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»

|  |  |
| --- | --- |
| Выполнил: | Проверил: |
| cтудент гр. 210901  Бужинский И.И. | Давыдович К. И. |
|  |  |

Минск 2024

**Цель:** сформировать понимание и специфику работы с инструментом контроля версий «Git» и научиться пользоваться его основным функционалом.

Курс располагается по ссылке *https://githowto.com/ru*.

**Прохождение курса**

**Часть I: Основы Git**

**Задание 2. 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:

mkdir work

cd work

touch hello.html

Содержание файла:

Hello, World

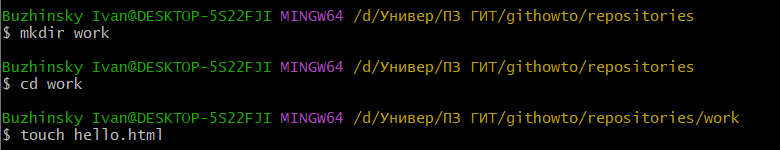


Рисунок 2 – Создание файла hello.html

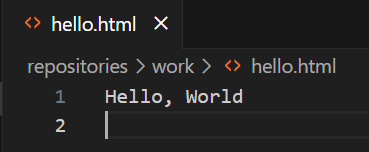


Рисунок 3 – Содержание файла hello.html

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

git init

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

git add hello.html

git commit -m "Initial Commit"

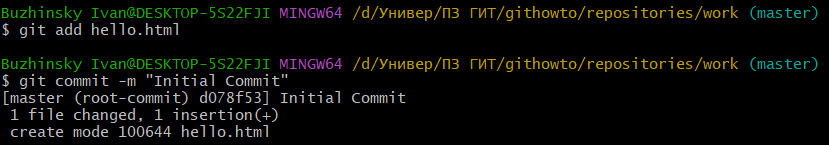


Рисунок 4 – Добавление файла hello.html в Git-репозиторий

**Задание 3. Проверка состояния**

Используйте команду git status, чтобы проверить текущее состояние репозитория.

git status

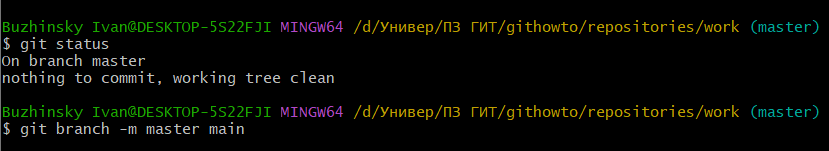


Рисунок 5 – Проверка текущего состояния репозитория

**Задание 4. Внесение изменений**

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

<h1>Hello, World!</h1>

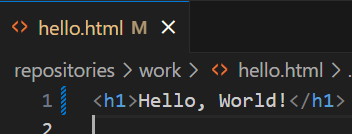


Рисунок 6 – Изменение файла hello.html

Теперь проверьте состояние рабочей директории.

git status

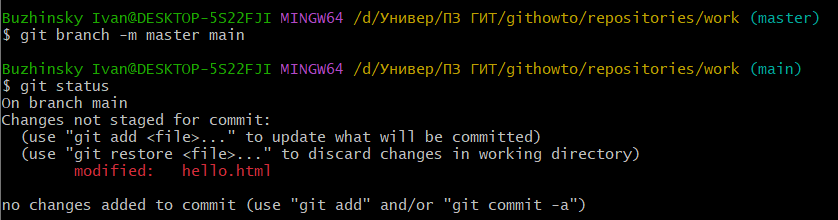


Рисунок 7 – Проверка состояния репозитория

**Задание 5. Индексация изменений**

Теперь дайте команду Git проиндексировать изменения. Проверьте состояние:

git add hello.html

git status

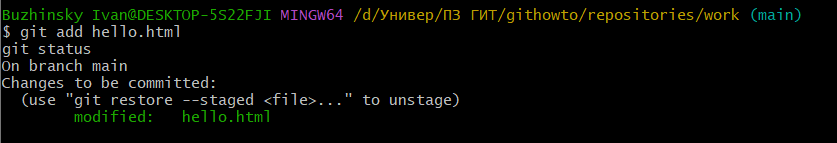


Рисунок 8 – Индексация изменений в репозитории

**Задание 6. Индексация и коммит**

Отдельный шаг индексации в Git позволяет вам разделять большие изменения на маленькие коммиты.

Предположим, что вы отредактировали три файла (a.html, b.html, и c.html). Теперь вы хотите закоммитить все изменения, при этом чтобы изменения в a.html и b.html были одним коммитом, в то время как изменения в c.html логически не связаны с первыми двумя файлами и должны идти отдельным коммитом.

В теории, вы можете сделать следующее:

git add a.html

git add b.html

git commit -m "Changes for a and b"

git add c.html

git commit -m "Unrelated change to c"

Разделяя индексацию и коммит, вы имеете возможность с легкостью настроить, что идет в какой коммит.

**Задание 7. Коммит изменений**

Достаточно об индексации. Давайте сделаем коммит того, что мы проиндексировали, в репозиторий.

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

Сделайте коммит сейчас и проверьте состояние.

git commit

В первой строке введите комментарий: Added h1 tag. Сохраните файл и выйдите из редактора.

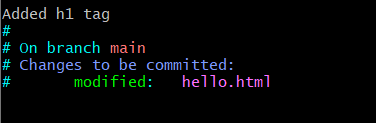


Рисунок 9 – Коммит в Visual Studio Code

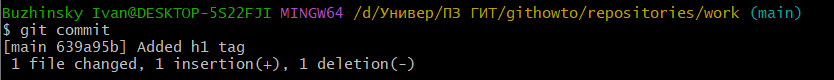


Рисунок 10 – Проверка состояния репозитория

В конце давайте еще раз проверим состояние.

git status

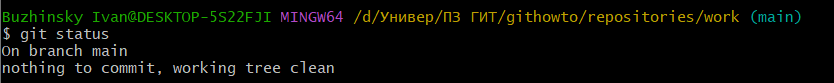


Рисунок 11 – Проверка состояния репозитория

Рабочая директория чиста, можем продолжить работу.

**Задание 8. 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.

Мы попытаемся исследовать эту разницу в данном уроке.

Измените страницу «Hello, World», чтобы она содержала стандартные теги <html> и <body>.

<html>

<body>

<h1>Hello, World!</h1>

</body>

</html>

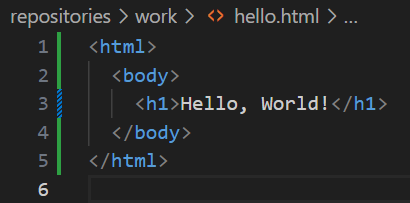


Рисунок 12 – Изменение файла hello.html

Теперь добавьте это изменение в индекс Git.

git add hello.html



Рисунок 13 – Добавление изменения в индекс Git

Теперь добавьте заголовки HTML (секцию <head>) к странице «Hello, World».

<html>

<head>

</head>

<body>

<h1>Hello, World!</h1>

</body>

</html>

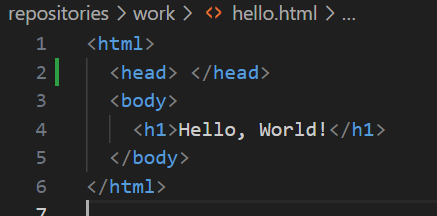


Рисунок 14 – Изменение файла hello.html

Проверьте текущий статус

git status

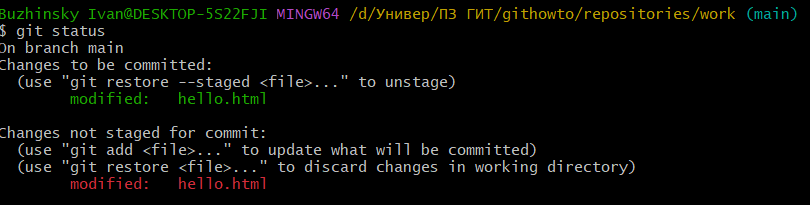


Рисунок 15 – Проверка состояния репозитория

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.

Давайте проверим.

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

git commit -m "Added standard HTML page tags"

git status

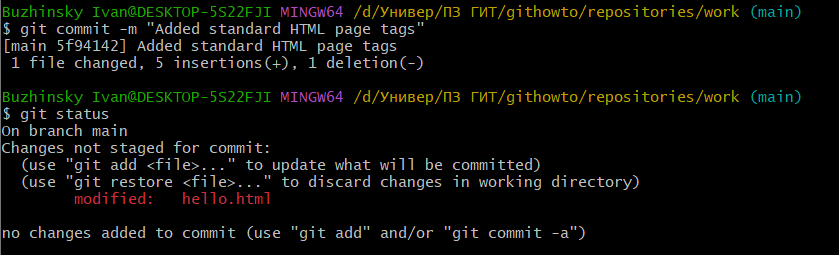


Рисунок 16 – Коммит и проверка состояния репозитория

Команда status показывает, что в файле hello.html ещё есть незаписанные изменения, но область подготовки уже пуста.

