Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Уральский федеральный университет имени первого Президента России
Б. Н. Ельцина»

УТВЕРЖДАЮ

Директор по образовательной деятельности

_____ С.Т. Князев

«14» декабря 2021 г.

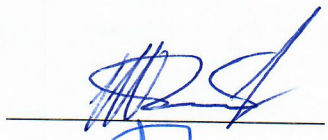
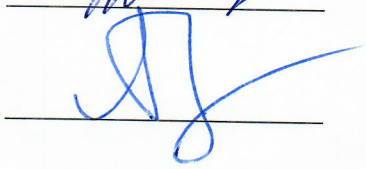
Программная инженерия

Учебно-методические материалы по направлению подготовки
09.04.01 Информатика и вычислительная техника
Образовательная программа «Инженерия искусственного интеллекта»

Екатеринбург

2021

РАЗРАБОТЧИКИ УЧЕБНО-МЕТОДИЧЕСКИХ МАТЕРИАЛОВ

Доцент, канд.техн.наук		Обабков Илья Николаевич
Доцент, канд.техн.наук		Созыкин Андрей Владимирович

СОДЕРЖАНИЕ

1 семестр

Модуль № 1
Название: Программная инженерия для приложений искусственного интеллекта

Образовательные результаты:

- Студент знает особенность дисциплины Программной инженерии, ее структуру.
- Студент знает, как устроен жизненный цикл ИИ-приложения.
- Студент может использовать готовые модели для решения практических задач.
- Студент умеет разрабатывать программное обеспечение в команде.

В этом модуле:

В настоящее время искусственный интеллект перестал казаться чем-то фантастическим. Мы постоянно сталкиваемся с системами искусственного интеллекта: от уже ставших привычных голосовых помощников и ассистентов, автоматических переводчиков, чат-ботов, до беспилотных такси, тестирование которых на дорогах Москвы Яндекс начинает осенью 2021 года, и системы оплаты проезда с использованием распознавания лиц, которая внедряется в Московском метро.

Большая часть систем искусственного интеллекта использует машинное обучение. Однако для успешной работы таких систем недостаточно просто обучить высокоточную модель машинного обучения. Необходимо разработать программное обеспечение, которое запускает модель на различных устройствах: компьютерах, смартфонах, серверах в облаке, встраиваемых системах и т.п. Такое программное обеспечение должно быть отказоустойчивым и высокопроизводительным, в противном случае его использование может привести к катастрофам с человеческими жертвами (например, в случае беспилотного автомобиля). Также программное обеспечение должно иметь возможность обслуживать тысячи или миллионы пользователей одновременно, например, в облачных системах автоматического перевода.

Этот модуль посвящен рассмотрению основ программной инженерии – дисциплины, изучающей приложение инженерного подхода к разработке

программного обеспечения с целью создания надежных, высокопроизводительных и отказоустойчивых программных систем.

В модуле вы:

- Познакомитесь с предметной областью программной инженерии.
- Научитесь использовать готовые модели машинного обучения в программных системах.
- Изучите основы командной разработки.

Модуль № 1. Юнит № 1. Организация курса

В курсе “Программная инженерия” вы познакомитесь с основными инструментами, которые применяются для создания крупных программных систем. Курс будет длиться два семестра. В первом семестре вы изучите базовые основы программной инженерии, необходимые для командной разработки приложений искусственного интеллекта, а во втором: продвинутые инструменты, которые помогут сделать вашу работу более эффективной.

Первый семестр курса включает 5 недель, каждая из которых посвящена изучению отдельного раздела программной инженерии.

В курсе будет использоваться язык Python и библиотеки для этого языка.

Неделя 1. Программная инженерия для приложений искусственного интеллекта

Вы узнаете, что представляет собой программная инженерия как дисциплина, какие особенности есть в проектах по созданию крупных приложений искусственного интеллекта. Также вы узнаете, как разрабатывать приложения искусственного интеллекта на основе готовых библиотек машинного обучения. Мы подробно рассмотрим популярные Python библиотеки Hugging Face и TensorFlow.

По настоящему большую систему искусственного интеллекта невозможно создать в одиночку. Поэтому на первой неделе мы рассмотрим командную разработку программного обеспечения и изучим наиболее популярный инструмент командной разработки – систему git и облачный сервис GitHub.

Неделя 2. Архитектура приложений искусственного интеллекта

Вторая неделя посвящена важной теме: как организованы приложения искусственного интеллекта. Вы изучите наиболее популярные подходы к архитектуре программного обеспечения и особенности их применения для приложений искусственного интеллекта.

Наибольшей популярностью сейчас пользуется микросервисная архитектура, в которой приложение строится из набора небольших сервисов, взаимодействующих по сети между собой. Каждый такой сервис предоставляет API (application programming interface) – набор функций, доступных внешним разработчикам. Мы рассмотрим инструменты работы с API, а также научимся использовать API популярных сетевых сервисов: социальные сети, крупные сетевые порталы и т.п.

Неделя 3. Разработка API для приложений искусственного интеллекта

На третьей неделе мы продолжим изучать архитектуру приложений искусственного интеллекта и API. Вы узнаете, как создать API для вашей модели машинного обучения и организовать к ней доступ по сети. В качестве инструмента для создания API мы будем использовать популярную систему FastAPI на Python.

Неделя 4. Развертывание приложений искусственного интеллекта в облаке

На четвертой неделе курса мы рассмотрим, как обеспечить доступ другим разработчикам к вашей модели машинного обучения по сети. Для этого нужно развернуть приложение, реализующее созданный вами API на каком-либо сервере, подключенном к сети. Сейчас для этой цели чаще всего используют облачные платформы.

Мы рассмотрим облачную платформу Heroku, научимся разворачивать на этой платформе приложение на Python, а также рассмотрим интеграцию Heroku с GitHub для автоматического развертывания приложений.

Неделя 5. Тестирование программного обеспечения

Тестирование – обязательная часть процесса промышленной разработки программного обеспечения. Приложения искусственного интеллекта также невозможно разработать без качественного тестирования. В противном случае пользователи вашего приложения будут получать ошибки и жаловаться вам. Если ошибок будет много, то пользователи могут совсем отказаться от вашего приложения.

Тестирование приложений на Python мы рассмотрим на пятой неделе. Также вы познакомитесь с концепцией программной инженерии Continuous Integration (CI, непрерывная интеграция) и узнаете, как она позволяет повысить качество программного обеспечения.

Проект

Обучение на курсе организовано на основе проекта, который вы будете делать командами из трех-четырех человек.

Цель проекта: развернуть в облаке модель машинного обучения и обеспечить к ней доступ через API. Модель вы можете выбрать сами. Мы рекомендуем использовать готовую модель из библиотек машинного обучения.

Проект мы будем реализовывать по шагам в течение всего курса. На первой неделе вы познакомитесь с готовыми библиотеками машинного обучения и сможете выбрать модель, для которой хотите реализовать приложение. Вы создадите репозиторий на GitHub, чтобы работать над приложением могли все участники вашей команды. На второй и третьей неделе вы создадите API для выбранной вами модели. На четвертой неделе настроите автоматическое разворачивание кода, реализующего API для вашей модели, из GitHub репозитория на облачную платформу Heroku. И, наконец, на пятой неделе вы разработаете тесты для своего приложения магинного обучения и настроите автоматический запуск тестов с помощью Continuous Integration на GitHub перед разворачиванием приложения на Heroku. Таким образом вы сможете быть уверены, что разворачиваемое приложение будет работать без ошибок.

Экзамен по курсу будет проходить в виде защиты проекта.

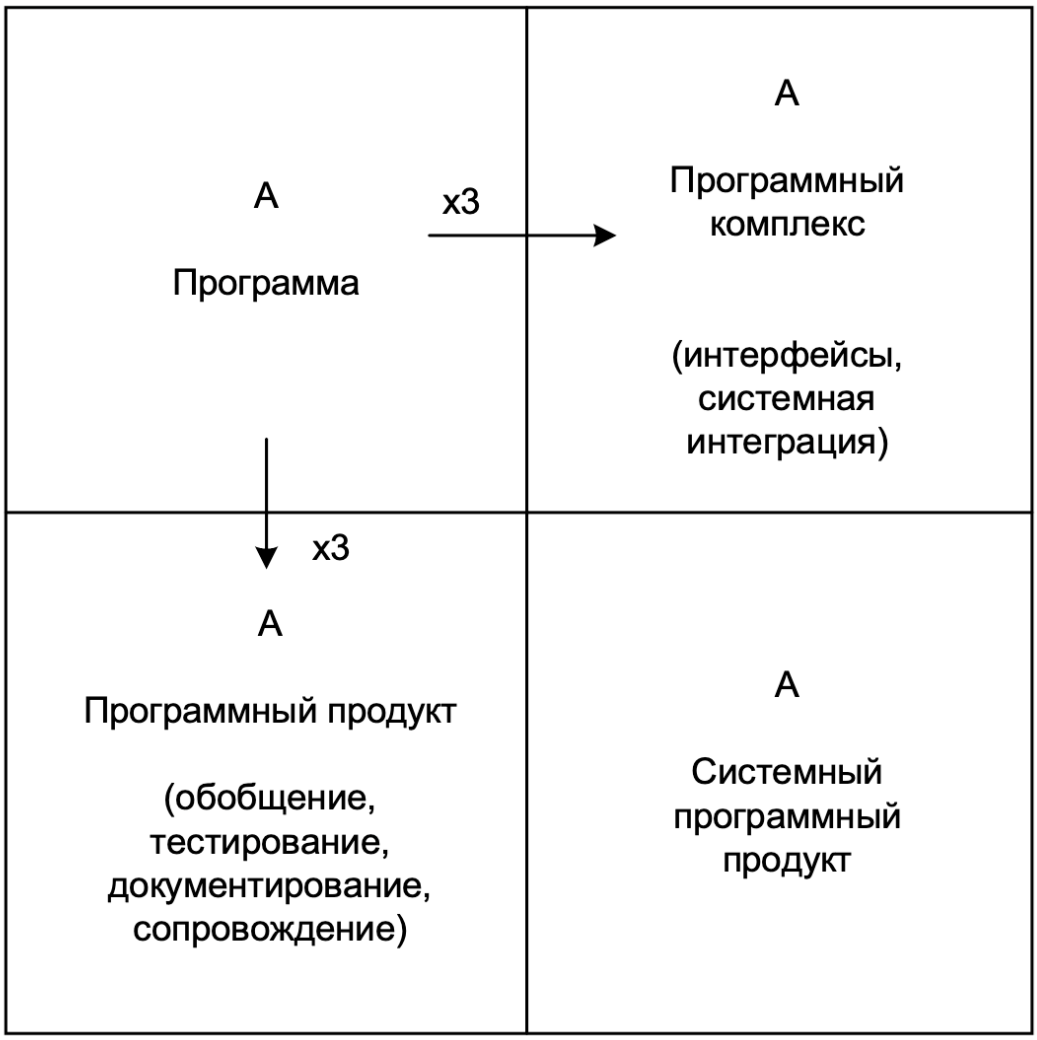
Итоги

- Курс состоит из пяти недель, во время которых вы будете изучать различные инструменты программной инженерии, помогающие создавать крупные приложения искусственного интеллекта.
- Обучение на курсе организовано на основе проекта, который вы будете выполнять по шагам на каждой неделе курса.
- Для получения оценки за курс необходимо реализовать и защитить проект.

Модуль № 1. Юнит № 2. Программная инженерия как дисциплина

Приложения искусственного интеллекта, такие как система FacePay оплаты проезда с использованием распознавания лиц в Московском метро, голосовой помощник Алиса от Яндекса или автоматический переводчик от компании Google – это крупные программные системы, которыми пользуются тысячи или миллионы людей. Создают такие системы десятки или сотни разработчиков. Как показывает опыт, разработка крупных программных систем сильно отличается от разработки небольших программных продуктов, таких как мобильные приложения или сайты. В результате, крупные программные системы создаются долго, стоят дорого и часто работают нестабильно. Более того, некоторые крупные программные системы вообще не удается довести до работоспособного состояния.

Объяснение такой ситуации с разработкой крупных программных систем дает Фредерик Брукс в книге “Мифический человеко-месяц”. Разработку программной системы он представляет на схеме в виде квадрата, разделенного на четыре части.



Системный программный продукт по Фредерику Бруксу.

В левой части квадрата находится программа – реализация приложения, которое работает на компьютере разработчика и использовать которое может только его создатель. Например, разработчик может обучить модель на основе нейронной сети определять эмоциональную окраску текста: положительную или отрицательную. Как правило, для создания таких программ используется Jupyter Notebook. Код в ноутбуке обычно содержит обучение модели и несколько примеров ее применения для определения тональности текстовых вариантов текстов. Однако использовать такой код и обученную с его помощью модель вряд ли сможет кто-то, кроме авторов.

Программный продукт

Одна из осей изменений на схеме – это создание программного продукта на основе программы. Ключевое отличие программного продукта – это то, что его могут использовать сторонние пользователи, а не только авторы

программы. Чтобы такое стало возможным, необходимо создать обобщенную версию программы, которая работает на разных платформах и может обрабатывать данные в разных форматах.

Программу на пути к программному продукту необходимо тщательно протестировать, опять же на различных платформах и с различными форматами данных, а также для различных вариантов применения, которые нужны пользователям.

Разработчики программы знают, как ее использовать, но сторонние пользователи не имеют об этом никакого представления. Поэтому для программного продукта необходимо разработать документацию. Чем выше качество документации, тем больше пользователей будет у вашего продукта. И наоборот, плохая документация способна оттолкнуть пользователей даже от хороших программ.

Программный продукт нуждается в сопровождении и технической поддержке. К сожалению, ошибки в программном обеспечении неизбежны и не всегда их удается найти с помощью тестирования. Пользователи программного продукта будут сталкиваться с ошибками и разработчикам необходимо их исправлять.

Чтобы программа с моделью машинного обучения стала программным продуктом, необходимо реализовать приложение, использующее такую модель. Например, на основе модели определения эмоциональной окраски текста может быть создано мобильное приложение, приложение в облаке или на другой платформе. Для приложения каждого типа необходимо написать документацию, протестировать и обеспечить поддержку.

По оценке Фредерика Брукса, создание программного продукта требует минимум в три раза больше времени и денег, чем программы.

Программный комплекс

Второе направление изменений – это включение программы в программный комплекс: множество программ, которые взаимодействуют между собой для решения какой-либо задачи.

Рассмотренная ранее нами программа определения эмоциональной окраски текста сама по себе вряд ли принесет большую пользу. Но ее можно применить в составе программного комплекса крупной компании, которая использует информацию из социальных сетей в процессе создания новых продуктов. Первая часть такого комплекса собирает в социальных сетях отзывы на существующие и создаваемые продукты компании. Второй частью может стать наша программ определения эмоциональной окраски отзывов. И третья часть – система аналитики, которая использует собранную информацию об эмоциональной окраске для принятия решений о направлениях разработки продуктов.

Однако, чтобы несколько программ могли работать совместно в составе комплекса, необходимо провести работу по их интеграции: определиться с форматами данных которые передаются от одной программы к другой, задать интерфейсы программирования API, определить протоколы, по которым будут передаваться данные.

Создание программного комплекса, также как и программного продукта, по оценке Фредерика Брукса требует минимум в три раза больше времени и денег, чем создание программы.

Системный программный продукт

Наиболее совершенной программной системой, согласно Фредерику Бруксу, является системный программный продукт. Он представляет собой программный комплекс, каждой частью которого является программный продукт: обобщенные, протестированные и документированные программы, для которых реализуется сопровождение.

Система проектирования продуктов с использованием аналитики отзывов пользователей в социальных сетях вполне может быть системным программным продуктом, если каждый ее компонент не разрабатывается самостоятельно, а используется готовый программный продукт.

Создание системного программного продукта требует минимум в 9 раз больших затрат времени и денег, чем разработка программы. Однако именно системные программные продукты наиболее полезны пользователям, т.к. они позволяют решать их задачи в комплексе. Большая часть практических приложений искусственного интеллекта, которые создаются сейчас и будут

создаваться в ближайшие несколько лет являются системными программными продуктами.

Программная инженерия

Программная инженерия рассматривает применения инженерных методов для создания сложных программных систем. Фредерик Брукс в “Мифическом человеко-месяце” утверждает, что программирование занимает всего ⅙ часть времени создания сложной программной системы. Остальное время занимают другие активности, которые как раз и изучает программная инженерия.

Международная организация Институт инженеров электротехники и электроники (IEEE, Institute of Electrical and Electronics Engineers) разработала рекомендации по набору знаний специалистов по программной инженерии: Software Engineering Body of Knowledge (SWEBOK) (https://www.computer.org/education/bodies-of-knowledge/software-engineering). Основные разделы SWEBOK, которые необходимо изучать программным инженерам:

- Требования к программному обеспечению
- Проектирование программного обеспечения
- Тестирование программного обеспечения
- Поддержка программного обеспечения
- Управление конфигурацией программного обеспечения
- Менеджмент в программной инженерии
- Процессы в программной инженерии
- Модели и методы программной инженерии
- Качество программного обеспечения
- Профессиональные практики программной инженерии
- Экономика программной инженерии

Системное применение инструментов программной инженерии позволит вам успешно создавать, выводить в продуктивную эксплуатация и поддерживать сложные системы искусственного интеллекта.

В этом курсе мы рассмотрим вопросы проектирования и тестирования программного обеспечения, процессы в программной инженерии, качество программного обеспечения и некоторые профессиональные практики программной инженерии. Управление конфигурацией программного

обеспечения с использованием подхода Infrastructure as a Code будет рассмотрено в курсе “Автоматизация администрирования DevOps”. Менеджмент и экономика программной инженерии будут рассмотрены в курсе “Управление проектами машинного обучения”.

Итоги

- Прикладные системы искусственного интеллекта являются системными программными комплексами – сложными программными системами.
- Затраты времени и денег на создание системных программных комплексов минимум в 9 раз превышают затраты на создание простых программ.
- Программирование в процессе создания программного комплекса занимает ориентировочно ⅙ времени. Остальное время уходит на проектирование, тестирование, документирование, интеграцию, сопровождение и другие активности.
- Программная инженерия изучает применение инструментов инженерии для создания крупных программных систем, таких как системные программные продукты.

Тест

Чем программный продукт отличается от программы

1. Программный продукт может использовать только его разработчик.
2. **Программный продукт протестирован, документирован, может работать на разных платформах с данными разных типов.**
3. Программный продукт использует согласованные с другими программами типы данных и протоколы.
4. Разработка программного продукта в три раза дешевле, чем разработка программы.

Во сколько раз затраты на разработку системного программного продукта превышают затраты на разработку программы

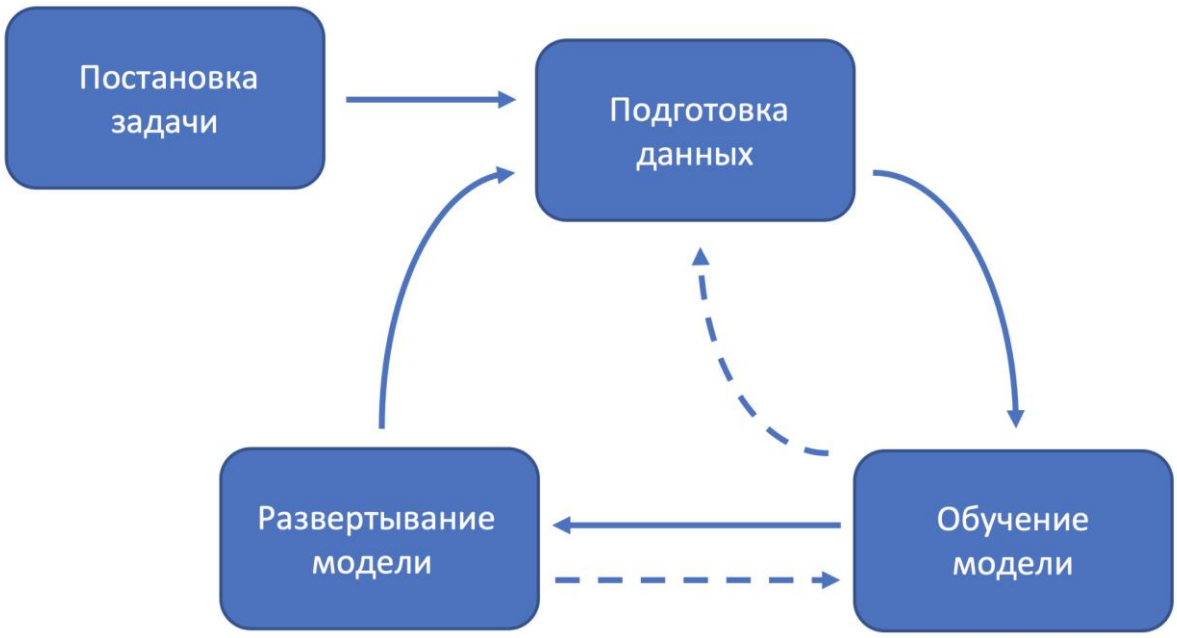
1. Минимум в 3 раза.
2. Минимум в ⅙ раза.
3. **Минимум в 9 раз.**
4. Более чем в ⅚ раза.

Что изучает программная инженерия

1. **Применения инженерных методов для создания сложных программных систем.**
2. Создание приложений искусственного интеллекта с помощью системного подхода.
3. Математические основы сложности алгоритмов программного обеспечения.
4. Алгоритмы машинного обучения.

Модуль № 1. Юнит № 3. Жизненный цикл разработки приложений машинного обучения

Создание систем машинного обучения – это итеративный процесс, который состоит из нескольких этапов. Эндрю Ын (Andrew Ng), один из основателей Coursera, признанный специалист в области машинного обучения, предлагает следующий жизненный цикл разработки систем машинного обучения (https://www.coursera.org/specializations/machine-learning-engineering-for-production-mlops):



Жизненный цикл разработки приложения машинного обучения от Эндрю Ын

Постановка задачи

Работа над созданием системы машинного обучения начинается с постановки задачи. На этом этапе определяется цель проекта, над которым предстоит работать например, создание системы автоматического перевода. Задаются архитектурные требования к системе: должно ли это быть мобильное приложение, Web-приложение, или API, предназначенный для вызова другими приложениями. Необходимо определить и содержательные требования к приложению: продолжая рассматривать пример системы автоматического перевода мы можем решить, что наша система будет переводить с английского языка на русский и наоборот.

Также на этапе постановки задачи необходимо сформулировать ожидаемые требования к качеству работы системы машинного обучения. Для оценки качества перевода можно использовать метрику BLEU (bilingual evaluation understudy), желаемые значения должны быть в диапазоне от 50 до 60 (что соответствует уровню fluent translation) (https://cloud.google.com/translate/automl/docs/evaluate).

Подготовка данных

Следующий этап – это подготовка данных, на котором необходимо сформировать набор данных и произвести его разметку. Для системы автоматического перевода набор данных будет включать предложения на русском языке и соответствующий им перевод предложения на английский язык. Набор данных можно подготовить самостоятельно или использовать один из существующих.

Обучение модели

На третьем этапе жизненного цикла производится работа с моделью машинного обучения. Нужно выбрать тип модели, который будет использоваться, например, глубокие нейронные сети или какая-то из классических моделей машинного обучения. Затем выполняется обучение модели с использованием подготовленного на предыдущем этапе набора данных. Последний шаг – оценка качества работы обученной модели. Для системы автоматического перевода мы можем выбрать в качестве модели машинного обучения глубокую нейронную сеть на основе BERT, дообучить ее на подготовленном наборе данных с русскими и английскими предложениями и затем оценить качество работы обученной сети с помощью метрики BLEU.

Если не получится добиться минимального значения качества перевода (50 по метрике BLEU), то возможна еще одна итерация с выбором модели нового типа, ее последующим обучением и оценкой качества работы. Также возможен возврат на предыдущий этап к работе с набором данных, т.к. во многом качество обучения модели зависит от качества набора данных, который используется для обучения.

Развертывание модели

Четвертый этап – это развертывание модели для практического применения. Именно на этом этапе создается приложение, которое использует модель машинного обучения, созданную на предыдущем этапе, и организуется доступ пользователей к приложению.

Вопреки распространенному мнению, на этапе развертывания разработка системы машинного обучения не заканчивается. Часто бывает, что данные от пользователей существенно отличаются от тех, которые мы применяли для обучения модели. Например, пользователи могут попытаться автоматически переводить сообщения из соцсетей, в то время как наша система обучалась на литературных текстах или статьях с Wikipedia. В результате у системы могут возникнуть проблемы при переводе слэнга, предложений с плохой грамматикой и эмоджи. Такое явление называется сдвиг данных (data shift).

Проблема сдвига данных часто встречается на практике. Компания Яндекс сейчас проводит соревнования Shifts Challenge (https://research.yandex.com/shifts), в рамках которых открыла доступ к трем большим набором данных: обучения систем управления автомобилей без водителя, автоматического перевода и прогноза погоды. Цель этих соревнований: разработать методы и технологии обеспечения качественной работы моделей даже в условиях сдвига данных.

После развертывания и запуска в продуктивное использование важно организовать мониторинг качества работы модели с данными от пользователей. Если качество работы начинает снижаться, то необходимо переобучение модели на новых данных. Для этого запускается еще одна итерация жизненного цикла: данные от пользователей добавляются в подготовленный ранее набор данных (этап подготовки данных), затем

модель обучается на новых данных (этап обучения модели), и новая модель развертывается. Так как данные от пользователей постоянно меняются, то и в жизненном цикле большинства приложений машинного обучения этапы подготовки набора данных, обучения модели, развертывание модели для продуктивного использования будут повторятся несколько раз.

Иногда бывает, что после запуска модели в продуктивное использование приходится не просто переобучать модель на новых данных, а полностью менять модель на более подходящую к данным пользователей. В этом случае с этапа развертывания происходит возврат к этапу обучения модели.

Итоги

- Жизненный цикл разработки приложений машинного обучения включает этапы постановки задачи, подготовки данных, обучение модели и развертывание модели для практического использования.
- Жизненный цикл разработки приложений машинного обучения – итеративный процесс. С каждого этапа возможен возврат на предыдущие.
- Качество работы развернутой модели со временем падает из-за сдвига данных. Поэтому цикл подготовка данных–обучение модели–развертывание модели необходимо повторять через определенные промежутки времени (от нескольких недель до нескольких месяцев).

Тест

Какие шаги включает жизненный цикл разработки приложений машинного обучения?

1. Проектирование, разработка, тестирование, развертывание.
2. **Постановка задачи, подготовка данных, обучение модели, развертывание модели.**
3. Сбор данных, очистка данных, обучение модели, оценка качества модели.
4. Проектирование, разработка, сопровождение, вывод из эксплуатации.

Что такое сдвиг данных

1. Запись данных в неправильные столбцы таблицы.
2. Переиспользование данных при обучении и тестировании модели.
3. **Отличие данных, которые поступают от пользователей от данных, на которых обучалась модель.**

4. Подготовка данных для модели машинного обучения нового типа.

Для какой цели используется мониторинг качества работы модели на данных пользователей.

1. Чтобы обнаружить выбросы в данных и заблокировать их.
2. Чтобы подобрать алгоритмы правильной подготовки данных для модели.
3. Чтобы обнаружить повышение качества работы модели в связи с поступлением новых данных от пользователей.
4. **Чтобы обнаружить снижение качества работы модели из-за сдвига данных.**

Модуль № 1. Юнит № 4. Определение тональности текстов

Давайте рассмотрим, как можно разрабатывать приложения искусственного интеллекта на примере системы определения тональности отзывов на тексты.

Полный жизненный цикл создания такой системы будет включать следующие этапы:

1. **Постановка задачи.** Предположим, что наша цель: научиться определять тональность отзывов в социальных сетях на продукты, которые производит наша компания. Эту информацию мы планируем использовать дальше для проектирования продуктов компании. В качестве метрики качества работы модели будем использовать долю правильных ответов, желаемая величина: минимум 95%
2. **Подготовка набора данных.** На этом этапе нам необходимо собрать в социальных сетях тексты отзывов на продукты нашей компании, или похожие по смыслу и эмоциональной окраске тексты. Затем нужно выполнить разметку: для каждого отзыва указать, положительный он, или отрицательный.
3. **Обучение модели**. Лучшие результаты анализа текстов сейчас дают нейронные сети. Поэтому нам необходимо обучить нейронную сеть с помощью набора данных, который мы подготовили на предыдущем этапе.
4. **Развертывание модели.** На этом этапе создается приложение, использующее полученную модель для определения эмоциональной окраски текстов, и разворачивается, например, в облаке.

Использование готовых моделей машинного обучения

Однако на практике не всегда необходимо создавать модели машинного обучения с нуля. Существует большое количество библиотек машинного обучения, содержащих готовые к использованию модели машинного обучения, обученные на популярных наборах данных. В этом случае можно пропустить этапы подготовки набора данных и обучения модели, и сразу перейти к разработке приложения, предназначенного для развертывания.

