Технології та Інструменти Розробки Програмного Забезпечення
ЛР №2 "Розширена робота з git"

Постановка задачі

1. Зклонувати репозиторій для ЛР2 використавши локальний репозиторій від ЛР1 в якості "віддаленого".
   Якщо для ЛР1 було використано не унікальний репозиторій, чи з інших причин для ЛР2 було обрано інший проект -- зклонуйте ваш новий репозиторій з інтернету, створіть свою гілку `lab1-branch` із кількома комітами і тоді використайте цей репозиторій в якості локального для клонування в ЛР2.
2. Робота з ремоутами.
    1. Додати новий ремоут використавши URI на інтернет-джерело вашого репозиторію.
    2. Показати список віддалених гілок, так щоби було видно гілки з різних ремоутів.
    3. Створити нову гілку `lab2-branch`, додати в нього кілька комітів.
    4. Пушнути гілку `lab2-branch` на ремоут, створений з ЛР1 **без зв'язування локальної гілки з віддаленою**.
    5. Додати до гілки ще коміт.
    6. Пушнути зміни гілки на ремоут, створений з ЛР1, цього разу зв'язавши гілки.
    7. Додати до гілки ще коміт.
    8. Переконатися в тому, що після зв'язування гілок тепер можна пушити просто через `git push`.
    9. Перевірити в репозиторії ЛР1, що після пушу тут з'явилася нова локальна гілка (яку ми власне пушнули).
3. Змерджити гілку, що була створена при виконанні ЛР1, в поточну гілку `lab2-branch`.
4. Перенесення комітів.
    1. Створити ще одну гілку від master-а `lab2-branch-2`, додати в неї три коміти.
    2. Перенести з гілки `lab2-branch-2` середній з трьох нових комітів в гілку `lab2-branch`.
5. Визначити останнього спільного предка між двома будь-якими гілками.
6. Робота з ничкою.
    1. Зробити трохи unstaged змін.
    2. Зберегти до ички.
    3. Зробити ще трохи unstaged змін.
    4. Зберегти до ички.
    5. Дістати з ички перші збережені зміни, ті що збереглися на кроці (6.2).
7. Робота з файлом `.gitignore`.
    1. Створити кілька файлів з якимось унікальним розширенням.
    2. Додати шаблон для цих файлів в ігнор.
    3. Перевірити статус -- файли повинні зникнути.
    4. Перевірити статус включно з ігнором.
    5. Почистити всі untracked файли з репозиторію, включно з ігнорованими.
8. Робота з `reflog`.
    1. Переглянути лог станів гілок.
    2. Створити нову гілку на будь-який стан зі списку та переключитися на цю гілку.

Контрольні питання

1. Що таке URI, який це має стосунок до `git clone`, та чому стоїть три слеша у `file:///`?
2. Що таке ремоути і для чого вони потрібні?
3. Навіщо зв'язувати локальну гілку з віддаленою через `git push -u`?
4. Яка команда використовується для об'єднання станів гілок?
5. Яка команда використовується для перенесення конкретного коміту в поточну гілку?
6. Що таке останній спільний предок та як його визначити?
7. Як користуватися ничкою?
8. Що таке файл `.gitignore`?
9. В чому небезпека існування ігнорованого сміття, як його переглянути та повністю очистити?
10. Що таке лог станів гілок та як ним скористатися?

Методичні вказівки

- Робота з віддаленим репозиторієм
- Об'єднання станів (merge)
- Перенесення комітів до поточної гілки (cherry-pick)
- Визначення останнього спільного предка (merge-base)
- Ничка (stash)
- Файл .gitignore
- Лог станів гілок (reflog)

Робота з віддаленим репозиторієм

Для початку створимо папку для виконання ЛР:

```
mkdir ~/lab2
cd ~/lab2
```

Наступні команди передбачатимуть виконання з середини директорії `lab2`.

Передбачається наявність доступ до локального клону репозиторію, в якому виконувалася ЛР №1, який для прикладу знаходиться за шляхом `~/lab1` (підставляйте свій локальний шлях)

Зклонуємо репозиторій для ЛР2. Використовуємо абсолютний шлях до локальної папки з репозиторієм від ЛР1.

```
$ git clone file:///home/omarchenko/lab1/OpenRGB
```

Оскільки `git` -- розподілена система контролю версій, кожна локальна копія може трактуватися як повноцінний репозиторій, тому для нашого клону ЛР2 -- локальна копія ЛР1 виглядає точно таким же "віддаленим сервером", як власне посиланя на github виглядало для ЛР1.