Теперь добавьте второе изменение в индекс, а затем проверьте состояние с помощью команды git status.

git add .

git status

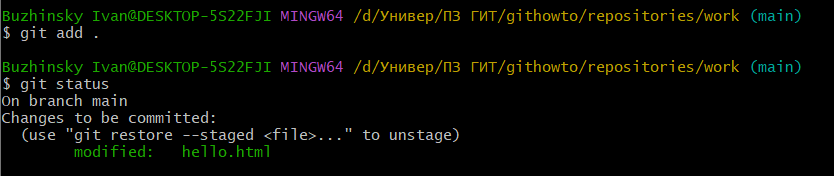


Рисунок 17 – Добавление второго изменения в индекс и проверка состояния репозитория

Второе изменение было проиндексировано и готово к коммиту. Сделайте коммит второго изменения.

git commit -m "Added HTML header"

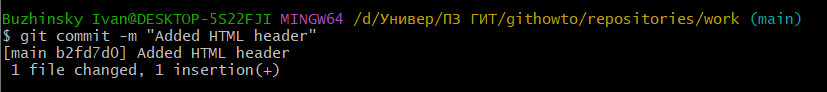


Рисунок 18 – Коммит второго изменения

**Задание 9. История**

Получение списка произведенных изменений – функция команды git log.

git log

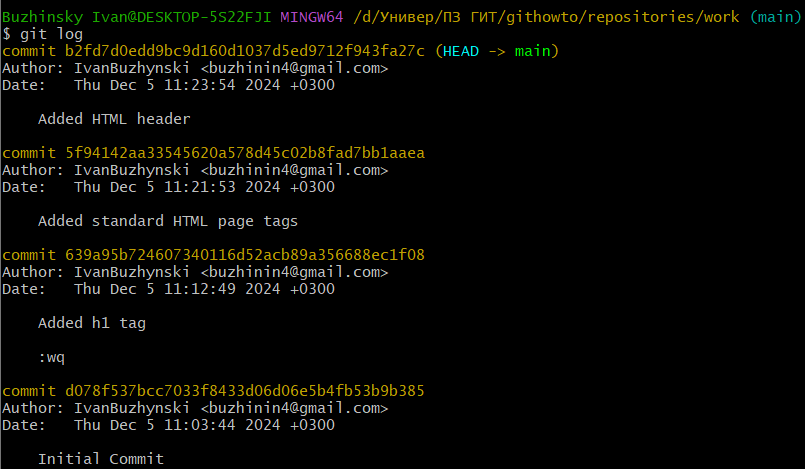


Рисунок 19 – Получение списка изменений

Вот список всех четырех коммитов в репозиторий, которые мы успели совершить.

Вы полностью контролируете то, что отображает log. Например, однострочный формат:

git log --pretty=oneline

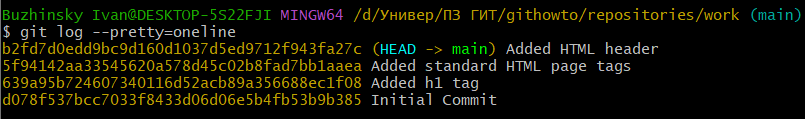


Рисунок 20 – Получение списка изменений в однострочном формате

Вот еще интересные варианты просмотра истории:

git log --oneline --max-count=2

git log --oneline --since="5 minutes ago"

git log --oneline --until="5 minutes ago"

git log --oneline --author="Your Name"

git log --oneline –all

Существует огромное количество вариантов просмотра истории, вы можете порыться на странице руководства git-log, чтобы увидеть их все.

Вот что я использую для просмотра изменений, сделанных за последнюю неделю. Я добавлю --author=Alexander, если я хочу увидеть только изменения, которые сделал я.

git log --all --pretty=format:"%h %cd %s (%an)" --since="7 days ago"

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

git log --pretty=format:"%h %ad | %s%d [%an]" --date=short

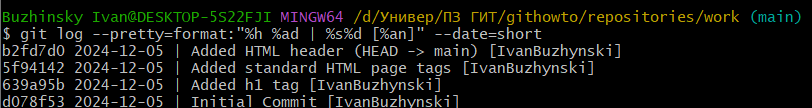


Рисунок 21 – Получение списка изменений в подходящем формате

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:

git config --global format.pretty '%h %ad | %s%d [%an]'

git config --global log.date short

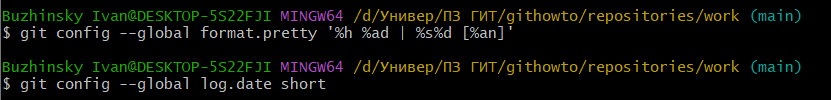


Рисунок 22 – Настройка формата вывода истории по умолчанию

Оба gitx (для Mac) и gitk (для любой платформы) полезны в изучении истории изменений.

**Задание 10. 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.

Получите хеши предыдущих коммитов:

git log

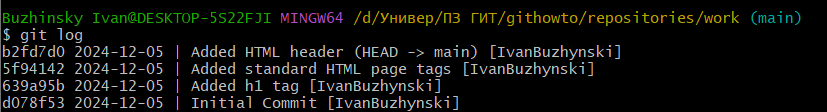


Рисунок 23 – Получение хешей предыдущих коммитов

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.

git checkout <hash>

cat hello.html

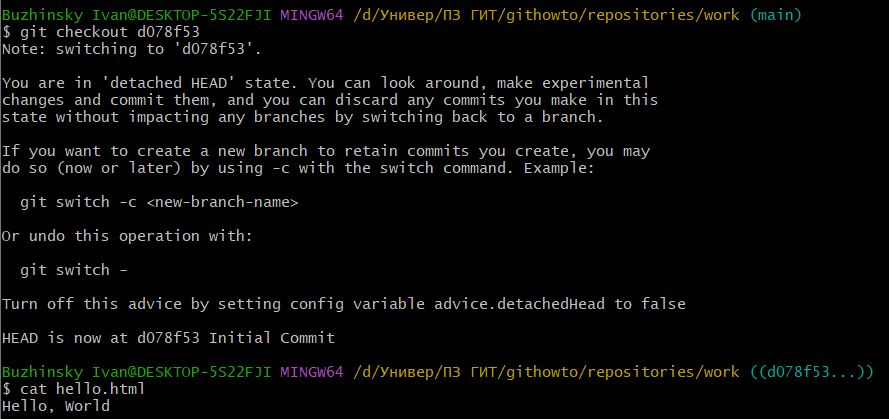


Рисунок 24 – Возвращение к первой версии файла hello.html

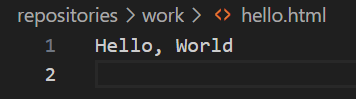


Рисунок 25 – Содержание файла hello.html

Обратите внимание, что сейчас содержимое файла hello.html — это тот самый текст, с которого мы начинали.

Чтобы вернуться к последней версии нашего кода, нам нужно переключиться на ветку по умолчанию, main. Для переключения между ветками можно воспользоваться командой switch.

git switch main

cat hello.html

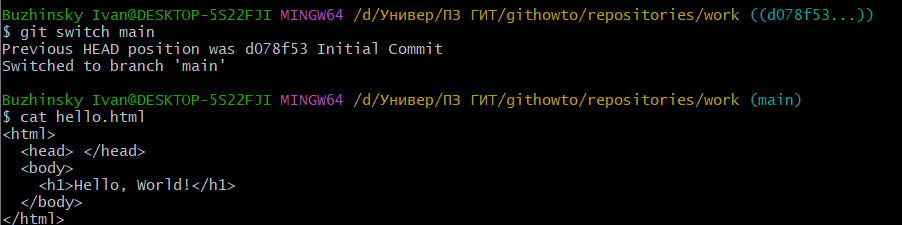


Рисунок 26 – Возвращение к последней версии в ветке main

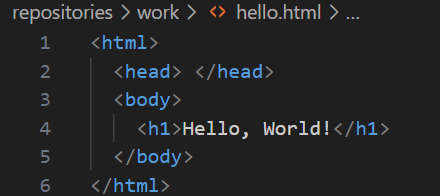


Рисунок 27 – Содержание файла hello.html

Здесь main — имя ветки по умолчанию. Переключаясь на ветку, вы попадаете на её последнюю версию.

**Задание 11. Создание тегов версий**

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.

Давайте назовем текущую версию страницы hello.html первой, то есть v1. Создайте тег первой версии:

git tag v1

git log

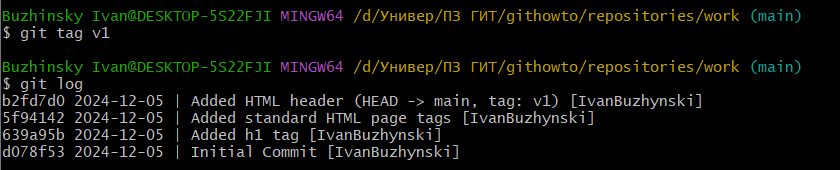


Рисунок 28 – Создание тега первой версии

Теперь текущая версия страницы называется v1.

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.

git checkout v1~1

cat hello.html

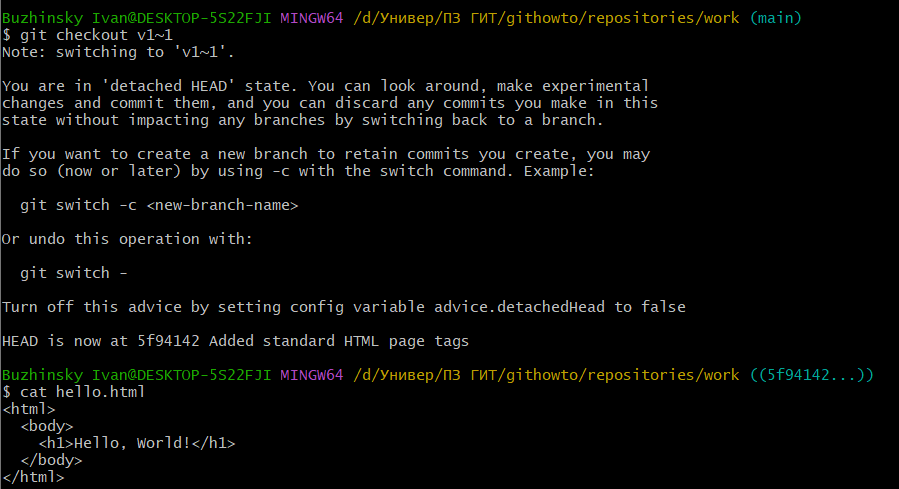


Рисунок 29 – Переключение на предыдущую версию

Это версия с тегами <html> и <body>, но еще пока без <head>. Давайте сделаем ее версией v1-beta.

git tag v1-beta

git log

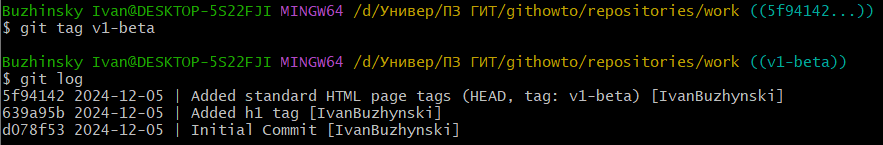


Рисунок 30 – Обозначение версии v1-beta

Теперь попробуйте попереключаться между двумя отмеченными версиями.

