Основи Git

Git це розподілена система контролю версій нашого коду. Для чого вона нам? Для розподілених команд потрібна якась система управління. Потрібно, щоб відстежувати зміни, які відбуваються з часом. Тобто, крок за кроком ми бачимо, які файли змінилися і як. Особливо це важливо, коли аналізуєш, що було зроблено в рамках одного завдання: це дає можливість повертатись назад. Уявімо ситуацію: був працюючий код, все в ньому було добре, але ми вирішили щось покращити, там підправити, там підправити.

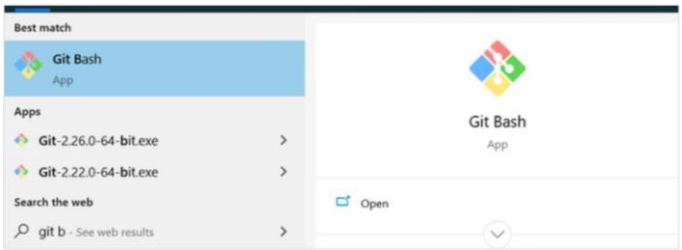
Все нічого, але таке покращення поламало половину функціоналу, унеможливило роботу. І що далі? Без Гіта треба було б годинами сидіти і згадувати, як усе було спочатку. А так ми просто відкочуємось на коміт назад — і все. Чи що робити, якщо є два розробники, які роблять одночасно свої зміни у коді? Без Гіта це виглядає так: вони скопіювали код із оригіналу, зробили що потрібно. Настає момент, і обидва хочуть додати свої зміни до головної папки. І що робити в цій ситуації?.. Я навіть не беруся оцінити час, щоб зробити цю роботу. Таких проблем не буде, якщо користуватися Гітом.

Встановлення Git

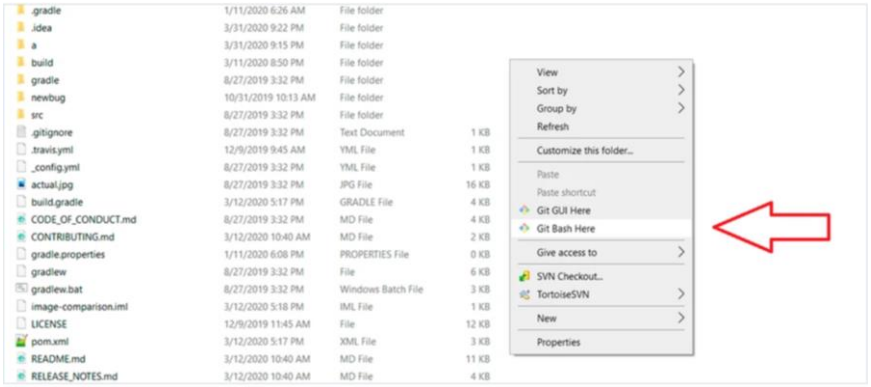
Встановимо гіт на комп'ютер. Я розумію, що всі різні OS, тому постараюся описати для кількох випадків.

Встановлення для Windows

Як завжди, необхідно завантажити exe файл та завантажити його. Тут усе просто: натискаємо на перше посилання гуглу, встановлюємо та все. Для роботи будемо використовувати bash консоль, яку вони надають. Щоб працювати в ОС, необхідно завантажити Git Bash. Ось як він виглядає у меню пуск:



І це вже консоль, у якій можна працювати. Щоб не переходити кожного разу в папку з проектом, щоб там відкрити git, можна у дирикторії правою кнопкою миші відкрити консоль з необхідним нам шляхом:



Встановлення для Linux

Зазвичай git вже встановлений та є в дистрибутивах лінуксу, т. як це інструмент, першочергово написаний для розробки ядра лінуксу. Але бувають ситуації, коли його не має. Щоб перевірити це, необхідно відкрити термінал та прописати:

git --version

Якщо буде зрозуміла відповідь, нічого встановлювати не потрібно. Відкриваємо термінал та встановлюємо. Для Ubuntu необхідно писати:

sudo apt-get install git

І все: тепер у будь-якому терміналі можна користуватися гітом.

