Лаба на Git

Цель работы

Изучить возможности системы управления версиями и коллективной разработкой программного проекта;

Освоить методы работы с утилитой Git для управления версиями и коллективной разработкой программного проекта.

Теория

Система контроля версий Git

Git или Гит — система контроля и управления версиями файлов.

Система контроля версий — это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определённой версии. Как пример, для контроля версий файлов может использоваться исходный код программного обеспечения, но на самом деле вы можете использовать контроль версий практически для любых типов файлов.

GitHub или Гитхаб — веб-сервис для размещения репозиториев и совместной разработки проектов.

Основные термины в Git

Репозиторий Git – каталог файловой системы, в котором находятся: файлы конфигурации, файлы журналов операций, выполняемых над репозиторием, индекс расположения файлов и хранилище, содержащее сами контролируемые файлы. Есть 2 вида: локальный и удаленный.

Локальный репозиторий – репозиторий, расположенный на локальном компьютере разработчика в каталоге. Именно в нём происходит разработка и фиксация изменений, которые отправляются на удалённый репозиторий.

Удаленный репозиторий – репозиторий, находящийся на удалённом сервере. Это общий репозиторий, в который приходят все изменения и из которого забираются все обновления.

Коммит (Commit) – это фиксация изменений или запись изменений в репозиторий, содержащая комментарий к внесенным изменениям. Коммит происходит на локальной машине.

Ветка в Git (Branch) – это простой перемещаемый указатель на один из таких коммитов. По умолчанию, имя основной ветки в Git — **master** (мастер). Как только вы начнёте создавать коммиты, ветка master будет всегда указывать на последний коммит. Каждый раз при создании коммита указатель ветки master будет передвигаться на следующий коммит автоматически. Ветку можно использовать как параллельную версию репозитория. Она включена в репозиторий, но не влияет на главную версию, тем самым позволяя свободно работать в

параллельной. Когда вы внесли нужные изменения, то вы можете объединить их с главной версией.

Форк (Fork) — копия репозитория. Его также можно рассматривать как внешнюю ветку для текущего репозитория.

Обновиться из апстрима — обновить свою локальную версию форка до последней версии основного репозитория, от которого сделан форк.

Обновиться из ориджина — обновить свою локальную версию репозитория до последней удалённой версии этого репозитория.

Клонирование (Clone) — скачивание репозитория с удалённого сервера на локальный компьютер в определённый каталог для дальнейшей работы с этим каталогом как с репозиторием.

Пул (Pull) – получение последних изменений с удалённого сервера репозитория.

Пуш (Push) – добавление данных в вашу ветку и отправка их в удаленный репозиторий.

Пулреквест (Pull Request) – запрос на слияние форка репозитория с основным репозиторием. Пулреквест может быть принят или отклонён вами, как владельцем репозитория.

Мёрдж (Merge) — слияние изменений из какой-либо ветки репозитория с любой веткой этого же репозитория. Чаще всего слияние изменений из ветки репозитория с основной веткой репозитория.

Кодревью — процесс проверки кода на соответствие определённым требованиям, задачам и внешнему виду.

Работа с утилитой git

Управлять версиями можно, используя командную строку (терминал). Также можно работать с управлением версиями с помощью графических интерфейсов. Самый популярный и часто используемый – Git GUI. Также можно использовать графический интерфейс TortoiseGit.

Git работает локально и все репозитории хранятся в определенных папках на жестком диске. Но также репозитории можно хранить и в интернете. Обычно для этого используют три сервиса:

- GitHub
- Bitbucket (с недавнего времени закрыт у нас в стране, но vpn никто не отменял)
- GitLab

Перечень основных команд

Рассмотрим базовые команды — команды, без которых невозможно обойтись в разработке.

