

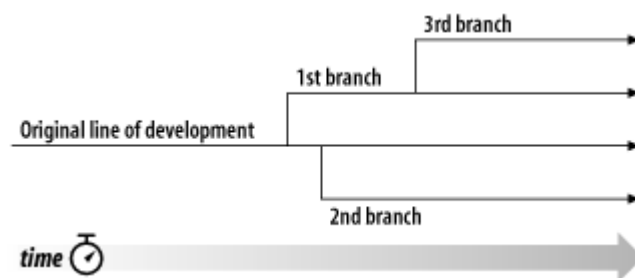
## Лекция 4. Git и Subversion. Работа на сервере.

Для работы в команде необходимо организовать совместную работу. В данной лекции мы рассмотрим ветвление, слияние изменений, разрешение конфликтов для Subversion и Git, а так же работу с GitLab и OpenSource разработку с помощью GitHub.

В приложениях будут рассмотрены такие важные темы, как модель ветвления git flow, и нахождение ошибок в коде методом половинного деления.

### 4.1. Ветвление в Subversion

*Ветка* - это направления разработки, которое существует независимо от другого направления, однако имеющие с ним общую историю, если заглянуть немного в прошлое. Ветка всегда берет начало как копия чего-либо и движется от этого момента создавая свою собственную историю.

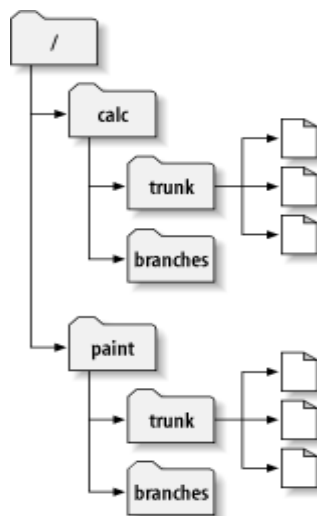


Пример веток разработки в Subversion

У Subversion есть команды, которые помогают сопровождать параллельные ветки файлов и директорий. Эти команды позволяют создавать ветки, копируя информацию и запоминая, что копии относятся друг к другу. Они позволят вам «смешивать и согласовывать» различные направления разработки в своей каждодневной работе.

### Использование веток

Рассмотрим пример: вы и ваш соразработчик Салли делите хранилище, содержащее два проекта, paint и calc. Каждая директория проекта содержит поддиректории с названиями trunk и branches.



Начальная структура хранилища

Как и раньше, будем считать что Салли и вы, оба, имеете рабочие копии проекта «calc». А конкретно, каждый из вас имеет рабочую копию /calc/trunk.

Скажем, перед вами была поставлена задача коренной реорганизации проекта. Это займет много времени и затронет все файлы проекта. Проблема заключается в том, что вы не хотите мешать Салли, которая прямо сейчас занимается исправлением небольших ошибок. Если вы начнете пошагово фиксировать свои изменения, вы конечно же смешаете Салли все карты.

Недостатки работы в одной ветви:

- Во-первых это не очень надежно. Большинство людей предпочитают часто сохранять свою работу в хранилище, на случай если вдруг что-то плохое случится с рабочей копией;

- Во-вторых, это не достаточно гибко. Если вы работаете на разных компьютерах, вам придется вручную копировать изменения взад и вперед, либо делать всю работу на одном компьютере.

Лучшим решением является создание вашей собственной ветки, или направления разработки, в хранилище. Это позволит вам часто сохранять наполовину поломанную работу не пересекаясь с изменениями других людей, и, кроме того, вы можете выборочно разделять информацию с другими соразработчиками.

### **Создание ветки**

Создать ветку очень просто — при помощи команды `svn copy` делаете в хранилище копию проекта.

Итак, вам нужно сделать копию директории /calc/trunk. Вам необходимо создать новую директорию /calc/branches/my-calc-branch которая будет являться копией /calc/trunk.

Есть два способа создания копии:

- С клонированием рабочей копии и созданием ветви;
- Просто с созданием новой ветви, которая вам необходима.

Сначала мы покажем первый неудачный способ, просто для того, что бы прояснить основные моменты. Для начала, создается рабочая копия корневой директории проекта /calc:

```
$ svn checkout http://svn.example.com/repos/calc bigwc
```

Теперь создание копии заключается в простой передаче двух путей в пределах рабочей копии команде `svn copy`:

```
$ cd bigwc
$ svn copy trunk branches/my-calc-branch
$ svn status
A + branches/my-calc-branch
```

В этом случае, команда `svn copy` рекурсивно копирует рабочую директорию trunk в новую рабочую директорию branches/my-calc-branch.

А теперь, простой способ создания ветки, о котором мы говорили раньше: команда `svn copy` может оперировать с двумя URL напрямую.

```
$ svn copy http://svn.example.com/repos/calc/trunk \  
http://svn.example.com/repos/calc/branches/my-calc-branch \  
-m "Creating a private branch of /calc/trunk."  
Committed revision 341.
```

В сущности, между этими двумя методами нет разницы. Оба варианта создают в правке 341 новую директорию и эта новая директория является копией `/calc/trunk`. Обратите внимание на то, что второй метод, кроме прочего, выполняет немедленную фиксацию. Эта процедура более проста в использовании, так как нет необходимости в создании рабочей копии, отражающей большое хранилище. В этом случае вам вовсе можно не иметь рабочей копии.

### Работа с веткой

После создания ветки проекта, можно создать новую рабочую копию для начала ее использования:

```
$ svn checkout http://svn.example.com/repos/calc/branches/my-calc-branch  
A my-calc-branch/Makefile  
A my-calc-branch/integer.c  
A my-calc-branch/button.c  
Checked out revision 341.
```

В этой рабочей копии нет ничего особенного; она является просто отражением другой директории хранилища. Когда вы фиксируете изменения, то если Салли сделает обновление, она их даже не увидит. Ее рабочая копия является копией `/calc/trunk`.

