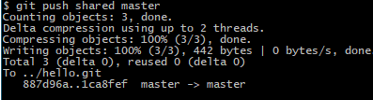
Оскільки чисті репозиторії зазвичай разшарюються на якомусь мережевому сервері, нам необхідно відправити наші зміни до інших репозиторіїв.

Почнемо зі створення змін для відправки. Відредагуйте файл README і зробіть коміт.





Тепер надішліть зміни до загального репозиторію.



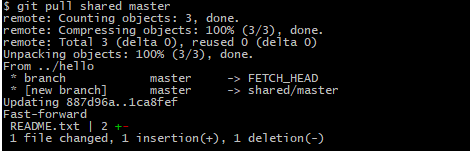
**49. Витяг загальних змін**

Швидко переключіться в клонований репозиторій і витягніть зміни, щойно відправлені в загальний репозиторій.











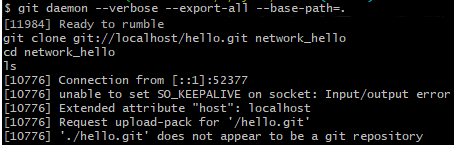


**50. Розміщення ваших git репозиторіїв**

Навчитися налаштовувати git сервер для спільного використання репозиторіїв.

Є багато способів поділитися (розшарити) git репозиторієм по мережі. Ось швидкий спосіб.

Запуск git сервера



**Контрольні питання**

1. Аліас це лише ще одна, створена вами команда для git'а, яка дозволяє запускати будь-яку іншу команду git'а або ж зовнішній по відношенню до нього скрипт. Додавати аліаси можна за допомогою утиліти **config** , наявної в git'е. Дана утиліта розрізняє три рівні дії задаються Вами налаштувань:

1) Для всіх користувачів в системі

1. $ Git config - system

2) Для поточного користувача

1. $ Git config - global

3) Для поточного git сховища

1. $ Git config

Легкий тег - це щось дуже схоже на гілку, яка не змінюється - просто показник на певний Ком.

А ось анотовані теги зберігаються в базі даних Git як повноцінні об'єкти. Вони мають контрольну суму, містять ім'я автора, його e-mail і дату створення, мають коментар і можуть бути підписані і перевірені за допомогою GNU Privacy Guard (GPG). Зазвичай рекомендується створювати анотовані теги, щоб мати всю перераховану інформацію; але якщо ви хочете зробити тимчасову мітку або з якоїсь причини не хочете зберігати іншу інформацію, то для цього годяться і легковагі.

1. **Анотовані теґи**

Створити анотований теґ у Git просто. Найлегший спосіб — додати -a до команди tag:

$ git tag -a v1.4 -m "моя версія 1.4"

$ git tag

**Легкі теґи**

Другий спосіб позначати коміти — за допомогою легких позначок. Це просто хеш коміту збережений у файлі — ніякої іншої інформації не зберігається. Щоб створити легкий теґ, не додавайте жодної з опцій -a, -s та -m, вкажіть лише назву теґу:

$ git tag v1.4-lw

$ git tag

1. tree (дерево) - по суті це просто папка з конкретним іммутабельним вмістом. Об'єкт типу tree містить поіменний список файлів з конкретним вмістом (blobs) і підпапок (trees). А дерево, на яке указиватся кожен Комміт - це коренева папка проекту, точніше її певний стан.
2. **Коміт** в **git** - **це** об'єкт, що містить посилання на: знімок стану (snapshot), що записаний раніше в індекс. метадані автора. коментарі. нуль чи більше вказівників на інші **коміти**, що є прямими батьками даного **коміту**.
3. Однак запам'ятовувати хеш для кожної версії незручно, до того ж втрачається ім'я файлу, зберігається лише вміст. Об'єкти такого типу називають БЛОБ (англ. Blob - binary large object). Маючи SHA-1 об'єкту, можна попросити Git показати нам його тип за допомогою команди cat-file -t:

$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

Blob

1. Віддалені посилання — це посилання (вказівники) у ваших віддалених сховищах: гілки, теґи тощо. Для повного списку віддалених посилань виконайте git ls-remote [remote], або git remote show [remote] для детальної інформації про віддалені гілки. Проте, найпоширеніше застосування — це віддалено-відслідковувані гілки.

Віддалено-відслідковувані гілки — це вказівники на стан віддалених гілок. Локально ці вказівники неможливо змінити, але їх змінює Git, коли ви виконуєте мережеві операції, щоб вони точно відповідали стану віддаленого сховища. Вважайте їх закладками, що нагадують вам про стан віддалених репозиторіїв на момент вашого останнього зв’язку з ними.

Віддалені гілки мають такий запис: <віддалене сховище>/<гілка>. Наприклад, якщо ви хочете побачити як виглядала гілка master з віддаленого сховища origin, коли ви востаннє зв’язувалися з ним, перейдіть на гілку origin/master. Припустимо, ви працювали з колегами над одним завданням і вони вже виклали свої зміни. У вас може бути своя локальна гілка iss53, але гілці на сервері відповідатиме віддалена гілка origin/iss53.

Це може трохи спантеличувати, розгляньмо приклад. Скажімо, ви працюєте з Git сервером, що доступний у вашій мережі за адресою git.ourcompany.com. Коли ви склонуєте з нього, команда clone автоматично іменує його origin, стягує всі дані, створює вказівник на те місце, де зараз знаходиться master і локально іменує це посилання origin/master, щоб ви могли з чогось почати працювати.

1. При використанні  pull, git намагається зробити все за вас. Він зливає будь внесені коммітов в гілку, в якій ви зараз працюєте. Команда pullавтоматично зливає коммітов, не даючи вам спочатку переглянути їх. Якщо ви не пильно стежте за гілками, виконання цієї команди може привести до частих конфліктів.

При використанні  fetch, git збирає все коммітов з цільової гілки, яких немає в поточній гілці, і зберігає їх в локальному репозиторії. Однак він не зливає їх в поточну гілку. Це особливо корисно, якщо вам потрібно постійно оновлювати свій репозиторій, але ви працюєте над функціональністю, неправильна реалізація якої може негативно позначитися на проекті в цілому. Щоб злити коммітов в основну гілку, потрібно використовувати merge.

Грубо кажучи, по дефолту git pull- це шоткод для послідовності двох команд: git fetch(отримання змін з сервера) і git merge(зливання в локальну копію).