Встановлення на macOS

Тут також для початку необхідно перевірити, чи є вже git. Якщо все ж таки ні, найпростіший шлях — це завантажити звідси останню версію. Якщо встановлений XCode, то git вже точно буде автоматично встановлений.

Налаштування git

У git є налаштування користувача, від якого буде йти робота. Це розумна й необхідна річ, т. як коли створюється коміт, git бере саме цю інформацію для поля Author. Щоб налаштувати ім’я користувача та пароль для усіх проектів, необхідно прописати наступні команди:

git config --global user.name ”Ivan Ivanov”

git config --global user.email ivan.ivanov@gmail.com

Якщо є необхідність для конкретного проекту змінити автора (для особистого проекту, наприклад), можна забрати --global, вийде:

git config user.name ”Ivan Ivanov”

git config user.email ivan.ivanov@gmail.com

Трішки теорії

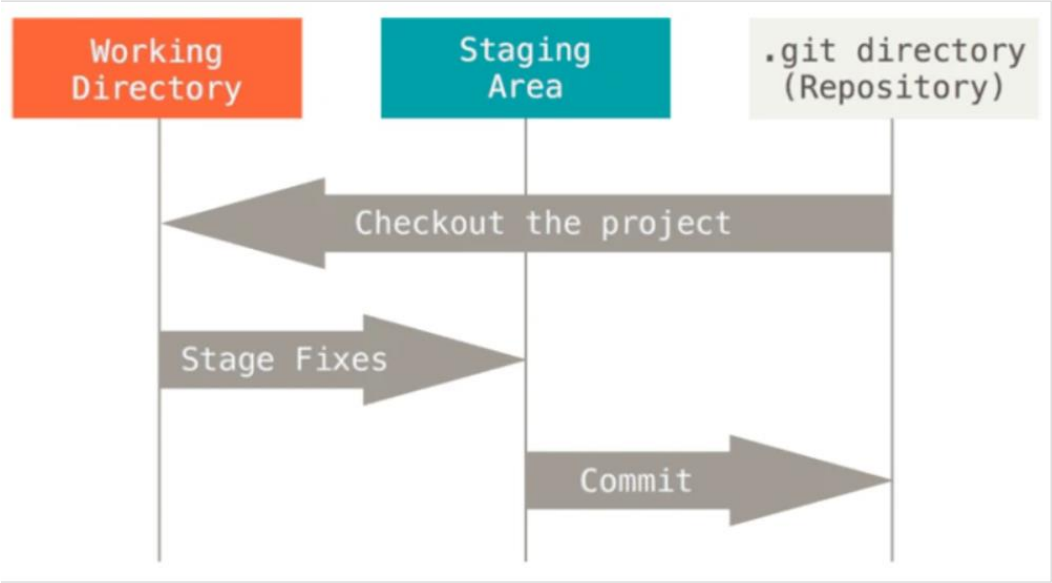
Щоб бути у темі, бажано додати у своє звернення декілька нових слів та дій.

- гіт репозиторій (git repository);
- коміт (commit);
- гілка (branch);
- змерджити (merge);
- конфлікти (conflicts);
- зпулити (pull);
- запушити (push);
- як ігнорувати якісь файли (.gitignore);
- тощо.

Стани, в яких знаходяться файли нашого коду. Тобто їхній життєвий шлях зазвичай виглядає так:

1. Файл, який створений і не доданий до репозиторію, буде в стані untracked.
2. Робимо зміни у файлах, які вже додані в гіт репозиторій – перебувають у стані modified.
3. З тих файлів, які ми змінили, вибираємо тільки ті (або всі), які потрібні нам (наприклад, скомпіловані класи нам не потрібні) і ці класи зі змінами потрапляють у стан staged.
4. Із заготовлених файлів зі стану staged створюється коміт і перетворюється на гіт репозиторій. Після цього staged стан – порожній. А ось modified ще може щось утримувати.