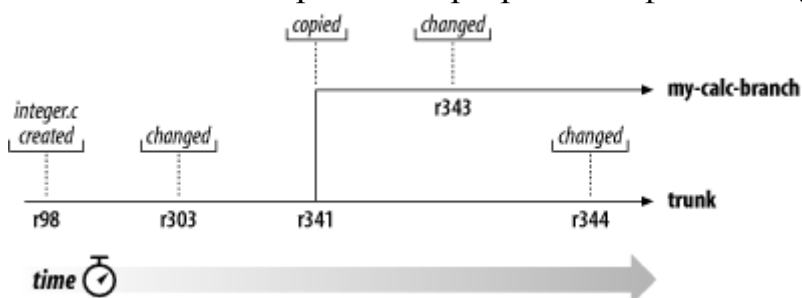
Предположим, что прошла неделя и были сделаны следующие фиксации:

Вы внесли изменения в `/calc/branches/my-calc-branch/button.c`, с созданием правки 342.

Вы внесли изменения в `/calc/branches/my-calc-branch/integer.c`, с созданием правки 343.

Салли внесла изменения в `/calc/trunk/integer.c`, с созданием правки 344.

Теперь есть два независимых направления разработки файла `integer.c`.



Пример истории ветвления для одного файла

Обратите внимание на то, что Subversion прослеживает историю ветки integer.c во времени полностью, в том числе пересекая точку создания копии. Создание ветки показывается как событие в истории, потому что файл integer.c был неявно скопирован, при копировании всей директории /calc/trunk/. Теперь давайте посмотрим, что будет когда такую же команду Салли выполнит для своей копии файла:

Салли увидит свои собственные изменения, а ваши сделанные в правке 343 нет.

### **Ключевые идеи, стоящие за ветками:**

- В отличие от других систем управления версиями, ветки в Subversion существуют в хранилище не в отдельном измерении, а как обычные нормальные директории файловой системы. Такие директории просто содержат дополнительную информацию о своей истории;

- Subversion не имеет такого понятия как ветка — есть только копии. При копировании директории результирующая директория становится «веткой» только потому что вы рассматриваете ее таким образом. Вы можете по-разному думать о директории, по-разному ее трактовать, но для Subversion это просто обычная директория которая была создана копированием.

### **Копирование изменений между ветками**

Сейчас вы и Салли работаете над параллельными ветками проекта: вы работаете над своей собственной веткой, а Салли работает над главной линией разработки (trunk).

В проектах, имеющих большое количество участников, когда кому-то необходимо сделать долгосрочные изменения, которые возможно нарушат главную линию, стандартной процедурой является создать отдельную ветку и фиксировать изменения туда пока работа не будет полностью завершена.

Чтобы избежать слишком большого расхождения веток, можно продолжать делиться изменениями по ходу работы. А тогда, когда ваша ветка будет полностью закончена, полный набор изменений ветки может быть скопирован обратно в основную ветку.

### **Копирование отдельных изменений**

Настал момент воспользоваться командой `svn merge`. Эта команда способна сравнивать любые два объекта в хранилище и показывать изменения, а также их применять.

Например, вы можете попросить `svn merge` применить правку 344 к рабочей копии в виде локальных изменений:

```
$ svn merge -r 343:344 http://svn.example.com/repos/calc/trunk
```

```
U integer.c
```

```
$ svn status
```

```
M integer.c
```

Изменения из правки 344, в другой ветке были «скопированы» из главной линии разработки в вашу рабочую копию, вашей личной ветки и теперь существуют в виде локальных изменений. С этого момента вы можете просмотреть локальные изменения и убедиться в том, что они корректно работают.

По другому сценарию, возможно, что не все будет так хорошо и `integer.c` может оказаться в состоянии конфликта. Вам необходимо будет при помощи стандартной процедуры решить конфликт.

После просмотра результата объединения изменений, можно их как обычно зафиксировать (`svn commit`). После этого изменения будут внесены в вашу ветку хранилища. В терминах контроля версий такую процедуру копирования изменений между ветками обычно называют *портированием изменений* (слиянием, “мерджем” (*merge*)).

### **Ключевые понятия, стоящие за слиянием**

Очень просто понять механизм того, как именно ведет себя `svn merge`.

В замешательство приводит, главным образом название команды. Термин «слияние» как бы указывает на то, что ветки соединяются вместе, или происходит какое-то волшебное смешивание данных. На самом деле это не так. Лучшим названием для этой команды могло быть `svn diff-and-apply` потому что это все, что происходит: сравниваются два файловых дерева хранилища, а различия переносятся в рабочую копию.

Команда принимает три аргумента:

1. Начальное дерево хранилища (как правило, называемое левой частью при сравнении),
2. Конечное дерево хранилища (как правило называемое правой частью при сравнении),
3. Рабочую копию для применения отличий, в виде локальных изменений (как правило, называемую целью слияния).

Когда эти три аргумента указаны, сравниваются два дерева и результирующие различия применяются к целевой рабочей копии в виде локальных изменений. Если результат вас не устраивает, просто отмените (`svn revert`) все сделанные изменения.

### **Конфликты при объединении**

Однако `svn merge` иногда не может гарантировать правильного и может вести себя более хаотично: пользователь может запросить сервер сравнить любые два дерева файлов, даже такие, которые не имеют отношения к рабочей копии! Из этого следует большое количество потенциальных человеческих ошибок.

Еще одно небольшое отличие между `svn update` и `svn merge` заключается в названиях файлов, создаваемых при возникновении конфликта. При обновлении создаются файлы с названиями `filename.mine`, `filename.rOLDREV`, и `filename.rNEWREV`. А `svn merge` в конфликтной ситуации создает три файла с названиями `filename.working`, `filename.left` и `filename.right`. Здесь, термины «left» и «right» указывают на две стороны сравнения, то есть на используемые при сравнении деревья.

### **Учитывать или игнорировать происхождение**

При общении разработчиков, использующих Subversion очень часто можно услышать упоминание термина *происхождение*. Это слово используется для описания отно-

шений между двумя объектами хранилища: если между ними есть связь, тогда говорят, что один объект является предком другого.

Мы обращаем на это ваше внимание, для того, чтобы указать на важные отличия между `svn diff` и `svn merge`. Первая команда игнорирует происхождение, в то время, как вторая его учитывает. Данный аспект поведения обеих команд можно переопределять. Подробнее об этом в `svn help`.

Мы рассмотрели необходимые операции для совместной работы с Subversion, теперь перейдём к Git.

## 4.2. Ветвление в Git

Ветвление Git очень легковесно. Операция создания ветки выполняется почти мгновенно, переключение между ветками - также быстро. В отличие от многих других СКВ, Git поощряет процесс работы, при котором ветвление и слияние выполняется часто, даже по несколько раз в день.

