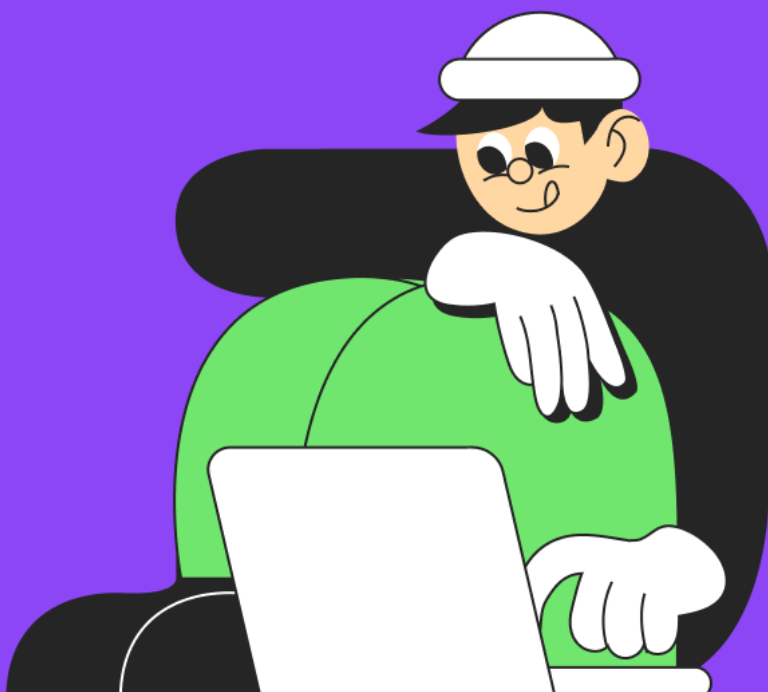
GeekBrains

Установка и настройка системы контроля версий



Оглавление

[00:01:36]

Вступление

Добрый день, дорогие друзья.

Рад вас приветствовать на второй лекции нашего курса «Введение в контроль версий». снова я — Ильнар Шафигуллин. Думаю, что вы уже знаете меня и запомнили. Сегодня мы пройдём немного дальше, будем заниматься ветвлениями. Посмотрим, что позволяет делать гид, помимо того, чтобы сохранять наше состояние, переключаться между ними. Но прежде, вспомним, а чему же мы научились на первой лекции?

[00:02:09]

На прошлой лекции

Сохранения

Мы умеем делать сохранения. Они называются коммиты. Коммиты — это, устраивающие нас версии, между которыми можно переключаться. В самом начале мы разбирали самый простой пример, сайт. У нас есть какая-то рабочая версия примерно на 30 декабря. Чтобы вносить изменения, необходимо сохранить файл в отдельную папку, тогда мы сможем к нему вернуться. Далее появляются новые версии, мы их сохранили и переходим дальше. Этот пример уже умеем реализовывать с помощью Git. В нашем случае сохранения — это коммиты, с которым мы работали. Когда мы пишем git commit -m и указываем в кавычках какой-то текст, то можно указать, что это была рабочая версия сайта на такую-то дату. К каждому сохранению можно указать своё сообщение. Если нужно увидеть все коммиты, то набираем команду git log, тогда мы увидим список наших коммитов (сохранений). И сможем между ними переключаться.

[00:03:26]

Команда git init

Итак, вспомним основные команды, которые позволяют организовать вот такую работу. Пока всё у нас локально, только на нашем компьютере. Интернет мы практически не используем и делаем всё только в рамках нашего локального компьютера. У нас есть набор команд, есть команда git init, вот она пишется таким образом. Напомню, когда мы указываем в терминале, мы сначала пишем, какую команду необходимо вызвать (запустить). Далее указываем пробел, чтобы терминал понял, что это название программы, а вот это уже что-то следующее. Мы всегда отделяем пробелами. Затем указываем, какую команду необходимо программе выполнить. В итоге запустится программа Git и внутри неё выполняется git init. Напоминаю, это сокращение от initialization. Таким образом, что делает программа Git? Она в папке, из которой мы это запускаем, создаст репозиторий (скрытая папка). Именно в скрытой папке будет происходить вся магия, пока мы работаем с системой контроля версий. Так вот, чтобы стартануть работу в нашей папке, мы запускаем команду git init внутри этой папки.

[00:04:39]

Команда add

Идём дальше. Если мы захотим, чтобы шёл контроль версий файлов и программа Git их контролировала, то нужно добавить эти файлы как отслеживаемые. Это делается с помощью команды add внутри программы Git. Мы можем в терминале вызвать нашу программу, обратиться к программе Git, передать ей команду add (с англ. добавь) и указать файл, который Git необходимо будет отслеживать. Но здесь есть ряд напоминаний, о которых не стоит забывать. Разделителями, также являются пробелы. Сами файлы указывают с расширением, если они есть, то есть указывается полное название файла. И в этом примере расширение файла txt, а мы на лекции работали с Markdown, и расширение было .MD. Напомню, что для Git не имеет никакого значения, какое расширение у файла он может работать с совершенно любыми. Главное, чтобы название было указано полностью. Недостаточно указать просто имя файла. Необходимо указать имя и расширение. Таким образом, Git сможет однозначно найти тот файл, за которым ему необходимо будет следить.

