Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Кафедра инженерной психологии и эргономики

**Пользовательские интерфейсы информационных систем**

Отчет по практическим занятиям на тему

«Образовательный курс GitHowTo»

|  |  |
| --- | --- |
| Выполнила: | Проверил: |
| cтудентка гр. 210901  Глушко А. Ю. | Давыдович К. И. |
|  |  |

Минск 2024

**Часть 1. Основы git**

**Финальные приготовления**

Репозиторий Git — это хранилище, в котором расположен ваш проект и его история. Это может быть локальное хранилище где-то на вашем компьютере или удаленное хранилище на сервисе типа GitHub или другом хостинге в Интернете. Репозиторий служит для отслеживания изменений в проекте, координации работы между несколькими людьми и отслеживания истории проекта.

Вы можете думать о коммите как о снимке вашего проекта в определенный момент времени. Правда, коммит содержит только информацию об изменениях, которые были внесены в репозиторий с момента последнего коммита. Он не содержит все файлы репозитория (если только это не первый коммит). Таким образом, каждый коммит — это небольшой кусочек истории репозитория, основанный на предыдущем коммите. Все они связаны между собой в цепочку, формируя историю изменений вашего проекта.

Ветка — это параллельная версия репозитория. Ветки позволяют вам работать над отдельными функциями вашего проекта, не влияя на основную версию. Закончив работу над новой фичей, вы можете объединить эту ветку с основной версией проекта.

Если вы никогда ранее не использовали Git, для начала вам необходимо осуществить установку. Выполните следующие команды, чтобы Git узнал ваше имя и электронную почту. Эти данные используются для подписи изменений сделанных вами, что позволит отслеживать, кто и когда сделал изменения в файле.

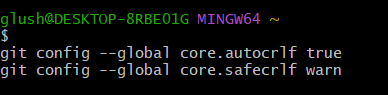




Мы будем использовать main в качестве имени ветки по умолчанию. Чтобы установить его, выполните следующую команду:

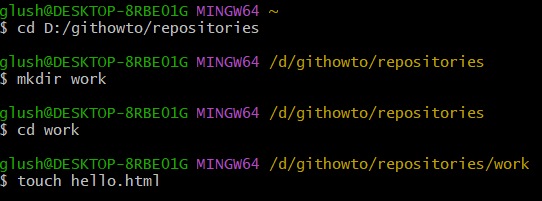


Корректная обработка окончаний строк:

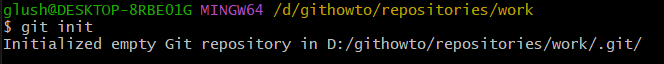


**Создание проекта**

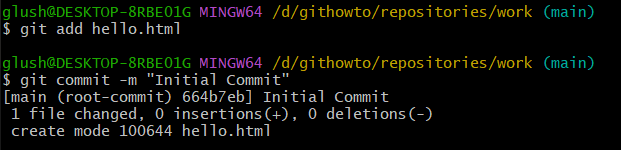
Начните работу в пустой директории (например, respositories, если вы скачали архив с предыдущего шага) с создания пустой поддиректории work, затем войдите в неё и создайте там файл hello.html с таким содержанием:



Теперь у вас есть директория с одним файлом. Чтобы создать Git-репозиторий из этой директории, выполните команду git init.

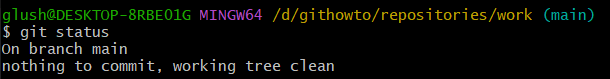


Теперь давайте добавим в репозиторий страницу «Hello, World».



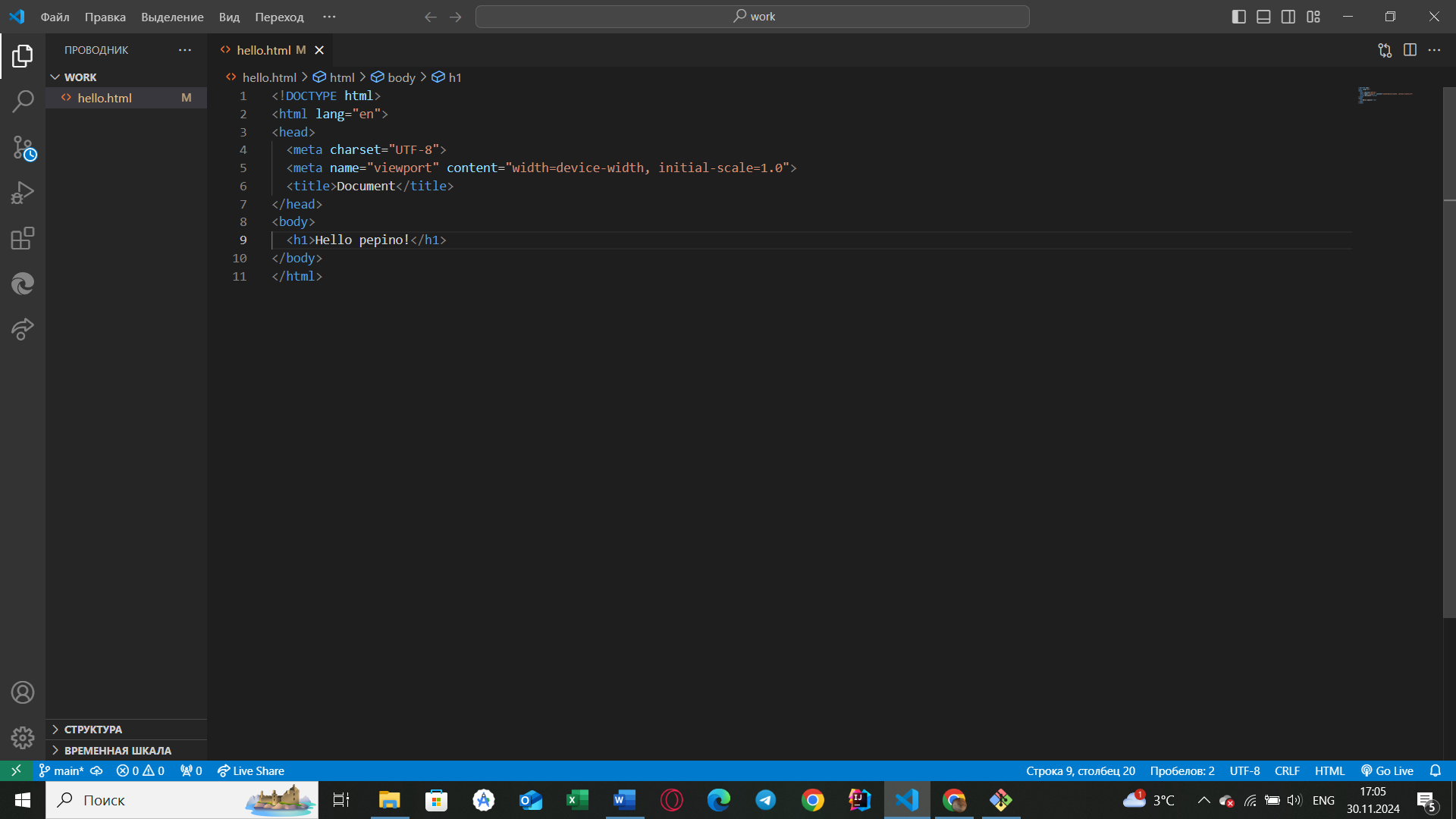
**Проверка состояния**

Используйте команду git status, чтобы проверить текущее состояние репозитория.

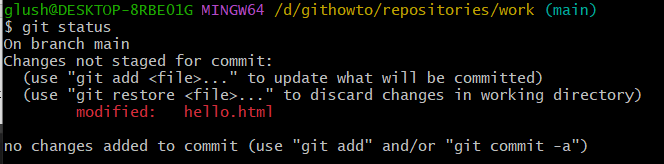


**Внесение изменений**

Добавим кое-какие HTML-теги к нашему приветствию. Измените содержимое файла на:



Теперь проверьте состояние рабочей директории.

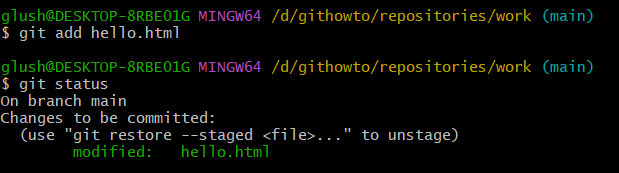


Первое, что нужно заметить, это то, что Git знает, что файл hello.html был изменен, но при этом эти изменения еще не зафиксированы в репозитории.

Также обратите внимание на то, что сообщение о состоянии дает вам подсказку о том, что нужно делать дальше. Если вы хотите добавить эти изменения в репозиторий, используйте команду git add. В противном случае используйте команду git сheckout для отмены изменений.

**Индексация изменений**

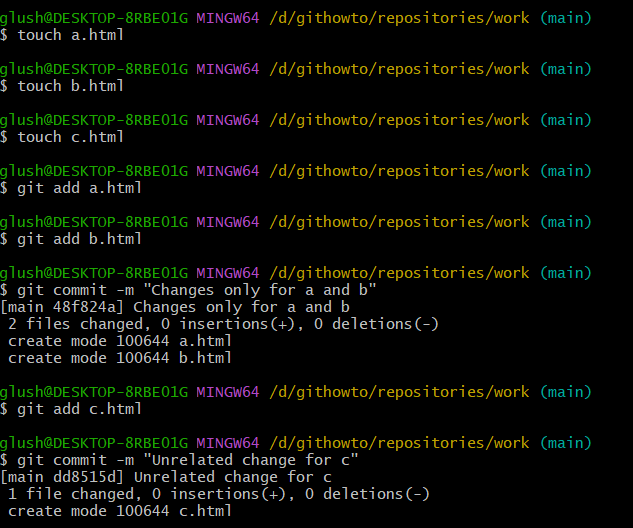
Теперь дайте команду Git проиндексировать изменения. Проверьте состояние:



Изменения файла hello.html были проиндексированы. Это означает, что Git теперь знает об изменении, но изменение пока не *перманентно* (читай, *навсегда*) записано в репозиторий. Следующий коммит будет включать в себя проиндексированные изменения.

Если вы решили, что *не* хотите коммитить изменения, команда состояния напомнит вам о том, что с помощью команды git reset можно снять индексацию этих изменений.