Перевіримо, що в копії репозиторію ЛР2 у нас присутні гілки, які ми створювали для ЛР1:

```
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/my-branch
```

Перевіримо, куди "дивиться" наш поточний ремоут `origin`:

```
$ git remote -v
origin  file:///home/omarchenko/lab1/OpenRGB (fetch)
origin  file:///home/omarchenko/lab1/OpenRGB (push)
```

Додамо ще один ремоут до нашої локальної копії:

```
$ git remote add upstream https://gitlab.com/CalcProgrammer1/OpenRGB.git
```

І перевіримо список ремоутів:

```
$ git remote -v
origin  file:///home/omarchenko/lab1/OpenRGB (fetch)
origin  file:///home/omarchenko/lab1/OpenRGB (push)
upstream    https://gitlab.com/CalcProgrammer1/OpenRGB.git (fetch)
upstream    https://gitlab.com/CalcProgrammer1/OpenRGB.git (push)
```

Отримаємо список змін з нового ремоута:

```
$ git fetch upstream
. . .
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/my-branch
  remotes/upstream/acermonitor
  remotes/upstream/alienware_aw510k_merge
```

Тепер стає зрозуміло, навіщо в гілках був префікс `remotes/origin`. Бо коли більше одного ремоута -- це спосіб зрозуміти якому саме ремоуту належить віддалена гілка.

З точки зору гіта один і той же репозиторій може знаходитися в різних віддалених місцях, з трошки різними станами (гілок, комітів). Ремоути -- один зі способів вести облік таких місць. Саме тому переважна більшість команд `push` вимагають або явне вказання ремоута -- конкретної віддаленої копії, в яку ми хочемо залити локальні зміни, або щоби конкретна локальна гілка була зв'язана із конкретною гілкою на конкретному ремоуті.

Створимо нову гілку від головної гілки, і покладемо в неї кілька комітів.
ВАЖЛИВО: обов'язково створюйте нові файли, не такі які ви додавали в ЛР1.

```
$ git checkout -b lab2-branch
. . .
# додали кілька комітів в нову гілку
. . .
$ git push
fatal: The current branch lab2-branch has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin lab2-branch
```

Але при спробі залити ці зміни на віддалений сервер гіт видав наступну помилку. Чому? Бо він не хоче брати на себе відповідальність обирати, на яку саме віддалену копію репозиторію ви хочете покласти нову гілку. Є два варіанти подальших дій:

1. Щоразу при кожній команді `push` вказувати явно назву віддаленої копії
2. Зв'язати локальну гілку із конкретною віддаленою

Розглянемо їх:

```
$ git push origin lab2-branch # перший варіант -- зв'язка не відбулася,
подальші пуші також вимагають явних параметрів
$ git push
fatal: The current branch lab2-branch has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin lab2-branch


$ git push -u origin lab2-branch # другий варіант -- подальші пуші саме з
цієї гілки можна виконувати без параметрів, оскільки локальна гілка
зв'язалася з конкретною віддаленою
$ git push
Everything up-to-date
```

Відрізняються лише наявністю ключа `-u`, який і проводить зв'язку. Після цього кожен наступний пуш можна робити просто як `git push` -- але саме в гілці `lab2-branch` -- `git` буде автоматично заливати у вказану віддалену копію та гілку.

Після успішного пушу, в локальній копії репозиторія ЛР1 з'явилася гілка від ЛР2.

Зверніть увагу -- віддалені гілки не є транзитивними. ЛР2 бачить "віддалені гілки" з ЛР1 лише ті, що в ЛР1 є локальними. Відповідно наш пуш, який ми щойно виконали, створив в ЛР1 нову локальну гілку.

Об'єднання станів (merge)

Розглянемо базову роботу з об'єднанням станів, інакше відому як `merge`.

Припустимо ми хочемо затягнути собі в нашу нову гілку `lab2-branch` ті зміни, що ми робили в гілці для ЛР1.

Перевіримо поточний стан гілок:

```
$ git branch -a
* lab2-branch
  master
  remotes/origin/HEAD -> origin/master
  remotes/origin/lab2-branch
  remotes/origin/master
  remotes/origin/my-branch
  remotes/upstream/acermonitor
  . . . . .
```

Бачимо, що локальної копії гілки ЛР1 у нас наразі немає. Але це не страшно, адже можна використати явно адресу коміту з віддаленої гілки:

```
$ git merge origin/my-branch

$ git log --pretty=oneline --graph
```

Тепер на графі буде явно видно, як в якийсь момент ваші гілки розійшлися, буде відмічено дві голови ваших гілок, а також останній -- об'єднуючий -- мердж-коміт.

