Лабораторная работа №3 . Система контроля версий Git

3.1 Создание Git-репозитория.

Цель задания: Научиться создавать новые репозитории в системе контроля Git в среде GitBash. Научиться правильно перемещаться внутри проекта с использованием команд в командной строке, а также установка авторства для проекта, с целью отслеживания изменений.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Система спроектирована как набор программ, специально разработанных с учётом их использования в сценариях. Это позволяет удобно создавать специализированные системы контроля версий на базе Git или пользовательские интерфейсы. Например, Cogito является именно таким примером оболочки к репозиториям Git, а StGit использует Git для управления коллекцией исправлений (патчей).

Git поддерживает быстрое разделение и слияние версий, включает инструменты для визуализации и навигации по нелинейной истории разработки. Как и Darcs, BitKeeper, Mercurial, Bazaar и Monotone[en], Git предоставляет каждому разработчику локальную копию всей истории разработки, изменения копируются из одного репозитория в другой.

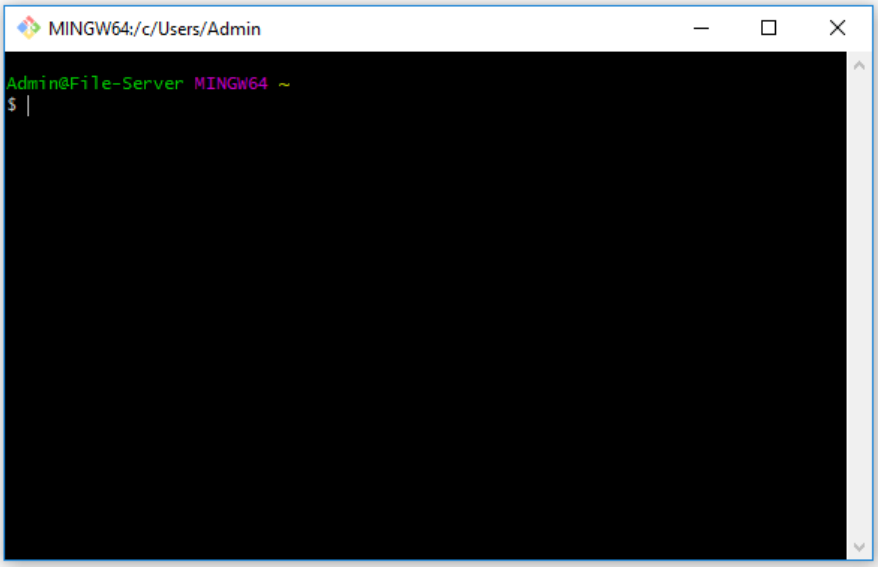
Удалённый доступ к репозиториям Git обеспечивается git-демоном, SSH- или HTTP-сервером. TCP-сервис git-daemon входит в дистрибутив Git и является наряду с SSH наиболее распространённым и надёжным методом доступа. Метод доступа по HTTP, несмотря на ряд ограничений, очень популярен в контролируемых сетях, потому что позволяет использовать существующие конфигурации сетевых фильтров.

Практическая часть.

Создание репозитория.

2. Запустите GitBash из меню Пуск - Git.

Откроется следующая консоль:

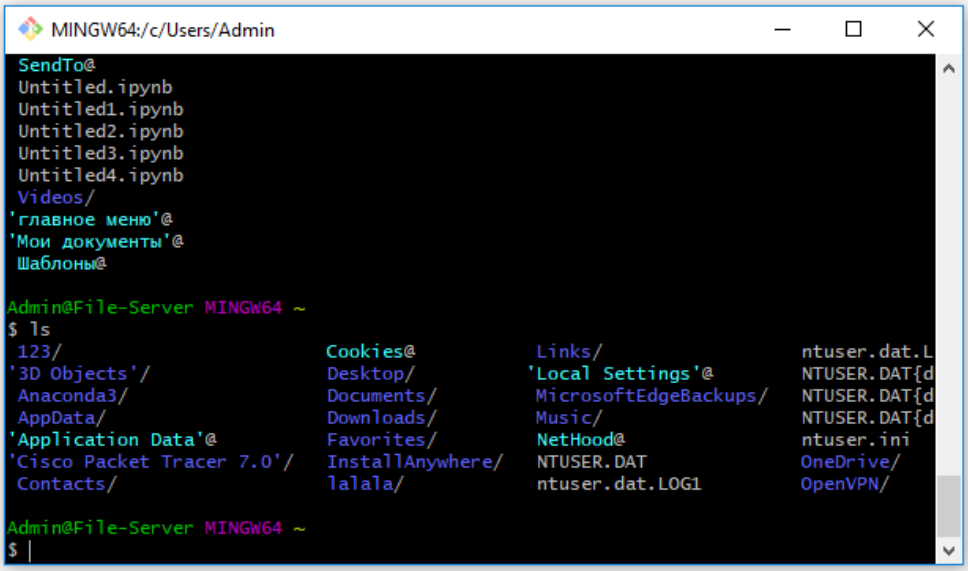


Это командная строка Linux (наподобие консоли командной строки Windows), аккуратно перенесенная в Windows.

Далее работа с Git будет объясняться на примере работы с консольным клиентом по следующим причинам:

- Чтобы у вас складывалось понимание происходящего и при возникновении проблем вы могли четко объяснить, что вы делали, и было видно, что пошло не так.
- Все нажатия кнопок в графических клиентах в итоге сводят к выполнению определенных команд консольного клиента, в то же время возможности графических клиентов ограничены по сравнению с консольным

3. Наберите команду ls. Если после этого вы получите список файлов и папок, значит, Bash успешно установилсяи запустился.

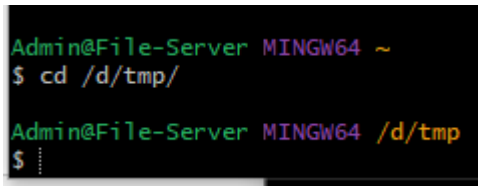


4. Далее для работы нам потребуется создать папку. Создайте её, например, в корне диска d (непосредственно из под Windows), назовите TMP.

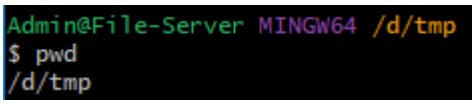
Теперь нужно в Bash перейти в эту папку. Для этого используем команду cd (changedirectory):

$ cd /d/tmp/

Нажмите Enter и вы окажетесь в этой папке. Если никаких сообщений об ошибке не выводится, значит, команда выполнена правильно.



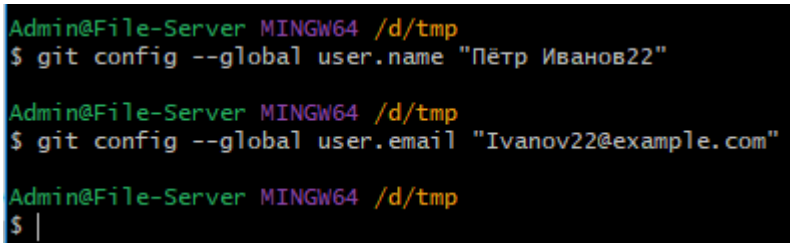
5. Команда pwd показывает, какая директория текущая в данный момент. Наберите команду и проверьте, где вы находитесь.



6. Далее следует задать настройки Git. Они используются для того, чтобы отслеживать авторов изменений. На своем домашнем компьютере следует задать реальные имя, фамилию и email. В учебной лаборатории задайте имя «Пётр ИвановN», «IvanovN@example.com», где N – номер компьютера в лаборатории (используйте свои имя и фамилию).

Обратите внимание на то, что нажимая кнопку ↑ на клавиатуре, можно повторять ранее использованные команды, они будут выводиться в командной строке, после чего их можно редактировать и выполнять. Это значительно ускорит работу.

Попробуйте после ввода имени повторить команду и отредактировать её, задав адрес электронной почты.



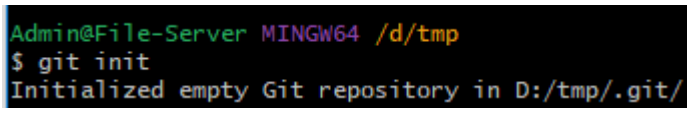
Ключ *–global* означает, что для всех репозиториев будут действовать одни и те же настройки (если задать ключ --local или вообще не задать ключ, настройки будут храниться в данном репозитории и распространяться только на него).

7. Убедитесь, что вы находитесь в папке будущего репозитория (команда *pwd*). Выведите содержимое репозитория (команда *ls*). Убедитесь, что в данный момент папка пуста.

8. Дайте команду

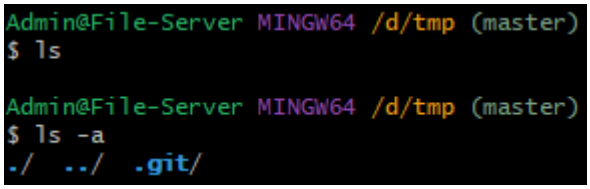
gitinit

Эта команда инициализирует репозиторий в текущей пустой папке, о чем выведется сообщение:



9. Выполните команду ls. Папка по-прежнему пуста.

Теперь введите ту же команду с ключом –a:

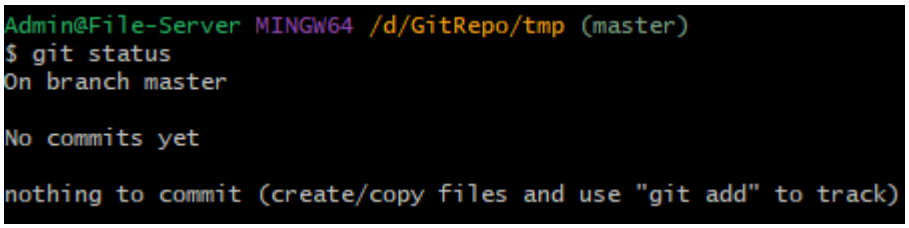


Вы видите, что в папке появились скрытые папки для служебных целей, созданные Git.

10. Попробуйте на диске D: создать пустую папку GitRepo и перенести туда данную папку tmp(сделайте это непосредственно из папки Мой компьютер, Bash можно не использовать). Далее в GitBashперейдите в эту папку (команда cd).

11. Проверьте, что скрытые файлы по-прежнему на месте, т.е. это по-прежнему репозиторий:

12. Дайте команду *git status*.



Эта команда показывает, в каком состоянии в данный момент находится наш репозиторий.

В данном случае Git сообщает, что фиксировать нечего, изменений внутри репозитория не было. Т.о. к абсолютному пути репозиторий не привязан.

Контрольные вопросы:

1. Что такое GitBash?
2. Для чего нужна команда ls?
3. Как сменить директорию?
4. Как отобразить текущую директорию?
5. Для чего задаются настройки Git?
6. Как применить одинаковые настройки для всех репозиториев?

Задание №3.2. Просмотр истории коммитов. Операции отмены.

Цель задания:

Научиться вести историю изменений. Уметь правильно интерпретировать текущий статус репозитория.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Нижний уровень git является так называемой контентно-адресуемой файловой системой. Инструмент командной строки git содержит ряд команд по непосредственной манипуляции этим репозиторием на низком уровне. Эти команды не нужны при нормальной работе с git как с системой контроля версий, но нужны для реализации сложных операций (ремонт повреждённого репозитория и так далее), а также дают возможность создать на базе репозитория git своё приложение.