**Раздельная индексация**

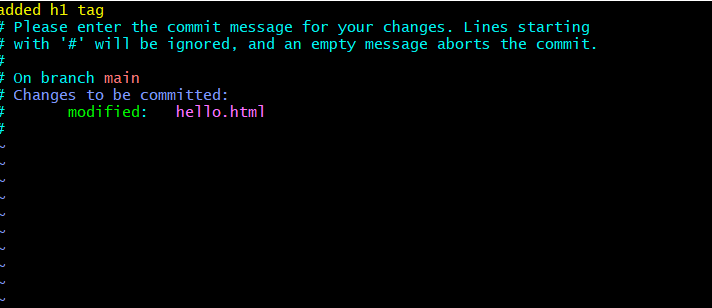


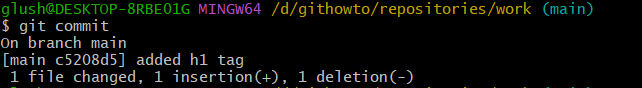
**Коммит изменений**

Когда вы ранее использовали git commit для коммита первоначальной версии файла hello.html в репозиторий, вы включили метку -m, которая делает комментарий в командной строке. Команда commit позволит вам интерактивно редактировать комментарии для коммита. Теперь давайте это проверим.

Если вы опустите метку -m из командной строки, Git перенесет вас в редактор по вашему выбору. Редактор выбирается из следующего списка (в порядке приоритета):

* переменная среды GIT\_EDITOR
* параметр конфигурации core.editor
* переменная среды VISUAL
* переменная среды EDITOR

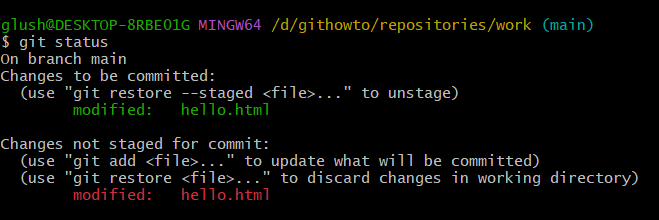




**Изменения, а не файлы**

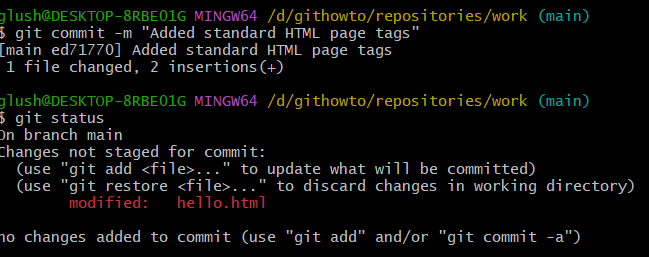
Большинство систем контроля версий работает с файлами: вы добавляете файл в систему, и она отслеживает изменения файла с этого момента.

Git фокусируется на изменениях в файле, а не самом файле. Когда вы осуществляете команду git add file, вы не говорите Git добавить файл в репозиторий. Скорее вы говорите, что Git надо отметить текущее состояние файла, коммит которого будет произведен позже.

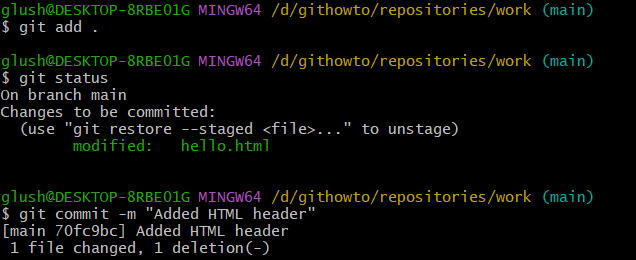


Обратите внимание на то, что hello.html указан дважды в состоянии. Первое изменение (добавление стандартных тегов) проиндексировано и готово к коммиту. Второе изменение (добавление заголовков HTML) является непроиндексированным. Если бы вы делали коммит сейчас, заголовки не были бы сохранены в репозиторий.

Произведите коммит проиндексированного изменения (значение по умолчанию), а затем еще раз проверьте состояние.

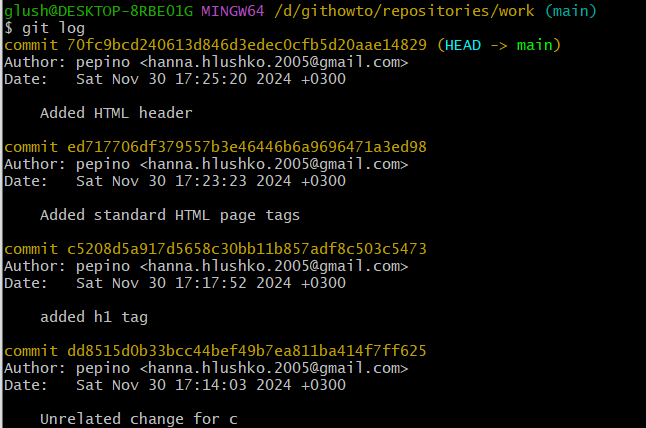


Теперь добавьте второе изменение в индекс, а затем проверьте состояние с помощью команды git status. Мы использовали текущую директорию (.) в качестве аргумента для добавления. Это самый короткий и удобный способ добавления всех изменений в текущей директории. Но поскольку Git добавляет в индекс *всё*, то *не лишним* будет проверить состояние репозитория перед запуском add, просто чтобы убедиться, что вы не добавили какой-то файл, который не следовало бы добавлять.



**История**

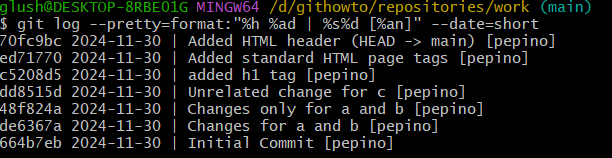
Получение списка произведенных изменений — функция команды git log.



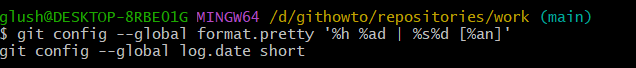
Со временем, я решил, что для большей части моей работы мне подходит следующий формат лога.

Давайте рассмотрим его в деталях:

* --pretty="..." — определяет формат вывода.
* %h — укороченный хеш коммита.
* %ad — дата коммита.
* | — просто визуальный разделитель.
* %s — комментарий.
* %d — дополнения коммита («головы» веток или теги).
* %an — имя автора.
* --date=short — сохраняет формат даты коротким и симпатичным.



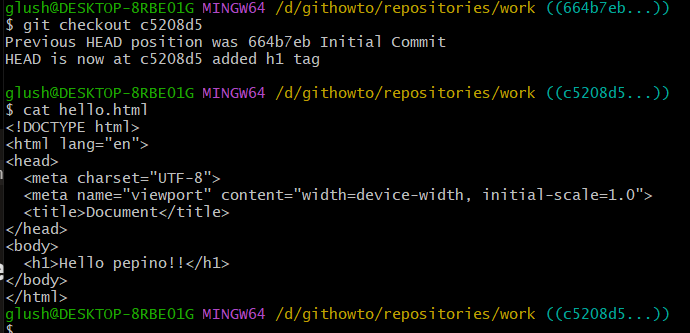
Таким образом, каждый раз, когда вы захотите посмотреть лог, вам придется много печатать. К счастью, существует несколько опций конфигурации Git, позволяющих настроить формат вывода истории по умолчанию:



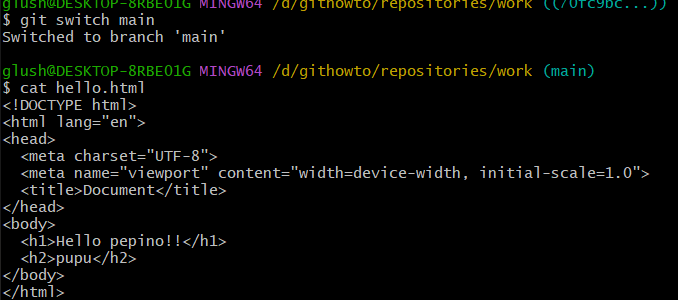
**Получение старых версий**

Git позволяет очень просто путешествовать во времени, по крайней мере, для вашего проекта. Команда checkout обновит вашу рабочую директорию до любого предыдущего коммита.

Просмотрите историю изменений и найдите хеш первого коммита. Он должен быть в последней строке результата git log. Используйте этот хеш (достаточно первых 7 символов) в команде ниже. Затем проверьте содержимое файла hello.html.

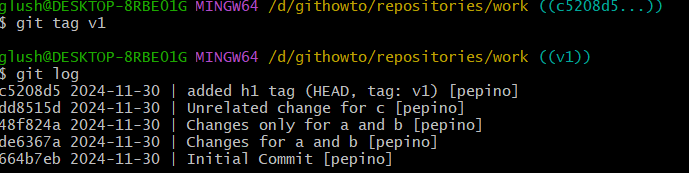


Команда checkout в течение длительного времени была своеобразным швейцарским ножом в мире Git. Она имеет множество различных опций, которые позволяют выполнять совершенно разные вещи: переключать ветки, сбрасывать код и так далее. В какой-то момент команда Git решила разделить ее на несколько команд. Команда switch является одной из них — ее единственным назначением является переключение между ветками. Команда checkout все еще доступна, но использовать ее для переключения веток больше не рекомендуется.

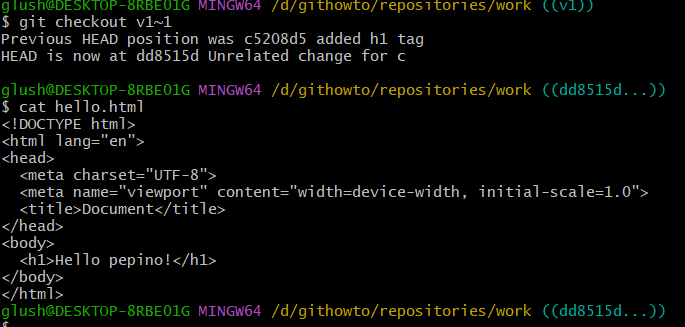


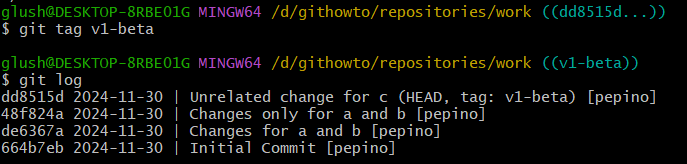
Думаю, вы согласитесь, что работать с хешами коммитов напрямую просто неудобно. Разве не было бы здорово, если бы вы могли обозначать конкретные коммиты понятными для человека названиями? Таким образом, вы могли бы четко видеть важные вехи в истории проекта. Кроме того, вы могли бы легко перейти к определенной версии проекта по ее названию. Именно для этого в Git придумали теги.