Виглядає це так:



Що таке коміт

Коміт – це основний об'єкт в управлінні контролю версій. Він містить усі зміни під час цього комміту. Комміти пов'язані між собою як однозв'язковий список. А саме: Є перший коміт. Коли створюється другий коміт, він (другий) знає, що йде після першого. І в такий спосіб можна відстежити інформацію. Також комміт має ще свою інформацію, так звані метадані:

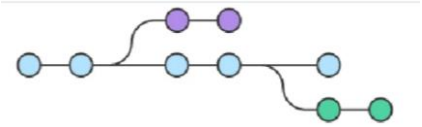
- унікальний ідентифікатор коміту, яким можна його знайти;
- ім'я автора комміту, який створив його;
- дата створення комміту;
- коментар, який визначає, що було зроблено під час цього комміту.

Виглядає це так:

```
commit 5a37824fa89b4f7650637637e022188d0ffba322
Author: romankh3 <roman.beskrovnyy@gmail.com>
Date:   Thu Feb 6 15:47:58 2020 +0200

    added order.xls file for using drools rules engine.
```

Що таке гілка?



Гілка – це покажчик якогось комміту. Так як коміт знає, який коміт був до нього, коли гілка вказує на якийсь комміт, до неї належать і всі попередні. Виходячи з цього можна сказати, що гілок, що вказують на той самий коміт, може бути скільки завгодно багато. Робота відбувається у гілках, тому коли створюється новий коміт, гілка переносить свій покажчик більш новий комміт.

Початок роботи з Гітом

Можна працювати і лише з локальним репозиторієм, і з віддаленим. Для відпрацювання необхідних команд можна скористатися лише локальним репозиторієм. Він зберігає всю інформацію лише локально у проекті в папці .git. Якщо говорити про віддалений, то вся інформація зберігається десь на віддаленому сервері: локально зберігається лише копія проекту, зміни якої можна запустити (git push) у віддалений репозиторій. Тут і далі обговорюватимемо роботу з гітом у консолі. Звичайно, можна користуватися якимись графічними рішеннями (наприклад, Intellij IDEA), але спершу потрібно розібратися, які команди відбуваються і що вони означають.

Робота з гітом у локальному репозиторії

Щоб створити локальний репозиторій, потрібно написати:

git init

Після цього буде створено порожню приховану директорію .git у тому місці, де знаходиться консоль.

.git – це директорія, яка зберігає всю інформацію про гіт репозиторії. Її видаляти не потрібно. Далі, додаються файли в цей проект, і їхній стан стає Untracked. Щоб подивитися, який статус роботи на даний момент пишемо:

git status

Ми знаходимося в master гілці, і поки ми не перейдемо в іншу, то все й залишиться. Таким чином видно, які файли змінені, але ще не додані до стану

staged. Щоб додати їх до стану staged, потрібно написати git add. Тут може бути кілька варіантів, наприклад:

• git add -A – додати всі файли зі стану в staged;

• git add . — додати всі файли з цієї папки та всі внутрішні. По суті те саме, що й попереднє;

• git add <ім'я файлу> — додає лише конкретний файл. Тут можна скористатися регулярними виразами, щоб додавати за якимось шаблоном. Наприклад, git add *.py: це означає, що потрібно додати лише файли з розширенням py. Зрозуміло, що перші два варіанти прості, а ось із додаванням буде цікавіше, тому пишемо:

```
git add *.txt
```

Щоб перевірити статус, використовуємо вже відому нам команду:

```
git status
```

Звідси видно, що регулярне вираз відпрацювало вірно, і тепер test_resource.txt знаходиться в стані staged. І, нарешті, останній етап (при локальному репозиторії, з віддаленим буде ще один);

```
git commit -m “all txt files were added to the project”
```

