GeekBrains

Система контроля версий Git — advanced

Методичка к уроку 3
Практики и инструменты для работы в Git

Оглавление

Введение

В первой лекции этого курса мы с вами говорили о работе с удалёнными репозиториями. Начали с обзора и сравнения систем GitHub и GitLab, поговорили о способах подключения к удалённым репозиториям, изучили механизмы настройки связей с ними, а также о разрешении типичных проблем, которые могут возникать при работе с удалёнными репозиториями и о путях их решения.

Во второй лекции мы говорили о работе с изменениями: о том, какие бывают изменения в Git, каковы их жизненный цикл, как их просматривать на разных стадиях этого жизненного цикла, как отменять сохранённые и несохранённые изменения, как откладывать изменения и даже как отменять слияние веток.

В этой лекции вы познакомитесь с практиками и инструментами работы с системой контроля версий Git. Мы изучим четыре темы:

Модели работы с ветками. Мы рассмотрим две наиболее популярные в настоящее время модели работы с ветками в Git. Они называются git flow и trunk based. Эти модели используются в большинстве современных проектов и команд разработки.

Процесс работы в команде. Как в целом выглядит процесс работы с Git в команде от начала — подключения нового пользователя к репозиторию, до конца

— до попадания выполненной им задачи в продакшн — на боевой сервер или на рабочий сайт, на котором размещается продукт для пользователей.

Общепринятые правила. В этом блоке вы познакомитесь с некоторыми наиболее важными общепринятыми правилами работы с Git, придерживаясь которых вы сможете избежать большинства типичных проблем командной работы.

Командная строка и среды разработки. Мы сравним работу с Git в командной строке и в разных средах разработки, чтобы вы из сред разработки тоже могли работать с Git, несмотря на то, что наиболее полноценным вариантом такой работы является работа именно из командной строки.

Модели работы с ветками в Git

В этом блоке мы с вами разберём отличия двух наиболее популярных моделей ветвления — моделей, которые устанавливают правила работы с ветками в командах — “git flow” и “trunk-based”.

Начнём мы с ответа на важнейший вопрос: “Зачем нужны модели ветвления и почему нельзя пользоваться ветками тогда, когда это просто нужно?”

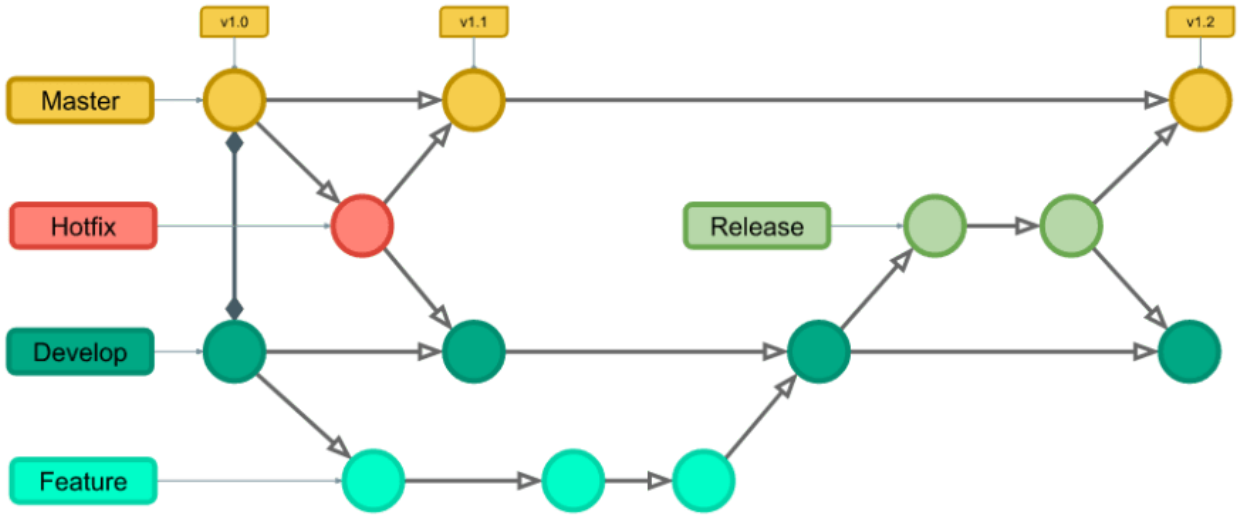
Во-первых, выбор той или иной модели необходим для того, чтобы в рамках команды было единое понимание того, как происходит ветвление, в каких случаях создаются ветки, как их правильно называть, сколько базовых веток есть в проекте, как они называются и для каких целей используются.

Если команда будет соблюдать единый порядок, единые принципы при работе с Git-репозиторием, то в этом репозитории всё будет систематизировано и понятно. И новые члены команды, как, собственно, и текущие, смогут легко разбираться в истории изменений и в ветках.

Во-вторых, обе модели ветвления, которые мы будем рассматривать, придумывались с целью ускорения разработки и так называемого деплоймента — то есть выкатки сделанных изменений и выполненных задач в production, например, на боевой сервер, если это веб-приложение, или в виде новой версии

установочного файла, если это мобильное приложение. Проблема скорости разработки стояла и стоит всегда и во всех компаниях. Бизнесу требуется, чтобы задачи решались как можно быстрее, чтобы результат их решения как можно быстрее был виден и доступен пользователям того или иного сервиса.

Итак, существуют две самые популярные модели ветвления, используемые большинством команд разработки, которые применяют систему контроля версий Git. Первая система — более классическая, называется “Git flow”. “Flow” — от слова “поток”, имеется в виду поток непрерывных изменений и выкатки — то есть deployment-а — программного обеспечения. Обычно ветки в этой модели выглядят примерно таким образом:



Источник изображения — https://devsday.ru/blog/details/46338

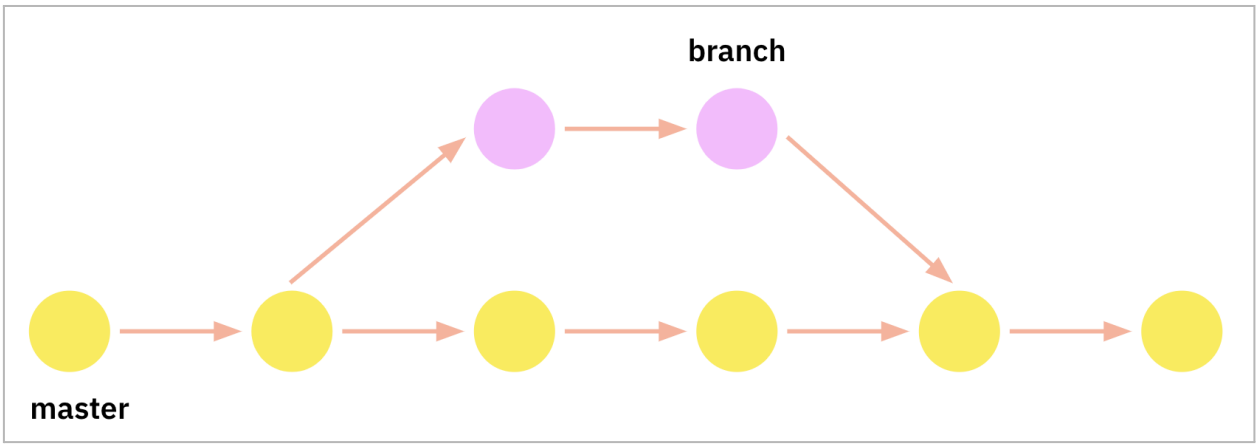
Суть этой модели состоит в том, что у вас есть ветка “master”, которую вы тщательно оберегаете от вливания в неё непроверенных изменений. Есть также ветка “develop”, в которую вливаются все свежие изменения и в которой проверяется общая работоспособность кода. Эту ветку в разных проектах могут называть по-разному — “dev”, “devel” или как-то ещё. И вот от неё уже отпочковываются ветки, в которых разрабатываются различные фичи продукта, над которым работают программисты. Как только работа над какой-то задачей завершается, результат вливается в ветку “release”, уже тщательно тестируется и потом заливается в ветку “master”.

Также в такой системе часто присутствует ветка для внедрения быстрых и мелких изменений. Её называют “hotfix” или “hotfixes”. Hot — это горячий, fix —

это исправление. Эта ветка позволяет обходить основной процесс выкатки изменений, который может занимать несколько дней, а то и недель.

Основная особенность такой модели состоит в том, что в ней довольно много веток, которые живут долго. Во-первых, каждая фича лежит в своей ветке и может разрабатываться довольно долго. А во-вторых, есть несколько постоянных веток разного назначения. Их, кстати, может быть и больше. В одном из проектов, в которых я работал, была ещё отдельная ветка “test” для отдельной категории тестировщиков.

Вторая модель называется “trunk-based development” или сокращённо TBD. Trunk в переводе с английского означает “ствол”. То есть это разработка, основанная на едином “стволе” — единой ветке. Этой веткой является ветка “master”. Никаких других постоянных веток при этой модели не создаётся, а ветки для отдельных задач создаются ненадолго и удаляются после вливания в ветку “master”:



В идеальной ситуации в этих ветках реализуются не сразу целиком все задачи, а их части, которые можно быстро выливать по-отдельности в ветку “master” и которые можно в ней выключать. Это делается с помощью так называемых “feature flags” — специальных переключателей, с помощью которых можно включать или выключать тот или иной функционал в итоговой ветке “master”.