**Создание тегов версий**

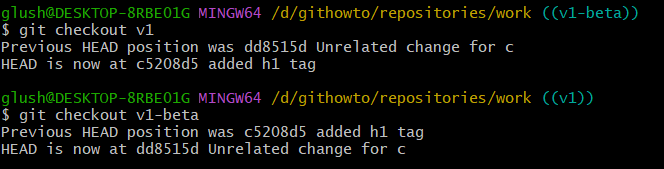


Обозначим версию, предшествующую текущей, названием v1-beta. Прежде всего, мы переключимся на предыдущую версию. Вместо того чтобы искать хеш коммита, мы будем использовать обозначение ^, а именно v1^, указывающее на коммит, предшествующий v1.

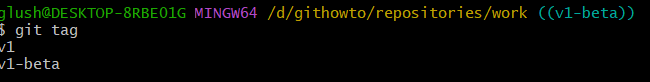




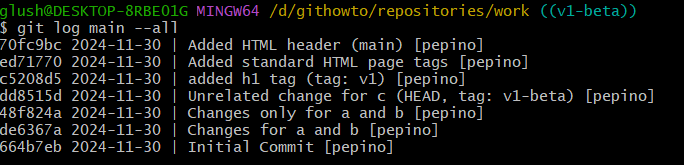
Теперь попробуйте попереключаться между двумя отмеченными версиями.



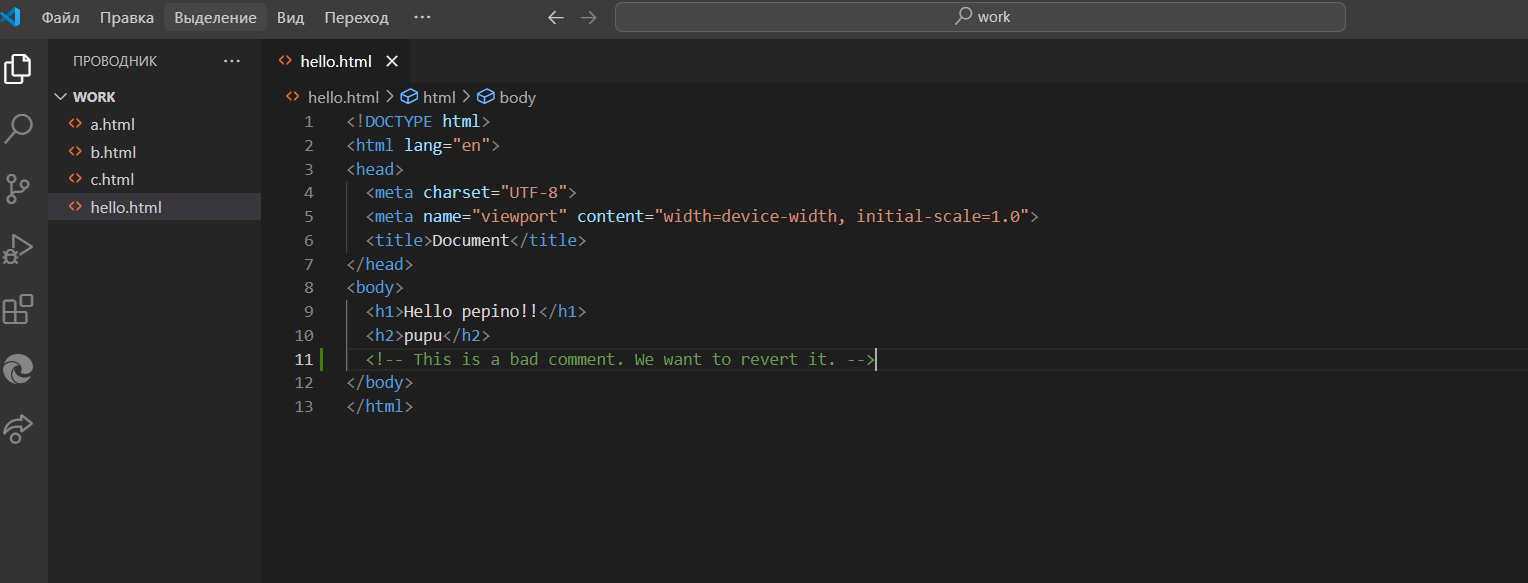
Вы можете увидеть, какие теги доступны, используя команду git tag.

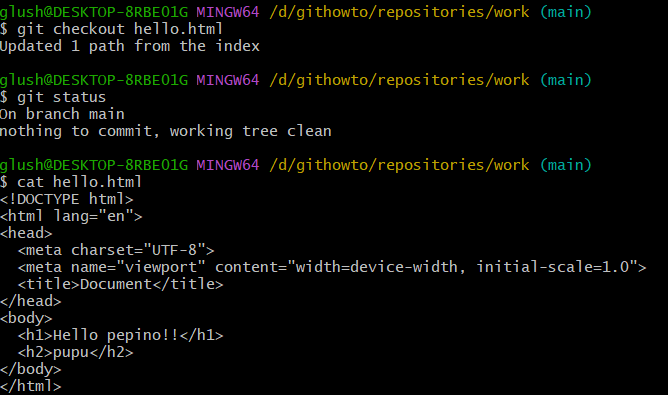


Вы также можете посмотреть теги в логе.



**Отмена локальных изменений (до индексации)**

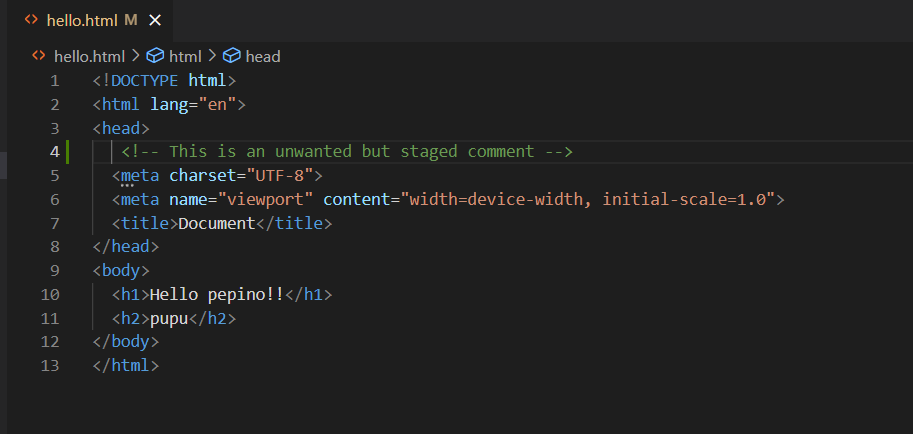




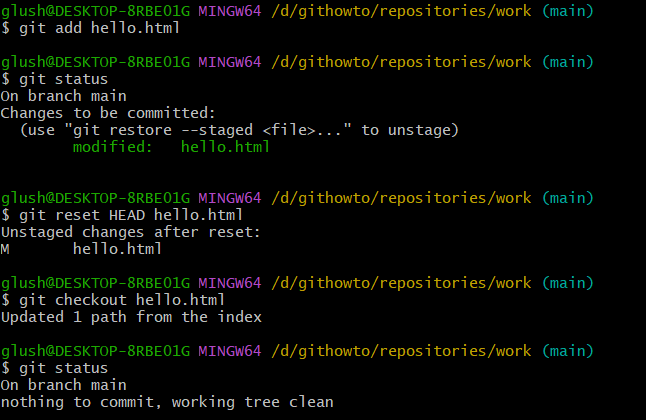
Команда status показывает нам, что в рабочей директории не было сделано никаких незафиксированных изменений. И «нежелательный комментарий» больше не является частью содержимого файла.

**Отмена проиндексированных изменений (перед коммитом)**

Внесите изменение в файл hello.html в виде нежелательного комментария



Команда reset (по умолчанию) не изменяет рабочую директорию. Поэтому рабочая директория всё еще содержит нежелательный комментарий. Мы можем использовать команду checkout из предыдущего урока, чтобы убрать нежелательные изменения в рабочей директории.

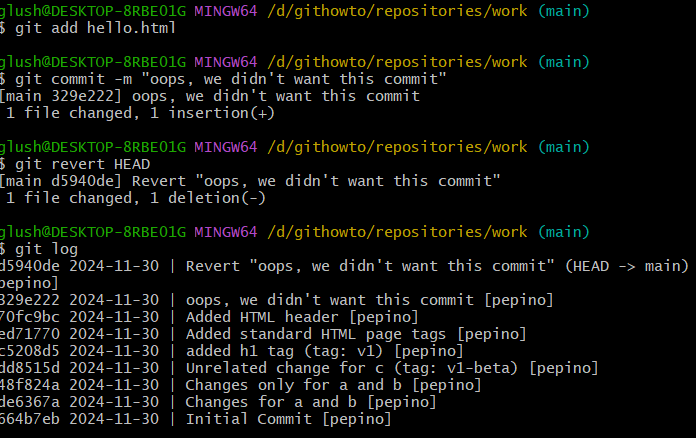


**Отмена коммитов**

Иногда вы понимаете, что новые коммиты являются неверными, и хотите их отменить. Есть несколько способов решения этого вопроса, здесь мы будем использовать самый безопасный.

Мы отменим коммит путем создания нового коммита, отменяющего нежелательные изменения.

Чтобы отменить коммит, нам необходимо сделать коммит, который удаляет изменения, сохраненные нежелательным коммитом.



**Удаление коммитов из ветки**

revert из предыдущего раздела является мощной командой, которая позволяет отменить любые коммиты в репозиторий. Однако, и оригинальный и «отмененный» коммиты видны в истории ветки (при использовании команды git log).

Часто мы делаем коммит, и сразу понимаем, что это была ошибка. Было бы неплохо иметь команду «возврата», которая позволила бы нам сделать вид, что неправильного коммита никогда и не было. Команда «возврата» даже предотвратила бы появление нежелательного коммита в истории git log.

Мы уже видели команду reset и использовали ее для согласования области подготовки с выбранным коммитом (в предыдущем уроке мы использовали коммит HEAD).