[00:05:51]
Команда commit

Что мы делаем дальше, после того как начали отслеживать файл? Когда мы в файл внесли какие-то изменения, сохранили их обязательно. Напоминаю, что Git работает только с записанными на диск файлами, а не с находящимися в работе. Можно зафиксировать его состояние, сделать сохранение, как в компьютерных играх, мы приводили такой пример. Для этого мы вновь обращаемся в терминале к нашей программе Git. И ей передаём команду commit (с англ. фиксация, зафиксировать). Мы хотим сохранить наше текущее состояние, зафиксировать наше текущее состояние. И далее можем указать некоторое сопроводительное сообщение. Вот идёт -m и мы указываем комментарий, который необходимо к этому сохранению добавить. Чтобы в дальнейшем мы, открыв список всех наших сохранений, смогли понять, что же было в том или ином сохранении.

[00:06:52]
Команда git diff

Что дальше? Когда у нас есть несколько зафиксированных состояний, мы можем между ними переключаться. Более того, можно посмотреть разницу между текущим и зафиксированным состоянием файла. Например, мы сделали commit. После внесли какие-то изменения, и можем с помощью команды git diff (от difference — разница) посмотреть, а чем же отличается файл, отредактированный сейчас, от файла, сохранённого с помощью git commit. Команда git diff может показать разницу очень наглядно. Если мы строки удалили, то в отображении увидим их красного цвета с минусами. А если добавили, они будут зеленого цвета и с плюсами. Таким образом, можно посмотреть, что же мы сделали файлом относительно предыдущего сохранения (commit).

[00:07:51]

Команда git log

Команда git log позволяет вывести список всех имеющихся коммитов, которые мы сделали в этой ветке. Это я сейчас уже забегаю, немного вперёд, с ветками будем сегодня работать, запомните об этом. Итак, git log позволяет вывести список всех ваших коммитов. Точно так же мы обращаемся к программе Git. Передаём ей некоторые параметры, в этом случае параметр log – это журнал всех наших действий, и видим действия, которые были совершены. Здесь указано время, дата совершения коммита, кто коммит сделал и сопроводительный текст. То есть мы можем понять, а что же мы хотели запомнить, когда фиксировали это состояние. И есть ещё коммит. После слова commit указан набор букв и символов. Это так называемый хэш код. Если вы захотите переключиться с одного коммита на другой, то в Git необходимо будет указывать именно этот набор символов. Мы в прошлый раз разбирали, что не обязательно копировать всю стройку, достаточно первые 4,5,6 символов, чтобы Git смог понять, какой коммит имеется в виду.

[00:09:08]
Команда git checkout

Далее у нас команда git checkout, которая позволяет переходить между разными коммитами. Это вариант, который мы сегодня будем использовать. Здесь указывается ветка, на которую будем переключаться. Об этом и поговорим сегодня. А в предыдущей версии мы вместо branch_name указывали набор символов (коммит), на который нам необходимо переключаться. Когда мы обращаемся к программе Git, передаём ей команду checkout (перейди) и указываем номер коммита, то переходим в состояние файлов, коммит, в котором они тогда были.

[00:09:52]
Разберём пример с курсовой или книгой

Теперь посмотрим, а что ещё нам нужно из примеров, разобранных на 1 лекции? Вот вариант с сайтом, когда мы делаем backup, просто сохраняем разные версии сайта в отдельной папке, мы его разобрали. С помощью имеющихся команд мы можем это реализовывать.

Но, ещё был пример с курсовой или книгой, которую мы пишем. Например, когда вы пишете черновики книги и есть человек, допустим, научный руководитель или главный редактор. Когда вы пишете книгу, тот, кто проверяет уже написанное, вносит какие-то корректировки. Вы ведёте совместную деятельность. Такой вариант взаимодействия пока ещё, мы не умеем организовывать, но к нему немного приблизимся.

Приблизиться можно следующим образом. Вспомните, учёбу в школе. Скорее всего, вы писали сначала в черновике, а потом всё переносили в чистовик. Иначе вашу работу могли не принять или поставить не очень хорошую оценку. У меня в детстве был случай. Учительница математики любила ставить мне тройку, даже если всё решено правильно, но написано было неаккуратно, у меня почерк не очень хороший. И мне приходилось сначала всё решать в черновике, а потом переписывать на чистовик.

Такой процесс работы, когда есть чистовики и черновики, можно реализовать в Git с помощью веток. У нас есть какая-то ветвь с черновиками и другая ветвь с чистовиками. Можно представить это, например, как отдельные папки. В одной папке будут лежать черновики, а в другой папке будут лежать файлы, которые уже приняты нашим коллегой. Это организуется с помощью веток. Мы сейчас посмотрим это на практике. И аккуратно со всем разберёмся.

[00:11:58]

Практическая часть

Перейдём к практической части. Познакомимся с ветками на практике. Для этого я создам новую папку, чтобы повторить процесс инициализации Git. Процесс создания первых файлов с нуля, как мы смотрели на курсе умения учиться. Лишних касаний не бывает, они только позволяют лучше настроить нейронные связи, чтобы мы лучше осознали, что делаем и это стало выполняться автоматически.