Управление включением или выключением тех или иных фич происходит посредством, например, конфигурации приложения. А в самом приложении пишутся условия, которые проверяют, установлен ли тот или иной флаг или нет, например, так:

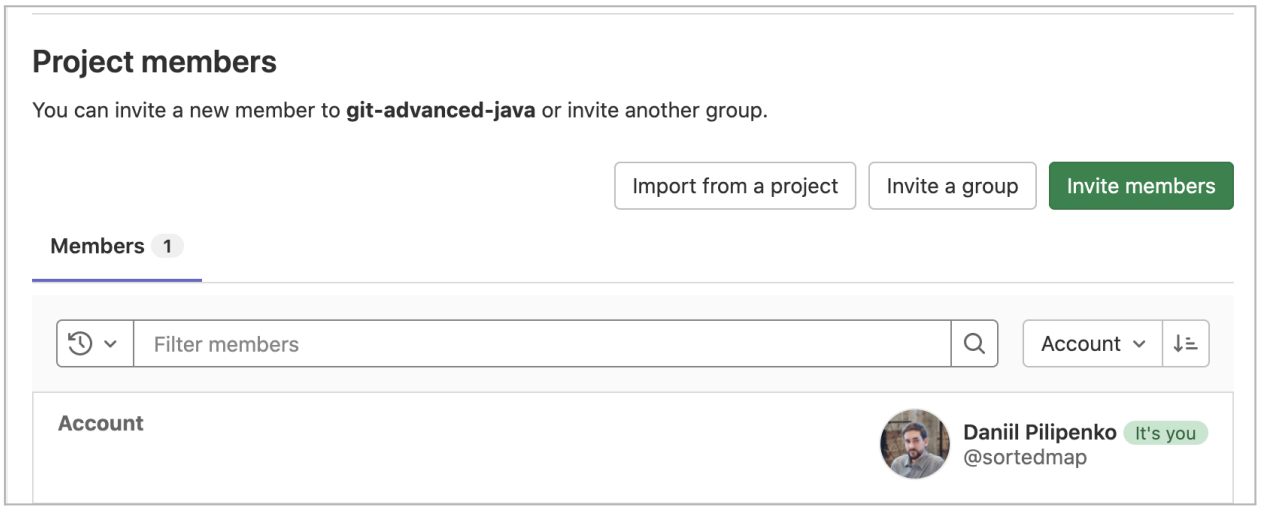
```
1 if (flags.get('showAdminMenu')) {
2     showAdminMenu();
3 }
```

Такая система позволяет разрабатывать и публиковать изменения гораздо быстрее, чем система Git Flow. И в настоящее время она считается более современной. Её преимущество — малое количество веток, которые “живут” тоже недолго. Это обеспечивает данной модели высокую гибкость, удобство и в такой модели работать сильно проще.

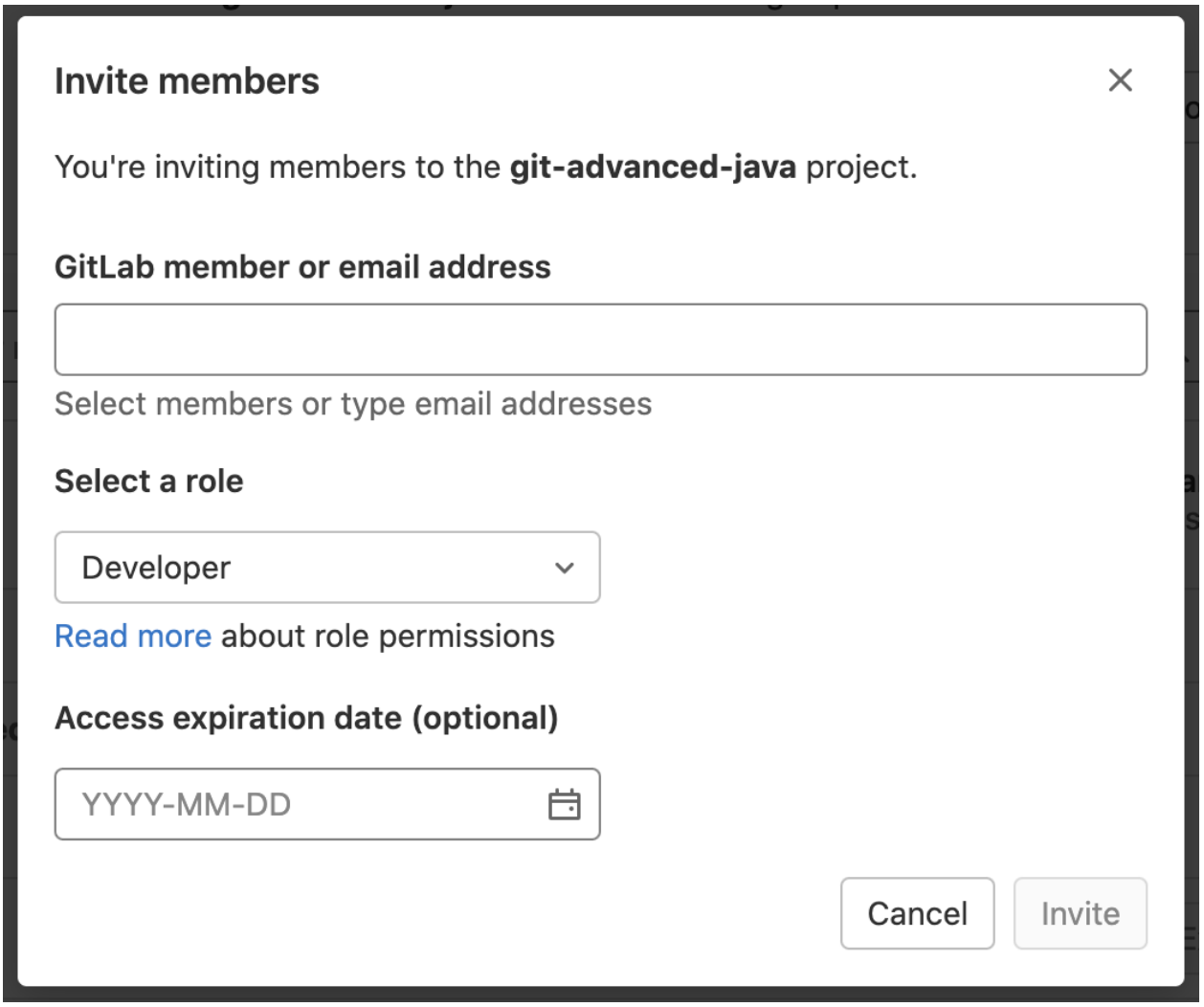
Процесс работы в команде

Лучше всего процесс реальной работы в команде показывать на практике, поэтому в этом разделе мы перечислим лишь основные его этапы и опишем их особенности, которые важны для практической работы в Git на примере Gitlab.

Подключение к проекту. Работа в команде начинается с подключения нового пользователя к репозиторию. Представьте, что вы — администратор проекта, и вы приглашаете нового разработчика в проект. Для этого обычно нажимают кнопку “Invite members” в разделе со списком пользователей проекта:

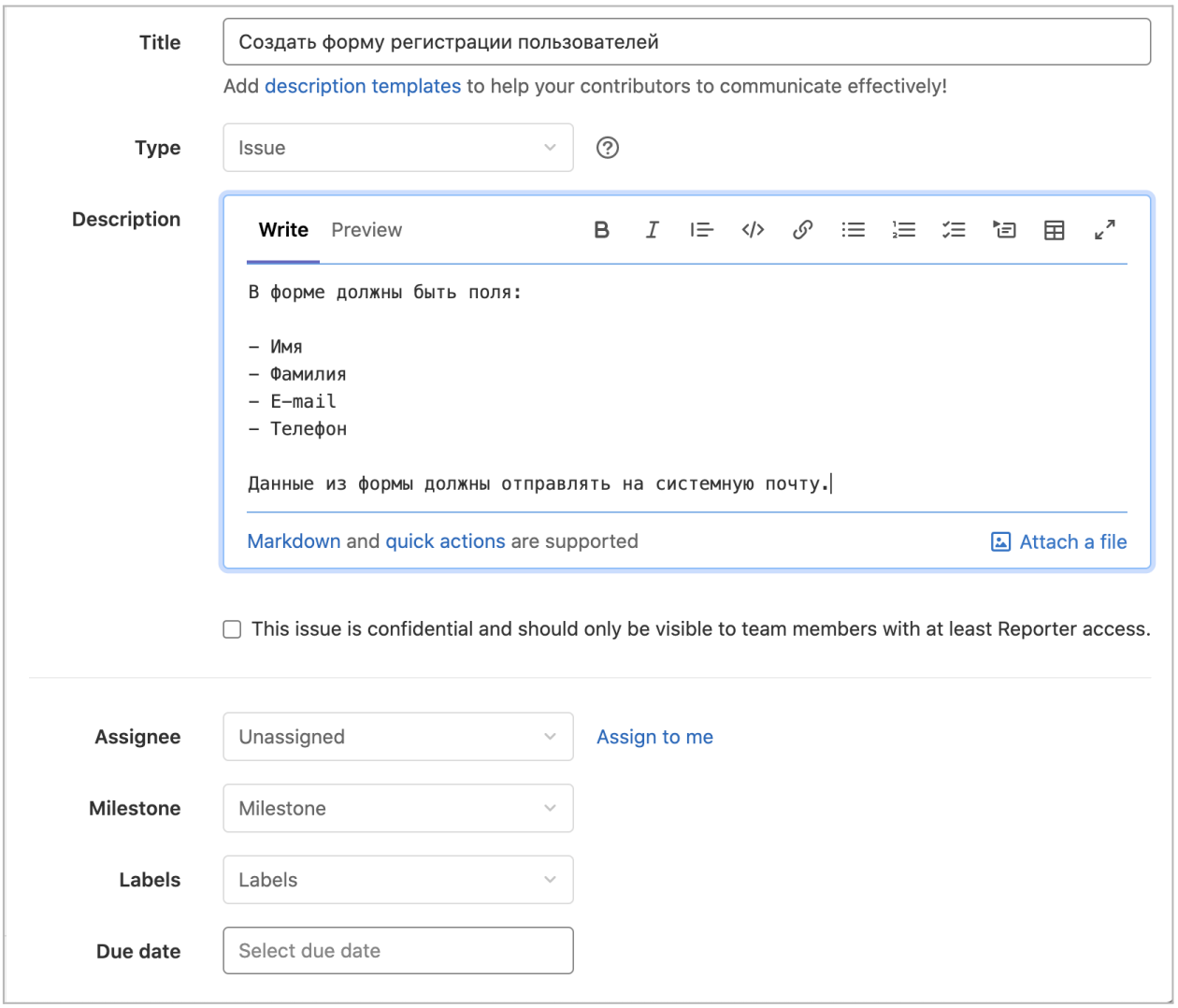


В появившемся окне указывают адрес электронной почты приглашаемого пользователя, если он ещё не зарегистрирован в этом Gitlab, или его логин, если он уже регистрировался ранее:

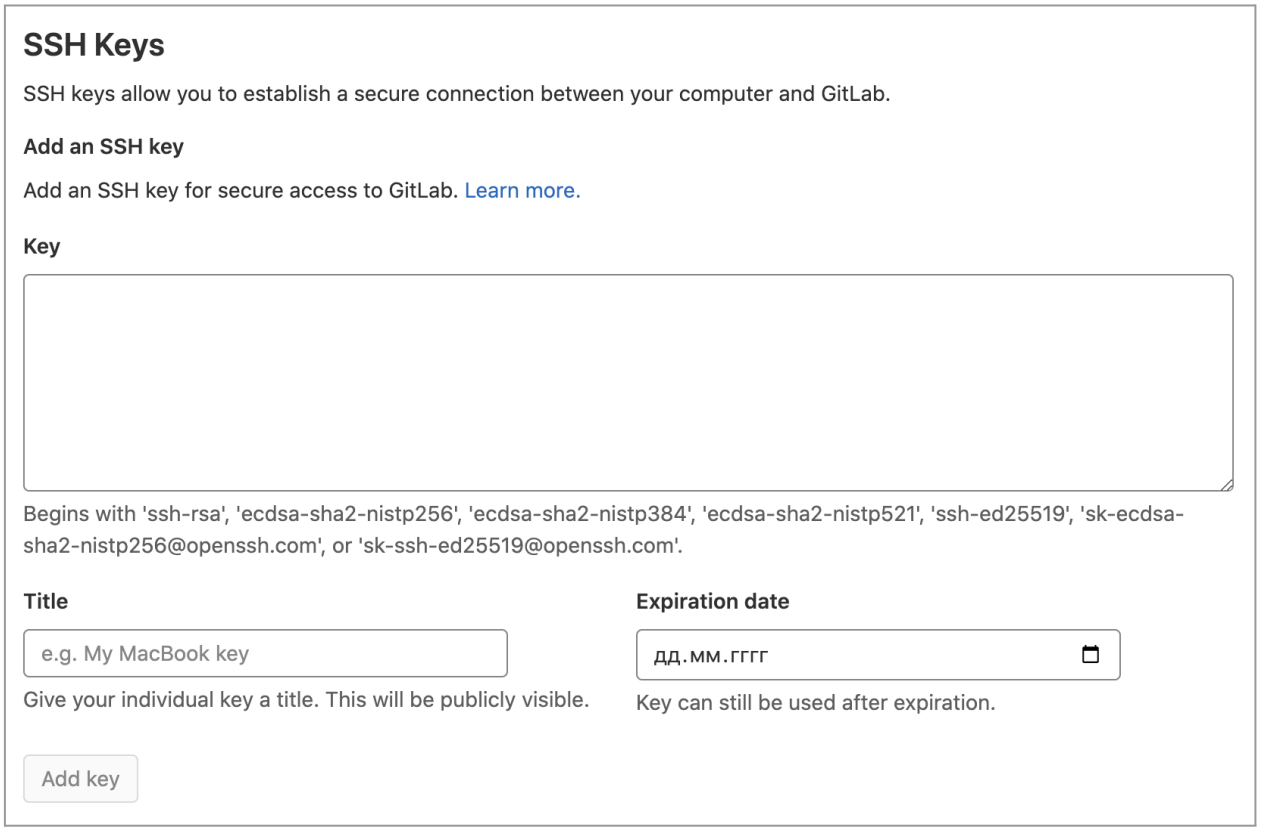


Также указывают роль пользователя. Если подключают разработчика, то обычно ему присваивают роль “Developer”, поскольку в ней собран оптимально необходимый набор прав доступа, необходимый разработчику для выполнения задач в системе Gitlab.

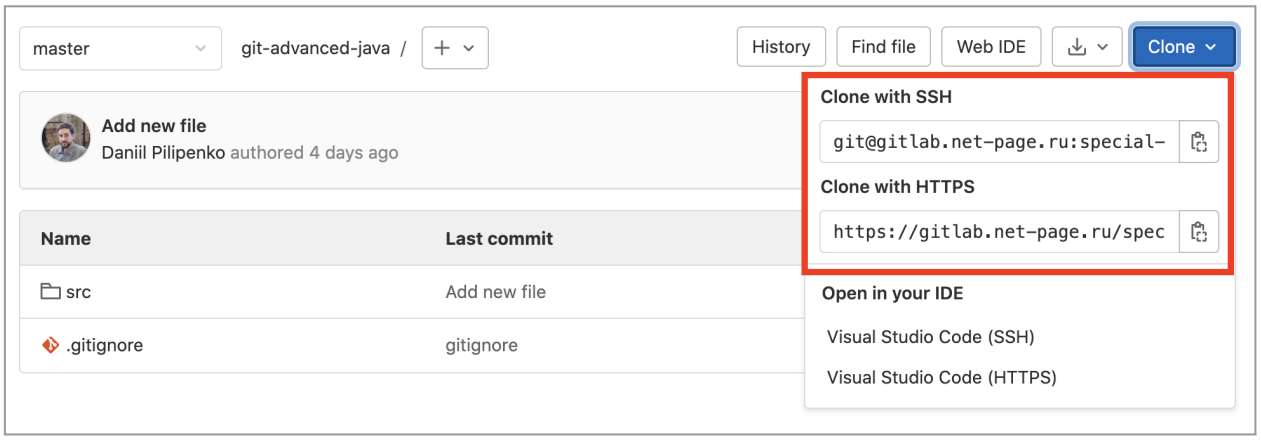
После того, как пользователь приглашён и добавлен к проекту, ему можно начинать ставить задачи. В разных командах это может делать менеджер проектов, системный аналитик, менеджер по продукту, владелец продукта, ведущий разработчик или тимлид. Создавая задачу в разделе “Issues”, он указывает нового разработчика в качестве исполните в поле “Assignee”:



После того, как задача поставлена, исполнитель увидит её у себя в GitLab в разделе “Issues”. Перед тем, как приступить к выполнению первой задачи, ему следует настроить доступ к репозиторию. Если он не делал этого ранее, ему необходимо сгенерировать на своём компьютере публичный и приватный ключи доступа и добавить публичный ключ в своём профиле в разделе “SSH Keys”:



После этого ему нужно скопировать путь к репозиторию:

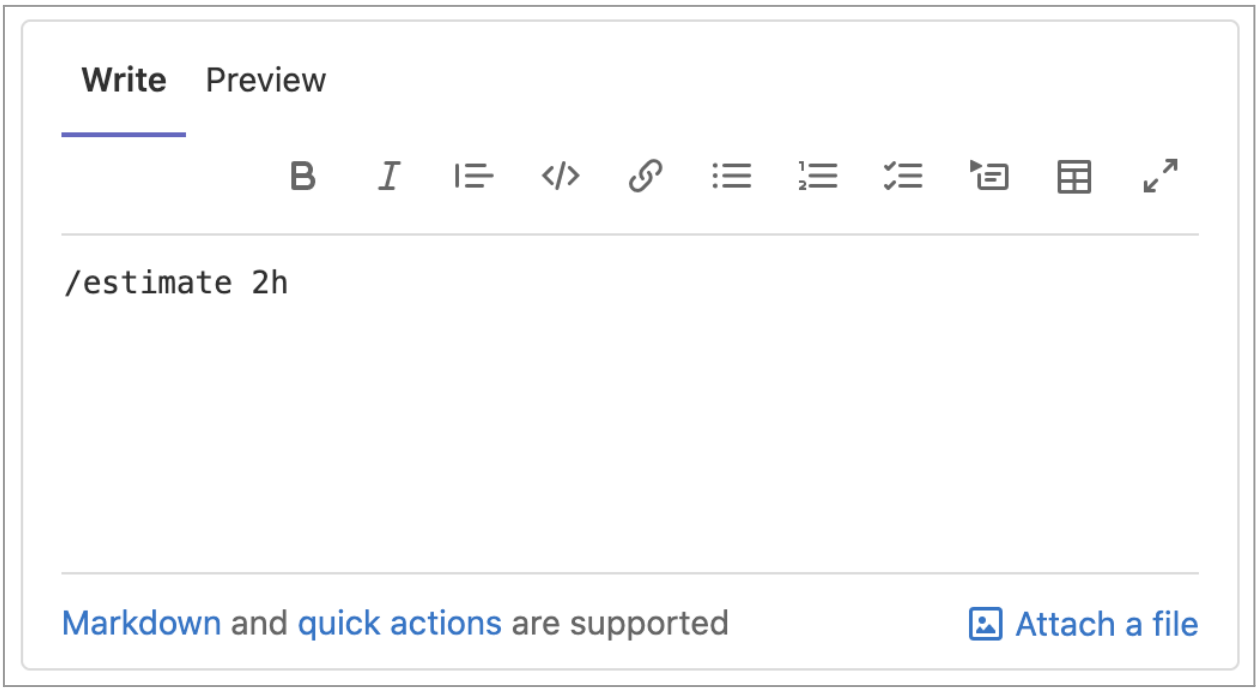


И затем склонировать его на свой компьютер:

```
git clone path
```

Если это веб-сайт или приложение, разработчику следует развернуть и/или научиться запускать его у себя локально, чтобы сразу проверять сделанные изменения до того, как они будут отправлены в удалённый репозиторий.