Если выполнить команду reset с указанием ссылки на коммит (т.е. метки HEAD, имени ветки или тега, хеша коммита), то команда...

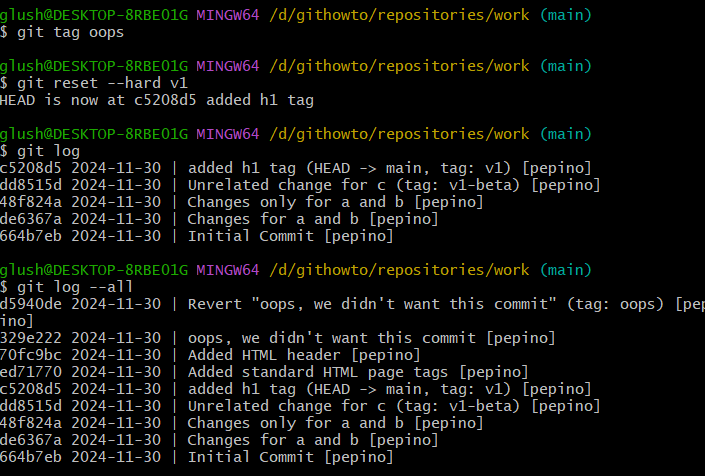
1. Изменит текущую ветку, чтобы она указывала на указанный коммит.
2. Опционально сбросит область подготовки до соответствия с указанным коммитом.
3. Опционально сбросит рабочую директорию до соответствия с указанным коммитом.

Что же случается с ошибочными коммитами? Оказывается, что коммиты все еще находятся в репозитории. На самом деле, мы все еще можем на них ссылаться. Помните, в начале этого урока мы создали для отмененного коммита тег oops? Давайте посмотрим на *все* коммиты.

Мы видим, что ошибочные коммиты не исчезли. Они все еще находятся в репозитории. Просто они отсутствуют в ветке main. Если бы мы не отметили их тегами, они по-прежнему находились бы в репозитории, но не было бы никакой возможности ссылаться на них, кроме как при помощи хешей этих коммитов. Коммиты, на которые нет ссылок, остаются в репозитории до тех пор, пока не будет запущен сборщик мусора.

Сброс в локальных ветках, как правило, безопасен. Последствия любой «аварии» как правило, можно восстановить простым сбросом с помощью нужного коммита.

Однако, если ветка уже стала общедоступной на удаленных репозиториях, сброс может сбить с толку других пользователей ветки.



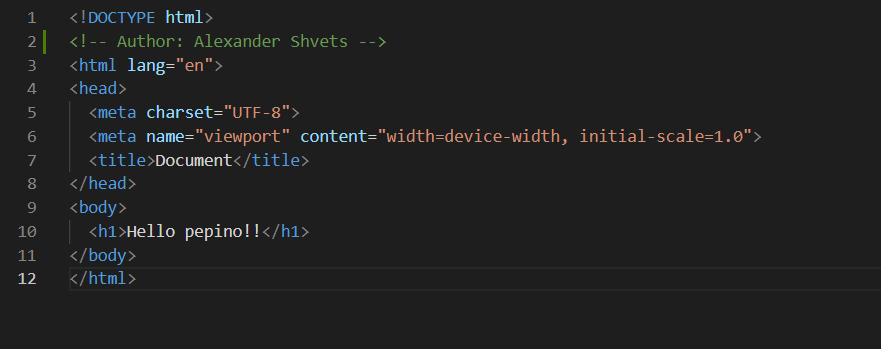
**Удаление тега**oops

Тег oops свою функцию выполнил, давайте удалим его. Это позволит внутреннему механизму Git убрать остаточные коммиты, на которые теперь не ссылаются никакие ветки или теги.



**Внесение изменений в коммиты**

Добавьте в страницу комментарий автора.

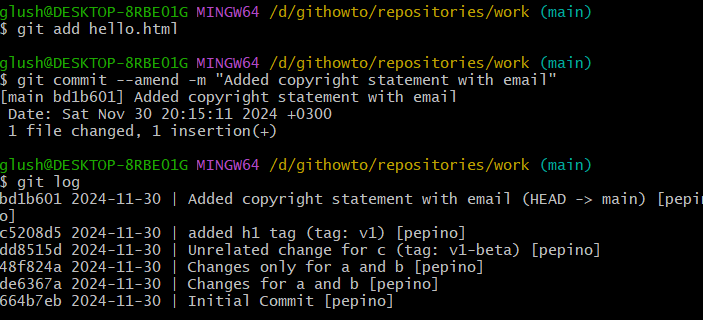




Однако после создания коммита вы понимаете, что любой хороший комментарий должен включать электронную почту автора. Обновите страницу hello.html, включив в нее email.

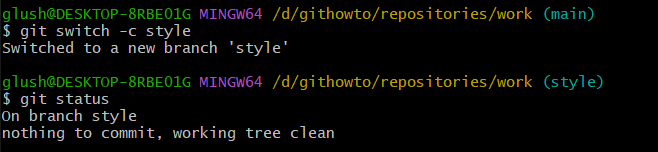


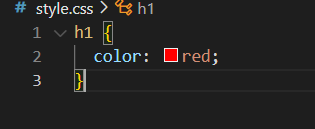
Мы действительно не хотим создавать отдельный коммит только ради электронной почты. Давайте изменим предыдущий коммит, включив в него адрес электронной почты.

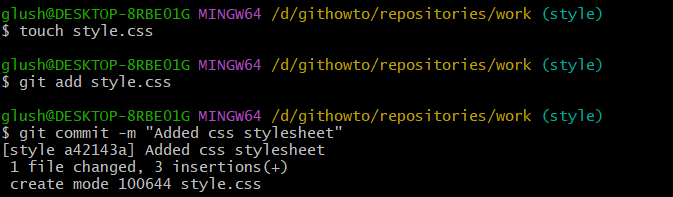


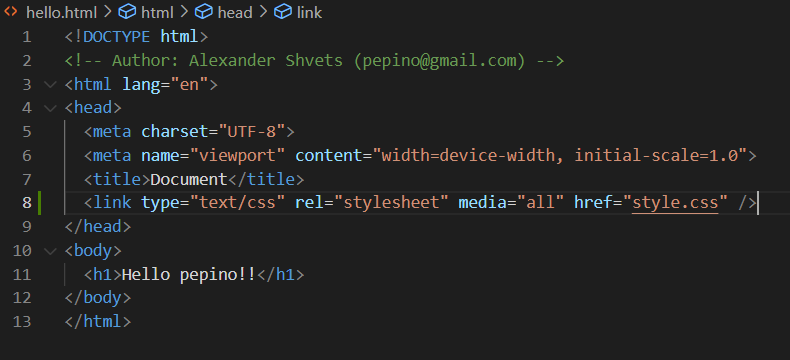
**Создание ветки**

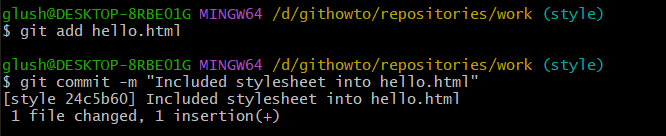
Разработка новой функциональности всегда связана с риском: разработка может занять много времени, вы можете в конечном итоге отказаться от неё и т. д. По этой причине лучше всего изолировать разработку фичи в отдельной ветке. Когда фича будет готова, вы сможете слить эту ветку с веткой main. До того времени ветка main будет защищена от рискованного и непроверенного кода. Кроме того, вы можете работать над несколькими фичами параллельно, над каждой в собственной ветке. Вы также можете в любой момент вносить изменения в ветке main, например, чтобы исправить ошибку в стабильном коде. Пришло время сделать нашу страницу более стильной с помощью CSS. Мы будем развивать эту возможность в новой ветке под названием style.











**Переключение веток**

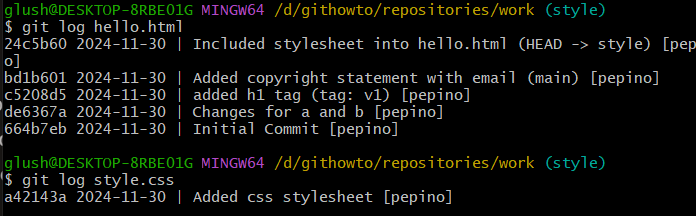
Просто используйте команду git switch для переключения между ветками.



**Перемещение файлов**

Я доволен нашими CSS-изменениями, но есть только один момент, который я хотел бы решить до того, как мы объединим наши изменения с main. Давайте переименуем файл hello.html в index.html. Также давайте перенесем наш файл стилей в специально отведенную директорию css.

Git позволяет просматривать историю изменений конкретного файла. Давайте посмотрим историю изменений файла hello.html перед его переименованием.



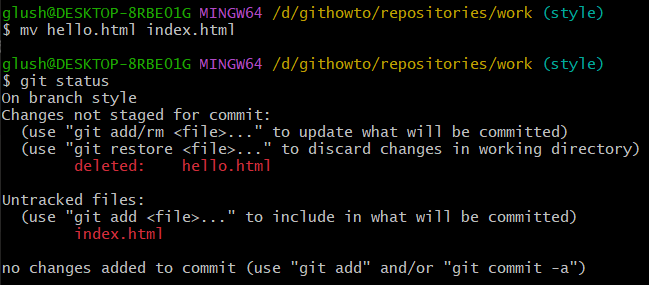
Возможность просмотра истории изменений для конкретного файла очень полезна. Она позволяет увидеть, что именно изменилось, а также кто и когда внес эти изменения. Кроме того, существует возможность увидеть изменения, связанные с конкретным коммитом. Я постоянно пользуюсь этим, чтобы разобраться, почему та или иная штука была реализована именно так в настоящей версии кода.

Команда show используется для просмотра изменений в конкретном коммите. Посмотрим изменения в файле hello.html в коммите, с тегом v1 (можно использовать любую ссылку на коммит, например, метку HEAD, хеш коммита, имя ветки или тега и т.д.).



Как видите, очень удобно иметь возможность видеть историю изменений конкретного файла. Но когда вы переименовываете или перемещаете какой-либо файл, есть риск потерять историю этого файла, если вы выполните эту процедуру неправильно.

Давайте переименуем наш файл hello.html в index.html с помощью стандартной команды mv и посмотрим, что из этого получится.