git checkout v1

git checkout v1-beta

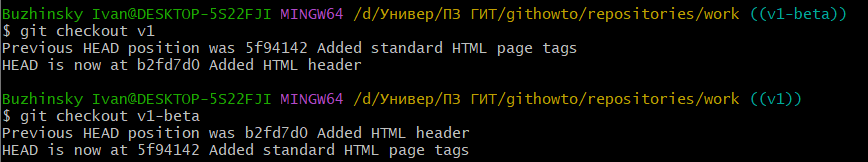


Рисунок 31 – Переключение на версию v1

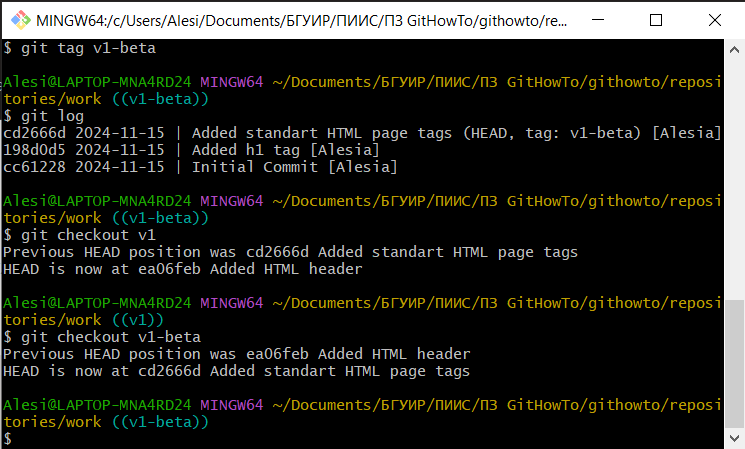


Рисунок 32 – Переключение на версию v1-beta

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

git tag

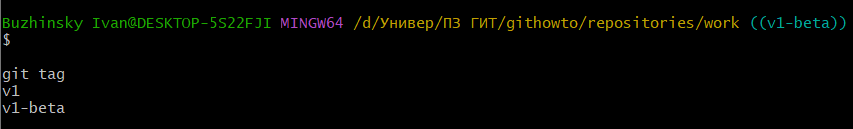


Рисунок 33 – Просмотр доступных тегов

Вы также можете посмотреть теги в логе.

git log main --all

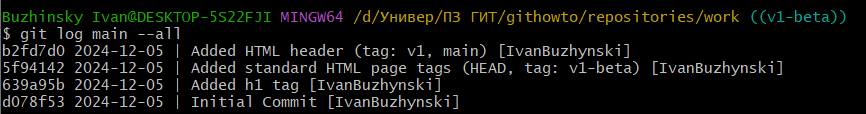


Рисунок 34 – Просмотр тегов в логе

Вы можете видеть теги (v1 и v1-beta) в логе вместе с именем ветки (main). Кроме того, метка HEAD показывает коммит, на который вы переключились (на данный момент это v1-beta).

**Задание 12. Отмена локальных изменений (до индексации)**

Убедитесь, что вы находитесь на последнем коммите ветки main, прежде чем продолжить работу.

git switch main

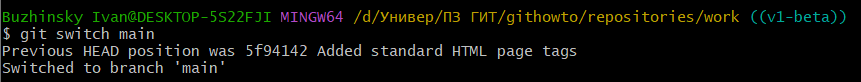


Рисунок 35 – Проверка, что мы на последнем коммите ветки main

Иногда после того как вы изменили файл в рабочей директории, вы передумали и хотите просто вернуться к тому, что уже было закоммичено. Команда checkout справится с этой задачей.

Внесите изменение в файл hello.html в виде нежелательного комментария.

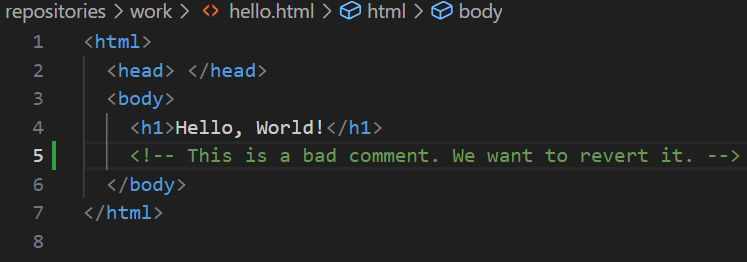


Рисунок 36 – Изменение файла hello.html

Сначала проверьте состояние рабочей директории.

git status

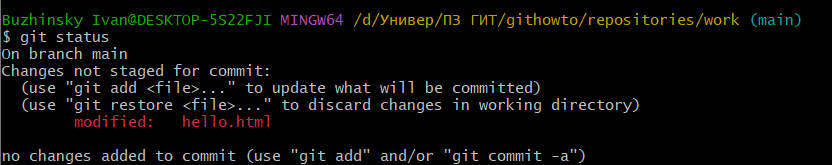


Рисунок 37 – Проверка состояния рабочей директории

Мы видим, что файл hello.html был изменен, но еще не проиндексирован.

Используйте команду checkout для переключения в версию файла hello.html в репозитории.

git checkout hello.html

git status

cat hello.html

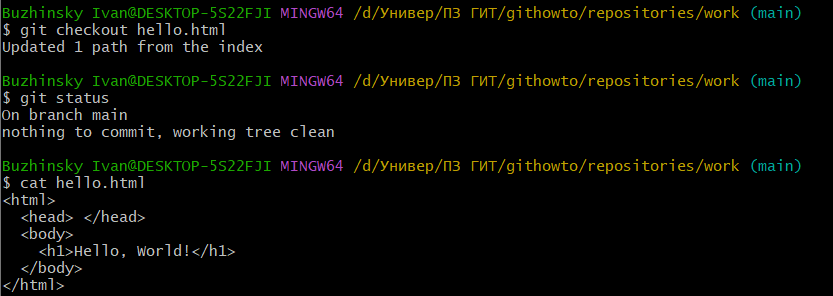


Рисунок – 38 Переключение в версию файла hello.html в репозитории

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

**Задание 13. Отмена проиндексированных изменений (перед коммитом)**

Внесите изменение в файл hello.html в виде нежелательного комментария:

<html>

<head>

<!-- This is an unwanted but staged comment -->

</head>

<body>

<h1>Hello, World!</h1>

</body>

</html>

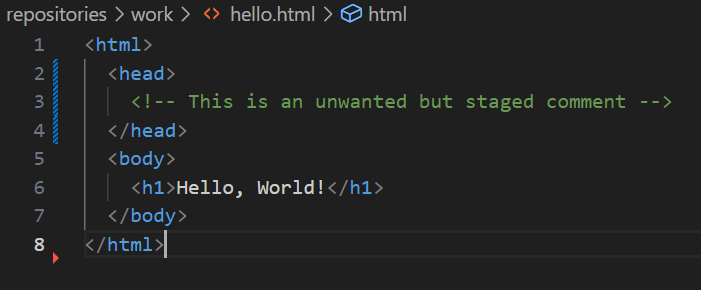


Рисунок – 39 Изменение файла hello.html

Проиндексируйте это изменение.

git add hello.html

Проверьте состояние нежелательного изменения.

git status

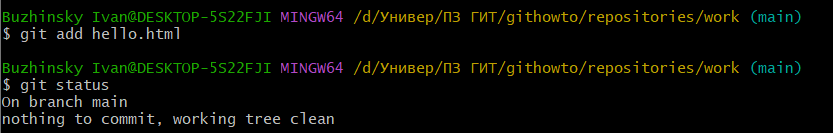


Рисунок 40 – Индексирование изменения и проверка состояния

Состояние показывает, что изменение было проиндексировано и готово к коммиту.

Команда reset сбрасывает область подготовки к HEAD. Это очищает область подготовки от изменений, которые мы только что проиндексировали.

git reset HEAD hello.html

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.

Переключитесь на версию коммита:

git checkout hello.html

git status

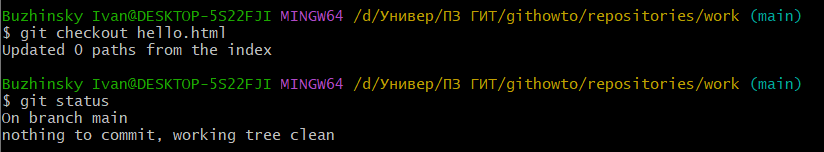


Рисунок 41 – Сброс области подготовки и убирание нежелательного комментария

Наша рабочая директория опять чиста.

**Задание 14. 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.

Измените файл hello.html на следующий.

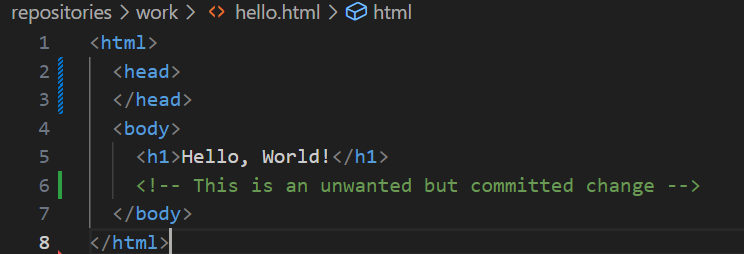


Рисунок 42 – Изменение файла hello.html

Затем выполните команды:

git add hello.html

git commit -m "Oops, we didn't want this commit"

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

git revert HEAD

Перейдите в редактор, где вы можете отредактировать коммит-сообщение по умолчанию или оставить все как есть.

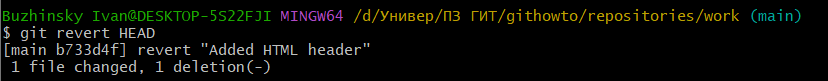


Рисунок 43 – Удаление нежелательного коммита

Так как мы отменили самый последний произведенный коммит, мы смогли использовать метку HEAD в качестве аргумента для отмены коммита. Мы можем отменить любой произвольной коммит в истории, указав его хеш.