Я создам папку на рабочем столе, назову её lesson 2. В Visual Studio Code открою эту папку. Всё так же, как мы делали раньше. Выбираем рабочий стол (стрелочка) lesson 2. И запускаем терминал, он нам сразу пригодится. Кстати, в комментариях к первой лекции я читал, что работающие на Windows, хотели бы выполнять команды для терминала, но почему-то в Visual Studio Code это не работает. Надо понимать, что Visual Studio Code не предоставляет полноценный терминал, она является лишь оболочкой, предоставляемой операционной системой. Если мы говорим про Mac или Linux, то у нас есть полноценный терминал со всеми командами, которые некоторые из вас уже знают. Если же мы говорим про работу в ОС Windows, то там построено всё немного иначе, там командная строка, а не терминал. И набор команд, которые необходимо указывать, немного отличается. Поэтому если вы работаете Windows, то терминал Visual Studio Code достаточно для того, чтобы настроить работу системы контроля версий, но это не является полноценным терминалом. В том смысле, который люди понимают под этим словом.

[00:13:58]

Создаём новый файл

Мы сегодня будем снова писать инструкцию для Markdown, но сделаем её более полной, красивой. Собственно, создаём файл и называем "Markdown instruction.md", надеюсь, в этот раз написал без ошибок. Пользоваться будем языком разметки Markdown.

Нам необходимо инициализировать наш репозиторий, вводим команду git init. Появляется новый репозиторий, смотрим на статус (git status). И видим, что у нас есть не отслеживаемый файл. Всё как на предыдущих занятиях. Мы вводим git add, указываем наш файл. Смотрим, ещё раз на статус. Видим, теперь файл отслеживается, и он изменён. Теперь можем создать свой первый commit. Сделаем git commit -m "initial commit", это сообщение довольно типовое при работе системой контроля версий. Это самый первый коммит, с которого мы начинаем. Вот мы только инициализировали репозиторий, и это самый первый коммит, который мы сделали. Его принято называть initial commit. Запускаем, у нас появился 1 коммит. Взглянем на лог. Вводим git log и видим, что у нас этот коммит уже появился.

[00:15:45]

Нюанс начала работы с Git

Кстати, в первой лекции не у всех получилось сразу начать работать с Git, это связано с тем, что первые настройки Git требует, чтобы вы представились Git. Нам необходимо указать git config и ваше имя, git config и ваш email. Но на первом семинаре мы это уже разобрали. Думаю, сейчас у всех уже всё получается.

Сегодня будем делать небольшие паузы, чтоб вы смогли догнать меня и параллельно со мной выполнять все действия. Но сейчас мы далеко не убежали, потому что пауза можно не делать. Главное — мы создали файл, инициализировали репозиторий и при этом сделали наш первый коммит.

[00:16:28]

Создаём заголовки

Зададим структуру файла, как мы хотели бы видеть нашу инструкцию. Создадим сначала набор заголовков. Это мы уже умеем. Один большой заголовок и назовём наш файл «Инструкция для работы с Markdown». Дальше сделаем набор заголовков, которые нам хотелось бы видеть в этом файле. Например, первый подзаголовок будет «Выделение текста». Следующий подзаголовок «Списки». Дальше «Работа с изображениями». Помимо этого, в Markdown могут быть «Ссылки». Кроме этого, Markdown позволяет «Работать с таблицами». Могут быть «Цитаты». На самом деле Markdown позволяет больше, чем мы здесь указали. Но допустим, лежащие на поверхности, укажем так. Ну и в конце «Заключение» или «Выводы».

Сохраним изменения в файле, запишем их на диск и проведём ту же самую процедуру. Добавим, файл в отслеживаемый Git и за коммитим его. Соответственно, что получается? Мы делаем git add Markdown instruction. Напоминаю, что tab позволяет заполнять тот текст, который вы ещё не написали. Пользуетесь им в терминале. Посмотрим на статус, git status. Вот у нас файл изменён и можем сделать новый коммит. Git commit новое Сообщение, например, «добавили структуру файла». И отправим это сообщение. Теперь буквально на две минуты прервёмся, чтобы вы смогли догнать меня, выполнить все действия. И перейдём дальше к работе напрямую с ветками.

[00:21:02]

Ветки в Git и их использование на практике

Перейдём к знакомству с ветками (англ. branch). Команда git branch выводит на экран список имеющихся у нас веток. Напишем её и посмотрим результат. По звёздочке мы видим, что находимся в ветке master. У нас лишь одна ветка, соответственно, в другом месте мы не можем располагаться. С master мы знакомились на первой лекции. Когда мы переключались между сохранениями, переходили от коммита к коммиту, в конце, чтобы продолжить работу над файлом, мы делали git checkout master. Это значит, что мы возвращались на самую вершину ветки, чтобы её продолжить. Помимо этого, есть команда clear, очищающая поле терминала от ранее введённых команд.

[00:22:17]
Создаём новую ветку

Что мы сейчас сделаем? Мы продолжим работу сначала в черновике. Master будем считать чистовиком. Всё хорошо сделанное, готовое мы будем помещать в ветку master. Если мы хотим поэкспериментировать, что-то попробовать сделать, но ещё не закончили и не хотим выкладывать в чистовик, мы будем делать отдельные ветки. Например, мы сейчас поработаем над блоком выделения текста в отдельной ветке. Для этого необходимо её создать. Это можно сделать, с помощью команды git branch, но помимо этой команды необходимо указать имя новой ветки. Мы будем заниматься выделением или форматированием текста. Поэтому назовём её text_formatting (англ. форматирование текста). Таким образом, если мы нажимаем Enter, у нас нет выводов об ошибках, соответственно, всё нормально. Ветка создана. Чтобы убедиться в этом, вызовем команду git branch. Теперь видим две ветки. Есть ветка master, на которой мы сейчас располагаемся и ветка text_formatting. Напоминаю, звёздочка отмечает ветку, на которой мы сейчас располагаемся.