Щоб подивитися на історію коммітів у гілці:

```
git log
```

Тут уже видно, що з'явився наш перший коміт із текстом, який ми передали. Дуже важливо зрозуміти, що текст, який ми передаємо, має максимально точно визначати те, що було зроблено за цей коміт. Це в майбутньому допомагатиме багато разів. Допитливий читач, який ще не заснув, може сказати: а що сталося із файлом GitTest.java? Зараз дізнаємося, використовуємо для цього:

```
git status
```

Як бачимо, він так і залишився в змозі untracked і чекає свого часу. А може, ми зовсім не хочемо його додавати в проект? Буває й таке. Далі, щоб стало цікавіше, спробуємо змінити текстовий файл test_resource.txt. Додамо туди якийсь текст і перевіримо стан:

```
git status
```

Тут добре видно різницю між двома станами — untracked і modified. GitTest.java знаходиться в стані untracked, а test_resource.txt знаходиться в modified. Тепер, коли вже є файли в стані modified, ми можемо подивитися на зміни, які були зроблені над ними. Зробити це можна за допомогою команди:

git diff

Тобто тут добре видно, що я додав до нашого текстового файлу hello world! Додаємо зміни в текстовому файлі та комітімо:

git add test_resource.txt

git commit -m “added hello word! to test_resource.txt”

Щоб подивитися на всі комміти, пишемо:

git log

Як бачимо, вже є два коміти. Так само додаємо і GitTest.java. Тепер без коментарів, просто команди:

git add GitTest.java

git commit -m “added GitTest.java”

git status

Робота з .gitignore

Ми хочемо зберігати лише вихідний код і нічого іншого у репозиторії. А що ще може бути? Як мінімум, скомпіловані класи та/або файли, які створюють середовища розробки. Щоб гіт їх ігнорував, є спеціальний файл, який потрібно створити. Робимо це: створюємо файл у корені проекту під назвою .gitignore, і в цьому файлі кожен рядок буде шаблоном для ігнорування. У цьому прикладі гіт ігнор виглядатиме так:

```
*.class
target/
*.iml
.idea/
```

Дивимося тепер:

• перший рядок – це ігнорування всіх файлів з розширенням .class;

• другий рядок – це ігнорування папки target та всього, що вона містить;

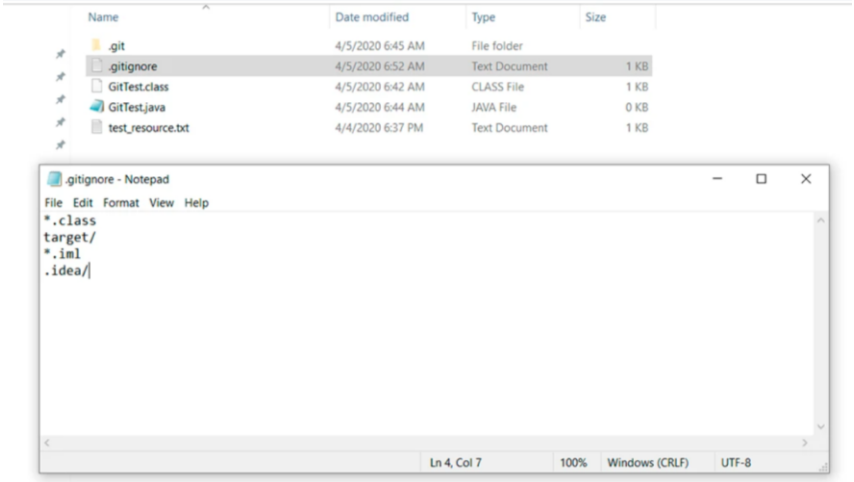
• третій рядок – це ігнорування всіх файлів з розширенням .iml;

• четвертий рядок – це ігнорування папки .idea.

