# Практическая работа

# Изучение работы в системе контроля версий

Целью работы является изучение порядка работы с системой контроля версий GIT. Результатом практической работы является отчет, в котором должны быть приведено описание созданного ре- позитария, демонстрация приемов работы с ним.

Для выполнения практической работы № 4 студент должен изучить приведенный ниже теоретический материал. Отчет сдается в распечатанном и электронном (файл Word) видах.

# Изучение работы в системе контроля версий

* Основы

Git – это набор консольных утилит, которые отслеживают и фиксируют изменения в файлах (чаще всего речь идет об исходном коде программ, но вы можете использовать его для любых файлов на ваш вкус). С его помощью вы можете откатиться на более ста- рую версию вашего проекта, сравнивать, анализировать, сливать изменения и многое другое. Этот процесс называется контролем версий. Существуют различные системы для контроля версий. Вы, возможно, о них слышали: SVN, Mercurial, Perforce, CVS, Bitkeeper и другие.

Git является распределенным, то есть не зависит от одного центрального сервера, на котором хранятся файлы. Вместо этого он работает полностью локально, сохраняя данные в папках на жест- ком диске, которые называются репозиторием. Тем не менее, вы можете хранить копию репозитория онлайн, это сильно облегчает работу над одним проектом для нескольких людей. Для этого ис- пользуются сайты вроде github и bitbucket.

* Установка

Установить git на свою машину очень просто:

Windows – мы рекомендуем git for windows, так как он содер- жит и клиент с графическим интерфейсом, и эмулятор bash.

* Настройка

Итак, мы установили git, теперь нужно добавить немного на- строек. Есть довольно много опций, с которыми можно играть, но

мы настроим самые важные: наше имя пользователя и адрес элек- тронной почты. Откройте терминал и запустите команды:

git config --global user.name «My Name»

git config --global user.email [myEmail@example.com](mailto:myEmail@example.com)

Теперь каждое наше действие будет отмечено именем и по- чтой. Таким образом, пользователи всегда будут в курсе, кто отве- чает за какие изменения – это вносит порядок.

* Создание нового репозитория

Как мы отметили ранее, git хранит свои файлы и историю пря- мо в папке проекта. Чтобы создать новый репозиторий, нам нужно открыть терминал, зайти в папку нашего проекта и выполнить ко- манду init. Это включит приложение в этой конкретной папке и соз- даст скрытую директорию .git, где будет храниться история репози- тория и настройки. Создайте на рабочем столе папку под названием git\_exercise. Для этого в окне терминала введите:

$ mkdir Desktop/git\_exercise/

$ cd Desktop/git\_exercise/

$ git init

Командная строка должна вернуть что-то вроде:

Initialized empty Git repository in /home/user/Desktop/git\_exercise/.git/

Это значит, что наш репозиторий был успешно создан, но пока что пуст. Теперь создайте текстовый файл под названием hello.txt и сохраните его в директории git\_exercise.

* Определение состояния

status – это еще одна важнейшая команда, которая показывает информацию о текущем состоянии репозитория: актуальна ли ин- формация на нѐм, нет ли чего-то нового, что поменялось, и так да- лее. Запуск git status на нашем свежесозданном репозитории должен выдать:

$ git status

On branch master Initial commit Untracked files:

(use «git add ...» to include in what will be committed) hello.txt

Сообщение говорит о том, что файл hello.txt неотслеживаемый. Это значит, что файл новый и система еще не знает, нужно ли сле- дить за изменениями в файле или его можно просто игнорировать. Для того, чтобы начать отслеживать новый файл, нужно его специ- альным образом объявить.

* Подготовка файлов

В git есть концепция области подготовленных файлов. Можно представить ее как холст, на который наносят изменения, которые нужны в коммите. Сперва он пустой, но затем мы добавляем на него файлы (или части файлов, или даже одиночные строчки) командой add и, наконец, коммитим все нужное в репозиторий (создаем сле- пок нужного нам состояния) командой commit. В нашем случае у нас только один файл, так что добавим его:

$ git add hello.txt

Если нам нужно добавить все, что находится в директории, мы можем использовать

$ git add -A

Проверим статус снова, на этот раз мы должны получить дру- гой ответ:

$ git status

On branch master Initial commit

Changes to be committed:

(use «git rm --cached ...» to unstage) new file: hello.txt

Файл готов к коммиту. Сообщение о состоянии также говорит нам о том, какие изменения относительно файла были проведены в области подготовки – в данном случае это новый файл, но файлы могут быть модифицированы или удалены.

