Лабораторная работа 4. Системы контроля версий.

Система контроля версий — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии.

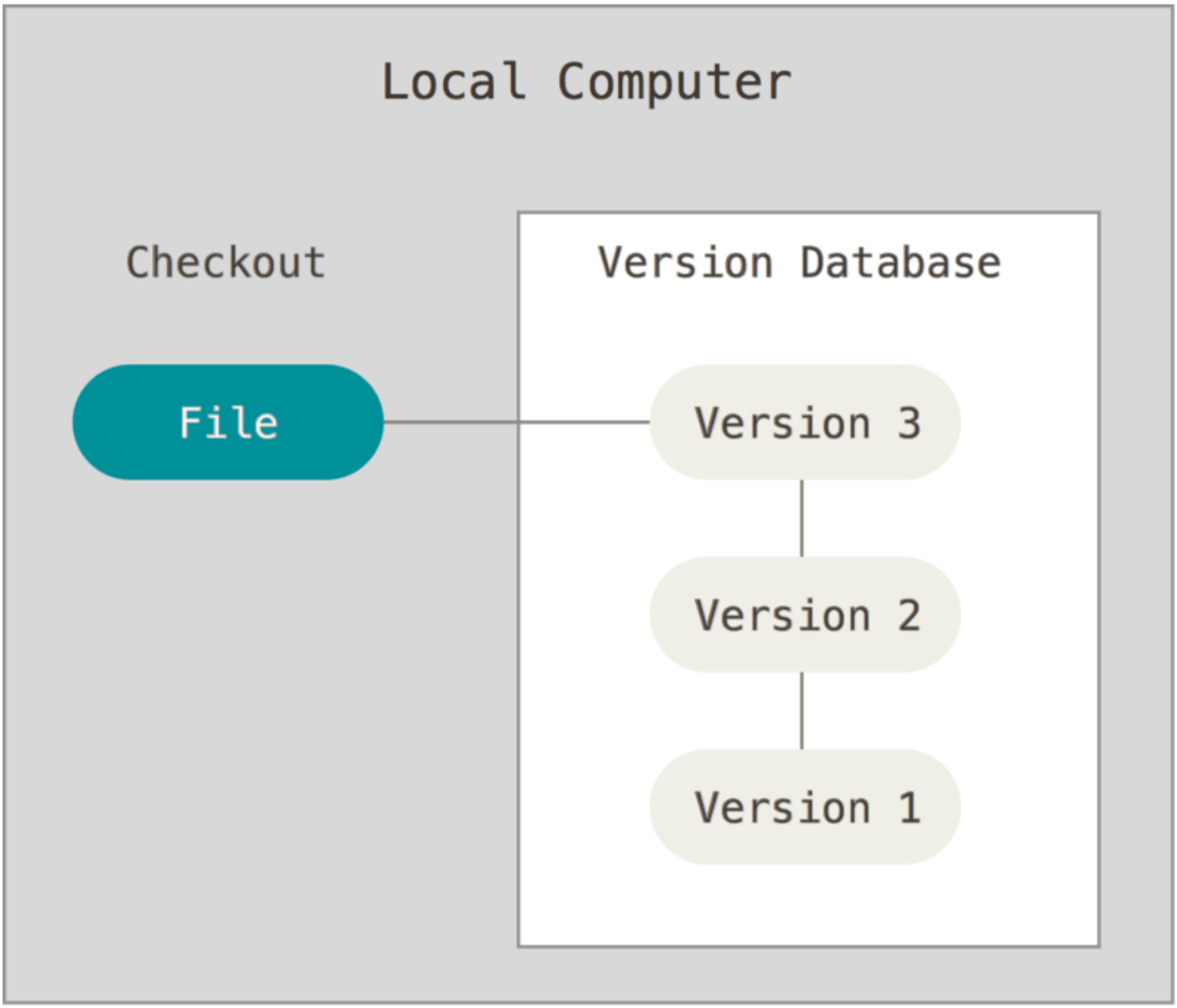
Системы контроля версий бывают следующих типов:

- Локальные системы контроля версий
- Централизованные системы контроля версий
- Распределённые системы контроля версий

Локальные системы контроля версий

Многие люди в качестве метода контроля версий применяют копирование файлов в отдельный каталог (возможно даже, каталог с отметкой по времени, если они достаточно сообразительны). Данный подход очень распространён из-за его простоты, однако он невероятно сильно подвержен появлению ошибок. Можно легко забыть в каком каталоге вы находитесь и случайно изменить не тот файл или скопировать не те файлы, которые вы хотели.

