Лекция 4-5. Git. Основные сведения

- Общие сведения и история создания
- Основные понятия
- Первоначальная настройка репозитория
- Полезные команды
- Жизненный цикл коммитов в Git (работа с удалённым репозиторием)
- Просмотр истории
- Метки
- Игнорирование файлов
- Ссылка на последний коммит (HEAD)
- Ветвление и конфликты в GIT

Общие сведения

Git – распределенная система управления версиями

- Проект создан Линусом Торвальдсом для управления разработкой ядра Linux,
- первая версия выпущена 7 апреля 2005 года.
- На сегодняшний день поддерживает Джунио Хамано.

Проекты, использующие Git

ядро Linux, Swift, Android, Drupal, Cairo, GNU Core Utilities, Mesa, Wine, Chromium, Compiz Fusion, FlightGear, jQuery, PHP, NASM, MediaWiki, DokuWiki, Qt, ряд дистрибутивов Linux

Автор -

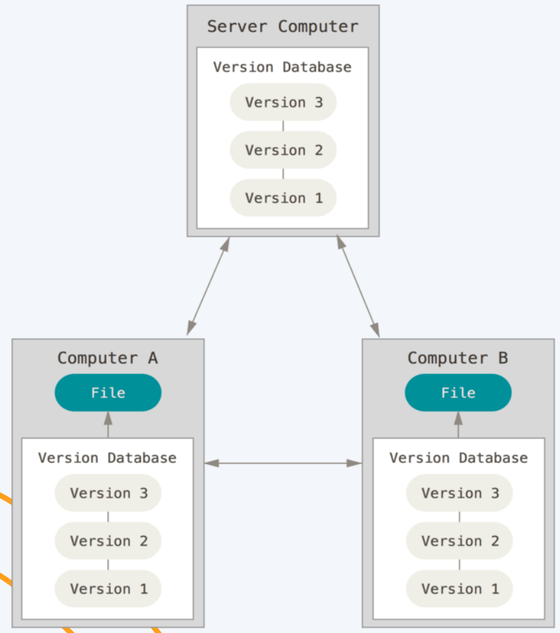
Линус Торвальдс

3 апреля 2005 года разработка началась, и уже 7 апреля код Git управлялся неготовой системой. 16 июня Linux был переведён на Git, а 25 июля Торвальдс отказался от обязанностей ведущего разработчика.

Преследуемые цели

- изменение подхода CVS
- добавление надёжности в систему
- **увеличение скорости**
- простая архитектура
- хорошая поддержка нелинейной разработки (тысячи параллельных веток)
- полная децентрализация
- высокая скорость работы и разумное использование дискового пространства

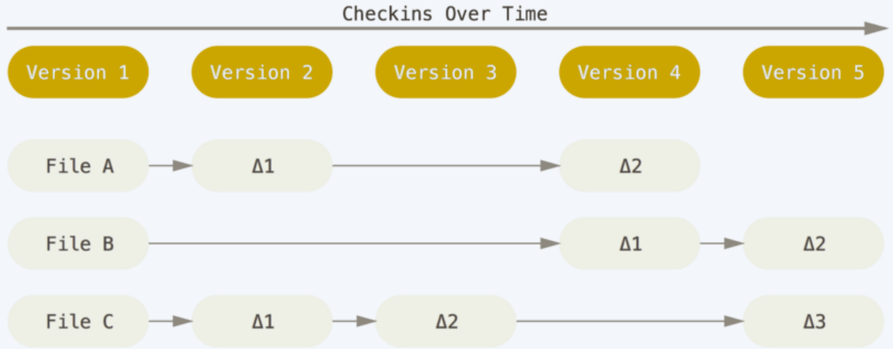
Хранение изменений в Git



В Git используется своя собственная модель разработки, основанная на «снимках» системы.

Обычно в других СКВ (CVS, Subversion, Perforce, Bazaar и т.д.) используется подход на основе диффов (diffs - иформация в виде набора файлов и изменений, сделанных в каждом файле, по времени).

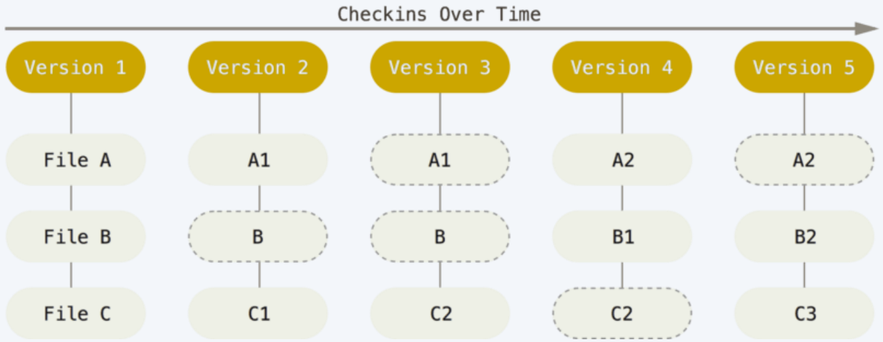
Хранение на основе диффов



Обычно в других СКВ (CVS, Subversion, Perforce, Bazaar и т.д.) используется подход на основе диффов (diffs - иформация в виде набора файлов и изменений, сделанных в каждом файле, по времени).

Подход Git’а к хранению данных больше похож на набор снимков миниатюрной файловой системы. Каждый раз, когда вы делаете коммит, то есть сохраняете состояние своего проекта в Git’е, система запоминает, как выглядит каждый файл в этот момент, и сохраняет ссылку на этот снимок. Для увеличения эффективности, если файлы не были изменены, Git не запоминает эти файлы вновь, а только создаёт ссылку на предыдущую версию идентичного файла, который уже сохранён. Git представляет свои данные как, скажем, **поток снимков**.

Хранение на основе "слепков"



Целостность Git

В Git для всего вычисляется хеш-сумма, и только потом происходит сохранение.

Git только добавляет данные

Жизненный цикл файла в git (локальный)

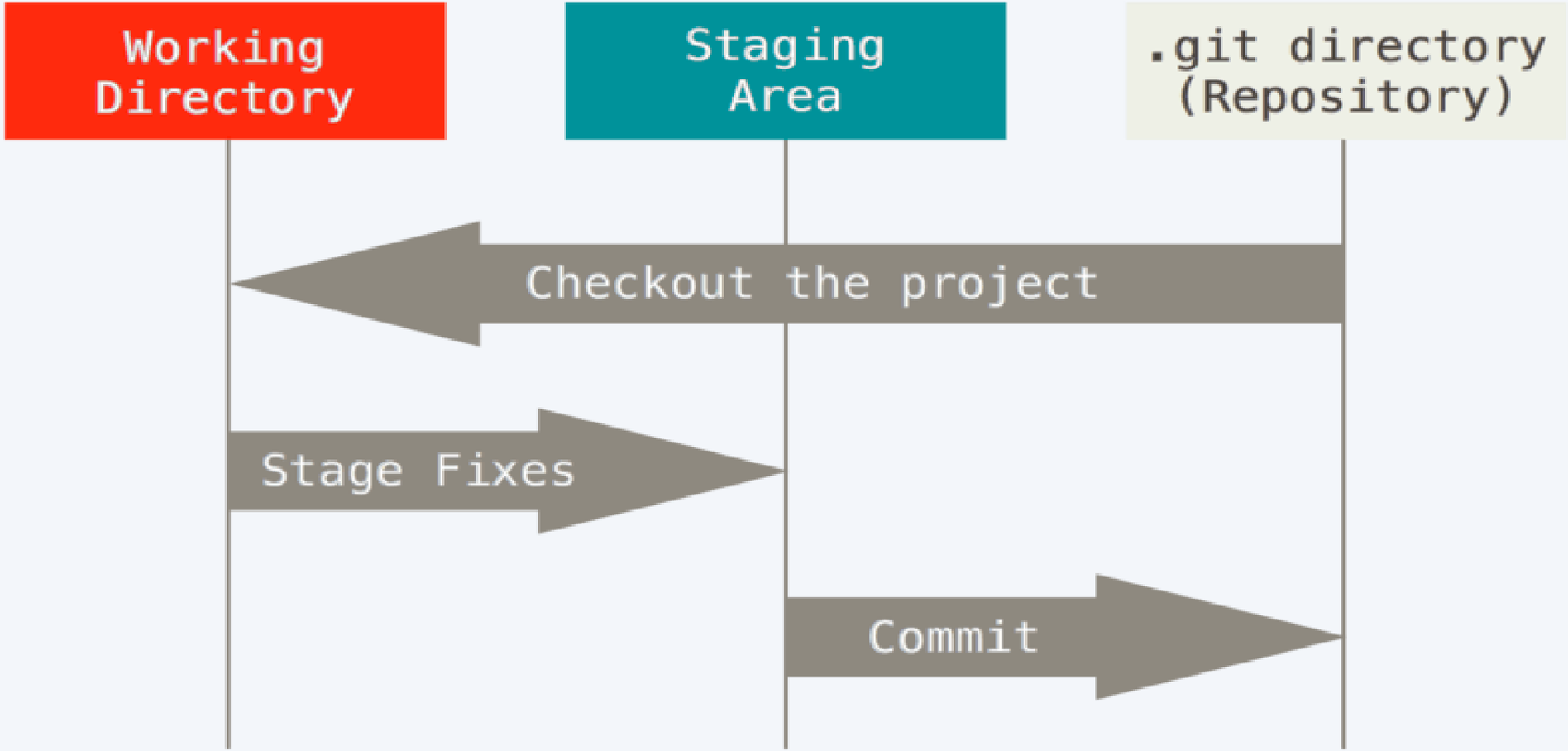
Жизненный цикл файла в git (локальный)