Перенесення комітів до поточної гілки (cherry-pick)

Часто трапляється, що вам необхідно отримати у вашу гілку якийсь один (чи кілька) конкретних комітів з іншої, паралельної, гілки, але при цьому не можна брати нічого, окрім саме цих комітів. Тобто, не можна провести просто мердж. В таких ситуаціях використовується команда `git cherry-pick`.

Завдання: створити ще одну гілку `lab2-branch-2`, додати до неї три коміти (створюйте нові унікальні файли щоби уникнути конфліктів), а потім перенести в гілку `lab2-branch` лише один -- середній -- із тих трьох.

Після додавання комітів в гілку `lab2-branch-2`, використавши `git log` отримати хеш бажаного коміту. Далі, переключившись на гілку, куди ми хочемо перенести коміт -- `lab2-branch` -- використати команду, підставивши ваш бажаних хеш коміту:

```
$ git cherry-pick 850585a008abe6f7a941b046370a624f16c3f163
[lab2-branch 219aab90] lab2 commit 4
 Date: Mon Nov 13 22:56:28 2023 +0200
 1 file changed, 1 insertion(+)
```

За допомогою `git log --pretty=oneline --graph -n 10 --branches` продемонструвати факт перенесення коміту.

Ключ `--branches` означає показати коміти не лише явно назад по історії поточної гілки, а всі локальні гілки, в тому числі паралельні.

Визначення останнього спільного предка (merge-base)

Розглянемо наступний "паровозик":

```
*   465085d40dd84aee172dd8aa2b6009457430a6fe Merge remote-tracking branch
'origin/my-branch' into lab2-branch
|\
| * 1fbbbb00ba8bef038b287c9e06c9124af991d9dc (origin/my-branch) Add support
for the ASUS TUF RTX 4090 O24G OG OC
| * a934c4c0341e427d21b3f3f880409b98ef1f21f2 i2c-smbus: linux: Remove stray
whitespaces
| * 49a6905ab5594be5e85a0dbc6a81e68c4fc05f70 i2c-smbus: linux: Fix interface
detection
| * f5dc4a62c437bdfba85e37df3990e3bc6f2da2dc allow editing individual keys
for ASUS ROG Strix SCAR 17
* | deef45bd4743cf795024dcd317bb7c75c33628c1 (origin/lab2-branch) lab 2
commit 2
* | 4fb589af75c786df4cef938a8ae0dcae483910ed lab 2 commit 1
|/
* 10e53074b2b0428fb0490101331e684ba2b7b0d6 (origin/master, origin/HEAD,
master) Support for ASUS TUF RTX 4070 12G Gaming graphics card
* 96dd52a5e9c4c7464abc02843932f45eb491f43e Fix k95_plat iso key mapping
```

Часто буває що треба довідатися, в який саме момент була створена гілка. Добре, якщо вона "коротка" -- тобто з моменту гілкування від базової гілки пройшло лише кілька комітів. Але якщо гілка довгограюча і має багато комітів -- скролити гіт лог буде не ефективно. Для цього існує спеціальна команда:

```
$ git merge-base origin/lab2-branch origin/my-branch
10e53074b2b0428fb0490101331e684ba2b7b0d6
```

Результатом її виконання є лише один хеш коміту, але цей коміт є саме тим, після якого гілки "розійшлися". Цей же коміт ще називають "базовим", оскільки саме відносно нього відбувається вирішення конфліктів. Взагалі конфлікт виникає саме в той момент, коли дві паралельні гілки зробили зміни до одного і того ж місця, після чого дані зміни намагаються об'єднатися.

Ничка (stash)

Ничка (stash) -- це спеціальне місце, куди можна заховати поточні незакомічені зміни. Наприклад, вам треба терміново перемкнутися на іншу гілку, зробити в ній якесь завдання, а потім продовжити поточну роботу. Як варіант -- можна зробити нову гілку і зберегти зміни в ній. Але якщо змін не багато -- швидше буде скористатися ничкою.