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

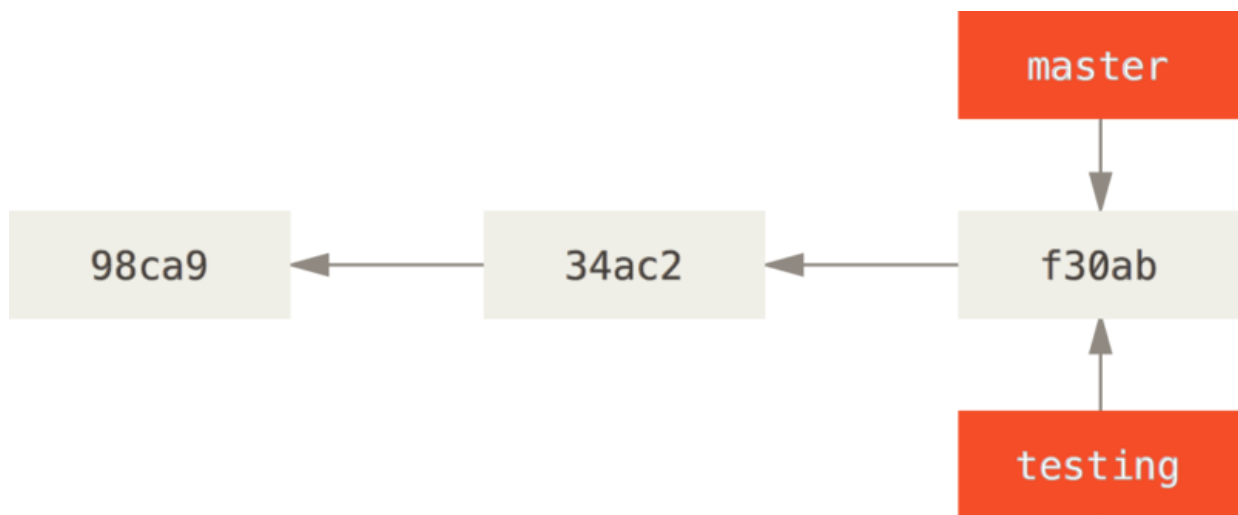
Когда вы делаете коммиты, то получаете основную ветку, указывающую на ваш последний коммит. Каждый коммит автоматически двигает этот указатель вперед.

Что же на самом деле происходит, когда вы создаете ветку? Всего лишь создается новый указатель для дальнейшего перемещения. Допустим вы хотите создать новую ветку с именем “testing” Вы можете это сделать командой **git branch**:

```
$ git branch testing
```

По умолчанию команда `git branch` создаёт новую ветку, но не переключается на неё.

В результате создается новый указатель на тот же самый коммит, в котором вы находитесь.



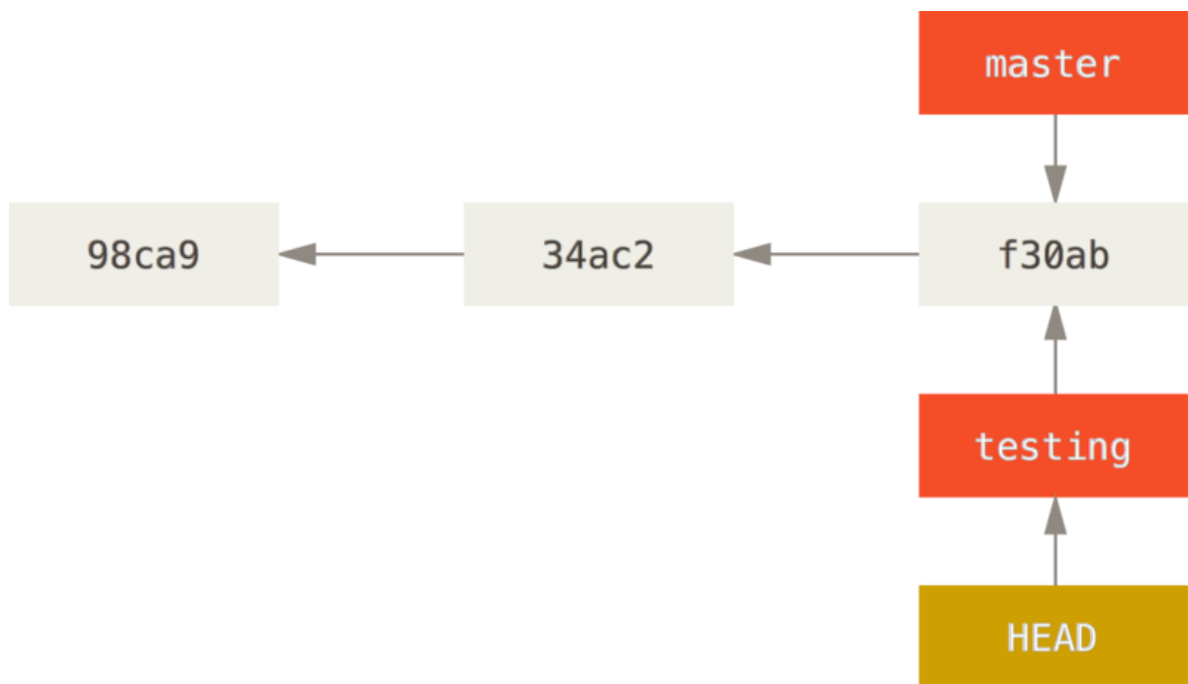
Создание указателя ветки testing

Для создания ветки в Git используется команда **git branch <branchname>**.

