GeekBrains

Знакомство с контролем версий

Оглавление

[00.01.36]

Приветствие

Добрый день, дорогие друзья! Рады приветствовать вас на нашем курсе «Введение в контроль версий».

[00.02.47]

На этом уроке

Сегодня мы познакомимся с контролем версий. Многие из вас слышали такие слова, как Git и GitHub. Кто-то знает про контроль версий и, возможно, проходил какие-то курсы. На платформе GeekBrains уже есть такой курс, и часть из вас, скорее всего, его видела. Ничего страшного, если вы его не смотрели или уже это сделали.

Курс, который мы начинаем, самодостаточен. Начнём с базы. Если вы не видели другие курсы по контролю версий, по Git, то после прохождения этого ознакомьтесь с ними.

[00:03:40]

Теория

Что это и для чего может быть нужно

Итак, разберёмся, для чего вообще нам нужен контроль версий как таковой. В качестве примера рассмотрим какую-нибудь задачу. Начнём с немного отдалённого и, возможно, не очень знакомого вам примера.

Допустим, мы написали какую-то программу или подготовили сайт, который работает прекрасно. Выгрузили его в интернет. Люди заходят на этот сайт, пользуются, им всё нравится. Но через некоторое время вы решили, что на сайте не хватает какой-то функциональности. Например, надо поменять меню, добавить страницы, возможно, регистрацию пользователей или что-то ещё. То есть вам понадобилось улучшить сайт и что-то в нём изменить.

Первое, что вы сделаете — сохраните рабочую версию сайта. То, что работает хорошо, никто менять на ходу, скорее всего, не будет. Наверняка на вашем компьютере, на сервере или ещё в каком-то месте появится папочка или архив, где будет написано что-то вроде «Версия 1.0» (рабочая версия сайта).

После этого вы начнёте создавать новое или видоизменять то, что уже написано. Спустя какое-то время у вас появится ещё одна версия — «Версия 2.0». Вы тоже куда-нибудь её аккуратно положите, чтобы в любой момент к ней вернуться.

Если произведённые изменения внезапно всё сломали, вы всегда сможете достать из своего архива старую версию и вернуться к рабочему состоянию.

[00:05:35]

Причины осуществления контроля версий

Приведём две основные причины для осуществления контроля версий, которые на бытовом уровне кажутся самыми очевидными:

1. Возможность хранить различные версии проекта.
2. Возможность возвращаться к этим версиям.

Соответственно, когда на компьютере мы храним версию нашего прекрасно работающего сайта, это первый пункт. А когда используем папку с сохранёнными материалами как рабочую, это считается возможностью возвращаться к различным версиям. Если таких версий много, то и папок на компьютере также будет много.

[00:06:16]

Пример 1

Например, у нас есть какая-то версия от 30 декабря, и мы решили в ней что-то видоизменить. Сохранили то, что есть, на наш сервер или компьютер. Сделали какие-то преобразования. Если теперь наш сайт работает прекрасно, будем пользоваться какой-то другой версией, например, от 10 января.

Спустя какое-то время мы опять переработали свой сайт. Значит, появится следующая версия.

Таким образом, мы будем хранить разные версии или, как говорят программисты, контролировать их. То есть у нас будет некоторое количество этих версий. У этого есть свои недостатки. Однако это и представляет собой контроль версий в самом базовом варианте.

[00:07:06]

Пример 2

Если вариант с сайтом вам не очень близок, замените его на любой другой пример.

Допустим, вы пишете курсовую, книгу или готовите ещё какой-то документ. У вас появляется первая версия, с ней всё в порядке, она всех устраивает. Но потом приходит вдохновение, и вы решаете что-то переделать. Начали всё править и менять текст, который уже написан. Но опять же будет здорово, если вы сохраните версию, которая всех уже устраивала, а потом продолжите работу.

[00:07:41]

Пример 3

Или возьмём, к примеру, написание картины. Если вы рисуете на холсте, то, естественно, сохранять промежуточные версии будет некомфортно. Придётся перерисовывать всю картину. Но если работаете в графическом редакторе, то сохраните текущую картину, положив её в отдельную папку, чтобы она не потерялась, и продолжите рисовать. Так вы всегда сможете вернуться к предыдущей версии.

Это и есть контроль версий, тема, которую мы рассмотрим на текущем курсе. А пока разберём самые простые, бытовые, варианты того, как организовать контроль версий в обычной жизни.

[00:08:28]

Сохранение в играх

Рассмотрим пример с компьютерными играми. Многие знают, что в компьютерных играх часто можно сохраниться, чтобы потом продолжить прохождение. Если сделали что-то неправильно, вы всегда сможете загрузиться и вернуться к прежнему состоянию. По сути, это тоже в некотором роде контроль версий. Это максимально близкий пример того, как программисты или кто-то ещё занимается работой с использованием контроля версий.

[00:09:24]

Пример 4

Разберём более сложный пример. До этого мы рассматривали случаи самостоятельной работы. Но что происходит, когда в работе участвует несколько человек? Для примера возьмём вариант с написанием текста, который занимает больше нескольких дней, и где

участвует несколько человек. Если можете выполнить работу за один присест, то вы, скорее всего, справитесь и без контроля версий. Но если работа занимает у вас неделю, месяц или несколько месяцев, вопрос с контролем версий будет очень актуален.

Поговорим о том, как строится в таких случаях работа. Для примера возьмём написание какого-либо текста:

- написание курсовой, над которой работаете вы и ваш научный руководитель;
- составление отчёта по работе, где участвуете вы, ваши коллеги и начальник, поручивший это делать;
- издание книги, где вы пишете какой-то текст, а главный редактор, читающий его, утверждает или каким-либо образом влияет на вашу работу.

Можете взять любой пример, наиболее близкий вам. Главное — написать какой-то текст, подключив несколько человек.

Допустим, мы пишем курсовую. Сели, написали что-то, и у нас появился первый черновик, занимающий 2–3 страницы. Подготавливаем получившийся вариант и отправляем своему научному руководителю, а он, в свою очередь, его читает. Во время работы над курсовой вместо «Черновик 1» появляется «Черновик 2», то есть её следующая версия.