Зробимо трохи змін:

```
$ git status
On branch lab2-branch
Your branch is ahead of 'origin/lab2-branch' by 6 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   11
    modified:   22

no changes added to commit (use "git add" and/or "git commit -a")
```

Збережемо ці зміни в ничку:

```
$ git stash
Saved working directory and index state WIP on lab2-branch: 219aab90 lab2
commit 3
$ git status
On branch lab2-branch
Your branch is ahead of 'origin/lab2-branch' by 6 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Як бачимо, отримали чисте робоче дерево.

Тепер можемо переглянути вміст нички:

```
$ git stash list
stash@{0}: WIP on lab2-branch: 219aab90 lab2 commit 3
```

Захованих станів може бути багато, і ничка працює як стек. Щоби видобути зміни з нички (доречі -- зовсім не обов'язково на ту ж гілку, з якої вони були заникані) можна скористатися або `git stash pop` -- він застосує і видалить останній заниканий стан, або ж `git stash apply` -- ця команда застосує останній збережений стан, але не видалить його з нички.

`git stash apply` може бути корисно коли, наприклад, є якась конфігурація, чи паролі -- щось, що не можна комітити, але постійно треба переносити з гілки на гілку і мати в робочому дереві.

Також, якщо є більше одного захованого стану -- можна передавати повну назву, щоби застосувати до поточного робочого дерева саме конкретний збережений стан, а не останній: `git stash apply stash@{0}`

Завдання: добути збережений стан будь-яким із описаних вище способів.

Файл .gitignore

Часто буває таке, що в репозиторії запускаються скрипти -- збірки, тестів -- що генерують файли, які не потрібно комітити. Але ці файли постійно світитимуться в статусі, тим самим відволікаючи від вагомих змін.

Щоби виключити можливість помилитися і не відображати в результатах такі непотрібні файли -- існує спеціальний файл `.gitignore`. Його ім'я починається з крапки, що в Лінуксі позначає прихований файл.

Вміст цього файлу -- шаблони, по одному на рядок.

Створимо кілька файлів з розширенням `.kvfpm`, а також файл `.txt`:

```
$ git status
On branch lab2-branch
Your branch is ahead of 'origin/lab2-branch' by 6 commits.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    1.kvfpm
    1.txt
    2.kvfpm
```

Створимо в корені репозиторію файл з назвою `.gitignore` (або відкриємо існуючий), і додамо до нього наступний рядок:

```
*.kvfpm
```

Після цього в статусі більше не будуть показуватися ці файли

```
$ git status
On branch lab2-branch
Your branch is ahead of 'origin/lab2-branch' by 6 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   .gitignore

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

УВАГА: використання `.gitignore` іноді може призвести до неочікуваних наслідків. При використанні ігнору необхідно *ЗАВЖДИ* пам'ятати, що просто `git status` не покаже вам чистоту репозиторію. Чому це так важливо? Бо ніколи не знаєте яке ігнороване сміття, що опинилося у вас в репозиторії випадково, призведе до проблеми. Добре, якщо на етапі збірки. А якщо є якийсь не дуже хороший скрипт, який просто пакує вміст папки і відправляє замовнику, і тоді уже на його стороні почнуться пригоди?

Тому важливо в моменти, коли у вас на проекті щось дуже дивним чином зламалося -- не впадати у відчай, а перевірити чи нема в репозиторії чогось зайвого, почистити це, і зібратися повністю з чистого стану.

Для перевірки репозиторію на наявність ігнорованих файлів використовується ключ `--ignored`

```
$ git status --ignored
On branch lab2-branch
Your branch is ahead of 'origin/lab2-branch' by 6 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   .gitignore

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    1.txt

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)
    1.kvfpm
    2.kvfpm

no changes added to commit (use "git add" and/or "git commit -a")
```

Для повної очистки **всіх** untracked файлів, в тому числі тих, що ігноруються, використовується наступна команда:

```
$ git clean -fdx
Removing 1.kvfpm
Removing 1.txt
Removing 2.kvfpm
```

Лог станів гілок (reflog)

Припустимо ви останній тиждень працювали над якоюсь фічею в своїй окремій локальній гілці. Зробили чимало роботи, але традиційно соромилися пушити на сервер, а то ж ось хтось почитає ваш код в процесі розробки.

Далі вам необхідно було перемкнутися на іншу тимчасову гілку, зробити там якийсь швидкий фікс. Після чого ви видалили цю тимчасову гілку для швидкого фіксу і готові повернутися до роботи над основною задачею... Але виявилося, що ви випадково видалили вашу фіча-гілку, разом з виконаною роботою.

Насправді ж видаливши гілку ви видалили лише посилання на коміт, при цьому самі коміти ще деякий час будуть жити в базі даних гіта (цей час залежить від налаштувань періодичності очистки комітів, на які не ведуть жодні посилання).