- git config

Одна из самых часто используемых git команд. Она может быть использована для

указания пользовательских настроек, таких как электронная почта, имя пользователя, формат и т.д. К примеру, данная команда используется для установки адреса электронной почты:

```
git config --global user.email адрес@gmail.com
```

- git init

Эта команда используется для создания GIT репозитория. Пример использования:

```
git init
```

- git add

Команда **git add** может быть использована для добавления файлов в индекс. К примеру, следующая команда добавит файл под названием temp.txt присутствующий в локальном каталоге в индекс:

```
git add temp.txt
```

- git clone

Команда **git clone** используется для клонирования репозитория. Если репозиторий находится на удаленном сервере, используется команда такого рода:

```
git clone имя.пользователя@хост:/путь/до/репозитория
```

И наоборот, для клонирования локального репозитория используйте:

```
git clone /путь/до/репозитория
```

- git commit

Команда **git commit** используется для коммита изменений в файлах проекта. Обратите внимание, что коммиты не сразу попадают на удаленный репозиторий. Применение:

```
git commit –m “Сообщение идущее вместе с коммитом”
```

- git status

Команда **git status** отображает список измененных файлов, вместе с файлами, которые еще не были добавлены в индекс или ожидают коммита. Применение:

```
git status
```

- git push

git push еще одна из часто используемых git команд. Позволяет поместить изменения в главную ветку удаленного хранилища связанного с рабочим каталогом. Например:

```
git push origin master
```

Команда, обновляющая удаленную ветку с обновлением всей истории, не смотря на расхождения.

```
git push -f
```

Команда, которая обновляет удаленные ссылки с помощью локальных ссылок, одновременно отправляя объекты, необходимые для выполнения заданных ссылок с созданием удаленной ветки.

```
git push –set-upstream <remote><branch>
```

- git checkout

Команда **git checkout** может быть использована для создания веток или переключения между ними. К примеру, следующий код создаст новую ветку и переключится на нее:

```
command git checkout -b<имя-ветки>
```

Чтобы просто переключиться между ветками используйте:

```
git checkout <имя-ветки>
```

- git remote

Команда позволяет пользователю подключиться к удаленному репозиторию. Данная команда отобразит список удаленных репозиториев, настроенных в данный момент:

```
git remote –v
```

Эта команда позволит пользователю подключить локальный репозиторий к удаленному серверу:

```
git remote add origin <адрес.удаленного.сервера>
```

- git branch

Команда **git branch** может быть использована для отображения, создания или удаления веток. Для отображения всех существующих веток в репозитории введите:

```
git branch
```

Для удаления ветки:

```
git branch –d <имя-ветки>
```

- git pull

Команда pull используется для объединения изменений, присутствующих в удаленном репозитории, в локальный рабочий каталог. Применение:

```
git pull
```

- git merge

Команда **git merge** используется для объединения ветки в активную ветвь.

Применение:

```
git merge <имя-ветки>
```

- git diff

Команда **git diff** используется для выявления различий между ветками. Для выявления различий с базовыми файлами, используйте

```
git diff --base <имя-файла>
```

Следующая команда используется для просмотра различий между ветками, которые должны быть объединены, до их объединения:

```
git diff <ветвь-источник> <ветвь-цель>
```

Для простого отображения существующих различий, используйте:

```
git diff
```

- git tag

Используется для маркировки определенных коммитов с помощью простых меток. Примером может быть эта команда:

```
git tag 1.1.0 <вставьте-commitID-здесь>
```

- git log

Запуск команды **git log** отобразит список всех коммитов в ветке вместе с соответствующими сведениями. Пример результата:

```
commit 15f4b6c44b3c8344caasdac9e4be13246e21sadw
Author: Alex Hunter <alexh@gmail.com>

Date: Mon Oct 1 12:56:29 2016 -0600
```

- git reset

Команда **git reset** используется для сброса индекса и рабочего каталога до последнего состояния коммита. Применение:

```
git reset --hard HEAD
```

- git rm

git rm используется для удаления файлов из индекса и рабочего каталога. Применение:

```
git rm имяфайла.txt
```

- git stash

Возможно одна из самых малоизвестных команд git. Она помогает в сохранении изменений на временной основе, эти изменения не попадут в коммит сразу. Применение:

```
git stash
```

- git show

Для просмотра информации о любом git объекте используйте команду **git show**. Для примера:

```
git show
```

- git fetch

git fetch позволяет пользователю доставить все объекты из удаленного репозитория, которые не присутствуют в локальном рабочем каталоге. Пример применения:

```
git fetch origin
```

- git ls-tree

Команда **git ls-tree** используется для просмотра дерева объекта вместе с названием, режимом каждого предмета и значением **SHA-1**. К примеру:

```
git ls-tree HEAD
```