Наш руководитель работает с первой версией — «Черновик 1». Вносит в неё какие-то правки, что-то изменяет, дописывает. Таким образом, у нас появляется несколько веток, по которым наша курсовая готовится.

Появилась версия «Черновик 2». У человека, которому мы отправили предыдущую версию, будет «Черновик 1» с комментариями.

Но после отправки «Черновик 1» научному руководителю, мы создаём «Черновик 2», внося изменения в свою версию «Черновик 1». Так возникает некоторый конфликт версий. У нас этот текст уже написан по-новому. А другой человек, думая, что в нашем варианте старый текст, переписал его по-своему. В итоге всё это придётся каким-то образом сводить, приводить к какому-то общему итогу.

Но, допустим, всё идёт нормально, мы разобрались в этих двух документах, в результате чего появился «Черновик 3». Далее процесс повторяется. Таким образом, итеративно проект пишется, мы делаем свою работу.

[00:12:41]

Недостатки подхода

На самом деле ситуация начинает усложняться. Появляется множество файлов, каждый из которых называется по-своему. В этом случае надо придумать некоторые правила:

- в одну сторону положить свой черновик, а в другую — утверждённую версию от руководителя или промежуточные версии от руководителя;
- вместо номеров указывать даты;
- организовать другой порядок.

Опять же, если работа длительная, и в ней участвует несколько человек, это очень неудобно. Появляется много проблем:

- хранение;
- необходимость перекидывать документы друг другу;
- контроль того, что и в каком файле находится;
- забота об актуальности документа;
- контроль того, от кого пришёл документ;
- определение того, в каком документе содержатся собственные правки, а где — правки научного руководителя.

Это очень неудобно и долго. А если работу выполняют уже более двух человек, могут возникнуть серьёзные проблемы. То есть эта система только разрастается, работать так практически невозможно.

Даже если взять Google Docs, который позволяет одновременно нескольким людям работать над одним и тем же файлом, такой вариант в реальных бизнес-задачах применим не во всех случаях. Мы далеко не всегда сможем воспользоваться такими облачными сервисами и одновременно работать над одним и тем же документом.

Возьмём самый простой пример. Допустим, вы подготовили какую-то часть документа. К этому документу есть доступ не только у вас, но и у кого-то ещё. И возможно, кто-то внёс некоторые изменения, о которых вы не в курсе. То есть у вас не всегда будет возможность контролировать какие-то изменения в документе. Его часто придётся перечитывать вам или человеку, ответственному за документ.

Или, например, в компании GeekBrains пользоваться Google Docs и облачными сервисами запрещено, потому что возникает большая дыра в безопасности. То есть писать курсовую, используя подобные документы, вы ещё можете, но, работая в крупных компаниях, у вас такой возможности не будет.

Таким образом, перед нами встают две большие проблемы: путаница в актуальности информации и её безопасность.

[00:15:13]

Git

Проблема, связанная с тем, о чём сказано выше, возникла давно. Если написание книги или подготовка курсовой — это довольно естественные, но редко встречающуюся примеры, то у программистов проблемы, связанные с совместной работой над проектами, возникли давно. Всё началось в те времена, когда интернет был медленным или он отсутствовал вовсе, память компьютеров стоила дорого, и сохранять полную версию или полную копию какого-то файла считалось роскошью. Программисты уже решали эту проблему.

На экране представлен логотип и название программы, которая позволяет программистам решать эту задачу. О ней вы, скорее всего, слышали. Эта программа называется git. Она считается самой популярной, но далеко не единственной и не первой системой контроля версий. Их на самом деле множество. Просто Git сейчас считается самой популярной из используемых систем, хотя разные компании пользуются и другими. Однако смысл и алгоритм их использования во многом похож. Мы начнём знакомиться с Git, как с самой популярной системой. И вам, скорее всего, придётся работать с ней в повседневной жизни. Неважно, будете ли вы программистом, тестировщиком, аналитиком или проджектом.

Не стоит бояться этой системы. Её задача — помогать вам.

Сначала может показаться, что пользоваться ей довольно сложно. Есть некие действия, к которым вы не привыкли. И гораздо проще сохранить какую-то версию файла, положить её на флешку или сделать с ней что-то ещё. Но на самом деле, если вы привыкнете к программе, то поймёте, что она сильно упрощает жизнь. Стоит её освоить, и ваша работа станет сильно проще. Но вернёмся к Git.

[00:17:34]

Linux

Git написан Линусом Торвальдсом. Это финно-американский программист, который больше известен тем, что написал ядро операционной системы Linux. Вы, скорее всего, о Linux что-то слышали. Естественно, что текущая версия Linux не полностью написана им. Если почитаете о нём статью на сайте «Википедия», то узнаете, что Линусом Торвальдсом написано буквально 2% от всего Linux. Но на самом деле это уже много. Однако самим идейным вдохновителем создания этой операционной системы и первым программистом, который в ней работал, считается Линус Торвальдс.

Но вернёмся к написанию текста, его версиям, к работе над ним нескольких человек, к примеру, студента и научного руководителя, и сложностям, возникающим из-за этого.

Представьте, что Линус Торвальдс пишет операционную систему — это огромный проект. От написания первого варианта, который он показал людям, до выхода релиза, то есть полностью рабочей версии, у него прошло три года. Линус Торвальдс написал много строк кода, при этом он работал не один, а с большой командой энтузиастов-программистов, помогавших ему в этом. Получается, что у нас есть десятки, сотни, а то и тысячи программистов, предлагающих какие-то улучшения. И есть вы, Линус Торвальдс, который принимает решение, что надо делать и как. У него есть разные версии, и всё это необходимо контролировать.

Если он, предположим, работал бы тем же способом, что и студент с научным руководителем над курсовой, Linux бы не появился. И в этом случае удобная система контроля версий считается не роскошью, а необходимостью, без которой указанный проект просто не смог бы существовать. И Линус Торвальдс написал программу, позволившая ему удобно настроить версионирование проекта и командную работу над ним.