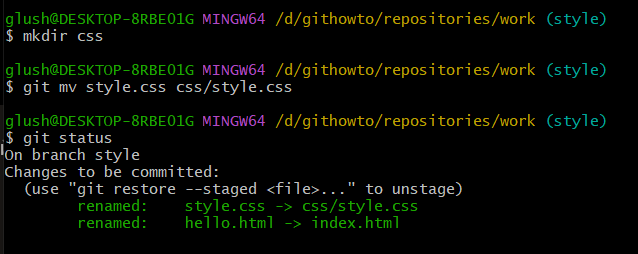


Git воспринимает наше изменение так, будто файл был удален и создан заново. Это тревожный звоночек. Нам нужно сообщить Git, что мы именно переименовали файл, а не удалили его и сразу создали новый. В простейшем случае Git сам поймёт, что файл был переименован, как только мы добавим его в индекс:

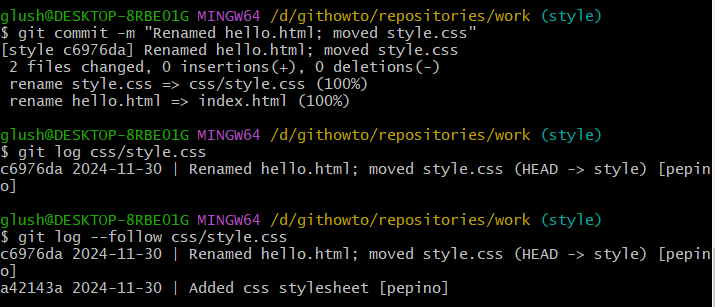


Видите, файл указан как переименованный. Но это всего лишь Git пытается проявить смекалку. Это не всегда работает. Например, если вы переименовали, **а также** изменили кучу файлов, Git может оказаться не в состоянии понять, что именно было переименовано. В этом случае вы можете потерять информацию об истории файлов до их переименования, поскольку файлы будут восприняты как новые.

В большинстве операционных систем переименование и перемещение файлов — это одно и то же. Итак, давайте переместим наш файл style.css в директорию css, но на этот раз сделаем это безопасно с помощью команды git mv. Эта команда гарантирует, что перемещение будет записано в истории Git как перемещение.

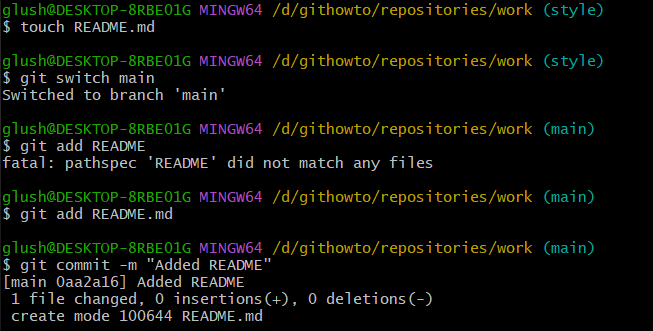


Давайте закоммитим наши изменения и проверим историю изменений в файле css/styles.css. Для просмотра истории файла до его перемещения нам потребуется добавить опцию --follow. Выполним оба варианта команды, чтобы понять разницу.



**Изменения в ветке**main

Как я уже говорил, Git позволяет работать с несколькими ветками одновременно. Это очень удобно при работе в команде, поскольку люди могут параллельно работать над разными функциями. Это также полезно при работе в одиночку: разрабатывая функции в отдельных ветках, вы можете исправлять ошибки и выпускать небольшие обновления, используя стабильный код в ветке main. Создадим файл README. В нем будет рассказано о сути нашего проекта.

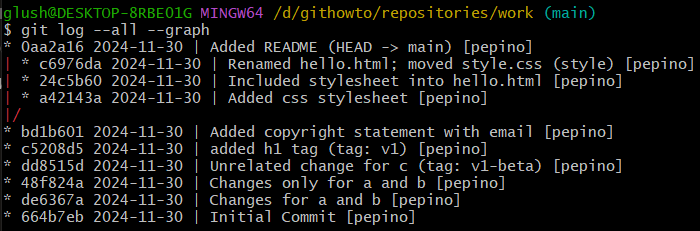


**Просмотр отличающихся веток**

Теперь у нас есть две расходящиеся ветки в репозитории. Используйте следующую команду log для просмотра веток и их расхождения.

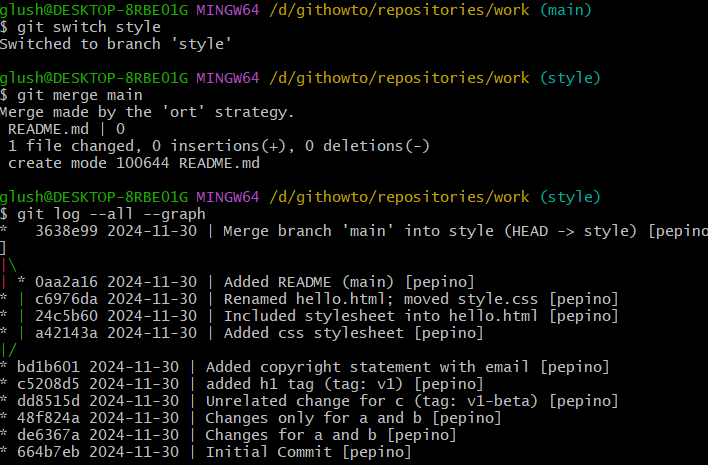
Опция --all гарантирует, что мы видим все ветки, так как по умолчанию в логе показывается только текущая ветка.

Опция --graph добавляет простое дерево коммитов, представленное в виде простых текстовых линий. Мы видим обе ветки (style и main) причём ветка main помечена как HEAD, что означает, что она является текущей. Общим предком для обеих веток является ветка, в которую был внесен коммит «Added copyright statement with email».



**Слияние**

Слияние переносит изменения из двух веток в одну. Давайте вернемся к ветке style и сольем main со style.

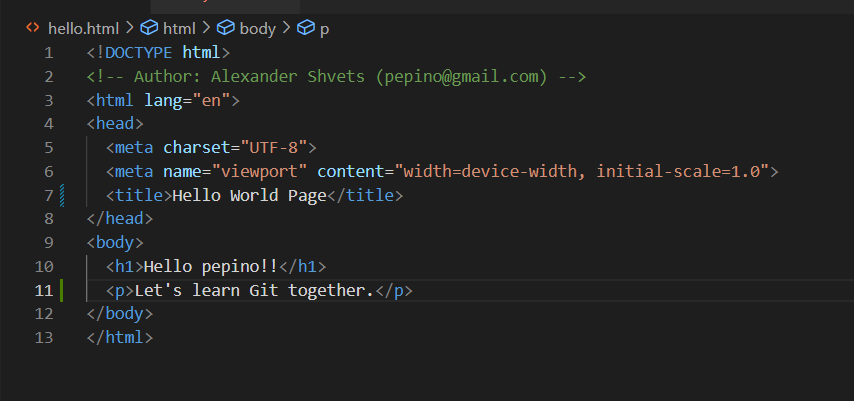


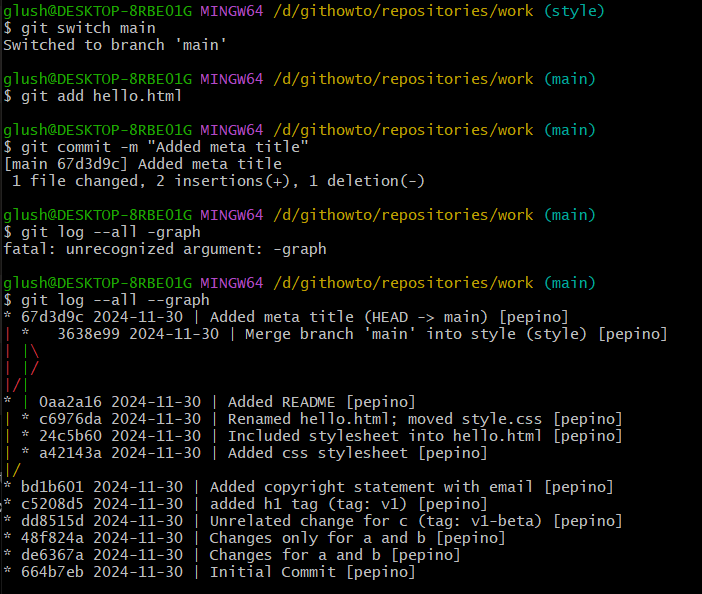
**Создание конфликта**

При слиянии двух веток Git пытается перенести изменения из одной ветки в другую. Если в обеих ветках была изменена одна и та же часть файла, Git может не справиться с автоматическим слиянием изменений. В этом случае Git сообщит о конфликте и попросит разрешить его вручную. В этом уроке мы смоделируем конфликт, а затем научимся его разрешать.

В реальной жизни конфликты слияния регулярно возникают при работе в команде. Например, вы и ваш коллега начали работать над двумя разными фичами, затрагивающими одни и те же файлы. Ваш коллега закончил работу первым и слил свои изменения в ветку main. Теперь вы хотите слить свои изменения в ветку main. Но ветка main теперь отличается от той, с которой вы начинали работать в начале — в ней появился новый код, присланный вашим коллегой. Вероятно, Git не сможет автоматически объединить ваши изменения и попросит помощи человека.