Три основных состояния файла

зафиксированное (committed)	изменённое (modified)	подготовленное (staged)
“Зафиксированный” значит, что файл уже сохранён в локальной базе.	К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы.	Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Жизненный цикл файла в git (локальный)

Три области в git



Жизненный цикл локального репозитория

Директории в Git

Метаданные

различные служебные данные для корректной работы приложения/утилиты и т.д.

Git – директория

Git-директория — это то место, где Git хранит метаданные и базу объектов вашего проекта. Это самая важная часть Git, и это та часть, которая копируется при клонировании репозитория с другого компьютера

Рабочая директория

является снимком версии проекта. Файлы распаковываются из сжатой базы данных в Git-директории и располагаются на диске, для того чтобы их можно было изменять и использовать.

Staging area

Область подготовленных файлов (staging area) — это файл, располагающийся в вашей Git-директории, в нём содержится информация о том, какие изменения попадут в следующий коммит. Эту область ещё называют “индекс”, или stage-область.

- Вы изменяете файлы в вашей рабочей директории.
- Вы добавляете файлы в индекс, добавляя тем самым их снимки в область подготовленных файлов.
- Когда вы делаете коммит, используются файлы из индекса как есть, и этот снимок сохраняется в вашу Git директорию

Первоначальная настройка репозитория

/etc/gitconfig - содержит значения, общие для всех пользователей системы и для всех их репозиториев;

Файл ~/.gitconfig или ~/.config/git/config хранит настройки конкретного пользователя. Этот файл используется при указании параметра –global

Файл config в каталоге Git'а (т.е. .git/config) в том репозитории, который вы используете, хранит локальные настройки.

Имя пользователя и почта

```
$ git config --global user.name "Your Name"
$ git config --global user.email your_name@example.com
```

Проверка настроек

Показать все настройки

```
git config --list
```

```
$ git config --list
user.name=Your Name
user.email=your_name@example.com
```

Проверка значения конкретного ключа

```
git config <key>
```

```
$ git config user.name
Your Name
```

Три способа открыть страницу

```
$ git help <глагол>
$ git <глагол> --help
$ man git-<глагол>
```

Открытие руководства по команде config

```
$ git help config
```

Жизненный цикл коммитов в Git (работа с удалённым репозиторием)

Работа с удалённым репозиторием

Создание нового проекта в текущей директории

```
$git init
```

Клонирование репозитория

```
git clone <url>
```

Изменение файлов

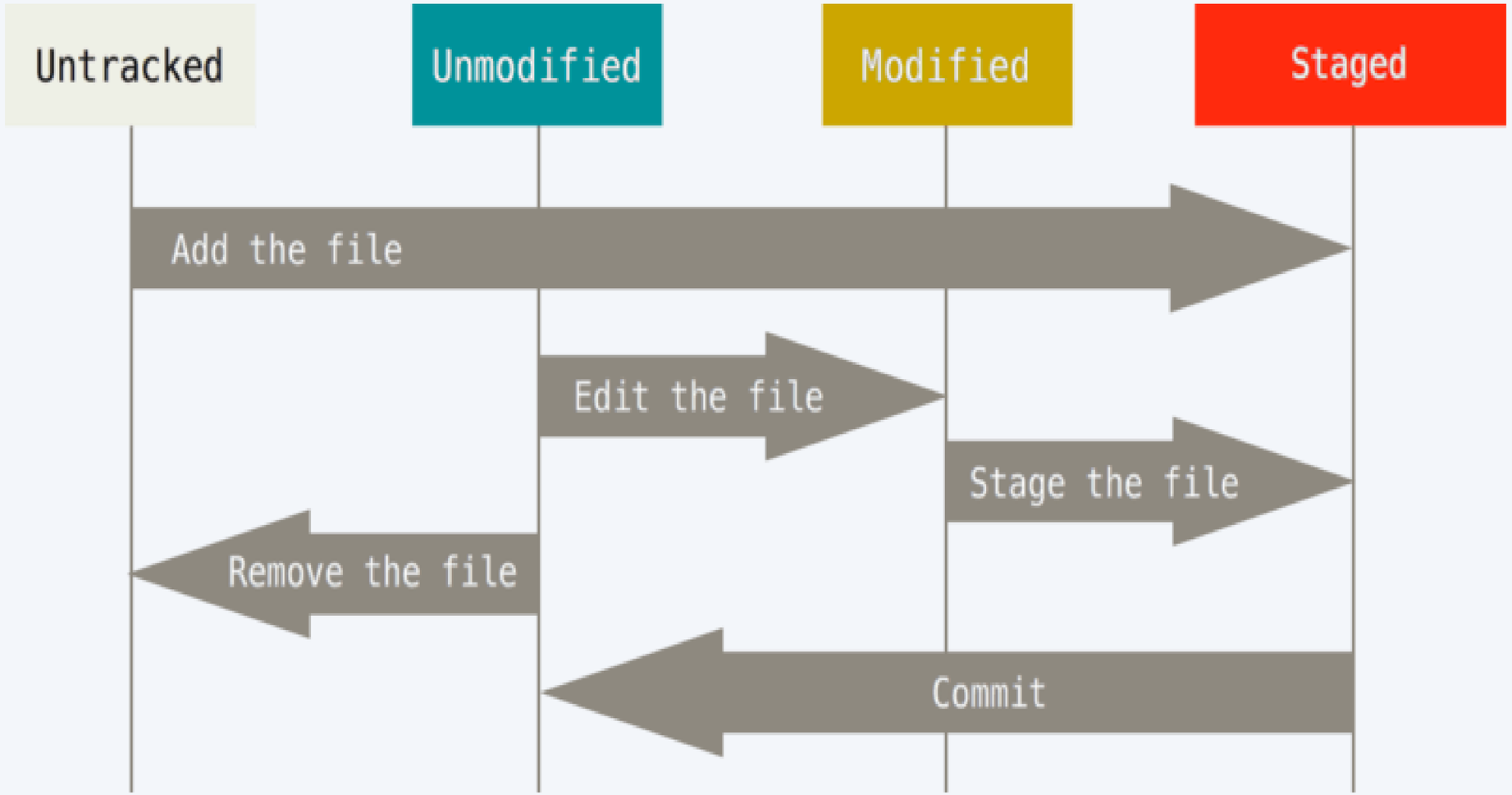
Отслеживаемые (под версионным контролем)

Отслеживаемые файлы — это те файлы, которые были в последнем слепке состояния проекта (snapshot); они могут быть неизменёнными, изменёнными или подготовленными к коммиту (staged).

Неотслеживаемые

Неотслеживаемые файлы — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний слепок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что вы только взяли их из хранилища (checked them out) и ничего пока не редактировали.

Состояния файлов



Запись изменений в репозиторий

Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда **git add**. Чтобы начать отслеживание файла README, вы можете выполнить следующее:

```
$ git add README
```

Если вы снова выполните команду status, то увидите, что файл README теперь отслеживаемый и индексированный (находится в staging area):

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
```

Для того, чтобы записать файлы в локальный репозиторий, применяется команда **git commit**. Её можно использовать с опцией -m, чтобы сразу написать комментарий к данному коммиту. Пример:

```
git commit -m “Добавил файл readme.md”
[master 463dc4f] Добавил файл readme.md
 1 files changed, 15 insertions(+)
 create mode 100644 README
```

Добавление параметра -a в команду git commit заставляет Git автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя вам обойтись без git add:

```
$ git commit -a -m 'added new benchmarks'
```

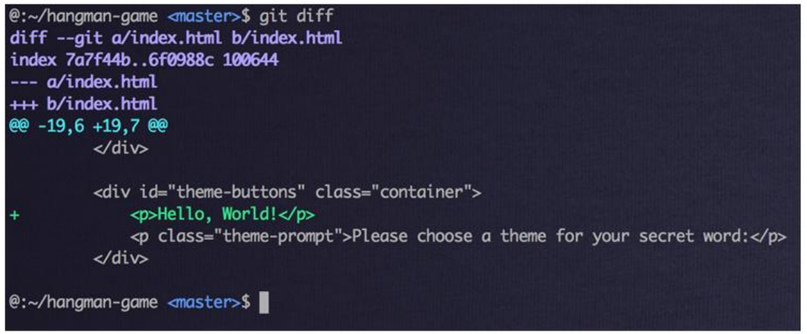
Просмотр изменений

git diff

Чтобы увидеть, что же вы изменили, но пока не проиндексировали:

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
```

Можно использовать git diff для просмотра как индексированных изменений в этом файле, так и тех, что пока не проиндексированы.