Для определения тональности текстов удобнее всего использовать библиотеку Hugging Face, т.к. она автоматизирует большое количество этапов обработки текста и позволяет создавать программы небольшого размера. Вот пример программы определения тональности текста с помощью библиотеки Hugging Face:

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis",
                      "blanchefort/rubert-base-cased-sentiment")

classifier("Я обожаю инженерию машинного обучения!")
```

В первой строке мы подключаем класс pipeline из модуля transformers библиотеки Hugging Face. Пайплайны в машинном обучении служат для автоматизации процесса обработки данных. Пайплайны в Hugging Face позволяют применять модели машинного обучения для обработки текстов с использованием нескольких строк кода.

Во второй строке мы создаем пайплайн, использую конструктор класса pipeline. При создании пайплайна в Hugging Face указывается:

- Тип пайплайна – название задачи машинного обучения, реализация которой автоматизируется. В нашем примере название задачи: sentiment-analysis, анализ эмоциональной окраски текстов.
- Модель машинного обучения, которая будет использоваться в пайплайне для решения нужной нам задачи. Название модели: blanchefort/rubert-base-cased-sentiment, описание модели можно посмотреть на сайте библиотеки Hugging Face по ссылке –

https://huggingface.co/blanchefort/rubert-base-cased-sentiment. Модель использует одну из наиболее популярных в настоящее время архитектур нейросетей для обработки текстов BERT. Модель дообучена определять тональность текстов на русском языке на составном объеме данных, включающем 351.797 текстов.

Созданный пайплайн записывается в переменную classifier, которую можно использовать для определения тональности текста.

В третьей строке мы используем пайплайн в переменной classifier, чтобы определить тональность текста "Я обожаю инженерию машинного обучения!". После запуска получаем следующий результат:

```
[{'label': 'POSITIVE', 'score': 0.9807393550872803}]
```

Это список, который содержит столько же элементов, сколько текстов было передано в пайплайн (в нашем случае один элемент). Каждый элемент списка на выходе из пайплайна содержит словарь. Элементы в словаре зависят от того, для какой задачи применяется пайплайн. В задаче определения тональности словарь на выходе из пайплайна содержит два элемента:

- 'label' – метка результата распознавания тональности. Чаще всего используются значения 'POSITIVE' – позитивная тональность и 'NEGATIVE' – негативная тональность. Иногда могут встречаться другие метки, например, 'NEUTRAL' – нейтральная тональность.
- 'score' – значение на выходе из модели. Для задачи определения тональности на выходе из модели выдается число в диапазоне от 0 до 1. Значение 0 означает максимальную уверенность, что тональность отрицательная, значение 1 – максимальную уверенность, что тональность положительная.

В нашем примере элемента 'label' содержит значение 'POSITIVE' – эмоциональная окраска текста положительная. Значение элемента 'score' 0.9807393550872803 близко к 1, значит, модель имеет высокую уверенность в положительной эмоциональной окраске.

Таким образом, с помощью трех строк кода нам удалось решить достаточно сложную задачу машинного обучения: определения эмоциональной окраски текстов на русском языке. Это стало возможным благодаря наличию готовых библиотек машинного обучения.

Библиотека Hugging Face является примером системного программного продукта, в котором интегрируется несколько программных продуктов: фреймворки машинного обучения TensorFlow и PyTorch, библиотеки обработки данных на Python (в первую очередь текстов), предварительно обученные модели машинного обучения и т.п. Создание системного программного продукта является трудозатратным процессом, но после создания на его основе можно быстро разрабатывать приложения. Что мы и сделали на примере приложения для определения эмоциональной окраски текста.

Библиотеки машинного обучения с предварительно обученными моделями

Hugging Face не единственная библиотека машинного обучения, которая содержит готовые к использованию модели. Вот еще несколько библиотек:

- TensorFlow Hub (https://www.tensorflow.org/hub) – Репозиторий с обученными моделями для библиотеки TensorFlow.
- PyTorch Hub (https://pytorch.org/hub/) – Репозиторий с обученными моделями для библиотеки PyTorch.
- Keras Applications (https://keras.io/api/applications/) – Предварительно обученные модели в библиотеке Keras. Преимущественно модели компьютерного зрения.

Итоги

- Использование предварительно обученных моделей из готовых библиотек машинного обучения позволяет существенно ускорить разработку приложений искусственного интеллекта.
- Пайплайны в машинном обучении применяются для автоматизации комплексных процессов обработки данных, состоящих из нескольких этапов.
- Библиотека Hugging Face – пример системного программного продукта, интегрирующего различные программные продукты: библиотеки машинного обучения, обработки данных, обученные модели машинного обучения и т.п.
- Рекомендуется изучить и использовать в реализации проекта одну из следующих библиотек: Hugging Face, TensorFlow Hub, PyTorch Hub, Keras Applications.

Практическое задание

Поэкспериментируйте с определением тональности текстов на русском языке с помощью библиотеки Hugging Face с использованием Colab-ноутбука – https://colab.research.google.com/drive/136RKDIVzzLkMsiO8H6VRMYO1h0ZsdIgv?usp=sharing

Модуль № 1. Юнит № 5. Выбор модели машинного обучения для решения задачи

Видео – https://www.dropbox.com/s/rbp4arlkuavn859/hf_models.mp4?dl=0

Практическое задание

1. Поэкспериментируйте с ноутбуками из видео:
   - Автоматический перевод – https://colab.research.google.com/drive/1MvqqKTOFqMepC1VVEiCxOuhggHQx2a4q?usp=sharing
   - Генерация аннотаций новостей – https://colab.research.google.com/drive/1P1tAK15CIYFd7WFvGN6ImBD-sQZXUuRi?usp=sharing
2. Найдите на сайте Hugging Face модель для решения задачи по вашему выбору и создайте приложение, использующее эту модель. Используйте документацию на Pipeline в библиотеке Hugging Face – https://huggingface.co/transformers/main_classes/pipelines.html

Модуль № 1. Юнит № 6. Командная разработка

Крупные программные системы, использующие искусственный интеллект, практически невозможно создать одному человеку. Поэтому важно организовать процесс командной разработки, при котором над кодом программной системы работают несколько человек. В этом разделе вы узнаете, как это делается.

Системы контроля версий

Основной инструмент, который используется для организации командной разработки – это система контроля версий. Она нужна для того, чтобы можно было хранить разные версии файлов и удобно работать с ними. Чаще всего системы контроля версий используются для хранения программного кода, но могут применяться для файлов любых типов, в том числе наборов данных для обучения моделей.

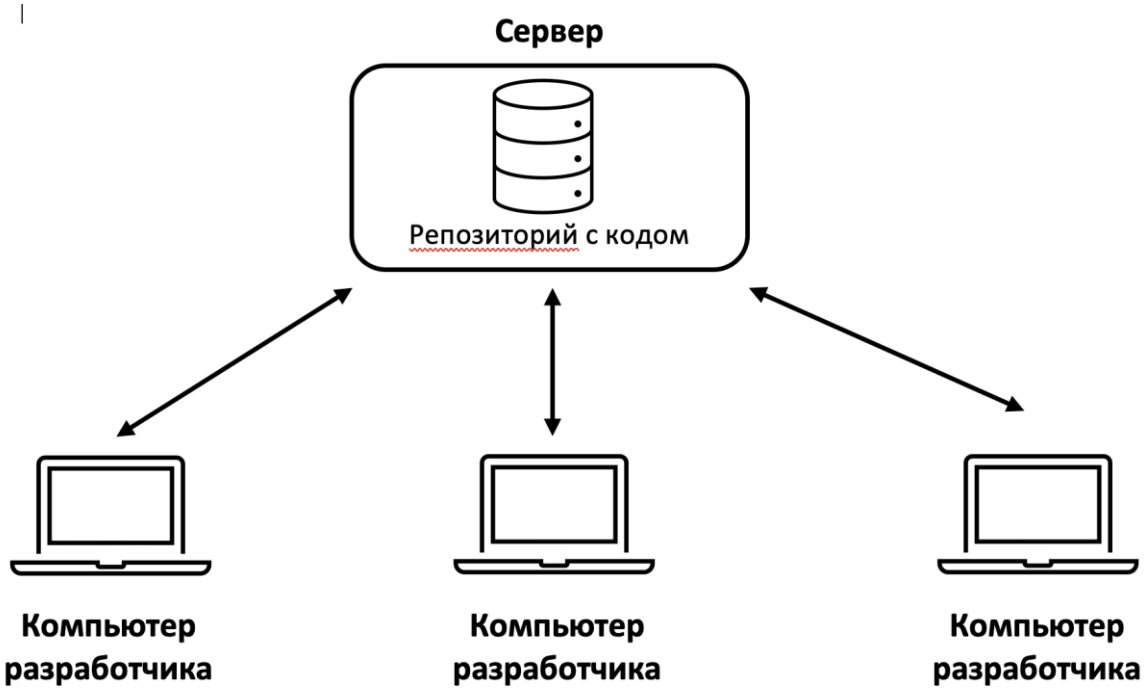
Чем полезны системы контроля версий в командной разработке? Представьте, что один из разработчиков в команде внес изменения в код, но в этих изменениях есть ошибка. В результате программа перестала работать. Решить ситуацию можно двумя способами:

- Найти ошибку в коде и исправить ее. Однако этот процесс может потребовать длительного времени, если ошибка не тривиальная. Особенно сложно искать ошибки в коде, который написал другой человек.
- Использовать одну из предыдущих версий кода, в которой нет ошибки. Такой подход называется откатом изменений и его можно реализовать достаточно быстро. Все, что нужно – это найти версию без ошибки и использовать ее в качестве действующей.

Самый простой вариант системы контроля версий – это каталог с файлами, причем к имени файла дописывается номер версии. Такую систему просто организовать, но с ней неудобно работать. Во-первых, когда файлов в проекте становится много, и у каждого файла несколько версий, может возникнуть путкница. Во-вторых, сложно найти, в какой именно версии файла были произведены интересующие изменения. Например, если мы обнаружили критическую ошибку в коде и хотим сделать откат к предыдущей версии без такой ошибки, то как нам узнать, в каком именно файле такой ошибки нет? Для этого придется последовательно просматривать все версии файла, что долго и не удобно. В третьих, в такой системе сложно организовать командную работу. Конечно, можно сделать так, чтобы каталог был доступен всем разработчикам по сети, но это не очень удобно. Поэтому для командной разработки были созданы специализированные системы контроля версий.

Централизованные системы контроля версий

Первые системы контроля версий, такие как CVS (Concurrent Versions System) и SVN (Subversion), использовали централизованный подход. В этом случае код программной системы храниться в одном месте – в репозитории на сервере. Разработчики подключаются со своих компьютеров к репозиторию и копируют часть данных себе. Такая копия данных из репозитория называется рабочей копией.



Централизованная система контроля версий

Если в команде несколько разработчиков, то каждый создают рабочую копию на своем компьютере. Изменения в код вносятся в локальные рабочии копии компьютеров разработчиков. Поэтому несколько человек могут работать с кодом независимо друг от друга.

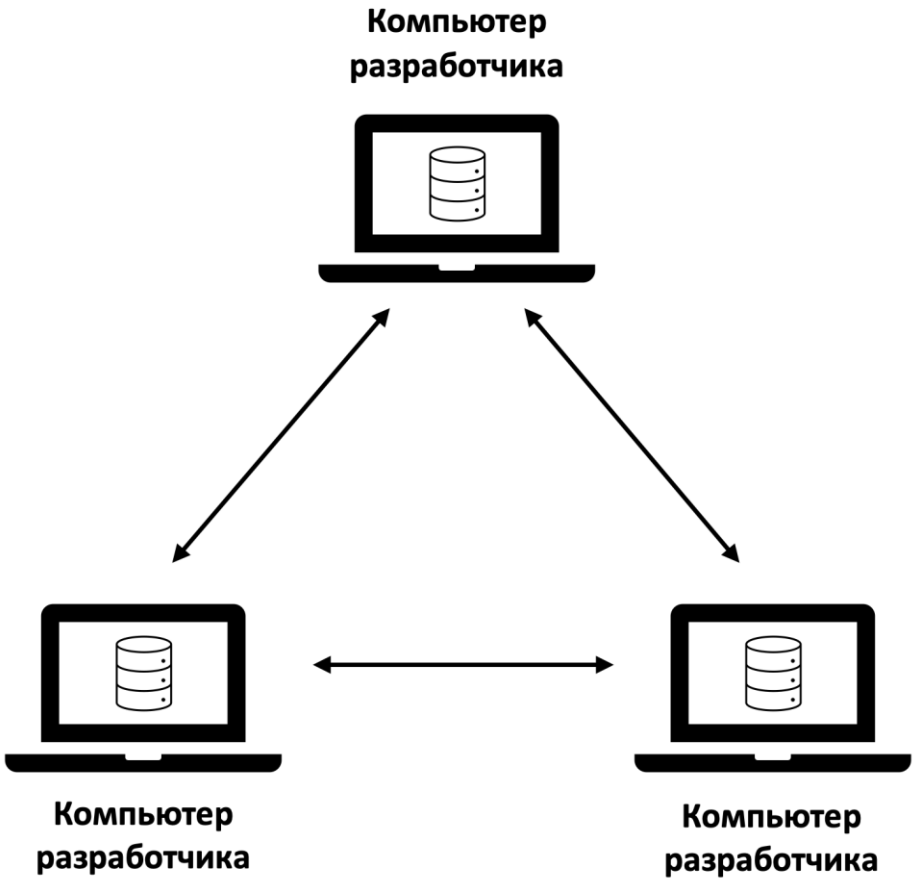
После того, как разработчик реализовал законченную часть изменений в коде и проверил ее работоспособность, изменения необходимо перенести в централизованный репозиторий. Для этого используется функция фиксации изменений (по английски commit) системы контроля версий).

Централизованные системы контроля версий активно использовались начиная с 1990-х годов, но в 2016 году их популярность начала снижаться. Недостатком централизованных систем контроля версий является наличие

единой точки отказа – сервера. В случае, если сервер перестает работать, то клиенты не смогут сохранять изменения и обмениваться этими изменениями с другими разработчиками из команды. А если сервер полностью выйдет из строя, то можно потерять весь репозиторий с кодом. Восстановить данные можно только из резервной копии (если, конечно, она у вас есть). Поэтому централизованные системы контроля версий постепенно были вытеснены распределенными системами контроля версий.

Распределенные системы контроля версий

В распределенных системах контроля версий полная копия репозитория с кодом хранится на каждом компьютере. Поэтому единой точки отказа нет. Если у разработчика пропадет связь с другими репозиториями, то он сможет продолжить работу со своей копией кода. Когда связь восстановится, изменения будут синхронизированы с другими участниками.



Распределенная система контроля версий

Потеря любого компьютера или сервера с репозиторием кода не страшна, т.к. существует несколько других полных копий репозитория.

Распределенная работа делает устройство системы контроля версий сложнее, чем централизованной. Изменения, внесенные на разных компьютерах, могут приводить к конфликтам. Например, два разработчика могут внести изменения в один и тот же файл. Разработчики делают это независимо друг от друга в копии репозитория на своих компьютерах. Но когда распределенная система контроля версий будет объединять изменения, она может не понять, как корректно обработать две разные версии одного файла от двух разработчиков. Иногда в процессе синхронизации изменений может потребоваться вмешательство человека.

В настоящее время самой популярной системой распределенного контроля версий является git (https://git-scm.com/). Именно ее мы и будем изучать в этом курсе, а также использовать в проектах в магистратуре.

Итоги

- Крупные программные системы, использующих искусственный интеллект, невозможно создать в одиночку. Необходима командная работа.
- Основной инструмент командной разработки программного обеспечения – системы контроля версий.
- Существует два типа систем контроля версий: централизованные и распределенные. Сейчас преимущественно используются распределенные системы контроля версий.
- Наиболее популярная в настоящее время распределенная система контроля версий git.

Задания

Тест

Какой тип систем контроля версий сейчас используется чаще всего

1. Локальные
2. **Распределенные**
3. Централизованные
4. Системные

Задание на перетаскивание карточек

Выберите характеристики, соответствующие типу системы контроля версий.

Тип системы контроля версий	Характеристика
Централизованная	● Полный репозиторий с кодом хранится на сервере. ● Сервер – единая точка отказа. ● Работа без соединения с сервером невозможна.
Распределенная	● Полный репозиторий с кодом хранится на каждом компьютере. ● Нет единой точки отказа. ● Возможна работа без соединения с сервером и другими компьютерами. ● Используются сложные алгоритмы синхронизации изменений между копиями репозитория.

Модуль № 1. Юнит № 7. Основы Git

В этом модуле вы познакомитесь с основами использования распределенной системы контроля версий git. Вы изучите команды git, научитесь создавать репозиторий git с кодом программы, отслеживать изменения в нем и синхронизировать изменения с удаленными репозиториями, чтобы ваш код стал доступен другим участникам вашей команды.

Репозиторий git

Репозиторий git – это хранилище кода, изменения в котором отслеживаются и сохраняются в виде версий. Технически репозиторий представляет собой каталог на компьютере. В этом каталоге хранится код программной системы, а также информация об изменениях в коде (версиях).

Работа с git начинается с создания репозитория. **Первый шаг** – это создание каталога в операционной системе, в котором вы хотите создать репозиторий. Например, это может быть каталог /home/user/ml_project. **На втором шаге** вам нужно перейти в интересующий каталог и инициализировать репозиторий git с помощью команды:

```
git init
```

При инициализации git создаст в каталоге подкаталог .git, в котором будет храниться служебная информация. (В Linux точка в начале имени файла говорит о том, что каталог является скрытым. Информация о нем не будет выдаваться в команде ls. Если вы хотите увидеть скрытые каталоги, используйте команду ls -la).

Добавление файлов в репозиторий git

Репозиторий создан, можно переходить к разработке кода программной системы. В качестве примера давайте создадим файл с кодом программы, определяющей тональность текста с помощью библиотеки Hugging Face. Для этого в каталоге репозитория создайте файл sentiment.py и запишите в него код, который мы рассматривали в разделе, посвященном определении тональности текстов:

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis",
                      "blanchefort/rubert-base-cased-sentiment")

classifier("Я обожаю инженерию машинного обучения!")
```

Отслеживание изменений в git

Git отслеживает любые изменения, выполненные в репозитории. Добавление файла sentiment.py тоже относится к таким изменениям. Получить информацию об изменениях в репозитории можно с помощью команды git status:

```
ml_project % git status
```

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        sentiment.py

nothing added to commit but untracked files present (use "git add" to track)
```

В выводе команды git status достаточно много информации. Во-первых, git сообщает нам, что работа выполняется в ветке main (“On branch main”). Ветки – это механизм git для работы с большими изменениями в коде, которые могут привести к потенциальным конфликтам при синхронизации между всеми репозиториями. Механизм устроен достаточно сложно, поэтому его изучение мы отложим на второй семестр нашей магистерской программы. В первом семестре мы будем использовать только одну ветку с названием по умолчанию “main” (основная ветка). Поэтому в первом семестре ваша задача – отслеживать, что вы всегда работаете именно в ветке main.

Второе сообщение, “No commits yet”, говорит о том, что зафиксированных изменений (commit) в репозитории нет. Разработчикам редко удается написать работающий код с первой попытки. Обычно код нужно протестировать и по результатам тестирования внести изменения. Записывать в репозиторий имеет смысл только окончательную работоспособную версию, а не промежуточные результаты. Именно для этого служит механизм фиксации изменений: сохранении версии программы в репозитории выполняется после того, как закончено ее изменение. Для фиксации изменений необходимо ввести специальную команду git commit.

Третье сообщение говорит о том, что в репозитории существуют неотслеживаемые git файлы:

```
Untracked files:
```

```
  (use "git add <file>..." to include in what will be committed)
        sentiment.py
```

Мы добавили в репозиторий файл с кодом sentiment.py, но git пока не отслеживает изменения в этом файле. Часто разработчик вносит изменения в несколько файлов, но зафиксировать изменения нужно только в некоторых из них. git выдает нам подсказку, что нужно сделать, чтобы подготовить файл для фиксации изменений (включить файл в commit): использовать команду git add.

Последняя строка говорит о том, что файлы для фиксации изменений не отмечены, но есть файлы, в которых произошли изменения.

```
nothing added to commit but untracked files present (use "git add" to track)
```

Фиксация изменений в git

Давайте зафиксируем изменения в файле sentiment.py, который мы создали на предыдущем этапе. Для этого нам необходимо выполнить два действия:

1. Указываем, что изменения в файле sentiment.py будут зафиксированны, с помощью команды git add:

```
git add sentiment.py
```

Просматриваем статус репозитория после выполнения команды:

```
ml_project % git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   sentiment.py
```

В выводе команды git status можно увидеть, что для фиксации изменений (“Changes to be committed:”) подготовлен новый файл sentiment.py (new file: sentiment.py). Теперь можно переходить к фиксации изменений.

2. Фиксируем изменения с помощью команды git commit:

```
ml_project % git commit -m "Added sentiment.py"
[master (root-commit) 9cfc2c4] Added sentiment.py
 1 file changed, 6 insertions(+)
 create mode 100644 sentiment.py
```

С помощью параметра -m необходимо указать строку с описанием внесенных изменений. В нашем случае строка "Added sentiment.py" (добавлен файл sentiment.py).

Фиксация изменений завершена, давайте снова проверим статус репозитория:

```
ml_project % git status
On branch main
nothing to commit, working tree clean
```

Изменений для фиксации нет.

Передача зафиксированных изменений в удаленный репозиторий

С помощью команды git commit мы зафиксировали изменения в репозитории на локальном компьютере. Однако этого недостаточно для командной разработки. Чтобы ваши изменения стали доступны другим разработчикам в команде, их нужно передать на удаленный репозиторий (или несколько репозиториев). Для этого используется команда git push:

```
ml_project % git push origin main
```

В команде git push необходимо указать, какую ветку мы хотим передать. Так как мы пока работаем только с одной веткой, то указываем ее название –

main. Также нужно указать, на какой компьютер нужно передать изменения. Это делается с помощью обозначения компьютера – origin.

Итоги

- Для создания репозитория git используется команда git init
- Изменения в репозитории git выполняются в следующей последовательности:
    1. Редактирование файлов в репозитории или создание новых файлов.
    2. Отметка файлов для включения в фиксацию изменений с помощью команды git add
    3. Фиксация изменений с помощью команды git commit.
    4. Отправка изменений на удаленный репозиторий с помощью команды git push.
- Просмотр состояния репозитория git выполняется с помощью команды git status.

Карточки

Выберите действие, которое выполняет команда git

Команда git	Действие команды
git init	Создание репозитория git
git status	Просмотр статуса репозитория git
git commit	Фиксация изменений в репозитории
git add	Выбор файлов для фиксации изменений
git push	Отправка изменений на удаленный репозиторий для синхронизации

Модуль № 1. Юнит № 8. Сервис GitHub

Систему контроля версий git для командной разработки можно установить и настроить самостоятельно, а можно воспользоваться одним из

существующих сервисов для создания репозиториев git. Сейчас популярны следующие сервисы:

- GitHub – https://github.com/
- BitBucket – https://bitbucket.org/
- GitLab – https://gitlab.com/

Мы будем использовать сервис GitHub, т.к. он является лидером в своей области. На GitHub размещается исходный код многих популярных библиотек машинного обучения: TebsorFlow (https://github.com/tensorflow/tensorflow), PyTorch (https://github.com/pytorch/pytorch), scikit-learn (https://github.com/scikit-learn/scikit-learn), Hugging Face (https://github.com/huggingface) и др.

Видео: Создание репозитория на GitHub – https://www.dropbox.com/s/m50b22itd5ndbl5/github.mp4?dl=0

Видео: Настройка ключа SSH на GitHub – https://www.dropbox.com/s/f5pj7i5i9gjz873/ssh_github.mp4?dl=0

Итоги

- GitHub – самый популярный сервис для создания и хранения репозиториев git.
- GitHub использует SSH ключи для проверки прав доступа пользователей при обновлении репозитория.
- Команды git для работы с удаленным репозиторием:
    - git clone – клонирование удаленного репозитория на локальный компьютер.
    - git push – отправка изменений с локального компьютера на удаленный репозиторий.
    - git pull – получение изменений с удаленного репозитория на локальный компьютер.

Тест

Какая команда git используется для клонирования удаленного репозитория на локальный компьютер?

1. git checkout
2. git copy
3. **git clone**

4. git download

Для чего используется ключ SSH на GitHub?

1. Для проверки прав доступа пользователя к Web порталу GitHub.
2. **Для проверки прав доступа пользователя при обновлении репозитория GitHub с локального компьютера командой git push.**
3. GitHub не использует ключи SSH.
4. Для организации доступа без ввода пароля к виртуальной машине Linux в GitHub.

Какая команда git используется для получения изменений с удаленного репозитория на локальный компьютер?

1. **git pull**
2. git push
3. git clone
4. git sync

Практическое задание

Цель задания: научиться в команде создавать приложения, использующие готовые библиотеки машинного обучения.

Задание:

1. Сформируйте команду из 2-3 человек.
2. Создайте репозиторий на GitHub, который будет использоваться для командной разработки приложения.
3. Изучите возможности готовых библиотек машинного обучения.
4. Сформулируйте задачу, которую вы хотите решить с помощью машинного обучения.
5. Реализуйте решение выбранной вами задачи в коде с использованием готовой библиотеки машинного обучения.
6. Отправьте реализованное решение в репозиторий на GitHub.
7. Оформите документацию на ваше решение в репозитории.

Рекомендуется использовать одну из следующих библиотек машинного обучения: Hugging Face (https://huggingface.co/), TensorFlow Hub (https://www.tensorflow.org/hub), PyTorch Hub (https://pytorch.org/hub/), Keras Applications (https://keras.io/api/applications/).

Пример репозитория – https://github.com/sozykin/ml_sentiment

Термины

Все потенциально-незнакомые и новые понятия выдели отдельно и “положи” сюда.

Список источников

- Фредерик Брукс: Мифический человеко-месяц, или Как создаются программные системы. 1975.
- Giannakis, M., Dubey, R., Yan, S. et al. Social media and sensemaking patterns in new product development: demystifying the customer sentiment. Ann Oper Res (2020). https://doi.org/10.1007/s10479-020-03775-6
- Software Engineering Body of Knowledge (SWEBOK) (рекомендации IEEE к области знаний «Программная инженерия») – https://www.computer.org/education/bodies-of-knowledge/software-engineering
- Andrew Ng. Специализация Machine Learning Engineering for Production (MLOps) – https://www.coursera.org/specializations/machine-learning-engineering-for-production-mlops
- Hugging Face Course – https://huggingface.co/course/
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding – https://arxiv.org/abs/1810.04805
- Hugging Face Models – https://huggingface.co/models
- Система контроля версий git – https://git-scm.com/
- Scott Chacon, Ben Straub. Pro Git. – https://git-scm.com/book/ru/v2

Дополнительные материалы

- Hugging Face Course – https://huggingface.co/course/
- Ноутбук в Colab с определением тональности текстов на русском языке с помощью библиотеки Hugging Face – https://colab.research.google.com/drive/136RKDIVzzLkMsiO8H6VRMYO1h0ZsdIgv?usp=sharing
- Ноутбук в Colab с автоматическим переводом с русского на английский язык с помощью библиотеки Hugging Face – https://colab.research.google.com/drive/1MvqqKTOFqMepC1VVEiCxOuhggHQx2a4q?usp=sharing

- Ноутбук в Colab с генерацией аннотаций новостей с помощью библиотеки Hugging Face – https://colab.research.google.com/drive/1P1tAK15CIYFd7WFvGN6ImBD-sQZXUuRi?usp=sharing
- BERT — state-of-the-art языковая модель для 104 языков. Туториал по запуску BERT локально и на Google Colab – https://habr.com/ru/post/436878/
- Документация на Pipeline в библиотеке Hugging Face – https://huggingface.co/transformers/main_classes/pipelines.html
- Готовые модели машинного обучения в библиотеке Hugging Face – https://huggingface.co/models)
- Готовые модели машинного обучения в библиотеке TensorFlow – (https://www.tensorflow.org/hub),
- Готовые модели машинного обучения в библиотеке PyTorch – https://pytorch.org/hub/
- Готовые модели машинного обучения в библиотеке Keras – https://keras.io/api/applications/

Модуль № 2
Название: Архитектура приложений

Образовательные результаты:

- Студент знает виды архитектур ПО и подходы к ним, включая микросервисную архитектуру и API (Application Programming Interface).
- Студент знает сетевые технологии, которые используются для реализации и использования API.
- Студент может применять инструменты для работы с API: curl и Postman.

В этом модуле:

Для обучения моделей чаще всего используются Jupyter или Colab ноутбуки. Они хорошо подходят для быстрых экспериментов с моделями машинного обучения. Однако для продуктивного использования систем машинного обучения ноутбуки не пригодны.

В модуле вы узнаете:

- Какие проблемы возникают при росте программной системы.
- Что такое архитектура программных систем, как она связана с качеством работы программных систем и возможностью их развития.
- Какие существуют популярные подходы к архитектурам.
- Как устроена наиболее популярная в настоящее время микросервисная архитектура, при которой программная система делится на большое количество небольших сервисов, взаимодействующих по сети с помощью Application Programming Interface.
- Какие сетевые технологии используются для организации взаимодействия независимых приложений.
- Какие инструменты используются для вызова методов удаленных приложений по сети с помощью API в микросервисной архитектуре.

Модуль № 2. Юнит № 1. Архитектура программного обеспечения

Прикладные системы искусственного интеллекта – это сложные программные системы, решающие большое количество связанных задач. Задачи включают сбор и подготовку данных, обучение моделей, построение приложений на основе моделей, применение моделей для обработки новых данных, а также дообучение при необходимости.

Подготовка данных и обучение моделей часто выполняется в Colab или Jupyter ноутбуках. Именно такой подход вы использовали на предыдущих занятиях.
Однако разрабатывать крупные программные системы в одном ноутбуке затруднительно по нескольким причинам:

- Ноутбук работает только в браузере. Автоматизировать его выполнение на сервере или в мобильном приложении очень сложно.
- Ячейки кода в ноутбуке нужно запускать последовательно друг за другом. Если одну или несколько ячеек пропустить, то можно получить неправильный результат. Также неверный результат может быть получен, если ячейки запустить в неправильной последовательности.
- При увеличении сложности системы растет объем кода. Код становится нечитаемым и сложным для понимания и сопровождения.
- Изменение кода в любой ячейке ноутбука может привести к неработоспособности всего ноутбука в целом.
- Отлаживать код в ноутбуке очень сложно.

Описанные проблемы характерны не только для разработки приложений в ноутбуках, но и для создания любого приложения как единого целого.

Архитектура программного обеспечения

Конструктивный способ создания крупной программной системы – это разделение ее на несколько независимых частей меньшего размера. Такой подход называется *декомпозицией.*

В процессе декомпозиции система делится на несколько модулей, которые взаимодействуют между собой. Декомпозиция обеспечивает следующие преимущества:

- Каждый модуль системы имеет относительно небольшой объем, поэтому логику его работы просто понять.

- Небольшие модули проще отлаживать и обеспечивать стабильность их работы.
- Внесение изменений в модуль в основном влияет на работу этого модуля и редко может повредить работоспособности системы в целом. Обеспечивать поддержку системы в таком случае значительно проще.
- При необходимость реализация модуля может быть полностью заменена на другую (например, на основе готовой библиотеки). При этом работа системы в целом не изменится.

Архитектура программного обеспечения описывает способ декомпозиции крупной программной системы на отдельные модули, организации их взаимодействие между собой и с внешним миром. Правильный выбор архитектуры может являться критическим для обеспечения успешной работы системы и ее развития со временем.

Большой комок грязи

Пример плохой архитектуры, ведущий к существенным проблемам с разработкой приложения, описали Брайан Фут и Джозеф Йодер в статье “Большой комок грязи (Big ball of mud)”:

“Большой комок грязи — это беспорядочно структурированные, растянутые, неряшливые, словно перемотанные на скорую руку изоляционной лентой и проводами, джунгли спагетти-кода. Эти системы показывают безошибочные признаки нерегулируемого роста и постоянных доделок. Информация делится беспорядочно между отдаленными элементами системы, часто до такой степени, что почти вся важная информация становится глобальной или дублируется. Общая структура системы, возможно, никогда не была четко определена. Если и была, то стала размыта до неузнаваемости. Программисты хоть немного понимающие архитектуру, обходят это болото стороной. И только те, кого она волнует мало и, возможно, те, кому нравится латать дыры в системе каждый день, довольны работой над такими системами.”

Метафора спагетти-кода используется чтобы показать сильную связность кода в различных частях системы. Когда вы пытаетесь достать одну спагетти из тарелки, она тянет за собой все остальные. В спагетти-коде изменение в одном месте ведет к необходимости менять код во многих других местах, иначе система потеряет работоспособность.

Поддерживать и развивать системы с архитектурой “Большой комок грязи” на протяжении длительного времени очень сложно. К сожалению, код в Colab или Jupyter ноутбуках при увеличении объема также превращается в большой комок грязи, его сложно понять, сопровождать и развивать. Поэтому такой код необходимо разбивать на отдельные, независимые части.

Итоги

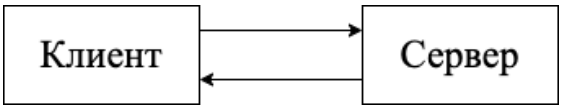
- Архитектура программного обеспечения описывает способ декомпозиции крупной программной системы на отдельные независимые части и взаимодействие этих частей между собой.
- Без декомпозиции крупные программные системы сложно сопровождать, развивать и обеспечивать их стабильную работу.
- Архитектуру программного обеспечения необходимо продумывать заранее, в противном случае можно прийти к архитектуре типа “Большой комок грязи”.

Модуль № 2. Юнит № 2. Популярные архитектуры программного обеспечения

Архитектура программной системы, как правило, не является уникальной разработкой отличающейся от всех других архитектур. Обычно, при разработке архитектуры используется один из популярных архитектурных шаблонов: проверенный способ разделения системы на компоненты и организации взаимодействия между ними. Такие шаблоны изучаются исследователями программной инженерии, поэтому их свойства, достоинства и недостатки, а также рекомендованные случаи для применении хорошо известны. В этом разделе мы рассмотрим наиболее популярные шаблоны архитектур.

Клиент-серверная архитектура

Один из первых вариантов разделения приложения на части является клиент-серверная архитектура. Клиентская часть приложения отвечает за интерфейс пользователя, а серверная – за логику обработки данных.

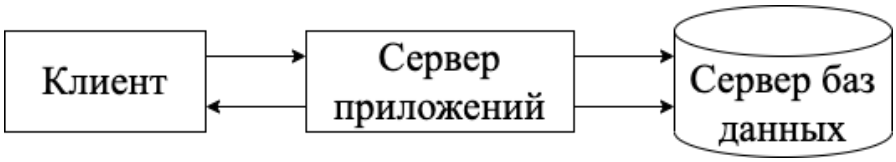


Архитектура клиент-сервер

Первоначально клиент-серверная архитектура реализовывалась в виде приложения, которое устанавливалось на компьютер (десктопное приложение или толстый клиент) и сервера баз данных. Затем в качестве клиента все чаще стал применяться Web-браузер, взаимодействующий с Web-сервером.

Трехуровневая архитектура

Развитием архитектуры клиент-сервер является трехуровневая архитектура. В ней серверная часть разделена на две: сервер приложений, который реализует логику обработки данных, и сервер баз данных, который хранит данные.



Трехуровневая архитектура

Уровневая архитектура

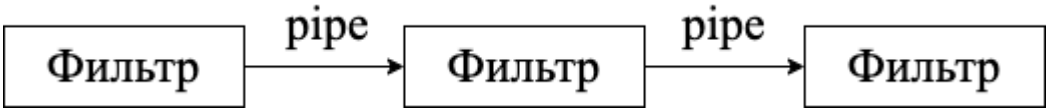
Еще один вариант организации работы приложения – разделение его на уровни. Уровни строятся друг над другом, каждый уровень взаимодействует только с двумя соседними уровнями: верхним и нижним.

Уровневая архитектура

Широко известными примерами уровневой архитектуры является модели компьютерных сетей TCP/IP и OSI.

Архитектура пайплайн

Архитектура пайплайн (от английского pipeline, трубопровод) берет свое начало из операционной системы Unix. Linux является одним из современных вариантов Unix и использует те же принципы. При проектировании Unix было принято решение разрабатывать простые программы, каждая из которых решает только одну задачи, но делает это эффективно. Чтобы программы можно было использовать для решения сложных задач, был создан механизм комбинации программ с друг с другом, который в Unix назвали pipe (труба). С помощью pipe выходные данные от одной программы передаются на вход другой. Таким образом можно комбинировать несколько программ, чтобы реализовывать сложные задачи обработки данных.



Архитектура пайплайн

В архитектуре пайплайн используются компоненты двух типов: фильтры, которые производят обработку данных, и связи между фильтрами (pipe или трубы), которые передают данные. Передача данных, как правило, выполняется однонаправленно от одного фильтра к другому.

Архитектура пайплайн часто используется в приложениях анализа данных и машинного обучения. Например, первый фильтр может готовить данные для анализа, второй запускает модель машинного обучения для анализа данных, а третий – преобразует выходные данные из модели машинного обучения к понятному пользователю виду. Библиотека Hugging Face, которую мы рассматривали ранее, применяет именно этот архитектурный шаблон.

Другое популярное название архитектуры пайплайна – конвейерная архитектура. В пайплайне производится несколько последовательных операций по обработке данных. Это напоминает производство

промышленной продукции на конвейере, где операции также выполняются последовательно одна за другой.

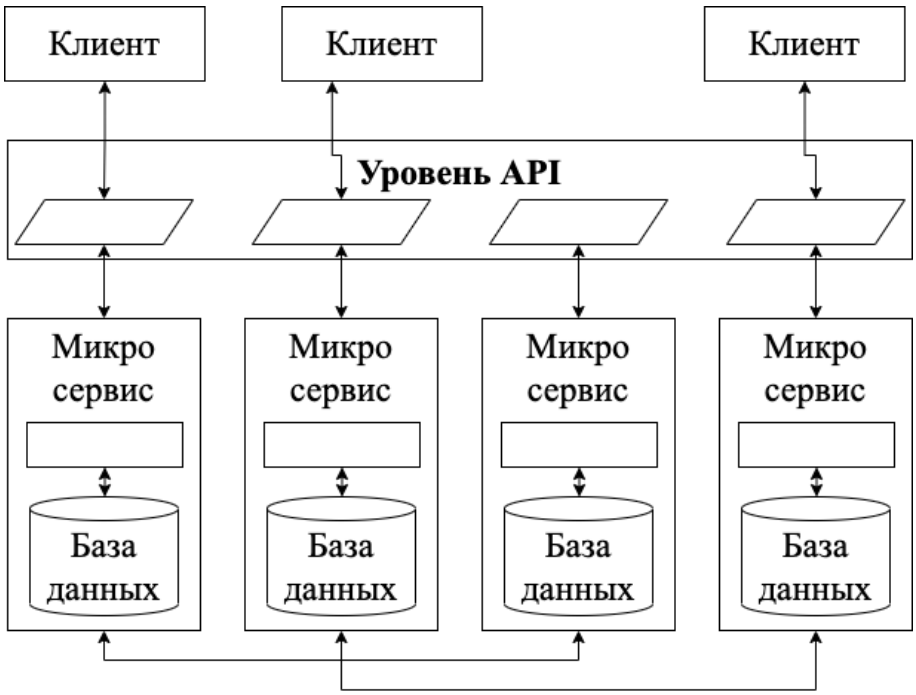
Архитектура, основанная на сервисах

Популярный вариант декомпозиции приложения – разделение его на логически независимые части, каждая из которых реализует отдельную функциональность, полезную пользователю. Такие части называются *Сервисами.* В данном типе архитектуры используется несколько сервисов, но единый интерфейс пользователя и одна база данных.

Архитектура, основанная на сервисах

Микросервисная архитектура

Развитием архитектуры, основанной на сервисах, является микросервисная архитектура. В этой архитектуре декомпозиция еще более глубокая: каждый микросервис содержит в себе все необходимое для изолированной работы, включая базу данных.



Микросервисная архитектура

Взаимодействие с микросервисами выполняется через уровень API (Application Programming Interface, программный интерфейс приложения) по компьютерной сети. API описывает способы, которыми можно использовать микросервис. Причем использовать API может как клиент, реализующий интерфейс пользователя, так и любое другое приложение.

В качестве примера приложения микросервисной архитектуры давайте рассмотрим социальную сеть Вконтакте (vk.com). У социальной сети есть несколько вариантов пользовательских интерфейсов: web-приложение и мобильное приложение. Оба эти варианта интерфейса взаимодействуют с серверной частью vk.com через API. Но этот же API может использовать любой разработчик, в том числе и вы. Например, вы можете создать свой собственный пользовательский интерфейс для работы с Вконтакте. Или разработать загрузчик данных, который анализирует социальную сеть и загружает из нее интересующие вас данные.

Подобным образом работает система Wizard Уральского федерального университета (https://wizard.urfu.ru/), которая анализурует профиль пользователя Вконтакте и сравнивает его с профилями 15 тыс.студентов УрФУ. По результатам анализа пользователю рекомендуются образовательные программы университета, которые могут его заинтересовать.

Документация на API Вконтакте доступна по ссылке – https://vk.com/dev/first_guide. Далее в этом модуле мы будем изучать инструменты работы с API более подробно.

В настоящее время микросервисная архитектура является наиболее популярной. Поэтому важно научится создавать микросервисы, использующие модели машинного обучения. Работа с такими микросервисами осуществляется по сети через API. Создав микросервис и API для своей модели, а также опубликовав документацию на API, вы сделаете свою модель доступной всем, кто заинтересован в ее использовании.

Итоги

- Для разработки архитектуры приложения часто используют один из готовых шаблонов архитектур.
- В настоящее время самым популярным архитектурным шаблоном является микросервисная архитектура.
- Микросервис представляет собой изолированную программную систему, которая содержит все необходимое для функционирования, включая базу данных и другие компоненты.
- Взаимодействие с микросервисами выполняется по сети с помощью программного интерфейса приложений API.

Тест

1. Какие компоненты входят в трехуровневую архитектуру?
    - **Клиент**
    - Труба
    - **Сервер баз данных**
    - Фильтр
    - **Сервер приложений**
2. Какая архитектура сейчас наиболее популярна?
    - Клиент-серверная
    - Уровневая
    - **Микросервисная**
    - Монолитная
3. Что такое API?
    - Интерфейс пользователя для Web-сервера
    - Язык для работы с данными в базе

- **Описание способов, с помощью который микросервис может быть использован другими приложениями**
- Язык для работы с социальными сетями, в том числе Вконтакте.

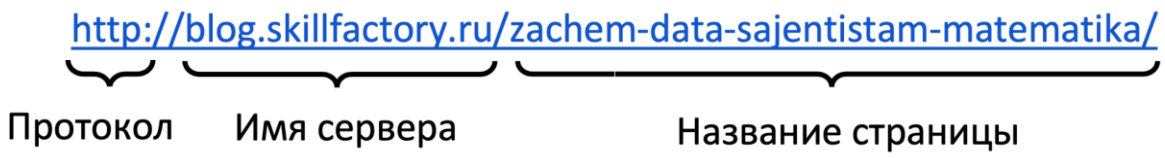
Модуль № 2. Юнит № 3. Основы сетевых технологий

Приложениям для взаимодействия по сети необходим общий язык. В противном случае приложения не смогут понять друг друга и сформулировать правильные ответы на поступающие запросы. В компьютерных сетях в качества такого языка выступают протоколы. Протоколы описывают, что именно приложения пересылают друг другу по сети и в каком формате.

Протокол HTTP

В микросервисной архитектуре для взаимодействия приложений чаще всего применяется протокол HTTP (Hyper Text Transfer Protocol). Этот протокол был разработан для передачи Web-страниц он по прежнему используется для этой цели. Когда вы открываете страницу в браузере, она передаётся с Web-сервера именно по протоколу HTTP.

Важным понятием в протоколе HTTP является ссылка, или унифицированный указатель ресурса (URL, Uniform Resource Locator). Именно она показывает, какую именно страницу нужно загрузить. Вот пример ссылки:



Ссылка состоит из следующих компонентов:

- http – идентификатор протокола, по которому передаются данные.
- blog.skillfactory.ru – адрес сервера, на котором расположена страница
- zachem-data-sajentistam-matematika – идентификатор страницы, которую нужно загрузить.

Протокол HTTP работает в режиме запрос-ответ. Например, бразуер передает на Web-сервер запрос на страницу http://blog.skillfactory.ru/zachem-data-sajentistam-matematika/, в ответ сервер передает содержимое страницы.

Протокол HTTP для взаимодействия приложений

С развитием сетевых технологий и архитектур приложений оказалось, что протокол HTTP хорошо подходит не только для передачи Web-страниц, но и для взаимодействия приложений по сети. В этом случае клиент передает на сервер в запросе HTTP не идентификатор страницы, которую он хочет получить, а метод, которых нужно вызвать. Сервер, получив такой запрос, запускает нужный метод и возвращает результат выполнения клиенту. Например, узнать регион сервера по его адресу можно с помощью следующего запроса:

http://ip-api.com/json/urfu.ru

Здесь:

- http – идентификатор протокола.
- ip-api.com – адрес сервера, реализующего сетевой сервис.
- json – название метода, который вызывается (определяет формат данных, в котором возвращается результат).
- urfu.ru – адрес сервера, для которого требуется узнать регион

В результате выполнения такого запроса получим ответ:


```
{"status":"success",
"country":"Russia",
"countryCode":"RU",
"region":"SVE",
"regionName":"Sverdlovsk Oblast",
"city":"Yekaterinburg",
"zip":"620000",
"lat":56.8419,
"lon":60.6143,
"timezone":"Asia/Yekaterinburg",
"isp":"Ural Federal University",
"org":"Services",
"as":"AS44868 UrFU",
```

"query":"93.88.179.200"}

Результат действительно в формате JSON и содержит много информации о сервере urfu.ru: страну, область, город, временную зону и некоторую другую информацию.

Пример выполнения запроса в браузере:

Пример выполнения запроса к API определения региона сервера в браузере

API сервиса описывает набор методов и их параметров, которые предоставляет сервис. Вот несколько примеров API популярных сервисов:

- vk.com API – https://vk.com/dev/first_guide
- HeadHunter API – https://dev.hh.ru/
- YouTube Data API – https://developers.google.com/youtube/v3
- Twitter API – https://developer.twitter.com/en/docs/twitter-api

С помощью API HeadHunter вы можете найти подходящие для вас вакансии на заходя на сайт. API vk.com позволит вам получить информацию из этой социальной сети в своем приложении.

Попробуйте найти API других популярных сетевых сервисов.

Протокол HTTP: методы и параметры

В этом разделе мы рассмотрим некоторые детали работы протокола HTTP, которые полезны для использования сложных API.

Протокол HTTP использует различные *методы,* которые определяют, что именно нужно сделать серверу с запросом, который он получил. Наиболее популярны два метода:

- GET – используется для запроса web-страницы или другого ресурса с сервера.
- POST – используется для передачи данных на сервер, например, при заполнении формы на Web-странице.

Также в протоколе HTTP можно использовать *параметры,* которые передаются на сервер вместе с запросом и учитываются при его выполнении. Они используются аналогично параметрам функций в языках программирования.

Например, на сервер ip-api.com можно передать параметр lang, который задает язык ответа. Способы передачи параметров зависят от типа используемого метода. Для метода GET запрос передается в ссылке:

http://ip-api.com/json/urfu.ru?lang=ru

В ссылке параметры указываются после знака вопроса: ?. Формат указания параметров: имя_парметра=значение. В примере приводится параметр lang=ru. При отправке такого запроса мы получим ответ на русском языке:


```
{"status":"success",
"country":"Россия",
"countryCode":"RU",
"region":"SVE",
"regionName":"Свердловская область",
"city":"Екатеринбург",
"zip":"620000",
"lat":56.8419,
"lon":60.6143,
"timezone":"Asia/Yekaterinburg",
"isp":"Ural Federal University",
"org":"Services",
"as":"AS44868 UrFU",
"query":"93.88.179.200"}
```


Если параметров несколько, то они указываются через символ &, например, так:

http://ip-api.com/json/urfu.ru?lang=ru&fields=city

Здесь указывается два параметра:

- lang=ru – параметр для языка: русский.
- fields=city – параметр для выбора полей ответа, название поля – city (город).

Запрос вернет одно поле на русском языке:

```
{"city":"Екатеринбург"}
```

Передача параметров в запросе POST устроена более сложно. Это делается не через ссылку, а через тело запроса. Пример передачи параметров в POST:

```
POST http://ip-api.com/json/urfu.ru
content-type: application/json

{
   "lang": "ru",
   "fields": "city"
}
```

Параметры запроса передаются в формате JSON. Здесь мы также передаем два параметра: lang и fields.

Протокол HTTP: заголовки

В примере вы можете обратить внимание на строку “content-type: application/json”. Эта строка является *заголовком* HTTP. Заголовки помогают Web-серверу или клиенту понять, что содержится в запросе или ответе. Например, заголовок “content-type” указывает, в каком формате передаются данные в теле запроса. Значение “application/json” говорит о том, что данные передаются в формате JSON. Примеры других популярных заголовков в HTTP:

- Content-Language – язык запроса
- Content-Encoding – тип кодировки
- User-Agent – тип клиента, которые обращается к серверу

Документация на API сервисов описывает заголовки, которые нужно использовать для работы с API.

Протокол HTTP: статус выполнения запроса

Вместе с ответом сервер возвращает статус выполнения запроса. Статус – это число, по которому можно определить, успешно ли выполнен запрос. А если с выполнение запроса есть проблемы, то по статусу можно определить причину ошибки. Наиболее популярные статусы ответов:

- 200 OK – запрос выполнен успешно.
- 301 Moved Permanently – перенаправление, страница перемещена навсегда.
- 302 Moved Temporarily – перенаправление, страница перемещена временно.
- 400 Bad Request – ошибка клиента, неправильный запрос.
- 404 Not Found – ошибка клиента, страница (или другой ресурс) не найдена.
- 500 Internal Server Error – ошибка сервера, внутренняя ошибка.
- 503 Service Unavailable – ошибка сервера, сервис недоступен.

Важно понимать, что в распределенных системах в любой момент может произойти ошибка, вызванная работой сети, проблемами с удаленным сервером или программным обеспечение на нем. Поэтому важно проверять статус выполнения запроса и разрабатывать код для проверки ошибок сетевого взаимодействия и устранения их последствий.

Итоги

- Для взаимодействия между приложениями по компьютерной сети используются протоколы.
- Наиболее популярный протокол для взаимодействия приложений с использованием API – HTTP.
- В HTTP используются ссылки (URL) для указания, какой ресурс нужно загрузить или какой метод API нужно вызвать.
- Большая часть современных крупных информационных систем предоставляют API с открытым доступом.

Тест

1. Какой протокол используется для взаимодействия между приложениями по API?
    - DNS
    - **HTTP**
    - SMTP
    - POP3
2. Для чего нужен статус выполнения запроса HTTP?
    - Чтобы передавать данные в теле ответа на запрос
    - **Чтобы обнаружить ошибку при выполнении запроса и причину этой ошибки**
    - Чтобы скрыть ошибку выполнения запроса от клиента
    - Статус HTTP на практике не используется, его можно игнорировать

Модуль № 2. Юнит № 4. Инструменты для работы с API: curl

Ранее мы рассматривали, как использовать API сетевых систем с помощью Web-браузера. Такой подход работает, но он не всегда удобен в использовании, т.к. браузеры рассчитаны на показ Web-страниц, а не взаимодействие с удаленными программами. Для решения этой проблемы разрабатываются специальные инструменты работы с API.

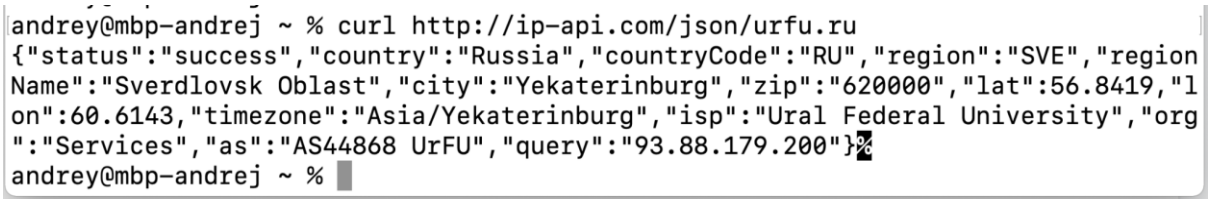
Мы будем изучать два инструмента работы с API: curl, который работает из командной строки, и Postman, который позволяет использовать графический интерфейс. В данном разделе мы рассмотрим curl, а в следующем – Postman.

curl – это утилита командной строки, которая позволяет взаимодействовать с использованием протоколов различных типов, использующих URL. Название утилиты расшифровывается как Client URL.

Самый простой способ использовать curl для взаимодействия по API – это вызвать утилиту и передать ей ссылку в качестве аргумента. Вот пример использования утилиты curl для определения региона сервера:

```
curl http://ip-api.com/json/urfu.ru
```

Результат выполнения команды в терминале:



Использование curl для вызова API

Как можно видеть, вызов API возвращает JSON, который в терминале выглядит таким же образом, как и в браузере.

При вызове curl можно указывать параметры для API в командной строке. Например, такой вызов позволит получить данные на русском языке:

```
curl http://ip-api.com/json/urfu.ru\?lang=ru
```

Обратите внимание, что в URL перед знаком вопроса добавлен символ \. Так приходится делать, потому что URL используется в командной строке, где знак вопросы является управляющим символом. Чтобы он перестал считаться управляющим символом, перед ним нужно написать символ \.

Указать метод HTTP, который будет использоваться при вызове, в curl можно с помощью параметра -X. Например:

```
curl -X GET http://ip-api.com/json/urfu.ru\?lang=ru
```

Напомню, что в HTTP чаще всего используется два метода: GET и POST. В документации на API вы увидите, какой именно метод нужно использовать.

При использовании метода POTS параметры указываются не в URL, а в теле запроса. Чтобы это сделать необходимо дополнительно указать HTTP заголовок с форматом тела запроса. Заголовки в curl указываются с помощью ключа -H (от английского header, заголовок). Данные для тела запроса указываются после параметра -d. Пример указания значения city для параметра fields в формате JSON:

```
curl -d '{"fields":"city"}' -H "Content-Type: application/json" -X POST
```

```
http://ip-api.com/json/urfu.ru
```

Необходимые для вызова метода API заголовки, параметры и их значения можно найти в документации на API.

Итоги

- curl – утилита командной строки для работы с API.
- Формат вызова: curl api_url
- Метод HTTP в curl указывается с помощью параметра -X
- Заголовки HTTP в curl указываются с помощью параметра -H
- Данные в теле запроса в curl указываются с помощью параметра -d
- Метод HTTP, заголовки, параметры и их значения указываются в документации на API.

Тест

1. Что такое curl
    - Утилита для работы с API с графическим интерфейсом
    - Инструмент для разработки архитектуры
    - **Утилита для работы с API из командной строки**
    - Утилита для проектирования API
2. Какой параметр в curl указывает использование метода HTTP POST?
    - -X GET
    - **-X POST**
    - -M POST
    - -H POST

Модуль № 2. Юнит № 5. Инструменты для работы с API: Postman

Утилита curl позволяет быстро и просто экспериментировать с API в командной строке. Однако работать с командной строкой не всем удобно. Кроме того, для решения достаточно сложных задач при работе с API curl не всегда удобен. Поэтому можно использовать инструмент для работы с API с графическим интерфейсом. Сейчас наиболее популярным инструментом является Postman.

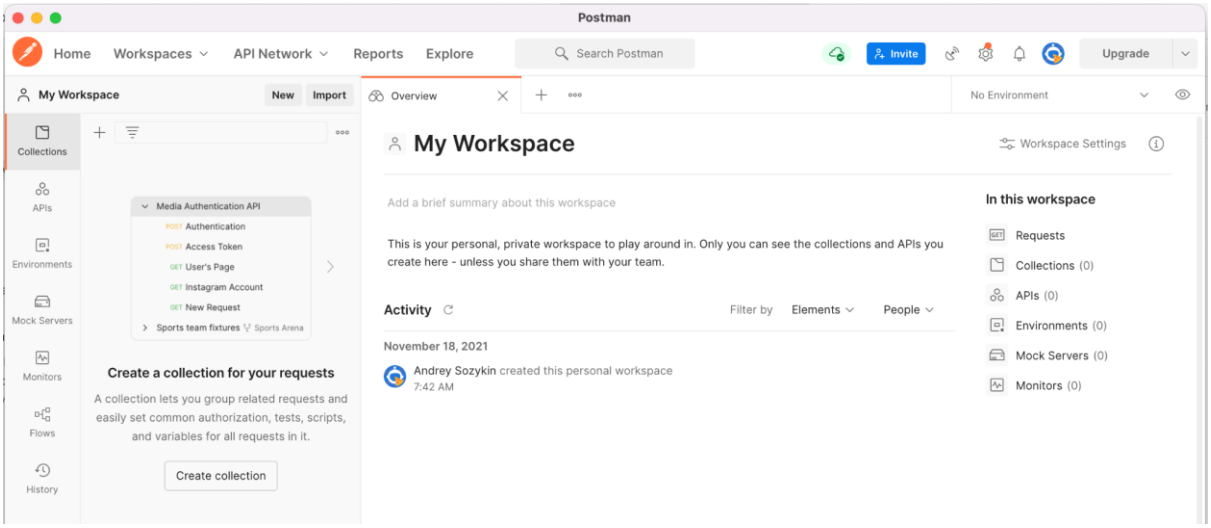
Установка Postman

Чтобы установить Postman на свой компьютер, перейдите по ссылке https://www.postman.com/downloads/. Скачайте версию Postman для своей операционной системы. После скачивания запустите установочный файл и выполните установку.

Также существует web-версия Postman, которую можно использовать, не устанавливая на свой компьютер. Web-версия доступна по ссылке – https://web.postman.co/

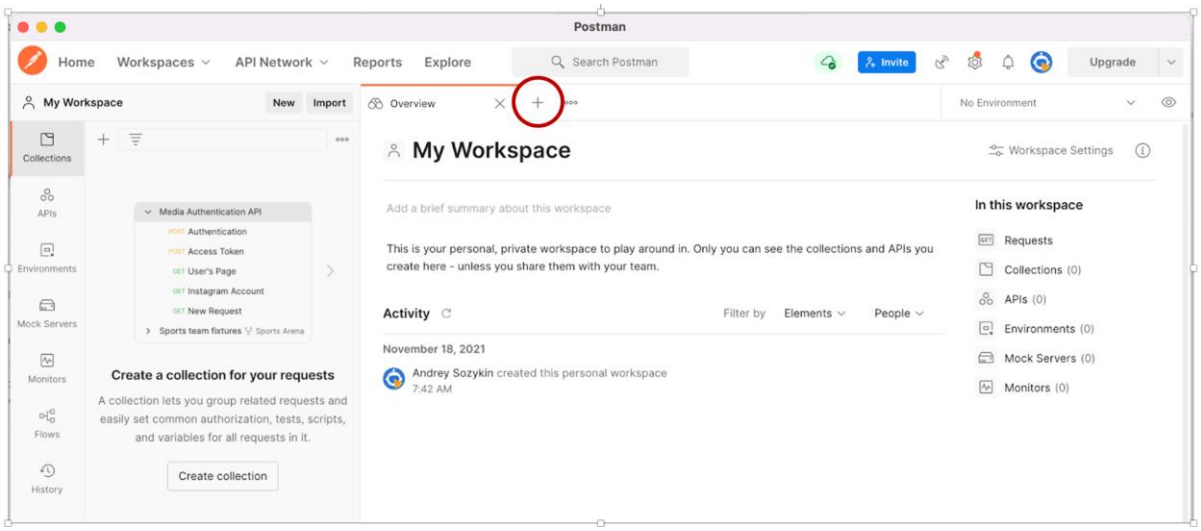
Работа с API в Postman

При запуске Postman вы попадаете в свое рабочее пространство (My Workspace).



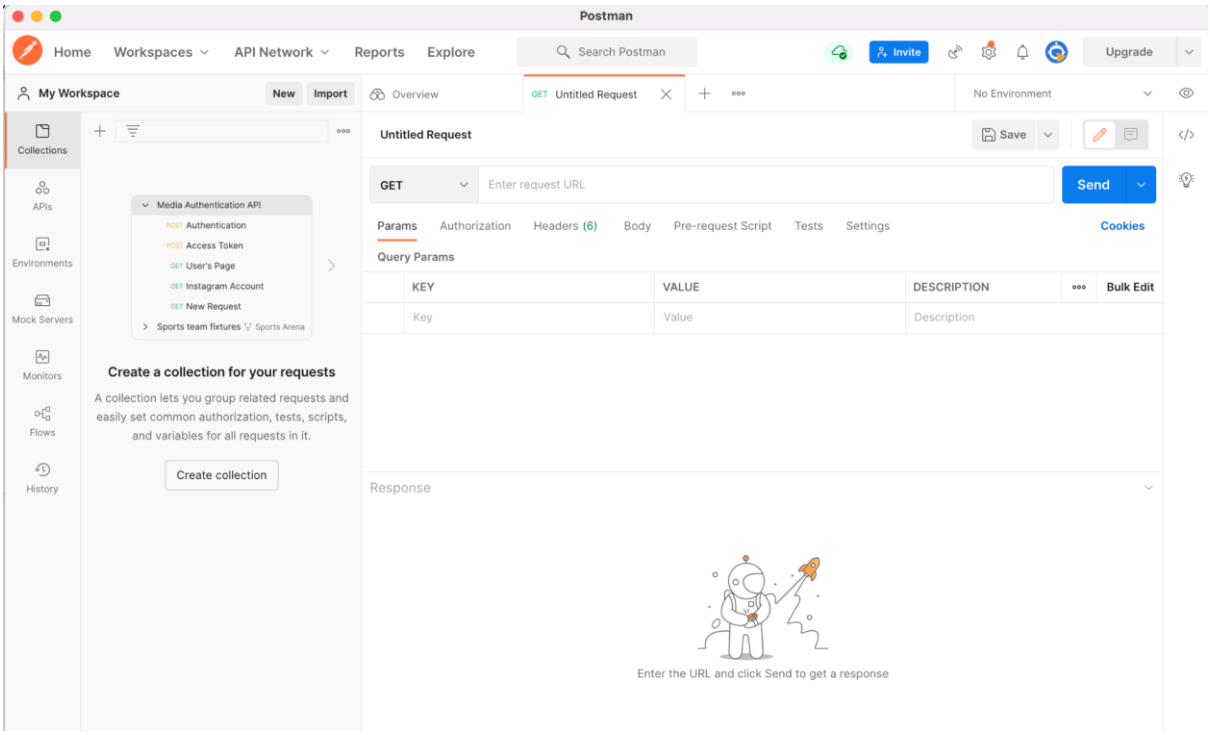
Рабочее пространство в Postman

Чтобы создать запрос к API, нужно нажать на кнопку + а верхней панели рабочего пространства.



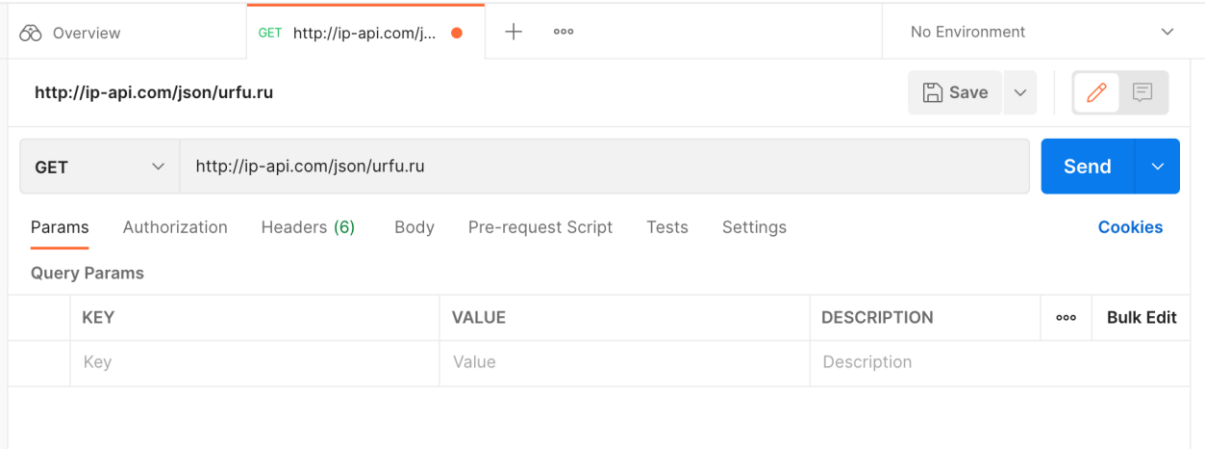
Создание окна для запроса к API

В результате будет открыто окно для ввода запроса к API.



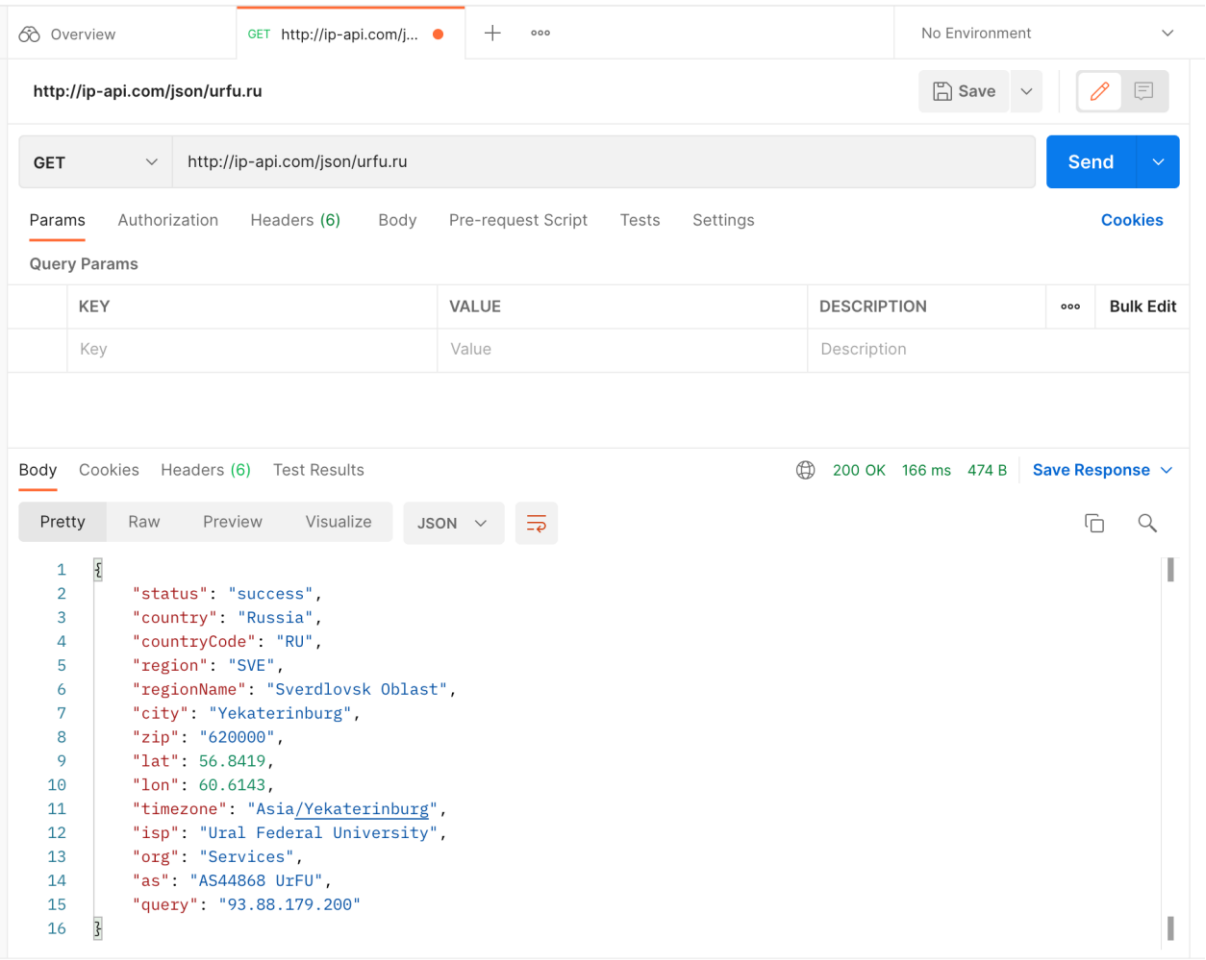
Окно запроса к API в Postman

Для запуска запроса необходимо вставить ссылку в поле “Enter request URL” и нажать кнопку “Send”.



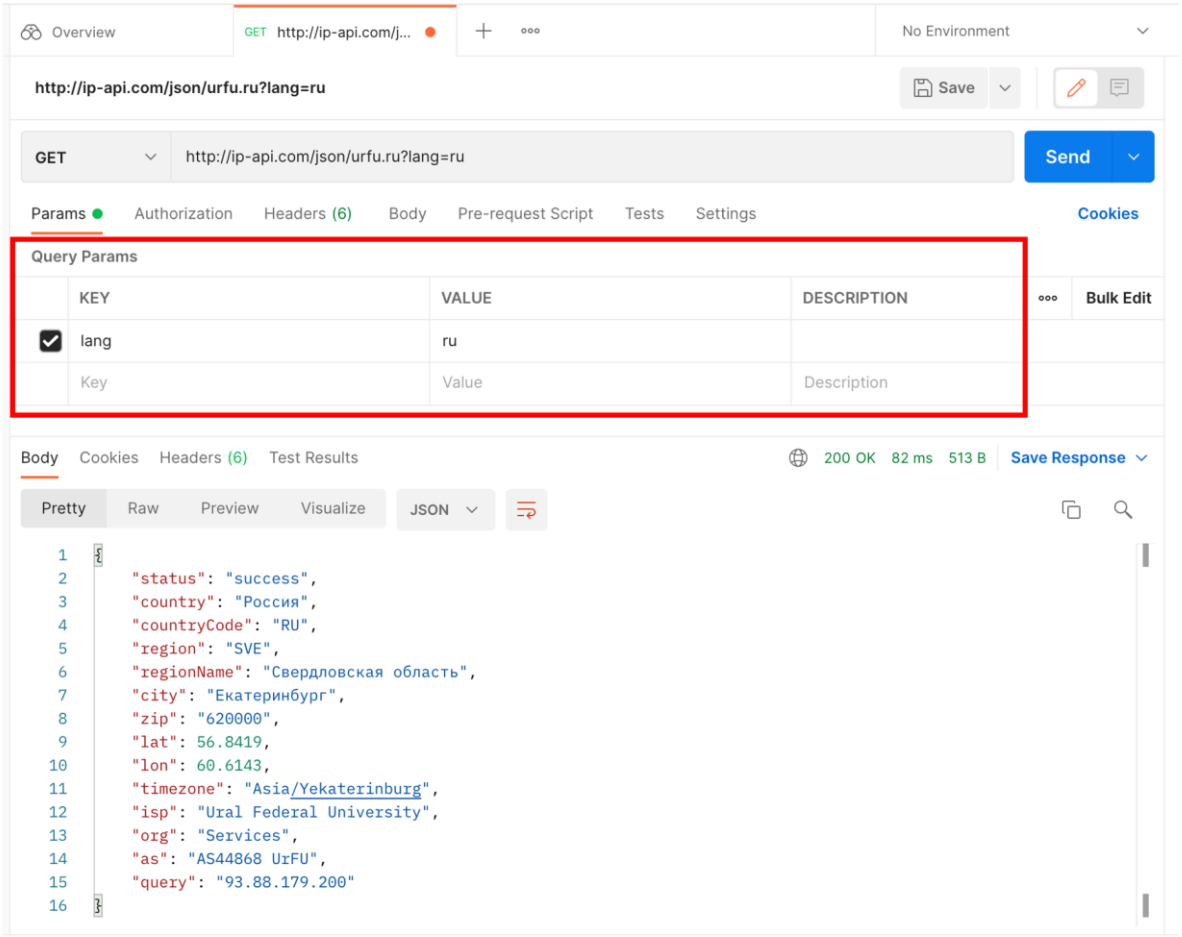
Ввод ссылки для запроса к API

Результат выполнения запроса будет показан в нижней части экрана окна Postman



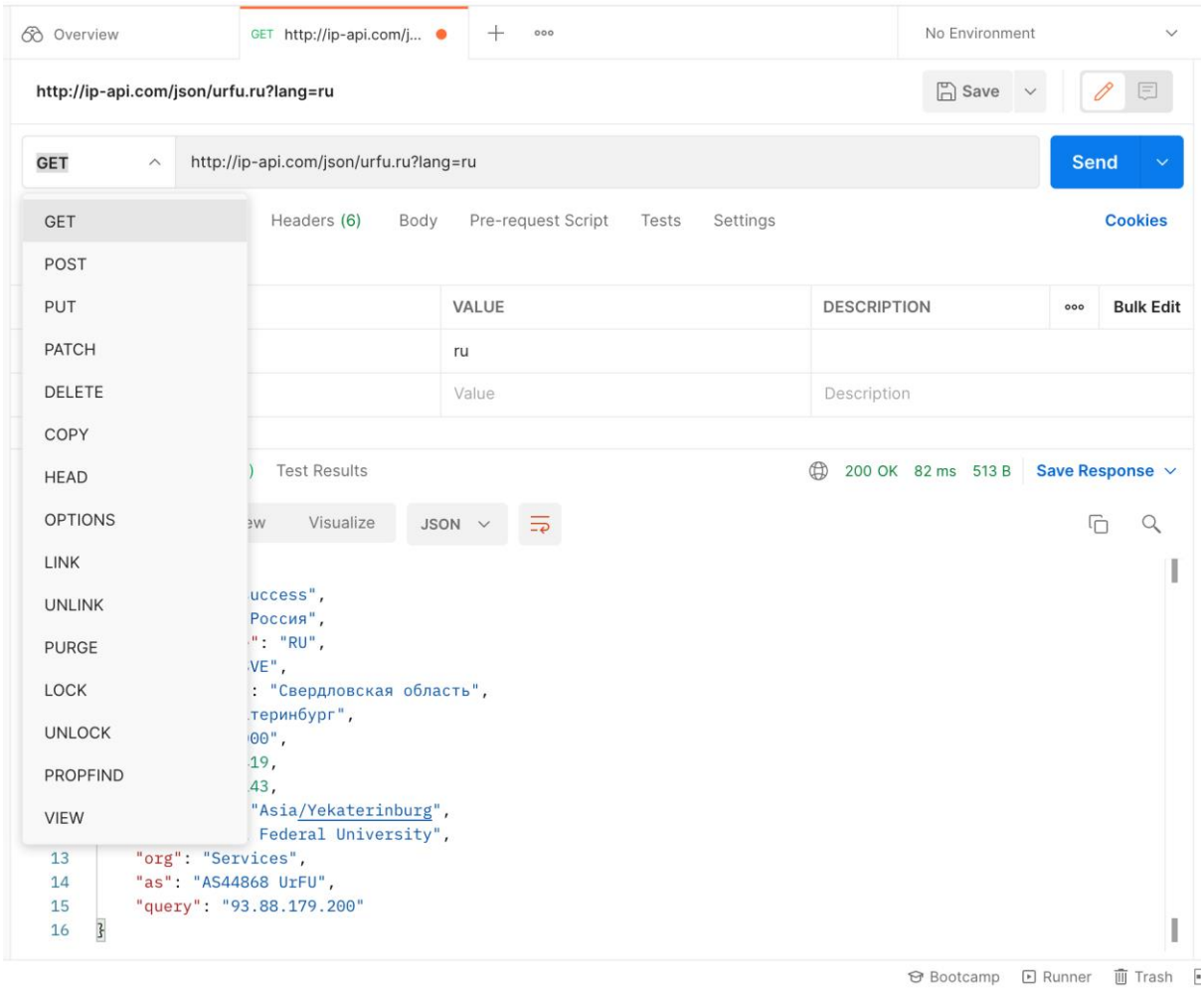
Вывод результата запроса к API в Postman

Параметры в Postman указываются в таблице под ссылкой для запроса. Вот пример указания параметра lang=ru. Сервер возвращает данные на русском языке.



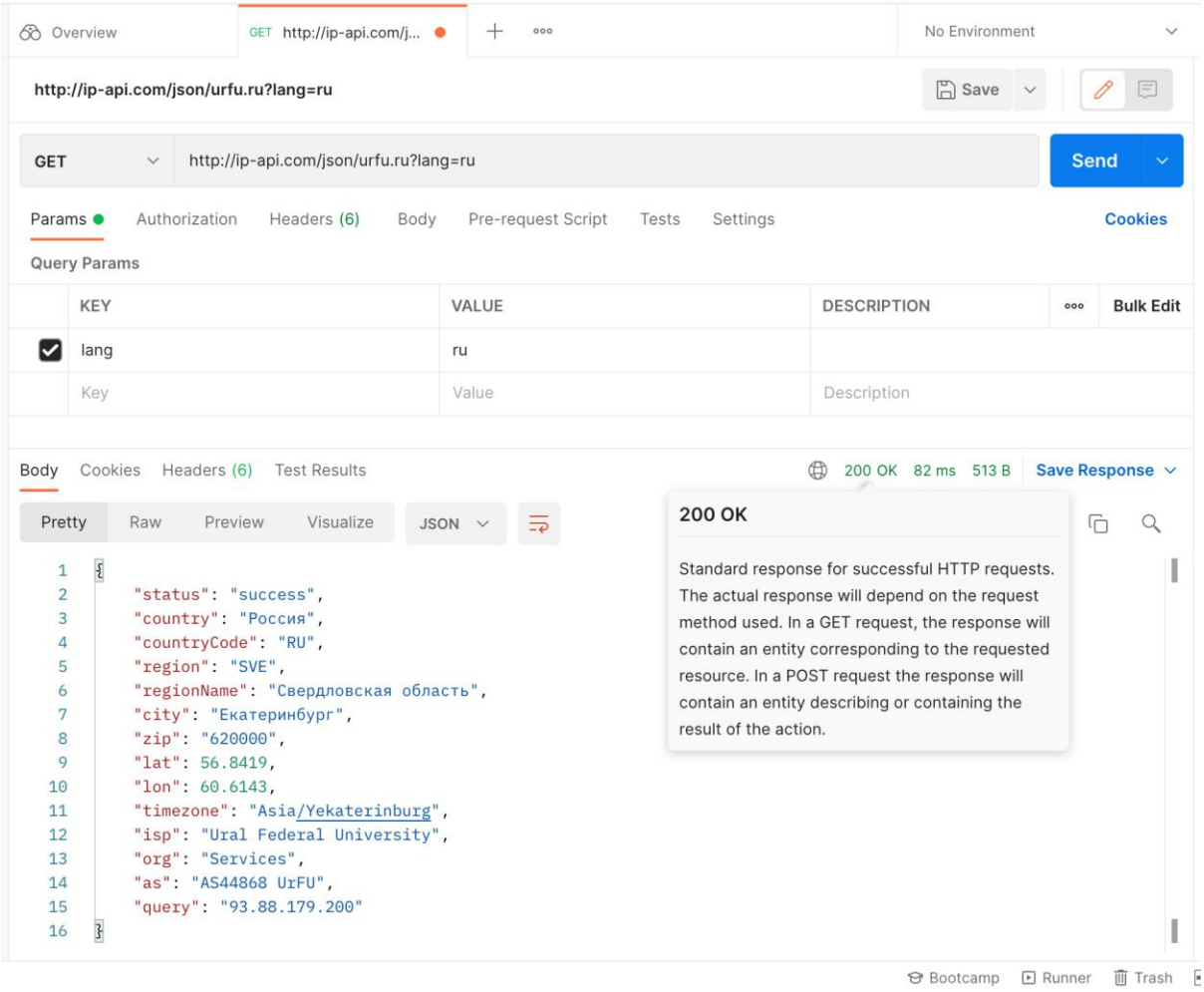
Пример указания параметров при запросе в Postman

Метод HTTP в Postman выбирается слева от URL. Список доступных методов достаточно большой, но, кроме GET и POST, все остальные методы почти не используются.



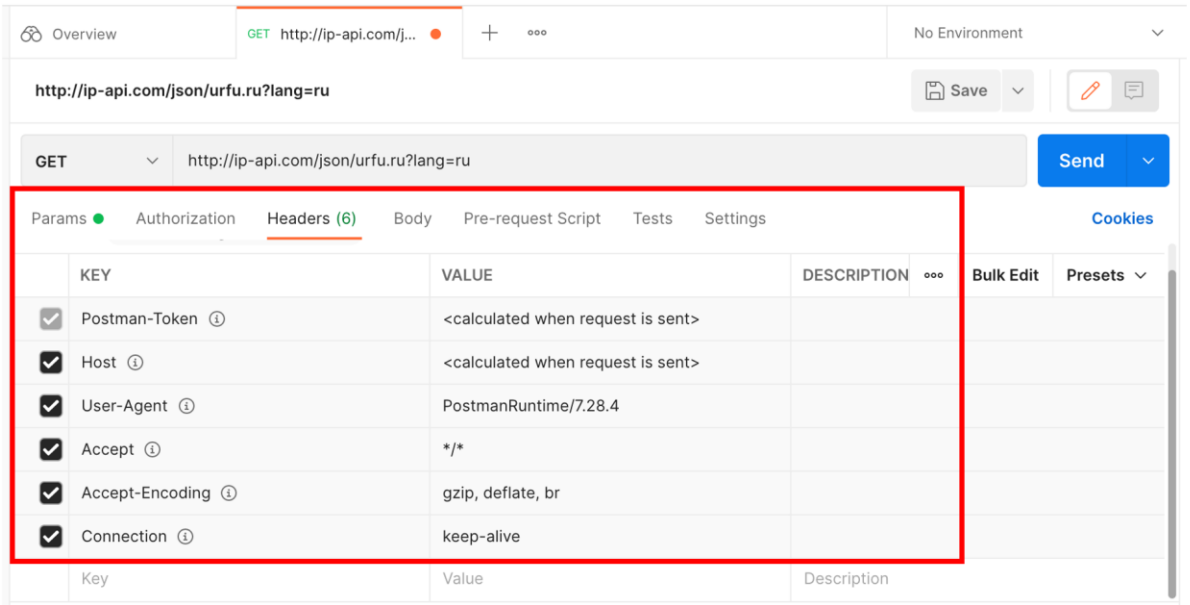
Выбор метода HTTP в Postman

Статус выполнения запроса HTTP показывается в правой верхней части окна результатов запроса. Если подвести мышь к статусу, то появится подсказка с описанием, что означает этот статус.



Статус выполнения HTTP запроса в Postman

Заголовки в Postman можно задать, если переключить таблицу под URL на вкладку Headers.



Заголовки HTTP запроса в Postman

Postman сохраняет запущенные запросы, включая URL и другие настройки (метод HTP, параметры, заголовки и др). Это удобно при постоянной работы с одним и тем же API. Особенно это полезно, когда вы работаете над своим API.

Итоги

- В этом модуле вы узнали, что такое архитектура приложений и какие архитектуры существуют.
- В настоящее время наиболее популярна микросервисная архитектура, в которой программная система состоит из отдельных сервисов, взаимодействующих друг с другом по сети с помощью API.
- Связь между приложениями по API организована с помощью протокола HTTP.
- Для работы с API существуют специальные инструменты, наиболее популярные из которых curl и Postman.

Практическое задание

Цель задания: научиться работать с API существующих систем.

Задание:

1. Сформируйте команду из 2-3 человек.
2. Изучите API какой-либо программной системы. Рекомендуемые API:
    - vk.com API – https://vk.com/dev/first_guide
    - HeadHunter API – https://dev.hh.ru/
    - YouTube Data API – https://developers.google.com/youtube/v3
    - Twitter API – https://developer.twitter.com/en/docs/twitter-api

   Можно также использовать любую другую систему по вашему выбору.
3. Проведите эксперименты с API выбранной системы с помощью curl или Postman.
4. Составьте отчет о работе с примерами 2-3 наиболее интересных запросов к API.

Термины

Все потенциально-незнакомые и новые понятия выдели отдельно и “положи” сюда.

Список источников

- Brian Foote, Joseph Yoder. Big Ball of Mud. – http://www.laputan.org/mud/mud.html
- Mark Richards, Neal Ford. Fundamentals of Software Architecture: An Engineering Approach.
- Martin Fowler, James Lewis. Microservices – https://martinfowler.com/articles/microservices.html
- James Kurose, Keith Ross. Computer Networking: A Top-Down Approach.
- Курс “Компьютерные сети” – https://www.asozykin.ru/courses/networks_online

Дополнительные материалы

- curl – https://curl.se/
- Postman – https://www.postman.com/
- Web-версия Postman – https://web.postman.co/
- vk.com API – https://vk.com/dev/first_guide
- HeadHunter API – https://dev.hh.ru/
- YouTube Data API – https://developers.google.com/youtube/v3
- Twitter API – https://developer.twitter.com/en/docs/twitter-api

Модуль № 3

Название: Разработка API для приложений искусственного интеллекта

Образовательные результаты:

- Студенты знают библиотеку для разработки API FastAPI.
- Студенты умеют создавать API для приложений машинного обучения.
- Студенты умеют устанавливать необходимые пакеты Python для работы API модели машинного обучения.

В этом модуле:

Ранее вы познакомились с архитектурой современных приложений и исследовали открытые API крупных программных систем. В этом модуле вы научитесь создавать API для своего приложения машинного обучения.

В модуле вы:

- Узнаете, как устроена одна из самых популярных и простых библиотек для создания API в Python – FastAPI.
- Научитесь создавать API для модели машинного обучения.
- Познакомитесь с системой автоматической генерации документации для API в библиотеке FastAPI.
- Узнаете, как автоматизировать процесс установки необходимых для работы приложения машинного обучения пакетов в Python с помощью файла requirements.txt.
- Познакомитесь с виртуальными окружениями в Python и научитесь их использовать.

Модуль № 3. Юнит № 1. Библиотека FastAPI

В настоящее время существует несколько популярных библиотек для создания API и Web-приложения на Python, например, Django (https://www.djangoproject.com/), Flask (https://flask.palletsprojects.com/), FastAPI (https://fastapi.tiangolo.com/). Django и Flask позволяют создавать полноценные Web-сайты и Web-приложения, поэтому устроены достаточно сложно. FastAPI же рассчитана только на разработку API, поэтому использовать ее гораздо проще, чем Django, Flask и другие полноценные Web-библиотеки. Именно из-за простоты и удобства в использовании в этом курсе мы будем изучать FastAPI.

Установка FastAPI

Перед тем, как начать использовать библиотеку FastAPI, ее необходимо установить. Это делается с помощью запуска в командной строке команды:

```
pip install fastapi
```

Кроме самой библиотеки, необходимо установить Web-сервер, в котором будет работать приложение, использующее FastAPI. Совместно с FastAPI часто используют Web-сервер Uvicorn (https://www.uvicorn.org/), т.к. он

простой в использовании, но при этом обеспечивает высокую производительность. Установить Uvicorn можно следующей командой:

```
pip install uvicorn
```

Первый запуск FastAPI

Давайте создадим наш первый API с помощью FastAPI. На начальном этапе мы не будем использовать модель машинного обучения, а создадим простое приложение, которое выдает через API традиционное сообщение “Hello World!”.

Создайте файл main.py и включите в него следующий код:

```
from fastapi import FastAPI

app = FastAPI()


@app.get("/")
async def root():
    return {"message": "Hello World"}
```

В первой строке мы подключаем класс FastAPI из библиотеки fastapi. Затем мы создаем приложение FastAPI в переменной app. Название этой переменной нам понадобится, когда мы будем запускать это приложение.

Затем создается функция root(), которая выполняет одно простое действие: возвращает сообщение, которое мы хотим передать в формате JSON {"message": "Hello World"}. Ключевое слово async перед функцией нужно для того, чтобы обеспечить высокую производительность работы этой функции в сервере Uvicorn.

Минимальный вариант API готов, давайте запустим наше приложение в Web-сервер Uvicorn. В каталоге, где находится файл main.py, запустите в командной строке следующую команду:

```
uvicorn main:app
```

Здесь:

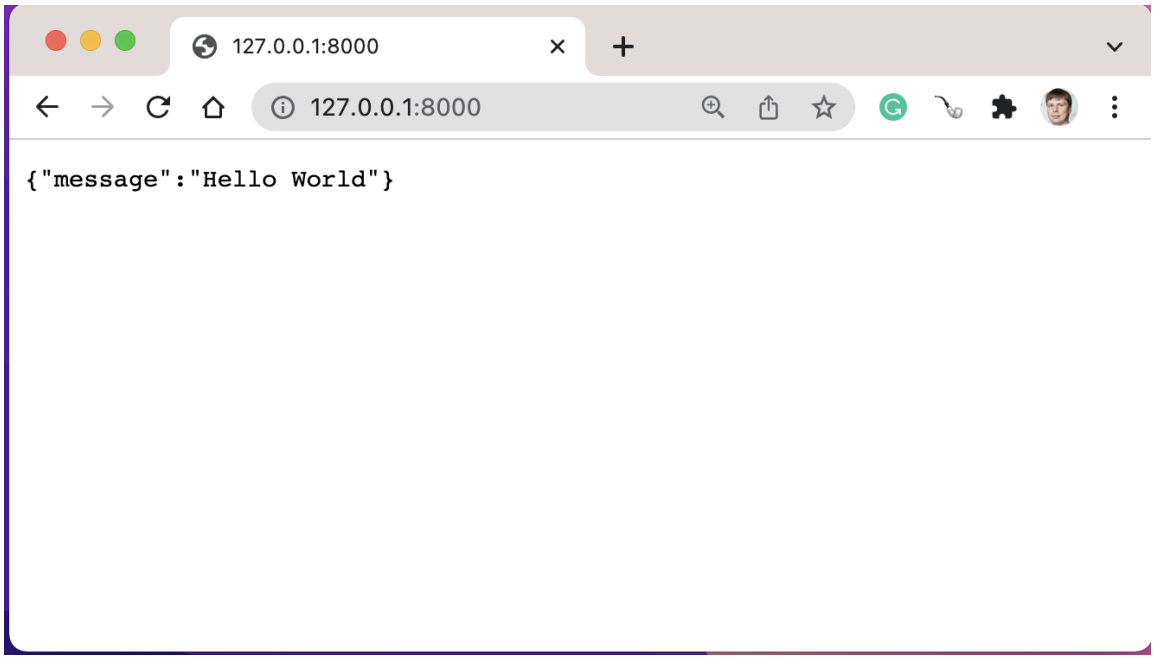
- uvicorn – команда для запуска Web-сервер Uvicorn.
- main – название Python файла с приложением FastAPI.
- app – название переменной FastAPI из файла main.py

В случае успешного запуска вы получите следующее сообщение:

```
andrey@MacBook-Pro-Andrej fastapi % uvicorn main:app
INFO:     Started server process [1406]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Запуск Web-сервера Uvicorn в командной строке

Сервер Uvicorn успешно запущен, указана ссылка, по которой он работает – http://127.0.0.1:8000. Давайте откроем эту ссылку в браузере:

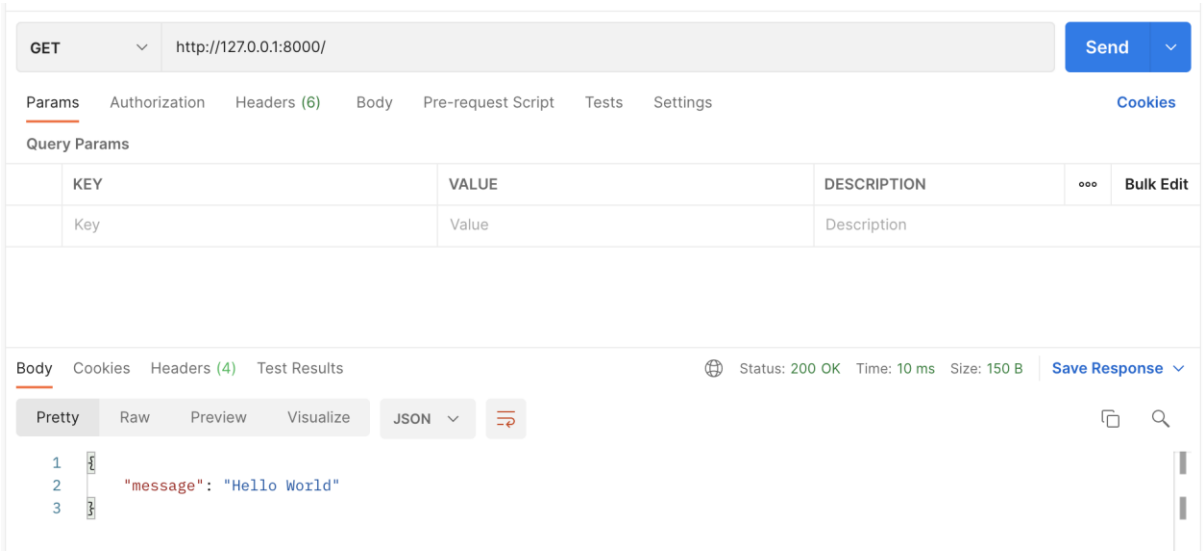


Сообщение “Hello World” от API в браузере

В окне браузера можно увидеть именно то сообщение в формате JSON, которое мы задали в функции root() в нашем простом API. Тот же самый ответ можно получить, если обратиться к ссылке http://127.0.0.1:8000 через curl или Postman.

```
andrey@MacBook-Pro-Andrej ~ % curl http://127.0.0.1:8000/
{"message":"Hello World"}%
andrey@MacBook-Pro-Andrej ~ %
```

Сообщение “Hello World” от API в curl



Сообщение “Hello World” от API в Postman

Как работает FastAPI

Давайте рассмотрим, каким образом взаимодействуют все компоненты созданного нами решения. Запросы обслуживает Web-сервер Uvicorn, мы подключаемся к корневому каталогу сервера, который указан при запуске: http://127.0.0.1:8000.

Как сервер Uvicorn узнает, что ему нужно показать при обращении к той или иной странице? Для этого сервер использует приложение на Python, которое мы ему указали при старте, и переменную с объектом FatsAPI в этом приложении. В нашем примере приложение хранится в файле main.py, а объект FastAPI находится в переменной app:

```
uvicorn main:app
```

Каким образом FastAPI узнает, какой ответ нужно возвращать на запросы к различным страницам? Для этого используется механизм декораторов в Python. Что такое декораторы и как они работают мы подробно рассмотрим

в следующем семестре, а сейчас просто будем их применять для корректной работы нашего API.

Декораторы в Python записываются перед определением функции и начинаются с символа @. Обратите внимание на декоратор перед определением функции root():

```
@app.get("/")
async def root():
    return {"message": "Hello World"}
```

Именно декораторы сообщают библиотеке FastAPI, какую функцию из приложения нужно вызывать для соответствующего запроса. В примере используется декоратор @app.get("/"), состоящий из следующих основных компонентов:

- @ – символ, говорящий о том, что далее указывается декоратор.
- app – имя переменной объекта FastAPI, который будет использоваться для обработки запросов API.
- get – название метода HTTP, который будет использоваться.
- ("/") – путь к запрашиваемому ресурсу.

Таким образом, декоратор @app.get("/") перед определением функции root() говорит о том, что если к Web-серверу поступит запрос GET к корневому каталогу Web-сервера (в нашем примере запрос к корневому каталогу сервера выглядит так – http://127.0.0.1:8000), то FastAPI должен запустить функцию root() и вернуть результат ее работы клиенту, который отправлял запрос к серверу Uvicorn. Именно такой ответ мы видели в клиентах различных типов: браузере, curl и Postman.

В этом юните вы узнали, как использовать библиотеку FastAPI. В следующем юните мы применим ее для создания API для собственного простого приложения машинного обучения.

Итоги

- FastAPI – одна из самых простых и удобных в использовании библиотек для создания API на Python.
- Для запуска приложений API используется Web-сервер Uvicorn.

- В приложении FastAPI применяются декораторы, чтобы указать, какую функцию приложения вызвать для подготовки ответа на различные запросы к серверу.

Модуль № 3. Юнит № 2. Создание API для приложения машинного обучения

В предыдущем юните вы познакомились с библиотекой FastAPI и создали приложение, возвращающее сообщение “Hello World”. В этом юните мы создадим API для модели машинного обучения. В качестве примера мы будем использовать модель из библиотеки Hugging Face, которая определяет тональность текста.

Проектирование API

Важным этапом разработки API является проектирование. На этом этапе необходимо определить, какие методы будут входить в API, и по каким правилам они будут использоваться.

На первом этапе создания модели мы разработаем только один метод, который будет использоваться для запуска этой модели. Традиционно, такой метод называется predict. Мы будем использовать именно такое название. Ссылка для вызова методы predict на Web-сервере Uvicron, запущенном локально на компьютере, будет выглядеть следующим образом:

http://127.0.0.1:8000/predict

Реализация API

Файл main.py для приложения, которое определяет тональность текста, будет выглядеть следующим образом:

```
from fastapi import FastAPI
from transformers import pipeline

app = FastAPI()
classifier = pipeline("sentiment-analysis")
```

```
@app.get("/")
def root():
    return {"message": "Hello World"}

@app.get("/predict/")
def predict():
    return classifier("I like machine learning engineering!")
```

1. В начале файла мы подключаем не только библиотеку FastAPI, но и pipeline из библиотеки Hugging Face.

```
from fastapi import FastAPI
from transformers import pipeline
```

2. Затем создается объект FastAPI и классификатор из библиотеки Hugging Face на основе пайплайна с типом “sentiment-analysis”.

```
app = FastAPI()
classifier = pipeline("sentiment-analysis")
```

3. На следующем этапе в файле определены две функции, обрабатывающие запросы к разным адресам. Функция root обрабатывает запросы к корневому каталогу сервера, она не изменилась с предыдущего юнита. Вторая функция predict используется для реализации нашего нового метода API – predict.

```
@app.get("/predict/")
def predict():
    return classifier("I like machine learning engineering!")[0]
```

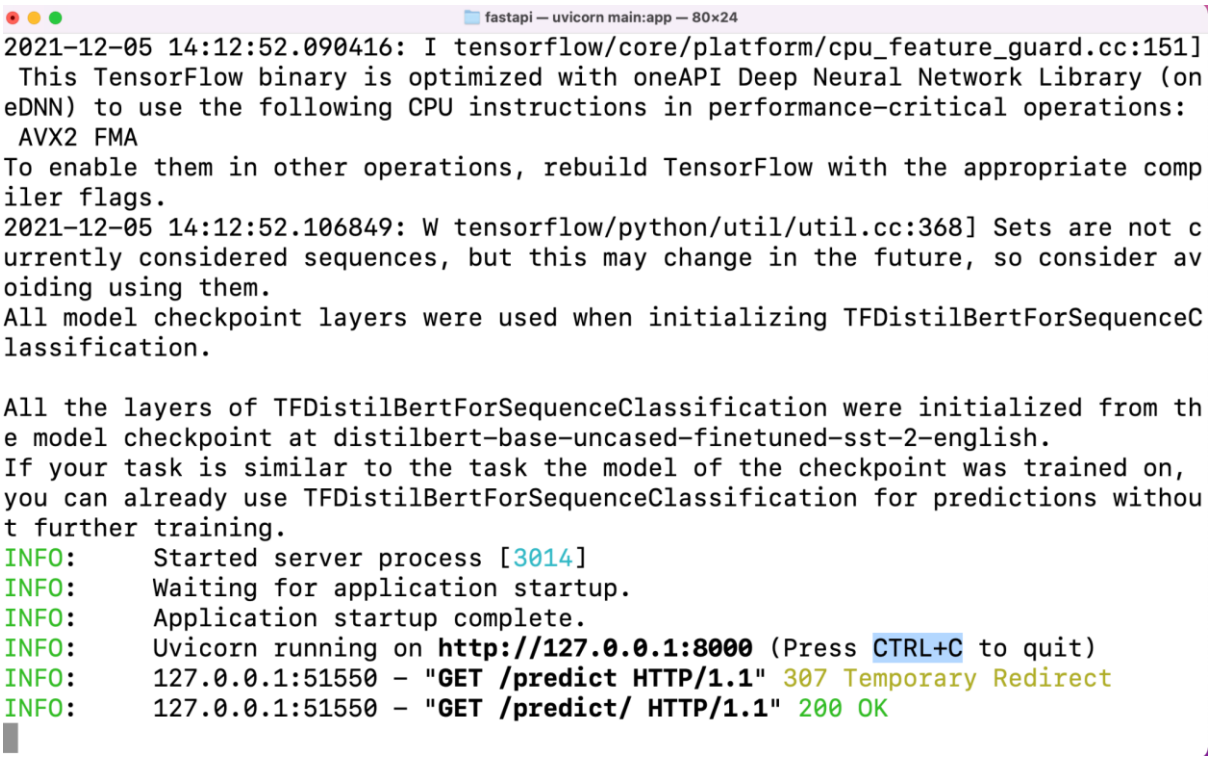
В этой функции вызывается классификатор, в который передается строка для определения тональности. Результат работы классификатора возвращается из функции.