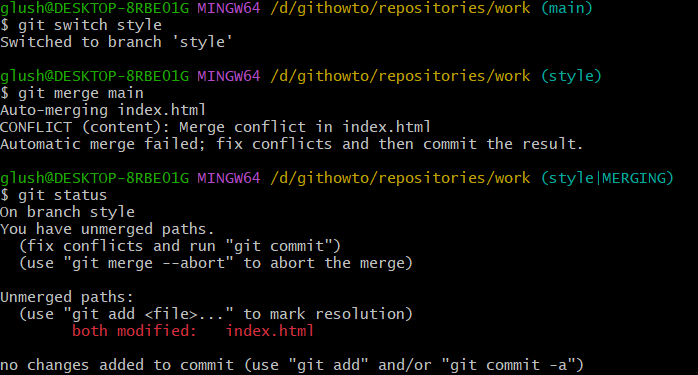
Помните, что в нашей ветке main страница по-прежнему называется hello.html? Переключитесь обратно на ветку main и внесите следующие изменения:

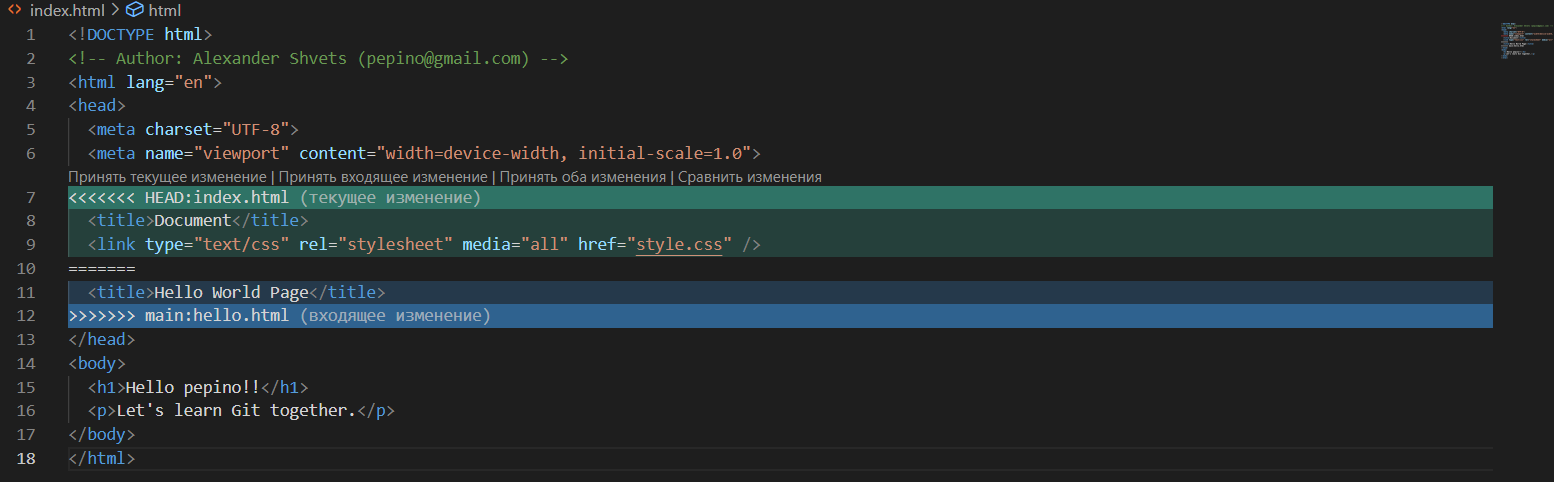


****

После коммита «Added README» ветка main была объединена с веткой style, но в настоящее время в main есть дополнительный коммит, который не был слит с style.

**Разрешение конфликтов**

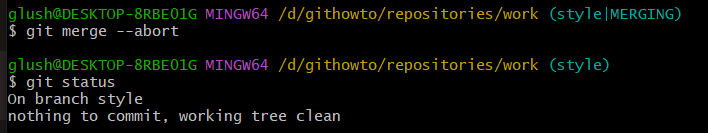




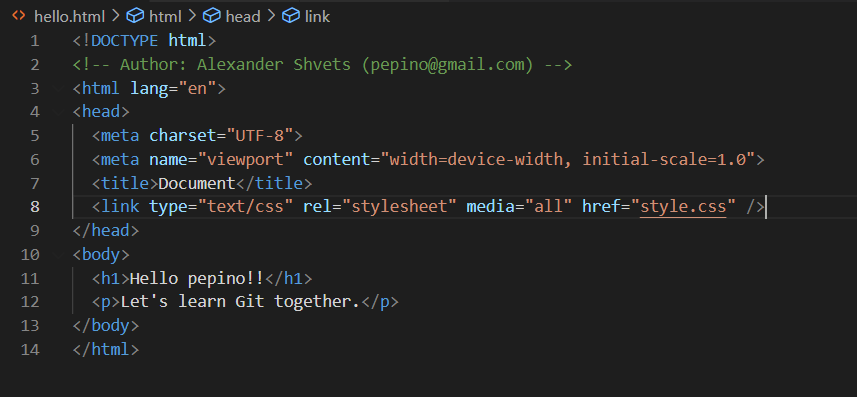
Часть между <<<<<<< >>>>>>> является конфликтом. Верхняя часть соответствует ветке style, которая является текущей веткой (или HEAD) репозитория. Нижняя часть соответствует изменениям из ветки main. Git не может решить, какие изменения применить, поэтому он просит вас разрешить конфликт вручную. Вы можете оставить изменения из ветки style или из main, либо объединить их любым удобным способом. Вы также можете внести в файл любые другие изменения.

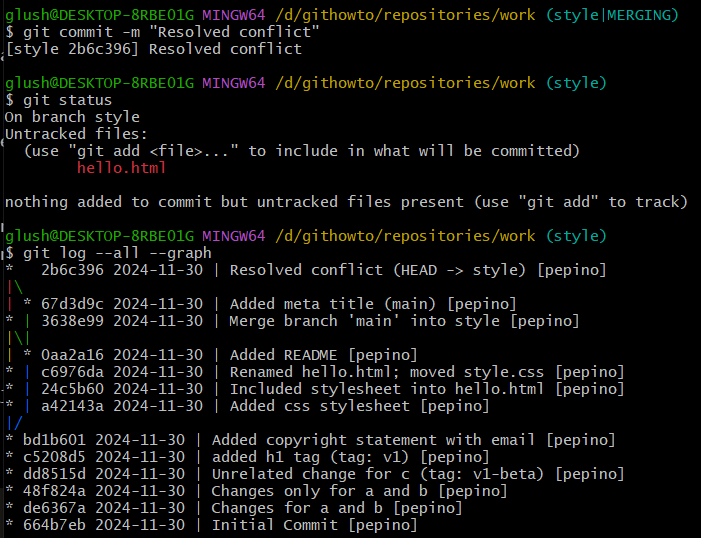
Кстати, вы заметили, что наше второе изменение, тег <p>, не является частью конфликта? Это потому, что Git сумел автоматически объединить ее.

Прежде чем мы приступим к разрешению нашего конфликта, хочу заметить, что сразу бросаться к разрешению конфликта не всегда оптимально. Конфликт может быть вызван изменениями, о которых вы не знаете. Или же изменения слишком велики, чтобы разрешить конфликт сразу. По этой причине Git позволяет прервать слияние и вернуться к предыдущему состоянию. Для этого можно воспользоваться командой git merge --abort, как это было предложено командой status, которую мы выполнили ранее.



Чтобы разрешить конфликт, нужно отредактировать файл до состояния, которое нас устраивает, и затем закоммитить его как обычно. В нашем случае мы объединим изменения из обеих веток. Поэтому мы отредактируем файл до следующего состояния:





rebase**против**merge

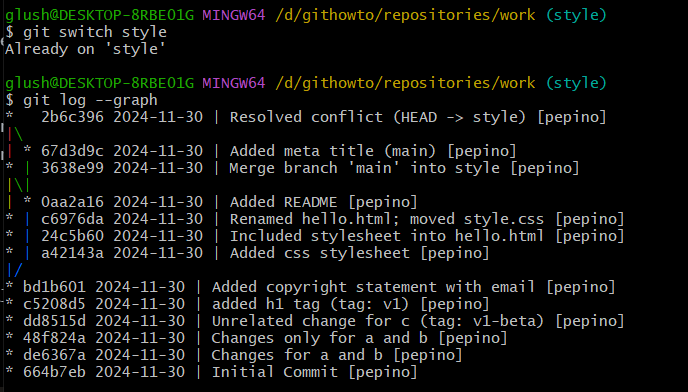
Давайте рассмотрим различия между слиянием и перебазированием. Для того чтобы это сделать, нам нужно вернуться в репозиторий в момент до первого слияния, а затем повторить те же действия, но с использованием перебазирования вместо слияния.

Мы будем использовать команду reset для возврата веток к предыдущему состоянию.

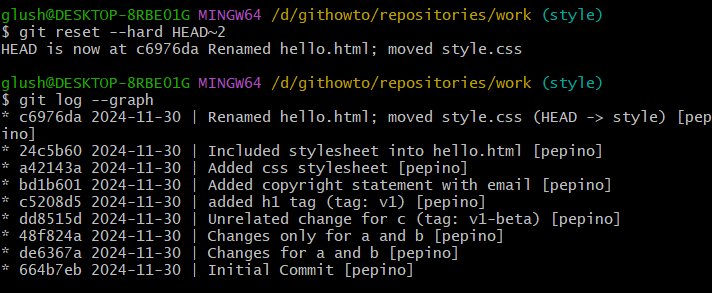
**Сброс ветки**style

Давайте вернемся во времени на ветке style к точке *перед* тем, как мы слили ее с веткой main. Мы можем сбросить ветку к любому коммиту при помощи команды reset. По сути, это изменение указателя ветки на любую точку дерева коммитов.

В этом случае мы хотим вернуться в ветке style в точку перед слиянием с main. Нам необходимо найти последний коммит перед слиянием.

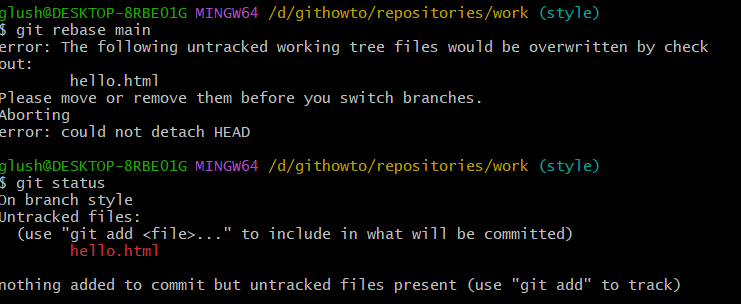


Это немного трудно читать, но, глядя на данные, мы видим, что коммит «Renamed hello.html; moved style.css» был последним на ветке style перед слиянием. Давайте сбросим ветку style к этому коммиту. Чтобы сослаться на этот коммит, мы либо используем его хеш, либо посчитаем, что этот коммит находится за 2 коммита до HEAD, то есть HEAD~2 в нотации Git.