- git cat-file

Используйте команду **git cat-file**, чтобы просмотреть тип объекта с помощью **SHA-1** значения. Например:

```
git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

- git grep

git grep позволяет пользователю проводить поиск фраз и слов в содержимом деревьев. К примеру, для поиска *www.hostinger.ru* во всех файлах используйте эту команду:

```
git grep «www.hostinger.ru"
```

- gitk

gitk — это графический интерфейс локального репозитория**.** Вызвать его можно выполнив данную команду:

```
gitk
```

- git instaweb

С помощью команды **git instaweb** можно запустить веб-сервер, связанный с локальным репозиторием. Браузер также автоматически будет перенаправляться на него. Например:

```
git instaweb –httpd=webrick
```

- git gc

Для оптимизации репозитория используйте команду **git gc**. Она поможет удалить и оптимизировать ненужные файлы:

```
git gc
```

- git archive

Команда **git archive** позволяет пользователю создать .zip или .tar файл содержащий компоненты одного из деревьев репозитория. Например:

```
git archive --format=tar master
```

- git prune

С помощью команды **git prune** удаляются объекты, не имеющие никаких входящих указателей. Применение:

```
git prune
```

- git fsck

Чтобы выполнить проверку целостности файловой системы git, используйте команду **git fsck**, при этом будут идентифицированы все поврежденные объекты:

```
git fsck
```

- git rebase

Команда **git rebase** используется для применения коммитов в другой ветке. Например:

```
git rebase master
```

Продолжить операции на следующий commit.

```
git rebase –continue
```

Остановить операцию с возвращением

```
git rebase –abort
```

Работа с репозиторием в терминале

Создание репозитория, коммитов и веток

Для создания нового репозитория достаточно просто зайти в папку проекта и набрать:

```
git init
```

Был создан пустой репозиторий — папка .git в корне проекта, в которой и будет собираться вся информация о дальнейшей работе. Предположим, уже существует несколько файлов, и их требуется проиндексировать командой git add:

```
git add .
```

Внесем изменения в репозиторий:

```
git commit -m "Первоначальный коммит"
```

Имеется готовый репозиторий с единственной веткой. Допустим, потребовалось разработать какой-то новый функционал. Для этого создадим новую ветку:

```
git branch new-feature
```

И переключимся на нее:

```
git checkout new-feature
```

Вносим необходимые изменения, после чего смотрим на них, индексируем и коммитимся:

```
git status git
add .

git commit -m "new feature added"
```

Теперь у нас есть две ветки, одна из которых (master) является условно (технически же ничем не отличается) основной. Переключаемся на нее и включаем изменения (сливаем с другой веткой):

```
git checkout master

git merge new-feature
```

Веток может быть неограниченное количество, из них можно создавать патчи, определять diff с любым из совершенных коммитов.

Теперь предположим, что во время работы выясняется: нашелся небольшой баг, требующий срочного внимания. Есть два варианта действий в таком случае. Первый состоит из создания новой ветки, переключения в нее, слияния с основой… Второй — команда git stash. Она сохраняет все изменения по сравнению с последним коммитом во временной ветке и сбрасывает состояние кода до исходного:

```
git stash
```

Исправляем баг и накладываем поверх произведенные до того действия (проводим слияние с веткой stash):

```
git stash apply
```

На самом деле таких «заначек» (stash) может быть сколько угодно; они просто нумеруются.

При такой работе появляется необычная гибкость; но среди всех этих веточек теряется понятие ревизии, характерное для линейных моделей разработки. Вместо этого каждый из коммитов (строго говоря, каждый из объектов в репозитории) однозначно определяется

хэшем. Естественно, это несколько неудобно для восприятия, поэтому разумно использовать механизм тэгов для того, чтобы выделять ключевые коммиты:

```
git tag
```

просто именует последний коммит;