[00:19:41]

Принцип работы Git

Итак, мы рассмотрели две основные особенности, которые возникают, когда нам надо работать над большим проектом. Сначала поговорили о сайте и резервных копиях — версионировании, когда у нас есть рабочая версия, следующая рабочая версия и так далее. Затем обсудили командную работу над проектом и привели в пример студента и научного руководителя, работающих совместно. Именно эти особенности мы и будем разбирать на этом курсе.

Не надо бояться контроля версий. Git — просто программа, которую вы устанавливаете на свой компьютер. Она решает за вас все вопросы, связанные с версионированием проекта. Ей просто надо немного помогать, а самое сложное она сделает за вас.

Важно понимать, что принцип работы Git несколько отличается от рассмотренного ранее нами. Когда мы копировали файлы целиком, то просто делали полные резервные копии и хранили их в отдельных папках, а кто-то, возможно, в облачном хранилище, на флешке или как-то ещё. Git работает иначе.

Предположим, у нас есть одна страница текста. Мы сохранили её как текущую версию. Затем дописали ещё два слова в этот документ и хотим сохранить новую версию, как описывали ранее. Сохраняя копию сайта, мы сохраним две страницы текста. У нас будет первая страница и та же страница, но с двумя дополнительными словами. То есть у нас, по сути, сохранятся две страницы текста. Один файлик и соседний файлик, предыдущая версия и новая версия.

Git же сохранит первую страницу и те два дополнительных слова, которые мы написали. То есть сохранит то, что было, а также добавленное. Удалив несколько слов, он сохранит только разницу между файлами. А когда понадобится воспользоваться этим файлом, он самостоятельно и очень быстро его соберёт. Таким образом, Git позволяет сильно экономить память.

Предположим, у нас была не одна страница, а тысяча. Мы писали «Войну и мир» и вдруг решили в заключительном слове добавить несколько слов благодарности. В итоге получится документ на тысячу страниц и новая версия, содержащая ту же тысячу страниц и абзац текста. Git сэкономит память (место) примерно в два раза. А если у нас будет не две версии, а, например, сотни версий, Git позволит уже сэкономить в сто раз больше памяти. Он будет с этим работать гораздо эффективнее.

В те времена, когда Линус Торвальдс только разрабатывал систему контроля версий, работал над созданием операционной системы Linux, память стоила очень дорого. Если сейчас скопировать файл и пересохранить его несколько раз для нас — простая операция, не требующая никаких усилий, то в те времена каждый лишний байт, тем более килобайт или мегабайт считался чем-то очень тяжёлым. Это бы заняло почти всю свободную память на компьютере. Поэтому Git — очень полезный инструмент, который позволяет легко наладить версионирование любого проекта.

[00:23:24]

Для работы потребуется

Итак, на этой лекции мы определимся с двумя принципами, с двумя частями системы контроля версий. Нам надо научиться **сохранять** разные версии и **перемещаться** между ними. Это похоже на то, как мы сохраняем разные версии файлов, но с использованием программы Git.

Для этого потребуется установить на свой компьютер несколько программ — Git и редактор, в котором будем писать. На самом деле можно обойтись и блокнотом, но лучше воспользоваться программой Visual Studio Code. Она бесплатная. Есть версия для Linux, Mac и Windows. Программа очень удобная. Её мы будем использовать как редактор кода и оболочку для системы контроля версий. Ещё она пригодится нам во время самого программирования.

Установите эти две программы — Git и Visual Studio Code — на свой компьютер. Под лекцией в описании вы найдёте ссылки на их установку. Если вам недостаточно указанных ссылок, зайдите на YouTube или в поисковом окне введите запрос того, как установить, например, VS Code на свою операционную систему, и найти все необходимые инструкции.

[00:25:10]

Наша задача

Сегодня научимся делать первые шаги, используя систему контроля версий Git. Но мы ещё не умеем программировать. А если вы смотрите этот курс после прохождения уроков по программированию или уже умеете программировать, но пока плохо знакомы с системой контроля версий, ничего страшного.

Мы будем работать с текстом, писать его и сохранять получившиеся версии, используя Git. Но этого не стоит бояться. Этим и занимаются программисты. Те, кто проходил курс «Введение в программирование», знает, что исходный код программы — всего лишь текстовый файл, который как-то оформлен. Вспомните, как мы сравнивали код, написанный на Java и Python.

Код на Java представляет собой текст, где есть точки с запятой и скобки, оформленные по одним правилам. А код на Python оформлен по другим правилам. Но так или иначе это лишь текстовый файл. Поэтому если вы научитесь использовать Git в обычных текстовых файлах, то аналогичным образом сможете работать и с Git, и с другими программами, а также с различными языками программирования.

Важно отметить, почему мы даём контроль версий до того, как вы начали напрямую заниматься программированием, то есть написанием кода на каком-то конкретном языке. В курсе «Введение в программирование» говорится о нескольких видах сложности:

- обязательная сложность, она же необходимая;
- случайная сложность;
- необязательная сложность.

Мы хотим начать работать с системой контроля версий. Это необходимая сложность, так как должны этому научиться. А если будем параллельно учиться программированию, возникнет дополнительная сложность. Подумайте, к какому типу сложности она относится — к необязательной или к случайной.

[00:27:36]

Практика

А теперь перейдём от теоретической части лекции, где говорилось о появлении контроля версий и его применении, сохраняя разные версии на своём диске, к тому, как это работает на практике. Вы узнаете, что можно делать с помощью программы Git, как ею пользоваться и с чего начать изучение системы контроля версий.

[00:28:09]

Установка Git и Visual Studio Code

Начнём с того, как эта система работает на практике. Для этого установите Visual Studio Code и систему Git, чтобы на своём компьютере повторить всё, что делалось на лекции. Пройдём все шаги и посмотрим, как это происходит на самом деле.

Чтобы отслеживать изменения, надо где-то наши файлы хранить. Создадим папку на рабочем столе и назовём её, например, git_education. Название можете выбрать любое, для системы контроля версий Git оно никакого значения не имеет. Теперь запустим русифицированную программу Visual Studio Code. Вы можете использовать любую версию, например, привыкать к английскому языку и не русифицировать её.