Декоратор @app.get("/predict/") говорит о том, что функцию predict нужно вызывать при получении запроса HTTP GET к каталогу "/predict/" на сервере.

4. Давайте запустим обновленное приложение с новым методом в сервере Uvicorn. Если сервер все еще работает с предыдущего запуска, его можно остановить клавишами CTRL+C. Запускаем сервер точно так же, как делали ранее:

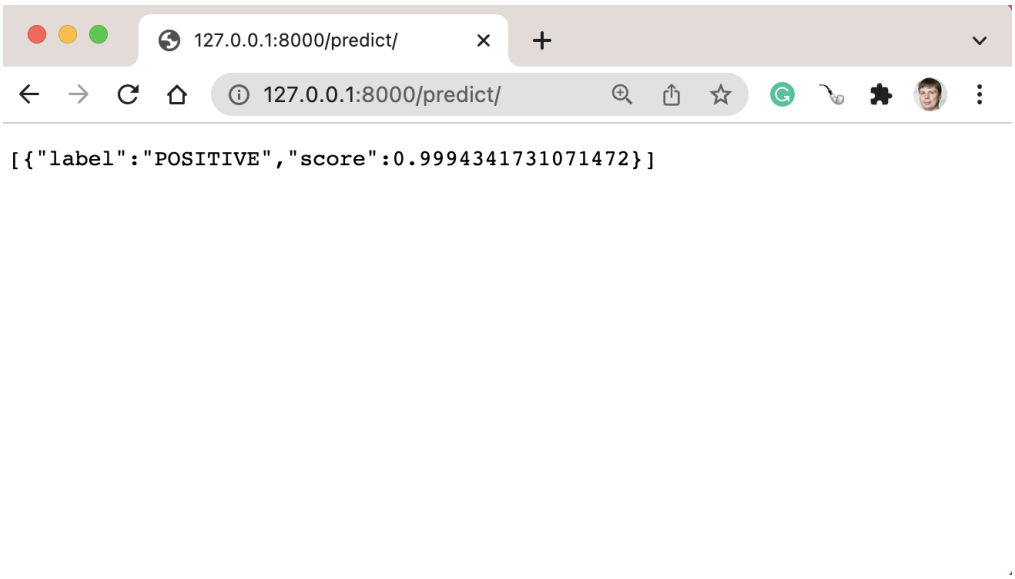
```
uvicorn main:app
```

В этот раз запуск будет выполняться дольше, т.к. необходимо загрузить модель машинного обучения. Также при запуске могут печататься диагностические сообщения от библиотек TensorFlow или Torch, которые используются для работы модели. Такие сообщения можно игнорировать. Главное, чтобы в итоге появилось сообщение “Uvicorn running on http://127.0.0.1:8000”, которое говорит об успешном запуске сервера.



Запуск сервера Uvicorn с загрузкой модели машинного обучения (диагностические сообщения от TensorFlow можно игнорировать).

5. Сервер запущен, теперь давайте обратимся к нашему новому методу predict через браузер. Для этого нужно открыть ссылку http://127.0.0.1:8000/predict. Результат выглядит следующим образом:



Результат выполнения метода API predict в браузере

Можно видеть, что в этот раз в результате вызова метода API возвращается не фиксированное значение, которое мы прописали в функции, а результат работы модели машинного обучения по распознаванию тональности текста. Ответ в формате JSON включает два поля:

- label:POSITIVE – тональность текста положительная.
- score:0.9994341731071472 – модель уверена в своем решении более, чем на 99%.

Эксперименты с API для модели

Попробуйте провести следующие эксперименты с моделью через API:

- обратиться к методу predict API через curl или postman.
- определить тональность другого текста с помощью вызова API.

Наверняка вы обратите внимание, что заменить текст для определения тональности достаточно сложно. Приходится останавливать сервер Uvicorn, изменять текст для распознавания в файле main.py, перезапускать сервер с новым файлом приложения. На практике так обычно не делают, вместо этого текст или другие необходимые данные передают в приложение в качестве параметра. В следующем модуле вы узнаете, как использовать

параметры в FastAPI и обновите приложение так, чтобы оно могло определять тональность разных текстов без необходимости перезапуска.

Итоги

- Мы разработали новый метод для API, который будет использоваться для запуска модели – predict.
- Метод вызывается по следующей ссылке – http://127.0.0.1:8000/predict/
- В приложение добавлена новая функция predict, которая вызывается при запросе HTTP GET к каталогу /predict на сервере.
- Текст для определения тональности в текущем варианте реализации вставлен прямо в код функции predict, что неудобно.

Тест

1. Как выглядит декоратор в FastAPI, который нужно записать перед функцией, предназначенной для обработки запросов GET к методу predict в API:
    - @app.post("/predict/")
    - **@app.get("/predict/")**
    - @app.get("/")
    - @app.get("/docs/")
2. С помощью каких инструментов можно вызывать API приложения машинного обучения, созданного с помощью библиотеки FastAPI (несколько вариантов ответа):
    - **Браузер**
    - **curl**
    - **Postman**
    - Google Colab

Модуль № 3. Юнит № 3. Передача параметров в API

В прошлом юните мы разработали метод predict, который запускает модель машинного обучения для определения тональности текста, написанного прямо в коде приложения. В этом юните вы узнаете, как научить API определять тональность любого текста с использованием параметров запроса к API.

Вспомним, что в протоколе HTTP, который используется для работы API, есть два варианта передачи параметров от клиента к серверу. В случае использования метода HTTP GET параметры передаются на сервер в строке URL. Такой подход хорошо работает, когда значения параметров небольшие. Альтернативный вариант используется в метода HTTP POST: параметры передаются в теле запроса HTTP. Мы хотим, чтобы API можно было использовать для определения тональности текстов разной длины, поэтому будем применять метод POST и передавать текст в теле сообщения.

Обновление приложения для передачи параметров

Модифицируем код в файле main.py следующим образом:

```
from fastapi import FastAPI
from transformers import pipeline
from pydantic import BaseModel

class Item(BaseModel):
    text: str

app = FastAPI()
classifier = pipeline("sentiment-analysis")

@app.get("/")
def root():
    return {"message": "Hello World"}

@app.post("/predict/")
def predict(item: Item):
    return classifier(item.text )[0]
```

Параметры, которые передаются в API, могут быть разные. Кроме того, некоторые методы API могут требовать для корректной работы не один, а несколько параметров. Поэтому нам необходимо сообщить FastAPI, какие именно параметры и какого типа мы ожидаем. Для этой цели мы будем использовать библиотеку pydantic (https://pydantic-docs.helpmanual.io/), которая занимается автоматической проверкой формата и типа данных. Добавляем подключение библиотеки pydantic:

```
from fastapi import FastAPI
from transformers import pipeline
from pydantic import BaseModel
```

pydantic использует модели данных для того, чтобы задать, какие именно данные требуются и какого типа. Мы импортируем из pydantic базовый вариант класса для моделей BaseModel и создаем на его основе свою модель:

```
class Item(BaseModel):
    text: str
```

Класс нашей модели называется Item(элемент) и он содержит единственное поле с названием text строкового типа (str). Именно в параметре text мы и будем передавать текст, для которого нужно определить тональность.

В функцию predict внесены три изменения. Во-первых, в качества параметра в функцию predict передается объект item типа Item (класс для наших данных на основе базовой модели из pydantic):

```
def predict(item: Item):
```

Во-вторых, классификатор вызывается не для заранее прописанного текста, а для поля text объекта item, который берется из параметра запроса HTTP к серверу:

```
    return classifier(item.text)[0]
```

И третье изменение связано с типом метода HTTP, который мы будем обрабатывать. Так как большие тексты в качестве параметра удобнее передавать в теле запроса HTTP, то мы будем использовать метод POST. Поэтому в декораторе перед определением функции predict метод get заменяем на post:

```
@app.post("/predict/")
```

Изменение приложения завершено, можно перезапустить сервер Uvicorn:

```
uvicorn main:app
```

Вызов API с параметрами

Вызвать наш новый вариант метода predict из адресной строки браузера не получится, т.к. нам требуется сформировать тело запроса с текстом для определения тональности. Давайте составим нужный запрос с помощью curl:

```
curl -X 'POST' \
  'http://127.0.0.1:8000/predict/' \
  -H 'Content-Type: application/json' \
  -d '{
  "text": "I hate machine learning engineering!"
}'
```

В этом запросе используются следующие компоненты:

- -X 'POST' – параметр, который указывает curl использовать метод POST HTTP.
- http://127.0.0.1:8000/predict/ – адрес метода API.
- -H 'Content-Type: application/json' – параметр, который устанавливает заголовок, определяющий тип тела запроса (application/json).
- -d '{ "text": "I hate machine learning engineering!"}' – данные для тела запроса в формате JSON. Данные содержат одно поле, заголовок "text" (именно этот заголовок указан в модели pydantic в нашем коде), значение "I hate machine learning engineering!".

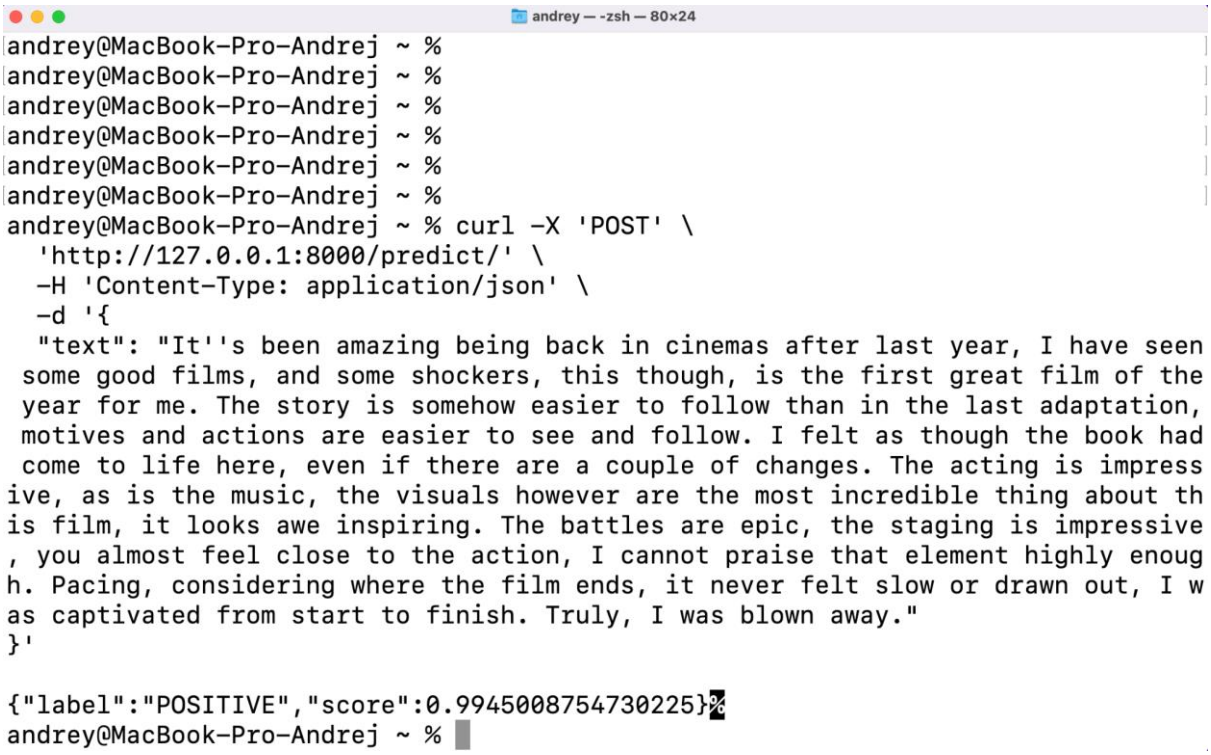
Так выглядит результат выполнения команды curl в терминале:

```
andrey@MacBook-Pro-Andrej ~ % curl -X 'POST' \
  'http://127.0.0.1:8000/predict/' \
  -H 'Content-Type: application/json' \
  -d '{
  "text": "I hate machine learning engineering!"
}'
{"label":"NEGATIVE","score":0.9995560050010681}%
andrey@MacBook-Pro-Andrej ~ %
```

Результат выполнения запроса к API модели определения тональности с передачей текста в виде параметра

Результата работы модели: тональность отрицательная, уверенность модели более 99%.

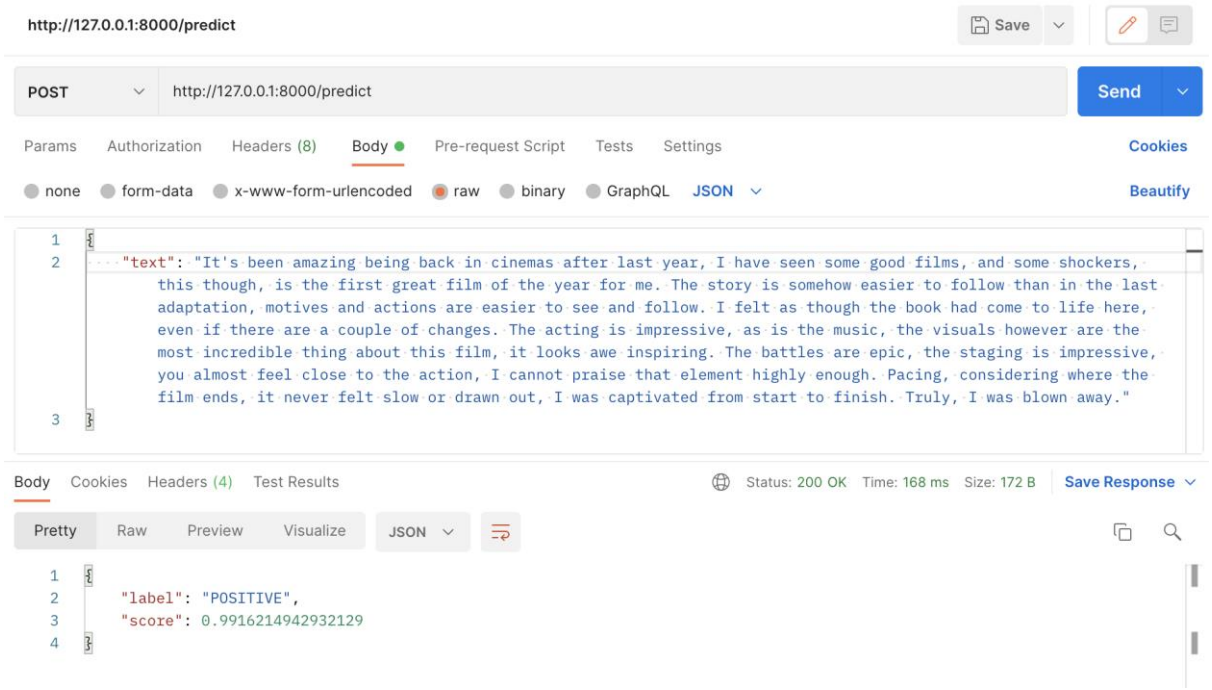
Давайте попробуем запустить запрос, который содержит текст большего размера, например, рецензию на фильм Дюна с сайта IMDB (https://www.imdb.com/review/rw7475243/?ref_=tt_urv).



Результат выполнения запроса к API с длинным текстом рецензии с помощью curl

Рецензия на сайте IMDB содержит 10 звезд из 10, тональность текста положительная. Именно такой результат выдала модель: тональность положительная, уверенность в результате больше 99%.

Теперь давайте запустим запрос к нашему API с помощью Postman. Вот так будет выглядеть окно запроса:



Запрос к API модели определения тональности текстов в Postman

Как и в curl, необходимо выбрать метод HTTP запроса POST, затем перейти в закладку Body (тело запроса), выбрать тип данных raw (необрабатываемые данные) и формат данных: JSON. Затем в поле ввода нужно вставить данные в формате JSON: ключ “text”, значение – текст, для которого нужно определить тональность.

Для запуска запроса нажимаем на кнопку “Send”. Результат выполнения появится в нижней части экрана.

Итоги

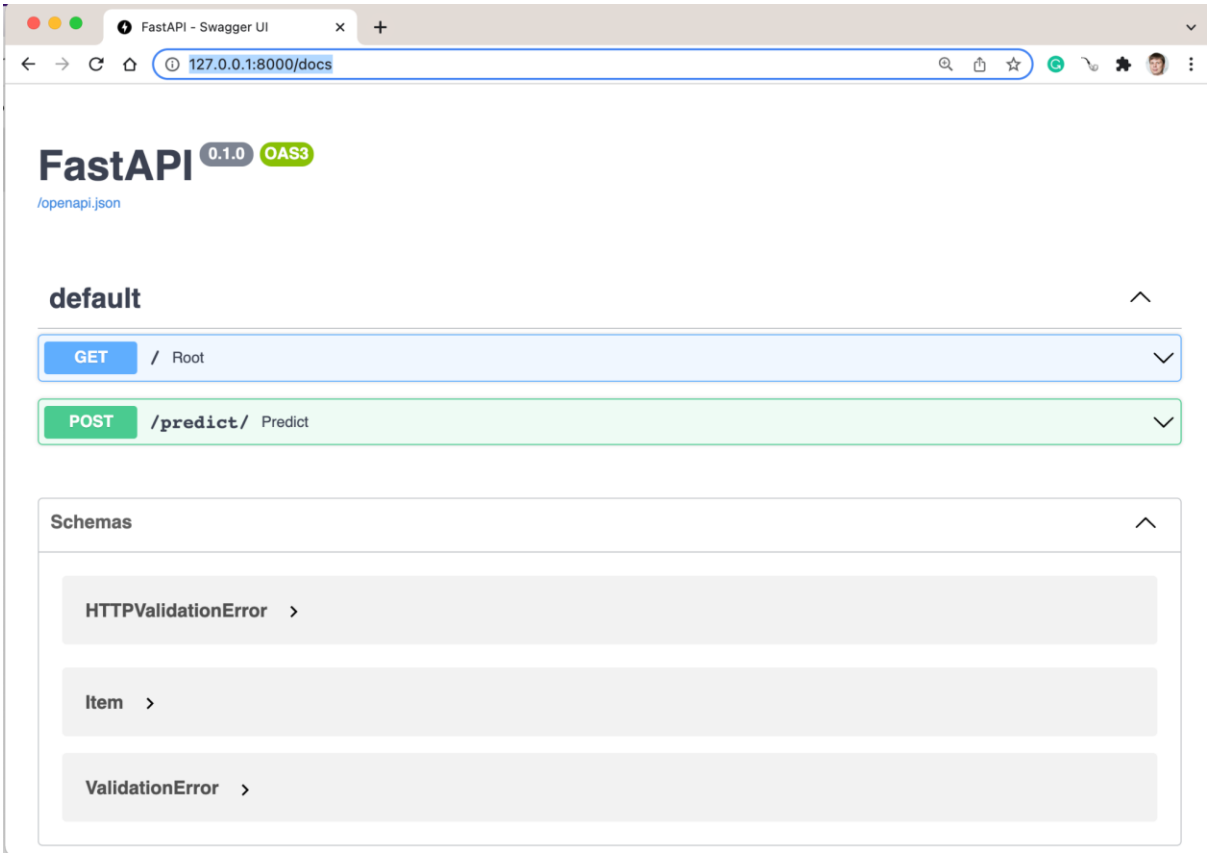
- Мы научились создавать API, который обрабатывает не фиксированный текст, а любой, который будет передан в качестве параметра.
- Для передачи параметров мы использовали метод POST и тело запроса. В теле запроса параметр text передается в формате JSON.
- Чтобы использовать параметры в коде приложения, мы применили модель данных из библиотеки pydantic. Модель содержит всего одно поле: text. В этом поле передается текст, который нужно обработать.
- Классификатор в коде вызывается не для фиксированного текста, а для поля text модели данных.
- Для вызова API с помощью метода POST и передачи параметров в теле запроса мы использовали curl и Postman.

Тест

1. Какой метод HTTP используется для передачи параметров в теле запроса:
    - **POST**
    - GET
    - HEAD
    - DELETE
2. Какой параметр в curl используется для указания данных в теле запроса:
    - -X
    - -H
    - **-d**
    - -p

Модуль № 3. Юнит № 4. Автоматическая документация API в FastAPI

Библиотека FastAPI не только создает API для нашего приложения, но еще и автоматически генерирует документацию для него. Документацию можно увидеть, если перейти по ссылке http://127.0.0.1:8000/docs. Страница с документацией выглядит следующим образом:

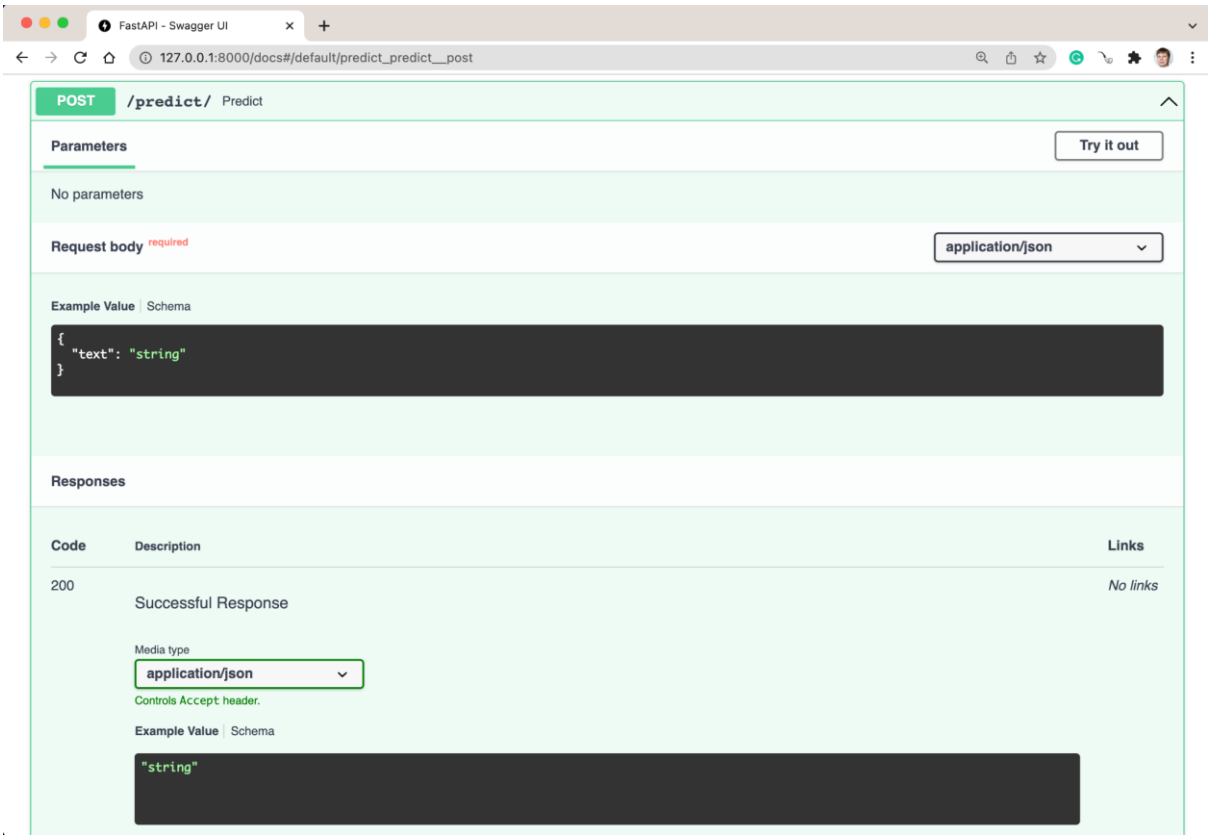


Страница с автоматически сгенерированной документацией для API

Методы API показаны в верхней части. Всего наш API содержит два метода:

- Корневой каталог /, HTTP GET – наш первый метод, который возвращает сообщение “Hello world”.
- Метод predict, HTTP POST – метод для определения тональности текста.

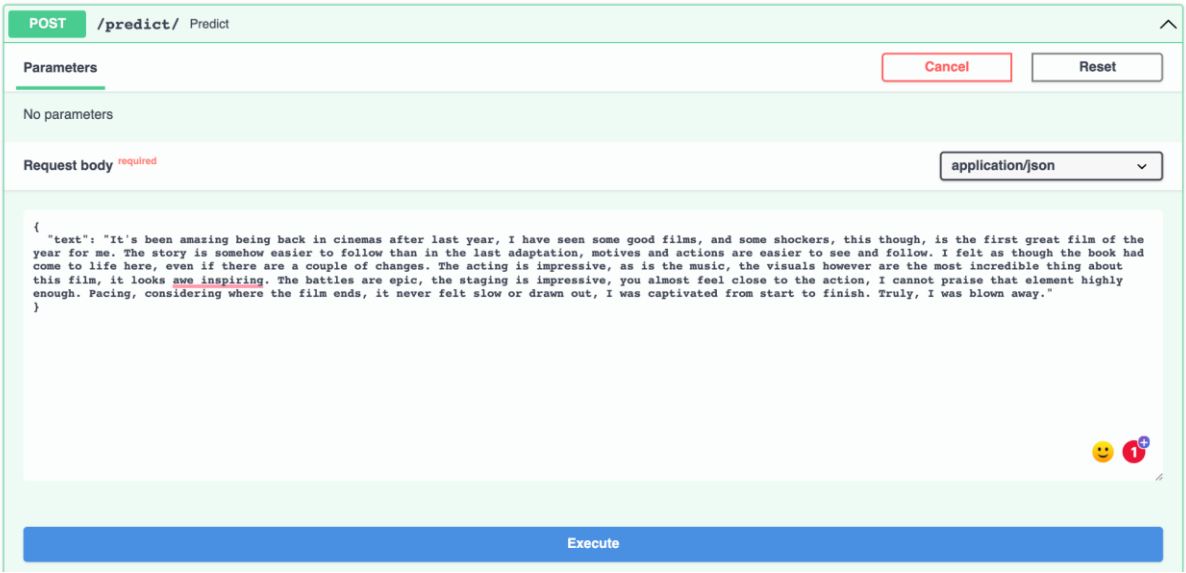
Можно открыть метод predict и посмотреть его более подробную документацию.



Автоматически сгенерированная документация для метода predict

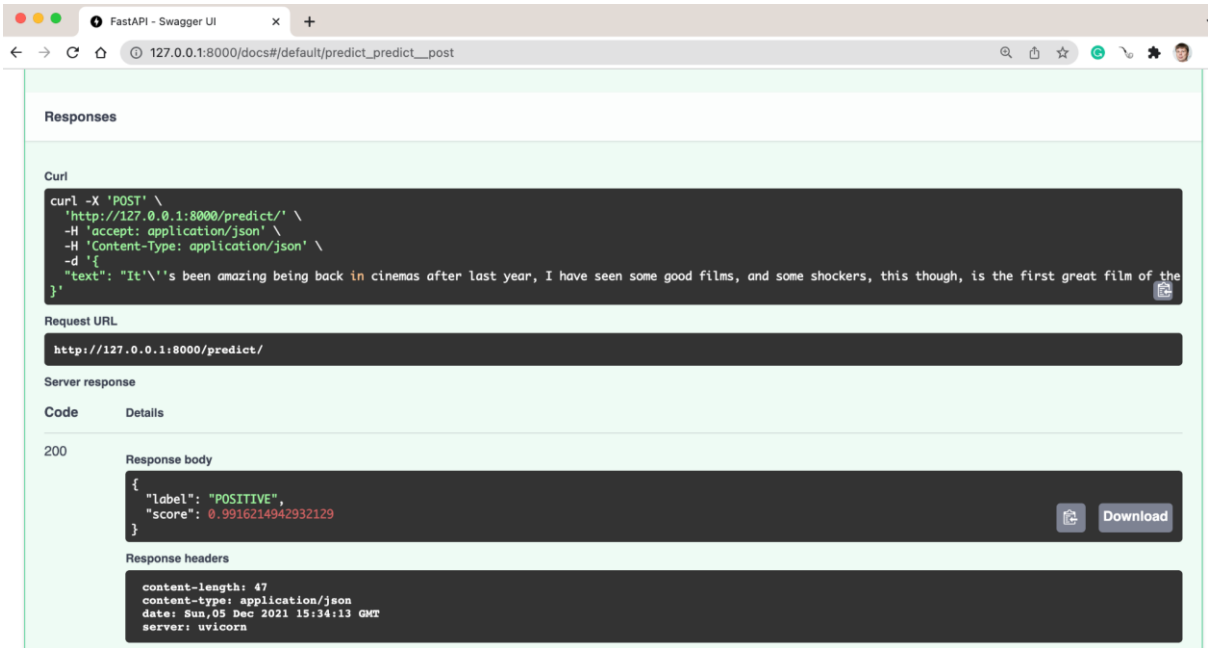
В документации видно, что для вызова данного метода обязательно требуется тело запроса (Request body), которое должно содержать значение в формате JSON, и в нем должен быть ключ “text” с соответствующим значением в текстовом формате. Результат также выдается в формате JSON.

Однако на этом возможности автоматически сгенерированной документации не заканчиваются! Вы можете нажать кнопку “Try it out” и попробовать вызвать API прямо в интерфейсе документации. Откроется окно заполнения тела запроса и появится кнопка “Execute”.



Запуска запроса к API через интерфейс документации FastAPI

При нажатии кнопки “Execute” запрос выполнится и покажется ответ. Кроме того, будет сформирована команда curl, которую можно использовать для запуска такого запроса. Если вы пока не очень хорошо научились применять curl, то удобно использовать автоматическую документацию, чтобы сформировать нужную команду.



