Совместная работа над проектом, основные понятия и команды

План урока

Аннотация

Мы продолжаем изучать работу с системой Git. На этом занятии мы разберём принципы работы с сетевыми репозиториями и остановимся на командной работе с системами контроля версий.

1. Командная работа в Git

На прошлом уроке мы познакомились с возможностями Git для самостоятельной работы над проектом на локальном компьютере:

— научились создавать репозиторий;

— создавать и объединять ветки;

— делать коммиты;

— и даже разрешать конфликты при объединении изменений.

Сейчас мы изучим применение Git в командной работе:

— научимся подключать удалённые репозитории;

— клонировать проект;

— отправлять изменения в репозиторий и получать их оттуда.

Обязательно разберём основные правила совместной работы с репозиторием.

Для начала давайте определимся с терминологией: **локальный репозиторий** — это тот репозиторий, который размещён на конкретной машине разработчика. **Удалённый репозиторий** (он же сетевой репозиторий) — это репозиторий, расположенный на удалённом сервере, в который вносят изменение все разработчики проекта.

В произвольный момент времени состав веток и коммитов во всех репозиториях может различаться, но в некоторые оговорённые моменты времени (как правило, перед началом работы над задачей, после окончания работы над задачей и в конце дня) разработчики синхронизируют свои локальные репозитории с удалённым.

Для этого они выполняют два действия:

1. Скачивают изменения с удалённого репозитория (pull).

2. Отправляют туда свои (push).

Перед отправкой своих изменений разработчик должен объединить (часто говорят «смержить» от английского слова *merge*) их с изменениями, подтянутыми с сервера. Этот процесс похож на объединение веток, которое мы изучали на предыдущем уроке.

Чаще всего для ведения сетевых репозиториев используют сервисы **GitHub** или **Bitbucket**. На этом уроке мы будем использовать **GitHub**, потому что он распространён среди разработчиков ПО с открытым кодом, и вы наверняка с ним неоднократно столкнётесь. Кроме того, хорошо выглядящий профиль на **GitHub** с большим числом полезных коммитов в свои и чужие открытые проекты — это, считайте, половина резюме успешного разработчика.

Поэтому сейчас мы создадим себе аккаунт на **GitHub**.

2. Создание аккаунта на GitHub

Первым делом перейдите по ссылке. Заполните Username — ваше имя пользователя в GitHub. Это **ник** — по нему вас будут узнавать. Имя пользователя видят все посетители ваших репозиториев.

Укажите адрес электронной почты, придумайте пароль и нажмите на кнопку.

Затем выберите **Unlimited public repositories for free**. Остальные галочки вам не нужны.

На третьем шаге пока можно ничего не трогать, а просто нажать **Submit**.

Теперь проверьте электронный почтовый ящик — вам придёт ссылка для подтверждения аккаунта.

Войдя на сайт, нажмите на ссылку **Start a project**, чтобы завести свой первый сетевой репозиторий. Введите имя репозитория **git_lesson_repository** и нажмите **Create repository**.

Поздравляем — ваш первый репозиторий создан!

Сохраните ссылку на него — она имеет вид **https://github.com/<ваш username>/git_lesson_repository.git**. Работая с примерами, не забывайте подставлять в ссылку своё имя пользователя.

После урока стоит почитать инструкцию GitHub для новых участников.

3. Синхронизация с сетевыми репозиториями

Давайте попробуем выгрузить наш локальный репозиторий из прошлого урока в новый сетевой репозиторий на **Гитхабе**.

Перейдём в папку с созданным ранее репозиторием:

```
> cd git_project_1
> pwd

    /files/git_project_1
```

У нас уже существует рабочая версия локального репозитория, а удалённый (в Git

он называется **remote**) репозиторий пока пуст. Нам необходимо подключить удалённый репозиторий к нашему локальному и отправить («запУшить») файлы на сервер.

Имейте в виду, что к локальному репозиторию можно подключить несколько удаленных.

Для управления **remote**-репозиториями используется команда **git remote**. Узнать детальнее про её параметры можно через **git help remote**.