Практическая часть.

История изменений.

1. Создайте в папке tmp файл README.txt.

2. Вызовите команду git status.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master

No commits yet

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

        README.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Мы видим, что появился неотслеживаемый файл (Untracked files). Т.е. в нашем репозитории появились посторонние файлы, которые Git видит, но пока не отслеживает.

3. Первое, что нужно сделать при таких изменениях – передать файл под контроль Git. Для этого используется команда git add.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add README.txt
```

Будьте внимательны с регистром!

Если команда ничего не вывела, значит, она завершилась успешно.

4. Проверьте git status снова.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README.txt
```

На этот раз Git видит этот файл, и сообщает, что он появился в репозитории. Эти изменения ещё не зафиксированы, но они уже проиндексированы. Т.е. Этот файл готов к тому, чтобы быть зафиксированным.

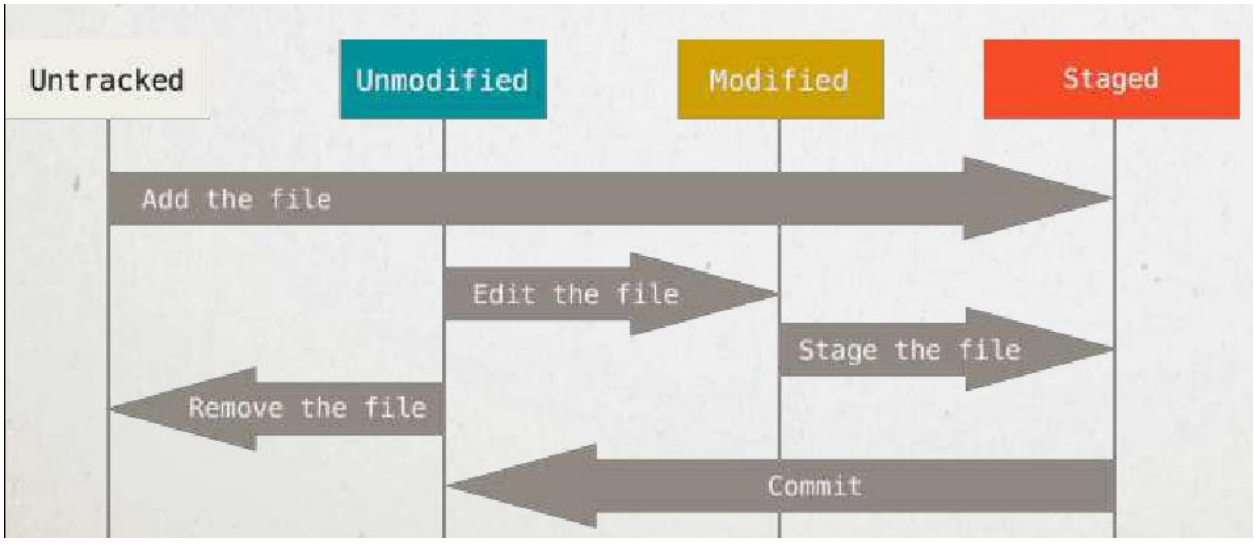
5. Следующий шаг, который мы должны сделать, передав файл под контроль Git и закончив все изменения в нём – закоммитить их (фиксировать изменения).

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git commit -m "Добавлен файл README"
[master (root-commit) 2519868] Добавлен файл README
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.txt
```

Команду gitcommit следует ОБЯЗАТЕЛЬНО использовать с ключом –m и комментарием!

Рассмотрим полученное от Git сообщение. Тут написано, что один файл изменён.

Итак, в Git файлы могут находиться в четырёх состояниях:



1) Untracked – файл просто находится в папке, но для Git он неизвестен, его в истории нет и никогда не было.

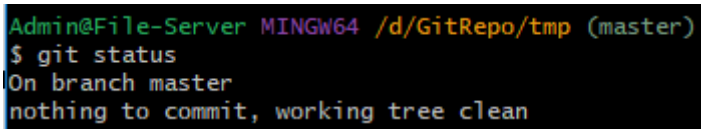
2) Staged – подготовленный для фиксации изменений (командой add). Это значит, что файл под контролем Git и он ждёт нашей команды, чтобы зафиксировать изменения (т.е. установить флажок в истории с комментарием).

3) Unmodified–означает, что файл со времени последней фиксации не менялся. Переводится в это состояние фиксацией (commit).

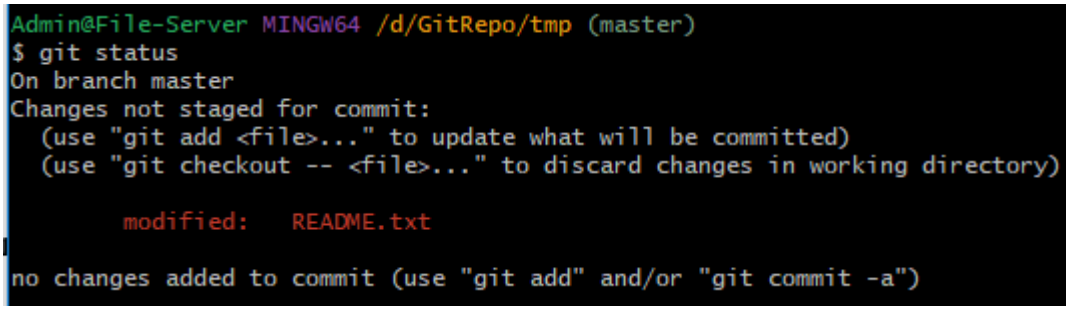
4) Modified – измененный файл. Этот файл уже был в истории, Git о нём знает, файл участвовал в каких-то коммитах, но мы его сейчас изменили. Теперь мы можем либо перевести его в Staged (add), либо отменить изменения.

Наша задача – отчётливо понимать, как происходит перемещение файла по этим состояниям. Результатом работы должно быть состояние файлов Unmodified.

5. Проверьте gitstatus снова. В данный момент все должно быть Unmodified – nothingtocommit.



6. Откройте файл README.txt и напишите в нем «Hello, world!». Сохраните файл.Проверьте gitstatus снова.



Gitсообщает о том, что файл был изменён. Ионнеготовкфиксации (changes not staged for commit).

Каким образом Git это определяет? У него в папке хранятся идеальные копии наших файлов, и он очень быстро их сравнивает с реальными файлами.

7. Готовим изменения к фиксации:

git add README.txt

8. Проверяем текущий статус.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add README.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt
```

Видим, что измененный файл *README.txt* готов к фиксации.

9. Далее следует зафиксировать изменения (commit). Не забудьте про ключ и комментарий!!!

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git commit -m "Изменения в README"
[master 49f0cfd] Изменения в README
 1 file changed, 1 insertion(+)
```

Дополнительные команды:

git rm FILE–удаляет файл из индекса и из рабочей папки;

git rm --cachedFILE – удаляет файл только из индекса Git.

Оба этих удаления требуют фиксации!

10. Для того, чтобы просмотреть историю коммитов, нужно использовать команду gitlog – это лог всех изменений.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git log
commit 49f0cfdb330fd2d8a4319f16a39e3cb55f0e4b11 (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 09:34:33 2018 +0300

    Изменения в README

commit 251986888c8f25d0f2c28bc0102b64c0f3cff567
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 09:00:59 2018 +0300

    Добавлен файл README
```

Здесь мы видим созданные нами два коммита. Длинные числа из шестнадцатеричных цифр – это их номера (кэш). Такие длинные номера нужны для того, чтобы они были уникальными. Они непоследовательны, это сделано специально, для веток (они будут рассмотрены далее).

К каждому коммиту указан автор и дата/время изменений.

11. Создайте в нашей папке ещё один файл test.txt, напишите внутри него TEST. В файле README добавьте пару восклицательных знаков или что-либо ещё. Таким образом мы получили два изменения в репозитории.

12. Проверьте текущий статус.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.txt

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

        test.txt

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

Мы видим сообщение о том, что два изменения дожидаются **стейджинга**, т.е. индексации. Для чего вообще выполняется индексация? Таким образом мы показываем, что изменения в данном файле важны и мы планируем их зафиксировать. Если файл не проиндексирован, значит, мы продолжаем его обрабатывать и не уверены, что будем сохранять изменения.

13. Добавляем (индексируем) оба файла:

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add README.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add test.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt
        new file:   test.txt
```

14. Зафиксируем оба изменения в одном коммите:

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git commit -m "Тестовый коммит с 2 изменениями"
[master 57f6580] Тестовый коммит с 2 изменениями
 2 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 test.txt
```

Просмотрите историю коммитов в gitlog. Там появился новый только что созданный коммит.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git log
commit 57f658087d95ad4feb553ef3f30309422abba458 (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 12:07:40 2018 +0300

    Тестовый коммит с 2 изменениями

commit 49f0cfdb330fd2d8a4319f16a39e3cb55f0e4b11
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 09:34:33 2018 +0300

    Изменения в README

commit 251986888c8f25d0f2c28bc0102b64c0f3cff567
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Aug 14 09:00:59 2018 +0300

    Добавлен файл README
```

Чтобы более подробно видеть информацию о коммитах, в gitlog можно использовать ключ -p. В этом случае по каждому изменению будет выведено, что было и что стало в каждом измененном файле.

Отмена коммитов.

Крайне нежелательно отменять коммиты, особенно при работе в группе. Это похоже на бухгалтерскую проводку – если документ был проведён, то его нельзя удалить, а можно лишь выполнить коррекцию.

Однако, если очень нужно, то для этого есть специальные команды.

gitresetHEAD– откат к последнему коммиту (здесь HEAD–указатель на текущий коммит). То есть, если что-то пошло не так, то можно отменить все изменения и откатиться к состоянию после последнего коммита.

gitreset --hard<commit> - самая мощная и самая опасная команда – откат к указанному коммиту с потерей всех изменений!

1. Для примера что-нибудь изменим, а затем откатимся к состоянию после коммита. Откройте файл README, замените !!! на ???. Файл test.txt удалите.

2. Выведите текущий статус.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   README.txt
        deleted:    test.txt

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

Далее выполните gitadddля обоих файлов, чтобы уж окончательно проиндексировать изменения.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add README.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git add test.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt
        deleted:    test.txt
```

И вдруг в этот самый последний момент мы решаем всё изменить!