Перейдём на ветку text_formatting. Вызываем команду git checkout text_formatting. Команда git checkout позволяет переходить между разными коммитами (сохранениями). Сейчас вместо коммита укажем ветку text_formatting, на которой хотим оказаться. Можно назвать ветку как угодно, ограничений нет. Мы видим сообщение о переходе на ветку и её имя. Файл у нас никак не поменялся, потому что, создавая новую ветку, мы ничего не меняем. Только получаем возможность пойти в работе немного в другом направлении, при этом пока изменений нет. Теперь выведем на экран вид нашего файла. Слева мы видим текст в Markdown. Справа, что видят конечные клиенты. Я спрячу всё, что было. И мы начнём работать над этой версией файла.

[00:24:58]
Редактируем ветку text_formatting

С помощью команды git branch проверим, на какой ветке сейчас находимся. Ветка master, теперь не активна. Звёздочка стоит около ветки text_formatting. Соответственно, всё, что мы будем делать с файлом, никак на чистовик не повлияет. Попробуем отредактировать файл. Допустим:

- Чтобы выделить текст курсивом, необходимо обрамить его звёздочками (*). Например, *вот так* .
- Чтобы выделить текст полужирным, необходимо его обрамить двойными звёздочками (**). Например, **вот так**.

По-хорошему вам необходимо будет доделать этот файл. Чтобы у вас была собственная инструкция по работе с Markdown и не приходилось каждый раз искать её в интернете. Во-первых, это практика работы с Markdown. Во-вторых, у вас останется готовый файл, с которым можно будет сверяться.

Взглянем на правую сторону экрана. Текст из разметки переводится в отформатированный. У нас в блоке выделения текста появились два пункта. Один написанный курсивом и второй

написанный полужирным. Теперь сохраним изменения. Сделаем ветку text_formatting первым коммитом. Выполним git status. Видим, что у нас есть изменённый файл. Сделаем git add. Добавим наш файл, выполним git commit, добавив ему некоторые сообщения. Например: «Добавили выделение полужирным и курсивом». Всё коммит создан. Проверим log, появился ли этот коммит. Git log показывает изменения в обратном порядке, сначала было добавлено выделение полужирным и курсивом, потом структура файла initial commit. Мы видим все коммиты, которые были сделаны.

Теперь показано, что мы располагаемся на последнем коммите. Дальше указана ветка text_formatting, ещё есть ветка master, теперь она от нас немного отстала. Можно перейти в неё и посмотреть происходящие там. Используем стандартный способ, введём команду git checkout и укажем название ветки, в которую необходимо переместиться. Команда git checkout master меняет не только наши файлы, но и результат команды log. Потому что из коммита, находящегося в нашем чистовике, мы видим только 2 результата. Сначала коммиты добавили в структуру файла и, чуть ниже, initial commit. Ничего другого отсюда мы не видим. Ответвления в text_formatting сейчас находится вне зоны нашего доступа. Я сейчас вернусь в ветку text_formatting. И дам вам буквально одну минуту, чтоб вы меня догнали, а те, кто где-то замешкался или что-то не успел, могли привести свой файл в похожее состояние.

[00:30:28]
Структура документа

Теперь взглянем на структуру нашего файла. Как выглядит дерево сохранений. На нашей презентации можно увидеть initial commit. Некоторый коммит с описанной структурой документа и какое-то ответвление. Командой commit задали структуру документа, остался в ветке master. Есть некоторые ответвление, находящиеся в ветке text_formatting, где добавлены инструкции по выделению курсивом и полужирным.

Обратите внимание, когда мы вызываем команду log, находясь в ветке text_formatting, мы видим всё в обратном направлении до основного (стартового) коммита. Вызываем команду git log, из ветки master, мы не видим, что было дальше или в соседних ветках. Мы от неё только можем вернуться и видеть будем только доступное из этого состояния.

[00:31:48]
Создаём ветку lists

Мы немного поработали над блоком выделения текста. Допустим, мы сделали всё, что знаем и хотим поработать ещё над какой-то частью. Ветка text_formatting ещё не закончилась, потому что мы этого ещё здесь не указали. Но мы знаем немного про списки и можем продолжить работу там.

Вернёмся в наш чистовик. Вызовем git checkout master. И отсюда создадим новую ветку для работы со списками. Для этого пишем git branch lists. Проверим список веток, появилась ли ещё одна ветка. Lists – это в переводе с английского список. Соответственно, в этой ветке поработаем над этим блоком.

По сути, это будет повторение того, что мы сейчас сделали. Напомню, сначала переходим на интересующую нас ветку (git lists). После убедимся в том, что мы находимся в нужной ветке и вызовем git branch. Звёздочкой отмечена ветка, в которой мы сейчас располагаемся. Если всё в порядке, можем редактировать файл.