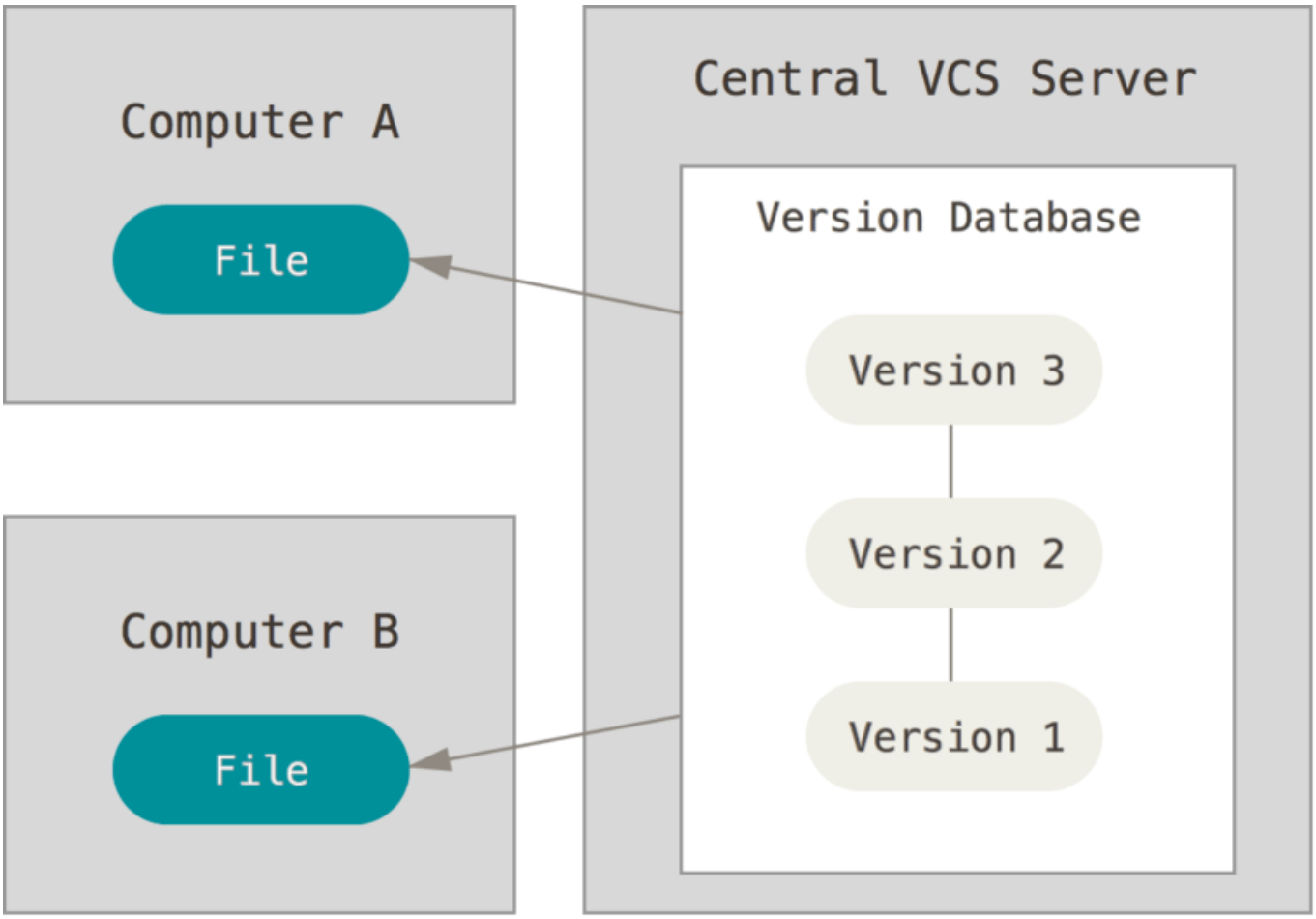
Для того, чтобы решить эту проблему, программисты давным-давно разработали локальные СКВ с простой базой данных, которая хранит записи о всех изменениях в файлах, осуществляя тем самым контроль ревизий.



Одной из популярных СКВ была система RCS, которая и сегодня распространяется со многими компьютерами. RCS хранит на диске наборы патчей (различий между файлами) в специальном формате, применяя которые она может воссоздавать состояние каждого файла в заданный момент времени.