Чтобы посмотреть, что вы проиндексировали и что войдёт в следующий коммит, вы можете выполнить git diff --staged. Эта команда сравнивает ваши индексированные изменения с последним коммитом.

Просмотр изменений

Git Diff во внешних инструментах

git difftool вместо git diff, позволяет просмотреть изменения в файле с помощью таких программ как Araxis, emerge, vimdiff и других.

git difftool --tool-help чтобы увидеть какие из них уже установлены в вашей системе.

Отправка изменений на удалённый сервер

git push

```
$ git push
Password for 'https://Ksenia989@bitbucket.org':
Counting objects: 20, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (20/20), done.
Writing objects: 100% (20/20), 534.20 KiB | 17.23 MiB/s, done.
Total 20 (delta 5), reused 0 (delta 0)
To https://bitbucket.org/Ksenia989/labs.git
  c920410..64ac818  master -> master
```

Просмотр истории

git log

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

По умолчанию (без аргументов) git log перечисляет коммиты, сделанные в репозитории в обратном к хронологическому порядке – последние коммиты находятся вверху. Из примера можно увидеть, что данная команда перечисляет коммиты с их SHA-1 контрольными суммами, именем и электронной почтой автора, датой создания и сообщением коммита.
Одним из самых полезных аргументов является -p, который показывает разницу, внесенную в каждый коммит. Так же вы можете использовать аргумент -n, который позволяет установить лимит на вывод количества коммитов (в количестве n).

Опция **--pretty=format** отображает лог в удобном для чтения виде. С параметроми этой опции вы можете ознакомиться в соответствующем разделе справки

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
```

где %h — сокращённый хэш коммита, а %s — сообщение коммита.

Метки:
легковесные и аннотированные.

это указатель на определённый коммит

хранятся в базе данных Git'а как полноценные объекты.

Просмотр меток:

Просмотр имеющихся меток (tag) в Git'е делается просто. Достаточно набрать git tag:

```
$ git tag
v0.1
v1.3
```

Данная команда перечисляет метки в алфавитном порядке.

Создание меток

Git использует два основных типа меток: легковесные и аннотированные

Легковесная метка — это указатель на определённый коммит.

*А вот аннотированные метки хранятся в базе данных **Git'а** как полноценные объекты. Они имеют контрольную сумму, содержат имя поставившего метку, e-mail и дату, имеют комментарий. Обычно рекомендуется создавать аннотированные метки, чтобы иметь всю перечисленную информацию; но если вы хотите сделать временную метку или по какой-то причине не хотите сохранять остальную информацию, то для этого годятся и легковесные метки.*

*По умолчанию, команда **git push** не отправляет метки на удалённые серверы.*

Аннотированные метки

Для создания аннотированный метки укажите **-a** при выполнении команды **tag** (сообщение добавляется с ключом -m):

$ git tag -a v1.4 -m 'my version 1.4'

Вы можете посмотреть данные метки вместе с коммитом, который был помечен, с помощью команды **git show**:

$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

Легковесные меткие

Легковесная метка — это ещё один способ отметки коммитов. В сущности, это контрольная сумма коммита, сохранённая в файл — больше никакой информации не хранится. Для создания легковесной метки не передавайте опций -a, -s и -m:

$ git tag v1.4-lw

На этот раз при выполнении git show на этой метке вы не увидите дополнительной информации. Команда просто покажет помеченный коммит:

$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

changed the version number

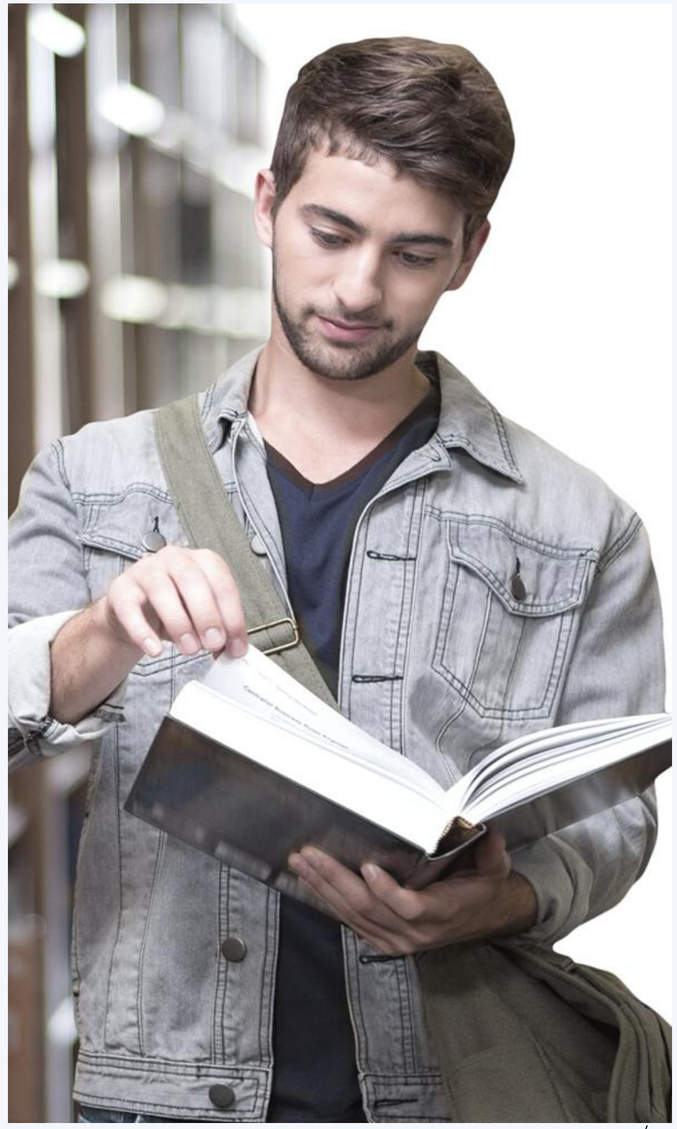
Выставление меток позже

Также возможно помечать уже пройденные коммиты. Предположим, что история коммитов выглядит следующим образом:

$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile

Для отметки коммита укажите его контрольную сумму (или её часть) в конце команды:

$ git tag -a v1.2 9fceb02



Игнорирование файлов

Зачастую, у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках отслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и тому подобное). В таком случае, вы можете создать файл **.gitignore**. с перечислением шаблонов соответствующих таким файлам.

Вот пример файла .**gitignore** (**cat** выводит содержимое файла):

```
$ cat .gitignore
*.[oa]
*~
```

Первая строка предписывает **Git** игнорировать любые файлы заканчивающиеся на ``.o`` или ``.a`` - объектные и архивные файлы, которые могут появиться во время сборки кода. Вторая строка предписывает игнорировать все файлы заканчивающиеся на тильду (~), которая используется для обозначения временных файлов.

Хорошая практика заключается в настройке файла .**gitignore** до того, как начать серьёзно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите.

.gitignore

К шаблонам в файле .gitignore применяются следующие правила:

- Пустые строки, а также строки, начинающиеся с #, игнорируются (это коммента-рии);
- Можно использовать стандартные glob шаблоны (см. ниже);
- Можно начать шаблон символом слэша (/) чтобы избежать рекурсии;
- Можно заканчивать шаблон символом слэша (/) для указания каталога;
- Можно инвертировать шаблон, использовав восклицательный знак (!) в качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения, используемые командными интерпретаторами:

- * соответствует 0 или более символам;
- последовательность [abc] — любому символу из указанных в скобках;
- знак вопроса (?) соответствует одному символу;
- квадратные скобки, в которые заключены символы, разделённые дефисом ([0-9]), соответствуют любому символу из интервала.

Вы также можете использовать две звёздочки, чтобы указать на вложенные директории: a//z соответствует a/z, a/b/z, a/b/c/z, и так далее.**

