Лекция 11

Git, GitHub
Команды git
Git Flow

Ещё раз о том, что такое Git

Git — это система контроля версий, которая позволяет удобно организовать рабочий процесс для команды разработчиков. Достигается это тем, что с помощью **Git** можно: сохранять разные состояния проекта и легко перемещаться между ними; делать новые ответвления от текущего состояния проекта, дабы “в изоляции” разрабатывать новые фичи; “подтягивать” новые изменения из других веток и довольно просто решать возникшие конфликты. При всём этом обилии полезных возможностей сам **Git** остаётся довольно маленькой и быстрой в работе утилиткой. Большая скорость обусловлена тем, что **Git** ставится и работает локально на вашем ПК.

Историю изменений проекта **Git** хранит в виде набора “замороженных” состояний, к каждому из которых вы сможете в последствии вернуться. На каждом этапе сохранения нового состояния проекта в **Git**, система запоминает, как выглядит каждый файл в этот момент и сохраняет ссылку на это состояние. В конечном итоге получается длинная история из изменений, каждое из которых накладывает какой-то новый кусок кода на ваш проект.

GitHub и распределённые системы контроля версий

Насколько бы “крут” и хорош не был Git, до тех пор, пока он хранит историю вашего проекта локально, он всё ещё не решает никаких проблем командной разработки.

GitHub — это онлайн сервис для хранения проектов. Также он включает в себя и перечень возможностей систем контроля версий. Хранятся проекты на GitHub в виде **репозиториев** (сокращённо “репа” или “реп”, если говорить в множественном числе). Каждый **репозиторий** имеет свой уникальный **URL** и хранит в себе все файлы, ветки и полную историю вашего проекта. Именно отсюда, как правило, и начинается разработка новых проектов. Первым делом всегда создаётся **репозиторий**, а все разработчики “стягивают” его себе и продолжают работать с ним локально, периодически внося изменения обратно в удалённый **репозиторий**.

Распределённые системы контроля версий, к которым относится **Git**, предоставляют одну очень важную возможность: полная “выкачка” абсолютно всей информации из **репозитория** к себе на ПК. В итоге каждый разработчик имеет копию всех данных проекта и в случае утери (отказа или “слёта” удалённого сервера) можно будет с лёгкостью в полном объёме восстановить всю базу данных проекта.

Основные возможности Git для работы с локальными репозиториями

- **git init** — инициализация нового репозитория;
- **git add** — поэтапное добавление изменений;
- **git commit** — регистрация некоторого перечня изменений с описанием и назначением уникального ID этим изменениям;
- **git status** — отслеживание текущего состояния репозитория;
- **git config** — запись и чтение конфигурации Git;
- **git branch** — отображение текущей ветки, создание новых веток, удаление веток;
- **git checkout** — переключение на другие ветки;
- **git merge** — слияние веток, т.е. соединение изменений двух веток в одну.

Git init

Данная команда “превращает” директорию на вашем ПК в пустой репозиторий **Git**. Это первый шаг на пути создания репозитория. После выполнения данной команды можно приступать к добавлению и регистрации (созданию коммитов) изменений.

Пример:

```
ramanaliaksanau@Ramans-MacBook-Pro ~ % cd Documents/Work/Git-repo-example
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git init
Initialized empty Git repository in /Users/ramanaliaksanau/Documents/Work/Git-repo-example/.git/
```

Git add

С помощью данной команды можно добавлять изменённые (добавленные) файлы в некоего рода промежуточный источник данных **Git**. Это необходимо для дальнейшей регистрации (создание коммитов на их базе) этих изменений. Существует несколько способов добавления файлов: добавление файлов каждого по отдельности, добавление директории файлов или добавление всех изменений сразу.