Подключим к нашему локальному репозиторию удалённый, применив команду **git remote add <name> <url>**. В нашем примере мы подключаем репозиторий с именем **github**:

```
> git remote add github https://github.com/user/git_lesson_repository.git
```

Чтобы увидеть подключённые репозитории, запускаем команду **git remote -v**:

```
> git remote -v

    github      https://github.com/user/git_lesson_repository.git (fetch)
    github      https://github.com/user/git_lesson_repository.git (push)
```

Удалённый репозиторий успешно добавился, но информация о нем отображается 2 раза:

1. Он добавлен как репозиторий для вытягивания (**fetch**) изменений;
2. И как репозиторий для сохранения удалённых изменений (**push**).

Теперь загрузим изменения из нашего локального репозитория в сетевой (удалённый) репозиторий. Для этого используется команда **git push -u <имя удалённого репозитория> <имя локальной ветки>**.

По умолчанию команда **git push** работает с текущей активной веткой, но требует указания имени удалённого репозитория (вспомните имя, написанное нами в качестве первого аргумента при выполнении команды **git remote add <name> <url>**).

Но сначала немного настроим наш Git:

```
> git config --global push.default current
```

```
> git checkout master
> git push -u github master
```

Команда запросит ваши логин и пароль, указанные при регистрации на GitHub.

```
Counting objects: 21, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (21/21), 1.93 KiB | 495.00 KiB/s, done.
Total 21 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), done.
To https://github.com/user/git_lesson_repository.git
* [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'github'.
```

После её выполнения появится надпись **[new branch] master → master**, которая означает, что в удалённом репозитории успешно создана новая ветка master, и в неё скопированы изменения из локальной ветки master.

Вернитесь в веб-интерфейс Github, открыв в вашем браузере ссылку на репозиторий (https://github.com/имя_пользователя/git_lesson_repository.git), и убедитесь, что в веб-интерфейсе появился наш файл program.py из ветки master.

Обратите внимание на ссылку **Commits (7)** в верхней части репозитория. Перейдя по ней, вы увидите все коммиты в ветке master, которые мы делали на прошлом уроке.

Важно: при загрузке ветки в удалённый репозиторий копируется не только актуальное состояние ветки, но и вся история коммитов в эту ветку, что позволяет всем пользователям удалённого репозитория легко восстановить хронологию «развития» вашей программы.

Чтобы «запУшить» сразу все локальные ветки в удалённый репозиторий, можно воспользоваться командой **git push <имя репозитория> --all**. Давайте попробуем:

```
> git push github --all

Total 0 (delta 0), reused 0 (delta 0)

To https://github.com/user/git_lesson_repository.git
 * [new branch]      addAuthorBranch -> addAuthorBranch
 * [new branch]      addFooter -> addFooter
 * [new branch]      python3branch -> python3branch
```

Все три ветки, которые нами были использованы на прошлом уроке, выгрузились в сетевой репозиторий. В этом можно убедиться, зайдя в репозиторий через web-интерфейс.

4. Клонирование сетевого репозитория

Теперь представим, что локального репозитория у вас нет (например, вы начали работать с другого компьютера или к вашему проекту присоединился ещё один разработчик). Есть только ссылка на удалённый репозиторий. В этом случае репозиторий необходимо **склонировать**.

Для этого есть команда **git clone <URL репозитория>**.

Выполняя эту команду, Git проверяет существование удалённого репозитория. Если репозиторий есть, то создаётся локальный репозиторий, и в него подтягиваются изменения из ветки, на которую указывает HEAD. Как правило, это **master** удалённого репозитория. Удалённый репозиторий добавляется и как **upstream** (с возможностью загрузки), и как **downstream** (с возможностью выгрузки), и получает имя **origin**.

Попробуем:

1. Создадим пустую папку **git_project_1_clone**.

2. Перейдём туда и выполним в ней команду **git clone <адрес вашего репозитория> <целевая директория>** (помните, что «.» в Linux — это «текущая директория»):

```
> cd files
> mkdir git_project_1_clone
> cd git_project_1_clone
> git clone https://github.com/user/git_lesson_repository.git .

    bash: cd: files: No such file or directory
    Cloning into '.'...
    remote: Counting objects: 21, done.
    remote: Compressing objects: 100% (7/7), done.
    remote: Total 21 (delta 4), reused 21 (delta 4), pack-reused 0
    Unpacking objects: 100% (21/21), done.
```

Проверим, что репозиторий действительно загрузился:

```
> cat program.py

    # I am author!
    print("My first Git program!!!")
    print("Hello, python")
    # 2017 (c) Me
```

Проверим созданные **remotes**-репозитории:

```
> git remote -v
```

```
origin      https://github.com/user/git_lesson_repository.git (fetch)
origin      https://github.com/user/git_lesson_repository.git (push)
```

Готово! Теперь удалённый репозиторий склонирован, и с ним можно работать как с полноценным локальным репозиторием. В том числе отправлять изменения в удалённый репозиторий, если есть права на запись. Обратите внимание, что локально вы увидите только ветку **master**:

```
> git branch

   * master
```

Почему так? Считается, что **master** — это основная рабочая ветка репозитория (на неё указывает **HEAD**) и по умолчанию копируется только она.