```
# no .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# ignore all files in the build/ directory
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
```

GitHub поддерживает довольно полный список примеров .gitignore файлов для множества проектов и языков (Java, c++, c, c# и т.д., различные IDE https://github.com/github/gitignore это может стать отправной точкой для .gitignore в вашем проекте.

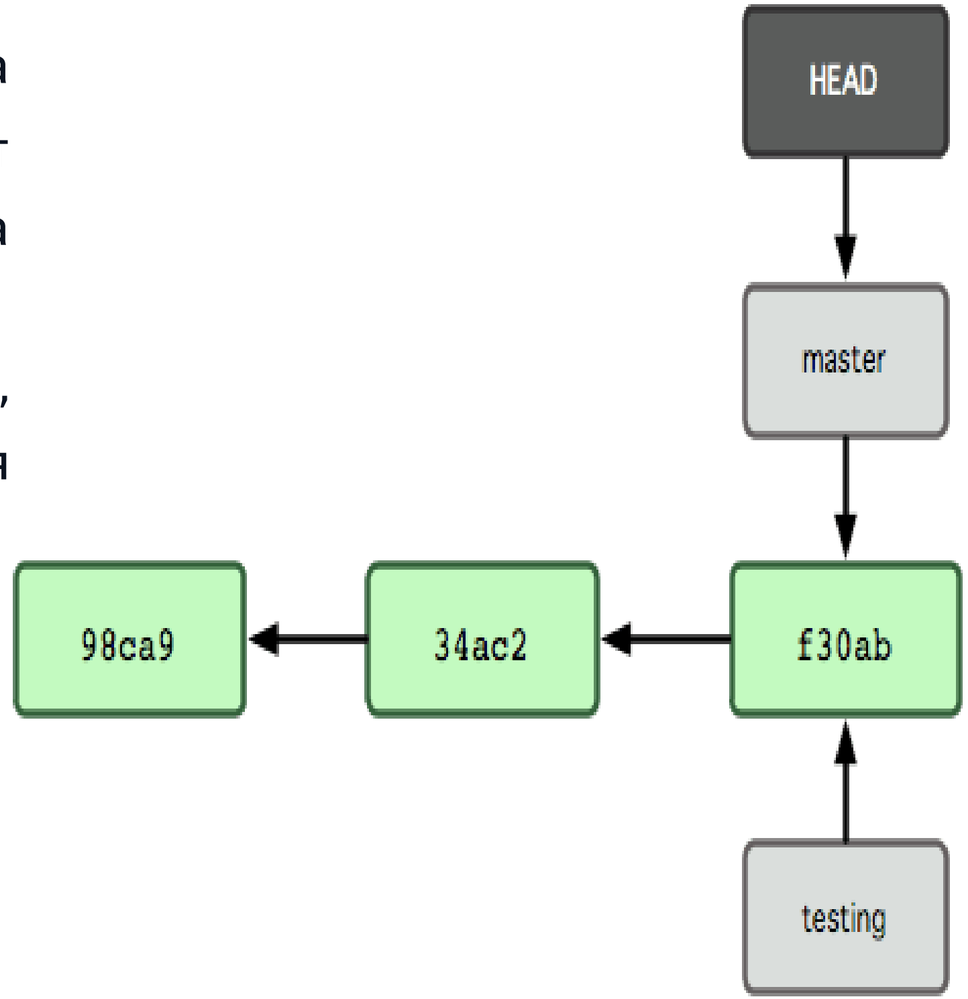
Ссылка на последний коммит (HEAD)

Ссылка на последний коммит (HEAD)

Файл HEAD – это символическая ссылка на текущую ветку. Символическая ссылка отличается от обычной тем, что она содержит не сам хеш **SHA-1**, а указатель на другую ссылку – на последний коммит.

При выполнении **git commit Git** создаёт коммит, указывая его родителем объект, **SHA-1** которого содержится в файле, на который ссылается **HEAD**.

- Знать о HEAD полезно для таких операций, как, например, возвращение к предыдущему коммиту или откат к нему.
- Например, переместиться на один коммит назад (с удалением своих изменений) можно командой:
- *git reset –hard HEAD^*



Интересные источники для введения в Git:

https://learngitbranching.js.org/ - интерактивное обучение Git. Есть русский язык: те-мы варьируются от базовых к более сложным и интересным операциям (занимает не много времени);

http://rogerdudler.github.io/git-guide/ - краткая “шпаргалка” по основным командам git (с переводом на русский).

Использованные источники:

Pro Git book

https://ru.wikipedia.org/wiki/Git

https://proselyte.net/tutorials/git/git_life_cycle/

Ветвление в Git

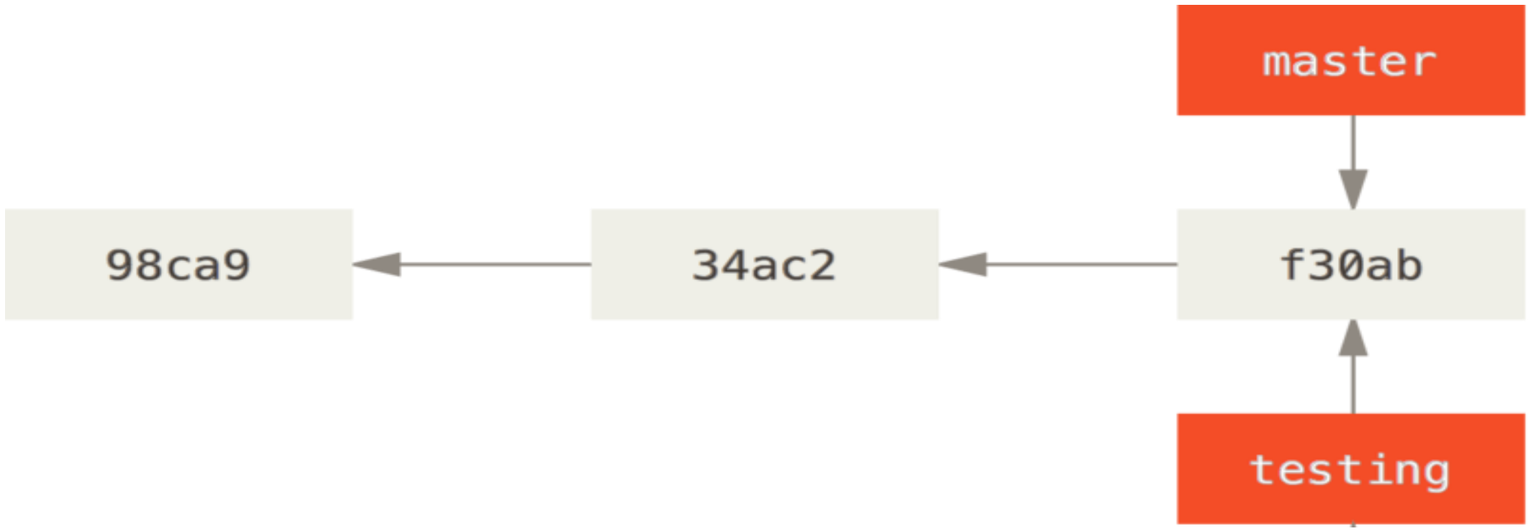
Ветвление в Git

Определение

Ветка (branch) в Git — это легко перемещаемый указатель на один из коммитов. Имя основной ветки по умолчанию в Git — master.

Команда создания ветки

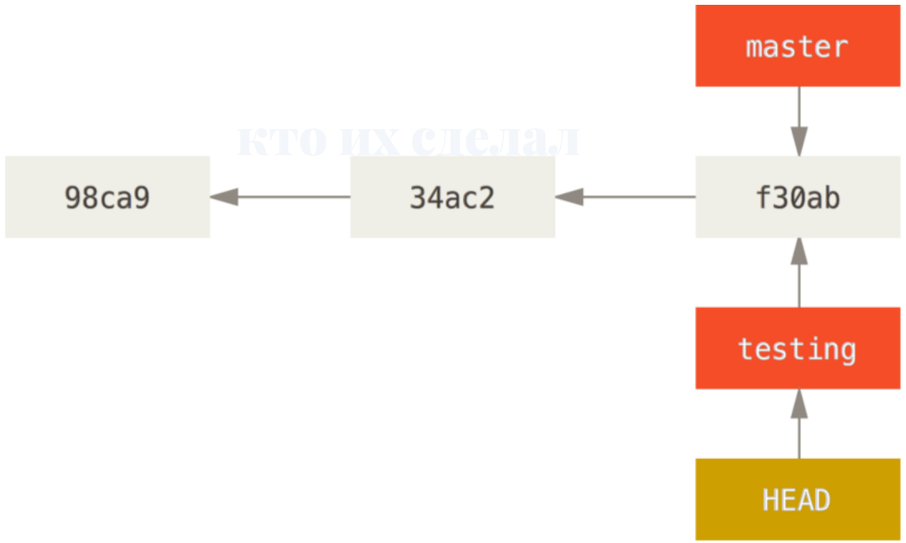
```
$ git branch testing
```



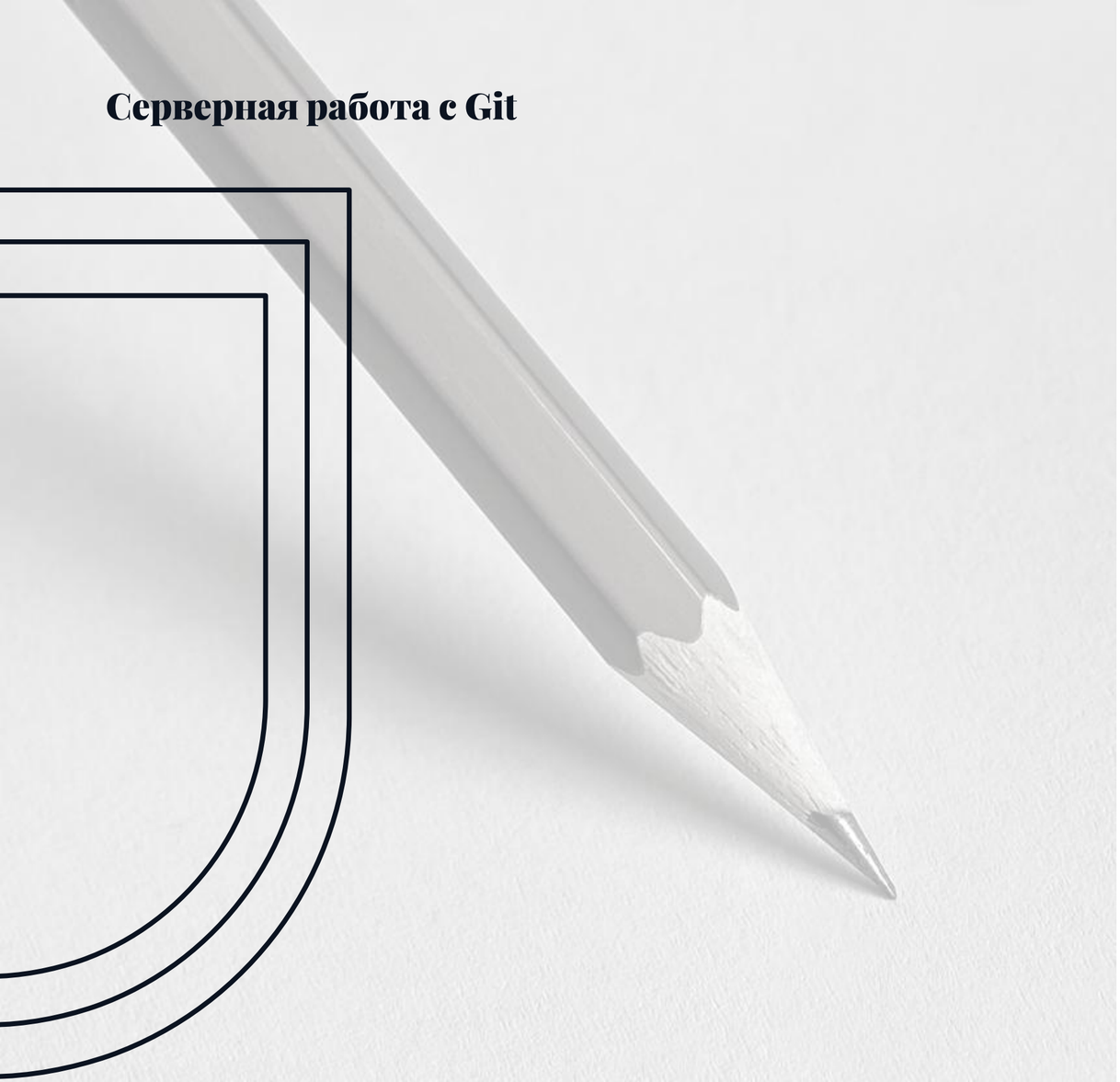
Команда смены ветки

Чтобы переключиться на существующую ветку, выполните команду git checkout. Давайте переключимся на ветку “testing”:

```
$ git checkout testing
```



Серверная работа с Git



Протоколы доступа к серверу git

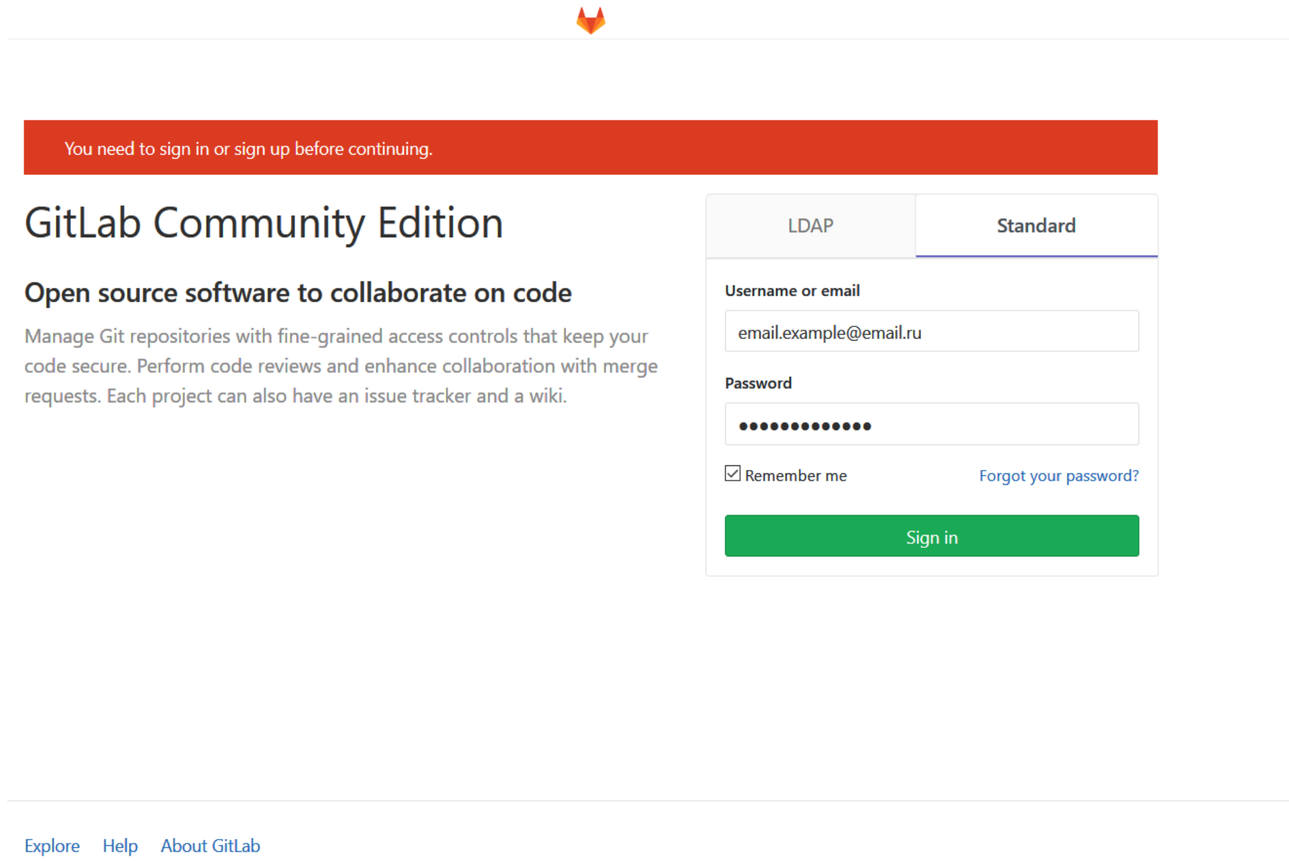
HTTP

SSH

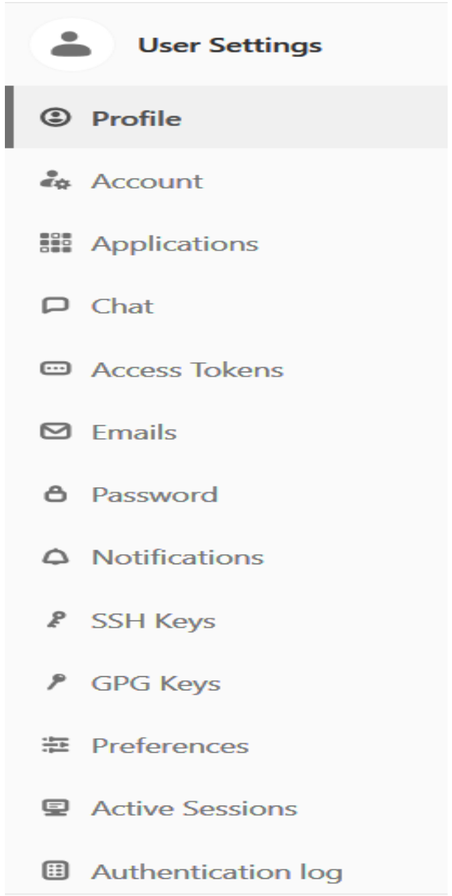
Git

GitLab

GitLab является проектом с открытым исходным кодом (open source project), для ко-торого существуют готовые образы виртуальной машины для простой настройки.

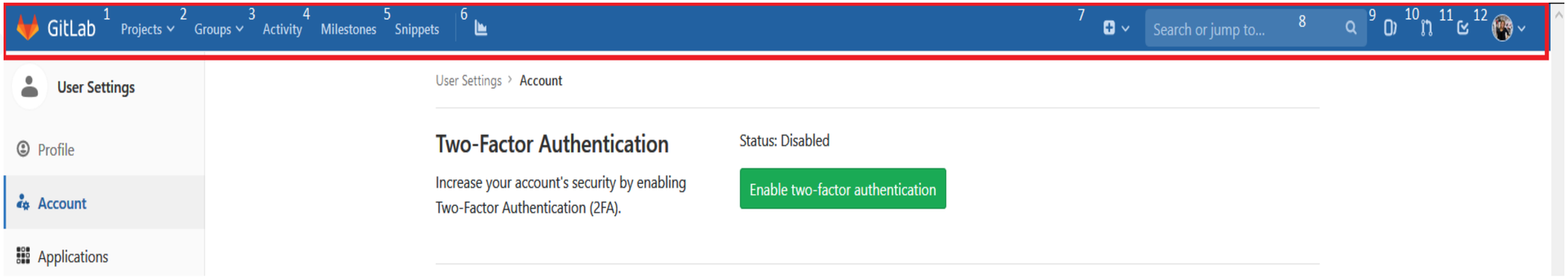


Основные функции сервиса



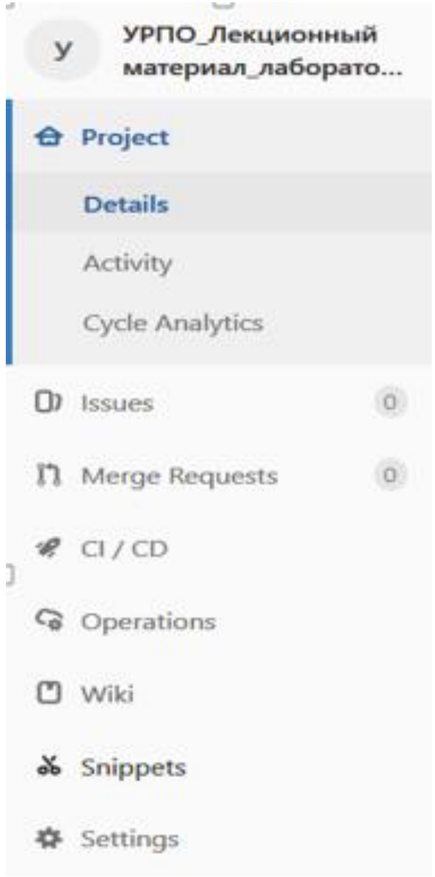
Панель настроек

Основные функции сервиса



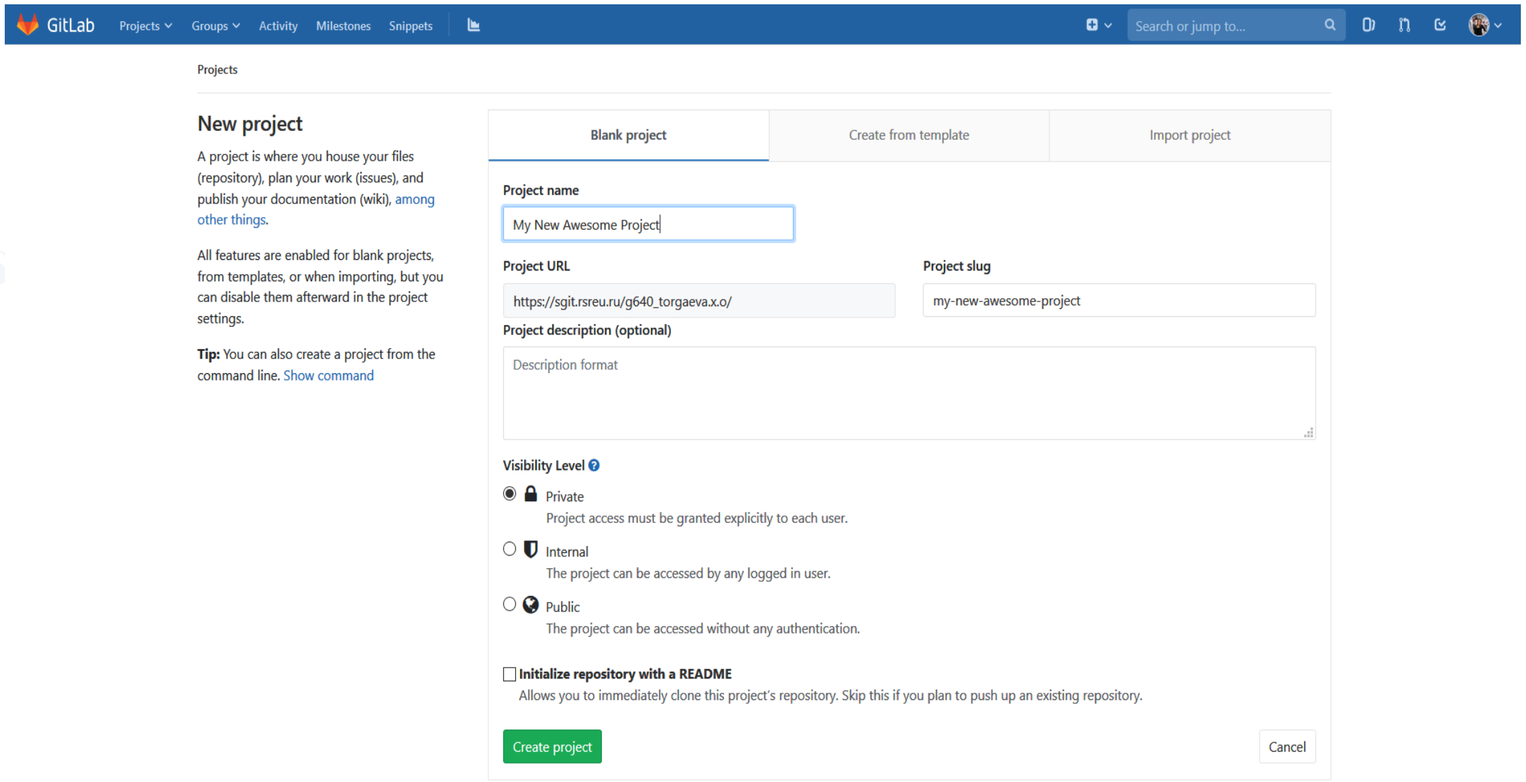