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

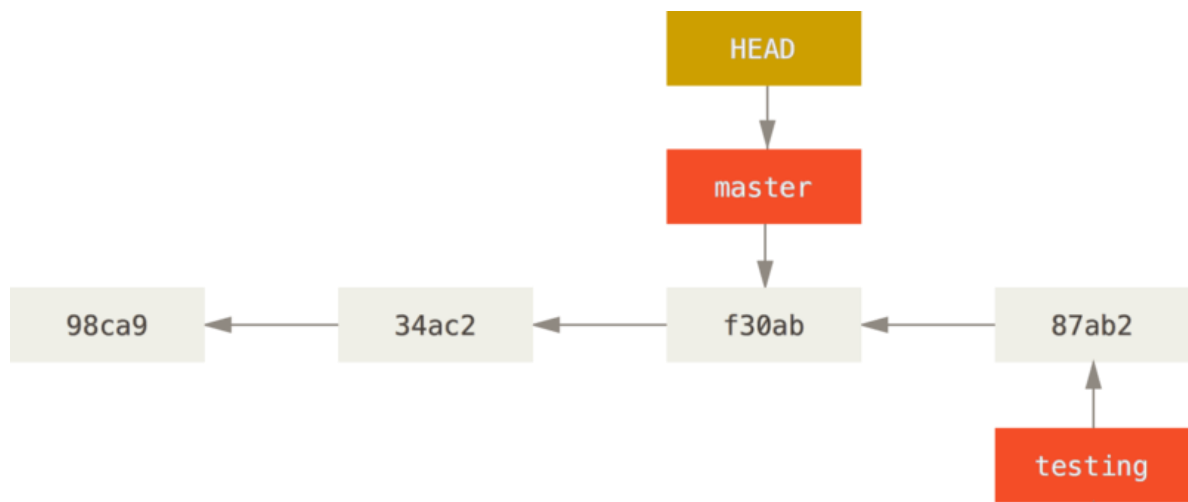
*\$ git checkout testing*

В результате указатель HEAD переместится на ветку testing.



Переход на ветку testing

Если мы сделаем ещё один коммит в ветке testing и перейдём на ветку master:



Вид репозитория при вышеперечисленных операциях

### Слияние веток

Слияние веток выполняется с помощью команды **git merge <branchname>**. Например, если мы хотим влить изменения из ветки `iss53` в текущую ветку, в которой мы находимся, необходимо выполнить:

```

$ git merge iss53
Updating f483254..3a0874c
Fast forward
 README /      1-
 1 file changed, 0 insertions( + ), 1 deletions( - )

```

### Просмотр истории изменений с ветвлениями в удобном виде

Вы можете увидеть историю ревизий помощи команды `git log`. Команда `git log --oneline --decorate --graph --all` выдаст историю ваших коммитов и покажет, где находятся указатели ваших веток, и как ветвилась история проекта.

```

$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
/ * 87ab2 (testing) made a change
//
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project

```

Создание и удаление веток совершенно не затратно, так как ветка в Git — это всего лишь файл, содержащий 40 символов контрольной суммы SHA-1 того коммита, на который он указывает. Создание новой ветки совершенно быстро и просто — это всего лишь запись 41 байта в файл (40 знаков и перевод строки).

Это совершенно отличает Git от ветвления в большинстве более старых систем контроля версий, где все файлы проекта копируются в другой подкаталог. Там ветвление для проектов разного размера может занять от секунд до минут. В Git ветвление всегда мгновенное. Эти возможности побуждают разработчиков чаще создавать и использовать ветки.



## Разрешение конфликтов слияния

Иногда процесс не проходит гладко. Если вы изменили одну и ту же часть одного и того же файла по-разному в двух объединяемых ветках, Git не сможет самостоятельно объединить их. Если ваше исправление ошибки #53 потребовало изменить ту же часть файла, что и hotfix, вы получите примерно такое сообщение о конфликте слияния:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git не создал коммит слияния автоматически. Он остановил процесс до тех пор, пока вы не разрешите конфликт. Чтобы в любой момент после появления конфликта увидеть, какие файлы не объединены, вы можете запустить git status:

```
$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
```

```
Unmerged paths:
(use "git add <file>..." to mark resolution)
   both modified:   index.html
no changes added to commit (use "git add" and/or "git commit -a")
```

Всё, где есть неразрешенные конфликты слияния, перечисляется как неслитое. Git добавляет в конфликтующие файлы стандартные пометки разрешения конфликтов, чтобы вы могли вручную открыть их и разрешить конфликты.

В вашем файле появился раздел, выглядящий примерно так:

```
<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
please contact us at support@github.com
</div>
>>>>>> iss53:index.html
```

Способы разрешения конфликта в git такие же, как и в Subversion:

- Применить чужое изменения:

```
git merge -X theirs branch;
```

- Применить свои изменения:

```
git merge -s ours;
```

- Вручную объединить файлы (посредством ручного редактирования файла с метками конфликта).

Как только вы завершили слияние, зафиксируйте изменения в репозитории командой git commit.

### 4.3. Серверная работа с Git

Для организации совместной работы необходимо разместить (to host) файл на удалённом сервере. Внутри одной организации никто не мешает сделать этого на одном из компьютеров, но это не совсем удобно, т.к. работа будет зависеть от питания этого компьютера.

Для хостинга можно использовать свой сервер или использовать готовое решение, платное или бесплатное, в зависимости от целей проекта. Самыми популярными решениями в настоящее время являются GitHub, Bitbucket, GitLab и другие.

На своём гит – сервере можно настроить доступ по протоколам:

- HTTP – распространённый протокол, но сложнее остальных в настройке;
- SSH – легко настраивается, или присутствует на большинстве серверов, но отсутствует возможность анонимного доступа к репозиторию;
- Git – самый быстрый протокол, но не имеет аутентификации.

Мы в лабораторных работах будем работать с сервисом GitLab, развёрнутым в РГРТУ, и с ним познакомимся подробнее.

#### GitLab

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

#### Совместная работа

Самый простой метод совместной работы над проектом GitLab — это выдача другому пользователю прямого доступа на запись (push) в git-репозитории. Вы можете добавить пользователя в проект в разделе “Участники” (“Members”) настроек проекта, указав уровень доступа. Получая уровень доступа “Разработчик” (“Developer”) или выше, пользователь может отправлять свои коммиты и ветки непосредственно в репозиторий.

Другой, также распространённый способ совместной работы — использование запросов на слияние (merge requests, pull requests). Эта возможность позволяет любому пользователю, который видит проект, вносить свой вклад подконтрольным способом. Пользователи с прямым доступом могут просто создать ветку, отослать в неё коммиты и открыть запрос на слияние из их ветки обратно в master или любую другую ветку. Пользователи без доступа на запись могут сделать собственную копию, “форкнуть” репозиторий (“fork”, создать собственную копию), отправить коммиты в эту копию и открыть запрос на слияние из их форка обратно в основной проект. Эта модель позволяет владельцу полностью контролировать, что попадает в репозиторий и когда, принимая помощь от недоверенных пользователей.