Проверка лога показывает нежелательные и отмененные коммиты в наш репозиторий.

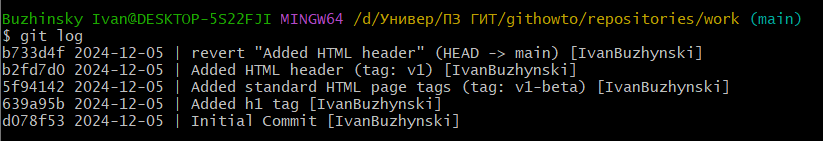


Рисунок 44 – Проверка лога

Эта техника будет работать с любым коммитом (хотя, возможно, возникнут конфликты). Она безопасна в использовании даже в публичных ветках удаленных репозиториев.

Далее давайте посмотрим на технику, которая может быть использована для удаления последних коммитов из истории репозитория.

**Задание 15. Удаление коммитов из ветки**

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.

Давайте сделаем быструю проверку нашей истории коммитов.

git log

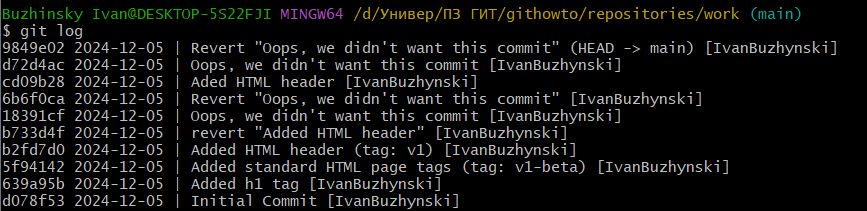


Рисунок 45 – Проверка истории коммитов

Мы видим, что два последних коммита в этой ветке - «Oops» и «Revert Oops». Давайте удалим их с помощью сброса.

Но прежде чем удалить коммиты, давайте отметим последний коммит тегом, чтобы потом можно было его найти.

git tag oops

Глядя на историю лога (см. выше), мы видим, что коммит с тегом v1 является коммитом, предшествующим ошибочному коммиту. Давайте сбросим ветку до этой точки. Поскольку ветка имеет тег, мы можем использовать имя тега в команде сброса reset (если она не имеет тега, мы можем использовать хеш коммита).

git reset --hard v1

git log

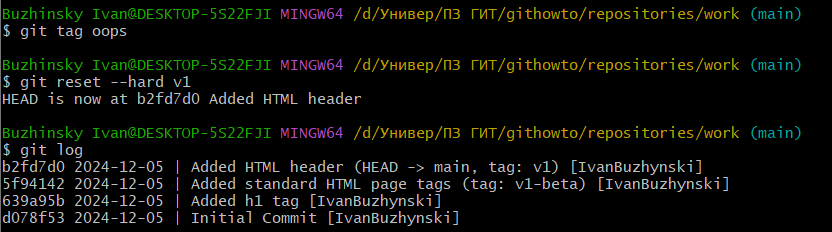


Рисунок 46 – Сброс к предыдущему коммиту

Наша ветка main теперь указывает на коммит v1, а коммитов "Revert Oops" и "Oops" в ветке уже нет. Параметр --hard указывает, что рабочая директория должна быть приведена к тому состоянию, которое соответствует HEAD-коммиту ветки.

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.

git log --all

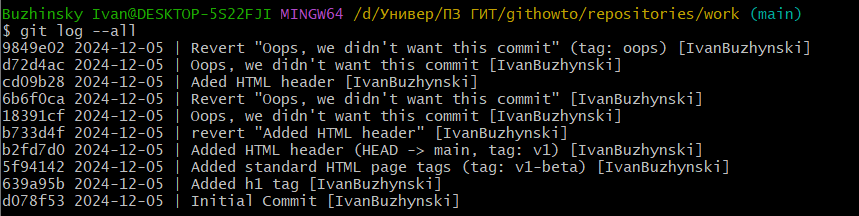


Рисунок 47 – Просмотр всех коммитов

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.

**Задание 16. 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.

git tag -d oops

git log --all

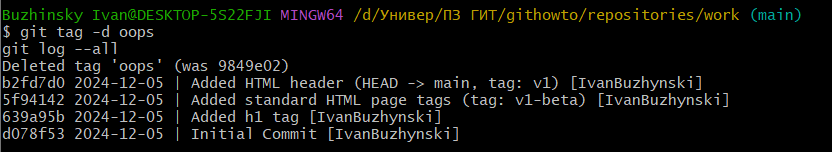


Рисунок 48 – Удаление тега oops

Тег oops больше не будет отображаться в репозитории.

**Задание 17. Внесение изменений в коммиты**

Добавьте в страницу комментарий автора.

<!-- Author: Alexander Shvets -->

<html>

<head>

</head>

<body>

<h1>Hello, World!</h1>

</body>

</html>



Рисунок 49 – Изменение файла helo.html

Затем выполните:

git add hello.html

git commit -m "Added copyright statement"

git log

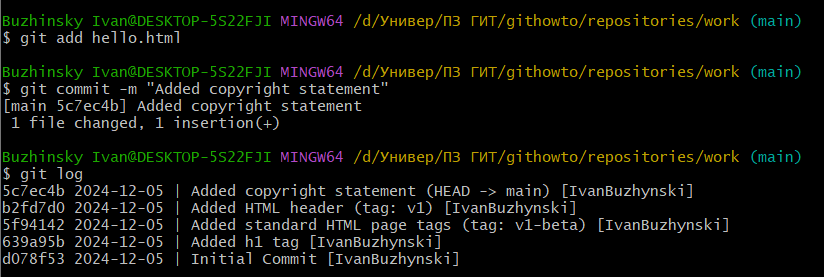


Рисунок 50 – Создание коммита

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

<!-- Author: Alexander Shvets (alex@githowto.com) -->

<html>

<head>

</head>

<body>

<h1>Hello, World!</h1>

</body>

</html>

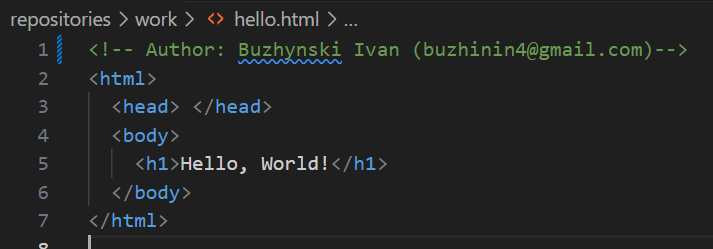


Рисунок 54 – Изменение файла hello.html

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

git add hello.html

git commit --amend -m "Added copyright statement with email"

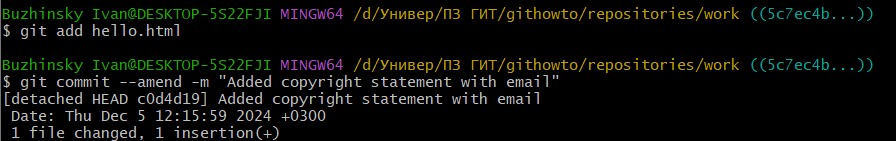


Рисунок 55 – Изменение коммита

Просмотр истории:

git log

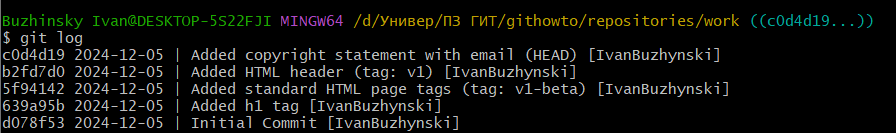


Рисунок 56 – Просмотр истории

Мы можем увидеть, что оригинальный коммит «автор» заменен коммитом «автор/email». Этого же эффекта можно достичь путем сброса последнего коммита в ветке, и повторного коммита новых изменений.

**Задание 18. Создание ветки**

Пришло время сделать нашу страницу более стильной с помощью CSS. Мы будем развивать эту возможность в новой ветке под названием style.

git switch -c style

git status

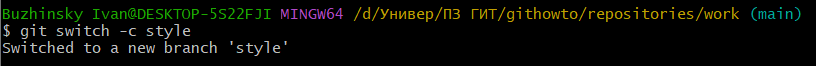


Рисунок 57 – Создание новой ветки style

Обратите внимание, что команда git status сообщает о том, что вы находитесь в ветке style.

Добавьте файл стилей style.css

touch style.css

h1 {

color: red;

}

git add style.css

git commit -m "Added css stylesheet"

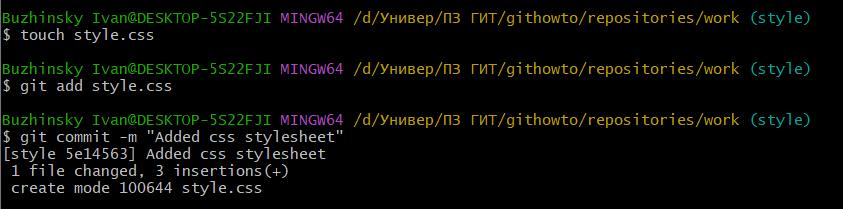


Рисунок 58 – Создание и коммит файла style.css

Измените hello.html, чтобы он использовал style.css.

<!-- Author: Alexander Shvets (alex@githowto.com) -->

<html>

<head>

<link type="text/css" rel="stylesheet" media="all" href="style.css" />

</head>

<body>

<h1>Hello, World!</h1>

</body>

</html>

git add hello.html

git commit -m "Included stylesheet into hello.html"

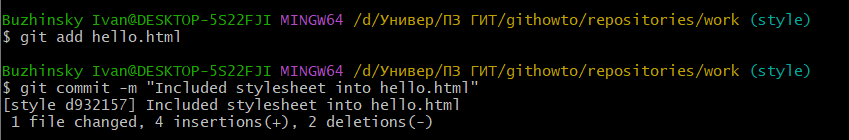


Рисунок 61 – Коммит изменений

Теперь у нас есть новая ветка под названием style с двумя новыми коммитами. Далее мы узнаем, как переключаться между ветками.