Примеры:

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git add ./index.html
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git add css
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git add .
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example %
```

Git commit

Данная команда позволяет записать/зарегистрировать изменения, внесённые в файлы локального репозитория. При этом новый набор изменений (в дальнейшем коммит) получает свой уникальный идентификатор, по которому его можно найти.

Хорошей практикой считается добавление исчерпывающего описания в сообщении нового коммита. Данное описание должно приводить объяснения к внесённым изменениям. В дальнейшем хорошие сообщения коммитов очень сильно упрощают поиск по истории изменений проекта.

Пример:

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git commit -m "Create initial project structure"
[master (root-commit) 8254583] Create initial project structure
 4 files changed, 23 insertions(+)
 create mode 100644 css/normalize.css
 create mode 100644 css/style.css
 create mode 100644 index.html
 create mode 100644 js/main.js
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example %
```

Git status

Данная команда сообщает вам о текущем состоянии репозитория.

Вызов команды **git status** отобразит ветку, на которой вы находитесь в данный момент. Кроме этого, если у вас есть файлы на стадии добавления, т.е. ещё не записанные в коммит, информация об этом также будет отображена. Если же никаких изменений готовых к регистрации нет, данная команда выведет сообщение “*nothing to commit, working tree clean*”.

Примеры:

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   css/normalize.css
        new file:   css/style.css
        new file:   index.html
        new file:   js/main.js
```

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git commit -m "Init project structure"
[master (root-commit) ceae1e9] Init project structure
 4 files changed, 23 insertions(+)
 create mode 100644 css/normalize.css
 create mode 100644 css/style.css
 create mode 100644 index.html
 create mode 100644 js/main.js
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git status
On branch master
nothing to commit, working tree clean
```

Git config

Git позволяет задавать нам множество самых разных настроек, а делается это с помощью команды **git config**. Самыми важными пунктами конфигурации являются **user.name** и **user.email**. Данные значения представляют собой информацию об авторе коммитов на локальном устройстве.

У команды **git config** есть флаг **--global**, который позволяет нам задать значение конфигурации для всех репозиториев сразу. Без данного флага настройки будут влиять лишь на текущий репозиторий.

Существует большое количество настроек, которые можно задать с помощью команды **git config**. Подробнее о них можно почитать здесь.

Примеры:

```
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git config --global user.name "Roman Alexanov"
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git config --global user.email alexanov.roman@gmail.com
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git config user.name "Raman Aliaksanau"
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git config user.name
 Raman Aliaksanau
```

Git branch

Данная команда нужна для определения текущей ветки, а также для создания и удаления веток.

Примеры:

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch
* master
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch feature-1
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch
  feature-1
* master
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch -a
  feature-1
* master
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch -d feature-1
Deleted branch feature-1 (was ceae1e9).
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch -a
* master
```

Git checkout

Данная команда нужна для смены рабочей (текущей) ветки. Используйте **git checkout** для переключения на новую ветку.

Также с помощью данной команды можно создавать новые ветки. В таком случае после вызова **git checkout -b new-branch-name** новая ветка будет создана и сразу же станет текущей.

Примеры:

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch
* master
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git checkout test
error: pathspec 'test' did not match any file(s) known to git
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git checkout -b test
Switched to a new branch 'test'
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch
  master
* test
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git checkout master
Switched to branch 'master'
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git branch
* master
  test
```

Git merge

Данная команда производит слияние изменений из двух веток. Как результат вызова **git merge** создаётся новый коммит, в котором находятся изменения текущей и целевой веток.

Частая ситуация: последние изменения из ветки **dev** (или **develop**) сливаются с изменениями на вашей ветке (**feature**). Это делается для того, чтобы получить доступ к новым возможностям из ветки **dev** или для разрешения конфликтов.

Пример:

```
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git checkout test
Switched to branch 'test'
[ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git merge master
Updating ceae1e9..9c12ac4
Fast-forward
 index.html | 138 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-
 1 file changed, 137 insertions(+), 1 deletion(-)
```

Основные возможности Git для работы с удалёнными репозиториями

- **git remote** — создание связи между локальным и удалённым репозиториями;
- **git clone** — создание локальной копии существующего удалённого репозитория;
- **git fetch** — получение данных об изменениях в ветке;
- **git pull** — получение самой последней версии репозитория;
- **git push** — отправка локальных изменений в виде перечня коммитов на удалённый репозиторий.