[00:29:22]

Настраиваем Visual Studio Code

Итак, чтобы начать работать в этой папке, а программа Visual Studio Code могла взаимодействовать с ней, надо её открыть. В левом верхнем углу есть проводник, кликаем по нему и открываем созданную на рабочем столе папку. Она совершенно пустая. Это обычная папка, какую вы можете создать на своём компьютере. Никакой магии здесь пока не происходит. А чтобы Git начал контролировать то, что происходит в созданной папке, сохранял версии и помогал в работе, надо дать ему команду на отслеживание происходящего.

Для этого используется графический интерфейс. Мышкой можно кликать по меню и найти там Git. Соответственно, всё, что мы будем делать на клавиатуре, также выполняется мышкой. Но начинать с этого не рекомендуется. Сразу познакомимся с тем вариантом, которым пользуются опытные программисты.

[00:30:38]

Терминал

Для этого надо запустить терминал. Чтобы его запустить, перейдём в меню, найдём пункт «Вид» и в выпадающем окошке выберем «Терминал». Если пользуетесь не Mac, а, например, Linux или Windows, пункты меню будут немного отличаться, но вкладку «Терминал» увидите.

Терминал — это окошко, которое появилось справа внизу. В нём будет указана ваша учётная запись, имя компьютера, на котором работаете, и поле для ввода команд. Когда программисты пишут программы, то, скорее всего, делают это в окне с чёрным фоном. Это и есть терминал.

Обычные люди привыкли пользоваться мышкой. Если мы работаем на Windows, то всегда можем правой кнопкой мыши нажать на какое-то поле, найти меню и выбрать из него что-то. Клавиатурой почти не пользуемся.

Чтобы в программе появился графический интерфейс, то есть все эти кнопочки, по которым можно нажимать, надо приложить много усилий. Нужен дизайнер, который нарисует эти кнопки, и программист, чтобы их аккуратно расположить. Главное — сделать так, чтобы меню было удобным. Это целое отдельное искусство и программы, которые обладают графическим интерфейсом. Они очень дороги как по времени на их разработку, так и в плане усилий программистов.

Поэтому программы, которыми пользуются программисты для работ с удалёнными серверами или которые вы самостоятельно станете писать без возможности привлечь дизайнеров, чтобы

сделать дизайн программы, будут консольными. То есть вызываться командами, написанными на клавиатуре.

Это может немного пугать. Сначала с непривычки людям кажется, что всё это очень сложно. Но когда привыкнете к этому и присмотритесь, вам станет гораздо удобнее пользоваться терминалом, чем искать мышкой пункты меню и запоминать их названия и место. Гораздо удобнее набрать несколько команд в терминале, и программа сделает всё за вас.

[00:33:22]

Настраиваем Git

Вернёмся к контролю версий. Сейчас пока никакого контроля версий в нашей папке git_education не происходит. Мы можем создавать файлы и сохранять их. Но самой магии, связанной с Git, ещё происходить не будет.

Чтобы эта магия случилась, и Git позволял нам контролировать версии и выполнять всё, требуется, надо его инициализировать. Но сначала проверим, что Git настроен.

Для этого в терминале:

- наберём git — это имя программы, которую хотим запустить;
- поставим пробел, то есть отделим имя программы от тех параметров, которые в неё передаём;
- добавим два тире;
- наберём слово version («версия»).

То есть сейчас мы попросим Git вывести свою версию на экран. И если Git установлен и настроен правильно, увидим на экране текущую версию этой программы.

Нажимаем на кнопку Enter. Если вы увидели git version, а потом — набор каких-то чисел, значит, Git настроен правильно. А если сделаете опечатку, или Git у вас не настроен, например, вместо git написали cit, появится ошибка. В этом случае терминал пишет, что определённая команда не найдена. Не надо этого пугаться, такое бывает. Вы можете сделать опечатку, и ничего страшного не произойдёт. Посмотрели, что написал терминал, какой команды он не может найти, увидели опечатку и написали всё правильно.

Теперь мы хотим, чтобы Git начал отслеживать всё, что будем делать в своей папке. Повторюсь, для этого нам надо инициализировать Git, запустив специальную команду. Но все команды в Git, как и в программах, работающих с терминалом, мнемонические. То есть их довольно легко запомнить. Особенно если вы хоть немного знаете английский язык.

Мы можем написать git, поставить пробел и попросить программу инициализироваться. Не обязательно писать слово целиком, программисты — люди немного ленивые, и они решили, что первых четырёх букв будет достаточно.

Если напишем git init, случится именно то, что мы от него хотим. В этой папке Git начнёт творить магию. Нажимаем на Enter и видим некоторые сообщения — в нашей папке Git начинает отслеживать все изменения.

Если посмотрите в проводник, то увидите, что поменялось в этой папке. А если откроете скрытые файлы, заметите, что в папке git_education появилась скрытая папка **.git**. В ней ничего менять не надо. Это специальная папка для работы Git.

Помимо этого, мы можем узнать, что Git думает о нашей папке и её содержимом, а также статус происходящего. Соответственно, для этого пишется тоже очень простая команда, которую легко запомнить — git status. Git говорит, что мы находимся в ветке master, и никаких коммитов нет. Со словом «коммит» мы сейчас познакомимся. Он говорит: «nothing to commit».

Если знаете английский язык, то слово commit уже будет знакомо. Для тех, кто с английским языком дружит плохо, надо запомнить, что commit — это фиксация.

Git говорит, что у нас пока нет никаких фиксаций, то есть ничего не зафиксировано. Либо можно запомнить как сохранение в играх. То есть пока никаких сохранений у нас нет, и сохранять, то есть фиксировать нечего.

В скобках написаны некоторые подсказки по сохранению. Git говорит, что сначала надо создать или скопировать какие-то файлы, а потом, соответственно, вызвать команду git add, чтобы отслеживать эти файлы.