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git reset HEAD
Unstaged changes after reset:
M       README.txt
D       test.txt
```

Всё вернулось в состояние до индексации (убедитесь в этом, проверив текущий статус).

3. Далее, чтобы вернуться в состояние до изменений, следует использовать команду gitcheckout--test.txtи gitcheckout--README.txt.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git checkout test.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git checkout README.txt

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Убедитесь, что удалённый файл вернулся на место, а во втором отменились изменения.

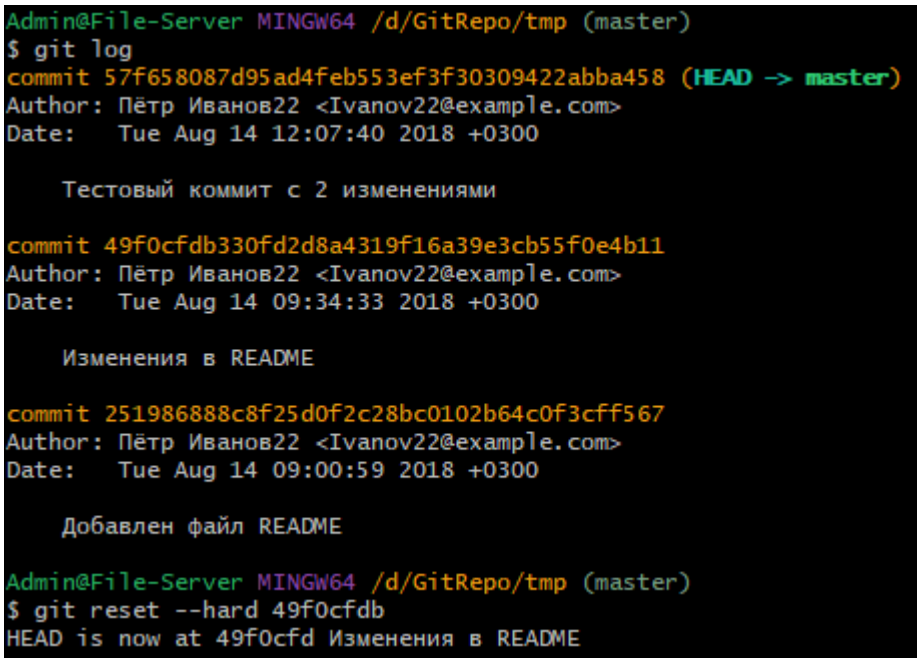
4. Команда gitreset --hard<commit>, в отличие от предыдущей, работает не по шагам, а разом. Рассмотрим её действие. Удалите и исправьте файлы, как в пункте 1.

Затем примените команду gitreset --hard и проверьте полученный статус:

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git reset --hard HEAD
HEAD is now at 57f6580 Тестовый коммит с 2 изменениями

Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Всё вернулось на места к состоянию последнего коммита.

5. Попробуем откатиться не на последний, а на предпоследний коммит. Для этого нужно указать его номер (несколько цифр, позволяющих идентифицировать его однозначно). Чтобы знать номера коммитов, сначала выведите лог. Затем выполните откат по номеру предпоследнего коммита:



Проверьте папку, убедитесь, что всё вернулось к состоянию предпоследнего коммита (второй файл исчез, а в первом только один восклицательный знак).

Примечание:

Если лог не помещается на экране, то программа будет выводить его по частям и после первой части ждать нажатия любой клавиши. Вверх-вниз его можно проматывать стрелками на клавиатуре. Для выхода из этого режима следует нажать «q».

Самостоятельная работа

1. Создайте репозиторий в пустой папке

2. Добавьте в папку файл. Изучите вывод команды git status. Проиндексируйте файл командой git add. Снова посмотрите вывод git status.

3. Зафиксируйте изменения командой git commit.

4. Сделайте и зафиксируйте следующие изменения (каждый подпункт - одна фиксация):

- Добавление сразу трех файлов
- Изменения в тексте двух файлов
- Изменения в тексте одного файла, удаление другого и добавление еще одного

5.* Удалите из какой-либо подпапки все файлы и зафиксируйте это изменение (подпапку, конечно же, надо предварительно создать). А затем удалите и саму пустую подпапку. Объясните результат.

6. Изучите вывод команды git log.Прочтите о ее различных ключах: https://git-scm.com/book/ru/v2/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D1%8B-Git-%D0%9F%D1%80%D0%BE%D1%81%D0%BC%D0%BE%D1%82%D1%80-%D0%B8%D1%81%D1%82%D0%BE%D1%80%D0%B8%D0%B8-%D0%BA%D0%BE%D0%BC%D0%BC%D0%B8%D1%82%D0%BE%D0%B2

7. * Создайте свой формат вывода git log с помощью ключа –pretty=format

8. * Внесите изменения в репозиторий, но не фиксируйте их. Посмотрите git status. Теперь дайте команду git stash и снова посмотрите статус и свои файлы. А теперь git stash apply. Попробуйте объяснить результат, не прибегая к документации.

В отчёт приложите все использованные вами команды и их вывод в консоль.

В отчёте опишите, как вы поняли принцип действия всех новых команд из самостоятельной работы.

Задания, помеченные знаком "*"требуют самостоятельного поиска дополнительных материалов.

Контрольные вопросы:

1. Как отобразить текущий статус репозитория?
2. Какие файлы называются Untracked files?
3. Как передать файл под контроль Git?
4. Как отобразить список изменений в репозитории?
5. Как отменить коммит и почему это делать не рекомендуется?
6. Как Git идентифицирует все этапы изменения?

Задание №3.3. Работа с удалёнными репозиториями

Цель задания:

Научиться работать с удаленными репозиториями. Загрузка веток из других источников. Операции с удаленными ветками

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Удалённые ветки — это ссылки на состояние веток в ваших удалённых репозиториях. Это локальные ветки, которые нельзя перемещать; они двигаются автоматически всякий раз, когда вы осуществляете связь по сети. Удалённые ветки действуют как закладки для напоминания о том, где ветки в удалённых репозиториях находились во время последнего подключения к ним.

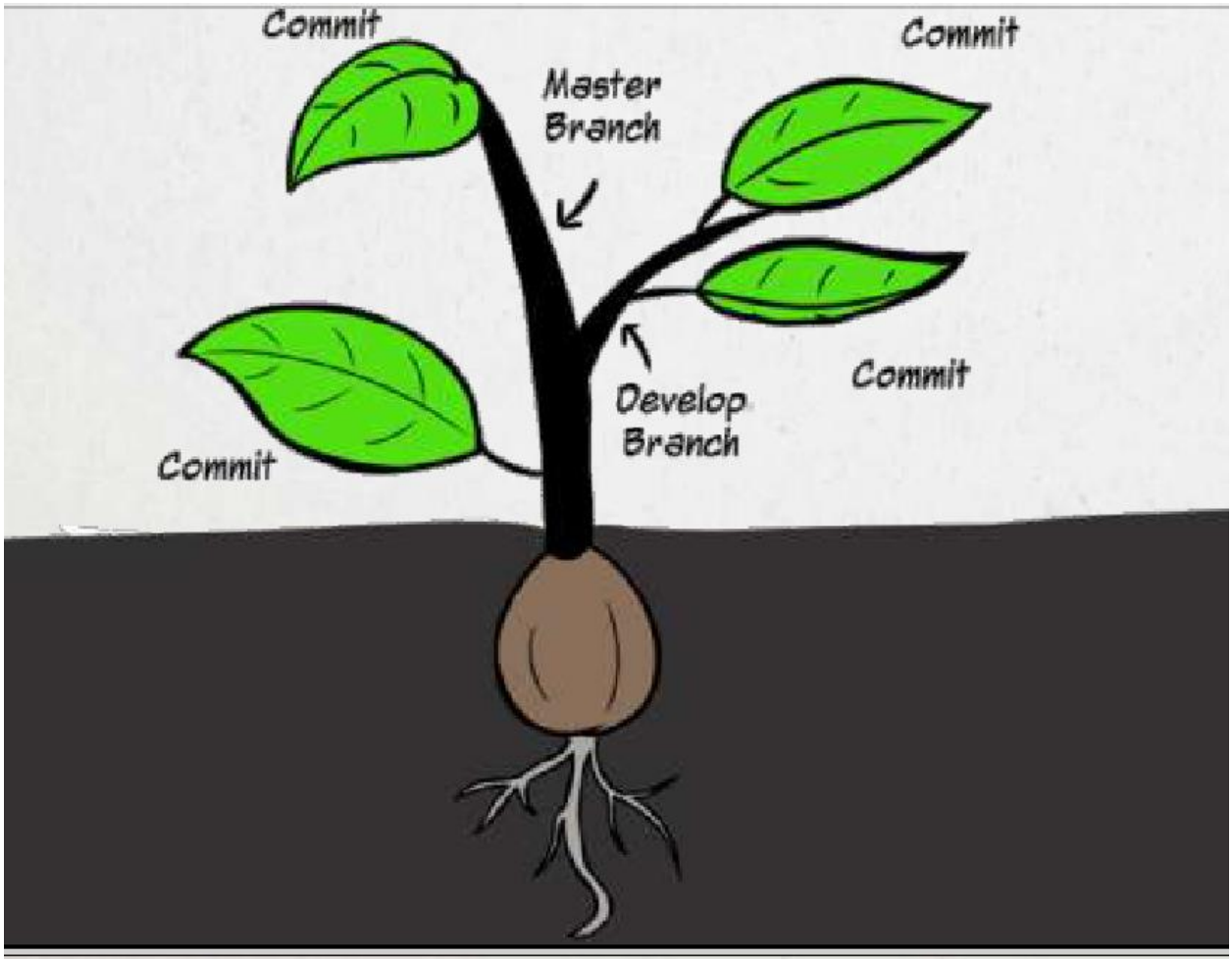
Они выглядят как (имя удал. репоз.)/(ветка). Например, если вы хотите посмотреть, как выглядела ветка master на сервере origin во время последнего соединения с ним, проверьте ветку origin/master. Если вы с партнёром работали над одной проблемой, и он выложил веткуiss53, у вас может быть своя локальная ветка iss53; но та ветка на сервере будет указывать на коммит в origin/iss53.

Практическая часть.

Когда на экран выводится git log, мы видим слово (master). Это – основная ветвь истории изменений. На данный момент она для нас единственная.

Ветка – это последовательность коммитов.

1. Дайте команду git status в своём репозитории. Будет выведено сообщение «On branch master» - «Я нахожусь на ветке master» - в главной ветви, которая начинается с самого первого коммита в истории и доходит до последнего.



В данной работе мы будем рассматривать только ветвь master.

Git – это распределенная система, это значит, что у нас может быть много репозиториев. Они могут быть распределены – находиться на разных компьютерах внутри локальной сети, в разных местах одного компьютера или где-то далеко в интернете. Распределенность Git заключается в том, что он умеет определенным образом связывать эти репозитории между собой.

В некоторых СКВ система репозиториев клиент-серверная, но не в Git. Здесь все репозитории равноправны между собой.

Репозиотрии делятся ан 2 типа – локальные (у нас на компьютере) и удалённые (где-то ещё). Git умеет связывать репозитории между собой. Цель этой лабораторной работы – изучить данную технологию, то, как эта связь выполняется.

Для работы с удаленными репозиториями используется команда git remote.

2. Создадим в папке GitRepo новый чистый репозиторий tmp2, а старый удалим. В Git Bash перейдем в новую папку (cd <путь>).

```
Admin@File-Server MINGW64 /d/GitRepo/tmp (master)
$ cd ..

Admin@File-Server MINGW64 /d/GitRepo
$ cd /d/GitRepo/tmp2

Admin@File-Server MINGW64 /d/GitRepo/tmp2
$
```

Инициализируйте репозиторий (git init).

3. Создайте аналогично второй репозиторий в корне диска D:

Добавьте в него файл README1.txt с каким-либо текстом, сделайте коммит. Затем в Git Bash сделайте текущим первый репозиторий.