* Коммит (фиксация изменений)

Коммит представляет собой состояние репозитория в опреде- ленный момент времени. Это похоже на снапшот, к которому мы можем вернуться и увидеть состояние объектов на определенный момент времени. Чтобы зафиксировать изменения, нам нужно хотя бы одно изменение в области подготовки (мы только что создали его при помощи git add), после которого мы может коммитить:

$ git commit -m «Initial commit.»

Эта команда создаст новый коммит со всеми изменениями из области подготовки (добавление файла hello.txt). Ключ -m и сооб- щение «Initial commit.» – это созданное пользователем описание всех изменений, включенных в коммит. Считается хорошей практи- кой делать коммиты часто и всегда писать содержательные коммен- тарии.

* Удаленные репозитории

Сейчас наш коммит является локальным – существует только в директории .git на нашей файловой системе. Несмотря на то, что сам по себе локальный репозиторий полезен, в большинстве случа- ев мы хотим поделиться нашей работой или доставить код на сер- вер, где он будет выполняться.

* 1. Подключение к удаленному репозиторию

Чтобы загрузить что-нибудь в удаленный репозиторий, снача- ла нужно к нему подключиться. В нашем руководстве мы будем ис- пользовать адрес https://github.com/tutorialzine/awesome-project, но вам посоветуем попробовать создать свой репозиторий в GitHub, BitBucket или любом другом сервисе. Регистрация и установка мо- жет занять время, но все подобные сервисы предоставляют хоро- шую документацию. Чтобы связать наш локальный репозиторий с репозиторием на GitHub, выполним следующую команду в терми- нале. Обратите внимание, что нужно обязательно изменить URI ре- позитория на свой.

# This is only an example. Replace the URI with your own repository ad-

dress.

$ git remote add origin https://github.com/tutorialzine/awesome-

project.git

Проект может иметь несколько удаленных репозиториев одно- временно. Чтобы их различать, мы дадим им разные имена. Обычно главный репозиторий называется origin.

* 2. Отправка изменений на сервер

Сейчас самое время переслать наш локальный коммит на сер- вер. Этот процесс происходит каждый раз, когда мы хотим обно- вить данные в удаленном репозитории. Команда, предназначенная для этого – push. Она принимает два параметра: имя удаленного ре- позитория (мы назвали наш origin) и ветку, в которую необходимо

внести изменения (master – это ветка по умолчанию для всех репо- зиториев).

$ git push origin master Counting objects: 3, done.

Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done. Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/tutorialzine/awesome-project.git

\* [new branch] master -> master

В зависимости от сервиса, который вы используете, вам может потребоваться аутентифицироваться, чтобы изменения отправи- лись. Если все сделано правильно, то когда вы посмотрите в уда- ленный репозиторий при помощи браузера, вы увидете файл hello.txt

* 3. Клонирование репозитория

Сейчас другие пользователи GitHub могут просматривать ваш репозиторий. Они могут скачать из него данные и получить полно- стью работоспособную копию вашего проекта при помощи коман- ды clone.

$ git clone https://github.com/tutorialzine/awesome-project.git

Новый локальный репозиторий создается автоматически с GitHub в качестве удаленного репозитория.

* 4. Запрос изменений с сервера

Если вы сделали изменения в вашем репозитории, другие пользователи могут скачать изменения при помощи команды pull.

$ git pull origin master

From https://github.com/tutorialzine/awesome-project

\* branch master -> FETCH\_HEAD Already up-to-date.

Так как новых коммитов с тех пор, как мы склонировали себе проект, не было, никаких изменений доступных для скачивания нет.