Здесь добавим немного информации о создании списков в Markdown. Мы делали это на первой лекции, поэтому никаких трудностей возникнуть не должно. Начнём, например, с ненумерованных списков.

- Чтобы добавить ненумерованные списки, необходимо пункты выделить звёздочкой (*). Например, вот так:

  * Элемент 1

  * Элемент 2

  * Элемент 3

Помимо ненумерованных списков, у нас были ещё нумерованные списки. Их тоже добавим в файл.

- Чтобы добавить нумерованные списки, необходимо пункты просто пронумеровать. Например, вот так:

  1. Первый пункт

  2. Второй пункт

Обратите внимание, в правой части экрана списки уже красиво оформились. Хотя мы писали всё слитно, без отступов. Сохраним изменения в файле. Теперь сделаем commit в ветке lists. Что у нас получится? Нам необходимо посмотреть на статус (git status). Убедимся, что у нас есть один изменённый файл (Markdown instruction), мы его добавим командой git add Markdown instruction и с помощью git commit добавим сообщение «Добавили нумерованные и ненумерованные списки». Всё коммит у нас прошёл. Взглянем на результат команды git log. Git log показывает созданный сейчас коммит и то, что было в ветке master.

Если мы сейчас перейдём в ветку text_formatting. То вызвав команду git log, не увидим этот commit. У нас есть commit про выделение полужирным и курсивом, но ничего про списки нет. Перейдя в ветку master, при вызове команды git log, мы не увидим ни один из дополнительных коммитов.

[00:37:24]
Проверяем структуру файла

Если взглянем на слайд презентации, то увидим следующие изменения структуры. Есть не изменившееся дерево сохранений. То есть, initial commit, самый верхний коммит для ветки master и text_formatting. Теперь появилась новая ветка, в которой мы выполняем какие-то действия.

Очень часто в реальной работе, ветки нужны для выполнения отдельных задач. Мы сейчас делаем нечто похожее. Можно одному отдать задачу по форматированию текста, а другому задачу по работе со списками. После выполнения задач можно результат добавить в ветку master. Можете сами использовать отдельные ветки, как черновики. Когда будете считать, что блок достаточен для добавления в ветку master, можете перевести из черновика (отдельной ветки) в чистовик (ветка master).

[00:38:32]
Добавим информацию в text_formatting

Продолжим работать над блоком text_formatting. В ветке text_formatting дополним информацию про выделение текста. Перейдём в необходимую ветку, используя команды git branch, для проверки нашего расположения, и git checkout text_formatting, для перехода на ветку text_formatting.

Вы, наверное, знаете, что выделять полужирным и курсивом можно несколькими способами. Мы указали только звёздочки, но ещё можно указывать нижнее подчёркивание. Добавим эту информацию. Допустим:

- Чтобы выделить текст курсивом, необходимо обрамить его звёздочками (*) или знаком нижнего подчёркивания (_). Например, *вот так* или _вот так_.

Проверим этот способ в правой части экрана. Всё правильно текст выделен курсивом. То же самое с полужирным, вместо двойных звёздочек можно использовать двойное нижние подчёркивание. Допустим:

- Чтобы выделить текст полужирным, необходимо его обрамить двойными звёздочками (**) или двойным нижним подчёркиванием (__). Например, **вот так** или __вот так__.

[00:40:34]
Совмещение форматирования текста в Markdown

Думаю, вы уже задались вопросом, а для чего существует два разных вида форматирования одних и тех же вещей? Это необходимо, чтобы вы могли совмещать. Текст может быть не только либо курсивный, либо полужирный, но и одновременно и курсивный, и полужирный.

Добавим это в инструкцию.

- Альтернативные способы выделение текста жирным или курсивом, нужны для того, чтобы мы могли совмещать оба этих способа. Например, _текст может быть выделен курсивом и при этом быть **полужирным**._

У нас часть текста написана только курсивом, а другая часть и курсивом, и полужирным. Мы бы не смогли это сделать, если бы всё было выделено только звёздочками. Система бы запуталась и не поняла, что происходит. Сохраним эту информацию и посмотрим на статус, git status показывает, что файл немного изменён. Далее, выполняем знакомую процедуру с добавлением нашего файла в коммит. Выполняем команду commit и указываем, например, «Добавили альтернативные способы выделения текста». Всё коммит появился. Можно

взглянуть на log изменений. Всё в порядке. Информация здесь тоже появилась. Обратите внимание, мы просто двигаемся дальше по ветке text_formatting. При этом ветки master и lists находятся на одном и том же месте, без изменений. Теперь прервёмся буквально на полторы минутки, чтобы те, кто не успел, смогли догнать и выполните действия самостоятельно.

[00:44:57]
Слияние ветвей

По сравнению с первоначальным вариантом наше дерево коммитов немного разрослось. Появился новый коммит в ветке text_formatting. Мы добавили альтернативный вариант, курсив и полужирного. Если сейчас нас полностью устраивает блок text_formatting, то можно залить его в чистовик.