В такій, чи будь-якій іншій схожій ситуації, коли ви впевнені що якийсь час тому гілка мала хороший, правильний стан, а наразі чи то видалилася, чи то має зовсім інший, незрозумілий, неправильний стан -- можна скористатися логом станів гілок, він же `reflog`.

```
$ git log --pretty=oneline --graph -n 15
* 219aab909b1979db7a009dc6e81c41233a007760 (HEAD -> lab2-branch) lab2 commit
3
*   465085d40dd84aee172dd8aa2b6009457430a6fe Merge remote-tracking branch
'origin/my-branch' into lab2-branch
|\
| * 1fbbbb00ba8bef038b287c9e06c9124af991d9dc (origin/my-branch) Add support
for the ASUS TUF RTX 4090 O24G OG OC
| * a934c4c0341e427d21b3f3f880409b98ef1f21f2 i2c-smbus: linux: Remove stray
whitespaces
| * 49a6905ab5594be5e85a0dbc6a81e68c4fc05f70 i2c-smbus: linux: Fix interface
detection
| * f5dc4a62c437bdfba85e37df3990e3bc6f2da2dc allow editing individual keys
for ASUS ROG Strix SCAR 17
* | deef45bd4743cf795024dcd317bb7c75c33628c1 (origin/lab2-branch) lab 2
commit 2
* | 4fb589af75c786df4cef938a8ae0dcae483910ed lab 2 commit 1
|/
* 10e53074b2b0428fb0490101331e684ba2b7b0d6 (origin/master, origin/HEAD,
master) Support for ASUS TUF RTX 4070 12G Gaming graphics card
* 96dd52a5e9c4c7464abc02843932f45eb491f43e Fix k95_plat iso key mapping
```

В цей момент ваша локальна гілка `lab2-branch` повинна мати чимало нових комітів порівняно із її станом на сервері `origin/lab2-branch`. Якщо ви в процесі виконання пушили зміни -- не біда, просто зробіть ще кілька комітів.

Переключимося на гілку `master` та видалимо `lab2-branch` (не можна видаляти поточну активну гілку), а також видалимо цю гілку із сервера:

```
$ git checkout master

$ git branch -D lab2-branch
Deleted branch lab2-branch (was 219aab90).

$ git push -d origin lab2-branch
To file:///home/omarchenko/lab1/OpenRGB
 - [deleted]           lab2-branch
```

На цей момент здавалося би стан її повністю втрачений. Але переглянемо `reflog`:

```
$ git reflog
10e53074 (HEAD -> master, origin/master, origin/HEAD) HEAD@{0}: checkout:
moving from lab2-branch to master
219aab90 HEAD@{1}: reset: moving to HEAD
219aab90 HEAD@{2}: reset: moving to HEAD
219aab90 HEAD@{3}: cherry-pick: lab2 commit 3
```

Ваш рефлог може відрізнятися, якщо виконувалися якісь інші операції з гілками. Але тепер ми можемо дуже легко відновити нашу гілку до стану, який був після чері-піка:

```
$ git branch lab2-resurrected 219aab90

$ git checkout lab2-resurrected

$ git log --pretty=oneline --graph -n 15
* 219aab909b1979db7a009dc6e81c41233a007760 (HEAD -> lab2-resurrected) lab2
commit 3
*   465085d40dd84aee172dd8aa2b6009457430a6fe Merge remote-tracking branch
'origin/my-branch' into lab2-branch
|\
| * 1fbbbb00ba8bef038b287c9e06c9124af991d9dc (origin/my-branch) Add support
for the ASUS TUF RTX 4090 O24G OG OC
| * a934c4c0341e427d21b3f3f880409b98ef1f21f2 i2c-smbus: linux: Remove stray
whitespaces
| * 49a6905ab5594be5e85a0dbc6a81e68c4fc05f70 i2c-smbus: linux: Fix interface
detection
| * f5dc4a62c437bdfba85e37df3990e3bc6f2da2dc allow editing individual keys
for ASUS ROG Strix SCAR 17
* | deef45bd4743cf795024dcd317bb7c75c33628c1 lab 2 commit 2
* | 4fb589af75c786df4cef938a8ae0dcae483910ed lab 2 commit 1
|/
* 10e53074b2b0428fb0490101331e684ba2b7b0d6 (origin/master, origin/HEAD,
master) Support for ASUS TUF RTX 4070 12G Gaming graphics card
* 96dd52a5e9c4c7464abc02843932f45eb491f43e Fix k95_plat iso key mapping
```