Верхняя панель GitLab

Основные функции сервиса



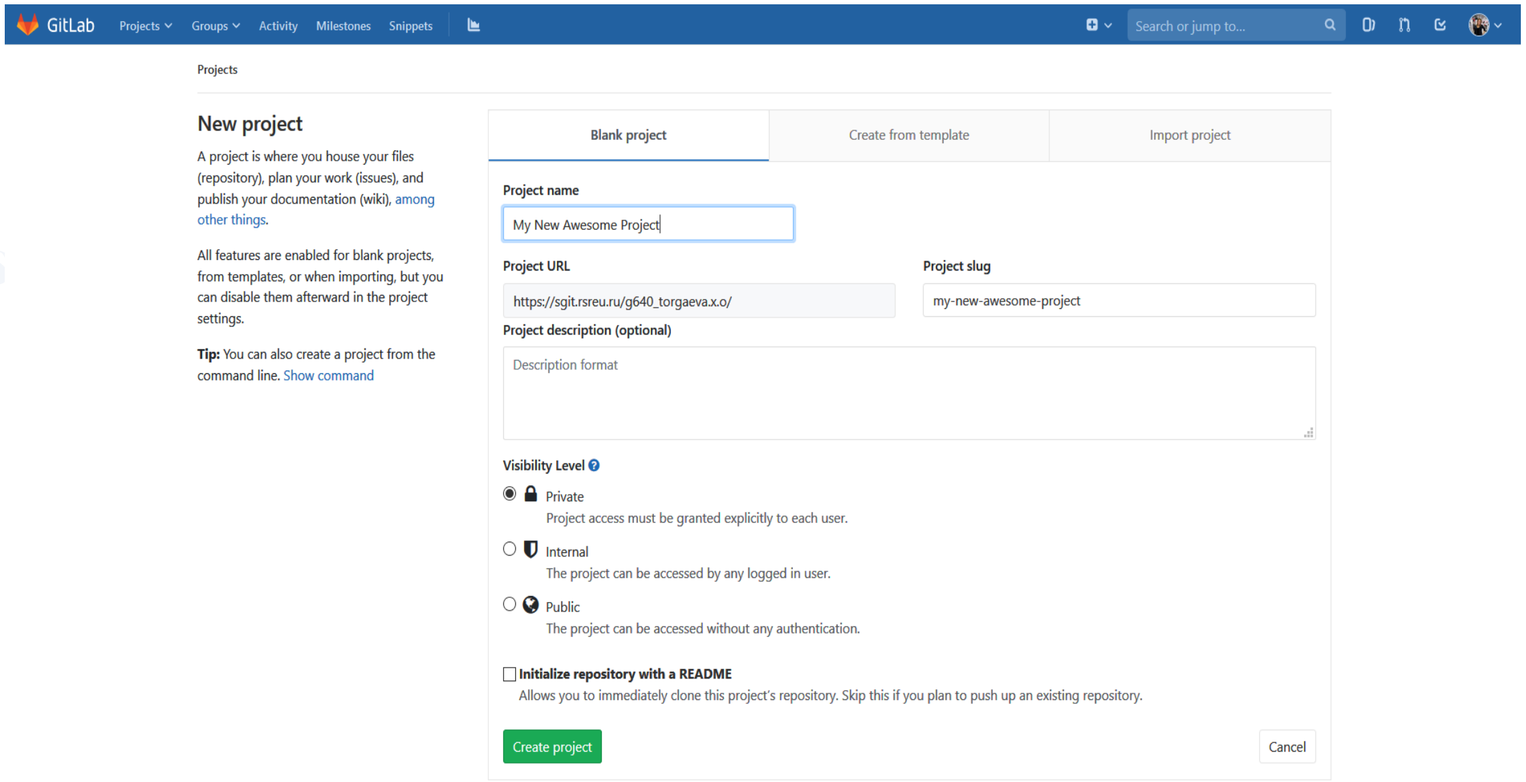
Настройки проекта GitLab

Основные функции сервиса



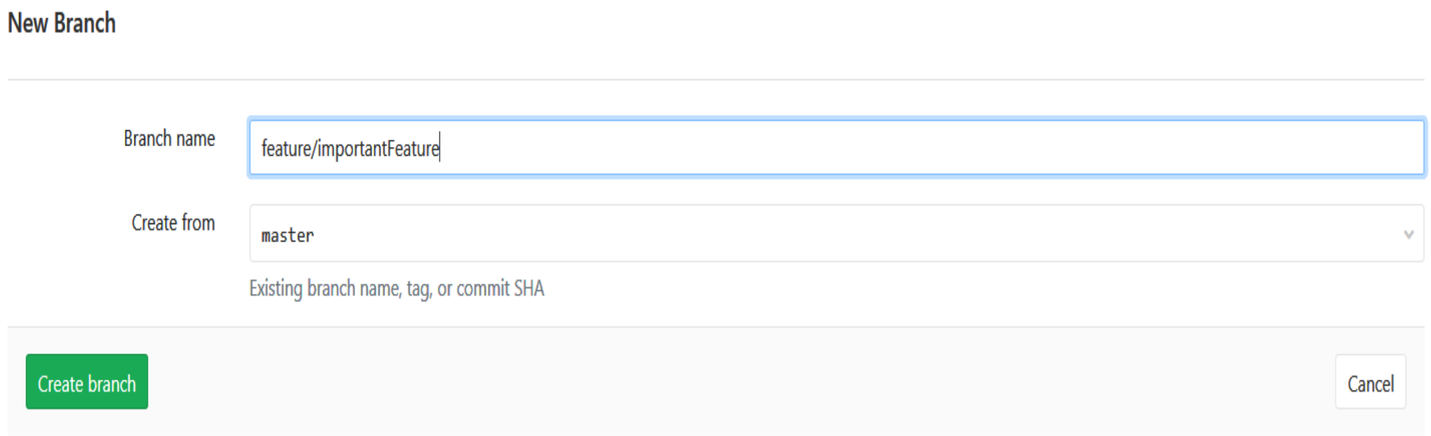
Создание проекта

Основные функции сервиса



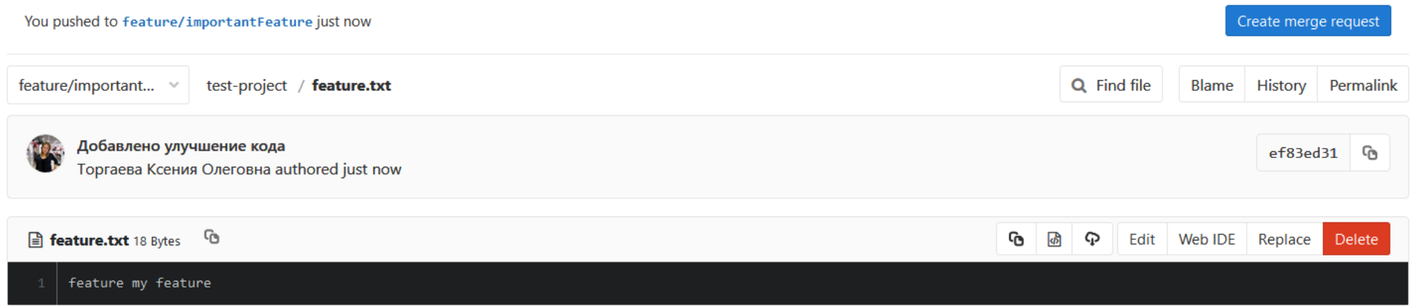
Создание проекта

Основные функции сервиса



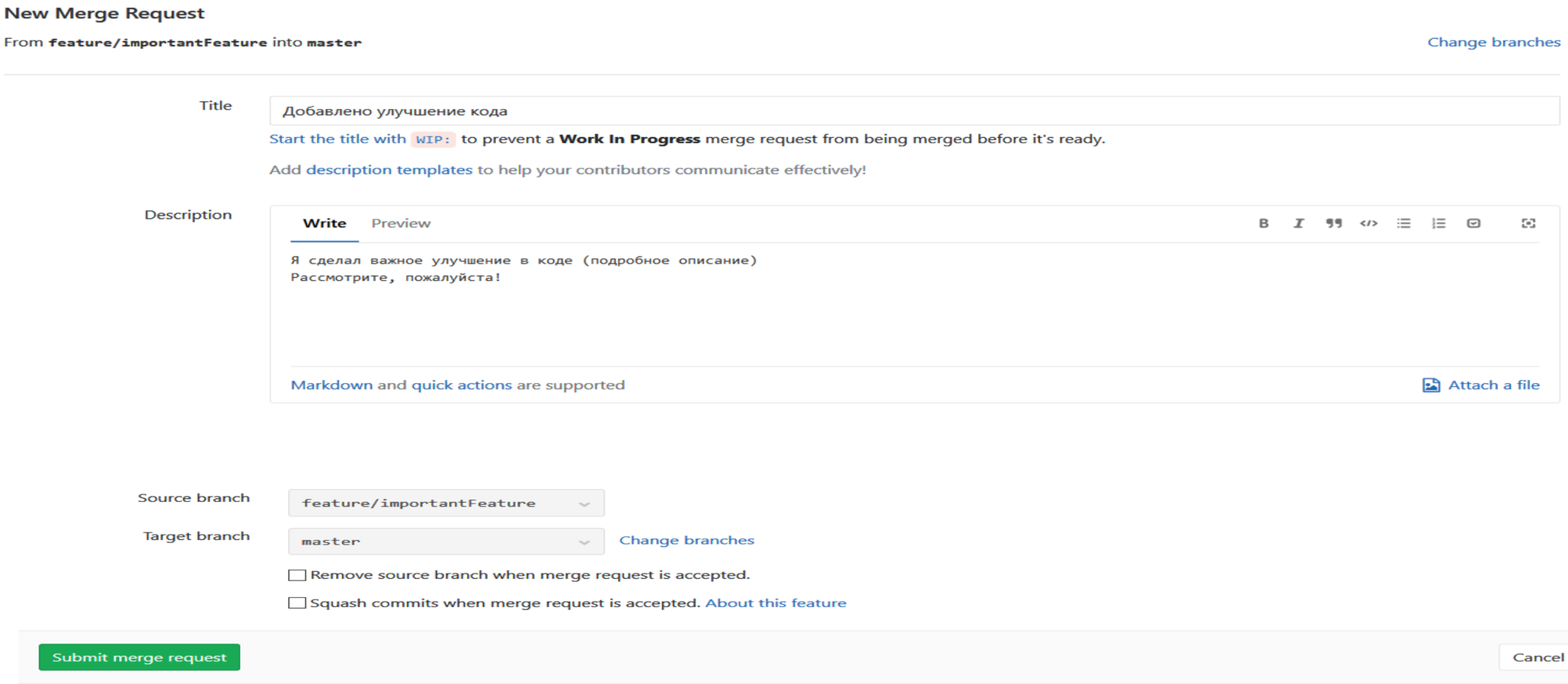
Создание ветки

Основные функции сервиса



Новая ветка "feature/importantFeature" с файлом feature.txt

Основные функции сервиса



Окно создания запроса на изменение

Основные функции сервиса