Git remote

Данная команда нужна для соединения локального репозитория с удалённым. При этом удалённый репозиторий может иметь какое-то название, чтобы не пришлось запоминать или хранить его полный URL-адрес.

Пример:

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git remote add test https://github.com/      /repo-example.git
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git remote -v
origin  https://github.com/      /repo-example.git (fetch)
origin  https://github.com/      /repo-example.git (push)
test    https://github.com/      /repo-example.git (fetch)
test    https://github.com/      /repo-example.git (push)
```

Git clone

Данная команда нужна для создания локальной копии существующего удалённого репозитория. После вызова команды **git clone**, **git** создаст новую директорию с именем равным названию удалённого репозитория и всем его содержимым (файлы, ветки, история изменений).

Пример:

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git clone https://github.com/[redacted]/andersen-lesson-5.git
Cloning into 'andersen-lesson-5'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example %
```

Git fetch

Данная команда нужна для получения самых свежих данных с удалённого репозитория. Это могут быть новые ветки, коммиты и т.д. Однако важно понимать, что данная команда просто получает эти данные, но ничего с ними не делает.

Пример:

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 733 bytes | 366.00 KiB/s, done.
From https://github.com/[redacted]/repo-example
   9c12ac4..97e4f07  master     -> origin/master
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example %
```

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git merge
Updating 9c12ac4..97e4f07
Fast-forward
 file-from-git.js | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 file-from-git.js
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example %
```

Git pull

Данная команда по сути совмещает в себе вызовы **git fetch** и **git merge**. Т.е. сначала идёт получение нового состояния ветки в удалённом репозитории, а затем автоматическое добавление новых данных в локальную ветку. При этом если ваша ветка уже находится в актуальном состоянии, то при вызове **git pull** будет выведено сообщение “*Already up to date.*”.

Пример:

Git push

Записывает локальные коммиты в удалённый репозиторий. Может принимать два параметра: название (или URL-адрес) удалённого репозитория и название ветки, изменения с которой мы хотим записать. Также в некоторых ситуациях данная команда может быть вызвана и без параметров.

Пример:

```
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git add .
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git commit -m "Add huge.js file"
[push-example 48abe93] Add huge.js file
 1 file changed, 2700 insertions(+)
 create mode 100644 huge.js
ramanaliaksanau@Ramans-MacBook-Pro Git-repo-example % git push test push-example
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 479 bytes | 479.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/      /repo-example.git
   12681c4..48abe93  push-example -> push-example
```

Ещё полезные команды Git

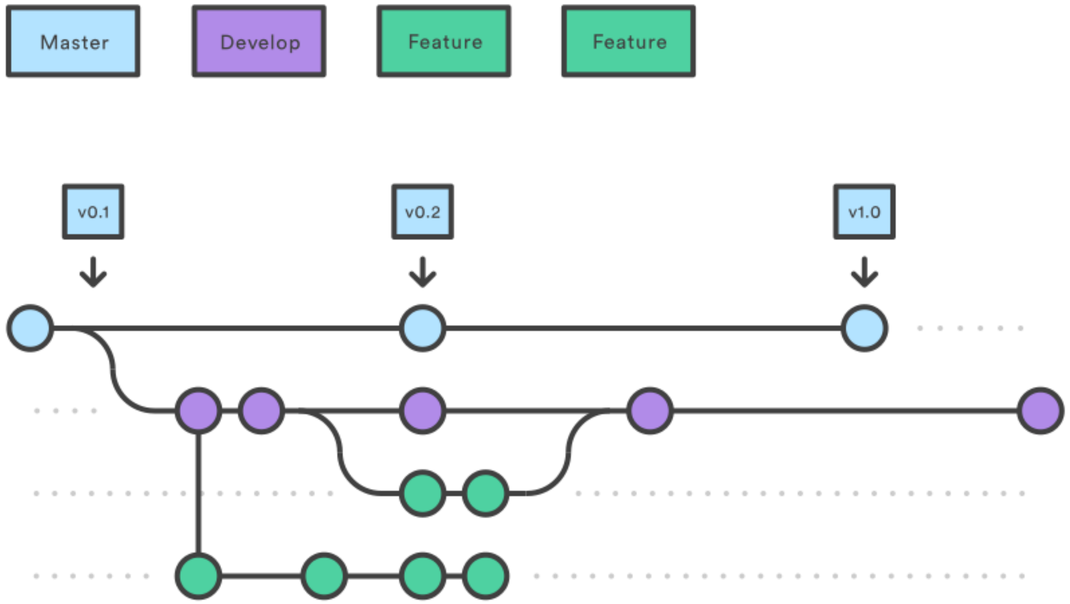
Для начала, конечно, стоит разобраться с командами с предыдущих слайдов, ибо они составляют основу, без которой сложно себе представить работу с системой контроля версий **Git**. Однако сразу после их усвоения на должном уровне, крайне рекомендую ознакомиться со следующим набором полезных команд:

- **git stash** — сохранение текущего состояния репозитория и очищение директории от всех изменений;
- **git cherry-pick** — вставка отдельного (-ых) коммита (-ов) в свою ветку;
- **git rebase** — перемещение нескольких коммитов к новому базовому коммиту.

Git Flow

Я думаю все вы не раз слышали о таком понятии как **Git Flow**. Его суть сводится к организации рабочего процесса, построенного с использованием **Git**.

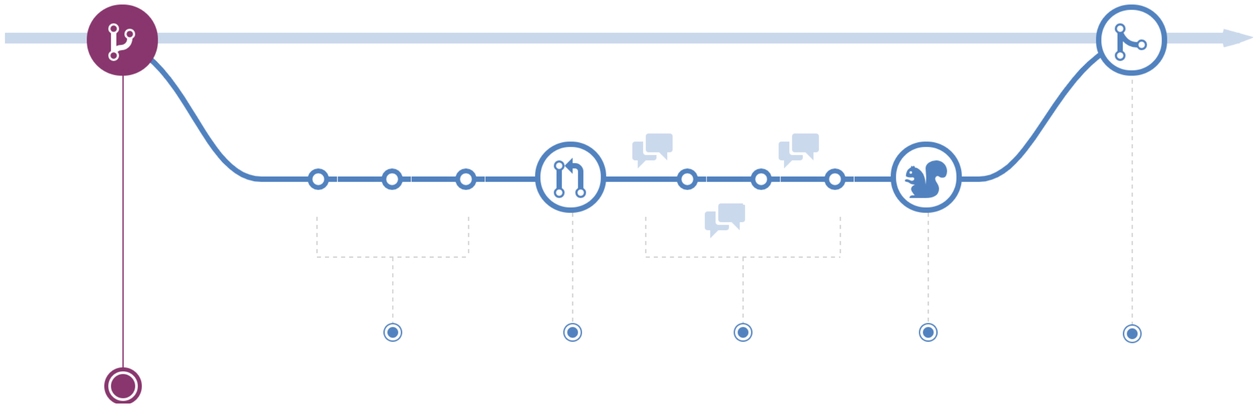
Git Flow задаёт строгую модель для надёжного управления и расширения более менее крупных проектов.



1. Создать новую ветку

В любой момент времени работы на проекте у вас наверняка будет целая куча разных фич или даже собственных идей в процессе разработки. Некоторые из них уже могут быть готовы, другие — нет. Именно для такой ситуации и существует возможность создания множества веток.