Последуем рекомендациям Git, содержащимся в скобках, и создадим новый файл. В нашей папке git_education можно создать новый файл. Отдадим дань программистам и назовём её hello world. Расширение дадим немного необычное — **.md**, но на самом деле сейчас это не имеет никакого значения. Можно создать файл **.txt**, **.doc** или тот, какой потребуется. Для системы контроля версий никакого значения это не имеет.

[00:38:31]

Создаём новый файл

Параллельно с контролем версий мы познакомимся с языком разметки Markdown, который обширно используется в интернете. Вам полезно будет уметь им пользоваться. На самом деле он очень простой, поэтому не добавит к процессу обучения новой сложности.

Создаём новый файл — он появляется в нашем окошке. Напишем в нём какой-нибудь текст, например, hello world. Текст может быть любым. Сохраним этот файл, нажав Ctrl+s или Cmd+s, если работаете на Mac. Теперь попросим Git указать текущий статус. Он говорит нам создать файлы. Посмотрим, что случилось с Git, после того как мы создали новые файлы.

[00:39:23]

Знакомимся с командами Git

Чтобы снова вызвать команду, которую уже писали, не обязательно набирать её заново. Можно воспользоваться стрелками на клавиатуре. Если нажать стрелочку «вверх», будут открываться те команды, которые создавались до этого. Вы можете переключаться между ними. Поэтому, работая с терминалом, если начинаете к нему привыкать, вам не надо заново писать уже написанные команды. Чтобы вызвать одну из них, используйте стрелочки.

Итак, мы вызвали команду git status. Теперь Git говорит:

- есть не отслеживаемые файлы — Untracked files;
- файлы появились, но он за ними пока не следит.

Если хотите, чтобы Git начал отслеживать изменения в этих файлах и творить свою магию в плане контроля версии, их надо добавить. В терминале для этого есть подсказки, где говорится, что мы можем использовать команду git add (add — «добавить») и указать какие-то файлы, чтобы включить их в фиксацию.

Воспользуемся подсказкой Git, наберём git add и укажем файл, который надо добавить. Писать название файла целиком не надо. В терминале есть удобное автодополнение.

Мы написали только первые две буквы файла. Если нажать Tab, терминал самостоятельно заполнит то, что надо написать. Появился файл hello world.md, поэтому писать название целиком не надо. Кнопка Tab сильно ускоряет и упрощает работу с терминалом. Нажимаем Enter — никаких сообщений об ошибках нет. Значит, всё прошло успешно.

Ещё раз взглянем на статус. Для этого напишем git status. Теперь Git начал отслеживать наши файлы. Он говорит, что некоторые изменения могут быть сохранены, и указывает на созданное. Здесь также есть информация, что у нас появился новый файл, new file, который называется hello world.

То есть всё нормально, магия началась, Git стал отслеживать файлы. Мы можем сохранить те изменения, которые совершили. Если надо сделать некоторые сохранения или зафиксировать текущий статус, вызываем команду git commit и указываем некоторые комментарии, чтобы в дальнейшем понять, какое было сделано сохранение и для чего. Для этого к команде commit добавим параметр m — сокращение от message («сообщение») — и в кавычках запишем то, что надо запомнить, некий комментарий к сохранению.

Их принято писать на английском языке, но сейчас, чтобы не повышать сложность, будем писать на русском. Однако старайтесь использовать английский язык, так как он принят в мире IT.

Напишем, что мы создали новый файл: «Создали новый файл». Сообщение пишется в кавычках. Далее нажимаем Enter — выскочило сообщение, в котором говорится:

- появился новый коммит с некоторым комментарием, который мы сами написали;
- один файл изменён — в него добавилась одна строка;
- у нас с файлом hello world идёт некоторая работа.

[00:43:30]

Изменяем файл

Изменим наш файл. Напишем в нём какой-нибудь текст, например, «Начинаем наше знакомство с контролем версий». Наш файл видоизменился. Сохраним то, что добавили. В итоге у нас есть сохранённая версия файла, предыдущая версия и какие-то изменения, которые мы внесли в указанный файл.

Посмотрим на текущий статус и вызовем команду git status. Не обязательно писать всё заново, можно просто стрелочками выбрать то, что уже вызывали. Далее мы видим некоторое сообщение от Git:

- мы находимся на ветке мастер — к этому мы вернёмся на второй лекции;
- появились некоторые изменения, которые пока не сохранены;
- изменён файл hello world.

Мы можем сохранить его новое состояние. Для этого достаточно опять сказать Git, чтобы в текущее сохранение он добавил файл hello world. Причина заключается в том, что файлов может быть несколько, они могут быть изменены, и текущее сохранение требуется сделать только для некоторых файлов. Даже перед тем, как их сохранить отслеживаемые файлы, надо вызвать команду git add.

Напишем git add. Далее ещё раз смотрим на статус. Снова строка hello world стала зелёной. То есть у нас есть какой-то изменённый файл, который теперь можем сохранить. Пишем git commit и указываем новое сообщение, например, «Добавили новую строку». Нажимаем Enter — появилось некоторое сохранение.

[00:45:13]

Создаём журнал изменений

Теперь у нас есть уже два сохранения, с которыми можно работать. Появилась возможность переходить от одной версии файлов к другой, что нам и хочется от системы контроля версий. Но сначала посмотрим, какие версии существуют. Для этого используется специальная команда git log. Log — это просто журнал изменений, каких-то событий, которые у нас хранятся.

Набираем git log и видим немного странный вывод, к которому вы скоро привыкнете. Главное, что у нас есть некоторые фиксации (сохранения) — первое сохранение и второе сохранение.

К каждому сохранению прикреплены текстовые комментарии, которые мы оставили. И уже можно понять следующее:

- первое сохранение относится к добавлению новой строки;
- второе сохранение относится к созданию нового файла.

Сейчас мы находимся на первом сохранении. Но если хотим перейти к предыдущей версии файла, воспользуемся для этого указанным commit. Это очень непонятное сочетание букв и цифр, по которым можем перейти. Чтобы перейти к какой-то версии, сохранению, надо вызвать команду git checkout.