Однако существует несложный способ посмотреть все ветки в удалённом репозитории.

Для этого используется команда **git branch -a**:

```
> git branch -a

   * master
     remotes/origin/HEAD -> origin/master
     remotes/origin/addAuthorBranch
     remotes/origin/addFooter
     remotes/origin/master
     remotes/origin/python3branch
```

Из вывода этой команды видно, что существует только одна локальная ветка master и четыре удалённых ветки:

— master;

— addAuthorBranch;

— addFooter;

— python3branch.

Есть и одна виртуальная ветка **HEAD**, которая синхронизирована с origin/master, т.е. удалённой веткой master.

На удалённые ветки можно переключаться. Для этого применяется уже знакомая вам команда переключения между ветками **git checkout**. Имя ветки, на которую мы хотим переключиться, указывается в формате <имя upstream>/<имя ветки>. Например: **git checkout origin/addFooter**.

Так как удалённые ветки по умолчанию не имеют связей с локальными веткам, для работы с ними рекомендуется создавать локальные ветки, привязанные к удалённым командой **git branch <имя ветки> -f <имя апстрима>/<имя удаленной ветки>**.

Принято давать локальным и удалённым веткам одинаковые имена:

```
> git branch addFooter -f origin/addFooter

    Branch 'addFooter' set up to track remote branch 'addFooter'
    from 'origin'
```

Теперь работать с веткой **addFooter** можно как с полноценной локальной веткой. Синхронизация изменений (push и pull) будет проходить с удалённой веткой **origin/addFooter**.

Теперь создадим новую ветку в скопированном репозитории и загрузим её на сервер. Для этого используем уже знакомую нам с прошлого урока команду **git checkout -b <имя ветки>** и рассмотренную на этом уроке команду **git push <имя downstream>**.

```
> git checkout -b myNewBranch

    Switched to a new branch 'myNewBranch'

> git push origin

    Total 0 (delta 0), reused 0 (delta 0)
    To https://github.com/user/git_lesson_repository.git
     * [new branch]      myNewBranch -> myNewBranch

> git branch -a

      addFooter
      master
    * myNewBranch
      remotes/origin/HEAD -> origin/master
      remotes/origin/addAuthorBranch
      remotes/origin/addFooter
      remotes/origin/master
      remotes/origin/myNewBranch
      remotes/origin/python3branch
```

Теперь переключимся на свой исходный локальный репозиторий git_project_1 (из предыдущего урока) и посмотрим на его список веток:

```
> cd git_project_1
> git branch -a
```

```
  addAuthorBranch
  addFooter
* master
  python3branch
  remotes/github/addAuthorBranch
  remotes/github/addFooter
  remotes/github/master
  remotes/github/python3branch
```

Что такое? Нашей новой ветки **myNewBranch** нет не только в списке локальных веток, но и в списке веток удалённого репозитория!

Неужели мы где-то ошиблись, и ветка не сохранилась в удалённом репозитории? Конечно нет!

Просто наша локальная копия удалённого репозитория пока ещё ничего не знает о состоянии самого удалённого репозитория. Ведь мы ещё не провели синхронизацию.

Для синхронизации используется команда **git fetch <имя upstream>**.

Попробуем:

```
> git fetch github

   From https://github.com/user/git_lesson_repository
    * [new branch]      myNewBranch -> github/myNewBranch

> git branch -a

    addAuthorBranch
    addFooter
  * master
    python3branch
    remotes/github/addAuthorBranch
    remotes/github/addFooter
    remotes/github/master
    remotes/github/myNewBranch
    remotes/github/python3branch
```

Вот теперь актуальный список веток удалённого репозитория виден и доступен для локальной работы. Выполнять команду **fetch** рекомендуется перед началом работы над любой задачей или перед любой работой с ветками (создание, удаление, слияние).