Чтобы это сделать вернёмся в редактор. Для удобства очистим наш терминал. Проверим наше расположение. Располагаемся мы в нужной ветке text_formatting. Взглянем на текущий статус, убедимся, что всё нормально и сохранено. Теперь всё готово для заливания ветки text_formatting в ветку master. Перейдём в чистовик и, находясь в нём, добавим наши действия в соседней ветке. Делаем это с помощью команды merge (с англ. слияние). Выполним git merge и укажем ветку, которую необходимо добавить. В нашем случае — text_formatting. Если всё написано правильно, нажимаем Enter и видим, что в ветке master обновился файл, подтянулась информация, связанная с выделением текста.

Посмотрим на log изменений. Git log показывает, что мы находимся на ветке master. Также есть ветка text_formatting, которая находится на этом же коммите. В log мы видим все совершенные изменения, вплоть до добавления альтернативных способов выделения текста. Однако, здесь ещё нет ничего про списки, потому что они находятся в отдельной ветке, и мы отсюда их не видим.

Взглянем теперь на то, что может произойти с деревом проекта. На презентации у меня есть ещё один дополнительный коммит. Важно! После объединения ветвей text_formatting и master мы получаем одну общую, не отдельную, ветвь master, в которой все коммиты расположены в одном порядке. При этом ветка lists располагается по соседству.

[00:47:45]
Удаление ветви

Ветка text_formatting больше не нужна, можем её удалить. Для этого используем команду git branch -d text_formatting. Где параметр d сокращение от deleted (с англ. удалить), а text_formatting имя удаляемой ветви.

Обратите внимание, необходимо указывать d маленькую. d большая тоже будет удалять, но лучше пользоваться маленьким. Это позволит Git выполнять удаление только тогда, когда у нас всё в порядке. Только тогда, когда у нас эта ветка полностью объединена с другой и удаление ничего не поломает. Выполняем git branch -d text_formatting. Взглянем на список. Мы видим, что есть ветка lists и ветка master, всё в порядке. Ветка text_formatting у нас удалена. При этом все действия, которые внутри неё были сделаны, теперь находится в ветке master.

[00:49:22]
Создаём ветку images

Именно так и происходит работа программистов. Например, мы выполняем необходимые действия для какой-то фичи сайта. Модернизируем программу в отдельной ветке. Потом показываем результат хозяину проекта, если он считает, что всё сделано правильно и устраивает его, то изменения из отдельной ветки добавляются в чистовик. А черновую ветку можно смело удалять. Получается, мы внесли вклад в создание проекта.

Собственно, продолжим практическую работу. Повторим операции с каким-нибудь другим блоком текста. Например, мы не работали, с изображением. Сделаем новый блок, в отдельной ветке, с новой фичей для нашей программы (инструкции). Создаём новую ветку git branch images (с англ. картинки). Имя файла говорит, что в этой ветке мы будем работать с изображениями. Убедимся, что всё прошло успешно. Напомню, что с помощью стрелок можно вызывать команды заново. Итак, появилась ещё одна веточка images. Перейдём на неё и проверим наше расположение командой git branch.

Для удобства очистим терминал. Изображение в Markdown вставляется очень просто.

- Чтобы вставить изображение в текст, достаточно написать следующее:

  ! [] ()

В квадратных скобках мы укажем текст, который будет выводиться, если изображение не загрузится. А в круглых скобках имя файла, из которого необходимо изображение достать. Подождём минуту, чтобы все смогли дойти до текущего состояния файла. Я за это время подготовлю фотографию для размещения.

[00:53:30]
Добавляем картинку

За время, пока вы доводили проект до актуального состояния, я добавил в нашу папку фотографию. Теперь в папке появляется помимо файла Markdown instruction, ещё какая-то фотография. Взглянем на реакцию Git. Узнаем текущий статус. Git status говорит, что есть изменённый файл Markdown instruction. Мы начали его редактировать, но пока не сохранили изменения. Ещё появился новый не отслеживаемый файл. Сначала разберёмся с уже известным. Есть изменённый, но не сохранённый файл. И есть фотография, с которой ещё не работали.

Закончим заполнять начатое в Markdown. На моей фотографии показан котик по имени Тефтелька, поэтому в квадратные скобки напишем:

- «Привет — это Тефтелька»

Если файл лежит в одной папке с инструкцией, которую мы редактируем, то достаточно указать просто имя. Например, Teftelka.jpg.

Внимание, в правом окошке уже добавилась фотография. Мы просто указали, что хотим добавить фотографию, но её саму не видим, а в правом окошечке уже отображается файл.

Если название файла написано с ошибкой. Например, Teftelka1.jpg, то изображение справа не увидим, но вместо него увидим текст из квадратных скобок, который отобразится, если файл не существует. Вернём котика на место. Сохраним изменения. И сделаем коммит, потому что работу с изображением мы немного посмотрели.

[00:55:42]
Добавление файлов в Git

Обратите внимание, теперь есть два файла, с инструкцией и с фотографией. Мы можем сделать сохранение, в котором сохраним состояние только одного файла, а другой файл проигнорируем. Всё это делается так же, как мы делали до этого. Только в коммит добавим тефтельку. Чтобы всё было правильно. Сначала посмотрим на статус. Убедимся, что нас всё в порядке, должно быть 2 файла, а теперь мы добавим только файл с инструкцией. Тефтельку пока добавлять не будем. Посмотрим на статус, есть модифицированный файл, это файл не отслеживается, и даже написано, что он не будет за коммичен. Git говорит, что необходимо добавить этот файл, чтобы его за коммитить. Если мы сейчас сделаем коммит, то файл Markdown instruction в него запишется, а Тефтелька нет. Сделаем git commit -m Добавили инструкцию по работе с изображениями. Коммит у нас готов, взглянем на log происходящего. Мы остановились на «Добавили альтернативный способ выделения текста». Ниже мы добавили новую ветку images, поэтому все эти изменения ветки images тоже находятся. Потом мы продвинулись дальше и добавили инструкцию по работе с изображением.