Fork проекта

Можно воспринимать форк проекта как копию чужого репозитоиря у себя. Чтобы сделать форк в GitLab, нужно просто перейти на страницу с проектом, и нажать «Fork».

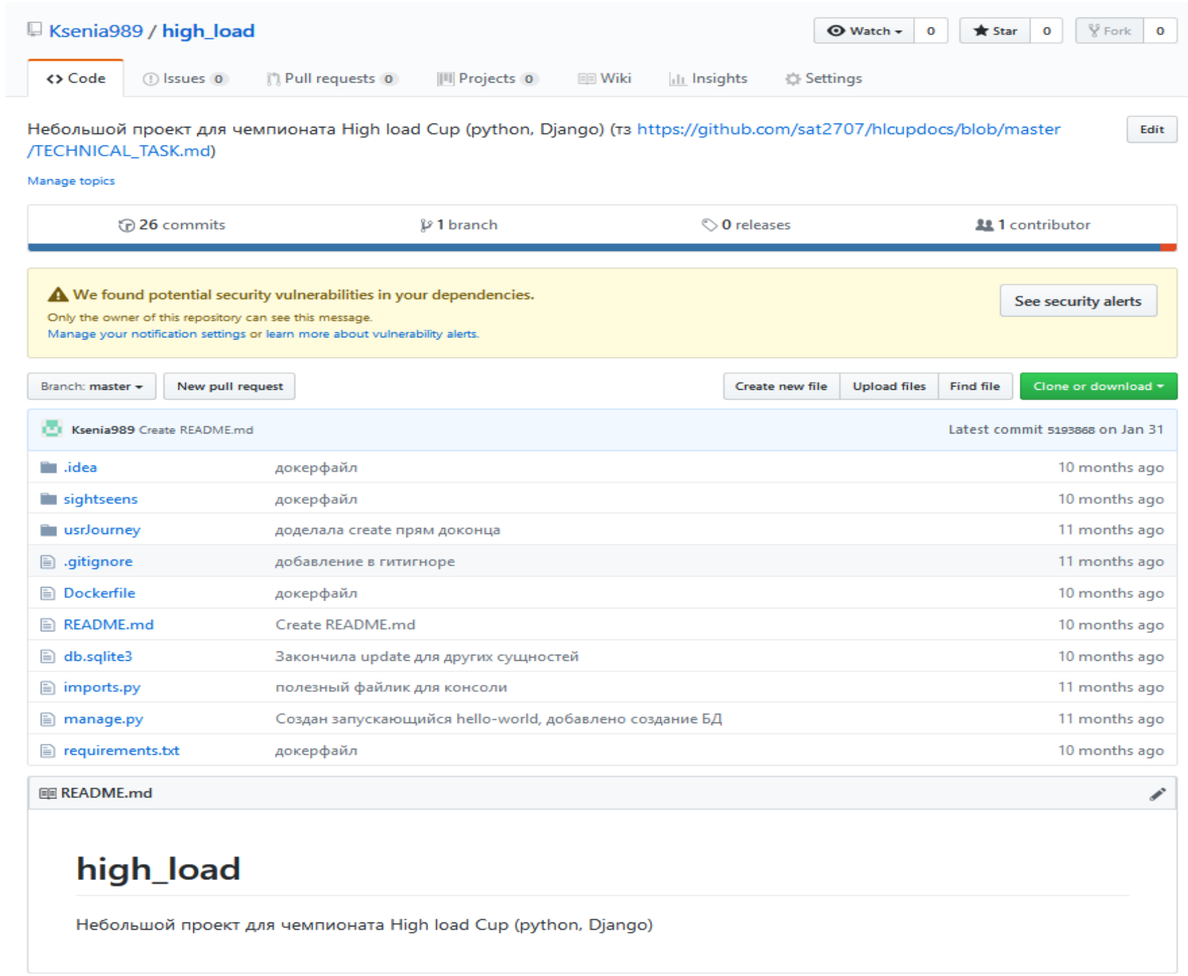
Добавление в группу для работы с проектом

В лабораторной работе вам понадобится добавить человека в группу для работы с одним проектом (make a collaboration). Для этого необходимо зайти в “Settings”->”Members”, на этой странице осуществляется добавление другого человека в участники проекта. Выберите имя участника по