Перед началом работы над задачей мы также рекомендуем её предварительно оценить. В GitLab это делается при помощи комментария к задаче следующего вида:



Такой комментарий установит оценочное время выполнение задачи, равное двум часам. Рекомендуем это делать, чтобы вы начинали чувствовать, сколько времени занимает та или иная задача, и в будущем оценки сроков, которые вы называете, были близки к реальным. Это очень важный навык разработчиков уровней middle и senior, развивать который очень удобно начинать при помощи таск-трекера, например, GitLab.

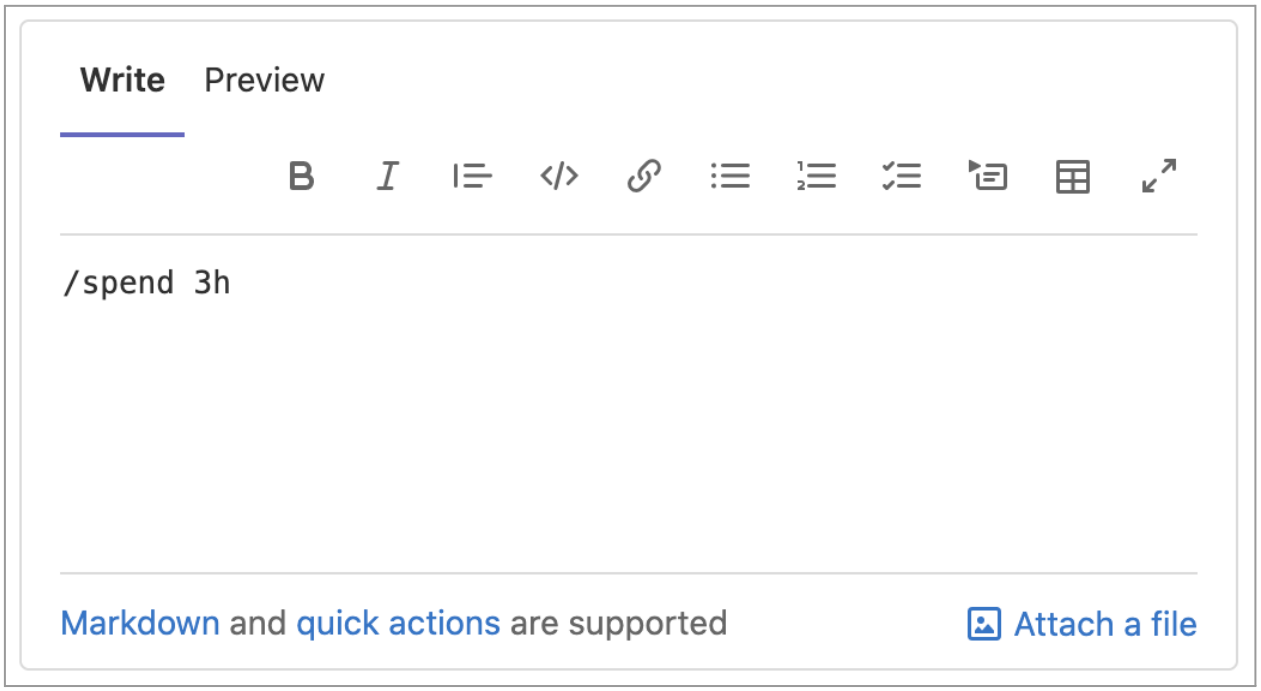
Когда всё готово к выполнению задачи, разработчику следует создать ветку, в которой он эту задачу будет выполнять. Название ветки должно начинаться с номера задачи и соответствовать сути выполняемой задачи, например, для задачи с номером 56 по созданию формы регистрации пользователей ветка должна называться примерно так:

```
56-users-reg-form
```

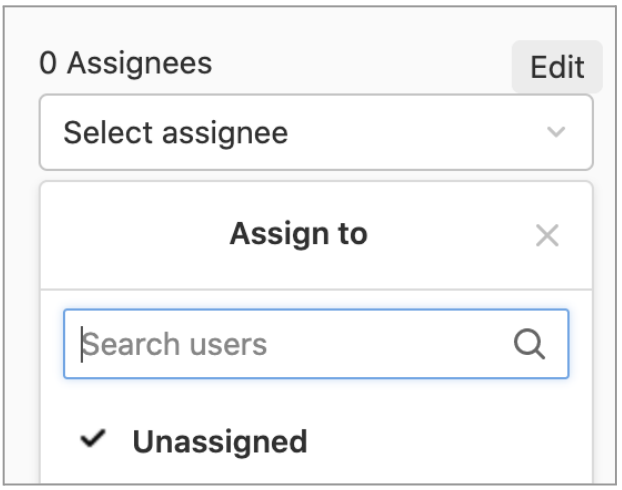
Номер в начале названия ветки автоматически привяжет коммиты из этой ветки к данной задаче, и они будут отображаться в ленте комментариев под ней. После выполнения задачи необходимо сделать коммит и отправить ветку в удалённый репозиторий, например, с помощью таких команд:

```
git add .
git commit -m 'users reg form created'
git push -u origin 56-users-reg-form
```

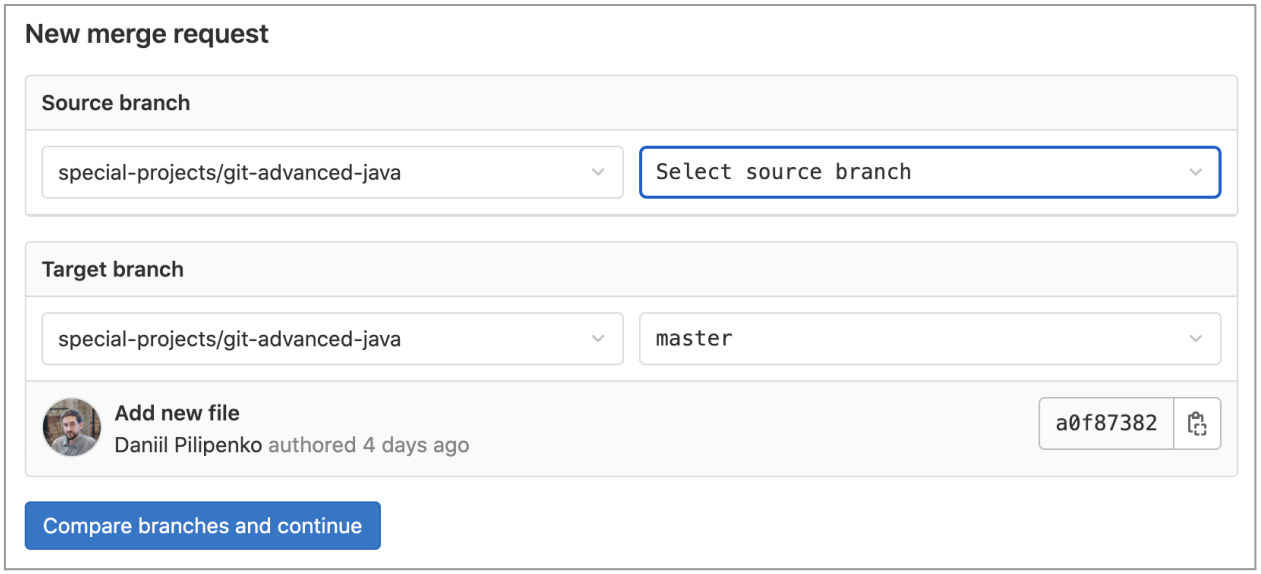
После этого вам следует отметить в задаче фактическое время её выполнения:



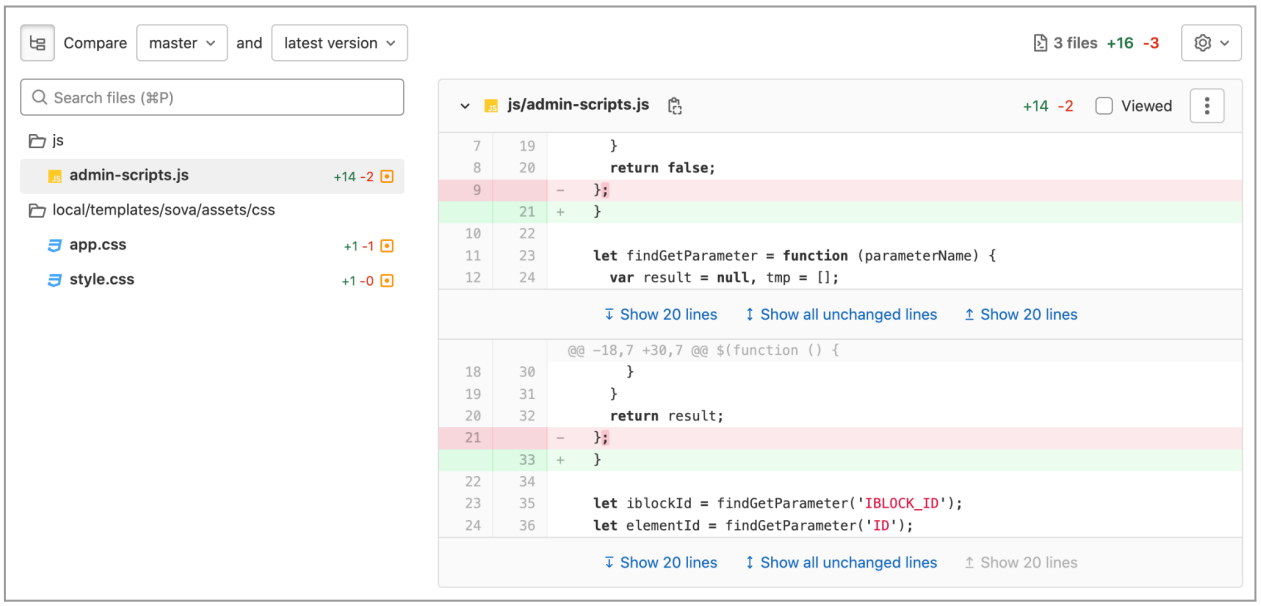
Задачу также можно перевести на того, кто вам её ставил, или на тестировщика, который должен будет её проверить. Обычно в GitLab это делается установкой нового “assignee”:



Можно также создать merge request и назначить на того, кто вам ставил задачу:



Merge request позволит этому человеку просмотреть внимательно все изменения, которые вы внесли в проект, провести так называемое “code review” и при необходимости прокомментировать любые строки вашего кода:



Как только merge request будет принят, изменения из вашей ветки попадут в ветку master, а исходная ветка при этом может быть удалена — в зависимости от модели ветвления, используемой в команде и проекте.

Вот так выглядит процесс работы с Git целиком — как с точки зрения программиста, так и с точки зрения тимлида или иного лица, которое подключает новых разработчиков к проекту, ставит им задачи, проводит code review и принимает merge request’ы. В разных компаниях могут быть какие-то особенности этого процесса, но, в целом, обычно процесс выглядит именно так.

Общепринятые правила работы в Git

Из этого блока вы узнаете о некоторых наиболее важных общепринятых правилах работы в системе контроля версий Git, которые сделают вашу работу удобной и понятной не только вам, но и вашим коллегам, с которыми вы будете работать в командах.

Именование веток. Первая группа правил относится к именованию веток в репозитории. Имена веток должны:

- Состоять из латинских букв в нижнем регистре, цифр и дефисов. Иные символы в именах веток использовать не рекомендуется.
- В начале содержать номер задачи, решаемой в этой ветке.
- Описывать суть задачи, решаемой в этой ветке.

Также, если в компании используется модель ветвления “git flow”, имена некоторых веток могут быть стандартными:

- master / main — главная ветка;
- dev / devel / develop — ветка, в которую вливаются результаты выполнения всех задач перед вливанием в ветку master;
- test / staging — ветка для выкладывания задач, которые необходимо протестировать перед вливанием в главную ветку или ветку dev;
- prod / production / releases — ветка для релизов;
- fixes / hot-fixes / hotfixes — ветка для “быстрых” и “горячих” изменений.

Содержимое коммитов. Коммиты не должны быть хаотичными. Они должны содержать либо завершённую работа, либо логически понятный фрагмент такой работы.

Также допустимо делать промежуточные коммиты, если выполнение задачи занимает больше одного рабочего дня, чтобы не потерять работу за прошедший день. Такие коммиты обычно принято делать в конце рабочего дня.

Комментарии к коммитам. Комментарии к коммитам должны отражать суть содержимого этих коммитов и соответствовать стандартам, принятым в компании, если таковые есть. Также рекомендуется их писать по-английски.

Один из самых распространённых форматов коммитов — так называемые “Conventional Commits”, в переводе с английского — “общепринятые коммиты”. Это специальный формат, предназначенный для унификации коммитов как для людей, так и для систем, которые анализируют коммиты. Например, систем, считающих статистику коммитов разных типов. Такие коммиты имеют строго регламентированную структуру: они обычно состоят из типа, двоеточия и некоторого описания. На официальном сайте по этому формату коммитов вы можете найти множество примеров того, как такие описания коммитов пишутся.

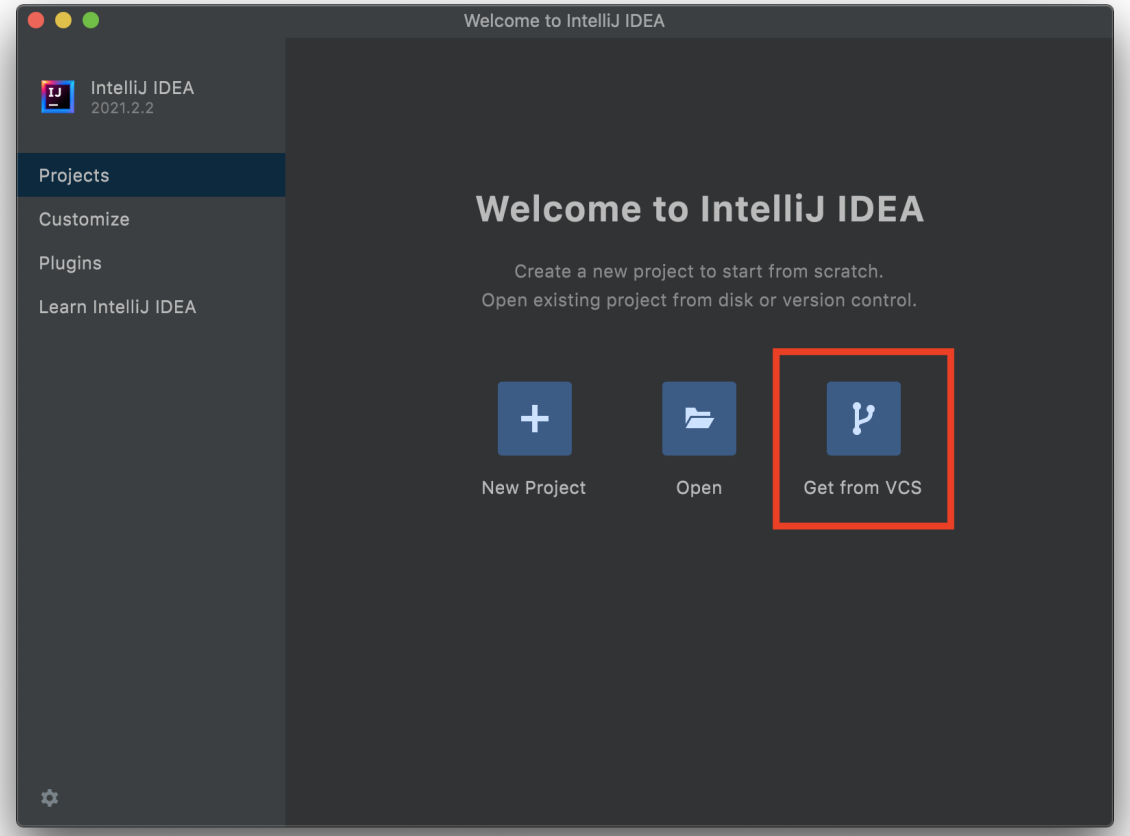
Командная строка и среды разработки

Вы уже знаете, что командная строка для работы с системой контроля версий Git — основной, самый полный и универсальный инструмент. В тоже время в реальной жизни разработчики обычно работают в специальных средах разработки.

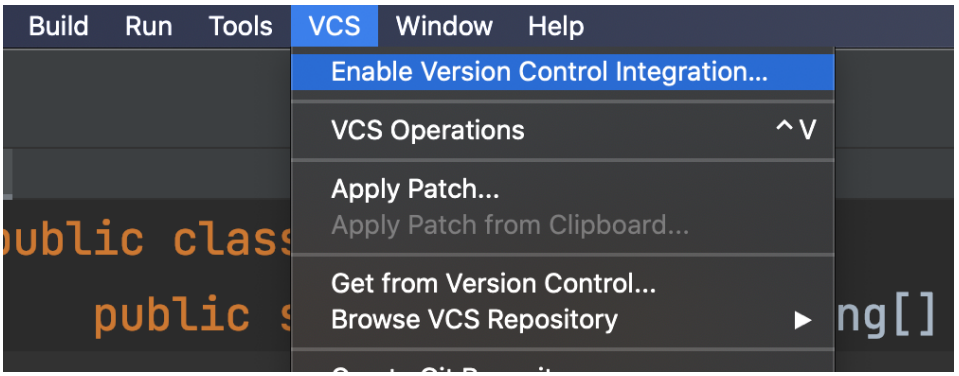
Среды разработки сокращённо называются IDE. Расшифровывается как Integrated Development Environment — интегрированная среда разработки. Такие среды бывают очень разные. Самые популярные — это среды разработки от компании JetBrains. К ним относятся, в частности, среды:

- IntelliJ IDEA — для разработки приложений на Java;
- PyCharm — для разработки приложений на Python;
- WebStorm — для разработки frontend- и веб-приложений в целом;
- PhpStorm — для разработки PHP- и веб-приложений в целом;
- ReSharper — для разработки приложений на C# и фреймворках ASP.NET;
- CLion — для разработки приложений на C и C++;
- GoLand — для разработки приложений на Go;
- RubyMine — для разработки приложений на Ruby.