Если мы снова вызовем git status, то увидим, что наш файл с Тефтелькой висит не отслеживаемый, но Git его видит. При этом не принято добавлять в Git какие-то фотографии или большие файлы. Их обычно игнорируют и работают только с текстовой составляющей. Также делают и программисты, когда исходный код программы, который является текстовым файлом, добавляется в систему контроля версий, а всё остальное лежит отдельно, либо в папках, либо на каких-то серверах или файловых хранилищах. А вот в систему контроля версий добавляются только текстовые файлы или исходный код нашей программы.

[00:58:16]
Команда .gitignore

Чтобы Git больше не спрашивал нас про тефтельку, можно добавить ещё один файл. Дать ему специальное, фиксированное название. Назовём файл .gitignore. Важно! Не допустить ошибки в названии файла. В созданном файле укажем имя файлов, которые необходимо игнорировать Git.

Если мы укажем .gitignore, напишем Teftelka.jpg и сохраним файл. Теперь нам необходимо всего лишь за коммитить этот файл. Мы делаем git add .gitignore . Далее выполняем git commit -m добавили gitignore файл. И посмотрим на статус ещё раз. Мы видим, что файл с тефтелькой не добавлен в репозиторий, а добавлен файл .gitignore. При этом Git перестал нас спрашивать, что делать с тефтелькой. Этот файл в Visual Studio Code даже помечен серым. Он показывает, что всё указанное в файле .gitignore Git будет автоматически будет пропускать.

[01:00:13]

Объединение веток images и master

Итак, продолжим работу. Сначала очистим наш терминал, скроем все изменения. Повторим операцию со слиянием нескольких веток. Работу с изображением, мы в целом посмотрели, добавили котика. И можем эти изменения добавить ветку в master.

Обратите внимание, мы добавили файл .gitignore в ветку с изображениями. На самом деле это не очень хорошо, но сейчас исправим всё объедение ветки, чтобы в чистовой версии всё работало точно так же.

Посмотрим, что здесь сохранено. Убедимся, что мы не в ветке master. Мы в ветке images. Теперь перейдём в ветку master. Видите, я дописал команду git checkout master. Перешли в ветку master.

Сейчас исчез файл .gitignore, но он должен будет появиться, когда мы сольём эти ветки. Кроме того, обратите внимание, что мы сейчас работаем немножко в тепличных условиях. Мы в ветку master только добавляем. Нет никаких конфликтов. Когда у нас вот в этом блоке master что-то написано, и это же отредактировано в соседней ветке. И когда мы будем их сливать две версии одного и того же текста, пока такого не возникает. Сейчас ещё раз это провернём, а потом посмотрим, что же происходит, если у нас один и тот же текст отредактирован по-разному в разных ветках.

Зальём информацию про тефтельку. Убедимся, что находимся в ветке master. И git merge добавим ветку images. Всё прошло успешно. Обратите внимание, что файл .gitignore переехал из ветки images в ветку master. Появилась тефтелька и информация о том, как необходимо выводить фотографии в Markdown.

[01:02:42]
Объединение ветвей с разной информацией

Посмотрим более сложную ситуацию. Когда один и тот же текст в разных ветках написан по-разному. Если будем сливать эти ветки, возникнет конфликт, Git не поймёт, какую версию необходимо использовать. Чтобы воссоздать эту ситуацию, мы в ветке master что-нибудь напишем про списки в ветке lists.

Чтобы выделить ненумерованный список, используйте (*).

Посмотрим, как этот блок текста оформлен в lists. Но до этого сделаем здесь коммит внутри нашего чистовика. Git status, видим, что мы изменили этот файл. Мы сделаем git add Markdown\ instruction.md и git commit -m «Добавили информацию про списки«. Коммит у нас появился.

Теперь в master и lists есть информация про списки. Взглянем на неё. Git checkout lists. Здесь информация про нумерованные и ненумерованные списки написана более подробно. Когда мы будем сливать ветки, важно посмотреть, произойдёт конфликт или обе версии сохраняться? Сначала появится то, что было в master, потом то, что было в lists.

Посмотрим на то, как же это будет работать. Очистим терминал. Перейдём в ветку master. Теперь с помощью команды git merge lists зальём сюда изменения из ветки lists.

Изменения прошли, но с конфликтом. Проверим, что теперь отображается. Если вы посмотрите, то в блоке со списками есть 2 варианта. В одном из них указано HEAD. Это то, где мы находимся. И есть вариант, который пришёл из lists. Git не может самостоятельно решить, какую же версию ему использовать. Использовать версию, которая у вас была в ветке master или в той ветке, в которую вы заливаете информацию. Или использовать то, что пришло снаружи. Git говорит, решить это самостоятельно, алгоритмом это сделать не получается.