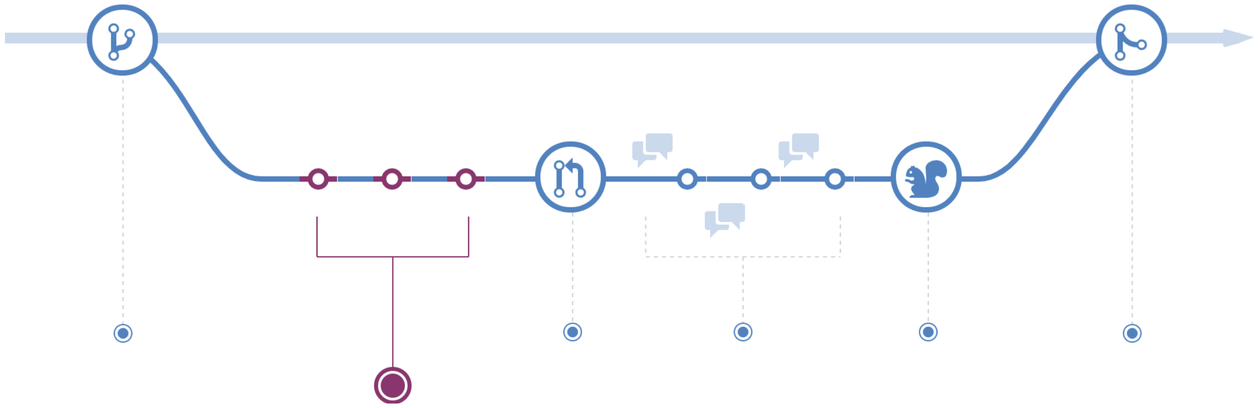
Всякий раз, создавая новую ветку в вашем проекте, вы как бы создаёте свою маленькую экосистему, которая отталкивается от текущего состояния проекта, но никак не влияет на него вплоть до момента, пока данные изменения не будут просмотрены, протестированы и внесены в основную ветку. Иными словами вы вольны экспериментировать и коммитить любые изменения в свою ветку не опасаясь “что-нибудь сломать” в основной версии проекта.



2. Добавить изменения (создать несколько новых коммитов)

Когда ваша ветка создана — время вносить изменения. Когда вы добавляете, редактируете или удаляете файлы, вы должны создавать коммиты на основе этих изменений и периодически добавлять их с свою ветку. Данный процесс добавления коммитов позволяет отследить ваш прогресс по введению новой фичи в проект.

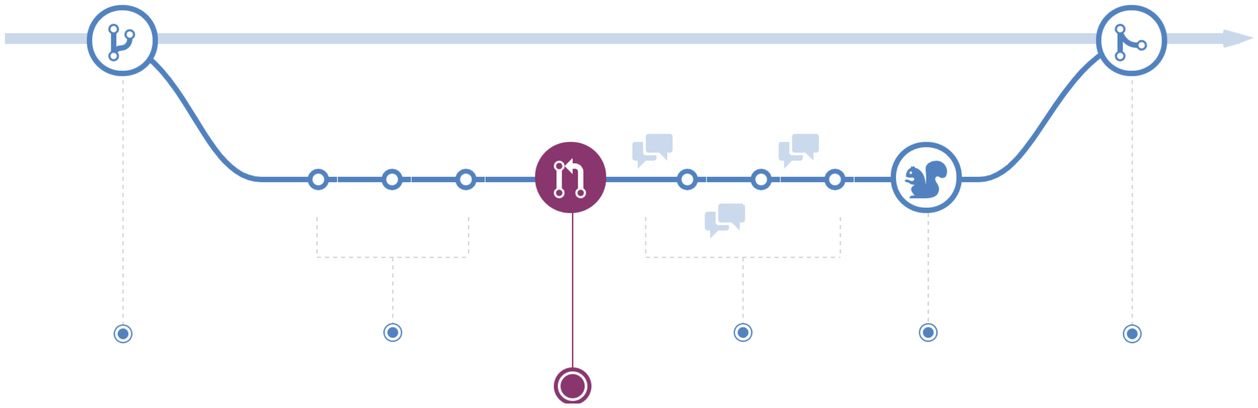
Кроме этого коммиты добавляют прозрачности истории вашей работы, которую впоследствии другие члены команды могут прочесть и понять, что было сделано и почему. Каждый коммит должен иметь сообщение с описанием внесённых изменений. Также при соблюдении этих правил не составит никаких проблем “откатить” вашу ветку на несколько коммитов назад в случае обнаружения бага.