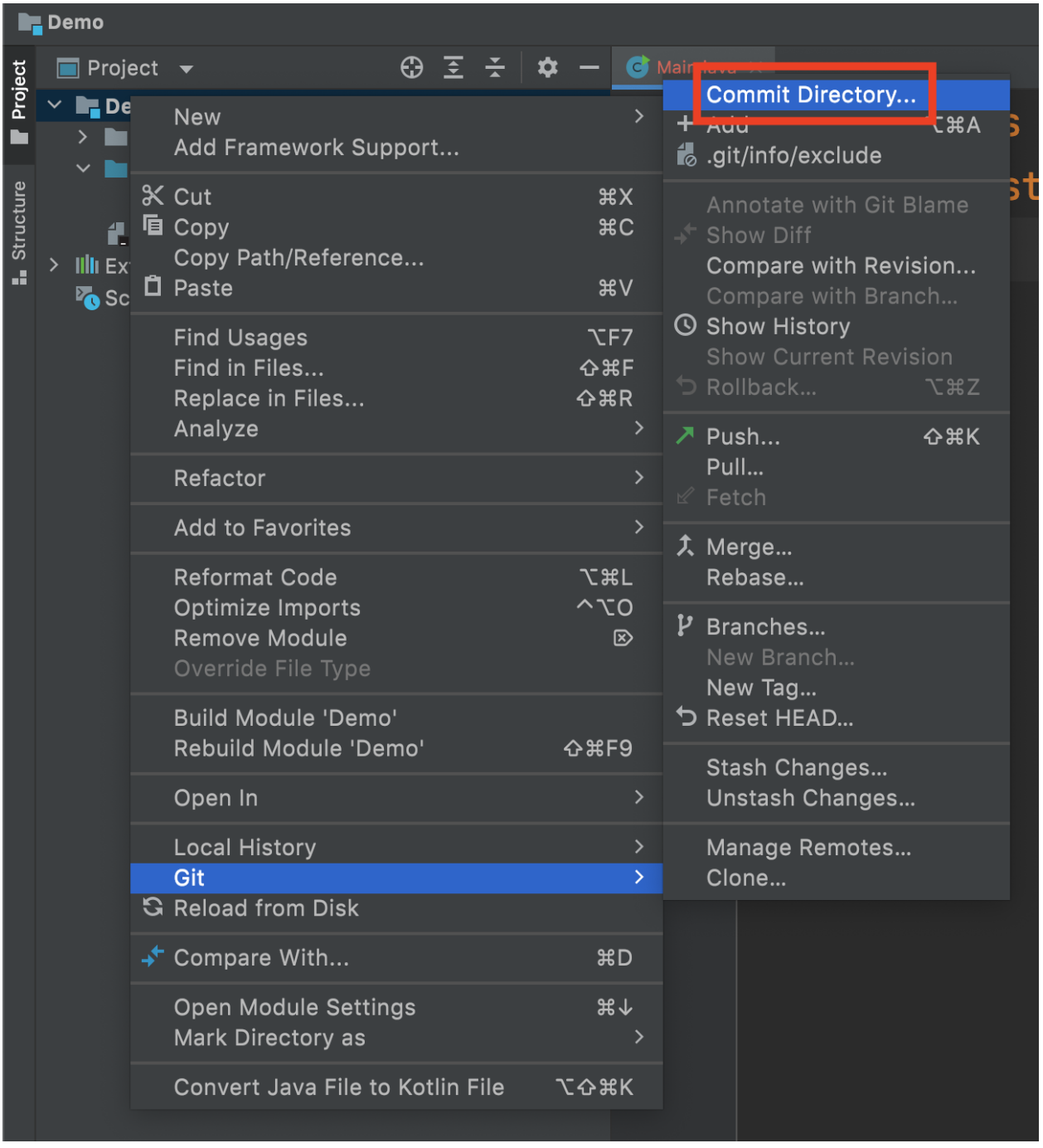
В этом блоке мы покажем вам, как можно работать с инструментами системы контроля версий Git из сред разработки от компании JetBrains на примере среды Intellij IDEA. Начнём с того, что в этой среде при создании проекта его можно не только создать с нуля, но и сразу склонировать из Git-репозитория или репозитория в другой системе контроля версий:



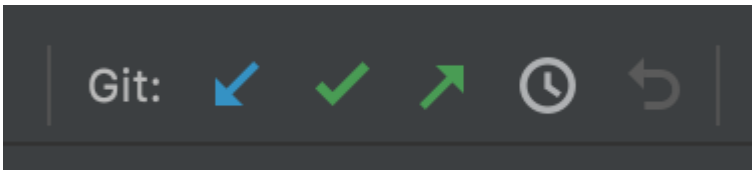
Вы также можете инициализировать репозиторий в существующем проекте, зайдя в меню “VCS” и нажав на пункт “Enable Version Control Integration”:



После того, как система контроля версий инициализирована, вы можете выполнять команды Git прямо из контекстного меню, которое появляется при нажатии на любой элемент проекта правой кнопкой мыши. Например, вы можете добавить в индекс и закоммитить сразу всю папку вашего проекта:

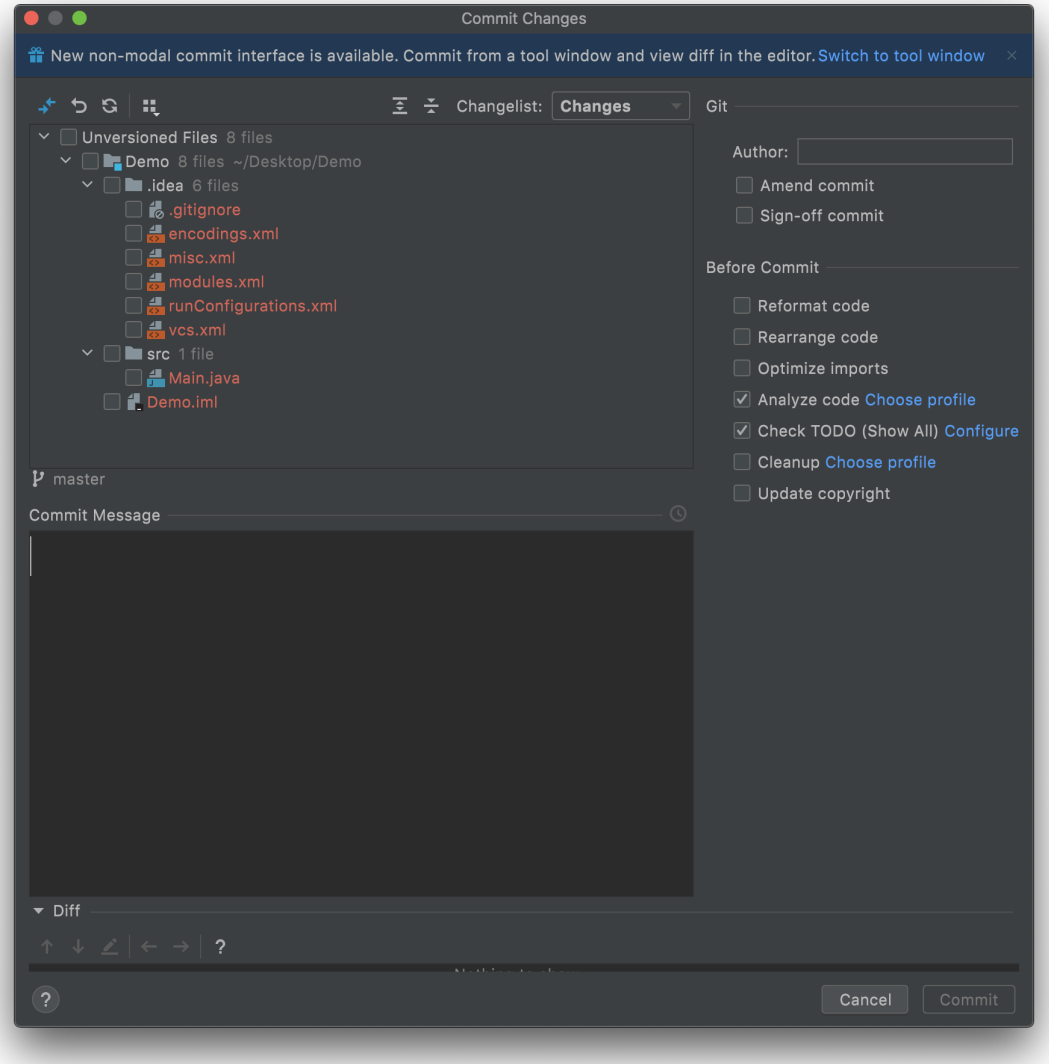


В правой верхней части среды разработки также есть несколько самых часто встречающихся команд Git:

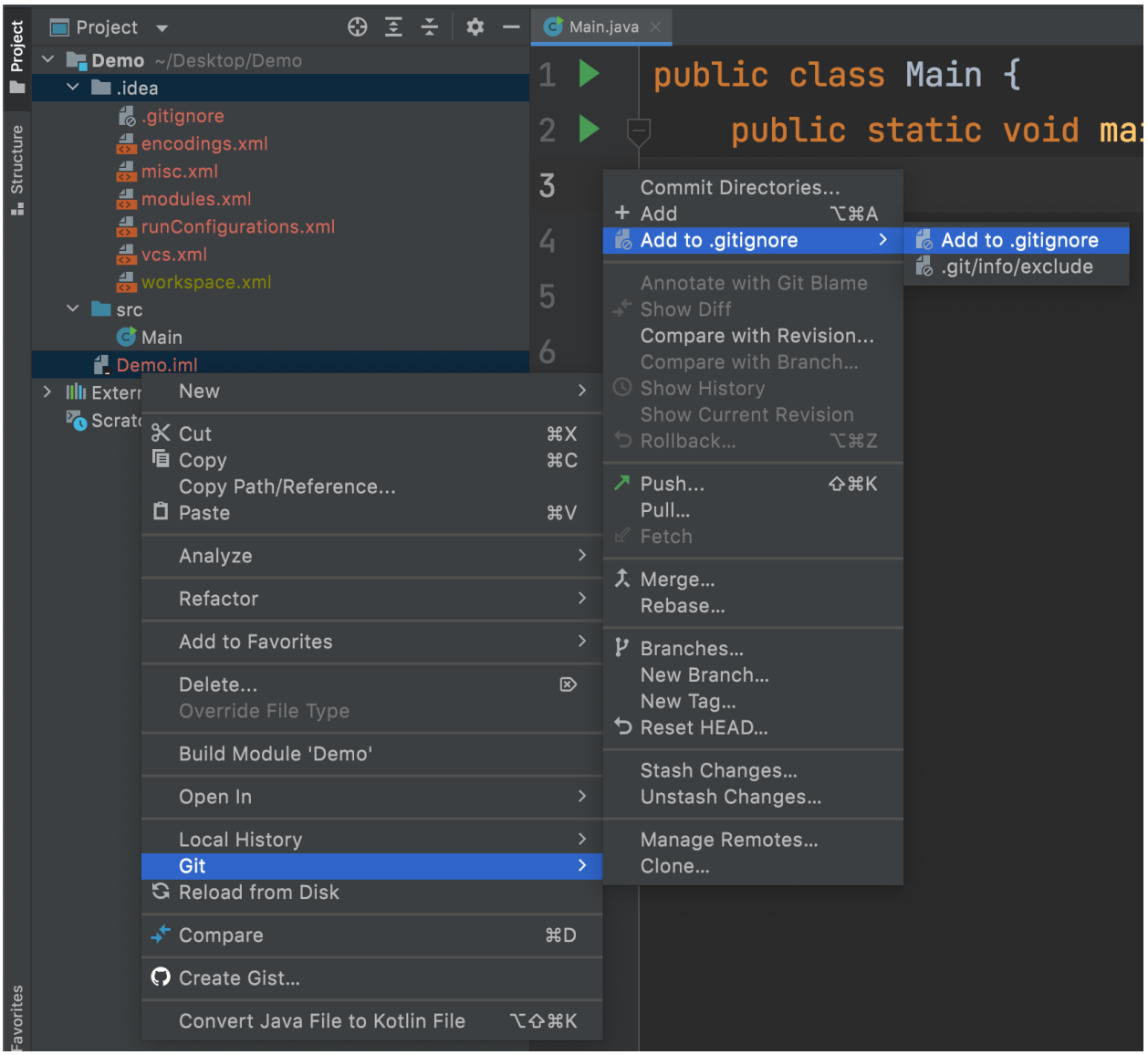


Синяя стрелка, направленная влево вниз — это команда обновления проекта из удалённого репозитория, зелёная галочка — коммит, зелёная стрелка — команда отправки изменений в удалённый репозиторий (git push), а часики — просмотр истории.

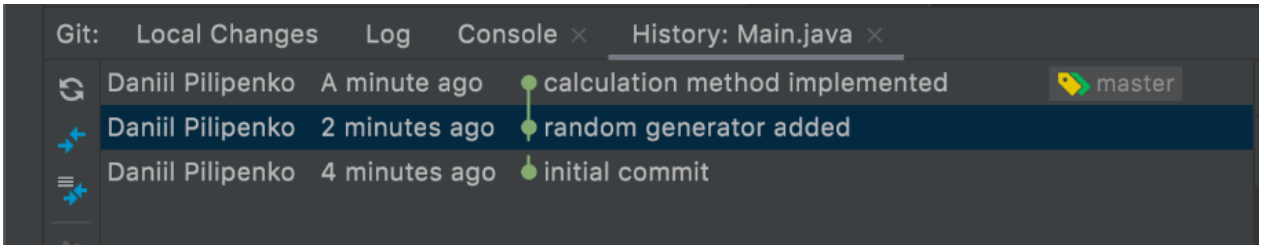
При попытке сделать коммит обычно показывается окно, в котором вы можете выбрать те файлы, которые хотите добавить в коммит, а также вписать сообщение коммита:



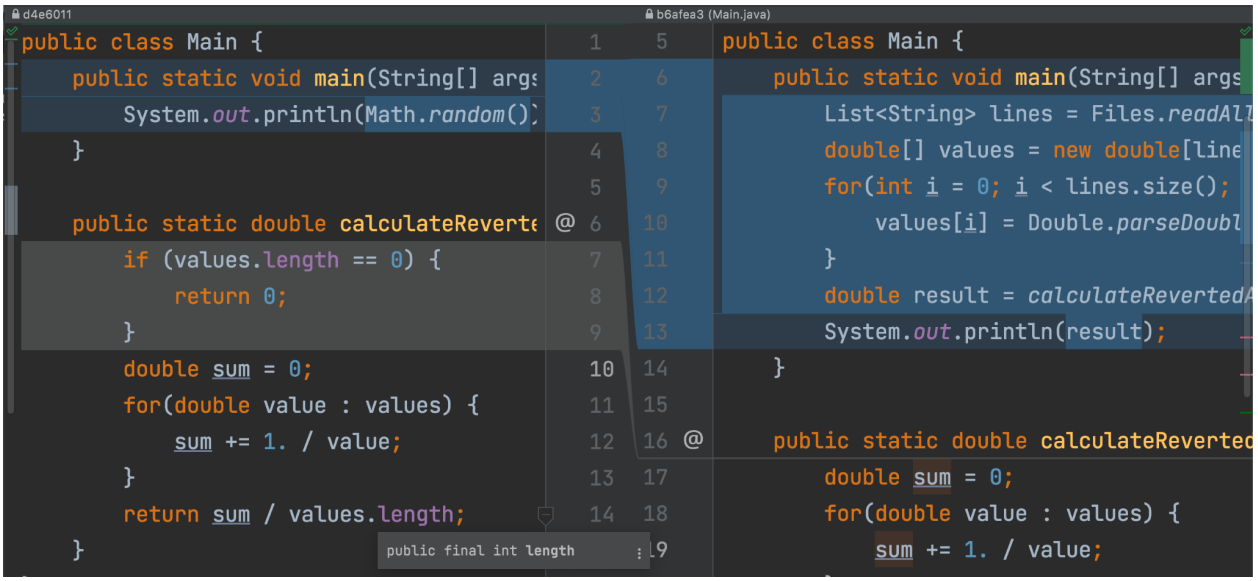
Добавлять файлы и папки в .gitignore также можно через контекстное меню:



Просматривать историю изменений и сравнивать файлы здесь также очень удобно и наглядно. В истории изменений показываются коммиты:



Изменения в файлах при просмотре коммитов или при сравнении двух версий файлов между собой также выводятся очень наглядно с построчной разметкой и выделением цветами добавленных, удалённых и изменённых строк:

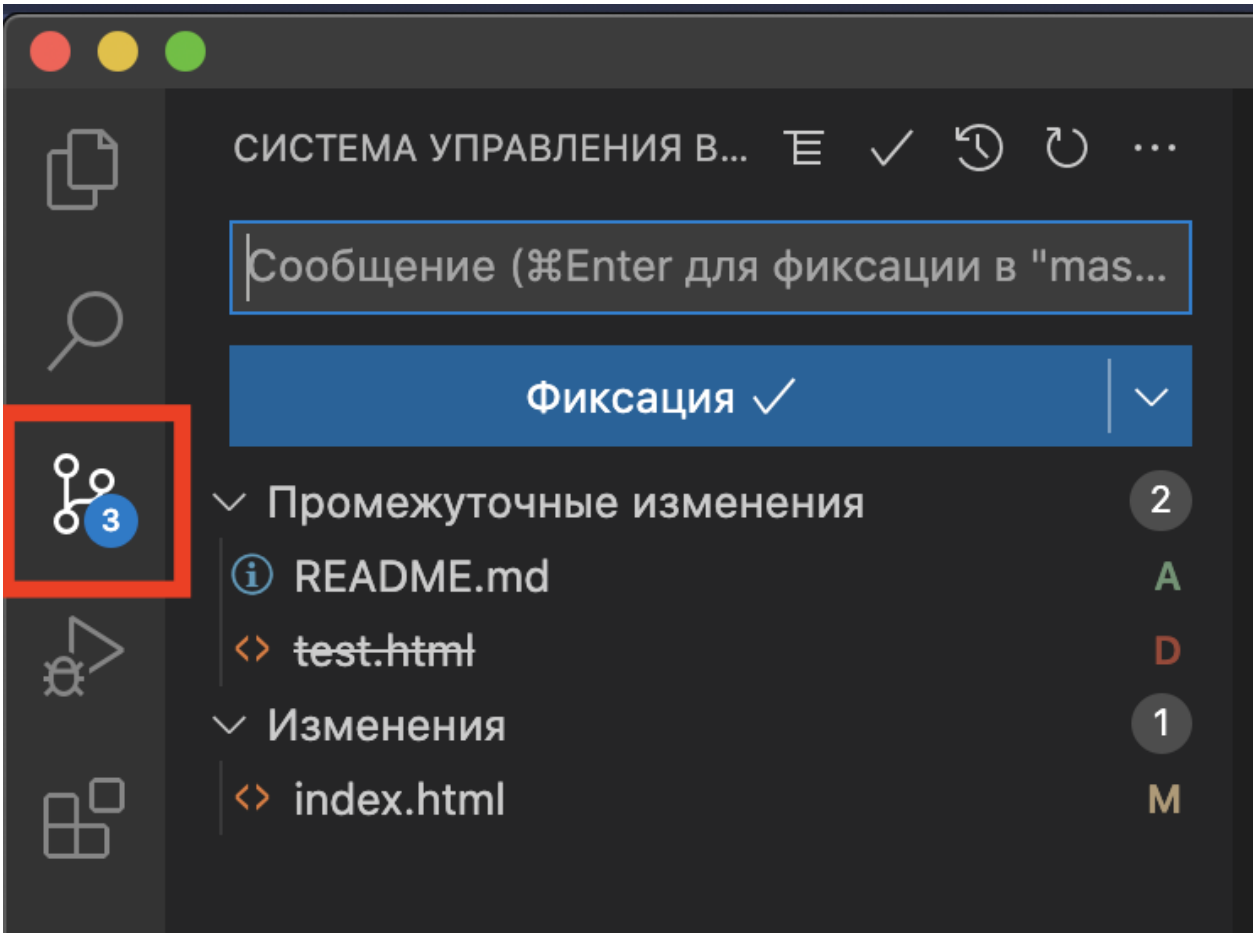


Так выглядит работа с проектом в Git-репозитории в среде разработки от компании JetBrains на примере среды Java-разработки IntelliJ IDEA определённой версии. В других средах компании JetBrains, например, в PHPStorm, WebStrorm или PyCharm процесс выглядит аналогичным образом. При этом в более ранних или более поздних версиях интерфейс этих программ может отличаться.