**Задание 19. Переключение веток**

Теперь в вашем проекте есть две ветки:

git log --all

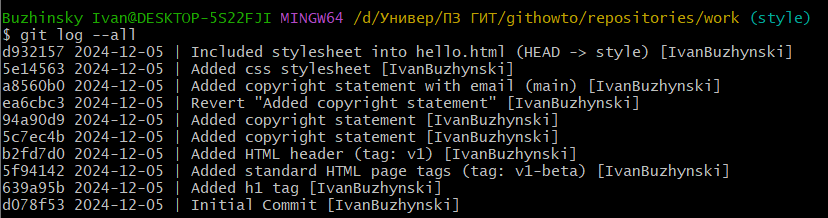


Рисунок 62 – Просмотр веток

Просто используйте команду git switch для переключения между ветками.

git switch main

cat hello.html

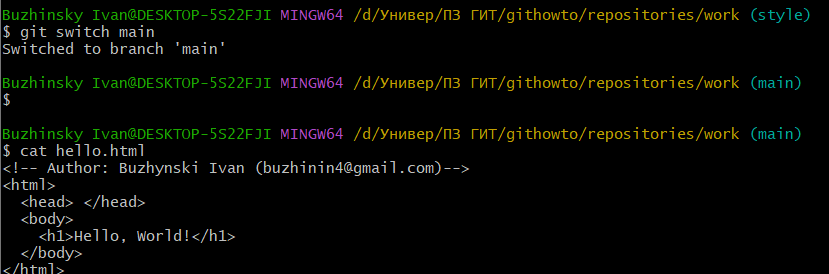


Рисунок 63 – Переключение между ветками

Теперь мы находимся в ветке main. Как видите, в hello.html нет никаких следов style.css. Не волнуйтесь, эти изменения все еще есть в репозитории, но мы не можем увидеть их из ветки main. Вернемся к ветке style:

git switch style

cat hello.html



Рисунок 65 – Возвращение к ветке style

Мы вернулись к ветке style. Как видите, наши изменения, связанные с CSS, присутствуют.

**Задание 20. Перемещение файлов**

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

git log hello.html

git log style.css

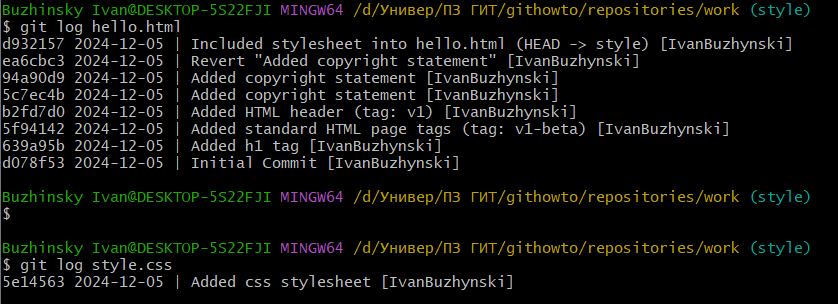


Рисунок 66 – Просмотр истории изменений в файлах hello.html и style.css

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.).

git show v1

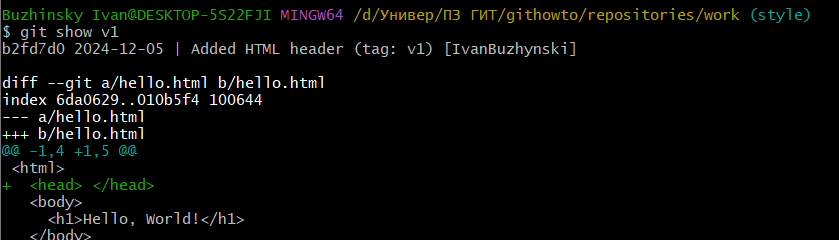


Рисунок 67 – Просмотр изменений в коммите

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

mv hello.html index.html

git status

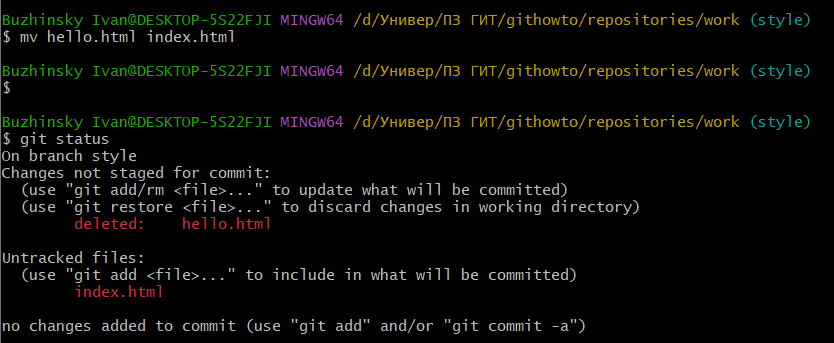


Рисунок 68 – Переименование файла hello.html в index.html

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:

git add .

git status

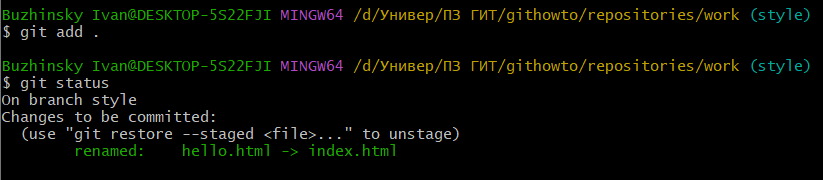


Рисунок 69 – Переименование файла в Git

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.

mkdir css

git mv style.css css/style.css

git status

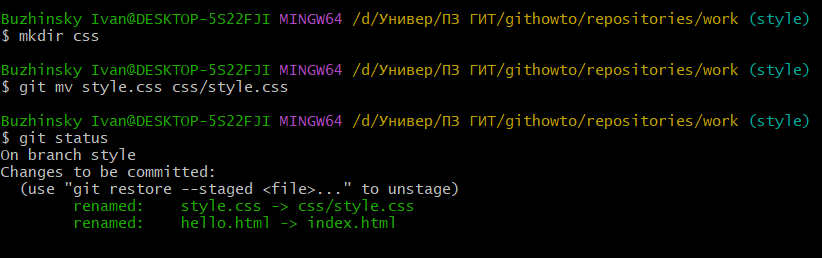


Рисунок 70 – Перемещение файла в Git

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.

git commit -m "Renamed hello.html; moved style.css"

git log css/style.css

git log --follow css/style.css

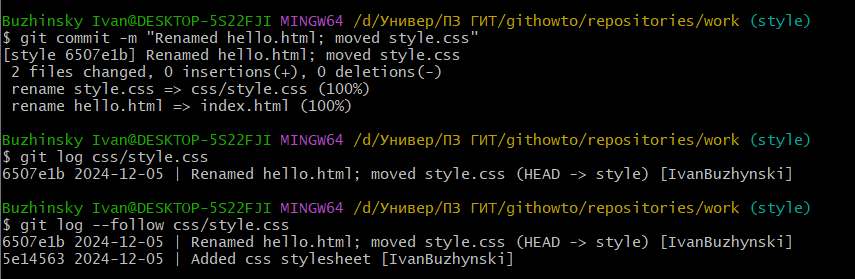


Рисунок 71 – Коммит изменений

**Задание 21. 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.

This is the Hello World example from the GitHowTo tutorial.

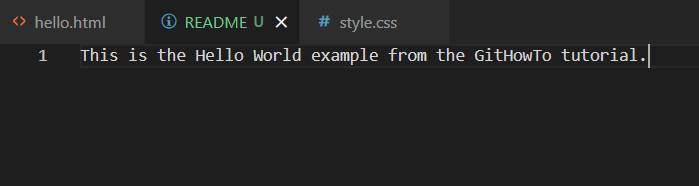


Рисунок 72 – Созданный файл README

Сделайте коммит файла README в ветку main. В настоящее время мы находимся в ветке style. Файл README не является частью этой ветки, поэтому перед коммитом мы должны переключиться на ветку main.

git switch main

git add README

git commit -m "Added README"

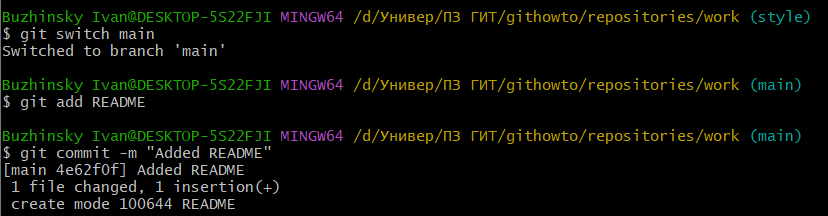


Рисунок 73 – Коммит файла README

**Задание 22. Просмотр отличающихся веток**

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

git log --all --graph

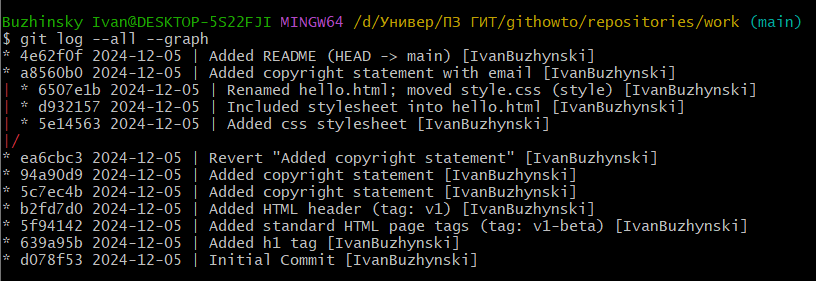


Рисунок 74 – Просмотр текущих веток

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».

**Задание 23. Слияние**

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

git switch style

git merge main

git log --all --graph

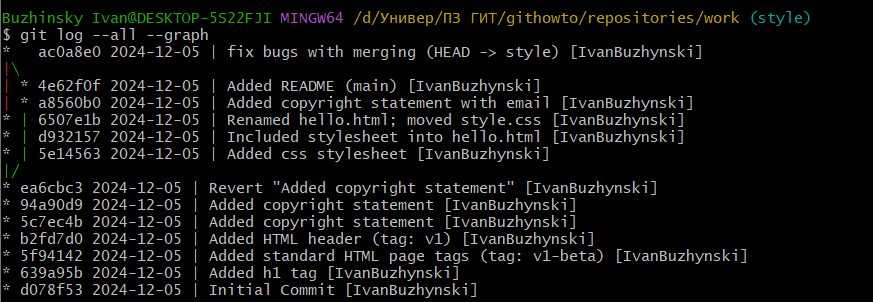


Рисунок 75 – Слияние веток

Путем периодического слияния ветки main с веткой style вы можете переносить из main любые изменения и поддерживать совместимость изменений style с изменениями в основной ветке.