Давайте теперь изменим файл program.py (добавим слово **new** во второй строке):

```
> cat program.py

    # I am new author!
    print("My first Git program!!!")
    print("Hello, python")
    # 2017 (c) Me
```

Сделаем коммит новой версии и сохраним её в удалённом репозитории:

```
> git commit -a -m 'Fix author name'

    [master 9336df2] Fix author name
     1 file changed, 1 insertion(+), 1 deletion(-)

> git push github

    Counting objects: 3, done.
    Delta compression using up to 4 threads.
    Compressing objects: 100% (2/2), done.
    Writing objects: 100% (3/3), 345 bytes | 345.00 KiB/s, done.
    Total 3 (delta 0), reused 0 (delta 0)
    To https://github.com/user/git_lesson_repository.git
       9a704f6..9336df2  master -> master
```

Переключимся на локальный репозиторий git_project_1_clone, перейдём на master, выполним **git fetch** и посмотрим на содержимое program.py:

```
> cd git_project_1_clone
> git checkout master
> git fetch

    Switched to branch 'master'
    Your branch is up to date with 'origin/master'.
    remote: Counting objects: 3, done.
    remote: Compressing objects: 100% (2/2), done.
    remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
    Unpacking objects: 100% (3/3), done.
    From https://github.com/user/git_lesson_repository
       9a704f6..9336df2  master     -> origin/master

> cat program.py

    # I am author!
    print("My first Git program!!!")
```

```
    print("Hello, python")
    # 2017 (c) Me
```

Синхронизация прошла, но файл не поменялся. Это связано с тем, что команда **git fetch** обновляет только мета-информацию репозитория, не затрагивая отслеживаемые файлы и не внося изменения в локальные ветки.

Если ещё раз выполнить **git checkout master**, то можно увидеть следующее:

```
> git checkout master

    Already on 'master'
    Your branch is behind 'origin/master' by 1 commit, and can be
    fast-forwarded. (use "git pull" to update your local branch)
```

Git сообщил нам о том, что текущая ветка master локального репозитория неактуальна.

Проигнорируем предупреждение и изменим нашу программу следующим образом (добавили 2018 год):

```
> cat program.py

    # I am author!
    print("My first Git program!!!")
    print("Hello, python")
    # 2017-2018 (c) Me
```

Сделаем коммит и отправим изменённую версию в сетевой репозиторий:

```
> git commit -a -m "Update copyright years"

    [master a2ffe3f] Update copyright years
     1 file changed, 1 insertion(+), 1 deletion(-)

> git push origin

    To https://github.com/user/git_lesson_repository.git
     ! [rejected]        master -> master (non-fast-forward)
    error: failed to push some refs to
           'https://github.com/user/git_lesson_repository.git'
    hint: Updates were rejected because the tip of your current branch
          is behind
    hint: its remote counterpart. Integrate the remote changes (e.g.
    hint: 'git pull ...') before pushing again.
```

```
    hint: See the 'Note about fast-forwards' in 'git push --help'
          for details.
```

Локальный коммит прошёл, но, при попытке **«залить»** (это такой профессиональный сленг) изменения в удалённый репозиторий, мы получили ошибку, так как вносили изменения не в последнюю версию ветки.

Добавим изменения из сетевого репозитория в нашу локальную ветку командой **git pull** (Git подсказал нам об этом), аналогом команды **git merge**, предназначенным для объединения версии из удалённого репозитория с локальной версией текущей активной ветки:

```
> git pull

    Auto-merging program.py
    <files/git_project_1_clone/.git/MERGE_MSG
    "Merge branch 'master' of
     https://github.com/user/git_lesson_repository/.git/MERGE_MSG"

    # Please enter a commit message to explain why this merge is necessary
    # especially if it merges an updated upstream into a topic branch.
    #
    # Lines starting with '#' will be ignored, and an empty message aborts
    # the commit.
```

После запуска команды **git pull** Git запросит комментарий к объединению версий (по аналогии с комментарием, который мы передаём с параметром **-m <сообщение>** при коммите).

Надо написать комментарий и ввести **:q** или нажать **CTRL+W, CTRL+O** для выхода из текстового редактора, после чего изменения будут объединены.

Обратите внимание, что на этом этапе могут возникнуть конфликты такого же плана, что появлялись у нас в ходе первого урока при объединении коммитов.

Для решения конфликта нужно действовать так же, как и в прошлый раз:

1. Вручную исправить файл с конфликтом.
2. Через **git commit** зафиксировать версию.