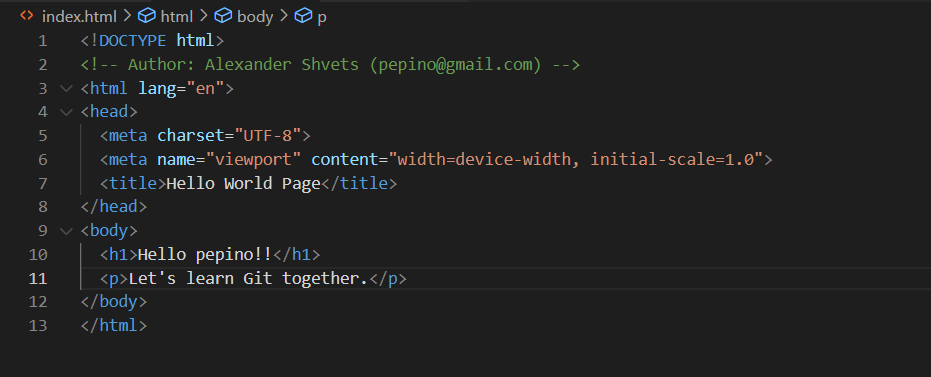
**Перебазирование**

Мы вернули ветку style к точке перед первым слиянием. При этом в ветке main есть два коммита, которых нет в ветке style: новый файл README и конфликтующее изменение в файле index.html. На этот раз мы перенесем эти изменения в ветку style с помощью команды rebase, а не merge.

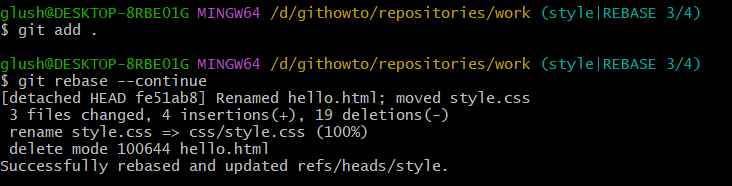


Опять возник конфликт! Обратите внимание, что конфликт возник в файле hello.html, а не в файле index.html, как в прошлый раз. Это связано с тем, что rebase находился в процессе применения изменений style поверх ветки main. Файл hello.html в main еще не был переименован, поэтому он все еще имеет старое имя.

При слиянии возник бы «обратный» конфликт. При слиянии изменения ветки main были бы применены поверх ветки style. В ветке style файл переименован, поэтому конфликт возник бы в файле index.html.



Но после этого нам не нужно коммитить изменения. Мы можем просто добавить файл в индекс и продолжить процесс rebase. Вот почему я люблю rebase! Он позволяет мне устранять конфликты, не создавая кучу уродливых конфликтов слияния.



Используйте команду rebase:

* Когда вы получаете изменения из удаленного репозитория и хотите применить их к своей локальной ветке.
* Если вы хотите, чтобы история коммитов была линейной и легко читаемой.

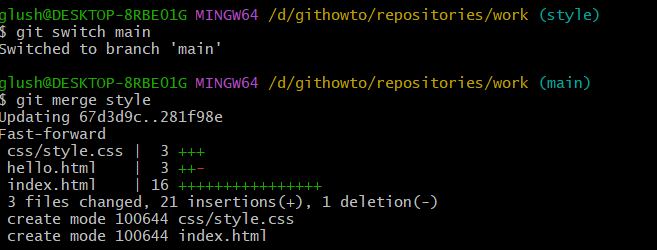
Не используйте команду rebase:

* Если текущая ветка является публичной и общей. Переписывание таких веток будет мешать работе других членов команды.
* Если важна *точная* история ветки коммитов (поскольку команда rebase переписывает историю коммитов).

Учитывая приведенные выше рекомендации, я предпочитаю использовать команду rebase для краткосрочных, локальных веток и команду merge для веток в публичном репозитории.

**Слияние в ветку**main

Мы поддерживали соответствие ветки style с веткой main (с помощью rebase), теперь давайте сольем изменения style в ветку main.



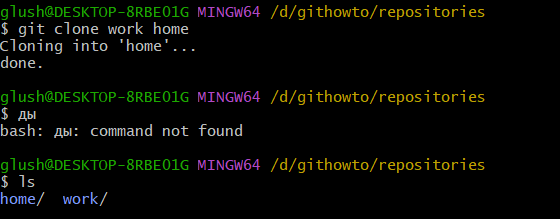
**Часть 2. Несколько репозиториев**

До сих пор мы работали только с одним Git-репозиторием. Однако Git - это **распределенная** система управления версиями, а значит, она отлично подходит для работы с несколькими репозиториями. Эти дополнительные репозитории могут храниться локально, быть доступными через сеть или Интернет. Они также могут быть размещены на GitHub, GitLab, BitBucket или любом другом Git-хостинге.

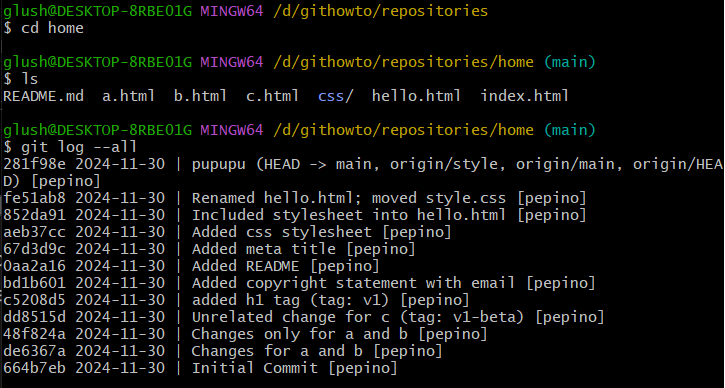
В следующем разделе мы представим, что решили взять некоторую работу на дом. В былые времена можно было бы перенести этот репозиторий на флешку и взять его с собой домой. Сегодня мы, скорее всего, поделимся репозиторием через GitHub. На самом деле, не важно, как вы делитесь своей работой: Git будет работать одинаково. Большая часть информации, изложенной в этом разделе, может быть применена и для работы с несколькими репозиториями, независимо от того, хранятся ли они локально или передаются по сети.

Поэтому для простоты представим, что мы используем два независимых репозитория, располагая их локально в разных директориях: work и home.

**Клонирование репозиториев**

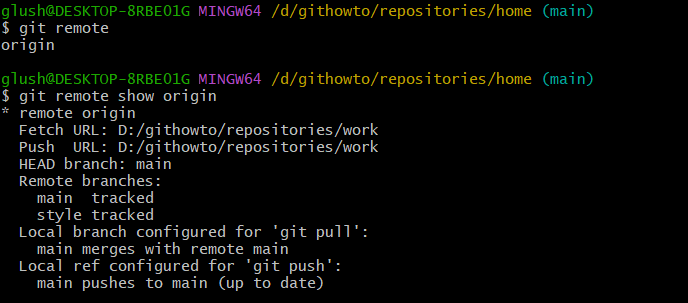


**Просмотр клонированного репозитория**



**Что такое origin?**

Мы видим, что клонированный репозиторий знает об имени по умолчанию удаленного репозитория. Давайте посмотрим, можем ли мы получить более подробную информацию об имени по умолчанию:



Мы видим, что имя по умолчанию (origin) удаленного репозитория — изначальное work. Удаленные репозитории обычно размещаются на отдельной машине, возможно, централизованном сервере. Однако, как мы видим здесь, они могут с тем же успехом указывать на репозиторий на той же машине. Нет ничего особенного в имени origin, однако существует традиция использовать origin в качестве имени первичного централизованного репозитория (если таковой имеется).

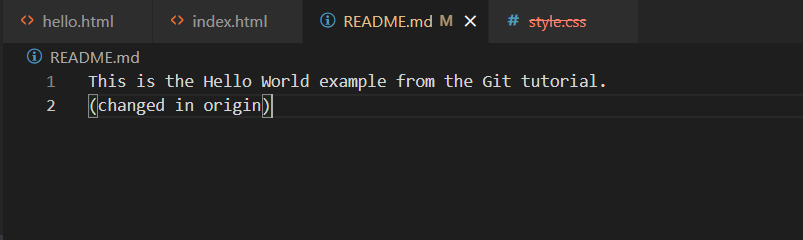
**Удаленные ветки**

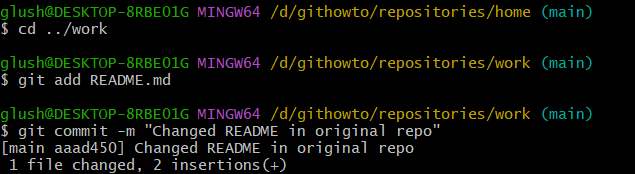
Давайте посмотрим на ветки, доступные в нашем клонированном репозитории.



**Изменение оригинального репозитория**

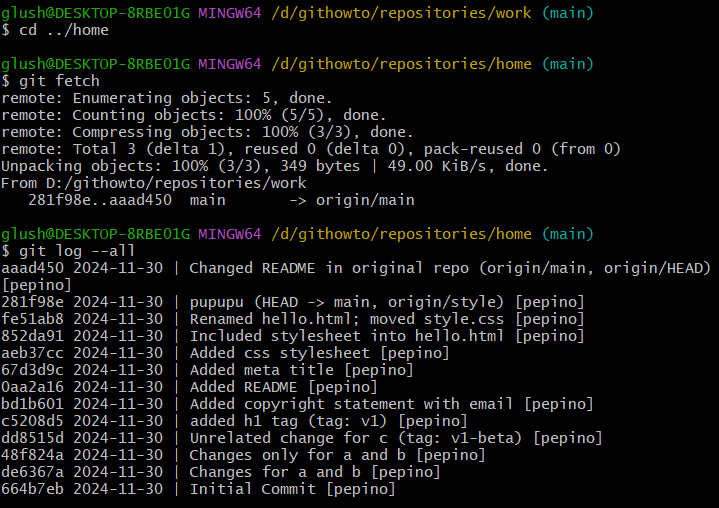
Внести некоторые изменения в оригинальный репозиторий, чтобы затем попытаться подтянуть и слить изменения из удаленной ветки в текущую.





Теперь в оригинальном репозитории есть более поздние изменения, которых нет в клонированной версии. Далее мы подтянем и сольем эти изменения в клонированный репозиторий.