3. Создать запрос на внесение изменений (pull/merge request)

Запрос на внесение изменений (в дальнейшем ПР) запускает процесс обсуждений ваших изменений. Т.к. данный ПР производится в рамках одного репозитория Git, каждый из членов команды может увидеть полный перечень изменений, которые будут внесены в случае апрува.

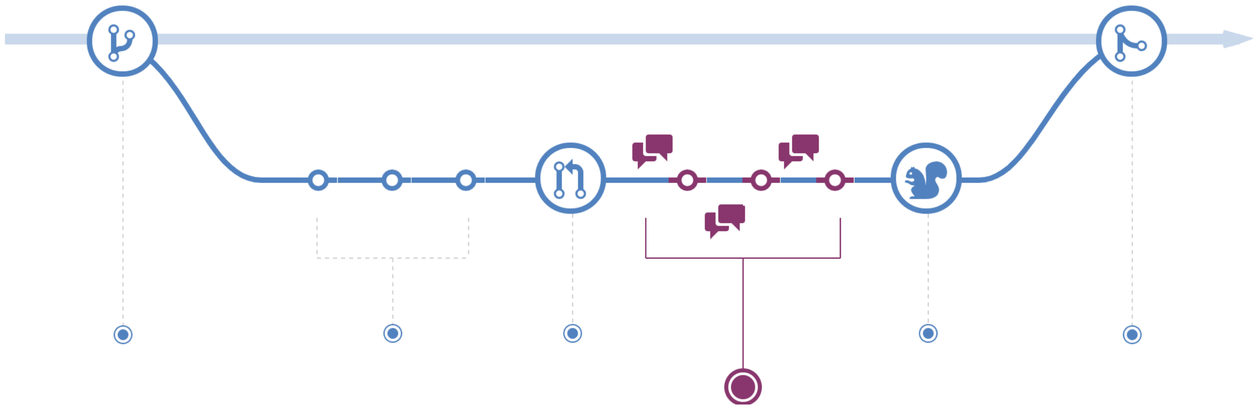
Вы можете создать ПР на любом этапе процесса разработки: когда у вас есть лишь наработки и вы хотите обсудить ваши идеи с кем-то ещё, когда вы застряли на каком-то из этапов и вам нужна помощь или совет, или же когда ваша работа полностью готова. С помощью системы **@упоминаний** в сообщениях к вашему ПР-у вы можете запросить отзыв у конкретных людей из команды.



4. Обзор и обсуждение кода
Внесение новых изменений

С момента создания ПР-а любой член вашей команды может сделать ревью кода, задавая при этом вопросы и оставляя комментарии. Такое может произойти если, например, не соблюдён кодстайл проекта, или для новой функции не написан юнит тест, а может наоборот всё выглядит замечательно и ваш подход к решению задачи оказался неочевидным, но очень действенным. ПР-ы буквально созданы для организации такого вида коммуникаций.

Даже в процессе обсуждений и отзывов о ваших изменениях вы можете продолжать добавлять новые коммиты. Если вы что-то забыли или какой-то фрагмент кода стал причиной возникновения ошибки, вы можете поправить это в своей ветке и тут же “запушить”. Новые коммиты также будут появятся в ПР-е.