Спробуємо з прикладу. Щоб подивитися, як це працює, додамо скомпільований клас GitTest.class в проект і подивимося статус проекту:

git status

Ми не хочемо якось випадково (якщо використовувати git add-A) додати скомпільований клас у проект. Для цього створюємо .gitignore файл і додаємо все, що описувалося раніше: Тепер додамо новим комітом гіт ігнор у проект:



git add .gitignore

git commit -m “added .gitignore file”

І тепер момент істини: у нас є в untracked стані скомпільований клас GitTest.class, який ми не хотіли додавати в гіт репозиторій. Ось тут і повинен заробити гіт ігнор:

git status

Все чисто) Гіт ігнору +1)

Робота з гілками

Зрозуміло, працювати в одній гілці незручно одному і неможливо, коли в команді більше однієї людини. І тому існує розгалуження.

Гілка – це просто рухливий покажчик на комміти. У цій частині розглянемо роботу в різних гілках: як змерджити зміни однієї гілки в іншу, які можуть виникнути конфлікти та багато іншого. Щоб подивитися список усіх гілок у репозиторії та зрозуміти, на якій перебуваєш, потрібно написати:

```
git branch -a
```

Видно, що у нас тільки одна гілка master, і зірочка перед нею каже, що ми на ній. До речі, щоб дізнатися, на якій гілці ми знаходимося, можна скористатися і перевіркою статусу (git status). Далі є кілька варіантів створення гілок (може їх і більше, я використовую ці):

• створити нову гілку на основі тієї, на якій перебуваємо (99% випадків);

• створити гілку з урахуванням конкретного комміту (1%).

Створюємо гілку на основі конкретного комміту

Спиратимемося на унікальний ідентифікатор коміту. Щоб знайти його, напишемо:

```
git log
```

Видімо коміт із коментарем “added hello world…”. У нього унікальний ідентифікатор - "6c44e53d06228f888f2f454d3cb8c1c976dd73f8". Створюємо гілку development з цього комміту. Для цього напишемо:

```
git checkout -b development 6c44e53d06228f888f2f454d3cb8c1c976dd73f8
```

Створюється гілка, в якій будуть лише перші два коміти з гілки master. Щоб перевірити це, ми спершу переконаємося, що перейшли в іншу гілку і подивимося на кількість комітів:

```
git status
git log
```

Вийшло, що у нас два коміти. До речі, цікавий момент: у цій гілці ще немає файлу .gitignore, тому наш скомпільований файл (GitTest.class) тепер підсвічується в untracked стані. Тепер можемо ще раз провести ревізію наших гілок, написавши:

```
git branch -a
```

Видно, що є дві гілки – master та development – і зараз стоїмо на development.

Створюємо гілку на основі поточної

Другий спосіб створення гілки – створення на основі іншої. Створюємо гілку на основі master гілки, тому потрібно спершу перейти на неї, а вже наступним кроком створити нову. Дивимося:

- git checkout master — переходимо на гілку master;
- git status — перевіряємо, чи на майстрі.

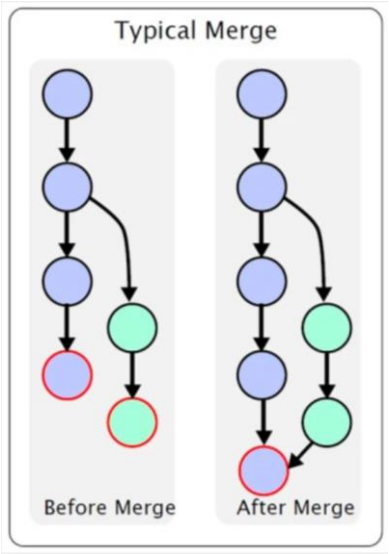
Ось тут видно, що ми перейшли на master гілку, тут вже працює гіт ігнор, і скомпільований клас уже не світиться як untracked. Тепер створюємо нову гілку на основі master гілки:

```
git checkout -b feature/update-txt-files
```

Якщо є сумніви, що ця гілка буде не такою ж, як і master, можна це легко перевірити, написавши git log і подивитися на всі комміти. Там їх має бути чотири.