```
git tag -a
```

также дает имя коммиту, и добавляет возможность оставить какие-либо комментарии (аннотацию). По этим тегам можно будет в дальнейшем обращаться к истории разработки.

Плюсы такой системы очевидны. Вы получаете возможность колдовать с кодом как душе угодно, а не как диктует система контроля версий: разрабатывать параллельно несколько «фишек» в собственных ветках, исправлять баги, чтобы затем все это сливать в единую кашу главной ветки. Удобно быстро создаются, удаляются или копируются куда угодно папки .git с репозиторием.

Гораздо удобней такую легковесную систему использовать для хранения версий документов, файлов настроек и т. д. К примеру, настройки и плагины для Емакса можно хранить в директории ~/site-lisp, и держать в том же месте репозиторий. Рабочее пространство можно организовать в виде двух веток: work и home; иногда бывает удобно похожим образом управлять настройками в /etc. Таким образом, каждый из личных проектов может находиться под управлением git.

Файл конфигурации .gitignore

Иногда по директориям проекта встречаются файлы, которые не хочется постоянно видеть в сводке git status. Например, вспомогательные файлы текстовых редакторов, временные файлы и прочий мусор.

Заставить git status игнорировать можно, создав в корне или глубже по дереву (если ограничения должны быть только в определенных директория) файл .gitignore [5.]. В этих файлах можно описывать шаблоны игнорируемых файлов определенного формата.

Пример содержимого такого файла:

```
>>>>>>>Начало файла
#комментарийкфайлу.gitignore
#игнорируемсам.gitignore
.gitignore
#все html-файлы…
*.html
#… кроме определенного
!special.html
#не нужны объектники и архивы
*.[ao]
>>>>>>>>Конец файла
```

Существуют и другие способы указания игнорируемых файлов, о которых можно узнать из справки:

```
git help gitignore.
```

Работа с github через терминал

Общественный (удаленный) репозиторий [6.] — способ обмениваться кодом в проектах, где участвует больше двух человек. Для этого можно использовать сайт github.com, из-за его удобства, многие начинают пользоваться git.

Итак, создаем у себя копию удаленного репозитория:

```
gitclonegit://github.com/username/project.gitmaster
```

Команда создала у вас репозиторий, и внесла туда копию ветки master проекта project. Теперь можно начинать работу. Создадим новую ветку, внесем в нее изменения, закоммитимся:

```
git branch new-feature
edit README
git add .
git commit -m "Added a super feature"
```

Перейдем в основную ветку, заберем последние изменения в проекте, и попробуем добавить новую фишку в проект:

```
git checkout master
git pull
git mergenew-feature
```

Если не было неразрешенных конфликтов, то коммит слияния готов.

Команда git pull использует так называемую удаленную ветку (remote branch), создаваемую при клонировании удаленного репозитория. Из нее она извлекает последние изменения и проводит слияние с активной веткой.

Теперь остается только занести изменения в центральный (условно) репозиторий:

```
git push
```

Нельзя не оценить всю гибкость, предоставляемую таким средством. Можно вести несколько веток, отсылать только определенную, жонглировать коммитами как угодно.

В принципе, никто не мешает разработать альтернативную модель разработки. Например, использовать иерархическую систему репозиториев, когда «младшие» разработчики делают коммиты в промежуточные репозитории, где те проходят проверку у

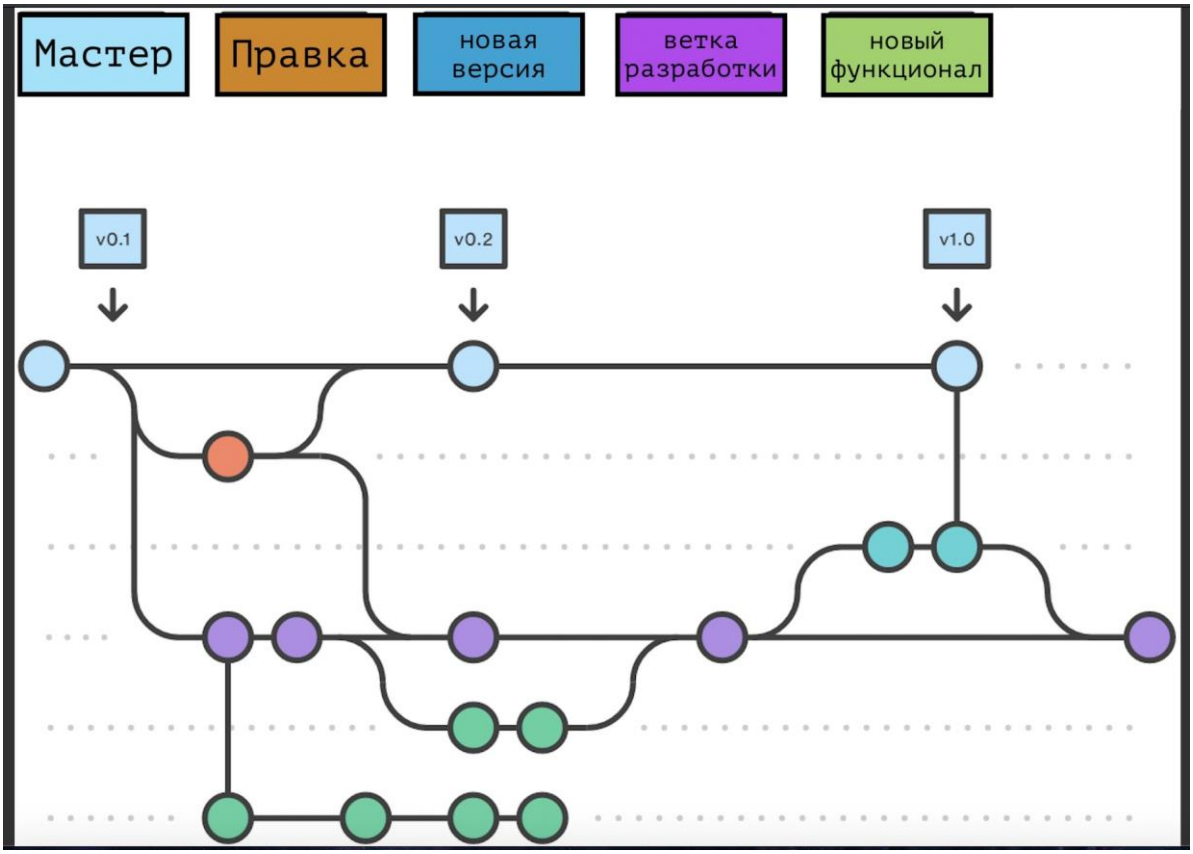
«старших» программистов и только потом попадают в главную ветку центрального репозитория проекта.

При работе в парах возможно использовать симметричную схему работы. Каждый разработчик ведет по два репозитория: рабочий и общественный. Первый используется в работе непосредственно, второй же, доступный извне, только для обмена уже законченным кодом.

Последовательность работы с репозиторием

Пример последовательности работы с Git в команде разработки приведен на рисунок.

Рисунок. Последовательность работы с Git в команде разработки.



В этом разделе будут показаны и разобраны подробно несколько обычных и чуть меньше необычных для работы с git ситуаций.

Действия при работе с локальным репозиторием

Git обладает необычайной легкостью в использовании не только как распределенная система контроля версий, но и в работе с локальными проектами. Разберем обычный цикл [2.] — начиная с создания репозитория — работы разработчика git над собственным персональным проектом:

1. Создаем рабочую директорию проекта

```
mkdir git-demo
```

2. Переходим в рабочую директорию

```
cd git-demo
```

3. Создаем репозиторий в директории

```
git init
```

4. Индексируем все существующие файлы проекта (добавляем в репозиторий)

```
git add.
```

5. Создаем инициализирующий коммит

```
git commit -m «initial commit»
```

6. Создаем новую ветку

```
git branch new-feature
```

7. Переключаемся в новую ветку

```
git checkout new-feature
```

Пункты 6-7 можно сделать в один шаг командой:

```
git checkout -b new-feature
```

8. После непосредственной работы с кодом индексируем внесенные изменения

```
git add.
```

9. Совершаем коммит

```
git commit -m «Done with the new feature»
```

10. Переключаемся в основную ветку

```
git checkout master
```

11. Смотрим отличия между последним коммитом активной ветки и последним коммитом экспериментальной

```
git diff HEAD new-feature
```

12. Проводим слияние

```
git merge new-feature
```

13. Если не было никаких конфликтов, удаляем ненужную больше ветку

```
git branch -d new-feature
```

14. Оценим проведенную за последний день работу

```
git log --since=«1 day»
```

Преимущества отказа от линейной модели:

- у программиста появляется дополнительная гибкость: он может переключаться между задачами (ветками);
- под рукой всегда остается «чистовик» — ветка master;
- коммиты становятся мельче и точнее.

Действия при работе с удаленным репозиторием

Предположим, что вы и несколько ваших напарников создали общественный репозиторий, чтобы заняться неким общим проектом. Рассмотрим, как выглядит самая распространенная для git модель общей работы [2.]:

1. Создаем копию удаленного репозитория (по умолчанию команды вроде *git pull* и *git push* будут работать с ним)