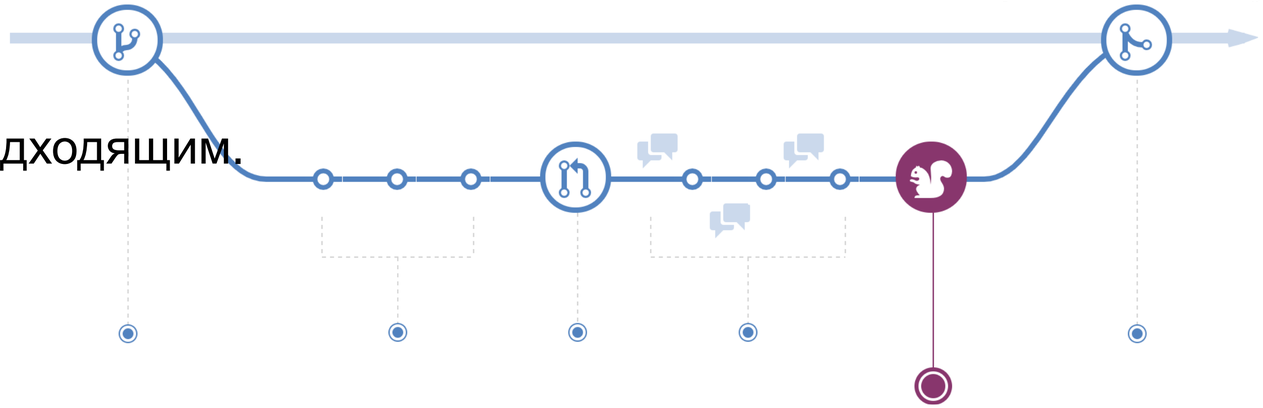


5. Развёртывание кода и финальное тестирование

С помощью GitHub вы можете развернуть продакшн вместе с изменениями из вашей ветки ещё до слияния с веткой **main** (**master**).

Когда ПР получил необходимые апрувы и все тесты прошли успешно, вы можете развернуть ваши изменения, чтобы проверить их прямо на продакшене. Если новый код становится причиной возникновение ошибок, вы можете “откатить” его обратно (просто развернув на продавшее версию проекта с ветки **main**).

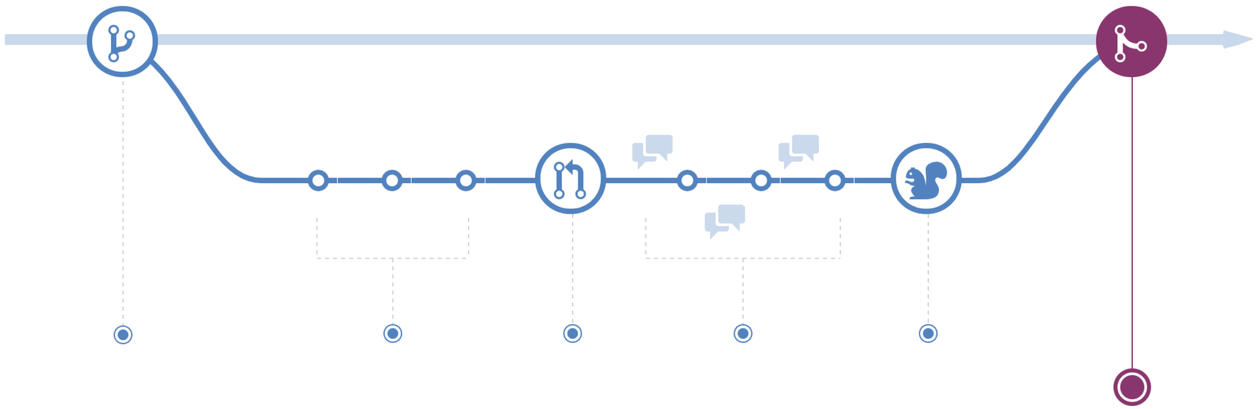
В разных командах также могут существовать специальные тестовые или промежуточные окружения. Данный подход на сегодняшний день является крайне популярным. Однако для некоторых именно вариант с продакшеном может оказаться наиболее подходящим.



6. Внесение изменений в основную ветку

Когда ваши изменения были утверждены на продакшене, самое время внести эти изменения в ветку **main**.

После добавления изменений, все ваши коммиты сохраняются в общей истории кодовой базы проекта. Благодаря этому в будущем любой разработчик сможет вернуться к вашему коду и понять, почему и как были приняты те или иные решения.



Q&A