Результат запуска запроса в интерфейсе документации к API

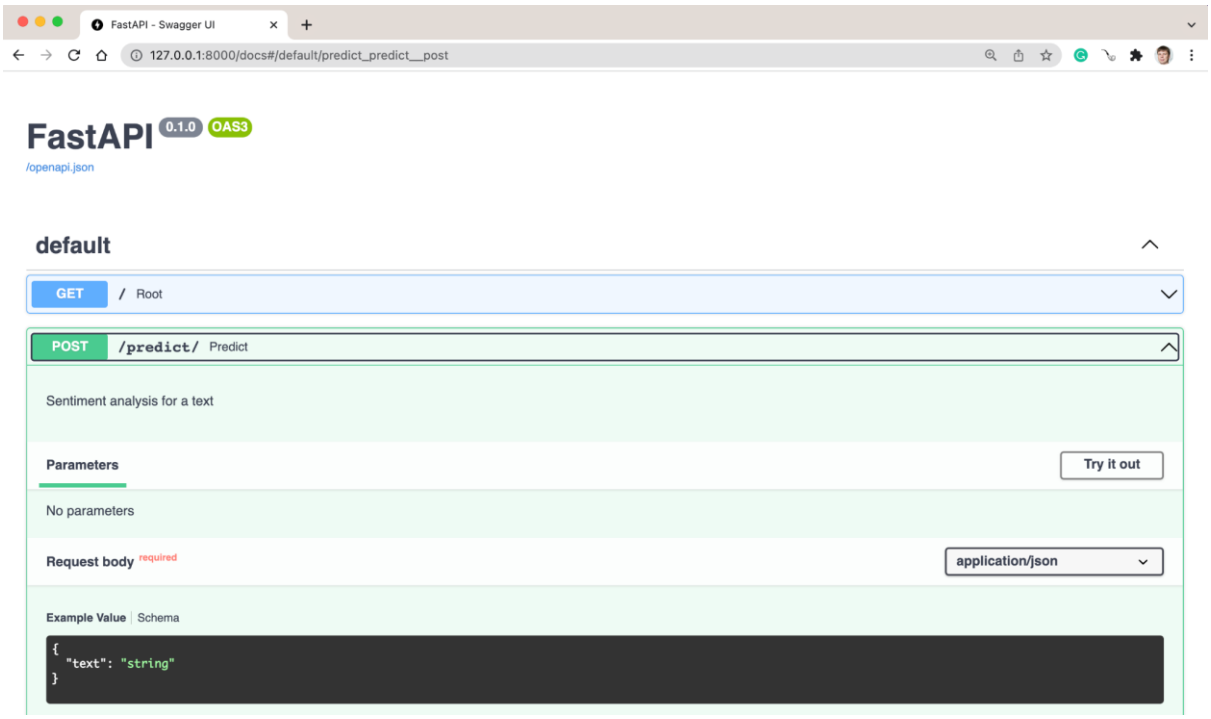
Вы можете помочь FastAPI составить автоматическую документацию. docstring, которые вы вставите в вашу программу на Python, будут

перенесены в документацию. Например, давайте добавим строку описания к функции predict:

```
@app.post("/predict/")
def predict(item: Item):
    """Sentiment analysis for a text"""

    return classifier(item.text)[0]
```

После изменения файла main.py перезапустите серве Uvicorn и описание функции будет добавлено к автоматически сгенерированной документации:



docstring из функции predict вставлена в автоматически сгенерированную документацию документацию для метода predict

Рекомендуется писать более подробную документацию для функций. Тогда разработчикам будет значительно проще использовать ваш API. Ведь вы создаете API именно для того, чтобы другие люди могли использовать ваши модели!

Итоги

- Библиотека FastAPI автоматически создает документацию на методы API.

- Для каждого метода в документации показаны входные параметры и выходные значения.
- Из интерфейса документации можно запустить запрос к API и посмотреть, как он работает.
- При запуске запроса API в интерфейсе документации формируется команда curl, которая выполняет такой запрос. Команду можно использовать, чтобы самостоятельно запускать интересующий запрос в дальнейшем.
- В автоматически сгенерированную документацию включается документация, которую вы создали в Python файле с помощью docstring.

Тест

1. По какой ссылке можно обратиться к автоматически сгенерированной документации FastAPI на локальном компьютере?
    - **http://127.0.0.1:8000/docs**
    - http://127.0.0.1:8000/
    - http://127.0.0.1:8000/test
    - http://127.0.0.1:8000/open_api
2. Как включить нужную вам информацию в автоматически сгенерированную документацию API?
    - FastAPI генерирует документацию автоматически, что-то изменить в ней нет возможности.
    - **Включить нужную информацию в файл Python приложения в формате docstring.**
    - Отредактировать файлы документации в каталоге docs сервера uvicorn.
    - Открыть в браузере ссылку http://127.0.0.1:8000/docs и внести необходимую информацию через интерфейс редактирования.

Модуль № 3. Юнит № 5. Установка библиотек в Python

Приложение, которое мы создали в этом модуле, использует достаточно большое количество библиотек Python. Чтобы приложение успешно работало, все эти библиотеки должны быть установлены в системе.

Ручная установка библиотек

Библиотеки можно установить вручную с помощью команды pip install в командной строке. Ранее мы именно так и делали. Чтобы установить все необходимые для нашего приложения библиотеки необходимо выполнить следующие команды:

```
pip install fastapi
pip install uvicorn
pip install transformers[torch]
```

Обратите внимание, что эти команды устанавливают не только три библиотеки fastapi, uvicorn и transformers, но и все их зависимости.

Более короткий вариант – установка всех необходимых пакетов с помощью одной команды:

```
pip install fastapi, uvicorn, transformers[torch]
```

Вручную пакеты устанавливать удобно, когда их немного и когда вы точно знаете, какие пакеты требуются. Список необходимых пакетов можно включить в документацию по установке, чтобы другие разработчики, которые будут использовать вашу систему, знали, что нужно устанавливать.

Однако даже при наличии документации, есть возможность ошибиться при установке пакетов. Кроме того, документация может устаревать и через некоторое время в ней будет указан неправильный перечень пакетов. Поэтому удобнее использовать средства Python, специально предназначенные для автоматической установки всех необходимых для приложения пакетов.

Установка библиотек с помощью файла requirements.txt

В Python есть инструменты для автоматической установки всех необходимых для корректной работы приложения пакетов. Для этого список пакетов нужно включить в файл с названием requirements.txt. Вот так будет выглядеть содержимое файла requirements.txt для нашего приложения:

```
fastapi
uvicorn
transformers[torch]
```

Названия пакетов для установка в файле requirements.txt указываются по одному в строке.

Установка пакетов, перечисленных в файле requirements.txt, выполняется запуском следующей команды в командной строке:

```
pip install -r requirements.txt
```

Строго говоря, вместо requirements.txt можно использовать любое имя файла. Однако requirements.txt является общепринятым, и если разработчик видит такой файл в репозитории с кодом приложения, он понимает, что для работоспособности приложения необходимо установить все пакеты, указанные в этом файле.

Кроме того, файл requirements.txt используется различными инструментами автоматического развертывания приложений. В следующем модуле мы будем изучать такие инструменты, которые позволяют автоматически развернуть приложение из репозитория на GitHub в облаке Heroku.

Узнать, какие именно пакеты нужны для работы приложения и правильно сформировать файл requirements.txt можно, запустив в командной строке следующую команду:

```
pip freeze > requirements.txt
```

Команда pip freeze выводит список всех установленных дополнительных пакетов. С помощью символа > производится перенаправление вывода команды pip freeze в файл requirements.txt.

Виртуальные окружения

Организация работы с пакетами в Python реализована способом, который не всегда подходит для крупномасштабной разработки. В частности, нет

возможности установить и использовать несколько версий одной библиотеки. Однако может возникнуть ситуация, когда вы работаете над двумя проектами одновременно, и каждому проекту нужна разная версия библиотеки. Для решения этой проблемы в Python используются виртуальные окружения.

Виртуальное окружение (или venv, сокращение от virtual environment) – это изолированная среда для проекта на Python, которая может содержать собственную копию дополнительных пакетов, которые не зависят от других виртуальных окружений и версии Python, установленной в операционной системе. Таким образом, для проектов, которые требуют определенных версий библиотек, можно создавать и использовать виртуальные окружения.

Для создания виртуального окружения нужно выполнить следующую команду в командной строке:

```
python3 -m venv /Users/andrey/pyton_venv/fast_api
```

В этой команде параметр -m venv говорит Python о том, что нужно создать виртуальное окружение, а /Users/andrey/pyton_venv/fast_api – это путь в котором создается виртуальное окружение. В этот каталог будут устанавливать пакеты для виртуального окружения. Именно благодаря отдельному каталогу достигается изоляция виртуальных окружений друг от друга.

Чтобы использовать виртуальное окружение, его нужно активировать. В Linux и MacOS для активации окружения необходимо в командной строке запустить команду:

```
source /Users/andrey/pyton_venv/fast_api/bin/activate
```

В Windows активация выполняется запуском следующей команды:

```
fast_api/Scripts/activate
```

После активации виртуального окружения в приглашении командной строки появится название этого виртуального окружения.

```
andrey@mbp-andrej ~ % source /Users/andrey/pyton_venv/fast_api/bin/activate
(fast_api) andrey@mbp-andrej ~ % which python
/Users/andrey/pyton_venv/fast_api/bin/python
(fast_api) andrey@mbp-andrej ~ %
```

Пример активация виртуального окружения с названием fast_api

Если вы запускаете команды по установке пакетов (pip install) после активации виртуального окружения, то пакеты будут устанавливаться в это окружение, а не в python, который установлен на уровне операционной системы. Поэтому можно устанавливать специфические для проекта библиотеки не боясь, что это приведет к потере работоспособности других проектов или системы в целом.

Выйти из виртуального окружения можно, запустив в командной строке команду

```
deactivate
```

После деактивации приглашение операционной системы вернется к своему обычному виду.

```
andrey@mbp-andrej ~ % source /Users/andrey/pyton_venv/fast_api/bin/activate
(fast_api) andrey@mbp-andrej ~ % which python
/Users/andrey/pyton_venv/fast_api/bin/python
(fast_api) andrey@mbp-andrej ~ % deactivate
andrey@mbp-andrej ~ %
```

Итоги

- Создать API для модели машинного обучения удобно с помощью библиотеки FastAPI на Python.
- Данные для обработки в модель машинного обучения передаются с использованием параметров в теле запроса POST HTTP.
- Вызывать API для приложения машинного обучения можно с помощью curl и Postman.
- Библиотека FastAPI автоматически генерирует документацию для разработанного API модели машинного обучения. Документация

позволяет не только изучить возможности API, но и попробовать его запустить.

- Документация приложения, оформленная с помощью docstring в Python файле, попадает в автоматически сгенерированную документацию API.
- Автоматизировать установку необходимых для работы приложения машинного обучения пакетов Python можно с помощью файла requirements.txt.

Тест

1. Как называется файл со списком пакетов, необходимых для корректной работы приложения?
    - В Python нет возможности установить пакеты на основе списка в файле.
    - **Можно использовать любое название, но обычно применяется requirements.txt.**
    - Допускается использовать только файл с названием requirements.txt.
    - Допускается использовать только файл с названием freeze.pip.
2. Какую команду нужно использовать, чтобы установит все необходимые пакеты Python, указанные в файле requirements.txt?
    - pip freeze > requirements.txt
    - **pip install -r requirements.txt**
    - pip install --all
    - pip upgrade -r requirements.txt

Практическое задание

Цель задания: научиться создавать API для модели машинного обучения.

Задание:

1. Сформируйте команду из 2-3 человек. Можно продолжать работу в команде, которую вы собрали в Модуле 1.
2. С помощью библиотеки FastAPI разработайте API для приложения машинного обучения, которое вы создали в Модуле 1.
3. Создайте репозиторий на GitHub, который содержит код реализации API для вашего приложения машинного обучения.

4. В репозитории на GitHub создайте файл requirements.txt со списком всех библиотек, которые необходимо установить для работы API вашего приложения машинного обучения.

Термины

Все потенциально-незнакомые и новые понятия выдели отдельно и “положи” сюда.

Список источников

- FastAPI Tutorial – https://fastapi.tiangolo.com/tutorial/
- Hugging Face Course – https://huggingface.co/course
- Pydactic – https://pydantic-docs.helpmanual.io/
- Рецензия на фильм Дюна с сайта IMDB – https://www.imdb.com/review/rw7475243/?ref_=tt_urv

Дополнительные материалы

- Web-сервер Uvicorn – https://www.uvicorn.org/
- Формат файла requirements.txt – https://pip.pypa.io/en/latest/reference/requirements-file-format/#requirements-file-format

Модуль № 4

Название: Развертывание приложений искусственного интеллекта в облаке

Образовательные результаты:

- Студенты могут перечислить характеристики типов сервисов, предоставляемых облачными платформами.
- Студенты могут перечислить крупные международные и российские облачные платформы.
- Студенты могут развернуть приложение машинного обучения на облачной платформе Heroku.

В этом модуле:

В предыдущем модуле вы разработали API для предварительно обученной модели машинного обучения. Однако API вы запускали на своем компьютере, поэтому никто другой, кроме вас, не мог использовать эту модель.

В модуле вы:

- Узнаете, что такое облачные вычисления и чем облачные платформы полезны при создании приложений машинного обучения.
- Познакомитесь с облачной платформой для приложений Heroku.
- Научитесь разворачивать приложение, реализующее API для предварительно обученной модели машинного обучения, на платформе Heroku.

Приложение, развернутое на облачной платформе, доступно всем заинтересованным пользователям в Интернет.

Модуль № 4. Юнит № 1. Облачные вычисления

После того, как приложение машинного обучения создано, его необходимо разместить где-то таким образом, чтобы с приложением могли работать пользователи. Для этого нужны вычислительные ресурсы, как правило, серверы, устройства хранения данных и подключение к интернет. Такая процедура называется развертыванием приложения (по английски deploy).

Традиционно для развертывания приложений компании создавали собственные центры обработки данных, устанавливали туда серверы, организовывали сетевую инфраструктуру и подключение к Интернет. При таком сценарии приложения работали на серверах компании в собственном центре обработки данных. Однако создание центров обработки данных и покупка серверного и сетевого оборудования требует значительных затрат. Кроме того, необходимы затраты на поддержание этой инфраструктуры: системы охлаждения, обеспечения бесперебойного питания, техническое обслуживание оборудования, резервное копирование для защиты от потери данных, обеспечение информационной безопасности и решение большого количества других задач.

Альтернативный вариант развертывания приложений – использование облачных вычислений. В этом случае центр обработки данных создает

специальная компания- провайдер услуг облачных вычислений (по-английски Cloud Provider). Провайдер покупает вычислительное и сетевое оборудование, а также обеспечивает его администрирование и бесперебойное функционирование. Используя это оборудование провайдер предоставляет заказчикам сервисы разного типа, наиболее популярными из которых являются:

- Infrastructure-as-a-Service (IaaS, инфраструктура как сервис) – предоставляется доступ к виртуальным машинам, устройствам хранения, сетевому оборудованию. Заказчики сами обеспечивают работоспособность операционных систем и приложений на предоставленной инфраструктуре.
- Platform-as-a-Service (PaaS, платформа как сервис) – предоставляется доступ к платформе для запуска приложений определенного типа, например, Web-приложений. Платформа, как правило, включает операционную систему, систему управления базами данных, программное обеспечение для разработки, тестирования и запуска приложений. Заказчик занимается только разработкой приложения, которое устанавливается на готовой платформе.
- Software-as-a-Service (SaaS, программное обеспечение как сервис) – предоставляется доступ к программному обеспечению, установленному на облачной платформе. Поддержкой программного обеспечения, платформы для его запуска, а также инфраструктуры занимается провайдер облачных услуг. Заказчик просто применяет программное обеспечение в своей работе.

Кроме описанных выше типов облачных сервисов можно встретить большое количество других: Backup-as-a-Service (BaaS) – резервное копирование как сервис, Monitoring-as-a-Service (MaaS) – мониторинг как сервис, Database-as-a-Service (DBaaS) – база данных как сервис, Backend-as-a-Service (BaaS) – бэкенд как сервис, Encryption-as-a-Service (EaaS) – шифрование как сервис, Content-as-a-Service (CaaS) – контент как сервис. Однако чаще всего на практике встречаются IaaS, PaaS и SaaS.

Популярные облачные платформы

В настоящее время существует большое количество провайдеров облачных вычислений, которые предоставляют заказчикам услуги облачных платформ. Среди наиболее популярных платформ можно выделить следующие:

- Amazon Web Services (AWS) (https://aws.amazon.com/) – облачная платформа копании Amazon, одна из первых облачных платформ и один из лидеров рынка облачных вычислений. Предоставляются услуги IaaS, PaaS, SaaS и других моделей.
- Google Cloud (https://cloud.google.com/) – облачная платформа компании Google, также один из лидеров рынка облачных вычислений. Именно в Google Cloud работают виртуальные машины, на которых выполняется код из Google Colab. Предоставляются услуги IaaS, PaaS, SaaS и других моделей.
- Microsoft Azure (https://azure.microsoft.com/) – облачная платформа компании Microsoft, один из лидеров наряду с платформами Amazon и Google. Также предоставляются услуги IaaS, PaaS, SaaS и других моделей.
- Yandex.Cloud (https://cloud.yandex.ru/) – облачная платформа от компании Яндекс. Предоставляет большое количество сервисов по моделям IaaS и PaaS. В отличие от зарубежных облачных платформ Яндекс.Облако соответствуют требованиям Российского законодательства по защите информации и персональных данных: ФЗ-152, постановления правительства № 1119 и Приказа ФСТЭК № 21 и обеспечивает первый уровень защищенности обрабатываемых персональных данных (УЗ-1) (https://cloud.yandex.ru/security). Поэтому на нем развертывают приложения, которые должны удовлетворять таким требованиям.
- VK Cloud Solutions (https://mcs.mail.ru/) – облачная платформа компании VK (ранее mail.ru). Также, как и Яндекс.Облако, является Российской системой и удовлетворяет требованиям Российского законодательства. Предоставляет сервисы по моделям IaaS и PaaS.
- Heroku (https://www.heroku.com) – облачная платформа приложений, предоставляющая услуги PaaS. Первоначально создавалась для приложений на языке Ruby, но сейчас позволяет использовать большое количество языков, в том числе Python. Именно эту облачную платформу мы будем использовать для развертывания API приложения машинного обучения.
- Salesforce (https://www.salesforce.com/) – облачная платформа одноименной компании, предоставляющей одну из самых популярных в мире CRM-систем по модели SaaS.

Кроме перечисленных выше существует большое количество других облачных провайдеров, как зарубежных, так и российских. Спрос на облачные вычисления сейчас стремительно растет, поэтому ожидается, что со временем количество облачных провайдеров и перечень предоставляемых услуг будет расширяться.

Дополнительные преимущества облачных платформ

Облачные платформы берут на себя существенную часть работы по созданию ИТ-инфраструктуры и поддержанию ее работоспособности. Однако на этом их преимущества для разработки и развертывания приложений машинного обучения не заканчиваются.

Одно из важных преимуществ облачных платформ – возможность динамически изменять объем используемых вычислительных ресурсов в зависимости от нагрузки. Например, обучение модели можно выполнять на нескольких серверах для ускорения, а когда обучение закончится освободить эти серверы, чтобы не платить за их использование. Другой пример: вы развернули приложение машинного обучения в облаке, через некоторое время его популярность выросла и количество запросов от пользователей увеличилось. В этом случае вы также можете увеличить вычислительные ресурсы в облаке, которые обслуживают приложение.

Другое преимущество, важное для обучения моделей: возможность динамического использования дорогостоящих ресурсов разных типов, например, серверов с ускорителями вычислений GPU или кластеров обработки больших данных Hadoop. Купить собственные серверы с мощными GPU или организовать кластер Hadoop часто очень дорого и экономически неэффективно для одной компании. Однако провайдер облака может разделить эти затраты между несколькими клиентами таким образом, чтобы ресурсы не простаивали и использовались эффективно.

Недостатки облачных платформ

Облачные платформы позволяют быстро развертывать приложения без существенных затрат на ИТ-инфраструктуру. Однако в случае роста начинают проявляться недостатки облачных платформ. Не все платформы рассчитаны на эффективное масштабирование приложений при высокой нагрузке (high-load). Кроме того, с увеличением нагрузки для работы

приложения требуется все больше вычислительного и сетевого оборудования. После достижения определенного уровня приобретение и эксплуатация собственного оборудования может быть экономически более эффективной, чем аренда облачных сервисов. Поэтому в ближайшее время не стоит ожидать повсеместного отказа от создания собственных центров обработки данных и перехода в облака.

Итоги

- Облачные платформы предоставляют быстрый и удобный способ разворачивать приложения машинного обучения без больших вложений в ИТ-инфраструктуру.
- Облачные платформы предоставляют сервисы разных типом, самыми популярными из которых являются инфраструктура как сервис (IaaS), платформа как сервис (PaaS) и программное обеспечение как сервис (SaaS).
- Наиболее крупные и популярные международные облачные платформы: AWS, Google Cloud и Microsoft Azure, российские облачные платформы: Яндекс.Облако и VK Cloud Solutions.

Тест

1. Какой сервис предоставляет облачная платформа типа PaaS:
    - Виртуальные машины и устройства хранения
    - **Платформу для запуска приложений**
    - Прикладное программное обеспечение
    - Управление контентом
2. Какая из перечисленных облачных платформ соответствует требованиям российского законодательства по защите информации и персональных данных:
    - Amazon Web Services
    - Google Cloud
    - **Yandex.Cloud**
    - Heroku

Модуль № 4. Юнит № 2. Облачная платформа Heroku

В этом модуле мы будем использовать облачную платформу Heroku для того, чтобы развернуть ранее созданный вами API для модели машинного обучения. Heroku выбрана по двум причинам:

- Heroku предоставляет услуги типа платформы как сервиса (PaaS), то есть не нужно будет тратить время на установку операционной системы, настройки необходимого программного обеспечения для запуска приложения и сетевых сервисов. Все это будет выполнено автоматически средствами Heroku.
- Бесплатная учетная запись Heroku позволяет запускать приложение в течение 550 часов в месяц, что достаточно для выполнения всех заданий курса (и даже для проведения дополнительных экспериментов).

Как устроена платформа Heroku

На платформе Heroku можно развернуть приложение, написанное на Ruby, Java, Python и некоторых других языках программирования. С точки зрения платформы Heroku приложение состоит из двух частей:

- Исходного кода приложения.
- Перечня зависимостей – библиотек и других компонентов, которые необходимы для корректной работы приложения. В Python зависимости указываются в файле requirements.txt.

Основной метод передачи исходного кода приложения на платформу Heroku – это git. Вы можете разместить код своего приложения в репозитории на GitHub или использовать локальный репозиторий git на вашем компьютере.

Платформе Heroku нужно знать, как именно запускать ваше приложение. Для этого в репозиторий git нужно включить файл с названием Procfile, который содержит команду для запуска приложения.

Когда платформа Heroku получает исходный код приложения, она запускает сборку приложения. Для Python сборка включает установку библиотек из файла requirements.txt.

После сборки приложения платформа Heroku формирует Slug – подготовленный архив с кодом приложения, библиотеками для его работы, средствами для запуска кода на нужном языке программирования, а также файлом Procfile с командой запуска приложения.

Для работы приложения Heroku использует Linux контейнеры, которые называются Dynos. При запуске приложения создается один контейнер Dyno, в который записывается подготовленный на предыдущем этапе Slug приложения. Затем производится запуск приложения с помощью команды, указанной в Procfile из Slug.

Контейнеры Dynos в Heroku бывают разных типов в зависимости от задач, которые в них выполняются. Нас в первую очередь будет интересовать тип контейнеров web, который предназначен для запуска web-приложений, к которым относится и API для приложения машинного обучения, который мы создаем.

Для работы с Web-приложениями платформа Heroku использует HTTP-маршрутизатор, который принимает HTTP запросы, направленные приложению, и пересылает его на контейнер, реализующий приложение. Платформа Heroku позволяет создавать несколько контейнеров, обслуживающих одно и то же приложение. В этом случае HTTP-маршрутизатор Heroku будет балансировать нагрузку между несколькими контейнерами. В результате появится возможность обслуживать больше пользователей в единицу времени.

Бесплатный аккаунт на Heroku

Heroku предоставляет возможность начать работать с платформой и познакомиться с ее возможностями без необходимости платить деньги. При регистрации вы получаете бесплатный аккаунт, в которых входит 550 часов в месяц работы контейнеров Dyno на платформе Heroku. Размер памяти бесплатного контейнера – 512 МБ.

Особенность работы бесплатных контейнеров на Heroku заключаются в том, что они останавливаются после 30 минут отсутствия активности. Такой подход позволяет сэкономить бесплатные часы работы контейнеров. В случае обращения к приложению контейнер запускается вновь.

Ограничения бесплатного аккаунта Heroku вполне подходят для запуска API приложения с предварительно обученной моделью машинного обучения. Нужно обратить внимание на размер модели: модель должна поместится ограниченную в память контейнера 512 МБ. При этом память нужна не

только для размещения модели, но и для системных процессов и Web-сервера uvicorn. Остановка контейнера после 30 минут отсутствия активности приемлемо для приложения, разрабатываемого с целью обучения или как часть портфолио.

Итоги

- Heroku – облачная платформа для разработки и запуска приложений.
- При использовании Heroku нет необходимости устанавливать и настраивать операционную систему и требуемое для работы приложения программное обеспечение, Heroku делает это автоматически.
- Код приложения передается на платформу Heroku из репозитория git.
- Для запуска приложения на Heroku в репозиторий git нужно добавить файлы Procfile с командой для запуска приложения и requirements.txt со списком пакетов, необходимых для работы приложения.

Модуль № 3. Юнит № 3. Развертывание приложений искусственного интеллекта на платформе Heroku

В этом юните мы рассмотрим, как развернуть API для приложения машинного обучения из репозиторий GitHub на облачной платформе Heroku.

Для развертывания приложения не нужно будет ничего платить, возможностей бесплатного аккаунта будет достаточно.

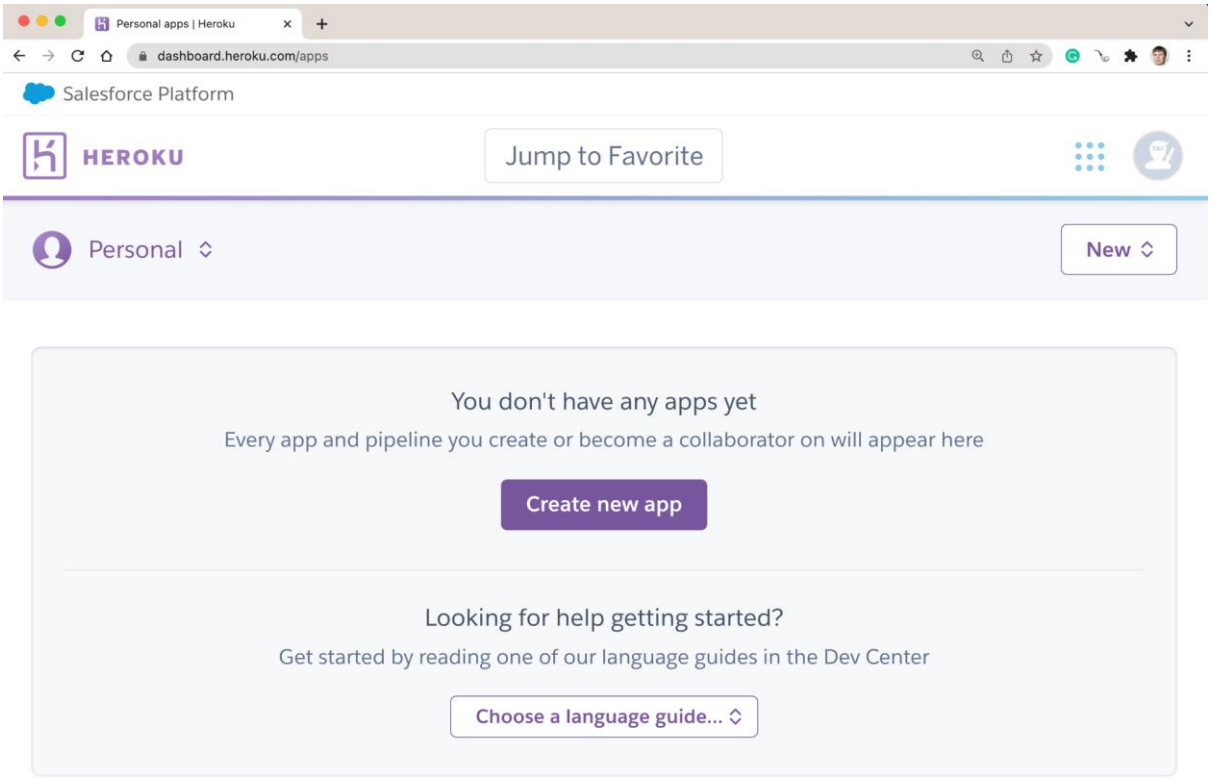
Регистрация на платформе Heroku

Перед началом работы с платформой Heroku вам необходимо зарегистрироваться по ссылке – https://signup.heroku.com/. В процессе регистрации в качестве своей роли выбирайте “Student”, а в качестве основного языка программирования (Primary development language) – Python.

Пожалуйста, обратите внимание, что вводить данные банковской карты для бесплатного аккаунта не требуется!

Создание приложения на платформе Heroku

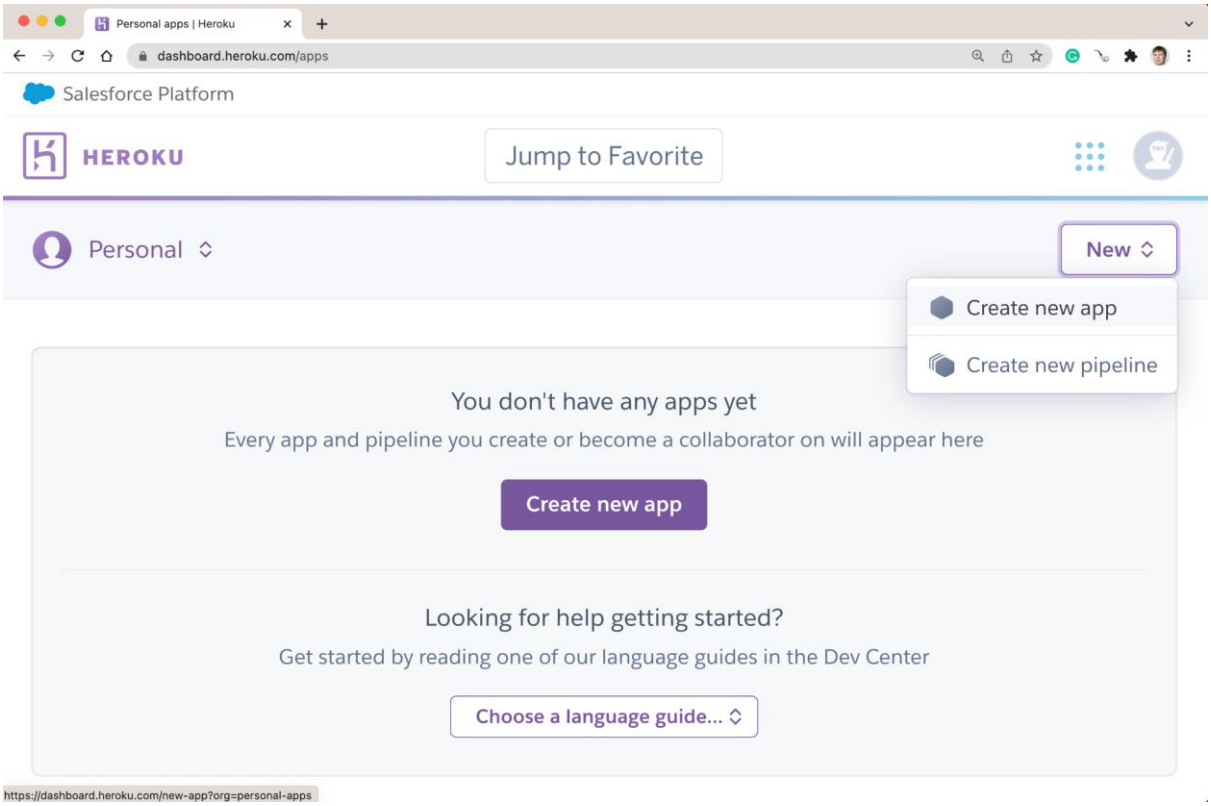
Первый шаг по разворачиванию вашего API для приложения машинного обучения – это создание приложения Heroku. Для этого нужно зайти на платформу Heroku используя логин и пароль, которые вы получили при регистрации. После входа на платформу вы попадаете на панель управления (Dashboard).



