

# Чеклист по оформлению репозитория

23 июля 2023 г.

## Чеклист по оформлению репозитория

1	Лицензия	
2	Лицензия правильно применяется к репозиторию	
3	Используемые третьесторонние компоненты и материалы совместимы с лицензией	
4	Настроенный CI	
5	Модульные тесты в CI	
6	Линтер в CI	
7	В репозитории нет результатов сборки, настроен .gitignore	
8	В репозитории нет секретной информации (паролей, ключей и т.п.)	
9	Различные сторонние анализаторы (если уместно)	
10	README.md, плашки CI и анализаторов	
11	README.md, общее описание проекта	
12	README.md, пример использования	
13	README.md, инструкция по сборке и запуску	
14	README.md, как помочь проекту (если уместно)	
15	Код соответствует принятому в сообществе стилю кодирования	
16	Имеется техническая документация (в README.md или на вики)	
17	В коде достаточно комментариев	
18	Комментарии к коммитам адекватны, коммиты показывают историю проекта	

## Пояснения

1. Отдавайте предпочтение разрешающим лицензиям. Код мы рекомендуем лицензировать под Apache License 2.0, MIT License, BSD 3-Clause License.

2. Каждая лицензия имеет требования к тому, как её правильно применить к файлам в репозитории. Например, Apache License 2.0 позволяет себя применять пофайлово, для чего требует включения в лицензируемые файлы стандартного заголовка. Также распространено использование файла LICENSE в корне репозитория и ссылка на него в заголовке каждого файла. Поищите для своей лицензии, как её правильно применять.
3. Если используете чужую интеллектуальную собственность, найдите на неё лицензию и проверьте, что вы действительно выполняете её требования (например, проект, лицензированный под Apache License 2.0 *не может* использовать код, лицензированный под GPL v2). Если чужой материал не имеет лицензии (например, просто картинка из интернета или кусок кода со Stack Overflow), использовать его *нельзя*.
4. Если вы используете GitHub, Continuous Integration-систему удобнее всего настраивать на GitHub Actions, однако вполне допустимы и сторонние системы, такие как AppVeyor, CircleCI. Если вы используете компилируемые языки, CI-система должна проверять собираемость кода в каждой ветке репозитория и при пуллреквесте. Если интерпретируемые, проверять качество кода и работоспособность.
5. В проекте должны быть модульные тесты (за редкими исключениями, где они неприменимы или бессмысленны), и модульные тесты должны запускаться в CI.
6. Должен быть настроен линтер, следящий за качеством кода, и также запускаться в CI. Если линтер выдаёт ошибки, сборка должна не проходить.
  - Например, для F# это Fantomas или FSharpLint, для Python — black и т.д.
  - Запуск линтера может быть отдельной задачей в CI, чтобы не гонять его по несколько раз в разных конфигурациях сборки.
  - Имеет смысл сделать запуск линтера локальным pre-commit hook в git, чтобы некорректный код даже не позволяли закоммитить.
7. На GitHub файл .gitignore можно выбрать при создании репозитория, но также часто требуется ручная модификация. Должно быть так, чтобы все файлы, которые .gitignore позволяет закоммитить, реально нужно было коммитить. *В репозитории не должно быть результатов сборки*, (то есть папок bin, obj, русache и т.п.), в идеале не должно быть бинарных файлов вовсе (только если очень надо и вы реально знаете, что делаете).
8. Разумеется, в репозитории (включая историю коммитов) не должно быть ничего, что вы не хотели бы публиковать (например, ключей авторизации от сообществ ВКонтакте). Если пользуетесь GitHub, кое-что он умеет ловить сам, для этого надо убедиться, что в «Settings/Code security and analysis» включено «Push protection». Но, разумеется, большую часть секретов он не найдёт.
9. Используйте сторонние анализаторы для слежения за качеством кода: например, CodeCov для анализа тестового покрытия, CodeFactor или Codacy как продвинутый статический анализатор. Чем больше инструментов следят за тем, что всё хорошо, тем лучше.