У Visual Studio Code есть возможность принять текущую версию, которая к нам пришла. Текущая версия — это Current Change (с англ. текущие изменения). Incoming change (с англ. входящие изменения) это то, что пришло с другой ветки. Есть возможность оставить и сравнить оба варианта. Мы оставим оба варианта и вручную отредактируем их, оставим то, что хотим.

Уберём повторяющиеся части, и отредактируем до следующего состояния.

Чтобы добавить ненумерованные списки, необходимо пункты выделить звёздочкой (*) или знаком +. Например, вот так:

* Элемент 1

* Элемент 2

* Элемент 3

+ Элемент 4

Выделить звёздочкой или знаком плюс. Например, вот так, и вот 4 элемент плюсиком, чтобы у нас был смысл в этом коммите. Например, элемент 4. Сохраним изменения.

Нужно будет посмотреть, как всё работает. Есть состояние после merge, но оно ещё не сохранено. После того как мы решили, как будет выглядеть итоговая версия, нам необходимо сохранить результат. Git говорит, что автоматическое слияние не получилось, исправьте конфликты, неточности и сохраните результат. Собственно, именно это и сделаем. Выведем команду git status. После этого мы добавим файл Markdown. И можем делать git commit. И мы сейчас укажем сообщение, что слили изменения с учётом конфликтов из ветки lists. Всё получилось. Взглянем на текущий статус. У нас всё в порядке, всё сохранено. Взглянем на log изменений. Мы видим, что у нас последний коммит, как раз слили изменения с учётом конфликтов из ветки lists.

При работе в 2 разных ветках, может возникнуть ситуация, когда в одной ветке и во второй ветке вы по-разному изменили один и тот же блок текста, тогда при их объединении, Git попросит вас разобраться с этим конфликтом. Вы можете выбрать либо одни изменения, либо другие. Либо просто сохранить оба и переписать всё заново начисто, чтобы сохранилась именно та версия, которую вы хотите оставить. В реальной работе такое встречается достаточно часто, но при этом у репозитория, с котором вы работаете, должен быть один ответственный человек, имеющий право объединять изменения и разрешать их конфликты. Но к совместной работе мы придём в третьей лекции. Сейчас вы работаете самостоятельно, и сами разрешаете все возникающие конфликты.

[01:09:54]

Команда git log -graph

Теперь посмотрим на то, как можно визуализировать все имеющиеся коммиты (ветки). Мы можем добавить к команде git log параметр graph. И увидеть иное отображение дерева коммитов.

Обратите внимание. У нас есть несколько веток (images, lists и master) с разными коммитами, принадлежащими разным веткам. Например, ветка master, у нас отмечена, а в ветках рядом ветки тоже происходят коммиты.

Мы можем удалить ветки, уже выполнившие свою роль. Допустим, ветки images и lists нас уже не интересуют, они выполнили все необходимые действия. Прервёмся буквально на 2–3 минуты. Чтобы вы могли догнать меня.

[01:14:13]

Повторим изученное сегодня

Дальнейшую работу над файлом можете продолжить самостоятельно. Рекомендую вам поработать над оставшимися пунктами инструкции. Под каждый пункт выделить отдельную ветку и выполнить в ней все необходимые действия. Дальше слить ветку с чистовиком или удалить, если она больше не нужна. Желательно несколько раз искусственно создать конфликты. Посмотрите, что происходит при объединении веток и появлении конфликтов.

Чтобы запомнить всё пройденное, повторим изученные сегодня команды.

Команда git branch. Позволяет выводить список всех созданных веток. Например, есть 5 веток и звёздочкой отмечена ветка, на которой мы располагаемся. Чаще всего есть одна основная ветка (чистовик) для работы. И дополнительные ветки (черновики) для экспериментов, которые можно удалить, если нас что-то не устроит. Если мы закончили работу в черновой ветке, то мы можем слить её с основной. В итоге чистовая ветка будет содержать готовую законченную работу.

Если к команде new_branch добавим имя, то создадим новую ветку с указанным именем. Например, new_branch_name. Вместо name укажем название. У нас были ветки lists и images, можете выбрать любое название, которое вам больше понравится и отражает суть ветки, объясняет происходящие внутри неё.

Мы смотрели, как удалять ветки. Вызываем команду git -branch, можем добавить ещё один параметр — d (delete, удалить). И указать имя ветки, которую хотим удалить.

Узнали, как переходить с одной ветки на другую. Если вместо хэша коммита указываем имя ветки, то сможем переключаться между ветками.

Посмотрели, как выводить дерево коммитов. Имея несколько то сливающихся, то расходящихся веток, мы можем командой git log -graph отобразить понятный список имеющихся коммитов.

Научились сливать ветки, используя команду merge. Самое важное, merge вызывается оттуда, куда вы хотите добавить изменения. Если вы захотите залить изменения в master. То сначала необходимо через git checkout master перейти в ветку master, и только после этого вызвать, например, git merge images. При этом могут возникнуть конфликты. Если у вас один и тот же блок текста или файл немного по-разному оформлен, то Git не будет знать, какую из версий использовать и попросит вас помочь.

[01:18:19]

Окончание занятия

Что же, на семинарах с преподавателями вы ещё раз пройдёте через все операции. Но очень рекомендую, до начала семинара ещё поработать с ветками. Повторите все изученные действия с ними. И дальше на семинарах вы сможете отработать все дополнительные действия.

Всем спасибо увидимся на следующей лекции.