Запросы на слияние и проблемы (issues) это основные источники долгоживущих дискуссий в GitLab. Каждый запрос на слияние допускает построчное обсуждение предлагаемого изменения. И те и другие могут присваиваться пользователям или организовываться в наборы задач/багов (milestones).

Мы в основном сосредоточились на частях GitLab, связанных с git, но это далеко не все возможности, среди которых, например, вики-страницы для проектов и инструменты поддержки системы. Одно из преимуществ GitLab в том, что, однажды запустив и настроив сервер, вам редко придётся изменять конфигурацию или заходить на него по SSH; большинство административных и пользовательских действий можно выполнять через веб-браузер.

Процедура регистрации чрезвычайно проста и её мы рассматривать не будем. Если вы работаете с сервисом РГРТУ GitLab, то вы уже зарегистрированы, и вам требуется только сменить свой пароль.



You need to sign in or sign up before continuing.

## GitLab Community Edition

### Open source software to collaborate on code

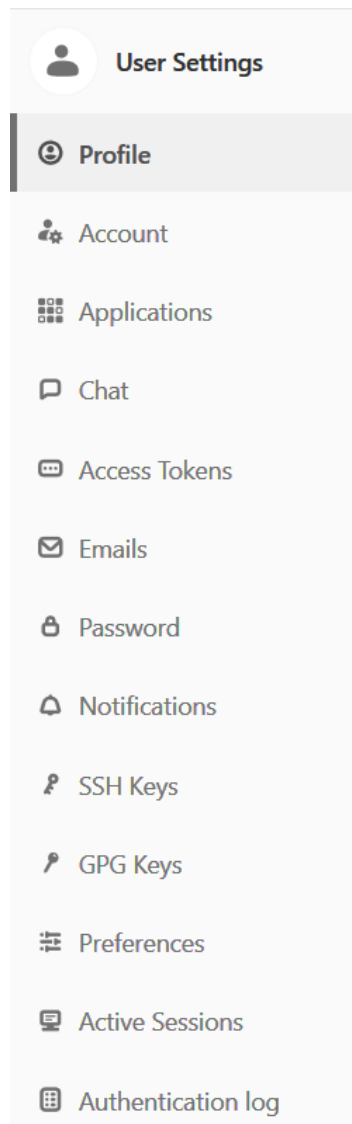
Manage Git repositories with fine-grained access controls that keep your code secure. Perform code reviews and enhance collaboration with merge requests. Each project can also have an issue tracker and a wiki.

| LDAP  | Standard |
|---|----------|
| <b>Username or email</b><br><input type="text" value="email.example@email.ru"/>       |          |
| <b>Password</b><br><input type="password" value="••••••••"/>                          |          |
| <input checked="" type="checkbox"/> Remember me <a href="#">Forgot your password?</a> |          |
| <input type="button" value="Sign in"/>  |          |

[Explore](#) [Help](#) [About GitLab](#)

Стартовое окно авторизации

Рассмотрим основные функции окна сервиса GitLab.

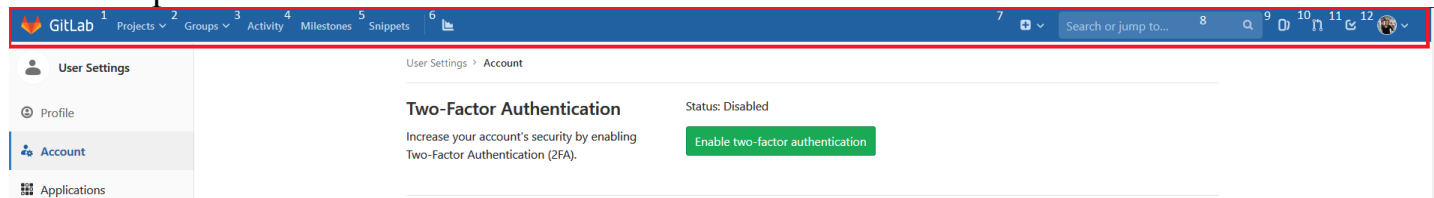


Панель настроек

- 1) «Profile» вы можете настроить свой аватар, статус, ФИО, электронную почту, настройки приватности, язык и указать контакты в различных социальных сетях, указать организацию. После изменения настроек не забудьте нажать «Update profile settings» для их сохранения;
- 2) «Account» вы можете удалить свой аккаунт или настроить двухфакторную аутентификацию через приложение;
- 3) «Applications» вы можете настроить, какие сторонние приложения могут иметь доступ к вашему репозиторию (производить действия pull и push);
- 4) «Chat» – здесь можно настроить чаты внутри GitLab (делается администраторами);
- 5) «Access Tokens» – генерация токенов для приложений, использующих GitLab API, так же для аутентификации через HTTP;
- 6) «Emails» – добавление дополнительной электронной почты;
- 7) «Password» – изменение пароля;
- 8) «Notifications» – оповещения на электронную почту об изменениях в проекте другими его участниками;
- 9) «SSH Keys» – настройка безопасного соединения между вашим компьютером и GitLab;
- 10) «GPG Keys» – верификация подписанных коммитов (signed commits);

- 11) «Preferences» – настройка цветовой гаммы GitLab, темы при просмотре исходных текстов кода в браузере и так далее;
- 12) «Active Sessions» – незавершенные сессии на другом компьютере. Здесь можно их остановить;
- 13) «Authentication Log» – журнал авторизаций.