Сейчас это может показаться сложным из-за такого количества терминов. Однако, когда немного поработайте с ними на практике, то быстро привыкнете и легко будете ими пользоваться.

После команды «git — пробел — checkout — пробел» укажем, какое сохранение надо загрузить. Можем указать либо название целиком, либо первые четыре символа, что и позволяет Git. Первых четырёх символов достаточно, чтобы понять, какое сохранение надо запустить. Скопируем их и после git checkout укажем именно этот набор символов. Далее нажимаем на Enter — исчезла строка, которую мы писали. То есть версия файла вернулась к сохранённым.

Теперь можем вернуться к версии, которая нас интересовала, посмотреть на сохранение и выбрать, например, то, что было выше, затем снова вызвать git checkout и ввести символы. Видим, что случилась магия. Мы перешли к новой версии. У нас опять появилась строка, которую писали.

По сути, сейчас мы прошли весь путь создания файла и фиксации изменений в нём.

[00:48:32]

Возвращаемся в актуальное состояние

Мы сохраняли разные статусы своего файла и научились между ними переключаться. На самом деле файл создан, и с ним ничего не происходит. У нас есть его актуальная составляющая. Git же показывает предыдущие состояния файла, и путём вызова команды git checkout мы можем переключаться между разными версиями. Однако чтобы дальше работать с этим файлом, нам требуется вернуться в актуальное состояние. Git должен понимать, что мы начинаем что-то менять с той версии, которая была.

В дальнейшем поговорим про ветки, где работа будет производиться иначе. Но сейчас, чтобы всё работало и ничего не сломалось, требуется вернуться в актуальное состояние. Для этого надо не просто указать необходимый commit, а что-то иное. Мы сами напишем git checkout master. Master — это название ветки, в которой работаем. Поговорим об этом подробнее на следующей лекции. А пока запомним: если хотим продолжить работу, легко переключаясь между версиями, нам надо вернуться в актуальное состояние. Для этого пишем git checkout master, нажимаем Enter — всё работает, можем двигаться дальше.

Чтобы сохранение изменений стало привычным, повторим эту процедуру несколько раз. Но для дополнительной пользы параллельно познакомимся с языком разметки Markdown.

Если видели, как выглядит HTML-страница в браузере, то знаете, что текст, написанный на языке разметки, может выглядеть иначе, когда отображается, например, в окне браузера. То есть при открытии в браузере страница выглядит как сайт или красивая картинка, а при работе с ней появляются разные HTML- и CSS-теги, что смотрится очень странно и непонятно.

Соответственно, Markdown тоже язык разметки. И у него также есть два состояния:

- то, что вы написали;
- то, что видит конечный пользователь.

Чтобы увидеть второе состояние, нажмём в программе VS Code специальную кнопку, которая отобразит наш файл. Слева располагается то, что пишем мы, а справа — вид для конечного пользователя. Пока и слева, и справа — одно и то же. Потому мы не пользуемся никакими специальными символами и разметкой.

[00:51:35]

Особенности Markdown

Изучим особенности Markdown. Например, как, применяя специальные символы, редактировать и форматировать текст. Параллельно будем сохранять произведённые изменения.

Для начала разберём самое простое действие. Например, выделение текста курсивом. Если бы мы делали это в какой-нибудь программе типа MS Word или в другом редакторе, то выделили текст мышкой и нажали соответствующую кнопку или сочетание клавиш. При работе с Markdown всё выглядит несколько иначе, но не менее просто.

Добавим какое-нибудь новое слово, например, «курсив», и поставим точку. Чтобы это слово было написано курсивом, надо слева и справа от этого слова поставить звёздочку. Неважно, где стоят звёздочки — до точки или после. Всё зависит от того, какой блок текста хотите выделить курсивом. Обратите внимание, что в левой части экрана около слова «курсив» стоят звёздочки, а в правой — никаких звёздочек уже нет, слово просто написано курсивом.

Итак, сохраним текущую версию и оставим заметку, что в этой версии добавлено выделение курсивом. Посмотрим, как теперь сохранить текущую версию файла.

Обратите внимание, что после всех изменений я всегда нажимаю Cmd+S или Ctrl+S. Это необходимо, чтобы изменения сохранились, записались в файл, который, кстати, назван с ошибкой. Следующим коммитом переименуем его.

Чтобы сохранить эти изменения, надо совершить несколько действий, которые мы уже делали.

1. Первым действием в сохранение добавим файл. Пишем git add и после этого указываем тот файл, который требуется сохранить. Указываем, например, hello world, нажимаем Enter. Никаких сообщений об ошибках нет.

2. Теперь создадим commit для сохранения текущего статуса и добавим ему некоторое сообщение. Например, «Добавили выделение курсивом».

3. Нажимаем Enter.

Вышло сообщение об опечатке — commit написан с двумя буквами о, это не команда Git. Ничего страшного, вводим команду заново. Просто нажимаем стрелочку «вверх» и смотрим то, что я ввёл. Исправляем опечатку и нажимаем Enter. В результате всё прошло успешно. У нас появился новый commit о выделении курсивом.

Переименуем наш файл и посмотрим, что при этом произойдёт внутри системы контроля версий. Мы можем нажать правую кнопку, переименовать и исправить получившуюся опечатку. Назовём правильно — hello world. Посмотрим на текущий статус Git после того, как переименовали файл.

Вызовем команду и увидим довольно странное сообщение от Git. Он говорит, что у нас удалён hello wrold, написанный неправильно, и появился какой-то не отслеживаемый файл. Разберёмся, что происходит при переименовании файла.

Git следит за файлами по их именам. Если у нас больше нет файла с каким-то названием, Git будет считать, что он удалён. Вместо этого появится новый файл, название которого Git пока неизвестно. Он его ещё не отслеживает. В этом случае мы можем добавить свой файл с правильным именем в отслеживаемый, сделать новый commit, и всё снова будет в порядке.

Введя git add, создадим файл, который называется правильно. Нажимаем Enter. Напишем git commit, добавим сообщение «Исправили название файла», и кликаем на Enter. В результате у нас появился ещё один commit.