Резолвім конфлікти

Перш ніж розібратися з тим, що таке конфлікт, слід поговорити про злиття (смерджування) однієї гілки в іншу. Ось таким малюнком можна показати процес, коли одну гілку мерджають в іншу:



Тобто є головна гілка. Від неї в якийсь момент створюють другорядну, в якій відбуваються зміни. Як тільки робота зроблена, потрібно злити одну гілку в

іншу. Я не описуватиму різні особливості: я хочу донести в рамках цієї статті лише розуміння, а вже деталі дізнаєтеся самі, якщо буде потрібно. Так ось, на нашому прикладі ми створили гілку feature/update-txt-files. Як написано в імені гілки – оновимо текст.

Тепер потрібно створити під цю справу новий коміт:

git add *.txt

git commit -m “updated txt files”

git log

Тепер, якщо ми хочемо смердити feature/update-txt-files гілку в master, потрібно перейти в master і написати git merge feature/update-txt-files:

git checkout master

git merge feature/update-txt-files

git log

Як результат – тепер і в майстер гілці є комміт, який був доданий у feature/update-txt-files. Ця функціональність додана, тому можна видалити фіче (feature) гілку. Для цього напишемо:

git branch -D feature/update-txt-files

Ускладнюємо ситуацію: тепер припустимо, що знову потрібно змінити файл txt. Але тепер ще й у майстрі цей файл буде змінено також. Тобто він паралельно змінюватиметься, і гіт не зможе зрозуміти що потрібно робити в ситуації, коли ми захочемо смердити в master гілку новий код.

Створюємо нову гілку на основі master, робимо зміни до text_resource.txt і створюємо коміт під цю справу:

git checkout -b feature/add-header

... робимо зміни у файлі

git add *.txt

git commit -m “added header to txt”

Переходимо на master гілку і також оновлюємо цей текстовий файл у тому ж рядку, що і фіче гілка:

git checkout master

… оновили test_resource.txt

git add test_resource.txt

git commit -m “added master header to txt”

І тепер найцікавіший момент: потрібно змерджити зміни з feature/add-header гілки в master. Ми знаходимося в майстер гілці, тому потрібно тільки написати:

git merge feature/add-header

Але ми отримаємо результат із конфліктом у файлі test_resource.txt. І тут ми можемо бачити, що гіт не зміг самостійно вирішити, як змерджити цей код і каже, що треба спочатку розсунути конфлікт, а вже потім зробити коміт. Відкриваємо в текстовому редакторі файл, у якому конфлікт, і бачимо: Щоб зрозуміти, що тут зробив гіт, потрібно згадати, що ми писали, і порівняти:

1. між “<<<<<<< HEAD” та “=======” знаходяться зміни майстер, які були в цьому рядку в майстер гілці.

2. між “=======” та “>>>>>>> feature/add-header” знаходяться зміни, які були у feature/add-header гілці.

Таким чином гіт показує, що тут він не зміг зрозуміти, як злити воєдино цей файл, розділив цю ділянку на дві частини з різних гілок і запропонував вирішити нам самим. Добре, твердою волею вирішую прибрати все, залишити тільки слово header:

Подивимося на статус змін, опис буде дещо іншим. Буде не modified стан, а Unmerged. Так що сміливо можна було додати п'ятий стан… Але я думаю, що це зайве, подивимося:

```
git status
```

Переконалися, що то інший випадок, незвичайний. Продовжуємо:

```
git add *.txt
```

В описі можна помітити, що пропонують написати лише git commit. Слухаємо та пишемо:

```
git commit
```

У такий спосіб ми зробили це — розрізнили конфлікт у консолі. Звичайно, в середовищах розробки можна це зробити трохи простіше, наприклад, Intellij IDEA все налаштовано так добре, що можна виконувати всі необхідні дії в ній. Але середовище розробки робить багато чого під капотом, і ми часто не розуміємо, що саме там сталося. А коли немає розуміння, то можуть виникнути й проблеми.