Но что если изменения в ветке main конфликтуют с изменениями в style?

**Задание 24. 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.

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

git switch main

<!-- Author: Alexander Shvets (alex@githowto.com) -->

<html>

<head>

<title>Hello World Page</title>

</head>

<body>

<h1>Hello, World!</h1>

<p>Let's learn Git together.</p>

</body>

</html>

git add hello.html

git commit -m "Added meta title"

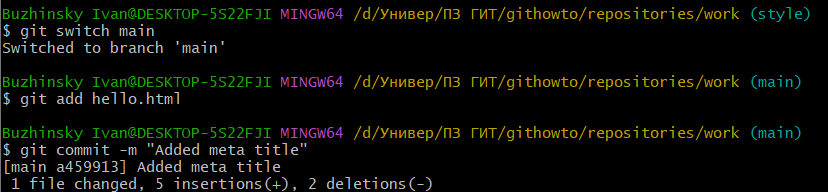


Рисунок 77 – Создание коммита

Просмотр веток:

git log --all --graph

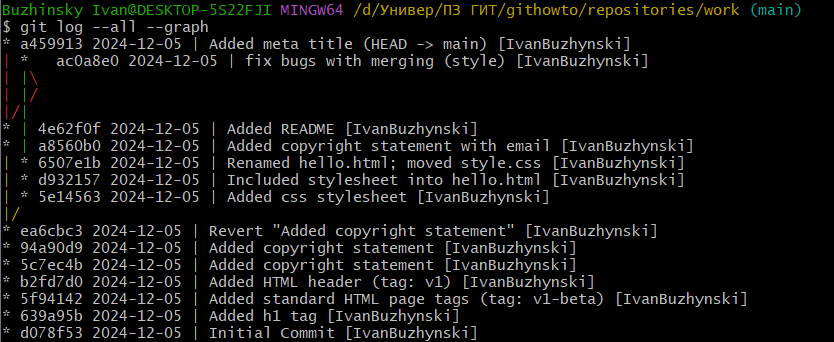


Рисунок 78 – Просмотр веток

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.

Последнее изменение в main конфликтует с некоторыми изменениями в style. На следующем шаге мы решим этот конфликт.

**Задание 25. Разрешение конфликтов**

Давайте вернемся в ветку style и сольем в нее все последние изменения из ветки main.

git switch style

git merge main

Похоже, что у нас возник конфликт. Ничего удивительного! Посмотрим, что скажет по этому поводу Git:

git status

Если открыть файл index.html, то можно увидеть:

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.

git merge --abort

git status

После небольшой медитации мы готовы к разрешению конфликта. Давайте снова запустим объединение.

git merge main

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:

<!-- Author: Alexander Shvets (alex@githowto.com) -->

<html>

<head>

<title>Hello World Page</title>

<link type="text/css" rel="stylesheet" media="all" href="style.css" />

</head>

<body>

<h1>Hello, World!</h1>

<p>Let's learn Git together.</p>

</body>

</html>

Закоммитьте разрешенный конфликт:

git add index.html

git commit -m "Resolved merge conflict"

git status

git log --all --graph

Давайте посмотрим на текущее состояние нашего хранилища и убедимся, что все в порядке:

git status

git log --all --graph

**Задание 26. 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.

**Задание 27. 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.

git switch style

git log --graph

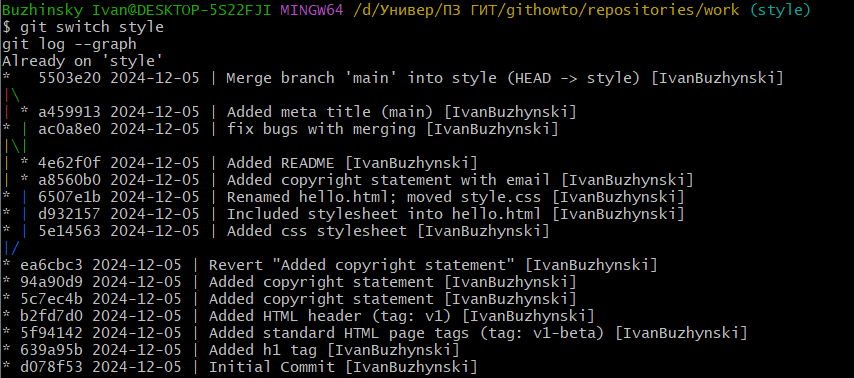


Рисунок 87 – Возврат к ветке style

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.

git reset --hard HEAD~2

Теперь проверим историю ветки style. В истории не должно быть коммитов слияния.

git log --graph

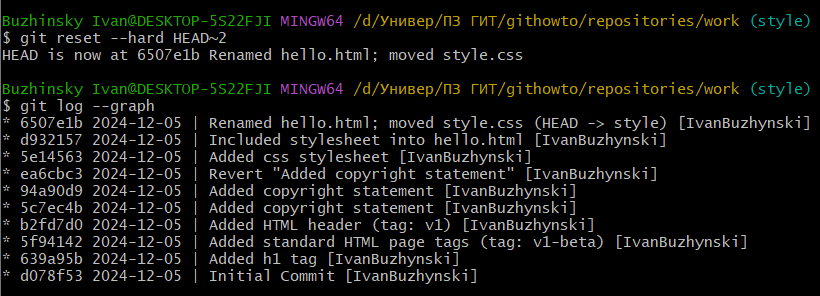


Рисунок 88 – Проверка ветки style

**Задание 28. 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.

Перебазируем ветку style на ветку main:

git switch style

git rebase main

git status

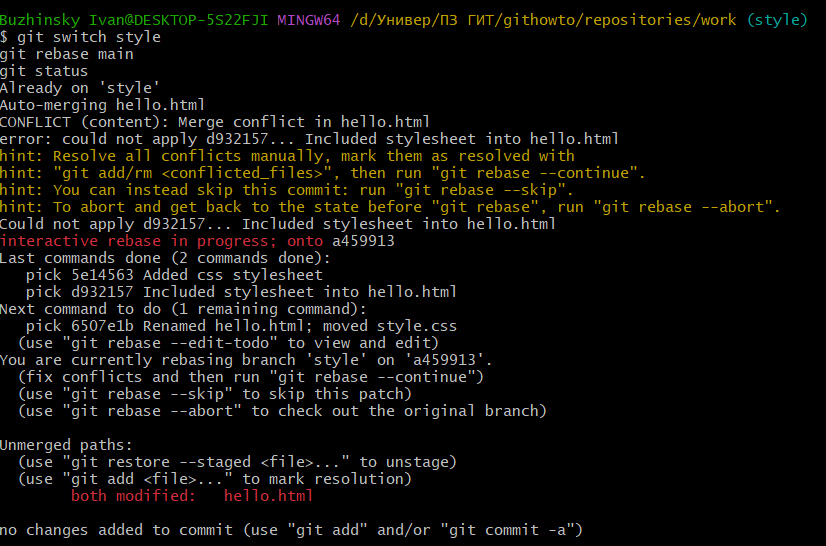


Рисунок 89 – Перебазирование ветки style на ветку main

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.

Сам конфликт может быть разрешен тем же способом, что и предыдущий. Сначала мы изменим файл hello.html, чтобы он соответствовал нашим ожиданиям.

<!-- Author: Alexander Shvets (alex@githowto.com) -->

<html>

<head>

<title>Hello World Page</title>

<link type="text/css" rel="stylesheet" media="all" href="style.css" />

</head>

<body>

<h1>Hello, World!</h1>

<p>Let's learn Git together.</p>

</body>

</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.

git add .

git rebase --continue

Здесь, скорее всего, Git снова откроет редактор, чтобы позволить нам изменить текст коммита. Мы можем оставить текст без изменений. После сохранения изменений Git завершит процесс rebase, и мы cможем выполнить следующие команды:

git status

git log --all --graph

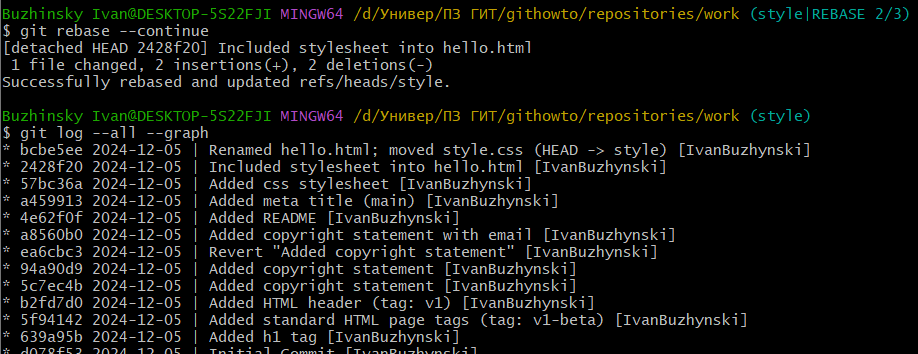


Рисунок 92 – Заверение rebase и просмотр состояния Git

Конечный результат перебазирования очень похож на результат слияния. Ветка style в настоящее время содержит все свои изменения, а также все изменения ветки main. Однако, дерево коммитов значительно отличается. Дерево коммитов ветки style было переписано таким образом, что ветка main является частью истории коммитов. Это делает цепь коммитов линейной и гораздо более читабельной.

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.