```
Admin@File-Server MINGW64 /d/Repo2 (master)
$ git add README1.txt

Admin@File-Server MINGW64 /d/Repo2 (master)
$ git commit -m "Новый файл в Repo2"
[master (root-commit) 928ae36] Новый файл в Repo2
 1 file changed, 1 insertion(+)
 create mode 100644 README1.txt

Admin@File-Server MINGW64 /d/Repo2 (master)
$ cd /d/GitRepo/tmp2/
```

4. Команда *git remote add* добавляет к текущему репозиторию ссылку на удалённый (находящийся в другом месте) под каким-то именем, назовём его super2.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git remote add super2 "d:\Repo2"
```

5. Команда git remote покажет нам список прикрепленных к нашему репозиторию удалённых репозиториев.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git remote
super2
```

С ключом –v вывод будет немного подробнее:

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git remote -v
super2  d:\Repo2 (fetch)
super2  d:\Repo2 (push)
```

Показывается не только название, но ещё и адрес удалённого репозитория.

Связь создана, однако на статус это никак не повлияет.

Для просмотра удаленного репозитория можно использовать команду

git remote show super2

6. Далее выполним команду git fetch super2

Эта команда возьмет информацию обо всех изменениях в удалённом репозитории и перенесёт в текущий.

Обратите внимание, что будет перенесена только информация об изменениях, а не они сами!!! fetch не меняет файлы!!!

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git fetch super2
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From d:\Repo2
 * [new branch]      master     -> super2/master
```

Выведите содержимое репозитория командой ls. Вы увидите, что он не изменился. В нем по-прежнему ничего нет. Однако, теперь мы знаем всё об удалённом репозитории.

После того, как мы сделали git fetch, мы получили в том числе список веток удалённого репозитория. В данный момент у нас есть одна ветка – master. А в удаленном репозитории может быть много разных веток. Для того, чтобы работать с удаленными изменениями, нам нужно связать ветки между собой. В самом простом случае – связать две ветки master между собой.

7. Выполните команду

git checkout --track super2/master

Здесь сказано, что Git должен создать связь текущей ветки данного репозитория с ветвью master репозитория super2.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git checkout --track super2/master
Already on 'master'
Branch 'master' set up to track remote branch 'master' from 'super2'.
```

В ответ Git пишет, что ветка master установлена для отслеживания удалённой ветки master из super2.

8. Команда git pull – команда, которая получает изменения.

```
Admin@File-Server MINGW64 /d/GitRepo/tmp2 (master)
$ git pull
Already up to date.
```

Т.е. Эта команда позволяет выкачать все изменения из удаленного репозитория.

В следующей работе рассмотрим использование для этих целей популярного в настоящее время сервиса Github.

Контрольные вопросы:

1. Что такое ветка?
2. На какие типы делятся репозитории Git?
3. Как добавить ссылку на удаленный репозиторий?
4. Как вывести подробный список прикрепленных репозиториев?
5. Что выполняет команда *git remote show*?

Задание №3.4. Работа с метками.

Цель задания:

Научиться присваивать метки к репозиториям. Работа с сабмодулями

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Как и большинство СКВ, Git имеет возможность помечать (tag) определённые моменты в истории как важные. Как правило, этот функционал используется для отметки моментов выпуска версий (v1.0, и т.п.). В этом разделе вы узнаете, как посмотреть имеющиеся метки (tag), как создать новые. А также вы узнаете, что из себя представляют разные типы меток.

Просмотр имеющихся меток (tag) в Git'е делается просто. Достаточно набрать git tag.

Git использует два основных типа меток: легковесные и аннотированные. Легковесная метка — это что-то весьма похожее на ветку, которая не меняется — это просто указатель на определённый коммит. А вот аннотированные метки хранятся в базе данных Git'a как полноценные объекты. Они имеют контрольную сумму, содержат имя поставившего метку, e-mail и дату, имеют комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG). Обычно рекомендуется создавать аннотированные метки, чтобы иметь всю перечисленную информацию; но если вы хотите сделать временную метку или по какой-то причине не хотите сохранять остальную информацию, то для этого годятся и легковесные метки.

Практическая часть.

Теги

Теги – это метки, которыми можно пометить коммит. Они часто применяются в работе.

Есть лёгкие теги. Они очень простые. Мы берем команду gittagи придумываем какую-либо метку. Часто теги используют, чтобы помечать какие-либо версии. Например, показать, что на каком-то коммите мы

достигли версии 1.0. То есть, это просто ещё одно имя для коммита, для того, чтобы пометить какие-то важные в истории коммиты.

1. Введите команду gittag– это команда, которая выводит состояние меток в текущем репозитории. В данном случае меток нет, ничего не будет выведено.

2. Вызовите команду gitstatus, убедитесь, что вы находитесь в ветке master. Теперь пометим тегом 1.0 текущий коммит.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git status
On branch master
nothing to commit, working tree clean

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git tag 1.0
```

3. Повторите команду gittag. Вы увидите, что у вас появился тег.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git tag
1.0
```

4. Если вы хотите увидеть помеченный коммит, используйте команду gitshow.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git show 1.0
commit ecd8cd72b1ebaf1946bc89ac90c50223d3065971 (HEAD -> ma
re/1)
Merge: fe36a38 2374d7c
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Oct 2 14:31:58 2018 +0300

    Конфликт решен

diff --cc first.txt
index 5ace0b5,cdc9555..51bd26d
--- a/first.txt
+++ b/first.txt
@@@ -1,3 -1,3 +1,3 @@@
  Это первая строчка.

- Красная кнопка
 -Тут две зелёных кнопки.
++Одна красная и две зеленые кнопки.
```

Вы видите, что в этом коммите был решен конфликт – проблема с красной и зелеными кнопками.

5. Меток в проекте может быть множество, поэтому у команды git tag может быть метка -l – маска. Например:

git tag -l ‘1.*’

Эта маска покажет, какие метки начинаются с «1.»

6. Для удаления тега применяется команда gittag -d.Например:

git tag -d 1.0

Кроме лёгких тегов существуют **аннотированные метки**. Это тоже метка (имя для коммита), однако, кроме имени в ней можно поместить

некоторое сообщение, дату и время создания, имя того, кто создал, и даже подписать цифровой подписью.

Это достаточно сложная возможность, она используется в больших проектах, где много разработчиков сливают свои изменения, и есть один главный разработчик, которому поручено принимать решения о выпуске продукта. Они используются для того, чтобы пометить коммит готового к сборке продукта. Этот человек (релиз-мастер) с помощью своей цифровой подписи свидетельствует о том, что он даёт визу на публикацию. Он ставит метку на этот коммит и подписывает её. Автоматические системы сборки на это реагируют и начинают сборку продукта (эти технологии не слишком используются обычными разработчиками).

Выглядит эта команда так:

git teg -a 1.0 –m “Тест тега”

7. Создайте такой тег.

8. Используйте команду git tag 1.0. В списке тегов вы увидите этот тег в обычном виде.

9. Вызовите команду git show 1.0. Вы увидите намного больше информации об этом коммите, чем с обычным тегом, в том числе (сразу после команды) сведения о том, кто эту метку поставил (строчка Tagger), когда он её поставил (строчка Date). Если бы мы воспользовались при создании тега своим ключом, то тут бы была также информация о нём, и некоторый автоматизированный софт смог бы проверить её подлинность. Это используется в очень крупных проектах, например, при работе над Линуксом.

Метки можно создавать и позже. Достаточно при создании указать номер уже существующего коммита:

git tag -a 1.0 -m "Сообщение" 1baf

10. Выведите список коммитов.

11. Создайте тег 1.1 для созданного ранее коммита.

Метки по умолчанию не передаются в удаленный репозиторий. Для передачи используйте команды:

git push origin 1.0 – ДЛЯ КОНКРЕТНОЙ МЕТКИ

git push origin --tags - ДЛЯ ВСЕХ ТЕГОВ

Это происходит, так как git push отправляет изменения, а метка изменением не является.

СУБМОДУЛИ

СУБМОДУЛЬ – это, фактически, репозиторий внутри вашего репозитория.

git submodule add РЕПОЗИТОРИЙ ПАПКА

Для чего он может быть нужен? Например, вы работаете над проектом и нашли какую-то библиотеку, которую хотите к нему подключить (например, нашли её на github). Однако, её нельзя просто склонировать на ваш компьютер, перетащить файлы к себе и закоммитить. Однако, в этом случае, если автор внесёт какие-то коррективы в свою библиотеку, вы об этом никогда не узнаете.

Хорошим подходом в этом случае является субмодуль. Т.е. Можно просто включить в ваш репозиторий ссылку на другой.

1. Создайте на hithub под своим профилем второй репозиторий, добавьте в него файлы и включите его в ваш текущий проект. Для этого после того, как скопировали ссылку на репозиторий, перейдите в Git Bash в текущий репозиторий и выполните команду

git submodule add ____вставить адрес___ app

(app – название нового репозитория)

2. Откройте папку с репозиторием и убедитесь, что там появилась новая папка с субмодулем.
3. Проверьте git status. Вы видите, что добавился автоматически файл gitmodules, который описывает все подключенный модули, его можно прочитать (cat .gitmodules). Также добавилась папка app, которая в статусе отображается как одно целое, хотя внутри она содержит другие файлы и папки библиотеки.
4. Сделайте коммит и у вас зафиксируется информация о том, что в папке app находится клон удалённого репозитория.

После добавления субмодуля нужно зафиксировать изменения, обычной командой git commit

5. Сделайте коммит.

Есть ещё две необходимые команды для работы с субмодулями:

git submodule init Инициализирует субмодули в репозитории, где это еще не сделано (например сразу после клонирования)

git submodule update

• Загружает файлы субмодулей

Контрольные вопросы:

1. Что такое метка?
2. Какие типы меток существует?
3. Как добавляется метка к репозиторию?
4. Как вывести информацию по метке?

5. Для чего нужен сабмодуль?
6. Как добавить сабмодуль?

Задание №3.5. Работа с ветками

Цель задания:

Теперь, когда вы уже попробовали создавать, объединять и удалять ветки, пора познакомиться с некоторыми инструментами для управления ветками, которые вам пригодятся, когда вы начнёте использовать ветки постоянно.

Компетенции:

Код	Формулировка:
ОПК-4	Способен участвовать в разработке технической документации, связанной с профессиональной деятельностью с использованием стандартов, норм и правил

Теоретическая часть.

Команда git branch делает несколько больше, чем просто создаёт и удаляет ветки. При запуске без параметров, вы получите простой список имеющихся у вас веток:

```
$ git branch
  iss53
* master
  testing
```

Обратите внимание на символ *, стоящий перед веткой master: он указывает на ветку, на которой вы находитесь в настоящий момент (т.е. ветку, на которую указывает HEAD). Это означает, что если вы сейчас выполните коммит, ветка master переместится вперёд в соответствии с вашими последними изменениями. Чтобы посмотреть последний коммит на каждой из веток, выполните команду git branch -v:

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

Практическая часть.

Понятие «Ветки».

Репозиторий хранит в себе информацию о состояниях репозитория. Они называются «снапшоты». Каждое состояние – это коммит. Коммит – это ***снапшот и данные об авторстве, времени и номере коммита, а также указатель на родительский коммит***.Снапшот – это информация о том,

каким был репозиторий в какой-то момент времени. Это полная информация обо всех файлах и папках, снимок файловой системы. Он тщательно упакован и занимает минимально возможное место.

В отличие от других систем контроля версий, это не запись об изменениях от предыдущего состояния, а вся информация о репозитории. Это позволяет быстро перемещаться по коммитам, не внося изменения в несколько этапов, а просто перейдя на нужный коммит.

Это также позволяет в случае какой-то потери данных пропустить битый коммит и перейти к предыдущей рабочей версии.

Коммит снабжен указателями на его родительские коммиты:

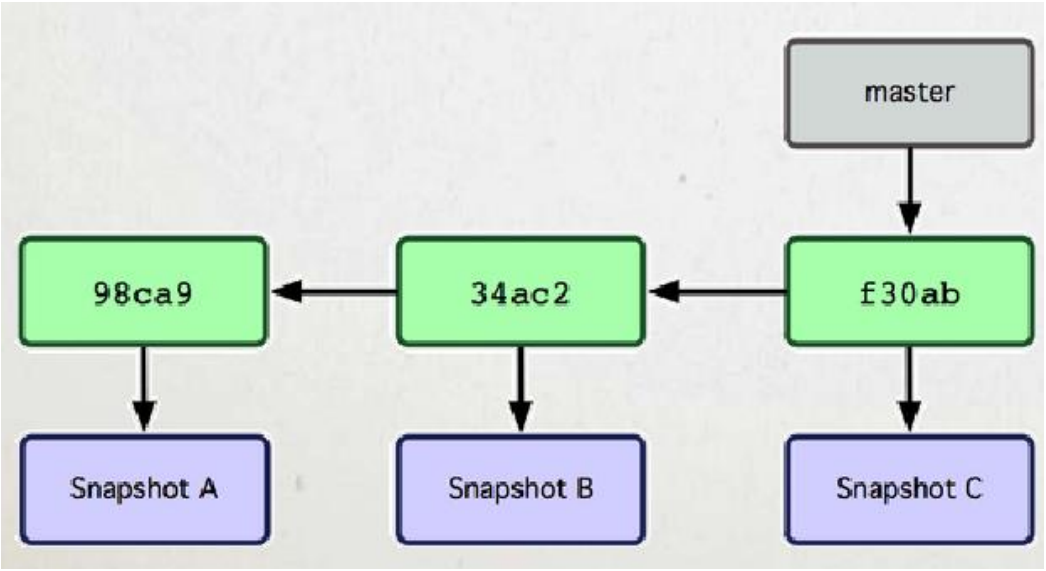
- Ноль указателей – если это первый коммит;
- Один – если это обычный;
- Несколько – в случае слияния изменений (об этом пойдет речь позже)

Таким образом, система всех этих указателей на родительские коммиты создает историю проекта, где в самом конце стоит самый первый коммит.

Ветка – это указатель на какой-либо коммит в истории.

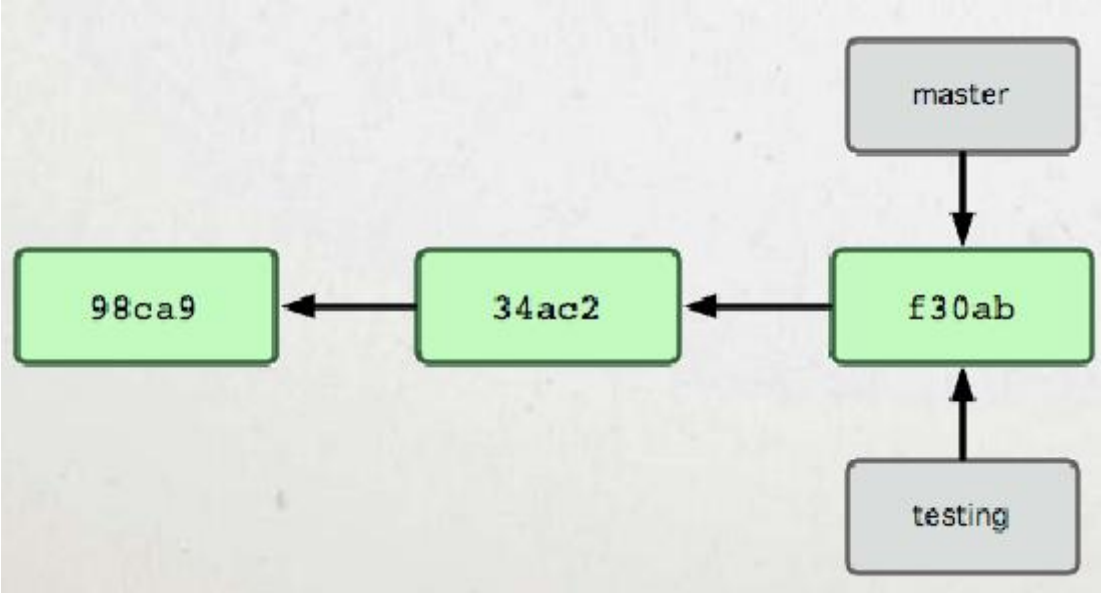
При создании репозитория автоматически создается ветка master. Указатель на нее автоматически сдвигается при каждом коммите. Ветка master считается основной, хотя это просто договорённость.

На рисунке представлено перемещение указателя «master». Сначала был создан снапшот А и указатель мастер был на нем, потом создали снапшотВ, указатель master переместился на него, затем перешел на станпшот С:



Репозитории в Git могут содержать неограниченное число новых веток. Для того, чтобы создать новую ветку, нужно дать команду gitbrunch<новая ветка>.

Например, команда gitbrunchtesting создаст новую ветку testing, то есть создаст новый указатель на текущее состояние репозитория и даст ему имя testing. Больше эта команда ничего не делает! В результате история будет выглядеть следующим образом:



Таким образом, на один и тот же коммит могут указывать сколько угодно веток.

Перейдем к практике.

1. Создайте на диске dпапку ProGit и в ней папку brunches:

```
Танюшка@NoteBook MINGW64 ~
 cd /d/

Танюшка@NoteBook MINGW64 /d
 mkdir ProGit

Танюшка@NoteBook MINGW64 /d
 mkdir /d/ProGit/branches/

Танюшка@NoteBook MINGW64 /d
 cd /d/ProGit/branches/

Танюшка@NoteBook MINGW64 /d/ProGit/branches
```

2. Инициализируйте репозиторий в этой папке

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git init
Initialized empty Git repository in D:/ProGit/branches/.git/
```

3. Проверьте статус репозитория, убедитесь, что он совершенно пуст. Мы находимся на ветке master (Onbranchmaster).

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

4. Создайте из-под Windows в этой папке какой-либо текстовый файл (например, README.txt), напишите в него текст «Проверка веток».

5. Добавьте этот файл и сохраните коммит под названием «Первый коммит».

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git add ./README.txt

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git commit -m "Первый коммит"
[master (root-commit) 05b133a] Первый коммит
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
```

6. Проверьте статус репозитория:

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git status
On branch master
nothing to commit, working tree clean
```

Мы по-прежнему находимся в ветви master.

7. Вызовите git log, и вы увидите, что существует только один «Первый коммит».

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit 05b133ad74ed028634f770e1dd94067aae6d9cbb (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

8. Теперь создаем новую ветку.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git branch testing
```

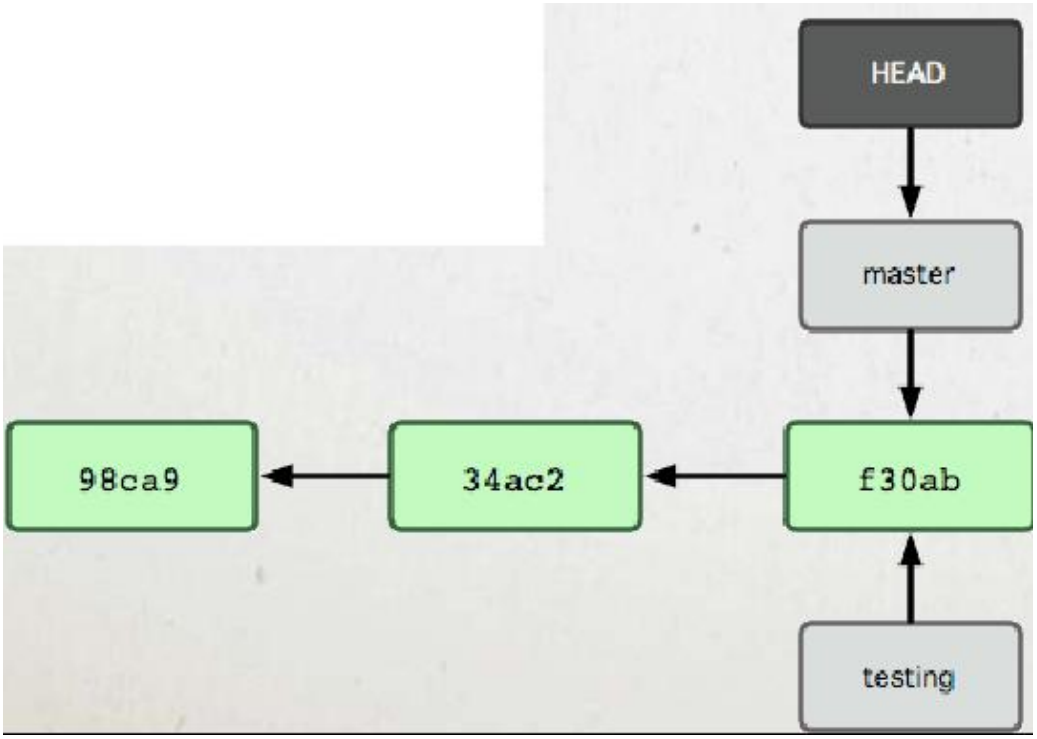
Мы создали новую ветку, которая указывает на тот же коммит, который на скриншоте виден под номером 05b133ad74…

9. Однако, мы до сих пор находимся в ветке master!!!Проверьте это командой git status.Это потому, что команда git branch ветку только создает.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git status
On branch master
nothing to commit, working tree clean
```

10. Для того, чтобы переключиться на новую ветку, используется команда git checkout. Для её понимания рассмотрим, что такое HEAD.