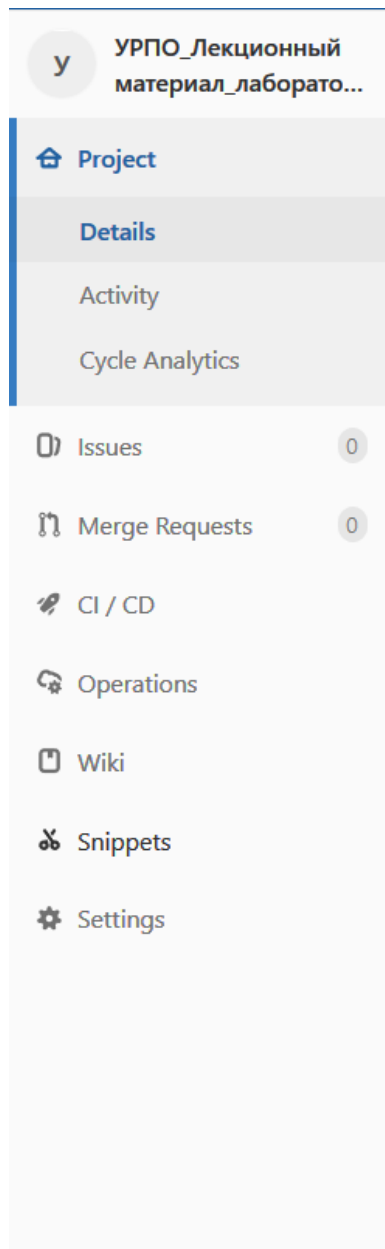
В верхней панели показаны:



Верхняя панель GitLab

- 1) «Projects» - ваши проекты, также проекты, оцененные вами, поиск проектов;
- 2) «Groups» - ваши группы (группой называется несколько проектов, объединённых одной тематикой), поиск групп;
- 3) «Activity» – последняя активность в ваших проектах или в оцененных вами (push, merge, issue (bug), комментарии);
- 4) «Milestones» – группа задач;
- 5) «Snippets» – позволяет сохранить нужные фрагменты кода в публичном или приватном доступе, если сниппет открыт для других пользователей, есть возможность комментирования;
- 6) «Instance Statistics» – статистика того, как вы используете GitLab, даже в сравнении с другими организациями;
- 7) «New» – новый проект или сниппет;
- 8) «Search» – поиск по вашим мёрдж реквестам или по вашим запросам на изменение;
- 9) «Issues» – задачи вашего проекта (дефекты);
- 10) «Merge Requests» – запросы на изменение / исправления вашего проекта;
- 11) «Todos» – список того, чего нужно сделать;
- 12) «Profile» – настройки профиля, статистика.

Настройки проекта:



Настройки проекта в GitLab

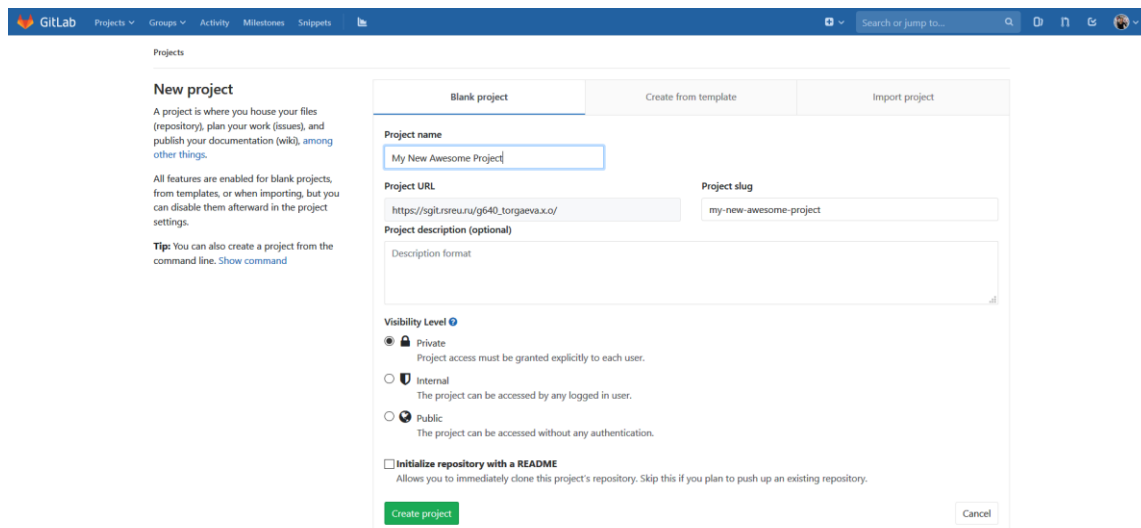
- 1) «Project» - детали проекта. Здесь можно настроить URL проекта, оповещения о действиях на вашу почту, добавить файлы в проект прямо через браузер. Во вкладке «Activity» вы можете увидеть активности проекта – коммиты, запросы на изменение (pull requests), задачи проекта, баги, комментарии. Во вкладке «Cycle Analytics» вы можете посмотреть встроенную в GitLab аналитику проекта – завершённые и открытые задачи, количество тестов;
- 2) Во вкладке «Issues» находятся открытые и закрытые задачи и баги;
- 3) «Merge Requests» – запросы на изменение. Текущие и уже выполненные;
- 4) «CI/CD» – Continuous Integration/Continuous Deployment – о нём будет рассказано позднее в нашем курсе, в двух словах можно определить CI и CD как набор инструментов для автоматизации развёртывания приложения и установки его на машину;
- 5) «Operations» – настройки CI/CD;
- 6) «Wiki» – способ написания документации к вашему проекту;
- 7) «Snippets» – то же самое, что сниппеты выше. Только эти непосредственно относятся к вашему проекту;

- 8) «Settings» – различные настройки интеграции, настройки участников команды, различные разрешения, экспортирование проекта.

## Создание проекта в GitLab

Рассмотрим пример создания нового проекта:

1. В окне создания проекта можно настроить его имя, URL, описание проекта, является ли он публичным и приватным. Также вам будет предложено проинициализировать проект, добавив в него файл README.

The screenshot shows the 'New project' page in GitLab. On the left, there is a sidebar with the title 'New project' and a description: 'A project is where you house your files (repository), plan your work (issues), and publish your documentation (wiki), among other things.' Below this, it states 'All features are enabled for blank projects, from templates, or when importing, but you can disable them afterward in the project settings.' and a tip: 'Tip: You can also create a project from the command line. Show command'. The main form has three tabs: 'Blank project' (selected), 'Create from template', and 'Import project'. Under 'Blank project', there are fields for 'Project name' (filled with 'My New Awesome Project'), 'Project URL' (filled with 'https://sgit.sreu.ru/g640\_torgaeva.x.o/'), and 'Project slug' (filled with 'my-new-awesome-project'). There is also a 'Project description (optional)' field. Below these is the 'Visibility Level' section with three radio buttons: 'Private' (selected), 'Internal', and 'Public'. Each has a brief description. At the bottom, there is a checkbox for 'Initialize repository with a README' and a 'Create project' button.