его никнейму и права (для лабораторной необходи-мо установить права на запись (права “developer”)).

Для чего используют GitHub

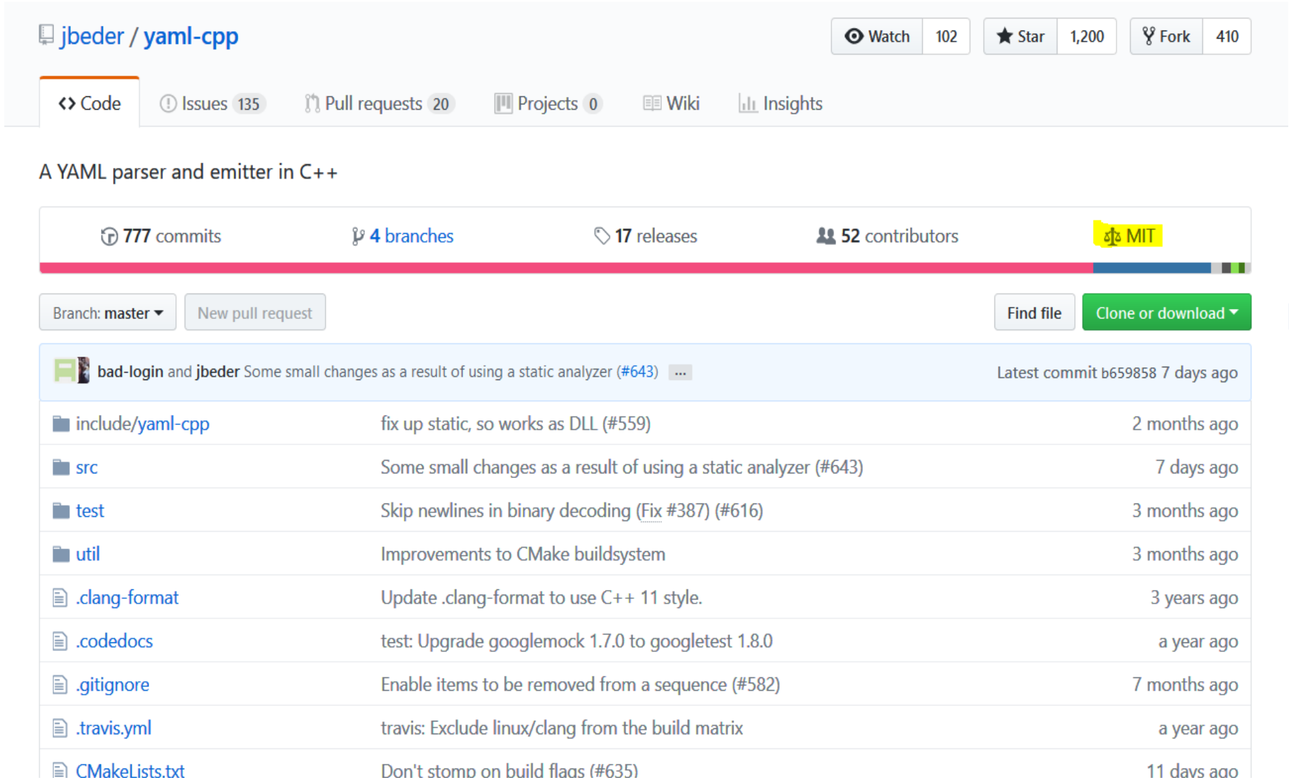
- Создать свой проект с открытым исходным кодом / использовать как хранилище кода, который не хотелось бы потерять
- Воспользоваться сторонней библиотекой, являющейся нестандартной
- Внести вклад в уже существующий проект на GitHub

Основные функции сервиса



Вид проекта на GitHub

Основные функции сервиса



yamp-cpp проект разработчика jbeder

Спасибо!

Жду ваших вопросов...