* Ветвление

Во время разработки новой функциональности считается хо- рошей практикой работать с копией оригинального проекта, кото- рую называют веткой. Ветви имеют свою собственную историю и изолированные друг от друга изменения до тех пор, пока вы не ре- шаете слить изменения вместе. Это происходит по набору причин:

* Уже рабочая, стабильная версия кода сохраняется.
* Различные новые функции могут разрабатываться парал- лельно разными программистами.
* Разработчики могут работать с собственными ветками без риска, что кодовая база поменяется из-за чужих изменений.
* В случае сомнений, различные реализации одной и той же идеи могут быть разработаны в разных ветках и затем сравниваться.
* 1. Создание новой ветки

Основная ветка в каждом репозитории называется master. Что- бы создать еще одну ветку, используем команду branch <name>

$ git branch amazing\_new\_feature

Это создаст новую ветку, пока что точную копию ветки master.

* 2. Переключение между ветками

Сейчас, если мы запустим branch, мы увидим две доступные опции:

$ git branch amazing\_new\_feature

\* master

master – это активная ветка, она помечена звездочкой. Но мы хотим работать с нашей «новой потрясающей фичей», так что нам понадобится переключиться на другую ветку. Для этого воспользу- емся командой checkout, она принимает один параметр – имя ветки, на которую необходимо переключиться.

$ git checkout amazing\_new\_feature

* 3. Слияние веток

Наша «потрясающая новая фича» будет еще одним текстовым файлом под названием feature.txt. Мы создадим его, добавим и за- коммитим:

$ git add feature.txt

$ git commit -m «New feature complete.»

Изменения завершены, теперь мы можем переключиться об- ратно на ветку master.

$ git checkout master

Теперь, если мы откроем наш проект в файловом менеджере, мы не увидим файла feature.txt, потому что мы переключились об- ратно на ветку master, в которой такого файла не существует. Чтобы

он появился, нужно воспользоваться merge для объединения веток (применения изменений из ветки amazing\_new\_feature к основной версии проекта).

$ git merge amazing\_new\_feature

Теперь ветка master актуальна. Ветка amazing\_new\_feature больше не нужна, и ее можно удалить.

$ git branch -d awesome\_new\_feature

* Дополнительно

В последней части этого руководства мы расскажем о некото- рых дополнительных трюках, которые могут вам помочь.

* 1. Отслеживание изменений, сделанных в коммитах

У каждого коммита есть свой уникальный идентификатор в виде строки цифр и букв. Чтобы просмотреть список всех коммитов и их идентификаторов, можно использовать команду log:

# Вывод git log

Как вы можете заметить, идентификаторы довольно длинные, но для работы с ними не обязательно копировать их целиком – пер- вых нескольких символов будет вполне достаточно. Чтобы посмот- реть, что нового появилось в коммите, мы можем воспользоваться командой show [commit]

# Вывод git show

Чтобы увидеть разницу между двумя коммитами, используется команда diff (с указанием промежутка между коммитами):

# Вывод git diff

Мы сравнили первый коммит с последним, чтобы увидеть все изменения, которые были когда-либо сделаны. Обычно проще ис- пользовать git difftool, так как эта команда запускает графический клиент, в котором наглядно сопоставляет все изменения.

* 2. Возвращение файла к предыдущему состоянию

Гит позволяет вернуть выбранный файл к состоянию на мо- мент определенного коммита. Это делается уже знакомой нам ко- мандой checkout, которую мы ранее использовали для переключе- ния между ветками. Но она также может быть использована для пе- реключения между коммитами (это довольно распространенная си- туация для Гита – использование одной команды для различных, на первый взгляд, слабо связанных задач). В следующем примере мы возьмем файл hello.txt и откатим все изменения, совершенные над

ним к первому коммиту. Чтобы сделать это, мы подставим в коман- ду идентификатор нужного коммита, а также путь до файла:

$ git checkout 09bd8cc1 hello.txt

* 3. Исправление коммита

Если вы опечатались в комментарии или забыли добавить файл и заметили это сразу после того, как закоммитили изменения, вы легко можете это поправить при помощи commit –amend. Эта команда добавит все из последнего коммита в область подготовлен- ных файлов и попытается сделать новый коммит. Это дает вам воз- можность поправить комментарий или добавить недостающие фай- лы в область подготовленных файлов. Для более сложных исправ- лений, например, не в последнем коммите или если вы успели от- править изменения на сервер, нужно использовать revert. Эта ко- манда создаст коммит, отменяющий изменения, совершенные в коммите с заданным идентификатором. Самый последний коммит может быть доступен по алиасу HEAD:

$ git revert HEAD

Для остальных будем использовать идентификаторы:

$ git revert b10cc123

При отмене старых коммитов нужно быть готовым к тому, что возникнут конфликты. Такое случается, если файл был изменен еще одним, более новым коммитом. И теперь git не может найти строч- ки, состояние которых нужно откатить, так как они больше не су- ществуют.

* 4. Разрешение конфликтов при слиянии

Помимо сценария, описанного в предыдущем пункте, кон- фликты регулярно возникают при слиянии ветвей или при отправке чужого кода. Иногда конфликты исправляются автоматически, но обычно с этим приходится разбираться вручную – решать, какой код остается, а какой нужно удалить. Давайте посмотрим на приме- ры, где мы попытаемся слить две ветки под названием john\_branch и tim\_branch. И Тим, и Джон правят один и тот же файл: функцию, которая отображает элементы массива. Джон использует цикл:

// Use a for loop to console.log contents. for(var i=0; i<arr.length; i++) { console.log(arr[i]);

}

Тим предпочитает forEach:

// Use forEach to console.log contents. arr.forEach(function(item) { console.log(item);

});

Они оба коммитят свой код в соответствующую ветку. Теперь, если они попытаются слить две ветки, они получат сообщение об ошибке:

$ git merge tim\_branch Auto-merging print\_array.js

CONFLICT (content): Merge conflict in print\_array.js

Automatic merge failed; fix conflicts and then commit the result.

Система не смогла разрешить конфликт автоматически, зна- чит, это придется сделать разработчикам. Приложение отметило строки, содержащие конфликт:

# Вывод

Над разделителем ======= мы видим последний (HEAD) коммит, а под ним – конфликтующий. Таким образом, мы можем увидеть, чем они отличаются и решать, какая версия лучше. Или вовсе написать новую. В этой ситуации мы так и поступим, пере- пишем все, удалив разделители, и дадим git понять, что закончили.

// Not using for loop or forEach.

// Use Array.toString() to console.log contents. console.log(arr.toString());

Когда все готово, нужно закоммитить изменения, чтобы за- кончить процесс:

$ git add -A

$ git commit -m «Array printing conflict resolved.»

Как вы можете заметить, процесс довольно утомительный и может быть очень сложным в больших проектах. Многие разработ- чики предпочитают использовать для разрешения конфликтов кли- енты с графическим интерфейсом. (Для запуска нужно набрать git mergetool).

* 5. Настройка .gitignore

В большинстве проектов есть файлы или целые директории, в которые мы не хотим (и, скорее всего, не захотим) коммитить. Мы можем удостовериться, что они случайно не попадут в git add -A при помощи файла .gitignore.

1. Создайте вручную файл под названием .gitignore и сохрани- те его в директорию проекта.
2. Внутри файла перечислите названия файлов/папок, которые нужно игнорировать, каждый с новой строки.
3. Файл .gitignore должен быть добавлен, закоммичен и от- правлен на сервер, как любой другой файл в проекте.

Вот хорошие примеры файлов, которые нужно игнорировать:

* Логи
* Артефакты систем сборки
* Папки node\_modules в проектах node.js
* Папки, созданные IDE, например, Netbeans или IntelliJ
* Разнообразные заметки разработчика.

Файл .gitignore, исключающий все перечисленное выше, будет выглядеть так:

\*.log build/

node\_modules/

.idea/ my\_notes.txt

Символ слэша в конце некоторых линий означает директорию (и тот факт, что мы рекурсивно игнорируем все ее содержимое). Звездочка, как обычно, означает шаблон.

# Контрольные вопросы

1. Приведите основные команды git
2. Как создать новую ветку в git?
3. Как переключиться в существующую ветку?
4. Как отправить изменения на сервер?