[00:56:20]

Возврат к предыдущим сохранениям

Теперь посмотрим, что случится, если перейти к предыдущим сохранениям. Вернёмся к первому сохранению, когда мы только создали свой файл. Что случится с названием нашего файла?

Через git log вызовем блок изменений и возьмём какой-нибудь самый первый commit. Если у нас многострочный вывод и введено слово and, достаточно нажать кнопку Q (quit), чтобы выйти из этого режима и вернуться к терминалу.

Через команду git checkout перейдём к следующему сохранению. Постепенно вы запомните все эти команды. Теперь перейдём к какой-нибудь старой версии нашего файла.

Обратите внимание, что файл переименовался, текущий файл удалён. Если открыть файл с исходным состоянием, мы увидим файл уже с неправильным названием. У меня состояние, которое было сохранено, что не удивительно. Git сохраняет все состояния папки, всё, что происходило внутри. Если вы вернулись к моменту, когда файл назывался неправильно, Git вернётся в это состояние. Это на самом деле очень удобно, хотя иногда немного пугает.

[00:58:03]

Добавляем выделение полужирным

Вернёмся к актуальной версии и продолжим изучать Markdown. Как к ней вернуться, вы помните. Если мы находимся в ветке master, то пишем git checkout master. Про ветки мы поговорим в следующей лекции. А пока просто запоминаем, что таким образом можно вернуться к актуальной версии.

Нажимаем git checkout master — всё вернулось. Закроем неправильные файлы и откроем правильный. Выведем предварительный просмотр на экран.

Выделим текст жирным, добавив слово «полужирный». Для этого с каждой стороны слова поставим по две звёздочки. Сохраним это. Напишем git status и посмотрим, что об этом думает Git. Кстати, если стоит **белая точка**, то **текущие изменения не сохранятся**. Нажимаем Ctrl+S, набираем git status и видим, что файл hello world изменился.

Чтобы добавить наши изменения в commit, введём git add hello world.

Когда мы возвращались к предыдущей версии, Git создал в папке файл, который мы удаляли. А когда вернулись, он его не удалил, а оставил. Можно этот файл удалить, но пока оставим — он будет полезен при работе с двумя файлами параллельно. Важно, что при выборе автозаполнения нам надо просто указать правильное написание этого файла. Например, hello world.md. Так мы будем работать с правильно именованным файлом.

Объект добавился, сохраним его через git commit. Добавляем сообщение «Добавили выделение полужирным». Нажимаем Enter — появилось ещё одно сохранение.

[01:00:36]

Создаём списки

Вернёмся к Markdown. Обратите внимание, если в левом окне просто нажать Enter, язык Markdown посчитает, что текст будет писаться слитно, то есть нового абзаца не будет. Чтобы новый абзац появился, в текст надо вставить пустую строку.

Помимо выделения текста, зачастую требуется написать список. В языке разметки Markdown списки создаются очень легко. Чтобы появились точки — пункты списка, просто ставим звёздочку и пробел. Добавление пробела уже вызовет не выделение курсивом, а элемент списка.

Добавим какой-нибудь список. Например, «Элемент 1», «Элемент 2» и «Элемент 3». В результате мы видим, что слева список написан со звёздочками, а справа — без них, но с точками.

Сохраняем изменения, чтобы они записались в файл, и повторяем уже знакомые действия. Добавляем наш файл в сохранения и сохраняем текущее изменение через git commit -m, написав сообщение «Добавили ненумерованные списки». У нас появился commit.

Ещё можно создать нумерованные списки. Нумерованные списки создаются легко: просто пишем «1. Первый элемент нумерованного списка», а потом — «2. Второй элемент».

Таким образом, при всей своей мощности Markdown очень лёгкий и удобный язык. Если нужен список с буллитами, ставим звёздочки. А если требуется нумерованный список, пишем 1, 2 и т. д. Так список станет таким, каким мы хотим его видеть.

Сохраним это изменение. Не пугайтесь часто добавляемых commit. Во-первых, нам надо к ним привыкнуть, запомнить, как они работают. Во-вторых, так удобнее смотреть историю изменений. Мы будем видеть всю историю создания этого файла и возвращаться ко всем версиям, которые понадобятся.

[01:03:54]

Добавляем заголовки и разделы

И последнее, что мы сегодня рассмотрим — заголовки и разделы в языке Markdown. Добавим сверху какой-нибудь большой заголовок. Для этого поставим решётку и напишем в файле «Первый файл по контролю версий».

Таким образом, слева мы видим решётку и какой-то текст, а справа — отформатированное отображение текста. Текст написан большим шрифтом, появилась отсечка, некоторая строка, отделяющая заголовок от остального текста.

Добавим заголовки других уровней, которые будут чуть меньше, например, выделение курсивом и полужирным вставим в блок «Выделение текста». А списки, которые мы создавали, добавим в блок «Списки». Удалим hello world, так как в нашем файле он уже не нужен.

В нашем файле появилась некоторая структура: заголовок и разделы с выделенным текстом. Мы знаем, как выделить текст курсивом и полужирным, и умеем создавать списки. Остался текст начала занятия. Удалим его, чтобы файл стал красивым.

Используя способы написания текста, напишем инструкцию, которой можно будет пользоваться. Например, «Для выделения текста курсивом оформите его звёздочками». Оставим это для вашей самостоятельной работы. В этом commit сохраним только добавление заголовков.

Мы можем опять добавить наш файл к сохранению — git add — и сохранить его. Перепишем файл, чтобы не совершать ту же опечатку. Напишем, например, «Добавлены заголовки» — коммит создан. Чтобы посмотреть историю произведённых и сохранённых нами изменений, введём команду git log.

Кстати, последний commit содержит «Добавили енумерованные списки». Мы же сделали коммит с заголовками, что же случилось? Внимательнее студенты, которые немного знакомы с контролем версий, скажут, что произошло. А те, кто ещё не заметил, что случилось, я подскажу.