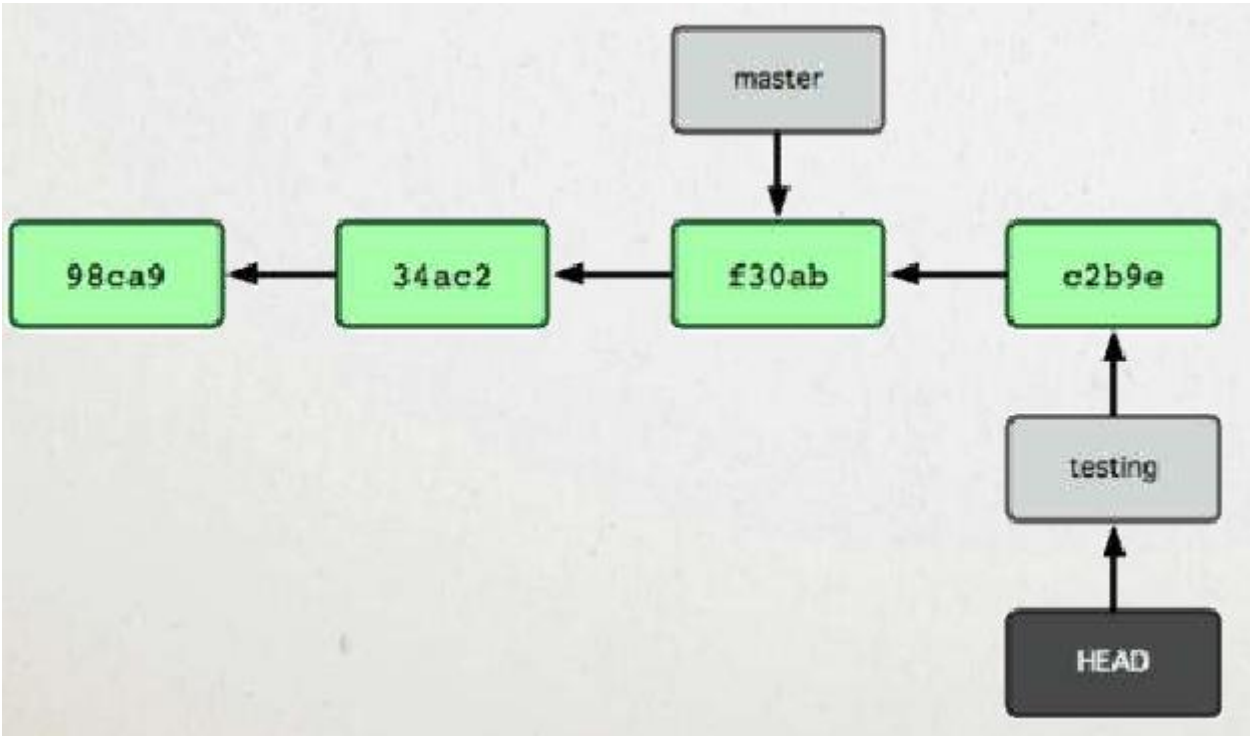
HEAD – это указатель на указатель. Это специальный указатель на текущую ветку и коммит. То есть, это указатель на текущее состояние репозитория (говорят, что это указатель на то, в какой ветке мы находимся. На самом деле мы в ней не находимся, просто текущий указатель указывает на нее). Команда git checkout передвинет HEADна указанную нами ветку.



11. Перейдите на ветку testing.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git checkout testing
Switched to branch 'testing'
```

В результате предыдущая схема будет выглядеть так:



12. В консоли вы видите текст: «Switched to branch ‘testing’», это означает, что мы переключились на ветку testing.

13. Также можно проверить текущую ветку с помощью gitstatus:

```
Танюшка@NoteBook MINGW64 /d/temp/branches
 git status
On branch testing
nothing to commit, working tree clean
```

Как видите, никаких изменений не произошло, мы просто переключились с одной ветки на другую.

14. Сделаем в ветке testing ещё один коммит. Для этого откройте файл README.txt и допишите в конце любое слово, например, «Тест». Добавьте и индексируйте изменения, сохранив коммит под именем «Коммит в ветке testing».

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git add ./README.txt

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git commit -m "Коммит в ветке testing"
[testing 450680d] Коммит в ветке testing
 1 file changed, 2 insertions(+), 1 deletion(-)
```

15. Проверьте git status, убедитесь, что все изменения зафиксированы. Выведите git log и вы увидите два коммита.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit 450680d12f80565cea4c641d8c49136e7ce3ddd5 (HEAD -> testing)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:39:30 2018 +0300

    Коммит в ветке testing

commit 05b133ad74ed028634f770e1dd94067aae6d9cbb (master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

16. Далее перейдите в ветку master.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git checkout master
Switched to branch 'master'
```

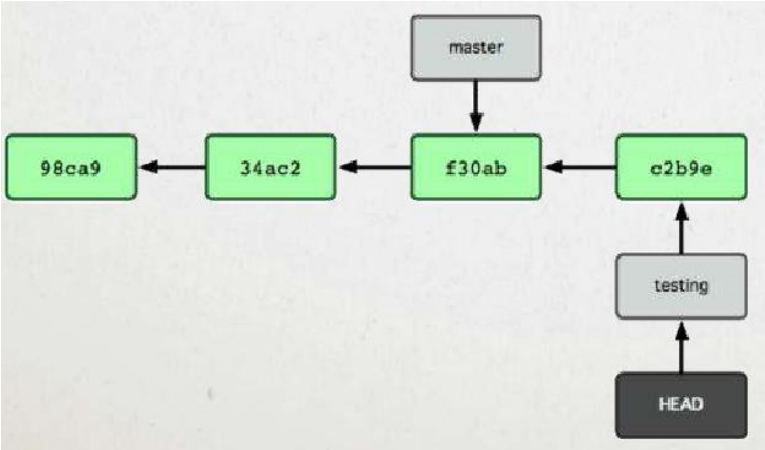
Мы вернули репозиторий в то состояние, с которым была связана ветка master. Чтобы убедиться в этом, откройте файл README.txt, и вы увидите, что дописанный текст «Тест», который был сохранен в ветке testing, исчез. Он существует только в ветке testing.

17. Выведите git log. Вы увидите, что коммит, относящийся к ветке testing, также не отображается.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit 05b133ad74ed028634f770e1dd94067aae6d9cbb (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

18. Перейдите опять на ветку testing. Проверьте лог, вы снова увидите два коммита. Откройте файл README.txt, там опять будет две строчки.

Итак, мы находимся в ситуации, представленной на рисунке:



Возникает вопрос, что произойдет, если мы перейдем на ветку master и создадим ещё один коммит? Получится, что у одного из коммитов несколько ветвей:

19. Промоделируем эту ситуацию в нашем репозитории. Перейдите на ветку master и выведите лог.

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git checkout master
Switched to branch 'master'

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit 05b133ad74ed028634f770e1dd94067aae6d9cbb (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

20. Внесем изменения в репозиторий. Добавьте файл index.txt, напишите в него текст «Главный файл».

21. Создайте коммит «Коммит в ветке master».

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git add ./index.txt

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git commit -m "Коммит в ветке master"
[master e915b88] Коммит в ветке master
 1 file changed, 1 insertion(+)
 create mode 100644 index.txt
```

22. Вызовите git log. Вы увидите в ветке master два коммита:

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit e915b88e7832956d5713b8c762b1e3cdc7c51929 (HEAD -> master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Fri Sep 21 12:20:54 2018 +0300

    Коммит в ветке master

commit 05b133ad74ed028634f770e1dd94067aae6d9cbb
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

23. Переключитесь в ветку testing и также посмотрите git log. Вы видите, что второй коммит совершенно другой:

```
Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git checkout testing
Switched to branch 'testing'

Танюшка@NoteBook MINGW64 /d/ProGit/branches
 git log
commit 450680d12f80565cea4c641d8c49136e7ce3ddd5 (HEAD -> testing)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:39:30 2018 +0300

    Коммит в ветке testing

commit 05b133ad74ed028634f770e1dd94067aae6d9cbb
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Tue Sep 18 14:07:56 2018 +0300

    Первый коммит
```

Таким образом мы получили разветвление в истории.

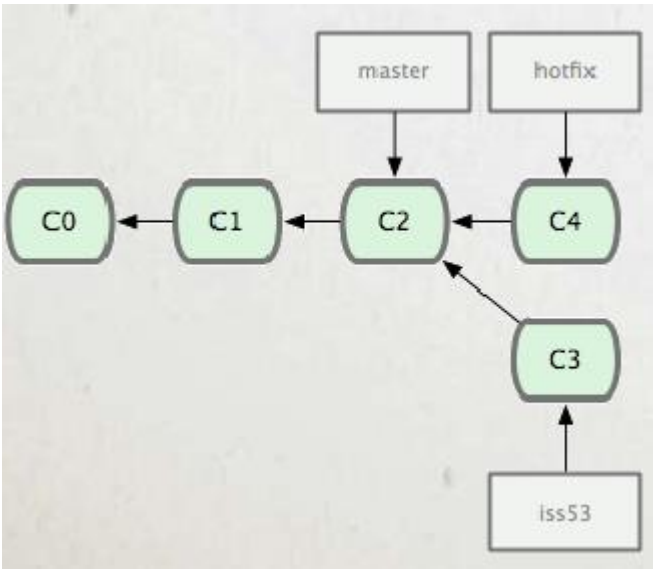
Важная рекомендация: переключаться между ветками следует в чистом состоянии репозитория! Не оставляйте непроиндексированных изменений! Иначе могут возникнуть проблемы: не произойдет переключения, пропадут незафиксированные данные и др.

Однако, если изменения пока не подлежат коммиту, стоит использовать git stash, переключиться, а по возвращении использовать git stash apply. Это стандартный метод работы с Git!!!

Слияние веток

Допустим, мы хотим объединить две ветки, слив изменения в одну.

Например, у нас в репозитории ситуация, представленная на рисунке:



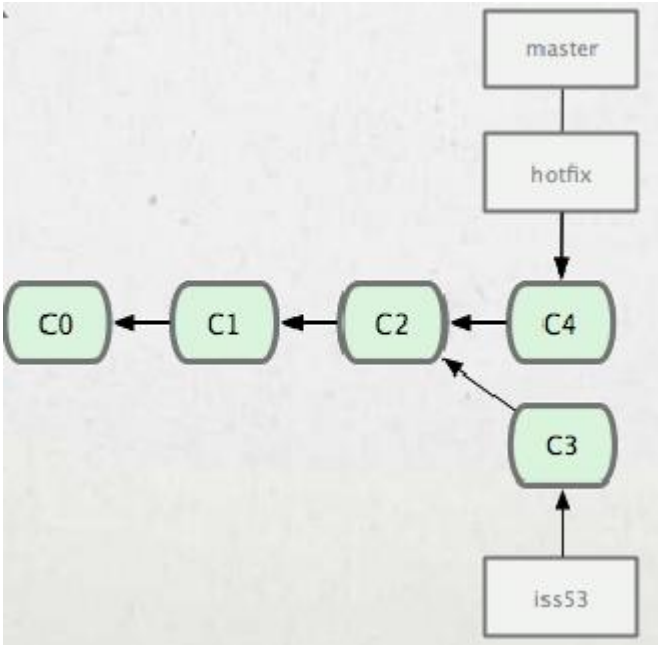
Здесь две ветки – master и hotfix находятся на одной линии истории (iss53 в данный момент не рассматриваем). То есть, в какой-то момент мы сделали ветку hotfix, внесли в нее какие-то изменения, закоммитили их, протестировали, и теперь готовы эти изменения влить в master.

Для этого нужно:

1) переключиться на ветку master (на ту ветку, которая будет принимать изменения);

2) дать команду git merge hotfix (merge – означает «слить»).

После этого Git рассуждает следующим образом: master и hotfix находятся на одной ветке истории. Для слияния их изменений достаточно историю просто перемотать вперёд. Что он и сделает. То есть, в данном случае никакого слияния фактически происходить не будет. Просто указатель мастер тоже переместится на C4:



Такой тип слияния называется fast-forward (ff). Такой сценарий реализуется только тогда, когда у нас ветки находятся на одной линии истории.