Централизованные системы контроля версий

Следующая серьёзная проблема, с которой сталкиваются люди, — это необходимость взаимодействовать с другими разработчиками. Для того, чтобы разобраться с ней, были разработаны централизованные системы контроля версий (ЦСКВ). Такие системы, как CVS, Subversion и Perforce, используют единственный сервер, содержащий все версии файлов, и некоторое количество клиентов, которые получают файлы из этого централизованного хранилища. Применение ЦСКВ являлось стандартом на протяжении многих лет.



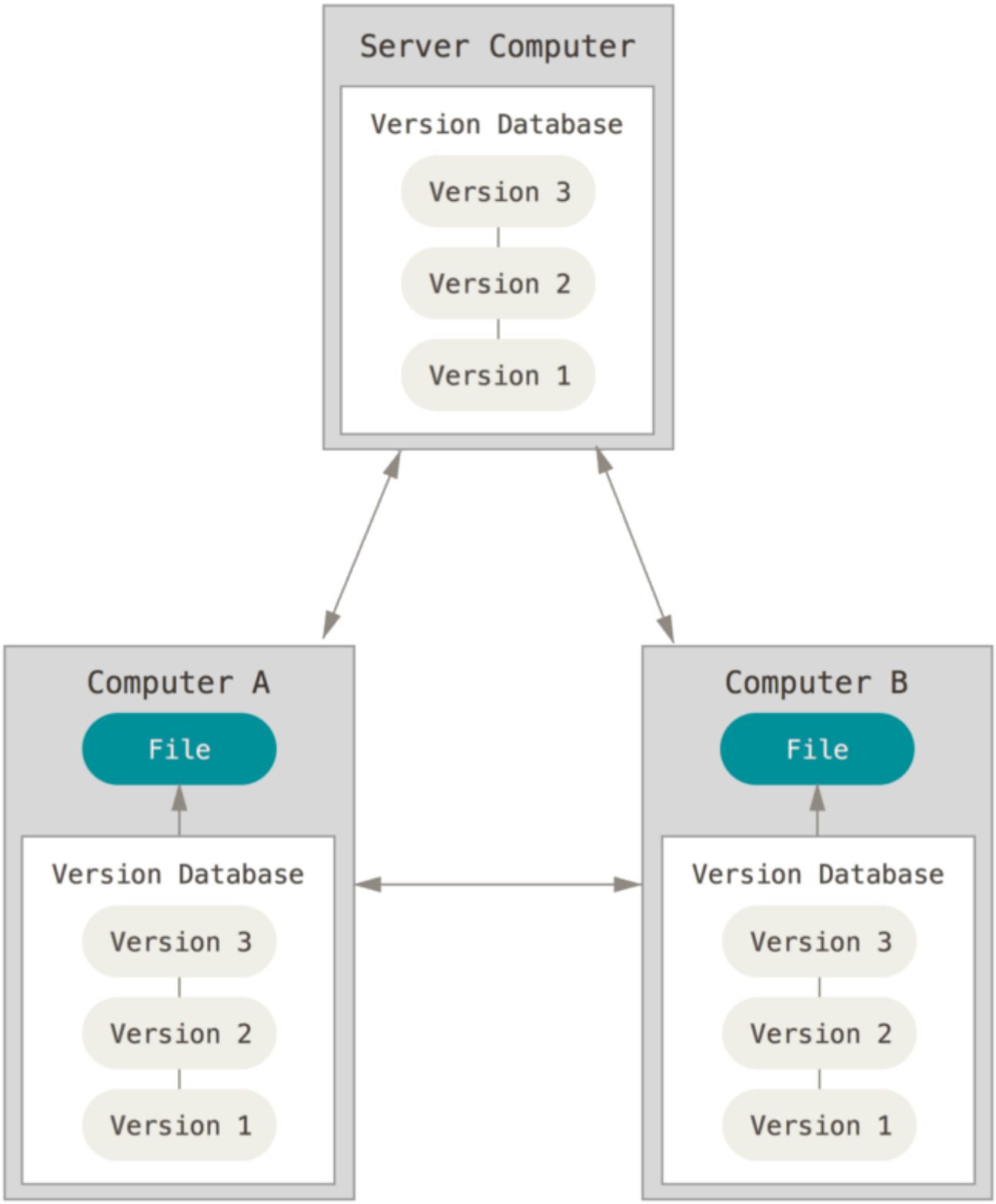
Такой подход имеет множество преимуществ, особенно перед локальными СКВ. Например, все разработчики проекта в определённой степени знают, чем занимается каждый из них. Администраторы имеют полный контроль над тем, кто и что может делать, и гораздо проще администрировать ЦСКВ, чем оперировать локальными базами данных на каждом клиенте.