Робота з віддаленими репозиторіями

Останній крок – розібратися ще з кількома командами, які потрібні для роботи з віддаленим репозиторієм. Як я вже казав, віддалений репозиторій — це місце, де зберігається репозиторій і звідки можна його клонувати. Які бувають видалені репозиторії?

GitHub — це найбільше сховище для репозиторіїв та спільної розробки.

GitLab — веб-інструмент життєвого циклу DevOps з відкритим вихідним кодом, що представляє систему управління репозиторіями коду для Git з власної вікі, системою відстеження помилок, CI/CD пайплайн та іншими функціями.

- Після новини про те, що Microsoft купила GitHub, деякі розробники продублювали свої напрацювання у GitLab.

BitBucket — веб-сервис для хостинга проектов и их совместной разработки, основанный на системе контроля версий Mercurial и Git. Одно время имел большое преимущество перед GitHub в том, что у него были бесплатные приватные репозитории. В прошлом году GitHub также открыл эту возможность для всех бесплатно.

- тощо.

Перше, що потрібно зробити у роботі з віддаленим репозиторієм, — клонувати проект собі у локальний. Для цього я експортував проект, який ми робили локально, і тепер кожен його може собі клонувати, написавши:

git clone https://github.com/romankh3/git-demo

Наразі локально є повна копія проекту. Щоб бути впевненим, що локально знаходиться остання копія проекту, потрібно, як кажуть, спулити дані, написавши:

git pull

У нашому випадку зараз нічого не змінилося віддалено, тому й відповідь Already up to date. Але якщо я внесу якісь зміни у віддаленому репозиторії, локальний оновиться після того, як ми їх спулимо. І, нарешті, остання команда запустити дані на віддалений репозиторій. Коли ми локально щось зробили і хочемо передати це на віддалений репозиторій, потрібно спочатку створити новий коміт локально. Для цього додамо в наш текстовий файл ще щось.

Тепер уже звичайна для нас річ — створюємо коміт під цю справу:

git add test_resource.txt

git commit -m “prepated txt for pushing”

І тепер команда, щоб відправити це на віддалений репозиторій:

git push

Шпаргалка з основними командами для Git

Конфігурація

git config --global user.name "[name]" — встановити ім'я, яке прикріплюватиметься до комміту.

git config --global user.email "[email address]" — встановити email, який прикріплюватиметься до комміту.

git config --global color.ui auto — увімкнути корисне підсвічування командного рядка.

git config --global push.default current — оновлювати віддалену гілку з таким самим ім'ям, що і локальна, при пуші змін (якщо не вказано інше).

git config --global core.editor [editor] — встановити редактор для редагування повідомлень комміту.

git config --global diff.tool [tool] — встановити програму для вирішення конфліктів при злитті.

Створення репозиторіїв

git init [project-name] — створити новий локальний репозиторій із заданим ім'ям.

git clone [url] — завантажити проект та його повну історію змін.

Робота із змінами

git status — повний список змін файлів, які чекають на комміт.

git status -s — короткий вид змін.

git diff — показати зміни у файлах, які ще не були додані до індексу комміту (staged).

git add [file] — зробити вказаний файл готовим до комміту.

git add . — зробити всі змінені файли готовими до комміту.

git add '*.txt' — додати лише файли, що відповідають зазначеному виразу.

git add --patch filename — дозволяє вибрати які зміни з файлу додадуться до комміту.

git diff --staged — показати що було додано в індекс за допомогою git add, але ще не було закоммічено.

git diff HEAD — показати, що змінилося з останнього комміту.

git diff HEAD^ — показати, що змінилося з передостаннього комміту.

git diff [branch] — порівняти поточну гілку із заданою.

git difftool -d — те саме, що і diff, але показує зміни в заданій difftool.

git difftool -d master.. — показать показати зміни, зроблені у поточній гілці.

git diff --stat — показати статистику які файли були змінені та як.

git reset [file] — прибрати файли з індексу комміту (зміни не губляться).

git commit — записати зміни до репозиторію. для написання повідомлення відкриється призначений редактор.

git commit -m "[descriptive message]" — записати зміни із заданим повідомленням.

git commit --amend — додати зміни до останнього коміту.