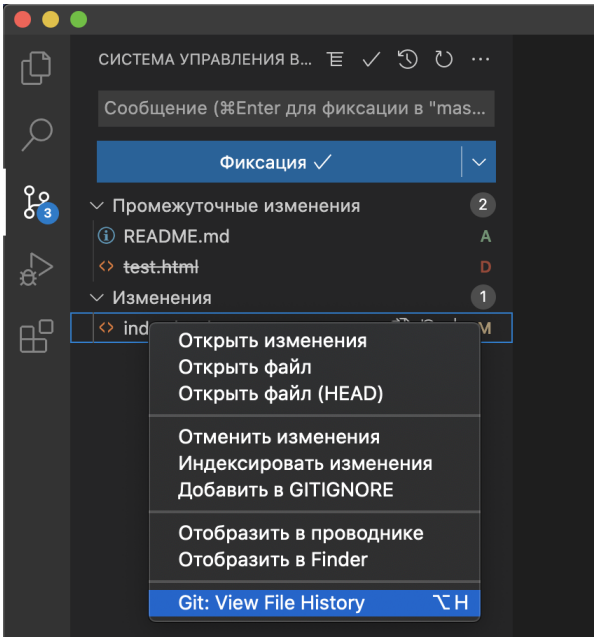
На втором месте по популярности в настоящее время среда разработки Visual Studio Code от корпорации Microsoft, которую сокращённо ещё называют VS Code. Эта среда в последнее время завоевала сердца современных веб-разработчиков своей простотой, лёгкостью и в то же время удобством, и поэтому работа с Git в ней также актуальна.

Слева вверху в этой среде есть иконка отображения текущего проекта в системе Git. Это отображение, по сути, соответствует выводу команды “git status” и показывает добавленные, изменённые и удалённые файлы в текущем проекте. В случае, если в текущем репозитории таких файлов нет, здесь ничего не отображается.

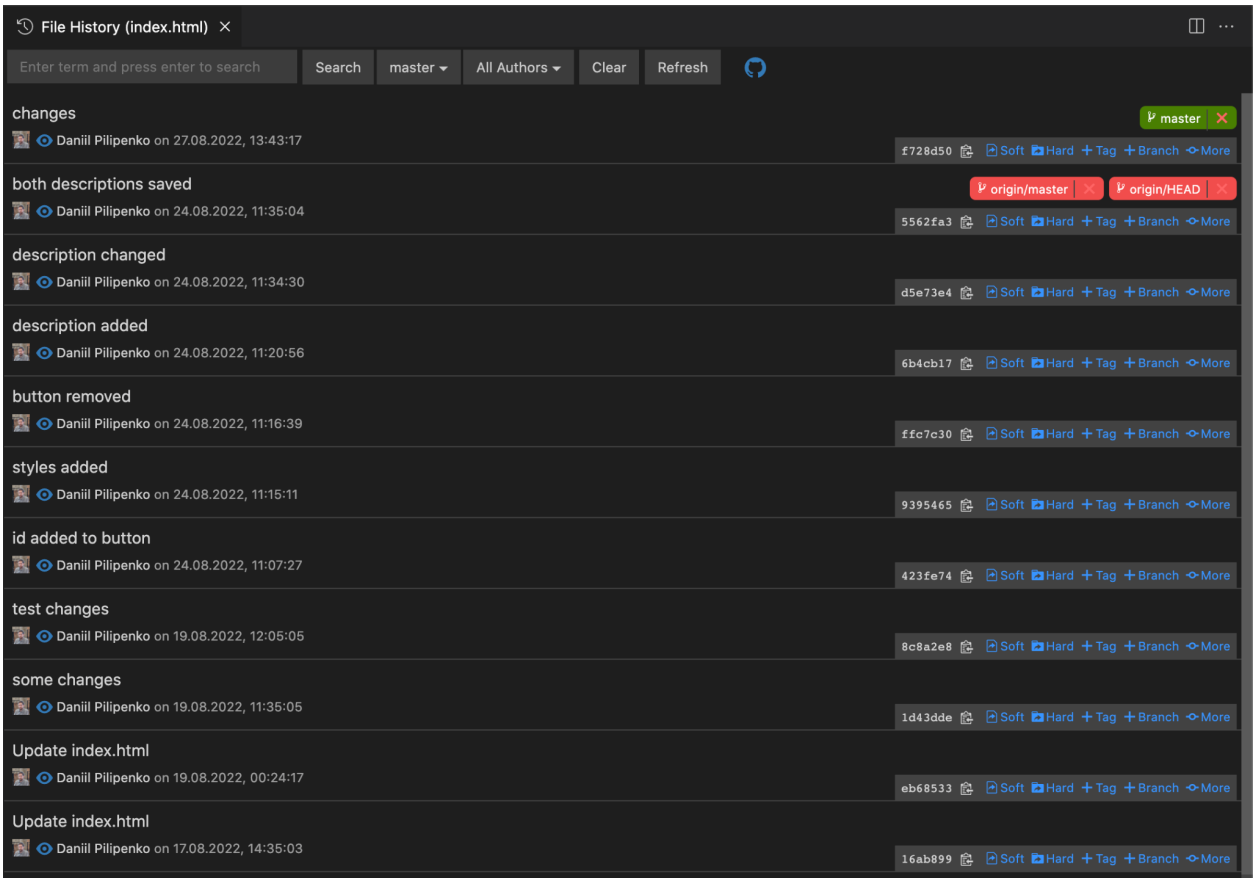
Вот так выглядит перечень изменённых и ещё незакоммиченных файлов в Visual Studio Code:



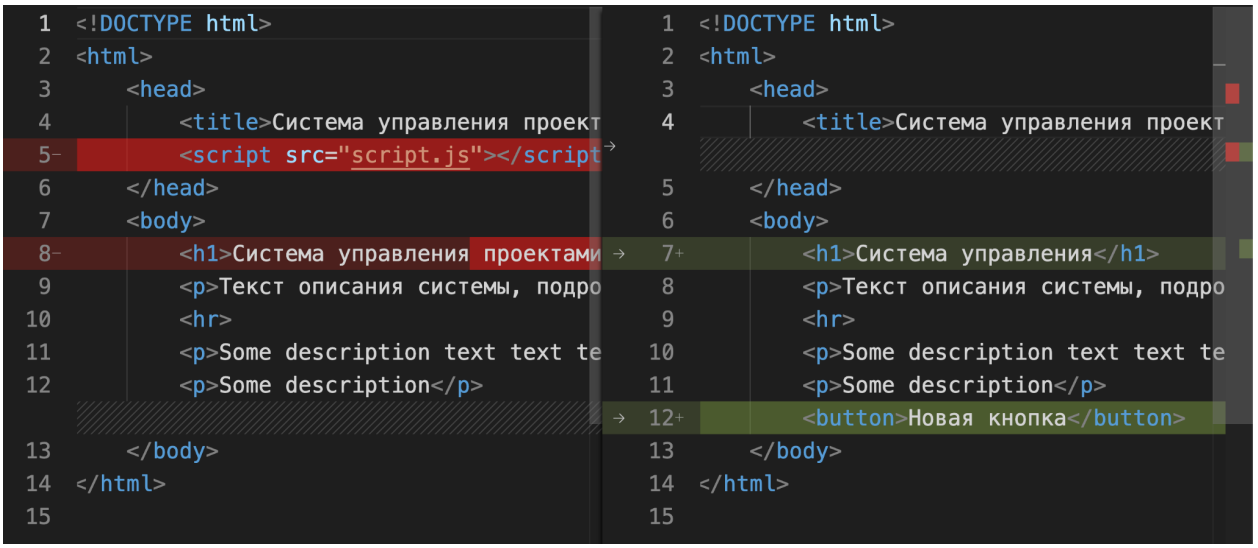
Нажав на кнопку “Фиксация” вы можете сделать коммит. В контекстном меню здесь вы также можете выбирать нужные вам команды Git:



Здесь также, как и в средах разработки от компании JetBrains, очень удобный интерфейс просмотра истории:



И просмотр изменений в отдельном файле между двумя коммитами также имеет удобное и понятное представление, в котором видны все добавленные, удалённые и изменённые строки программного кода:



Теперь вы знаете, как работать с системой контроля версий Git из среды разработки Visual Studio Code. Мы убедились, что Visual Studio Code, так же как и другие среды разработки, предоставляет достаточно широкий инструментарий для визуальной работы с Git.

В то же время напомним вам ещё раз, что самый основной и универсальный инструмент работы с Git — это командная строка, поэтому старайтесь по возможности пользоваться именно ей.

Что можно почитать ещё?

1. Удачная модель ветвления для Git — https://habr.com/ru/post/106912/
2. Почему Trunk Based Development – лучшая модель ветвления — https://habr.com/ru/post/519314/
3. Ежедневная работа в Git — https://habr.com/ru/post/174467/
4. Conventional Commits — https://www.conventionalcommits.org/