Несмотря на это, данный подход тоже имеет серьёзные минусы. Самый очевидный минус — это единая точка отказа, представленная централизованным сервером. Если этот сервер выйдет из строя на час, то в течение этого времени никто не сможет использовать контроль версий для сохранения изменений, над которыми работает, а также никто не сможет обмениваться этими изменениями с другими разработчиками. Если жёсткий диск, на котором хранится центральная БД, повреждён, а своевременные бэкапы отсутствуют, вы потеряете всё — всю историю проекта, не считая единичных снимков репозитория, которые сохранились на локальных машинах разработчиков. Локальные СКВ страдают от той же самой проблемы: когда вся история проекта хранится в одном месте, вы рискуете потерять всё.

Распределённые системы контроля версий

Здесь в игру вступают распределённые системы контроля версий (РСКВ). В РСКВ (таких как Git, Mercurial, Bazaar или Darcs) клиенты не просто скачивают снимок всех файлов

(состояние файлов на определённый момент времени) — они полностью копируют репозиторий. В этом случае, если один из серверов, через который разработчики обменивались данными, умрёт, любой клиентский репозиторий может быть скопирован на другой сервер для продолжения работы. Каждая копия репозитория является полным бэкапом всех данных.



Более того, многие РСКВ могут одновременно взаимодействовать с несколькими удалёнными репозиториями, благодаря этому вы можете работать с различными группами людей, применяя различные подходы единовременно в рамках одного проекта. Это позволяет применять сразу несколько подходов в разработке, например, иерархические модели, что совершенно невозможно в централизованных системах.

Основные консольные команды для работы с репозиторием Git

Создание репозитория

```
git init
```

Данную команду необходимо выполнять, находясь в папке с файлами, которые должны стать основой проекта.

Клонирование репозитория из удаленного хранилища

Как правило при помощи команды **git clone** создается копия указанного удаленного репозитория. Эта команда создает новую папку по названию репозитория и скачивает в нее файлы. Исходный репозиторий может находиться в локальной файловой системе или на удаленном сервере, к которому можно получить доступ с помощью поддерживаемых протоколов.

Пример клонирования проекта в дерикторию *how-to* по ssh-ключу:

```
git clone git@f.dev.gitflic.ru:user/how-to.git
```

Аргумент **-branch** позволяет выбрать ветку для клонирования. В противном случае клонируется ветка, на которую указывает **HEAD** в удаленном репозитории (обычно главная ветка). Кроме того, для этих целей в команде можно задать тег вместо ветки:

```
git clone –branch new_feature git://remoterepository.git
```

git clone --bare как и **git init --bare**, аргумент *-bare* при назначении команде *git clone* приводит к созданию копии удаленного репозитория без рабочего каталога. Это означает, что репозиторий будет содержать историю проекта, к которой можно выполнять запросы **push** и **pull**, но которую нельзя редактировать напрямую. Кроме того, в репозитории, клонированном с опцией -bare, не будут настроены удаленные ветки. Как и **git init --bare**, эта команда создает удаленный репозиторий, который разработчики не смогут редактировать напрямую.

git clone --mirror Вместе с аргументом --mirror команде неявно назначается и аргумент --bare. Поэтому можно сказать, что опция --mirror наследует поведение --bare, создавая чистый репозиторий без изменяемых рабочих файлов. Кроме того, --mirror клонирует ссылки удаленного репозитория и сохраняет конфигурацию отслеживания удаленных веток. Затем вы можете выполнить команду git remote update на созданном зеркале, в результате чего будут перезаписаны все ссылки из исходного репозитория. Так вы получите идентичные функциональные возможности для работы.

Добавление изменений в рабочей дерриктории в индекс

Команда **git add** добавляет изменения рабочей директории в промежуточную область. Он сообщает git, что вы хотите включить обновления определенного файла в следующий коммит. Однако git add на самом деле не влияет на репозиторий каким—либо существенным образом - изменения фактически не записываются до тех пор, пока вы не выполните **git commit**.