Панель управления Heroku

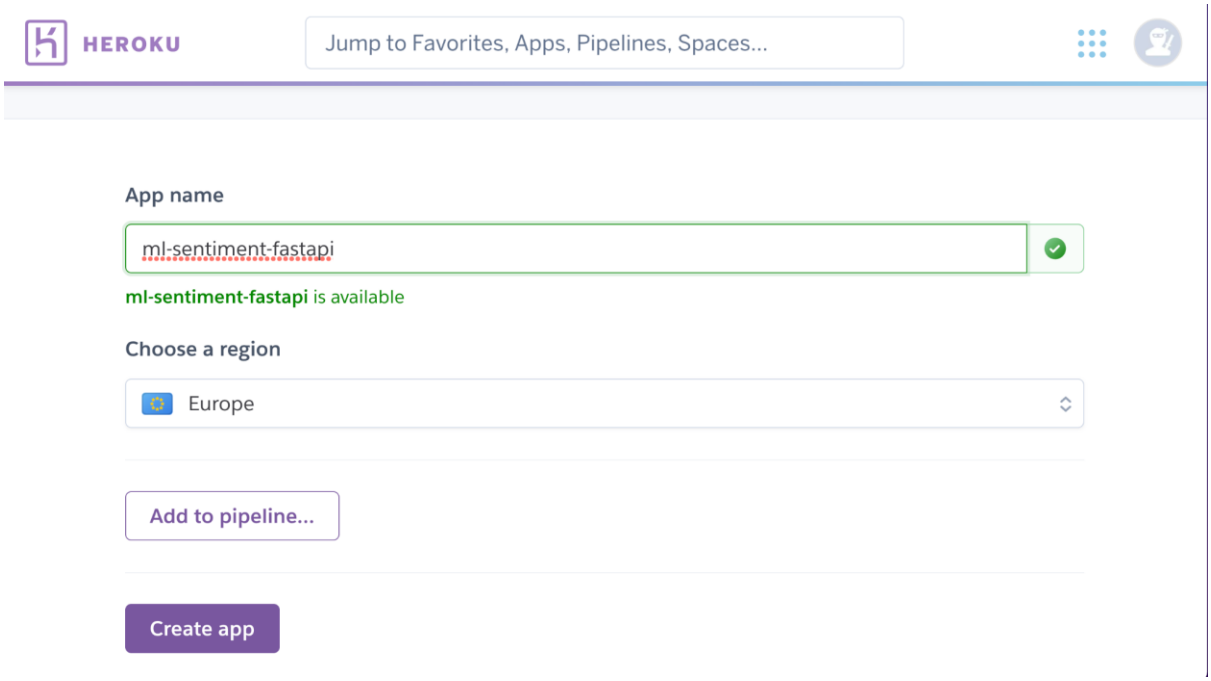
Пока у вас нет ни одного приложения. Чтобы создать приложение, нужно нажать на кнопку “Create new app” в середине экрана.

Альтернативный вариант создания приложения – нажать на кнопку “New” в правой верхней части экрана и в открывшемся меню выбрать “Create new app”. Этот способ будет работать, когда у вас появятся приложения и их список будет выводится в средней части панели управления вместо кнопки “Create new app”.



Создание нового приложения на платформе Heroku

Для создания приложения необходимо указать его имя (App name) и регион размещения сервера.



Окно создания приложения на Heroku

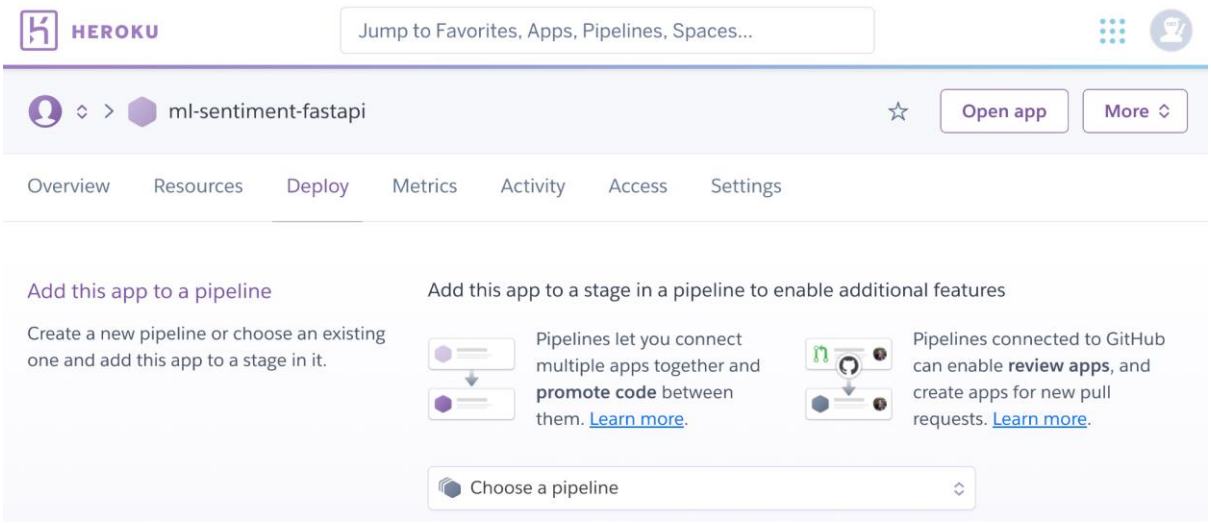
Имя приложения должно быть уникальным на всей платформе Heroku (включая приложения других пользователей). Имя приложения используется для формирования ссылки для обращения к нему. Например,

для приложения с названием ml-sentiment-fastapi ссылка будет выглядеть следующим образом: https://ml-sentiment-fastapi.herokuapp.com. Если имя приложения не уникально, то под полем для ввода имени появится сообщение об ошибке.

Heroku позволяет размещать приложения на серверах в двух регионах: США и Европе. Рекомендуется выбирать регион Europe, т.к. в этом случае серверы расположены ближе к России и сетевое взаимодействие с ними будет проходить быстрее.

После выбора имени приложения и региона нажимайте кнопку “Create app” для создания приложения.

После успешного создания приложения вы попадаете на страницу развертывания приложения (“Deploy”). Развертывание приложения можно организовать из репозитория на GitHub. Однако перед этим нам необходимо подготовить приложение в репозитории для развертывания на платформе Heroku.



Экран развертывания приложения на платформу Heroku

Подготовка приложения в репозитории на GitHub для развертывания на платформе Heroku

Подготовка приложения машинного обучения к развертыванию на платформе Heroku состоит из двух шагов.

1. Платформе нужно знать, какую команду использовать для запуска вашего приложения машинного обучения. Такая команда указывается в файле с названием Procfile. Необходимо создать файл Procfile в репозитории на GitHub и вставить в него следующую строку:

```
web: uvicorn main:app --host=0.0.0.0 --port=${PORT:-5000}
```

   web в начале строки означает тип контейнера на платформе Heroku. Затем идет команда запуска Web-сервера uvicorn для работы API, имя файла с приложением и название объекта FastAPI в этом файле. После этого указываются сетевые параметры запуска сервера uvicorn на платформе Heroku, их нужно оставить без изменений.

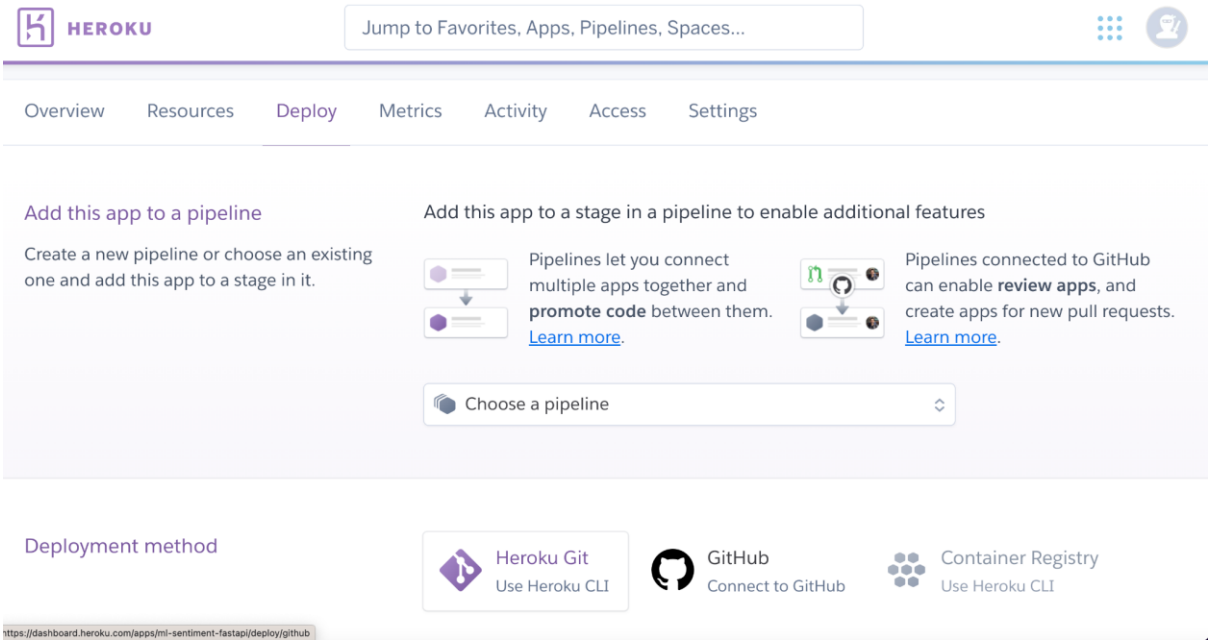
2. Запуск приложения машинного обучения будет выполняться не на вашем компьютере, а в контейнере в Heroku. Чтобы приложение работало успешно, перед его запуском необходимо установить все нужные для его работы библиотеки. Список таких библиотек, как вы знаете, указывается в файле requirements.txt. Если вы не создали такой файл раньше, то нужно это сделать. Пример файла requirements.txt для API модели машинного обучения из библиотеки Hugging Face:

```
-f https://download.pytorch.org/whl/torch_stable.html
fastapi
uvicorn
transformers
torch==1.9.0+cpu
```

Теперь репозиторий GitHub готов для развертывания приложения на Heroku.

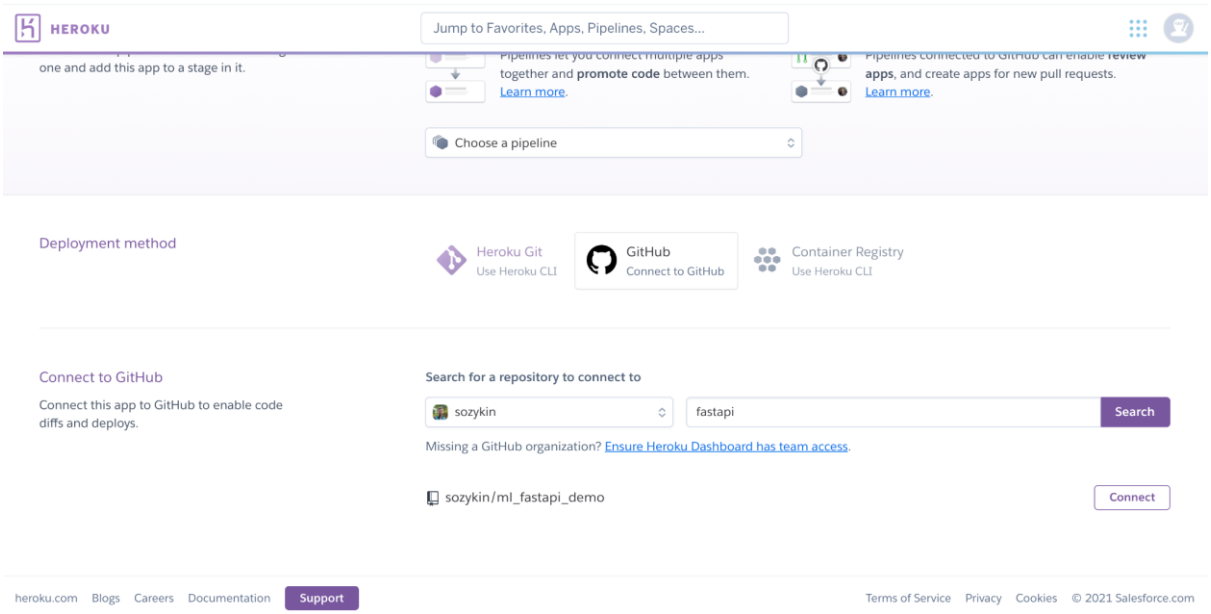
Интеграция приложения Heroku с репозиторием на GitHub

Платформа Heroku поддерживает несколько методов развертывания (Deployment method), среди которых основным является развертывание из репозитория на GitHub. Выбрать метод развертывания можно на закладке “Deploy” приложения. В разделе Deployment method выберите GitHub.



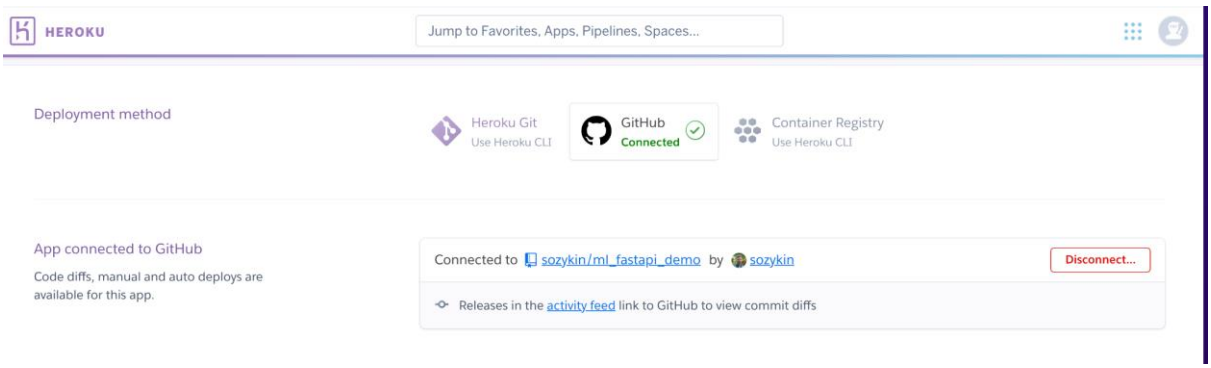
Выбор метода развертывания для приложения на платформе Heroku

После выбора метода развертывания “GitHub” нажмите кнопку “Connect to GitHub”. В появившемся окне введите ваш логин и пароль на GitHub. Heroku подключится к вашей учетной записи на GitHub и после этого можно будет выполнить поиск репозиториев у выбранного пользователя.



Подключение репозитория на GitHub к Heroku

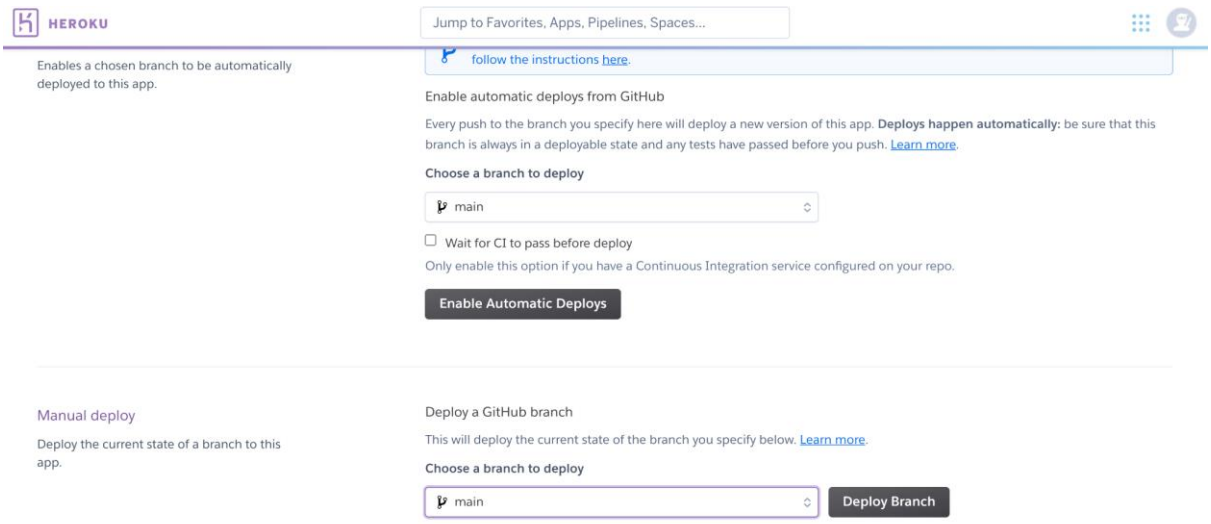
В списке репозиториев выберите нужный и нажмите кнопку “Connect”. Heroku подключится к выбранному репозиторию.



Репозиторий GitHub подключен в приложению Heroku

Развертывание приложения из репозитория GitHub на платформе Heroku

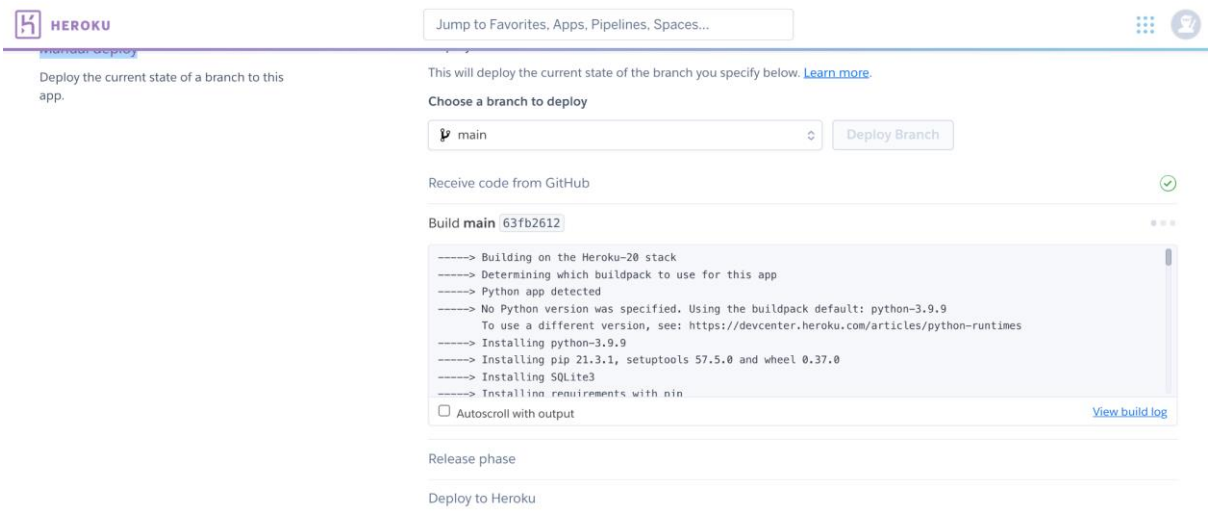
Репозиторий GitHub подключен к приложению Heroku, можно переходить к развертыванию приложения. Для этого прокрутите страницу ”Deploy” приложения до самого конца, где находится раздел “Manual deploy” (ручное развертывание). Автоматическое развертывание мы научимся настраивать в следующем модуле.



Настройка ручного развертывания приложения из репозитория GitHub на Heroku

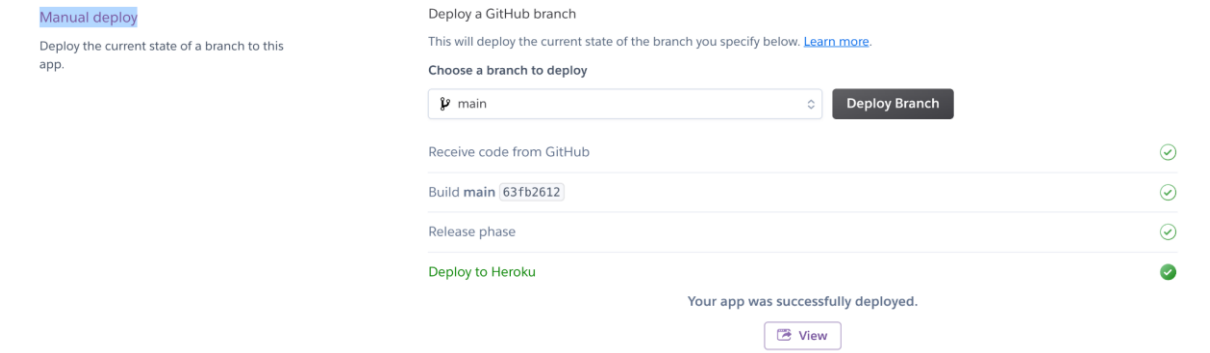
Выберите ветку (branch), из которой будет выполняться развертывание: main. Более подробно ветки в git мы рассмотрим в следующем семестре, а пока у нас только одна ветка main.

После выбора ветки нажмите кнопку “Deploy branch”. Откроется окно, в котором будут показываться журналы сборки приложения.



Окно журнала сборки приложения на Heroku

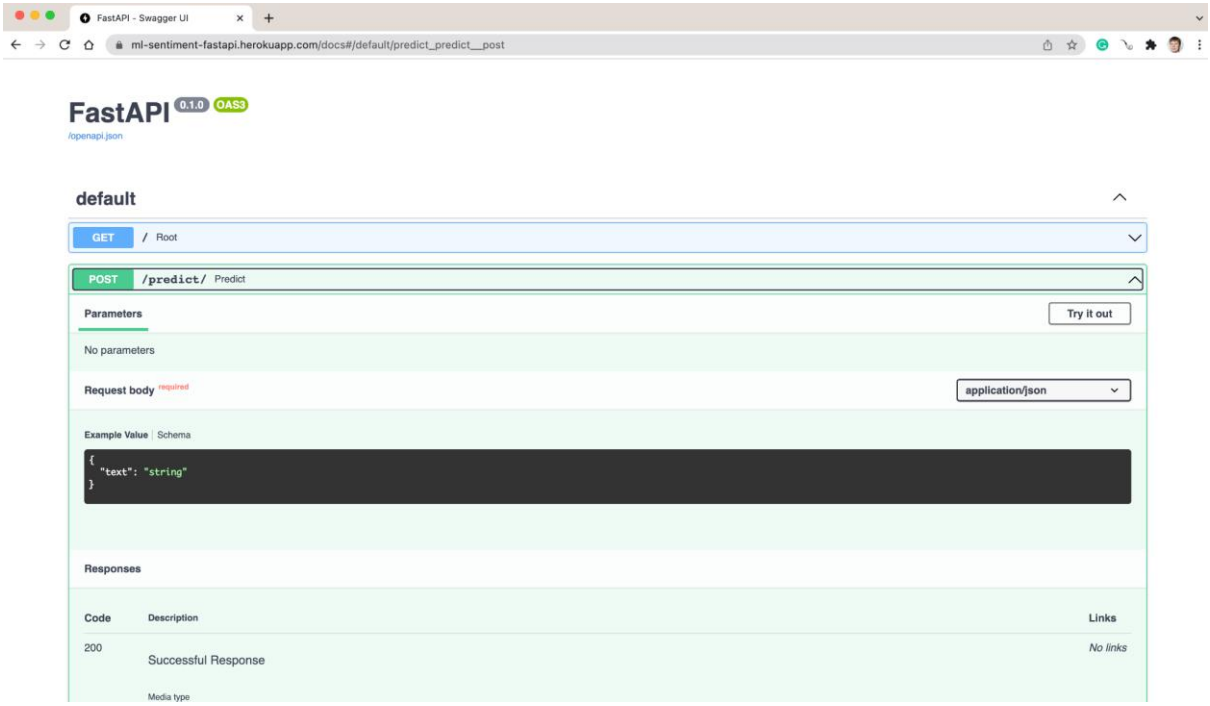
После успешного завершения сборки приложения оно будет развернуто на платформе Heroku.



Окно результатов развертывания приложения на платформе Heroku

Чтобы перейти к приложению можно нажать на кнопку “View”, в результате чего откроется корневая страница вашего приложения.

В качестве примера давайте откроем страницу с документацией на API, которая была автоматически сгенерирована библиотекой FastAPI. Наше приложение называется ml-sentiment-fastapi, документация доступна по адресу https://ml-sentiment-fastapi.herokuapp.com/docs



Документация к API модели машинного обучения на платформе Heroku

В прошлом модуле мы разворачивали API на локальной машине и доступ к нему был возможен только с локальной машины. Версия API для модели машинного обучения, развернутая на платформе Heroku, доступна из интернет всем заинтересованным пользователям. Например, чтобы обратиться к API с помощью curl, можно использовать такую команду:

```
curl -X 'POST' \
  'https://ml-sentiment-fastapi.herokuapp.com/predict/' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "text": "I like cloud computing!"
}'
```

Основное отличие заключается в том, что вместо ссылки на локальном компьютере (http://127.0.0.1:8000/predict) мы используем ссылку на платформе heroku – https://ml-sentiment-fastapi.herokuapp.com/predict

Попробуйте сами обратиться к API с помощью Postman. Также вы можете узнать URL, по которой развернуты API других студентов вашей группы и попробовать обращаться к ним с помощью curl или Postman.

Итоги

- Приложения машинного обучения удобно разворачивать в облаке.
- Облака различаются по типу сервисов, который предлагается: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), Software-as-a-Service (SaaS).
- Heroku – простая в использовании PaaS платформа. Она предоставляет платформу для развертывания приложений на различных языках программирования.
- Развертывание приложений на Heroku можно выполнять из репозитория на GitHub.
- Чтобы развернуть приложение на Heroku из GitHub, нужно добавить два файла:
    - Procfile с командой для запуска приложения
    - requirements.txt со списком пакетов Python, которые нужно установить для успешной работы приложения.
- Развернутые на Heroku приложения доступны всем заинтересованным пользователям в интернет.
- Heroku предоставляет бесплатный тариф, в который входит 550 часов работы контейнера в месяц. Обычно этого достаточно, чтобы изучить процесс развертывания приложений на облачной платформе.

Тест

1. Как называется файл с названием команды, которую нужно запустить для запуска приложения на Heroku?
    - **Procfile**
    - requirements.txt.
    - heroku.exe
    - readme.md
2. Имя приложения на Heroku – auto-translation-fastapi. Как будет выглядеть ссылка на доступ к приложению на Heroku?
    - https://127.0.0.1/predict
    - https://127.0.0.1/auto-translation-fastapi
    - **https://auto-translation-fastapi.herokuapp.com**
    - https://auto-translation-fastapi.com

Практическое задание

Цель задания: научиться разворачивать API для модели машинного обучения на облачной платформе Heroku.

Задание выполняется в команде из 2-3 человек, которую вы сформировали для выполнения предыдущего задания по разработке API для модели машинного обучения.

Задание:

1. Создайте приложение на платформе Heroku.
2. Модифицируйте репозиторий GitHub с API для модели машинного обучения, который вы создали в предыдущем домашнем задании:
    - Добавьте файл requirements.txt со списком всех пакетов Python, необходимых для работы вашего приложения (если вы этого не сделали раньше). Heroku автоматически установит все пакеты из этого файла перед запуском приложения.
    - Добавьте файл Procfile с информацией о том, как Heroku будет запускать ваше приложение.
3. Настройте интеграцию приложения на Heroku с репозиторием на GitHub.
4. Разверните приложение с API для модели машинного обучения из репозитория GitHub на платформе Heroku.
5. В качестве отчета о выполнении домашнего задания пришлите две ссылки:
    - Ссылка на репозиторий GitHub с кодом приложения (например, https://github.com/sozykin/ml_fastapi_demo).
    - Ссылка на работающее приложение на платформе heroku (например, https://ml-fastapi-demo.herokuapp.com/)

Термины

Все потенциально-незнакомые и новые понятия выдели отдельно и “положи” сюда.

Список источников

- Безопасность Яндекс.Облака – https://cloud.yandex.ru/security
- How Heroku Works – https://devcenter.heroku.com/articles/how-heroku-works

Дополнительные материалы

- Пример приложения машинного обучения, подготовленного для развертывания на платформе Heroku – https://github.com/sozykin/ml_fastapi_demo
- Kasun Indrasiri, Sriskandarajah Suhothayan. Design Patterns for Cloud Native Applications. – https://www.oreilly.com/library/view/design-patterns-for/9781492090700/
- Shivakumar R Goniwada. Cloud Native Architecture and Design: A Handbook for Modern Day Architecture and Design with Enterprise-Grade Examples – https://link.springer.com/book/10.1007/978-1-4842-7226-8

Модуль № 5
Название: Тестирование программного обеспечения

Образовательные результаты:

- Студенты понимают необходимость тестирования программного обеспечения.
- Студенты могут разрабатывать тесты для программ на Python с помощью библиотеки PyTest.
- Стыдент может перечислить особенности и преимущества практики программной инженерии Continuous Integration.
- Студенты могут использовать инструменты Continuous Integration на платформе GitHub.

В этом модуле:

Вы научились создавать приложения на основе готовых моделей машинного обучения и развертывать их в облачной платформе. В этом модуле вы узнаете, как обеспечить стабильность работы приложений с использованием тестирования.

В модуле вы:

- Узнаете, как разрабатывать и запускать тесты на Python с помощью библиотеки PyTest.

- Научитесь создавать тесты для API с помощью инструментов библитеки FastAPI.
- Познакомитесь с понятием Continuous Integration.
- Научитесь настраивать Continuous Integration для репозитория на GitHub с помощью GitHub Actions.
- Научитесь автоматически разворачивать приложение на облачной платформе Heroku в случае успешного выполнения тестов на GitHub.

Модуль № 5. Юнит № 1. Тестирование приложений на Python помощью PyTest

Тестирование программного обеспечения – это один из самых важных инструментов программной инженерии. В процессе тестирования выполняется поиск ошибок в коде и проверка соотвествия результатов выполнения программы ожидаемым результатам. Обеспечить стабильную работу приложений без тестирования практически невозможно.

Сейчас самой популярной библиотекой для разработки тестов для программ на Python является библиотека PyTest (https://docs.pytest.org/). Она позволяет быстро и просто разрабатывать тесты, а также автоматически находит код с тестами во время тестирования.

Перед использованием библиотеки PyTest ее нужно установить с помощью pip:

```
pip install pytest
```

Разработка тестов с использованием PyTest

Тесты в PyTest создаются в виде отдельных функций, в название которых включено слово test. Давайте рассмотрим простой пример создания теста для функции, которая выполняет сложение двух чисел.

```
def add(x, y):
    return x + y
```

```
def test_add():
    assert add(3, 5) == 8
```

Здесь мы определяем две функции:

1. Функция add, которая получает на вход два аргумента x и y, а на выход выдает их сумму.
2. Функция test_add, которая содержит код теста для функции add. В этой функции мы используем оператор assert Python, который выполняет проверку условия. Условие сконструировано следующим образом. Вызываем функцию add, работоспособность которой нам нужно проверить, передаем в нее два параметра: 3 и 5. Результат работы функции сравниваем с значение 8.

Оператор assert в Python работает достаточно просто. Он выполняет проверку условного выражения (в примере условие add(3, 5) == 8) и если выражение истинно, то не происходит ничего, а если выражение ложно – то программа останавливается с ошибкой.

Запуск тестов в PyTest

Для запуска тестов нужно выполнить команду pytest в каталоге с файлами проекта:

```
% pytest
============================= test session starts
==============================
platform darwin -- Python 3.9.1, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /Users/andrey/projects/tests
plugins: anyio-3.4.0
collected 1 item

test_sample.py .                                         [100%]

============================== 1 passed in 0.02s
==============================
```

PyTest автоматически ищет код с тестами следующим образом:

- Тестами считаются функции, название которых начинается на test (в нашем примере это функция test_add)
- Файлы с названиями test_*.py или *_test.py считаются полностью содержащими тесты.

Таким образом, при запуске PyTest нет необходимости указывать, какие тесты нужно запустить. Будут автоматически запущены все тесты, которые созданы с учетом указанных выше правил.

В примере был запущен один тест, который выполняется успешно. Давайте посмотрим, что произойдет, если в процессе теста возникнет ошибка. Для этого изменим код функции add, чтобы в ней была ошибка:

```
def add(x, y):
    return x - y

def test_add():
    assert add(3, 5) == 8
```

Запускаем тесты:

```
% pytest
============================= test session starts
==============================
platform darwin -- Python 3.9.1, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /Users/andrey/projects/tests
plugins: anyio-3.4.0
collected 1 item

test_sample.py F                                                 [100%]

=================================== FAILURES
===================================
_____________________________________ test_add
_____________________________________

    def test_add():
```

```
>       assert add(3, 5) == 8
E       assert -2 == 8
E        +  where -2 = add(3, 5)

test_sample.py:6: AssertionError
========================== short test summary info
==========================
FAILED test_sample.py::test_add - assert -2 == 8
============================== 1 failed in 0.06s
==============================
```

В этот раз мы видим, что тесты завершены не успешно (failed). Указано, какой именно тест не выполняется (в функции test_add) и какое значение условия (assert -2 == 8). Проблема с тестом вызвана ошибкой в функции add: мы ожидаем, что функция add вернет значение 8, а она вернула значение -2. Таким образом, тест позволил нам определить, что функция add работает неправильно.