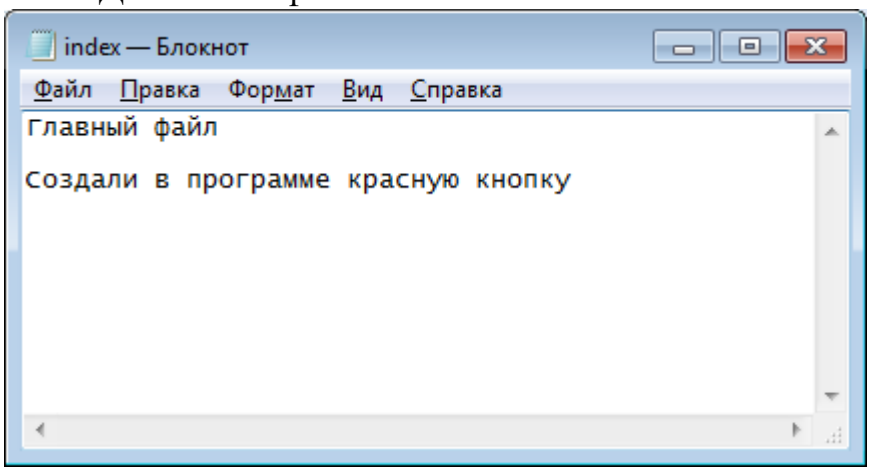
После этого ненужную ветку можно сразу удалить. Для этого используется команда git branch -d *hotfix* (вместо hotfix подставить название своей удаляемой ветки).

Рассмотрим пример.

1. Выведите в ветке master список коммитов, вы увидите 2 коммита.
2. Создадим от этой ветки ветку hotfix (например, нам надо срочно внести какое-то изменение в программу).

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git branch hotfix
```

3. Переключитесь на эту ветку.
4. Добавьте в файле index новый текст:



5. Проверьте git status, вы увидите, что появились изменения.
6. Сделайте коммит «Создали красную кнопку».
7. Проверьте лог, вы увидите все три коммита. При этом третий коммит относится только к ветке hotfix:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (hotfix)
$ git log
commit c7863fb49abecb7270696496008e8c0d00eae877 (HEAD -> hotfix)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:53:16 2018 +0300

    Создали красную кнопку

commit b2052d1d9887cb4ebacf27c92f56f54603bb9d47 (master)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:44:23 2018 +0300

    Коммит в ветке master

commit 912951b6ccfdf281502c3fafc1bed067619af5f9
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:39:03 2018 +0300

    Первый коммит
```

Допустим, мы внесли таким образом нужные изменения в нашу программу, их в ветке hotfix изучили инженеры по качеству, тестировщики и

так далее, после чего, когда эта версия принята, вам как программисту поступает команда влить эти изменения в основную ветку.

8. Переключитесь на ветку master и выведите лог. Вы увидите два коммита ветки master.

9. Выполните слияние:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git merge hotfix
Updating b2052d1..c7863fb
Fast-forward
 index.txt | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)
```

Вы видите, что в 4й строке написано Fast-forward. Это означает, что в результате команды произошла перемотка этой истории, и в её ходе был изменен файл index.txt (5я строка).

10. Проверьте статус. Мы в ветке master, коммиты не требуются.

11. Выведите лог. Вы увидите, что все коммиты появились в ветке master.

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git log
commit c7863fb49abecb7270696496008e8c0d00eae877 (HEAD -> master, hotfix)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:53:16 2018 +0300

    Создали красную кнопку

commit b2052d1d9887cb4ebacf27c92f56f54603bb9d47
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:44:23 2018 +0300

    Коммит в ветке master

commit 912951b6ccfdf281502c3fafc1bed067619af5f9
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:39:03 2018 +0300

    Первый коммит
```

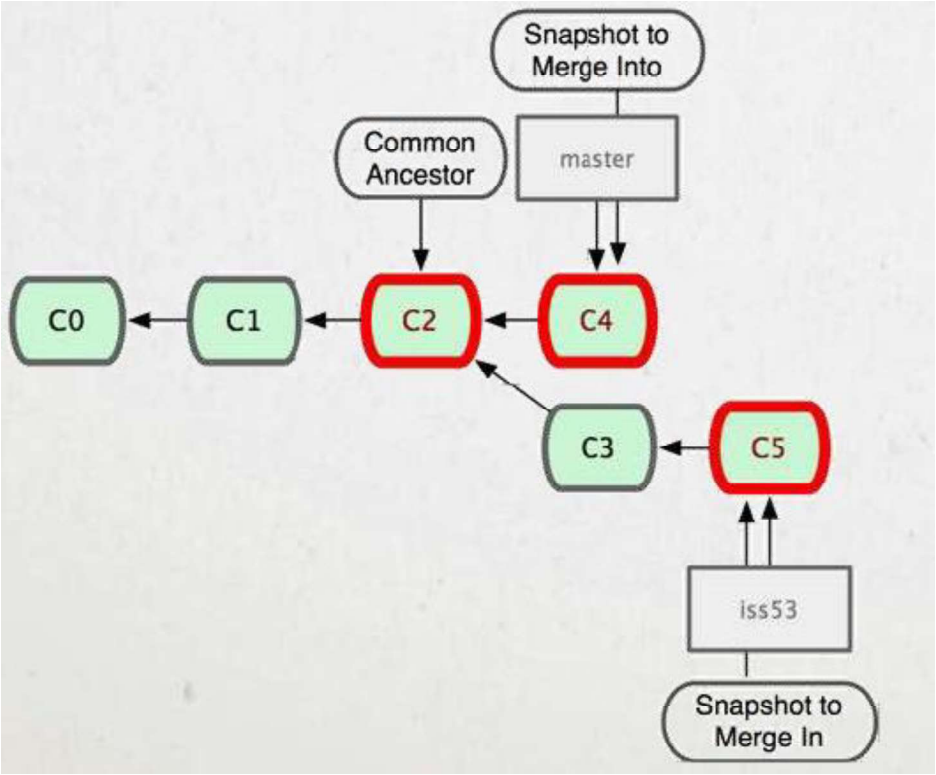
Таким образом, мы рассмотрели самый простой вид слияния – Fast forward.

12. Вызовите команду git branch –v

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git branch -v
  hotfix  c7863fb Создали красную кнопку
* master  c7863fb Создали красную кнопку
  testing 4d6079e Коммит в ветке testing
```

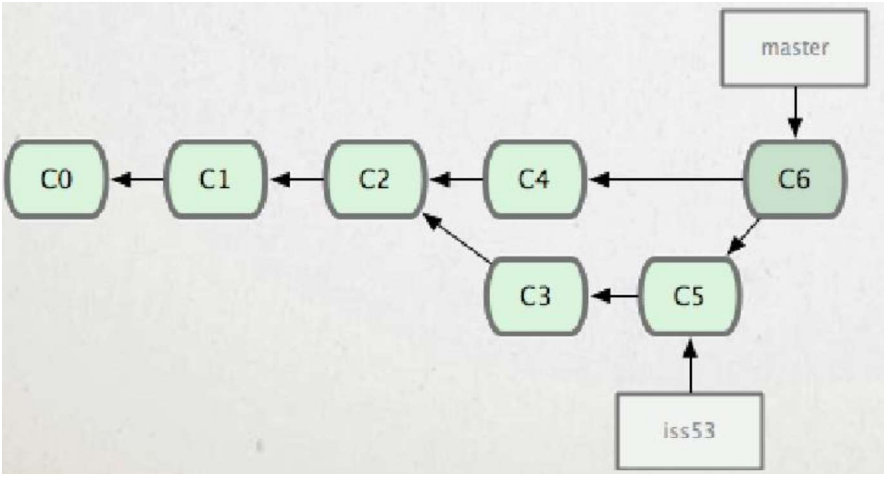
Этак команда выводит полный перечень веток репозитория. Звездочкой отмечена текущая ветка, на которую смотрит HEAD.

Теперь рассмотрим более сложный случай: как сливать изменения, если ветки уже разошлись? Допустим, что ветка master уже ушла вперёд от общего родителя (C4). А вторая ветка Iss53 уже ушла от неё на два коммита (C5). То есть, эти ветки не находятся на одной линии истории.



В этом случае git применяет алгоритм «Наилучшего общего предка». От обеих веток git просматривает историю назад, пока не найдет ближайшего общего предка. После этого он начинает последовательно к этому предку применять коммиты, ориентируясь на время, когда они были созданы, и таким образом создает новый коммит слияния.

Таким образом, в отличие от Fast forward, когда ничего нового не создавалось, здесь создастся новое изменение. Создастся новый коммит C6 в ветке master, который соберет в себе все изменения C3, C4 и C5:



Рассмотрим практический пример.

13. Проверьте статус, вы должны находиться в ветке master:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git status
On branch master
nothing to commit, working tree clean
```

14. Она уже ушла достаточно далеко от ветки testing, созданной ранее. Убедимся в этом, выведем лог для ветки master, затем перейдем в ветку testing и проверим лог для неё.

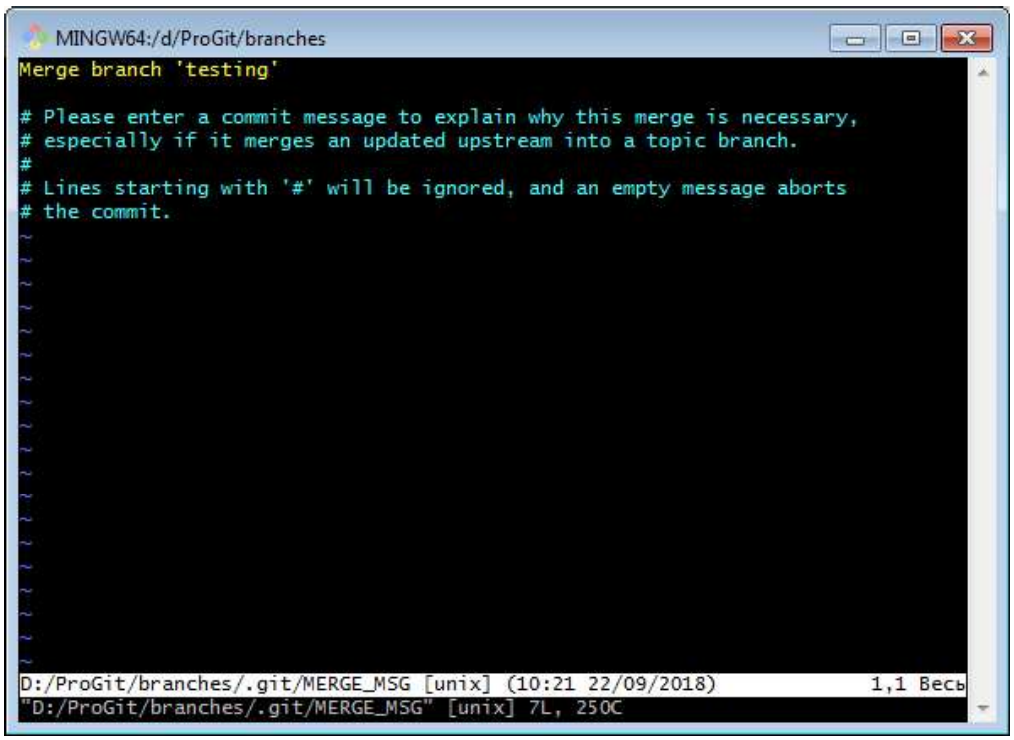
```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git log
commit c7863fb49abecb7270696496008e8c0d00eae877 (HEAD -> master, hotfix)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:53:16 2018 +0300

    Создали красную кнопку

commit b2052d1d9887cb4ebacf27c92f56f54603bb9d47
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:44:23 2018 +0300

    Коммит в ветке master

commit 912951b6ccfdf281502c3fafc1bed067619af5f9
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:39:03 2018 +0300

    Первый коммит
```