Робота з гілками

git branch — список усіх локальних гілок у поточній директорії.

git branch [branch-name] — створити нову гілку.

git checkout [branch-name] — перейти на вказану гілку та оновити робочу директорію.

git checkout -b <name> <remote>/<branch> — перейти на віддалену гілку.

git checkout [filename] — повернути файл у початковий стан якщо він ще не був доданий до індексу комміту.

git merge [branch] — з'єднати зміни в поточній гілці із змінами із заданої.

git merge --no-ff [branch] — з'єднати гілки без режиму “fast forwarding”.

git branch -a — переглянути повний список локальних та віддалених гілок.

git branch -d [branch] — видалити задану гілку.

git branch -D [branch] — примусово видалити задану гілку, ігноруючи помилки.

git branch -m <oldname> <newname> — перейменувати гілку.

Робота з файлами

git rm [file] — видалити файл із робочої директорії та додати до індексу інформацію про видалення.

git rm --cached [file] — видалити файл із репозиторію, але зберегти його локально.

git mv [file-original] [file-renamed] — змінити ім'я файлу та додати до індексу комміту.

Відстеження файлів

.gitignore — текстовий файл, у якому задаються правила виключення файлів з репозиторію. Наприклад:

- *.log
- build/
- temp-*

git ls-files --other --ignored --exclude-standard — список всіх ігнорованих файлів.

Збереження фрагментів

git stash — покласти в тимчасове сховище всі файли, що відстежуються.

git stash pop — відновити останні файли, покладені у тимчасове сховище.

git stash list — список усіх збережених змін у тимчасовому сховищі.

git stash drop — видалити останні файли, покладені в тимчасове сховище.

Перегляд історії

git log — список зміни поточної гілки.

git log --follow [file] — список змін поточного файлу, включаючи перейменування.

git log --pretty=format:"%h %s" –graph — зміна виду відображення історії змін.

git log --author='Name' --after={1.week.ago} --pretty=oneline --abbrev-commit — подивитися над чим працював заданий користувач останній тиждень.

git log --no-merges master.. — переглянути історію змін тільки для поточної гілки.

git diff [file-branch]..[second-branch] — подивитися відмінності між двома заданими гілками.

git show [commit] — показати методату та зміни у заданому коментарі.

git show [branch]:[file] — подивитися на файл в іншій гілці, не перемикаючись на неї.

Відміна коммітів

git reset — прибрати зміни з індексу комміту, самі зміни залишаться.

git reset [commit/tag] — скасувати всі коміти після вказаного комміту, зміни будуть збережені локально.

git reset --hard [commit] — примусово повернуться до зазначеного комміту, не зберігаючи історію та зміни.

Синхронізація змін

git fetch [bookmark] — завантажити всю історію із заданого віддаленого репозиторію.

git merge [bookmark]/[branch] — злити зміни локальної гілки та заданої віддаленої.

git push — запушити поточну гілку у віддалену гілку.

git push [remote] [branch] — запушити гілку у вказаний репозиторій та віддалену гілку.

git push [bookmark] :[branch] — у віддаленому репозиторії видалити задану гілку.

git push -u origin master — якщо віддалена гілка не встановлена як відстежувана, то зробити її такою.

git pull — завантажити історію та зміни віддаленої гілки та зробити злиття з поточною гілкою.

git pull [remote][branch] — вказати конкретну віддалену гілку для злиття.

git remote — переглянути список доступних віддалених репозиторіїв.

git remote -v — переглянути детальний список доступних віддалених репозиторіїв.

git remote add [remote][url] — додати новий віддалений репозиторій.

Документація:

https://git-scm.com/book/uk/v2