Посмотрим на обновленную версию программы и отправим её в удалённый репозиторий:

```
> cat program.py

    # I am new author!
    print("My first Git program!!!")
    print("Hello, python")
    # 2017-2018 (c) Me
```

```
> git push origin

    Counting objects: 6, done.
    Delta compression using up to 4 threads.
    Compressing objects: 100% (4/4), done.
    Writing objects: 100% (6/6), 624 bytes | 312.00 KiB/s, done.
    Total 6 (delta 2), reused 0 (delta 0)
    remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
    To https://github.com/user/git_lesson_repository.git
       9336df2..a72d9c7  master -> master
```

Теперь в файле есть изменения из обоих коммитов и версия зафиксирована в удалённом репозитории.

Любой разработчик, который будет работать с удалённым репозиторием, после выполнения **git fetch** получит уведомление об изменении в ветке и сможет забрать изменения командой **git pull**.

На этом моменте мы закончим изучение основных функций Git для работы с сетевыми репозиториями.

Больше информации вы можете самостоятельно почерпнуть из документации Git (man git, git help, git help <команда>).

Теперь вы умеете создавать репозиторий, заводить, просматривать и удалять ветки, фиксировать версии (коммитить), переключаться между версиями, подключать сетевой репозиторий, получать правки из него, вносить правки в удалённый репозиторий, объединять изменения, решать конфликты при объединении и клонировать удалённый репозиторий к себе.

Этого достаточно для того, чтобы начать работать над проектом в команде.

Далее мы изучим правила командного взаимодействия в Git.

5. Схема командной работы с репозиторием

Мы с вами уже научились работать с локальными и удалёнными репозиториями, создавать, удалять и объединять ветки, фиксировать изменения в локальном и удалённом репозитории. Но во всех наших упражнениях ситуация была упрощённой — с репозиторием работал только один разработчик.

Когда разработчик один, то не особенно важно, как называть ветки, когда создавать новые ветки, в какое время и куда их вливать. Но всё кардинально меняется, если с репозиторием работают несколько разработчиков:

— Многократно возрастает вероятность конфликтов в ветках;

— Угадать, какие изменения сделаны в ветке, невозможно, если нет понятных имён веток и комментариев к коммитам;

— Появляется необходимость как-то договориться о том, где мы храним полностью работающую программу, а где — ведём разработку и экспериментируем.

И, наконец, нужно как-то работать с удалённым репозиторием, не мешая друг другу.

Для решения перечисленных проблем нужно, чтобы все разработчики проекта следовали единому своду правил работы с репозиторием.

Перечислим требования, которые предъявляются к такому своду:

1. Единый стандарт именования веток.

2. Единый стандарт ветвления в репозитории.

3. Единые правила изменения стабильной ветки (master).

4. Единые правила объединения репозиториев.

5. Единые правила работы над проектными задачами.

Такие своды правил называют **flow**.

В современной разработке чаще всего используются две популярные схемы (в пределах одного проекта рекомендуется выбрать и использовать только одну):

1. Классический подход **Git Flow**.

2. **GitHub Flow**, адаптированный под коллективную работу на GitHub.

Перед началом изучения методологий работы с репозиторием давайте зафиксируем терминологию — она идентична для всех основных **flow**:

— Стабильная ветка (**Stable**) — ветка, в которой программа в любой момент считается рабочей и хорошо протестированной. В идеале программу из этой ветки всегда можно передать заказчику — или, как часто говорят, **в продакшн** (production — рабочее состояние продукта). В большинстве методологий в качестве стабильной используется ветка **master**;

— Ветка разработки (**Dev**) — ветка, относительно которой заводятся все новые изменения (именно от этой ветки отделяются новые ветки для разработки фич). Как правило, это ветка со всеми законченными в настоящий момент изменениями, без нестабильных изменений и изменений «в работе». Код в этой ветке чаще всего работоспособен, но не протестирован достаточно хорошо для передачи **в продакшн**. Из этой ветки часто собираются будущие релизы;

— **Релиз** — процесс вливания ветки релиз-кандидата в мастер-ветку и выкладки её в продакшн;

— **Релиз-кандидат/релизная ветка** — ветка, где зафиксированы на момент релиза (выпуска новой версии продукта) все изменения, которые следующим шагом *поедут*

в продакшн, то есть будут включены в новую версию. После сбора этой ветки вносить в неё изменения, кроме исправления ошибок (или *багов*), обычно запрещено;

— **Фича**-бранч — ветка, в которой разработчик ведёт разработку одной конкретной задачи или функциональности;

— **Хотфикс** — ветка, отходящая от стабильной ветки или релиз-кандидата, нужная для исправления критического бага и обратного вливания в ту же ветку и все дочерние ветки.