**Задание 29. Слияние в ветку main**

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

git switch main

git merge style

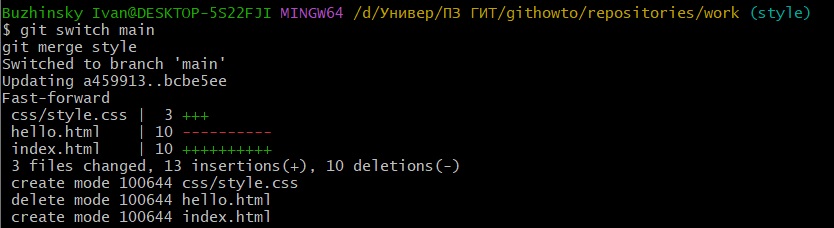


Рисунок 93 – Слияние ветки style в main

Поскольку последний коммит в main предшествует последнему коммиту ветки style, Git может выполнить ускоренное слияние, просто переместив указатель ветки вперед, на тот же коммит, что и ветка style.

При ускоренном слиянии конфликты не возникают. Кроме того, при ускоренном слиянии не создается фиксация слияния.

Просмотрите логи:

git log --all --graph

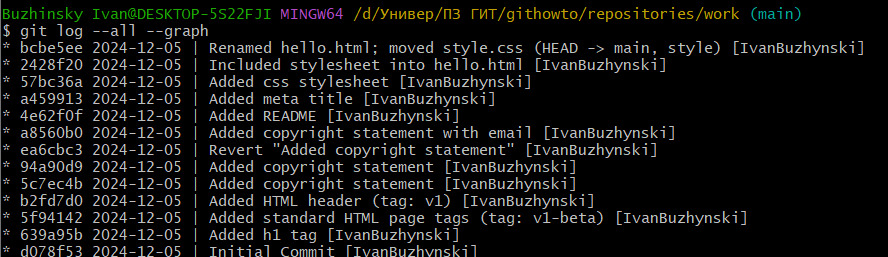


Рисунок 94 – Просмотр логов

Теперь ветки style и main идентичны.

**Часть II: Несколько репозиториев**

**Задание 30. Клонирование репозиториев**

Если вы работаете в команде, последующие 8 уроков довольно важны в понимании, т.к. вы почти всегда будете работать с клонированными репозиториями.

Перейдите в директорию repositories:

cd ..

pwd

ls

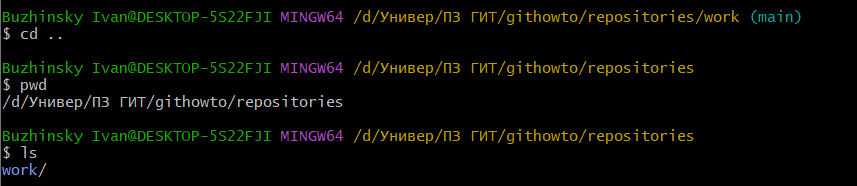


Рисунок 95 – Переход в директорию repositories

В этот момент вы должны находиться в директории repositories. Здесь должен быть единственный репозиторий под названием work.

Давайте создадим клон репозитория.

git clone work home

ls

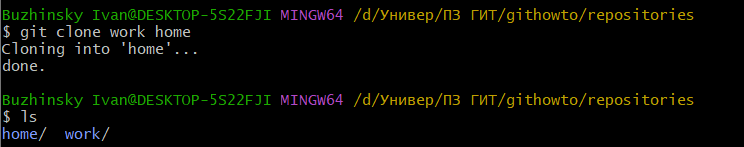


Рисунок 96 – Создание клона репозитория work

В вашем списке репозиториев теперь должно быть два репозитория: оригинальный репозиторий work и клонированный репозиторий home.

**Задание 31. Просмотр клонированного репозитория**

Давайте взглянем на клонированный репозиторий

cd home

ls

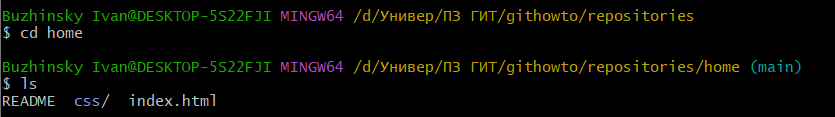


Рисунок 97 – Просмотр клонированного репозитория

Вы увидите список всех файлов на верхнем уровне оригинального репозитория README, index.html и css).

Просмотрите историю репозитория:

git log --all

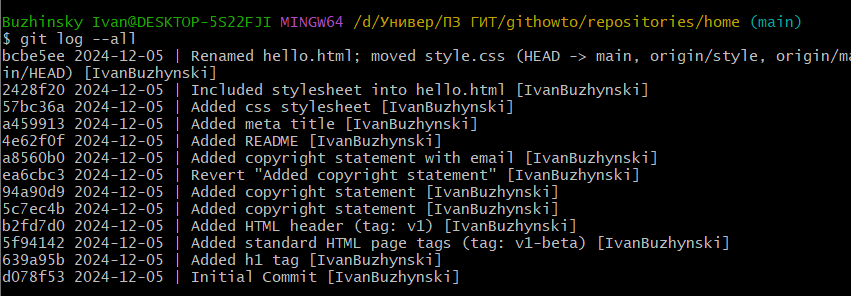


Рисунок 98 – Просмотр истории клонированного репозитория

Вы увидите список всех коммитов в новый репозиторий, и он должен совпадать с историей коммитов в оригинальном репозитории. Единственная разница должна быть в названиях веток.

Вы увидите ветку main (HEAD) в списке истории. Вы также увидите ветки со странными именами (origin/main, origin/style и origin/HEAD). Мы поговорим о них чуть позже.

**Задание 32. Что такое origin?**

Цель – узнать об именах удаленных репозиториев.

git remote

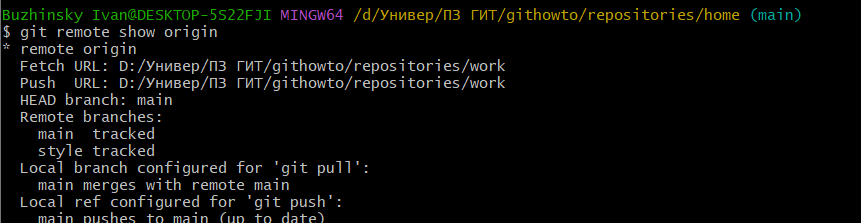


Рисунок 99 – Просмотр удаленного репозитория

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:

git remote show origin

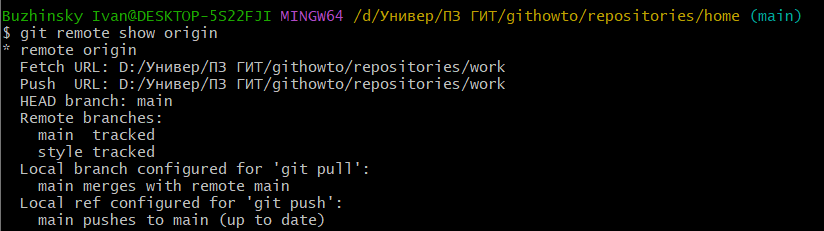


Рисунок 100 – Просмотр информации о репозитории

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).

**Задание 33. Удаленные ветки**

Давайте посмотрим на ветки, доступные в нашем клонированном репозитории.

git branch



Рисунок 101 – Просмотр доступных веток

Как мы видим, в списке только ветка main. Где ветка style? Команда git branch выводит только список локальных веток по умолчанию.

Для того чтобы увидеть все ветки, попробуйте следующую команду:

git branch -a

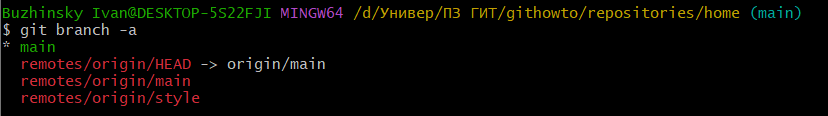


Рисунок 102 – Просмотр всех веток

Git выводит все коммиты в оригинальный репозиторий, но ветки в удаленном репозитории не рассматриваются как локальные. Если мы хотим иметь собственную ветку style, мы должны сами ее создать. Через минуту вы увидите, как это делается.

**Задание 34. Изменение оригинального репозитория**

Внесите изменения в оригинальный репозиторий work:

cd ../work

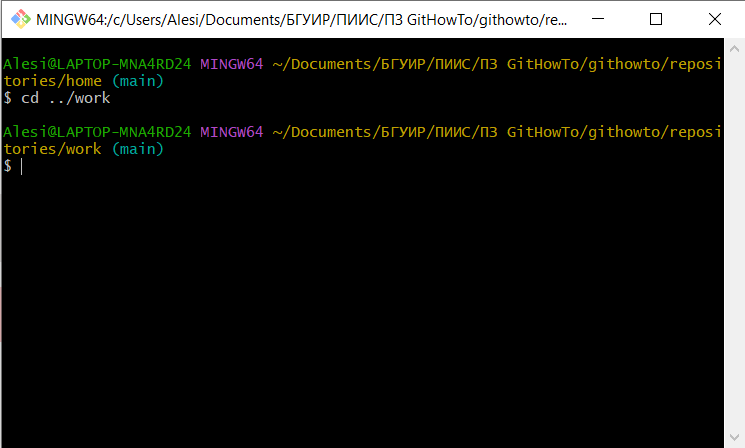


Рисунок 103 – Переход в репозиторий work

Внесите следующие изменения в файл README:

This is the Hello World example from the Git tutorial.