10. Добавьте в README.md плашки CI и анализаторов (штучки, на которых написано «CI passing» или что-то такое). В документации конкретной CI-системы или анализатора обычно легко найти, как добавить плашку в Markdown. Это поможет посетителям сразу посмотреть статус кода.
11. Напишите в README.md пару абзацев текста, про что вообще проект. Помните, что код вы пишете не только для себя, в ваш репозиторий придут люди, которые вообще не имеют идей, о чём это.
12. Опишите типичный пример использования, если уместно, с картинками или gif-ками. Включая информацию, откуда брать датасеты, куда подкладывать конфигурацию и т.п., чтобы любой пользователь мог с чистого листа запустить проект и понять, что у него получилось.
13. Опишите также действия по сборке и внешние зависимости (версию используемых SDK и т.п.). Это всё есть в CI, но в README это всё должно быть в удобной человеческой форме и заодно приводить к развёртыванию окружения, пригодного для работы над проектом (тогда как сборка в CI может быть весьма хитрой, использовать несколько Docker-образов и т.п.).
14. Если проект предполагает возможность стороннего участия (то есть имеет хоть один шанс стать знаменитым), опишите, как сторонний человек может вам помочь:
  - куда и как писать баги;
  - как связаться с разработчиками;
  - как контрибьютить;
  - где посмотреть техническую документацию и найти первый вводный баг, который можно поправить.
15. Проверьте, что код в репозитории адекватно оформлен. Если на Python, то PEP-8, если на C++, то в соответствии с Core Guidelines и т.п. — у каждого языка и даже у некоторых фреймворков есть свой стиль кодирования, проверьте, что код его уважает. Если в проекте используется свой стиль кодирования, он должен быть явно задокументирован и весь код должен ему соответствовать.
16. Где-то должно быть некое техническое описание проекта — из каких компонентов он состоит, кто за что отвечает. В идеале — полноценная архитектурная документация в виде страниц на вики, с UML-диаграммами, но если сил нет, можно ограничиться разделом в README, где кратко словами всё описать.
17. В коде должны быть комментарии (в принятом для языка формате — DocString, Doxygen, Javadoc, XML Documentation и т.п.), хотя бы у ключевых классов/интерфейсов/модулей, кратко описывающие, что вообще делает класс. В идеале — для всего, что public, с документированием предположений о входных данных, инвариантах, бросаемых исключений и свойств потокобезопасности (reentrant, thread-safe и т.п.), но насколько сил хватит.

18. Комментарии к коммитам пост-фактум исправить тяжело, поэтому за ними надо следить изначально. Рекомендуется следовать соглашению Conventional Commits. Если у вас в репозитории больше пяти маловменяемых комментариев (типа «fix») подряд, лучше либо сделайте squash и склейте коммиты в один, либо измените всю историю через git rebase -i. Коммиты не должны быть сделаны в последний день, а должны показывать, как шла работа, от создания пустого проекта до последнего релиза. Фразы вида «я тут локально разрабатывал, потом выложил, как получилось что-то разумное» очень сильно огорчают комиссию. Могут помочь инструменты типа Mergeable, Mergify.

## Чеклист по оформлению пуллреквеста

1	Адекватное название	
2	Описание пуллреквеста, список предлагаемых изменений (если применимо, со ссылками на закрытые issues)	
3	Описание пуллреквеста, описание работы предлагаемой функциональности (если возможно, с gif-кой демо)	
4	Описание пуллреквеста, техническое описание, изменения в архитектуре	
5	Модульные тесты на новый код	
6	CI проходит, при необходимости подредактирован, чтобы запускать новый код	
7	Содержимое пуллреквеста лицензионно совместимо с основным репозиторием	
8	Коммиты в истории следуют принятым в проекте соглашениям (отдельные или склеены в один)	
9	Стиль кодирования соответствует стилю проекта	

## Полезные ссылки для любопытных

- Правильное именование коммитов: <https://www.conventionalcommits.org/en/v1.0.0/>.
- Исправление комментариев к коммитам, редактирование истории (не делайте так без крайней нужды): <https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History>.
- Хорошее описание .gitignore: <https://www.atlassian.com/ru/git/tutorials/saving-changes/gitignore>.
- Модели работы с ветками:
  - <https://www.atlassian.com/ru/git/tutorials/comparing-workflows>;
  - <https://www.gitkraken.com/learn/git/best-practices/git-branch-strategy>;
  - <https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow>.

- Зачем нужен README.md и как его писать: <https://bulldogjob.com/readme/how-to-write-a-good-readme-for-your-github-project>.
- Шаблон README с плашками и другими хорошими вещами: <https://github.com/othneildrew/Best-README-Template>.
- Подборка примеров хороших README: <https://github.com/matiassingers/awesome-readme>.
- Генератор README: <https://readme.so/ru>.
- Гайд по лицензиям от GitHub: <https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository>.
- Слайды лекции про авторские права в IT от Я.А. Кириленко: [https://docs.google.com/presentation/d/1-xMvM\\_EyouDM9s1BpQHR3h7MRgYrMQCKZ2dxYeFKh20](https://docs.google.com/presentation/d/1-xMvM_EyouDM9s1BpQHR3h7MRgYrMQCKZ2dxYeFKh20).
- Может быть полезен dependabot (<https://docs.github.com/en/code-security/dependabot>), для автоматического обновления внешних зависимостей в проекте.