6. GitFlow

Самая старая и, вероятно, одна из наиболее распространённых методологий работы с репозиторием. Про неё часто спрашивают на различных собеседованиях при приёме на работу и применяют в относительно консервативных командах.

GitFlow использует описанные выше сущности следующим образом:

— Stable — ветка **master**;

— Dev — ветка **develop**;

— Фича-бранчи — ветки вида **feature/<имя фичи/номер тикета в трекинг-системе>** (это такие системы, в которых разработчикам ставят задачи и определяют сроки их выполнения, а разработчики отчитываются о результатах своей работы);

— Хотфиксы — ветки вида **hotfix/<имя исправляемого бага/номер тикета в трекинг-системе>**;

— Релизные ветки — ветки вида **release/<номер релиза/дата начала релиза>**. Часто в качестве номера релиза используется symver-нотация: x.y.z, где x — версия программного продукта, версии с разной x-версией часто обратно несовместимы; y — номер релиза, если меняется только y — обратная совместимость не нарушается, чаще — исправляется старый функционал или добавляется опциональный новый; z — номер *патча*, обнуляется при каждом обновлении y, часто используется для обозначения порядкового номера хотфикса. Подробнее о семантическом версионировании читайте тут;

— Релиз/Тег — тег в ветке **master**, совпадающий по имени с именем релизной ветки (подробнее про теги читайте в инструкции **git help tag**).

В качестве примера трекинг-систем можно привести следующие:

1. JIRA;
2. Redmine;
3. Яндекс-трекер.

Общая схема работы с репозиторием в рамках **GitFlow** представлена на рисунке.

Обратите внимание на то, как происходит выпуск **релизов**. Например, релиз с номером 1.1.1 получился только после внесения критичных правок, которые добавились и в **коммит** в ветке **develop**.

А версия 1.2.0 собралась уже из релизной ветки.

Предлагаем вам немного поработать по методологии **GitFlow**: выполните оба задания **GitFlow. Командная работа** из классных задач.

7. GitHubFlow

GitHubFlow — современная методология, которая распространена в компаниях, использующих систему GitHub. Она отлично совместима с процессом непрерывной поставки ПО и публичной работой над свободным программным обеспечением.

GitHubFlow оперирует следующими сущностями:

— Stable — ветка **master**;

— Dev — сущность отдельно не выделяется. В качестве ветки, от которой осуществляется ветвление, используется ветка **master**;

— Фича-бранчи — ветки вида **feature/<имя фичи/номер тикета в трекинг-системе>**;

— Хотфиксы — ветки вида **hotfix/<имя исправляемого бага/номер тикета в трекинг-системе>**;

— Релизные ветки — в виде отдельный сущности, как правило, не используются. GitHubflow приветствует идеологию 1 фича = 1 релиз;

— Релиз/Тег — тег в ветке **master**, совпадающий по имени с релизной веткой (подробнее про теги читайте в инструкции **git help tag**);

— Pull-request — сущность, специфичная для системы GitHub, заменяющая собой **push** в ветку **master**. После окончания работы над фича-веткой разработчик не делает **merge** в мастер-ветку, а создаёт **предложение** о внесении изменений в **GitHub** — так называемый **pull-request**. Далее основные разработчики проекта проверяют код пул-реквеста (смотрят на корректность вносимых изменений или делают codereview) и либо принимают пул-реквест (предварительно решив конфликты обычным слиянием), либо отклоняют пулл-реквест. Во втором случае изменения не попадают в ветку **master**.

Подробнее почитать про GitHubFlow можно тут.

Общая схема работы с репозиторием в рамках **GitHubFlow** представлена на рисунке.

Давайте попробуем свои силы и в этой методологии. Снова объединитесь в группы (можно остаться в тех группах, в которых работали ранее, только доверьте роль тим-лида кому-то

другому) и выполните задание **GitHubFlow. Командная работа**.

Когда закончите работу, покажите результат преподавателю.

Подробнее про работу с GitHub читайте тут.