Переходим в testing:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git checkout testing
Switched to branch 'testing'

d103@S-T000012680 MINGW64 /d/ProGit/branches (testing)
$ git log
commit 4d6079ec915327ceb8f4f719587e0d0ffede2ebe (HEAD -> testing)
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:41:53 2018 +0300

    Коммит в ветке testing

commit 912951b6ccfdf281502c3fafc1bed067619af5f9
Author: Пётр Иванов22 <Ivanov22@example.com>
Date:   Sat Sep 22 08:39:03 2018 +0300

    Первый коммит
```

У них общий только первый коммит, в остальном они разошлись, у каждого своя история.

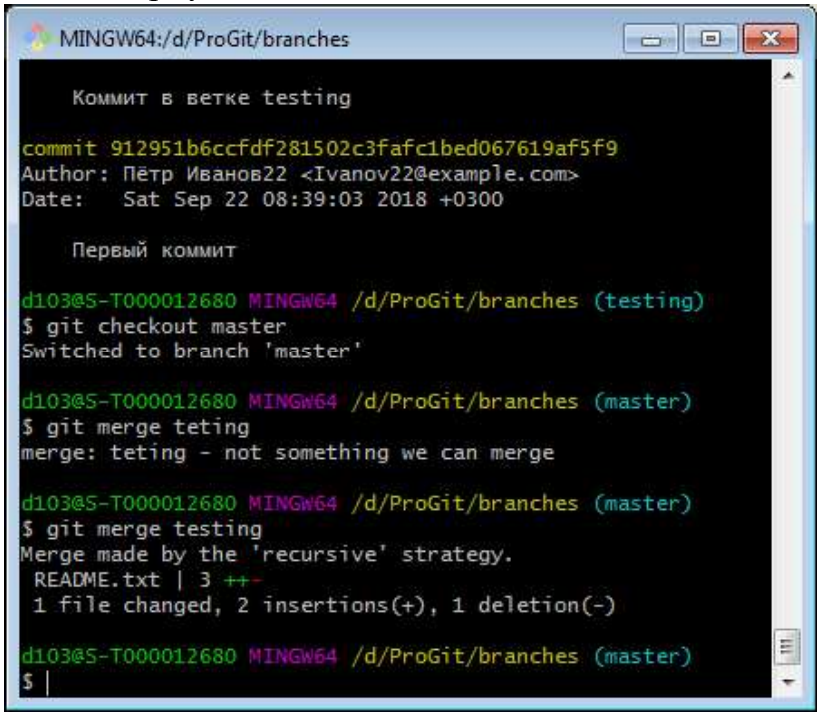
15. Снова переключаемся в master и вливаем неё ветку testing (git merge testing). Начнется слияние:



16. Слияние удалось и нам открылся текстовый редактор, который просит нас ввести сообщение для коммита. Чтобы выйти из него, нажмите последовательно <esc> <:> <q> (если в качестве текстового редактора по умолчанию у вас установлен другой, то вы можете просто закрыть его и Git Bash вернется в рабочий режим).

Также, если надо записать изменения в этом текстовом редакторе, нужно нажать <esc> <:> <w> (сейчас этого не делайте).

17. Вернулись в Git Bash



Слияние было осуществлено с использованием рекурсивной стратегии.

18. Выведите лог. Вы видите перечень того, какие коммиты применялись, последним создан коммит «Merge branch ‘testing’». Это название и можно было поменять в текстовом редакторе, но мы не стали этого делать.

19. Нажмите q для выхода в Git Bash.

Таким образом, Git при слиянии создает новый коммит и этим коммитом отмечает момент, когда это слияние было произведено. Это удобно для того, чтобы в случае необходимости это слияние было легко отменено, и вы могли сдвинуться на момент до него.

Итак, **в Git создание и слияние веток – это основная штатная операция. Если вы хотите внести какие-то изменения в проект, следует создать ветку, внести в ней изменения, протестировать, а затем, если всё нормально – влить в основную, если нет, удалить. Это стандартный метод работы с Git.**

Главный принцип: одна задача – это одна ветка!

Решение конфликтов

Конфликт возникает, если в одном и том же месте (файле) есть разные изменения. Это также штатная ситуация, Git предназначен для разрешения конфликтов.

Конфликт означает, что Git не смог слить версии автоматически и вам необходимо это сделать вручную.

Рассмотрим, что делать в случае возникновения конфликта.

1. Создадим конфликт искусственно. Проверьте, что вы находитесь в ветке master.

2. Создайте в папке branches текстовый файл first.txt и напишите внутри «Это первая строчка.».

3. Сделайте коммит.

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git add first.txt

d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git commit -m "Создали файл"
[master d081c15] Создали файл
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 first.txt
```

4. Сделайте ветку «feature/1» (Задача 1) и переключитесь на нее:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (master)
$ git checkout -b feature/1
Switched to a new branch 'feature/1'
```

5. Пусть наша задача №1 – сделать красную кнопку. Откройте файл first.txt и напишите в конце «Красная кнопка». Сохраните файл. Проиндексируйте и закоммитьте изменения в текущей ветке feature/1.

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/1)
$ git add first.txt

d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/1)
$ git commit -m "Создали красную кнопку"
[feature/1 d71c1cc] Создали красную кнопку
 1 file changed, 2 insertions(+)
```

6. Снова переключитесь на ветку master.
7. Создайте ветку под названием feature/2 и перейдите на неё.
8. Вторая задача – сделать две зелёных кнопки. Откройте файл first.txt, там в данной ветке должна быть только одна строка. Напишите во второй строке «Тут две зелёных кнопки».
9. Создайте коммит:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/2)
$ git add first.txt

d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/2)
$ git commit -m "Создали две зеленых кнопки"
[feature/2 f3fc0e9] Создали две зеленых кнопки
 1 file changed, 2 insertions(+)
```

Итак, у нас есть две ветки, у которых разошлась история.

10. Переключитесь на первую задачу:

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/2)
$ git checkout feature/1
Switched to branch 'feature/1'
```

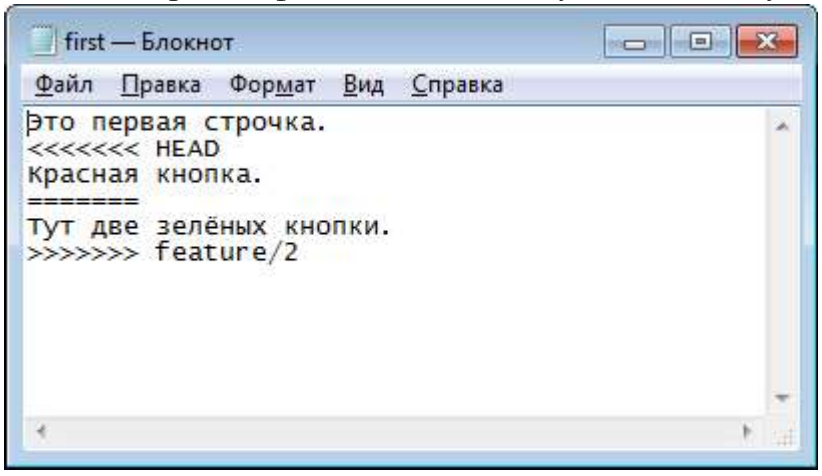
11. Попробуем в эту ветку влить изменения из второй.

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/1)
$ git merge feature/2
Auto-merging first.txt
CONFLICT (content): Merge conflict in first.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Мы видим надпись CONFLICT – в файле first.txt возник merge-конфликт. Автоматическое слияние невозможно. Нужно поправить конфликты и закоммитить результат.

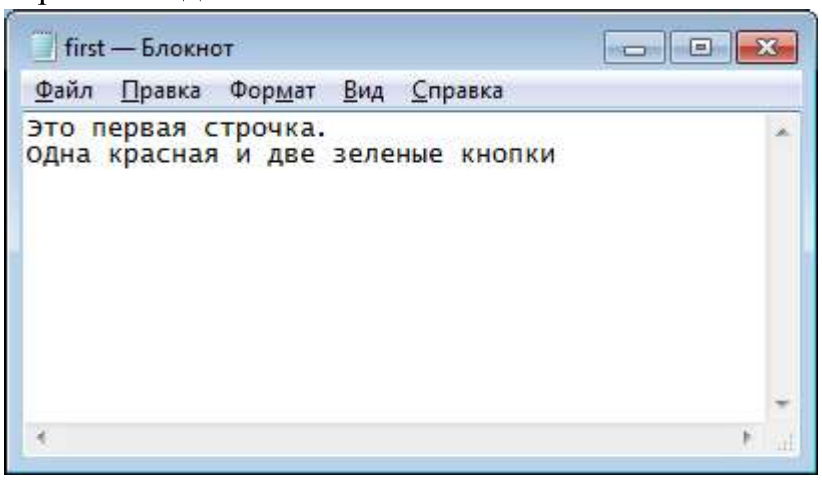
То есть, Git попытался применить последовательную стратегию слияния, обнаружил противоречия и вывел такое сообщение.

12. Откройте файл first.txt и вы увидите следующую картину:



13. В файле появились метки, которые указывают, что же вызвало конфликт, что именно вызывает противоречие, а именно: Красная кнопка - Две зеленые кнопки.

14. Git хочет, чтобы мы оставили что-то одно, требует выбрать некий результат вручную (или оба). Сотрите всю служебную секцию и напишите «Одна красная и две зеленые кнопки»:



Закройте и сохраните файл.

15. Проверьте статус репозитория.

```
d103@S-T000012680 MINGW64 /d/ProGit/branches (feature/1|MERGING)
$ git status
On branch feature/1
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   first.txt

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

16. Проиндексируйте файл и закоммитьте его под именем «Конфликт решён».

ДОМАШНЯЯ РАБОТА

1. Начните новый репозиторий на Гитхабе

2. Пользуясь своими знаниями HTML и CSS, создайте в нем файл index.html Пусть в этом файле будет информация о вас, профессиональный профиль. (достаточно пары абзацев, суть ДЗ не в том, чтобы красиво сверстать страницу). Зафиксируйте изменения.

3. Представьте теперь, что руководство ставит задачу: разместить на странице кнопку "Подписаться на новости"

- Создайте новую ветку для выполнения этой задачи

- Добавьте кнопку. Зафиксируйте изменения. Передайте их на гитхаб и убедитесь, что вы видите новую ветку в интерфейсе Гитхаба.

- Влейте изменения из ветки задачи в ветку master - вы должны увидеть merge методом fast-forward

- Передайте все изменения в удаленный репозиторий

4. Поступили сразу две новых задачи: изменить цвет кнопки из пункта 3. и добавить на страницу форму входа через социальные сети (условно, конечно!)

- Создайте для каждой задачи свои ветки
- Выполните их
- Влейте изменения в master
- Передайте изменения в Github

5. Смоделируйте возникновение конфликта, внося изменения в одно и то же место своего файла. Решите конфликт.

Контрольные вопросы:

1. Какая ветка создается сразу после создания репозитория?
2. Как задать новую ветку?
3. Что такое head и какая взаимосвязь между git checkout?
4. Как произвести слияние веток?
5. Что значит fast-forward?