Добавление всех изменений в следующий коммит выполняется следующей командой:

```
git add .
```

Чтобы добавить в следующий коммит только один измененный файл, то это делается командой, где - полное имя файла:

```
git add <file-name>
```

В сочетании с этими командами вам также понадобится **git status** для просмотра состояния рабочего каталога и промежуточной области:

```
git status
```

Сохранение проиндексированных изменений

Коммит проиндексированного состояния кода производится по следующей команде:

```
git commit
```

Стоит отметить, что эта команда откроет текстовый редактор, введите комментарий к коммиту. После ввода сохраните файл и закройте текстовый редактор, чтобы выполнить коммит. Выполнение коммита со всеми изменениями в рабочей директории. Эта команда

включает только изменения отслеживаемых файлов (которые были добавлены командой **git add**):

```
git commit –a
```

Данная команда создаст коммит с указанным комментарием. По умолчанию команда **git commit** открывает локальный текстовый редактор для ввода комментария к коммиту. При передаче параметра **-m** используется добавленный комментарий, минуя текстовый редактор:

```
git commit –m "commit message"
```

Также есть параметр, который позволяет команде commit изменять последний коммит. Вместо создания нового, все изменения добавляются в последний. Кроме того, после выполнения команды откроется текстовый редактор и предложит изменить ранее указанный комментарий к комиту:

```
git commit ––amend
```

Работа с ветками

```
git checkout –b develop
```

Также можно выполнить то же самое действие в 2 команды, отдельно создать ветку, отдельно в нее переключиться:

```
git branch develop
git checkout develop
```

Вывод списка всех веток в репозитории.

```
git branch
#или
git branch ––list
```

Удаление ветки с названием *omega*. Это «безопасная» операция, так как git не позволит удалить ветку, в которой есть неслитые изменения.

```
git branch -d omega
```

Принудительное удаление указанной ветки, даже несмотря на то, что в ней есть неслитые изменения. Эту команда следует использовать, если вы хотите полностью удалить все коммиты, которые относятся к определенному направлению разработки.

```
git branch -D <branch>
```

Вывод списка всех удаленных веток.

```
git branch -a
```

Забрать изменения из удаленного репозитория

Команда **git pull** запускает команду **git fetch** для загрузки содержимого из указанного удаленного репозитория. Затем выполняется команда **git merge**, осуществляющая слияние ссылок и указателей удаленного содержимого в новый локальный коммит:

```
git pull <remote>
```

Выше указанная команда берет указанную удаленную копию текущей ветви и объединяет ее с локальной копией. Это то же самое, что и git fetch ＜remote＞, за которым следует git merge origin/＜current-branch＞.

Существует команда, подобная команде по умолчанию, только она не создает новый коммит со слитым содержимым:

```
git pull --no-commit <remote>
```

Загрузить изменения в удаленный репозиторий

Публикация указанной ветки в удаленном репозитории вместе со всеми необходимыми коммитами и внутренними объектами. Эта команда создает локальную ветку в репозитории назначения. Чтобы предотвратить перезапись коммитов, git не позволит опубликовать данные, если в репозитории назначения нельзя выполнить ускоренное слияние:

```
git push <remote> <branch>
```

Если для параметра указать значение *origin* и оставить пустым параметр *branch*, то изменения будут отправленны в ветку, которая выбрана в данный момент.

Аналогично приведенной выше команде, однако данные будут опубликованы принудительно, даже если нельзя выполнить ускоренное слияние. **Не используйте флаг --force**, если вы не уверены в своих действиях:

```
git push <remote> --force
```

Существует процедура для очистки локальной и удаленной веток, которую целесообразно выполнять для поддержания порядка и не допущения накопления изменений, которые не будут загружены в основную ветку проекта:

```
git branch -D alpha
git push origin :alpha
```

Первая команда очистит локальную ветку *alpha*. Если в команде *git push* перед именем ветки поставить двоеточие, будет стерта удаленная ветка.

Глобальные настройки Git

```
git config --global user.name "gitflic-user"
git config --global user.email "mail@gitflic.ru"
```

Создание нового репозитория

```
git clone http://gitflic.ru/project/{user}/{proekt}.git #Здесь ссылка на ваш проект
cd {proekt}
touch README.md
git add README.md
git commit -m "add README"
git push -u origin master
```

Использовать существующую директорию

```
cd existing_folder
git init
git remote add origin http://gitflic.ru/project/user/proekt.git //Здесь
```

```
ссылка на ваш проект
git add .
git commit -m "Initial commit"
git push -u origin master
```

Запушить существующий репозиторий

```
cd existing_folder
git remote rename origin old-origin
git remote add origin http://gitflic.ru/project/user/proekt.git //Здесь
ссылка на ваш проект
git push -u origin --all
git push -u origin --tags
```