Наверху рядом с hello world.md стоит белая точка. Так Visual Studio Code показывает, что мы не сохранили изменения в своём файле. А Git смотрит не на то, что вы делаете, а на уже записанные файлы. Мы не записали свои изменения. Чтобы понять, отличаются ли наши файлы от того, что уже сохранено в commit, вызываем команду git diff. Git diff показывает (difference — «разница») разницу между текущим состоянием файла и тем, что уже сохранено.

Набираем git diff — команда ничего не показывает. Это значит, что сохранённый файл полностью идентичен текущему состоянию. Если сохраним изменения, то есть запишем их внутрь файла, нажав Ctrl+S или Cmd+S и вызвав git diff ещё раз, увидим все произведённые изменения.

Обратите внимание, Git сравнивает записанные версии файлов. Если что-то меняете, но изменения не записали, для Git файл не изменится.

В итоге:

- исчез hello world — видим минус;
- добавился заголовок «Первый файл по контролю версий»;
- добавилась пустая строка;
- добавился заголовок «Выделение списка»;
- добавился заголовок «Списки»;
- удалилась строка «Начинаем наше знакомство с контролем версий».

Таким образом, мы можем посмотреть все изменения внутри этого файла.

Нажимаем Q, чтобы выйти из этого режима. Вводим git add для изменения файлов, и выполняем сохранение. Пишем git commit, который мы уже пытались сделать, но он не прошёл. Если файл не сохранён, commit не пройдёт. Теперь мы сделали этот commit.

Видим, что один файл изменён, пять строк добавлено, а две — удалены. Git показывает, что случилось с этим файлом.

Теперь посмотрим на историю изменений, введя git log. Мы видим, что последнее сохранение — это «Добавлены заголовки». Соответственно, всё выглядит так, как и должно быть.

[01:10:40]

Практическое задание

На этом завершим наше знакомство с Git. Вы получили много информации, которую теперь надо переварить.

Сделайте на своих компьютерах то, что выполнялось на лекции.

1. Составьте инструкцию для языка Markdown:
    - чтобы сделать текст курсивным, надо...;
    - чтобы сделать текст полужирным, надо...;
    - чтобы добавить списки, надо...;
    - чтобы вставить заголовок, надо...
2. Сохраните все изменения в виде отдельных коммитов.

Вы быстро привыкнете к работе с терминалом. Не будете писать команды заново, а станете пользоваться стрелочками «вверх» и «вниз». Поймёте, насколько быстрее можно работать в терминале, чем бродить мышкой по пунктам меню. На самом деле опытные программисты практически не пользуются мышкой, а всё делают только на клавиатуре.

Теперь вернёмся к нашей презентации и посмотрим завершающие слайды.

[01:12:15]

Повторяем команды и элементы разметки в Markdown

Для вас мы сделали слайд, где указаны команды, используемые в Markdown, а также некоторые элементы разметки. Например, жирный текст выделяется двумя звёздочками, а курсивный — одной. Для зачёркнутого текста используется тильда. И так далее.

Посмотрите в интернете мануалы, инструкции по использованию языка Markdown. Он позволяет использовать множество элементов форматирования, а также удобным образом добавлять фотографии, ссылки и таблицы. Поэтому воспользуйтесь всеми указанными элементами, посмотрите, что ещё может делать Markdown, и дополните файл, который мы создали.

Ещё раз пробежимся по тем командам, которые мы делали. Важно запомнить эту информацию.

Первая команда — git init. Она позволяет сделать из нашей папки репозиторий. Репозиторий — это папка, в которой настроена система контроля версий. Если пользуетесь системой контроля версий Git, вам достаточно вызвать в терминале свою программу Git, поставить пробел и передать параметр и команду init. Вы говорите Git: «Инициализируй». Это надо выполнить в той папке, из которой хотите сделать репозиторий. То есть сначала переходите в нужную папку.

Например, на Windows в документах есть папка testgit. Внутри неё набираете git init, и эта папка становится репозиторием. Это то, с чего начинается вся магия — Git начинает работать с вашей папкой.

Далее идёт git add. Вспоминаем, как пользоваться командой add и добавить файл к отслеживанию в этой программе. Снова вводим команду git и команду add, после чего

указываем файл, который надо добавить. Обязательно добавляем расширение — в этом примере это .txt, то есть обычный текстовый файл. До этого было расширение .md, так как работали с языком разметки Markdown. Обратите внимание, для Git нет никакой разницы, с какими файлами он работает. Вы можете сохранять абсолютно любые файлы, используя команду git add, то есть добавлять их к отслеживанию.

Далее мы рассматривали команду git commit. Эта команда сохраняет текущее состояние — некий сейф в игре. Мы фиксируем текущее состояние наших файлов и оставляем некоторое текстовое напоминание, что конкретно здесь сделали. Пишем git commit, добавляем - m, сохраняя некоторый месседж, комментарий, и указываем тот текст, который напомнит, что в этом файле происходит.

В конце мы рассмотрели команду git diff как difference — отличие, разница. Она позволяет посмотреть, чем отличается наш текущий файл от того, что уже сохранено, или закоммичино. Мы видим, что минусом показываются удалённые строки, а плюсом — добавленные.

Ещё у нас была команда git log. Она позволяет посмотреть журнал всех изменений. На слайде, кстати, сообщения к commit уже указаны правильно — на английском языке. Лучше, если в своей практике вы постепенно начнёте пользоваться английским языком, а не русским. Но для начала это не важно. Главное — научиться пользоваться этими командами. То есть сначала пишем git, ставим пробел и указываем команду, которую надо выполнить. Напоминаю, что git log выводит журнал всех изменений, которые мы делали.

Таким образом, в качестве практики вам надо проделать всё то, что мы делали на лекции. А именно: дополнить инструкцию. В итоге у вас появится готовый файл и репозиторий, папка, в которой будут сохраняться версии, весь процесс работы над этим файлом. На семинарах мы подробнее поговорим об этом, выполним некоторые практические задания, которые позволят быстрее начать работать с системой контроля версий Git.

[01:17:53]

Заключение

Я очень рад, что вы посмотрели нашу лекцию до конца. Увидимся на следующей лекции. Всем пока!