```
git clone http://yourserver.com/~you/proj.git
```

Возможно, будет необходимо подождать некоторое время.

Далее выполняем итерационно:

2. «Вытягиваем» последние обновления

```
git pull
```

3. Смотрим, что изменилось

```
git diff HEAD^
```

4. Создаем новую ветвь и переключаемся в нее

```
git checkout -b bad-feature
```

Процесс выполняется некоторое время.

5. Индексируем все изменения и одновременно создаем из них коммит

```
git commit -a -m «Created a bad feature»
```

6. Переключаемся в главную ветвь

```
git checkout master
```

7. Обновляем ее

```
git pull
```

8. Проводим слияние с веткой bad-feature

```
git merge bad-feature
```

9. Обнаружив и разрешив конфликт, делаем коммит слияния

```
git commit -a
```

10. После совершения коммита отслеживаем изменения

```
git diff HEAD^
```

Запускаем тесты проекта, обнаруживаем, что где-то произошла ошибка.

В принципе, тесты можно было прогнать и до коммита, в момент слияния (между пунктами 8 и 9); тогда бы хватило «мягкого» резета.

11. Приходится совершить «жесткий» сброс произошедшего слияния, ветки вернулись в исходное до состояние

```
git reset --hard ORIG_HEAD
```

12. Переключаемся в неудачную ветку

```
git checkout bad-feature
```

Исправляем ошибку.

13. Вносим необходимые изменения и переименовываем ветку

```
git -m bad-feature good-feature
```

14. Совершаем коммит

```
git commit -a -m «Better feature»
```

15. Переходим в главную ветку

```
git checkout master
```

16. Опять ее обновляем

```
git pull
```

17. На этот раз бесконфликтно делаем слияние

```
git merge good-feature
```

18. Закидываем изменения в удаленный репозиторий

```
git push
```

19. Удаляем ненужную теперь ветку

```
git branch -d good-feature
```

Пункты задания для выполнения

- 1. Запустить консоль git. Создать новый репозиторий (в папке по фамилии студента).
- 2. Добавить в папку репозитория файлы. Зафиксировать состояние репозитория (выполнить commit).
- 3. Внести изменения в файлы. Зафиксировать новое состояние репозитория.
- 4. Создать новую ветку 1. Внести в нее изменения (добавить новый файл и изменить существующий файл: добавить, удалить и изменить строки) и зафиксировать их.
- 5. Переключиться на ветку мастера. Внести в нее изменения (добавить новый файл; изменить существующие файлы: добавить, удалить и изменить строки первоначального файла) и зафиксировать их.
- 6. Продемонстрировать слияние веток. Разрешить возникший конфликт.
- 7. Просмотреть дерево изменений веток (историю).
- 8. Продемонстрировать откат изменений в ветке 1.
- 9. Создать удаленный репозиторий (на github.com).
- 10. Отправить данные на удаленный репозиторий (выполняется одним из студентов группы). Добавить к удаленному репозиторию участников проекта.
- 11. Получить данные из удаленного репозитория (выполняется прочими студентами).
- 12. Изменить полученные данные.
- 13. Зафиксировать изменения и отправить их на удаленный репозиторий (выполняется всеми студентами группы).
- 14. Получить данные из удаленного репозитория.
- 15. Просмотреть историю изменений.

- 15. Продемонстрировать работу revert и reset;
- 16. Продемонстрировать сохранение изменений в stash с последующим восстановлением; создание и применение серии патчей;
- 17. Продемонстрировать создание и применение тегов;
  Продемонстрировать работу rebase.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое система управления версиями?
2. Как создать репозиторий?
3. Как создать ветку?
4. Как провести слияние? Как разрешить конфликт и что это такое?
5. Как зафиксировать изменения?
6. Как провести откат? Различия в reset и revert, мягкий и жесткий reset.
7. Какова последовательность действий при работе с локальным репозиторием?
8. Какова последовательность действий при работе с удаленным репозиторием?
9. Каковы возможности при работе с удаленным репозиторием? Как его клонировать, получать и отправлять данные?

Содержание отчета

- Титульный лист;
- Цель работы;
- Задание;
- Тексты запросов и команд в соответствии с пунктами задания.
- Результаты выполнения запросов (скриншоты);
- Ответы на контрольные вопросы
- Вывод;