**Подтягивание изменений**

****

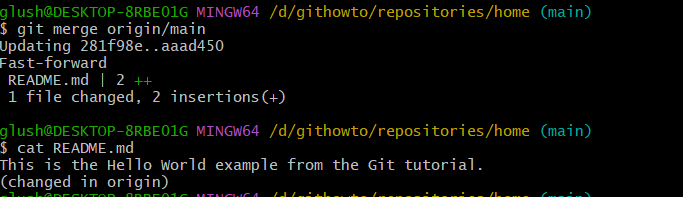
На данный момент в репозитории есть все коммиты из оригинального репозитория, но они не интегрированы в локальные ветки клонированного репозитория.

В истории выше найдите коммит «Changed README in original repo». Обратите внимание, что коммит включает в себя коммиты origin/main и origin/HEAD.

Теперь давайте посмотрим на коммит «Renamed hello.html; moved style.css». Вы увидите, что локальная ветка main указывает на этот коммит, а не на новый коммит, который мы только что подтянули.

Выводом является то, что команда git fetch будет подтягивать новые коммиты из удаленного репозитория, но не будет сливать их с вашими наработками в локальных ветках.

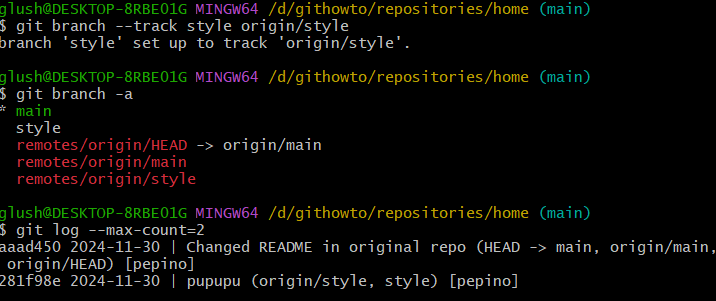
**Слияние подтянутых изменений**

****

Команда fetch позволяет контролировать то, что именно подтягивается и сливается в ваши локальные ветки, но для удобства существует также команда pull, которая подтягивает и сливает изменения из удаленной ветки в текущую одним вызовом.

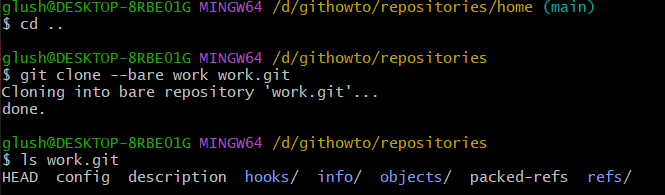
**Добавление ветки наблюдения**

Ветки, которые начинаются с remotes/origin являются ветками оригинального репозитория. Обратите внимание, что у вас больше нет ветки под названием style, но система контроля версий знает, что в оригинальном репозитории ветка style была.



**Чистые репозитории**

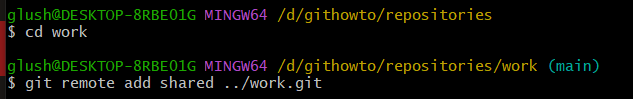
Чистый репозиторий — это репозиторий, не имеющий рабочей директории. Он содержит только директорию .git, в которой Git хранит все свои внутренние данные. Основное предназначение таких репозиториев — быть центральным хранилищем, в которое разработчики могут отправлять и из которого могут получать данные. Поэтому в них нет смысла создавать рабочие файлы, они будут только впустую занимать место на диске. Чистые репозитории также используются в сервисах Git-хостинга таких, как GitHub и GitLab. В следующих уроках мы узнаем, как создать чистый репозиторий и как отправлять в него изменения.



Принято считать, что репозитории, заканчивающиеся на .git, являются чистыми репозиториями. Мы видим, что в репозитории work.git нет рабочей директории. По сути, это просто директория .git из обычного репозитория.

**Добавление удаленного репозитория**

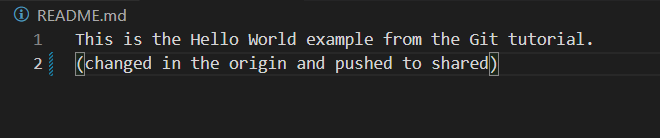
Давайте добавим репозиторий work.git к нашему оригинальному репозиторию.

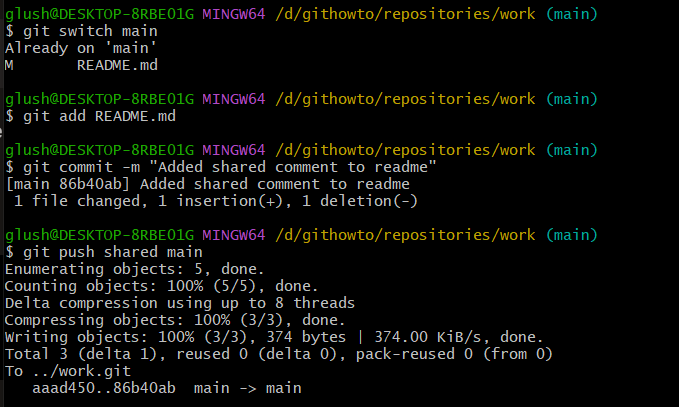


**Отправка изменений**

Поскольку чистые репозитории обычно располагаются на каком-либо удаленном сервере, вы не сможете туда просто зайти, дабы подтянуть изменения. Поэтому нам необходимо как-нибудь передать наши изменения в репозиторий.

Начнем с создания изменения, которое нужно передать в репозиторий. Отредактируйте README и закоммитьте его:

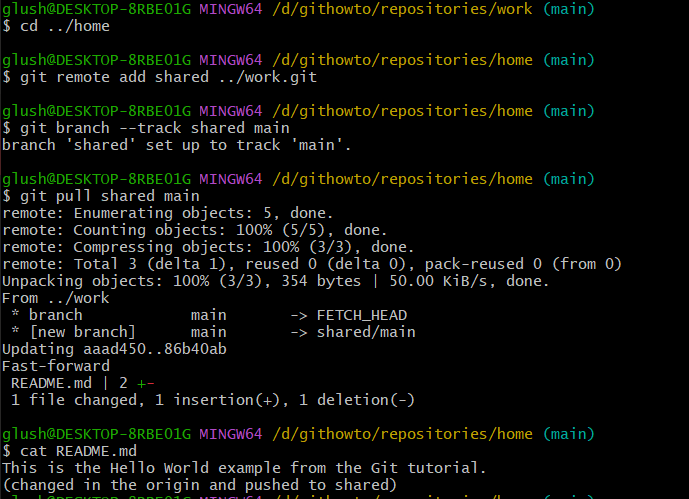




*Общим* называется репозиторий, получающий отправленные нами изменения. Помните, мы добавили его в качестве удаленного репозитория в предыдущем уроке?

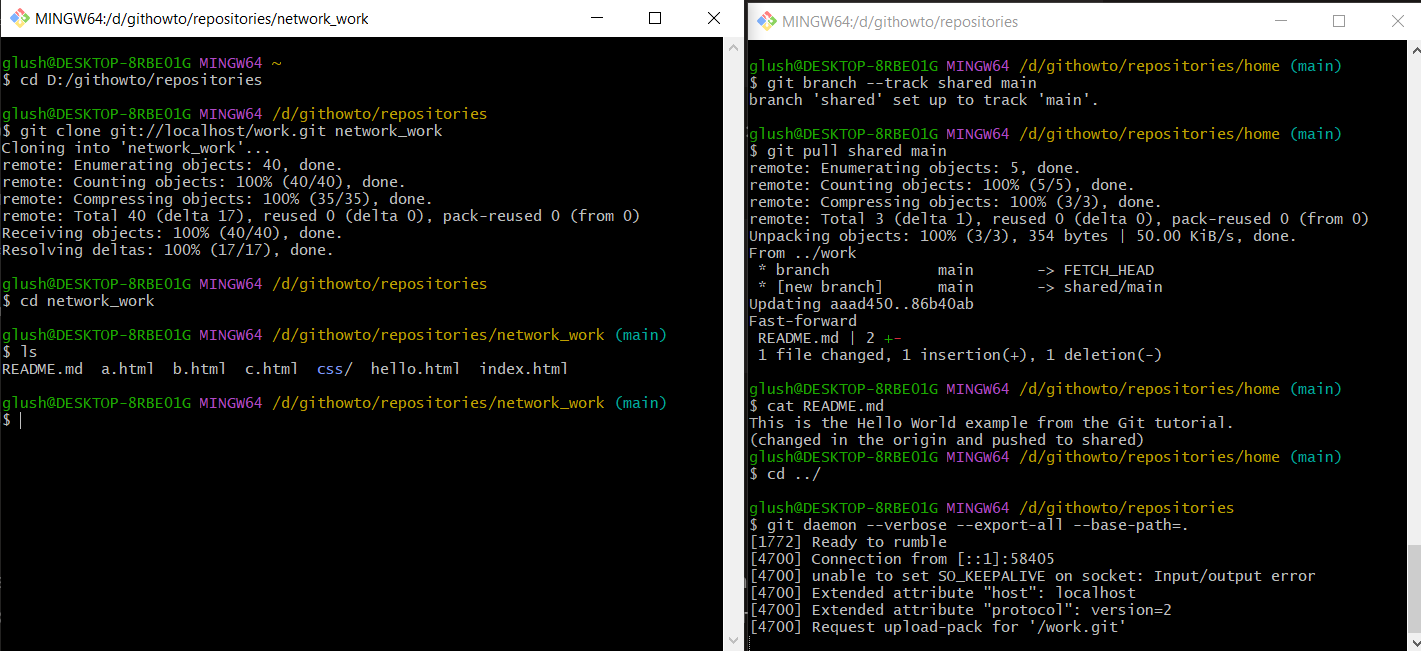
**Подтягивание общих изменений**

Быстро переключитесь в репозиторий home и подтяните изменения, только что отправленные в общий репозиторий.



**Размещение ваших Git-репозиториев**

Хотите создать свой собственный GitHub? Существует множество способов совместного использования репозиториев Git по сети. Здесь приведен простой и быстрый (но ненадежный и опасный) способ.



Если вы хотите разрешить отправку изменений (push) в репозиторий Git Daemon, добавьте метку --enable=receive-pack к команде git daemon. Будьте осторожны, этот сервер не производит аутентификацию, поэтому любой сможет отправлять изменения в ваш репозиторий.

**Выводы**: Git — мощная и сложная распределенная система контроля версий, которая работает с изменениями, а не файлами. Git позволяет создавать репозитории с ветками, коммитить и отслеживать изменения в файлах, а также позволяет откатываться к любой интересуемой версии проекта благодаря истории изменений.