(changed in origin)

Теперь добавьте это изменение и сделайте коммит:

git add README

git commit -m "Changed README in original repo"

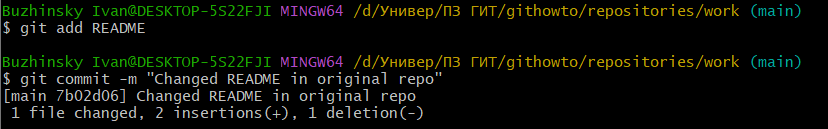


Рисунок 105 – Добавление изменения в Git и коммит

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

**Задание 35. Подтягивание изменений**

Цель – научиться подтягивать изменения из удаленного репозитория.

cd ../home

git fetch

git log --all

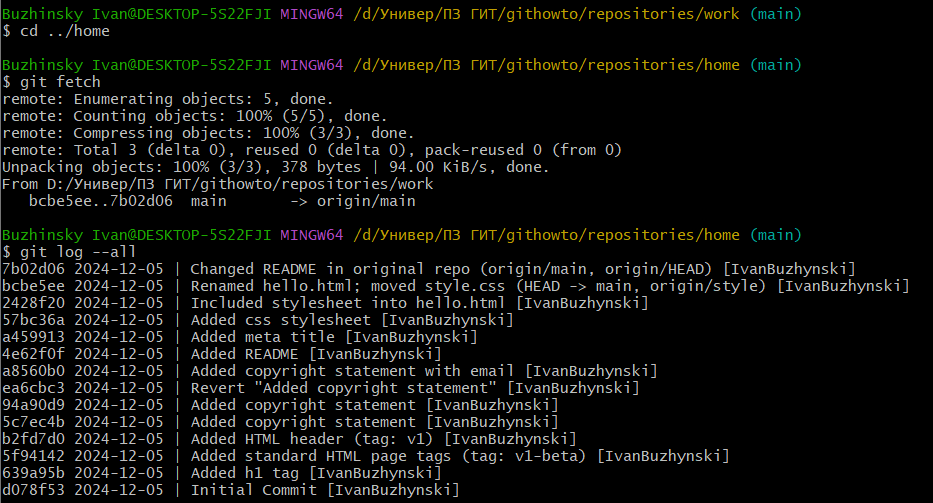


Рисунок 106 – Подтягивание изменений из удаленного репозитория

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.

Мы можем продемонстрировать, что клонированный файл README не изменился.

cat README



Рисунок 107 – Просмотр README

Как видите, никаких изменений.

**Задание 36. Слияние подтянутых изменений**

Слейте подтянутые изменения в локальную ветку main.

git merge origin/main

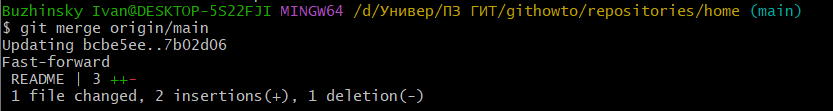


Рисунок 108 – Слияние подтянутых изменений в локальную ветку main

Еще раз проверьте файл README. Сейчас мы должны увидеть изменения.

cat README

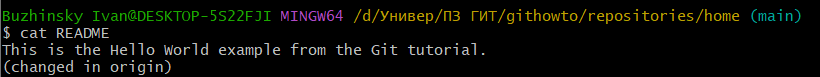


Рисунок 109 – Проверка файла README

Вот и изменения. Хотя команда git fetch не сливает изменения, мы можем вручную слить изменения из удаленного репозитория.

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.

git pull

...эквивалентна следующим двум шагам:

git fetch

git merge origin/main

**Задание 37. 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.

Добавьте локальную ветку, которая отслеживает удаленную ветку:

git branch --track style origin/style

git branch -a

git log --max-count=2

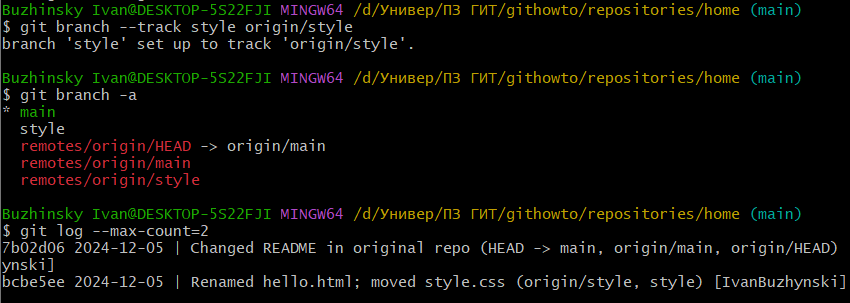


Рисунок 110 – Добавление локальной ветки для отслеживания удаленной

Теперь мы можем видеть ветку style в списке веток и логе.

**Задание 38. 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.

Создайте чистый репозиторий:

cd ..

git clone --bare work work.git

ls work.git

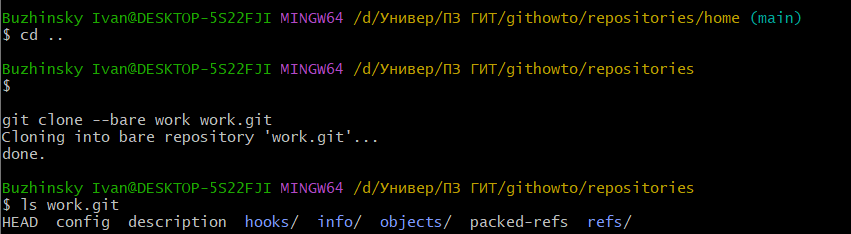


Рисунок 111 – Создание чистого репозитория

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.

**Задание 39. Добавление удаленного репозитория**

Давайте добавим репозиторий work.git к нашему оригинальному репозиторию.

cd work

git remote add shared ../work.git

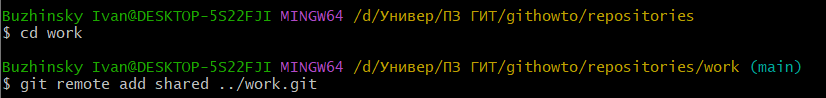


Рисунок 112 – Добавление удаленного репозитория

**Задание 40. 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:

This is the Hello World example from the Git tutorial.

(changed in the origin and pushed to shared)

git switch main

git add README

git commit -m "Added shared comment to readme"

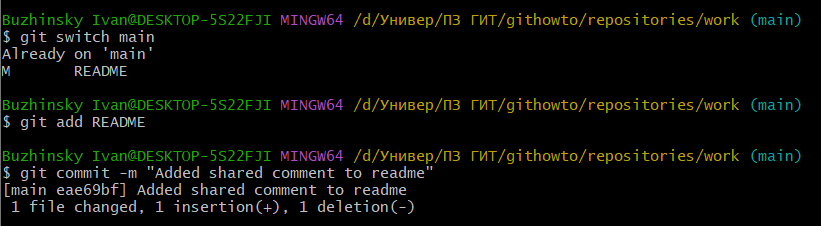


Рисунок 114 – Коммит изменений

Теперь отправьте изменения в общий репозиторий.

git push shared main

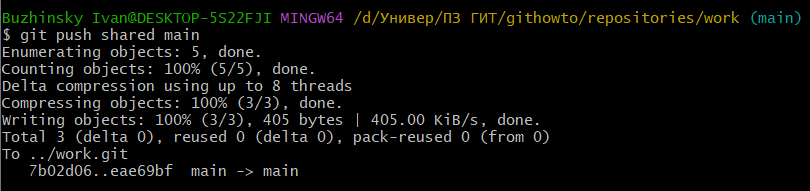


Рисунок 115 – Отправка изменений в общий репозиторий

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

**Задание 41. Подтягивание общих изменений**

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

cd ../home

Продолжите с...

git remote add shared ../work.git

git branch --track shared main

git pull shared main

cat README

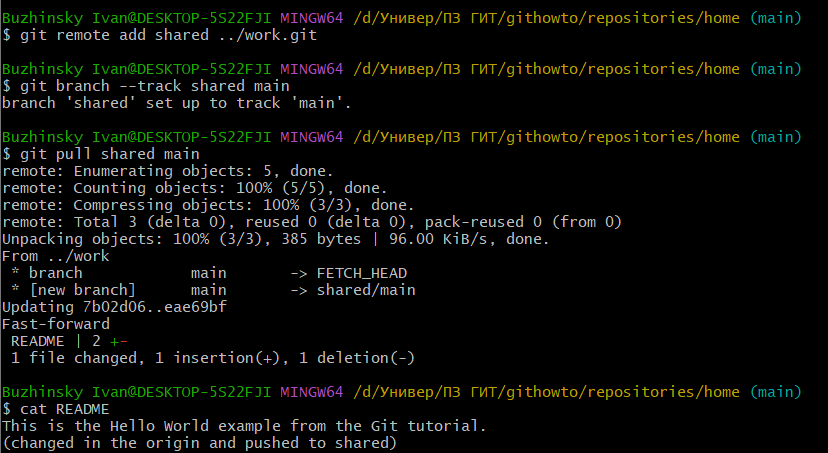


Рисунок 116 – Подтягивание общих изменений

**Задание 42. 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.

Запуск Git-сервера:

# (From the "repositories" directory)

git daemon --verbose --export-all --base-path=.

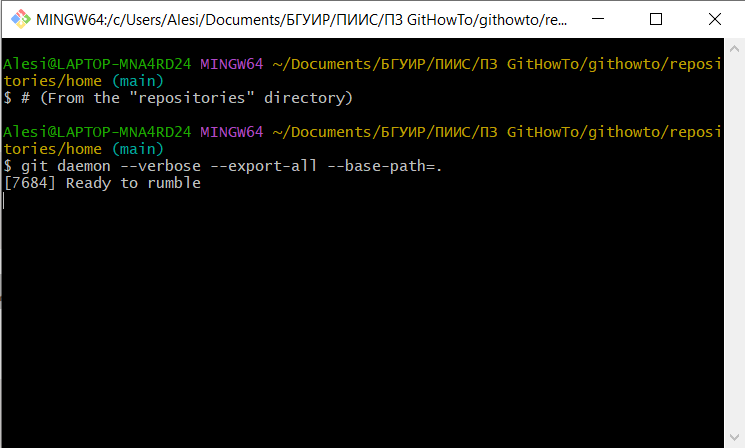


Рисунок 117 – Запуск Git-сервера

Теперь в отдельном окне терминала перейдите в вашу директорию repositories:

# (From the "repositories" directory)

git clone git://localhost/work.git network\_work

cd network\_work

ls

Вы увидите копию проекта work.

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.

На этом этапе вам открываются безграничные возможности. Смелее! Возьмите в аренду сервер, купите доменное имя, разместите на этом сервере свои репозитории и наслаждайтесь своим личным GitHub!

Если серьезно, то вы можете самостоятельно разместить свой личный сервер GitLab. Этот продукт бесплатный и с открытым исходным кодом.

*Спасибо за использование GitHowTo! Надеюсь, вам было интересно.*

**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.