Окно создания нового проекта

2. Сразу после этого можно будет создать рабочую копию на своём компьютере и работать с проектом – для этого перейдите на страницу с проектом и нажмите “clone”, скопируйте url и создайте рабочую копию репозитория в выбранной вами папке с помощью команды **git clone <url>**.

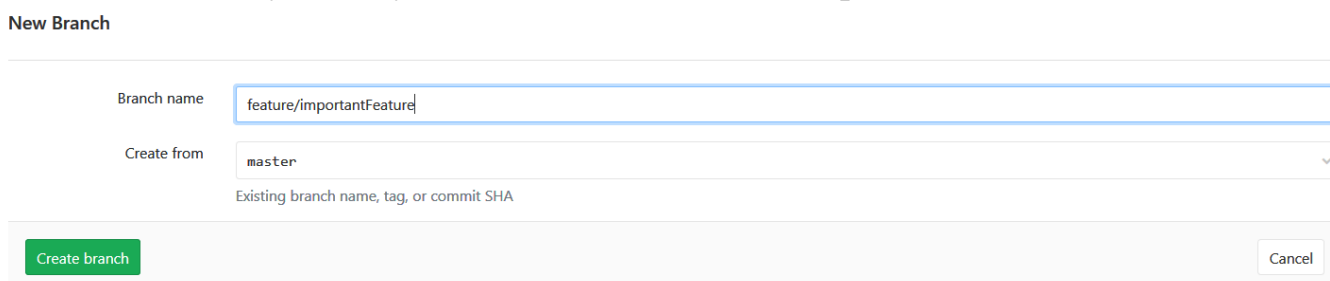
## Запросы на изменение

Выше мы говорили о таком способе взаимодействия в проекте, как запрос на изменение, или pull request (pr). Пулл реквесты применяются в случаях:

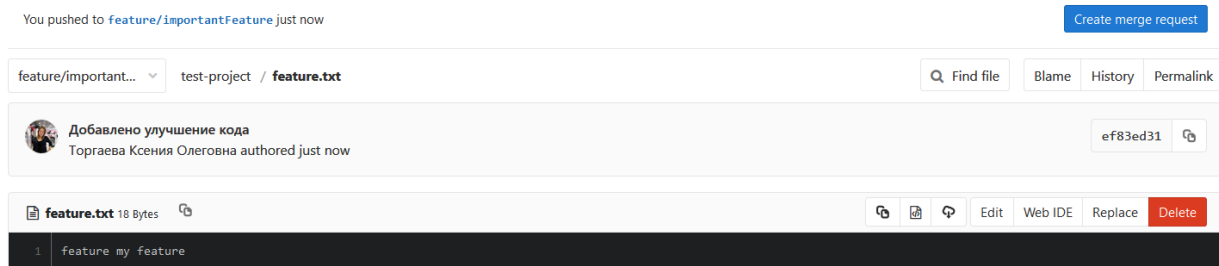
- если у вас нет доступа в репозиторий, в который вы хотите внести изменения. Это может быть проект с открытым исходным кодом, в который вы хотите внести свой вклад;
- если вы хотите, чтобы другие люди посмотрели ваш код (make a review) перед тем, как влить изменения в другую ветку (в рамках текущего репозитория, над которым в работаете);

Ниже мы рассмотрим пример заведения pull request’а в GitLab:

1. Мы создали новую ветку и поместили в ней новый файл.

The screenshot shows the 'New Branch' form in GitLab. It has a title 'New Branch'. There are two main fields: 'Branch name' (filled with 'feature/importantFeature') and 'Create from' (a dropdown menu showing 'master'). Below the dropdown, it says 'Existing branch name, tag, or commit SHA'. At the bottom, there is a 'Create branch' button and a 'Cancel' button.

Создание ветки в GitLab



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

## 2. Мы хотим внести свои изменения в основную ветку “master”. Нажимаем «create pull request».

**New Merge Request**

From **feature/importantFeature** into **master** [Change branches](#)

Title:

[Start the title with WIP:](#) to prevent a **Work In Progress** merge request from being merged before it's ready.

[Add description templates](#) to help your contributors communicate effectively!

Description:

**Write** [Preview](#)

Я сделал важное улучшение в коде (подробное описание)  
Рассмотрите, пожалуйста!

Markdown and [quick actions](#) are supported [Attach a file](#)

Source branch:

Target branch:  [Change branches](#)

☐ Remove source branch when merge request is accepted.

☐ Squash commits when merge request is accepted. [About this feature](#)

[Submit merge request](#) [Cancel](#)

Окно создания запроса на изменение

Затем подробно описываем наше изменение, выбираем ветку, из которой будем брать изменения (source branch), и ветка, в которую хотим влить наши изменения (target branch). В нашем случае ими являются, соответственно feature/importantFeature и master, и нажимаем “submit merge request”.

## Fork проекта

Можно воспринимать форк проекта как копию чужого репозитория 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 необходимо установить права на запись (права “developer”)).



## GitHub

GitHub это крупнейшее хранилище Git репозиторий, а так же центр сотрудничества для миллионов разработчиков и проектов. Мы коротко рассматриваем GitHub в нашем курсе, так как огромный процент репозиторий хранится именно на нём, и сейчас GitHub является стандартом де-факто разработки ПО с открытым исходным кодом. Вам придётся взаимодействовать с GitHub при профессиональном использовании Git.

В большинстве случаев на GitHub размещают проекты с открытым исходным кодом, так как для них существует бесплатные аккаунты GitHub. Если вы хотите пользоваться приватными репозиториями, которые будут не видны всему миру, то на GitHub необходимо купить платную подписку. Хотя большинство организаций, которые заинтересованы в приватных репозиториях, покупает подписку на BitBucket Server, или GitLab.

Разберём на примерах часто возникающие ситуации, когда вам нужно использовать GitHub:

- 1) создать свой проект с открытым исходным кодом / использовать как хранилище кода, который не хотелось бы потерять;
- 2) воспользоваться сторонней библиотекой, являющейся нестандартной;
- 3) внести вклад в уже существующий проект на GitHub.

Для начала рассмотрим вид проекта на GitHub:

The screenshot shows the GitHub interface for a repository named 'high\_load' by user 'Ksenia989'. At the top, there are navigation tabs: 'Code', 'Issues' (0), 'Pull requests' (0), 'Projects' (0), 'Wiki', 'Insights', and 'Settings'. Below the repository name, a description reads: 'Небольшой проект для чемпионата High load Cup (python, Django) (тз [https://github.com/sat2707/hlcupdocs/blob/master/TECHNICAL\\_TASK.md](https://github.com/sat2707/hlcupdocs/blob/master/TECHNICAL_TASK.md))'. A 'Manage topics' link is also present. A summary bar shows '26 commits', '1 branch', '0 releases', and '1 contributor'. A yellow security alert banner states: 'We found potential security vulnerabilities in your dependencies. Only the owner of this repository can see this message. Manage your notification settings or learn more about vulnerability alerts.' Below this, there are buttons for 'Branch: master', 'New pull request', 'Create new file', 'Upload files', 'Find file', and 'Clone or download'. The file list shows various files with their commit dates: '.idea' (10 months ago), 'sightseens' (10 months ago), 'usrJourney' (11 months ago), '.gitignore' (11 months ago), 'Dockerfile' (10 months ago), 'README.md' (10 months ago), 'db.sqlite3' (10 months ago), 'imports.py' (11 months ago), 'manage.py' (11 months ago), and 'requirements.txt' (10 months ago). The 'README.md' file is selected, showing its content: 'high\_load' followed by the project description: 'Небольшой проект для чемпионата High load Cup (python, Django)'.

## Вид проекта на GitHub

Вверху располагается имя автора и название проекта, люди, следящие за репозиторием. Так же располагается информация о трекере задач, запросов на изменение, википедия и настройки.

Ниже можно увидеть файлы проекта. А в самом низу – файл README.md. данный файл является текстовым описанием проекта и небольшой документацией. В нём можно коротко рассказать о проекте, чтобы было понятно, подходит ли пользователю ваше решение.

**1.** Создание нового проекта мы подробно рассматривать не будем, так как оно не сильно отличается от создания проекта на Bitbucket. Ещё раз хочется упомянуть о правильном выборе лицензии уже для своего проекта. Это важно из – за правовых аспектов разработки.

Так же, если вы просто хотите хранить проекта на GitHub, вы можете создать бесплатный открытый репозиторий, видный всем, под лицензией, например, MIT.

Открытый проект имеет некоторые плюсы, например, можно включить ссылку на свой GitHub в резюме (если в нём есть, на что посмотреть, а не редкие коммиты раз в месяц).

## 2. Использование сторонних библиотек.

Университетское образование призвано научить думать самостоятельно для того, чтобы понимать концепции алгоритмов и разработки программного обеспечения. Но в практической работе чаще всего вы будете использовать готовые решения, а свои решения писать, если не будет найдено готового решения или решения, должным образом подходящего под ваши задачи. Зачастую написание своих «велосипедов» не оправдано.

Рассмотрим пример.

Например, вам в приложении нужна библиотека для языка C++, умеющая работать с форматом файла YAML, и после поиска оказывается, что в стандарте такой библиотеки нет. Например, мы находим её на GitHub: <https://github.com/jbeder/yaml-cpp>.

jbeder / yaml-cpp

Watch 102 Star 1,200 Fork 410

Code Issues 135 Pull requests 20 Projects 0 Wiki Insights

A YAML parser and emitter in C++

777 commits 4 branches 17 releases 52 contributors MIT

Branch: master New pull request Find file Clone or download

| File             | Commit Message   | Time         |
|------------------|--|--------------|
| include/yaml-cpp | fix up static, so works as DLL (#559)                            | 2 months ago |
| src              | Some small changes as a result of using a static analyzer (#643) | 7 days ago   |
| test             | Skip newlines in binary decoding (Fix #387) (#616)               | 3 months ago |
| util             | Improvements to CMake buildsystem                                | 3 months ago |
| .clang-format    | Update .clang-format to use C++ 11 style.                        | 3 years ago  |
| .codedocs        | test: Upgrade googlemock 1.7.0 to googletest 1.8.0               | a year ago   |
| .gitignore       | Enable items to be removed from a sequence (#582)                | 7 months ago |
| .travis.yml      | travis: Exclude linux/clang from the build matrix                | a year ago   |
| CMakeLists.txt   | Don't stomp on build flags (#635)                                | 11 days ago  |

yaml-cpp проект разработчика jbeder

На что следует сразу обратить внимание, когда вы хотите добавить библиотеку к себе в проект?

- 1) В первую очередь, на то, подходит ли вам этот продукт. Рассмотрите документацию, удовлетворяет ли данное приложение или библиотека целям, для которых вы хотите её использовать;
- 2) Лицензия. Если вам подошёл продукт, посмотрите под какой лицензией он распространяется (т.е. условия использования данного продукта). Подходят ли вам условия использования программы. Лицензия может предполагать включение информации об авторе в исходный код вашей программы, или не предполагать оно. Так же автор может нести ответственность за поломку косвенным образом из – за ошибки в библиотеке (но чаще всего не несёт). Подробнее о лицензиях проектов, распространяющихся с открытым исходным кодом, можно прочитать здесь: <https://habr.com/post/243091/>. Обычно берутся стандартные лицензии типа MIT

(продукт распространяется «как есть», не нужно указывать в исходном коде авторство создателя библиотеки);

- 3) Если проект подходит вам по функциональности и типу лицензии, то смело нажимайте «Clone or download», получайте репозиторий и используйте. Зачастую в проекте или библиотеке поддерживаются разные версии, которые можно найти по тегам.

**3.** Если вам понравилось приложение / библиотека, и вы хотите улучшить её, или вы нашли в ней уязвимость, которую, как вам кажется, вы могли бы исправить, вы можете помочь проекту с открытым исходным кодом и внести свой вклад.

Вы можете клонировать себе код, внести изменения в новой ветви, тщательно протестировать и сделать запрос на слияние.

Зачастую у подобных проектов весьма ограниченное (или отсутствующее) финансирование, и любой будет рад подобному вкладу.

#### Использованные источники:

- <https://git-scm.com/book/ru/v2/%D0%92%D0%B5%D1%82%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B2-Git-%D0%9E-%D0%B2%D0%B5%D1%82%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B8-%D0%B2-%D0%B4%D0%B2%D1%83%D1%85-%D1%81%D0%BB%D0%BE%D0%B2%D0%B0%D1%85>