Итоги:

- Тестирование программного обеспечения позволяет обнаружить в нем ошибки и удостоверится в правильности результатов работы программы.
- В Python для тестирования используется библиотека PyTest.
- Тесты в PyTest создаются в функциях, название которых начинается на test, или в файлах, название которых начинается или заканчивается на test.
- В тестах используется оператор assert Python, который выполняет проверку условия и в случае ложного результата завершает программу с ошибкой.
- Запуск тестов выполняется с помощью команды pytest.

Модуль № 5. Юнит № 2. Тестирование API

Давайте рассмотрим, как можно применить библиотеку PyTest для тестирования приложения API предварительно обученной модели машинного обучения, которое вы создали ранее. Сложность тестирования API заключается в том, что для работы такого приложения необходимо

запускать Web-сервер, а взаимодействие с ним производится по сетевым протоколам с помощью инструментов curl, Postman или аналогичных. Однако в библиотеу FastAPI встроен инструмент для тестирования API с использованием PyTest, который называется TestClient. Давайте рассмотрим, как его использовать для тестирования API.

В крупных проектах код тестов рекомендуется отделять от кода приложения для удобства понимания и сопровождения. Поэтому мы не будем добавлять код теста в файл main.py, который содержит код приложения API, а создадим для этого отдельный файл test_main.py. Название файла начинается с test, поэтому библиотека PyTest поймет, что в этом файле расположены тесты.

В файл test_main.py добавим следующее:

```
from fastapi.testclient import TestClient
from main import app

client = TestClient(app)


def test_read_main():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello World"}
```

В первой строке подключается TestClient – клиент для тестирования API из FastAPI.

```
from fastapi.testclient import TestClient
```

Во второй строке мы импортируем объект app класса FastAPI из файл main.py в текущем каталоге. Именно в файле main.py содержится код приложения API для модели машинного обучения.

```
from main import app
```

После этого создается клиент для тестирования, которому при создании передается объект API app, тестирование которого необходимо выполнить.

```
client = TestClient(app)
```

Теперь все готово для создания тестов. Первый тест проверяет доступность приложения при обращении к корню сервера:

```
def test_read_main():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello World"}
```

В тесте сначала клиент запускает запрос HTTP GET к корню сервера (путь "/") и записывает результат в переменную response (ответ):

```
    response = client.get("/")
```

Затем выполняются два теста с помощью оператора assert. Первый проверяет код ответа HTTP:

```
    assert response.status_code == 200
```

Код ответа 200 означает, что запрос выполнен успешно. Другие коды свидетельствуют о том, что произошла ошибка при обработке запроса. Этого мы хотим избежать.

Второй тест проверяет содержание ответа:

```
    assert response.json() == {"message": "Hello World"}
```

Содержание ответа в формате JSON должно быть равно {"message": "Hello World"}. Именно такое значение в нашем примере возвращает функция, которая обрабатывает запросы к корню Web-сервера:

```
@app.get("/")
```

```
def root():
    return {"message": "Hello World"}
```

Давайте создадим более сложный пример теста, который вызывает метод predict нашего API, передает в него данные для распознавания и сравнивает полученный ответ с ожидаемым. Мы будем продолжать рассматривать пример приложения, которое определяет тональность текстов на английском языке.

Первой мы создадим функция, которая пытается распознать тональность положительной фразы:

```
def test_read_predict_positive():
    response = client.post("/predict/",
        json={"text": "I like machine learning!"}
    )
    json_data = response.json()

    assert response.status_code == 200
    assert json_data['label'] == 'POSITIVE'
```

В этом тесте клиент передает запрос POST, путь на сервере "/predict/", в теле сообщения в формате JSON передается сообщение для определения тональности:

```
    response = client.post("/predict/",
        json={"text": "I like machine learning!"}
    )
```

Тело ответа из формата JSON в виде словаря Python (dict) сохраняется в переменную json_data для последующего анализа:

```
    json_data = response.json()
```

Затем следуют два теста. Первый тест проверяет доступность приложения: код статуса ответа HTTP должен быть равен 200:

```
    assert response.status_code == 200
```

Второй тест проверяет результат распознавания. Тональность фразы положительная, поэтому результат определения тональности (ключ 'label' в словаре json_data) должен быть 'POSITIVE':

```
    assert json_data['label'] == 'POSITIVE'
```

Теперь давайте создадим функцию, которая проверяет тональность отрицательной фразы:

```
def test_read_predict_negative():
    response = client.post("/predict/",
        json={"text": "I hate machine learning!"}
    )
    json_data = response.json()

    assert response.status_code == 200
    assert json_data['label'] == 'NEGATIVE'
```

Работа этой функции аналогична предыдущей, только для распознавания передается фраза с отрицательной эмоциональной окраской и тест проверяет, что результат определения тональность равен 'NEGATIVE'.

Полностью файл с тестами доступен в репозитории с примером приложения с тестами – https://github.com/sozykin/ml_fastapi_tests

Запуск тестов производится с помощью команды pytest в каталоге с приложением:

```
% pytest
============================= test session starts
==============================
platform darwin -- Python 3.9.1, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /Users/andrey/projects/ml_fastapi_tests
plugins: anyio-3.4.0
```

```
collected 3 items

test_main.py ...                                        [100%]

============================== warnings summary ===============================
../../../../usr/local/lib/python3.9/site-packages/flatbuffers/compat.py:19
  /usr/local/lib/python3.9/site-packages/flatbuffers/compat.py:19:
DeprecationWarning: the imp module is deprecated in favour of importlib; see the module's documentation for alternative uses
    import imp

-- Docs: https://docs.pytest.org/en/stable/warnings.html
======================== 3 passed, 1 warning in 18.18s =========================
```

В этот раз тесты будут работать значительно дольше, т.к. для их запуска нужно запустить Web-сервер с API и отправлять к нему запросы по сети. Однако клиент тестирования TestClient из FastAPI делает все это автоматически.

В результате выполнения теста видно, что все три теста прошли успешно. Есть одно предупреждение, но в данном примере его можно игнорировать.

Итоги:

- Для тестирования API используется инструмент TestClient из FastAPI.
- Код тестов рекомендуется размещать в отдельном файле от основного кода приложения. Название файла с тестами должно начинаться или заканчиваться на test.
- Файлы с кодом приложения и тестами должны находится в одном каталоге.
- Запуск тестов выполняется командой pytest из каталога приложения (в котором находятся файлы с кодом приложения и тестов).

Модуль № 5. Юнит № 3. Continuous Integration и GitHub Actions

Мы научились создавать тесты для программ на Python, в том числе для тестирования API. Однако, чтобы находить ошибки в программах, тесты

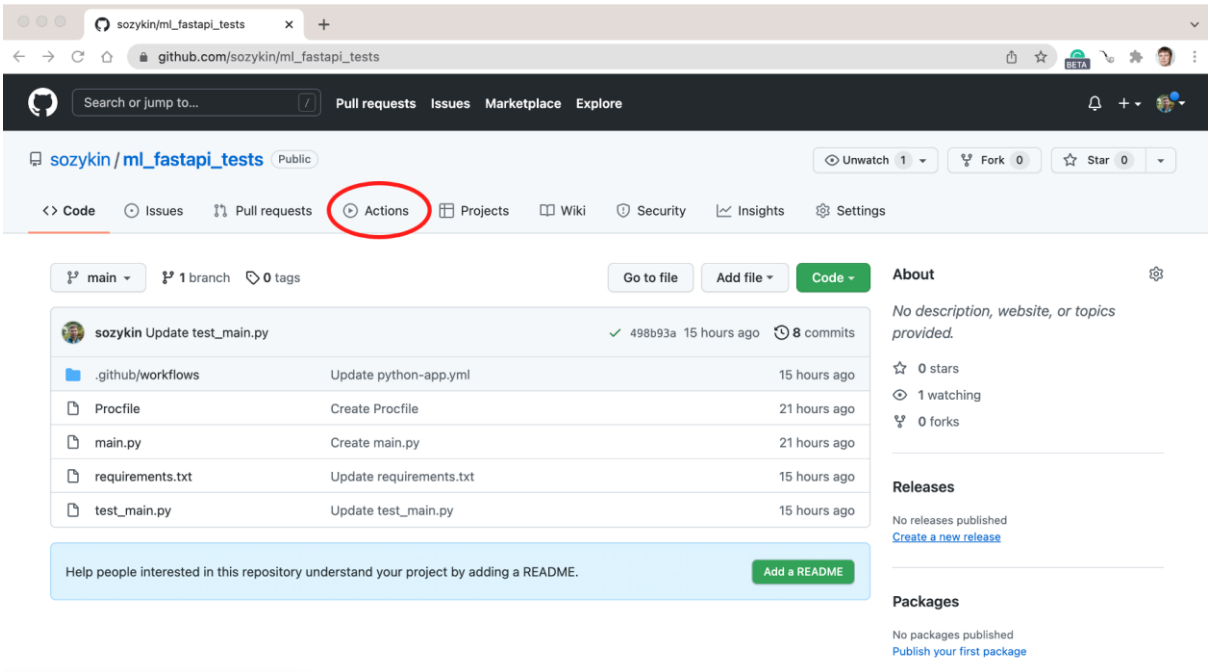
должны запускаться регулярно, иначе они будут бесполезны. Кто и когда должен запускать тесты, чтобы ошибки находились как можно раньше?

Для решения этой задачи в программной инженерии используется практика, которая называется Continuous Integration (CI, по-русски непрерывная интеграция, но этот термин почти никто не использует). Техника CI заключается в том, что разработчики должны достаточно часто обновлять код в разделяемом репозитории (например, на GitHub), и при каждом обновлении автоматически запускаются тесты для проверки корректности внесенных изменений. Например, тесты могут запускаться при выполнении каждого коммита в репозиторий.

Сейчас существует большое количество инструментов CI, например, Jenkins, TeamCity, CircleCI. Крупные облачные платформы предоставляют свои сервисы CI: AWS CodePipeline, Azure Pipelines, CI/CD on Google Cloud. Мы рассмотрим, как использовать инструменты CI GitHub Actions, т.к. они бесплатны и для их использования в репозитории на GitHub не нужно ничего дополнительно устанавливать и настраивать.

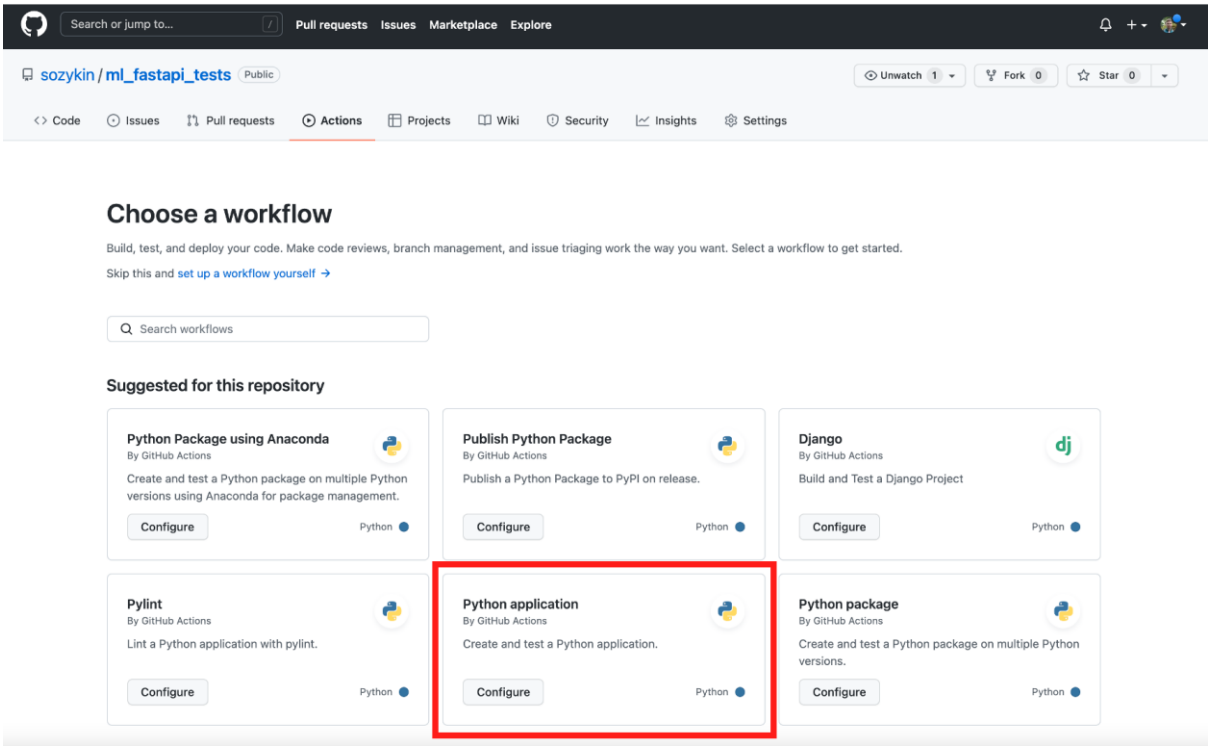
Настройка GitHub Actions для репозитория

Для того, чтобы настроить Continuous Integration для репозитория GitHub с помощью GitHub Actions, нужно перейти на вкладку Actions в репозитории:



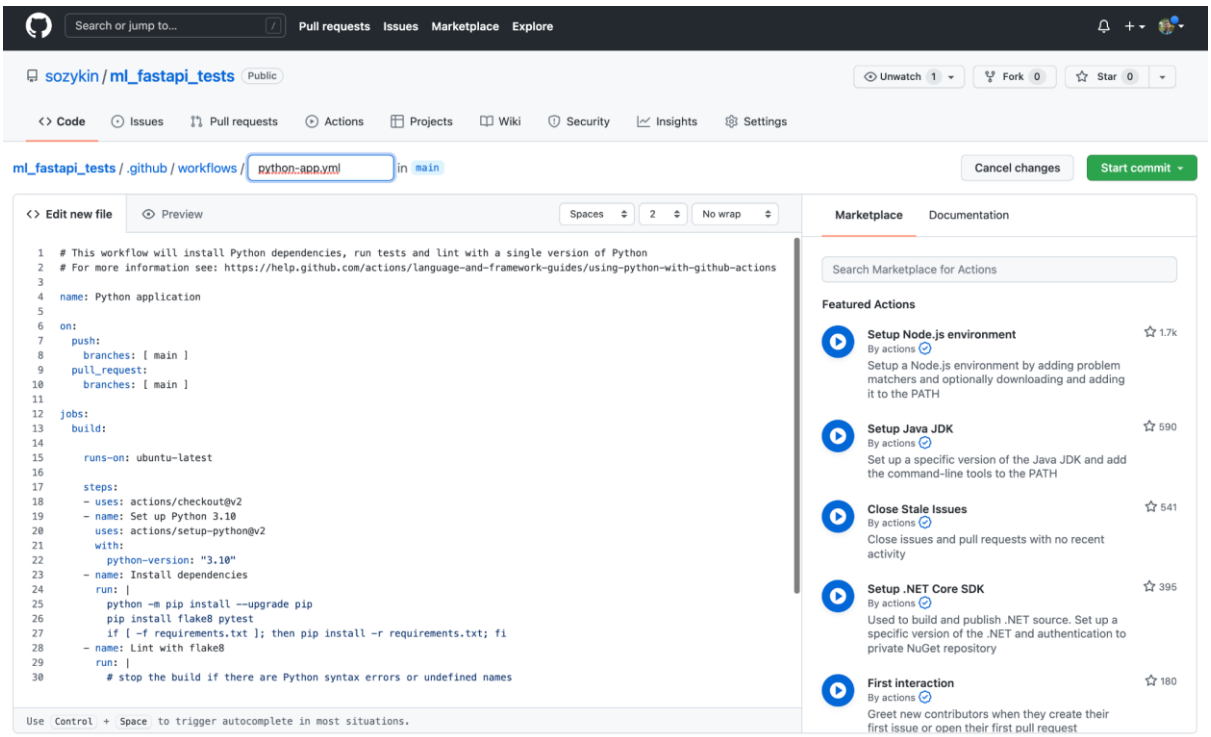
Переход к настройкам GitHub Actions для репозитория на GitHub

В открывшейся закладке GitHub Actions предложить выбрать тип потока работ (workflow) для реализации CI. GitHub определил, что приложение написано на Python и предлагает несколько вариантов потоков работ для Python. Выбираем вариант “Python application” – приложение на Python без каких-либо дополнительных инструментов.



Выбор типа потока работ CI для приложения на Python

После выбора типа потока работ откроется окно с шаблоном файла описания потока работ:



Шаблон файла описания потока работ

Файл описания потока работ рекомендуется оставить без изменений, только поменять версию Python с 3.10 на 3.9. Измененный файл будет выглядеть следующим образом:

```
# This workflow will install Python dependencies, run tests and lint with a single version of Python
# For more information see: https://help.github.com/actions/language-and-framework-guides/using-python-with-github-actions

name: Python application

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest
```

```
    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.9
      uses: actions/setup-python@v2
      with:
        python-version: "3.9"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars
wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 -
-statistics
    - name: Test with pytest
      run: |
        pytest
```

В файле описываются следующие составляющие потока работ. Условия, при которых поток работ запускается: выполнение commit или pull_request в ветку main:

```
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

Задача потока работ: собрать приложение (build) на виртуальной машине с ОС Linux ubuntu:

```
jobs:
  build:

    runs-on: ubuntu-latest
```

Шаги по сборке приложения – создание копии репозитория и установка нужной версии Python.

```
    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.9
      uses: actions/setup-python@v2
      with:
        python-version: "3.9"
```

Установка пакетов, необходимых для сборки, тестирования и работы приложения:

```
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
```

Запуск линтера flake8, который проверяет синтаксические ошибки и стиль кода приложения. В слудеющем семестры мы подробно рассмотрим работу линтеров, в том числе flake8. Сейчас предлагается оставить эту секцию без изменений.

```
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars
wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 -
```

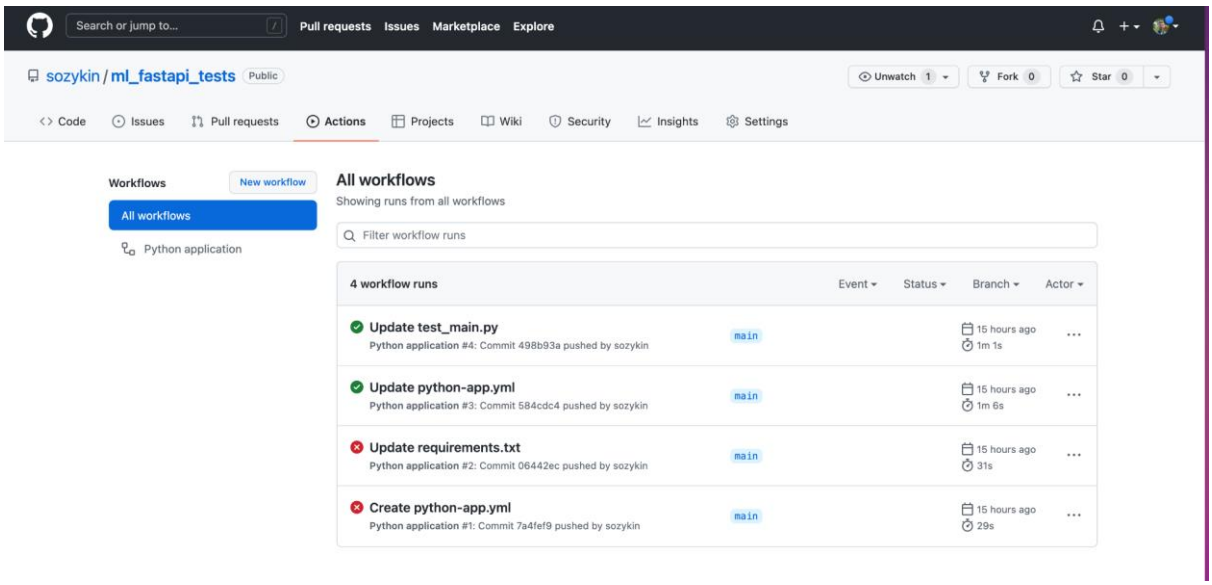
```
-statistics
```

Запуск тестов с помощью pytest:

```
  - name: Test with pytest
    run: |
      pytest
```

Как мы рассматривали в предыдущих юнитах, для запуска тестов используется одна команда pytest. Она автоматически находит код тестов по именам файлов или функций.

Файл с описанием потока работ CI хранится в репозитории на GitHub в каталоге .github/workflows. При наличии такого файла в репозитории, после каждого коммита в репозиторий, GitHub будет создавать виртуальную машину, в которой установит Python и другое необходимое программное обеспечение, которое указано в описании потока работ и файле requirements.txt в репозитории, запустит линтер и тесты. Результаты работы можно увидеть на вкладке Actions.

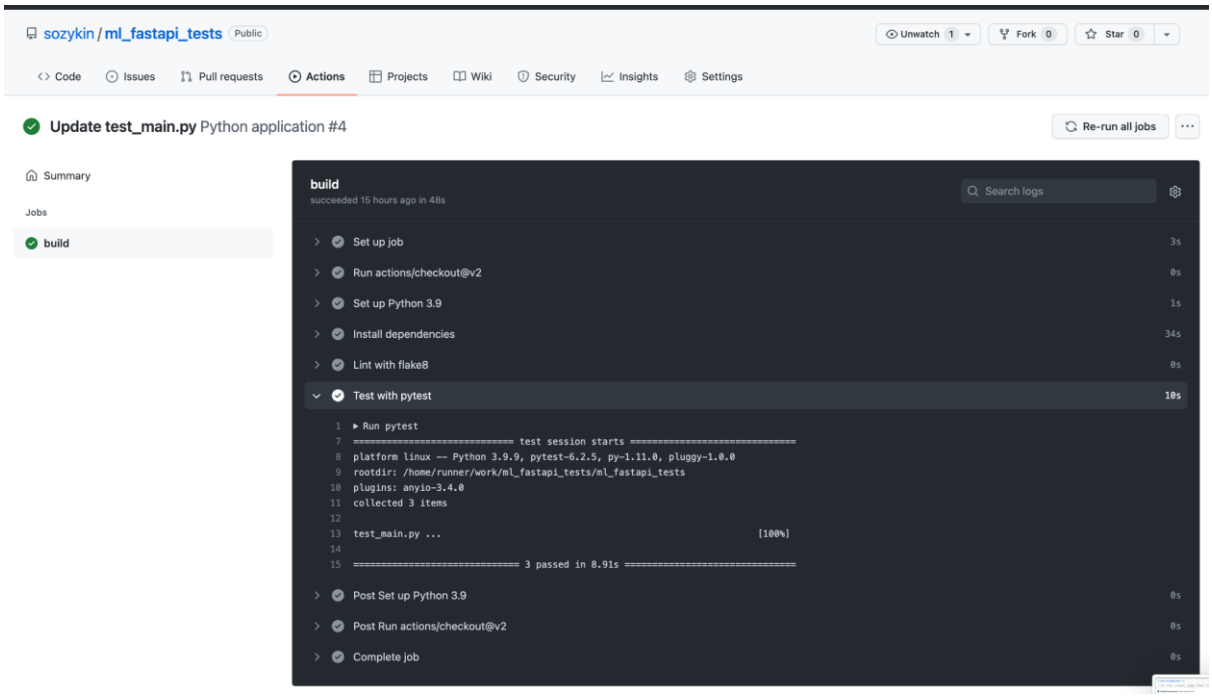


Результаты запусков потоков работ в GitHub Actions

Успешно выполненные потоки работ отмечены зеленой галочкой. Если же при выполнении потока произошли ошибки, например, в тестах, то такие потоки отмечены красным кружком с перечеркнутым крестом.

Рядом с каждым запуском потока работ указывается, каким коммитом он был вызван. Поэтому всегда можно определить, какие изменения в коде привели к ошибкам в сборке.

Посмотреть результаты запуска и причины ошибок можно, щелкнув на название потока работ. Вот пример окна сборки приложения в репозитории https://github.com/sozykin/ml_fastapi_tests/ с результатами запуска тестов, которые мы рассмотрели ранее.



Пример запуска потока работ по сборке и тестированию API приложения машинного обучения

Итоги:

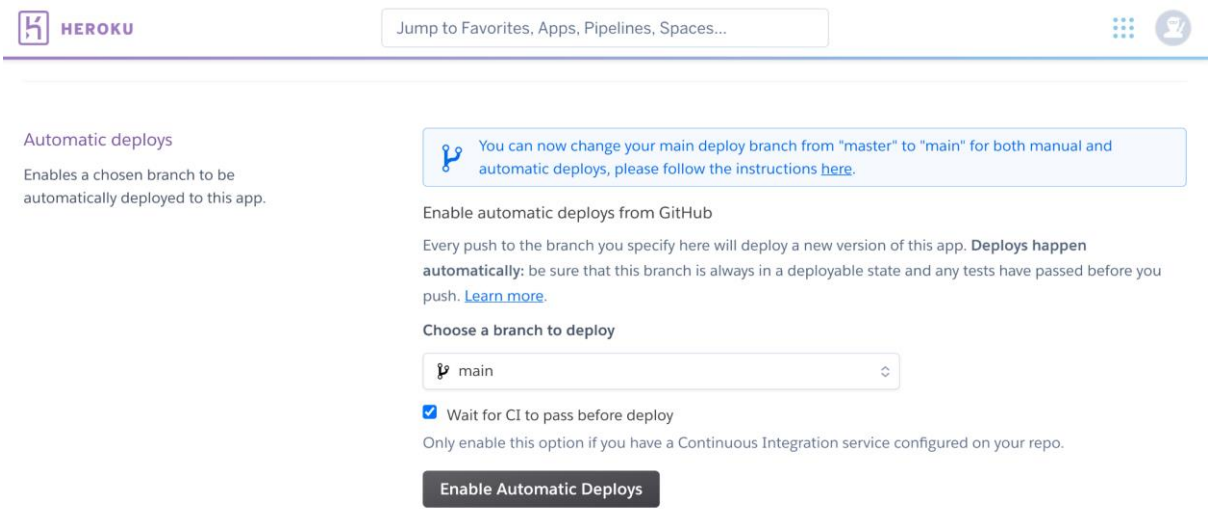
- Практика программной инженерии Continuous Integration включает автоматический запуск тестов при выполнении commit (или других событий) в репозитории с кодом.
- Регулярный автоматический запуск тестов позволяет быстро обнаруживать ошибки в коде.
- GitHub предоставляет простой и бесплатный инструмент для реализации Continuous Integration – GitHub Actions.
- Для настройки GitHub Actions нужно создать файл с описанием желаемого потока работ CI в каталоге .github/workflows репозитория.
- Поток работ GitHub Actions автоматически запускается при выполнении commit в репозиторий на GitHub.

Модуль № 5. Юнит № 4. Автоматическое развертывание приложений на Heroku

Сейчас, когда мы научились создавать тесты для API приложения машинного обучения и автоматически запускать их на GitHub, можно переходить к настройкам автоматического разворачивания приложения на облачной платформе Heroku.

Автоматически разворачивать приложение без выполнения тестов опасно, т.к. изменения в коде могут привести к ошибке в работе приложения. В этом случае приложение не нужно разворачивать, чтобы с такими ошибками не столкнулись пользователи.

Для настройки автоматического развертывания приложения на Heroku, нужно выбрать вкладку Deploy приложения и перейти к разделу “Automatic Deploys”.



Настройка автоматического развертывания приложения на Heroku из репозитория GitHub

В разделе “Automatic Deploys” нужно выполнить следующие действия:

1. Выбрать название ветки для развертывания. Пока у нас всего одна ветка main, поэтому оставляем ее.
2. Поставить галочку “Wait for CI to pass before deploy”. Эта настройка говорит Heroku о том, что развертывание приложения из репозитория GitHub нужно выполнять не сразу после выполнения commit, а только

в случае успешного завершения потока работ CI (в нашем примере GitHub Actions).
3. Нажать кнопку “Enable Automatic Deploys”

Последовательность действий при автоматическом разворачивании приложения на Heroku

В результате выполненных настроект, процесс разработки, тестирования и развертывания приложения с API для модели машинного обучения будет выглядеть следующим образом.

1. Разработчик изменяет код приложения на своей машине в локальной копии репозитория git. Для измененного кода создаются или модифицируются тесты.
2. После завершения изменений в коде, разработчик выполняет commit и отправляет изменения в репозиторий на GitHub.
3. GitHub, получит изменения из локального репозитория разработчика, автоматически запускает поток работ CI GitHub Action.
4. Поток работ GitHub Action запускает тесты для приложения с помощью команды pytest. В случае ошибки выполнения тестов работа останавливается.
5. В случае успешного прохождения тестов приложение автоматически разворачивается на платформе Heroku из репозитория на GitHub.

Итоги

- Регулярное тестирование программного обеспечения позволяет своевременно находить ошибки в нем.
- Для создания тестов на Python используется библиотека PyTest.
- Тестировать API можно с помощью инструментов, встроенных в FastAPI, которые основываются на PyTest.
- Исправление найденных ошибок перед развертыванием приложения для пользователей позволяет повысить стабильность работы приложения.
- Практика программной инженерии Continuous Integration позволяет автоматически запускать тесты при выполнении каждого commit в репозиторий. В результате скорость обнаружения ошибок увеличивается.

- GitHub предоставляет бесплатный и удобный инструмент Continuous Integration: GitHub Actions.
- Приложение, успешно пршедшее тесты на GitHub можно автоматически развернуть на Heroku.

Практическое задание

Цель задания: научиться создавать тесты для API модели машинного обучения и настраивать автоматический запуск тестов на GitHub.

Задание выполняется в команде из 2-3 человек, которую вы сформировали для выполнения предыдущего задания по разработке API для модели машинного обучения.

Задание:

1. Разработайте тесты для API модели машинного обучения, которое вы создали ранее.
2. Разместите файл с кодом тестов в репозитории на GitHub.
3. Настройте автоматический запуск тестов при выполнении commit в репозиторий на GitHub.
4. Настройте автоматическое разворачивание приложения на платформе Heroku в случае успешного завершения тестов.
5. В качестве отчета о выполнении домашнего задания пришлите две ссылки:
    - Ссылка на репозиторий GitHub с кодом приложения и тестами (например, https://github.com/sozykin/ml_fastapi_tests).
    - Ссылка на работающее приложение на платформе heroku (например, https://ml_fastapi_tests.herokuapp.com/)

Термины

Все потенциально-незнакомые и новые понятия выдели отдельно и “положи” сюда.

Список источников

- Библиотека PyTest – https://docs.pytest.org/
- Тестирования API в FastAPI с помощью PyTest – https://fastapi.tiangolo.com/tutorial/testing/

- GitHub Actions – https://docs.github.com/en/actions

Дополнительные материалы

- Пример приложения машинного обучения с тестами – https://github.com/sozykin/ml_fastapi_tests

2 семестр

Модуль № 1
Название: Стиль кода

Образовательные результаты:

- Студент может объяснять важность соблюдения стиля кода при разработке программного обеспечения.
- Студент может оформить код на Python в соответствии с рекомендациями по стилю в PEP 8.
- Студент умеет использовать линтер flake8 в Python для поиска нарушений стиля кода.
- Студент умеет использовать форматтер кода Black для автоматического оформления программы на Python в соответствии с PEP 8.

В этом курсе:

В этом семестре в курсе программной инженерии мы продолжим рассмотрение инструментов созданий крупных программных систем, которые используют машинное обучение.

Основное внимание будет уделено организации эффективной разработки в команде на промышленном уровне. Ранее вы научились использовать распределенную систему контроля версий git. В этом семестре вы изучите продвинутые инструменты командной разработки в git, включая использование веток (branches) для создания новых возможностей приложения, поиск и отмену изменений, приводящих к проблемам, а также создания запросов pull request для внесения изменений в репозиторий.

Мы продолжим рассматривать понятие качества кода. В прошлом семестре для повышения качества кода вы создавали тесты. В этом семестре будут рассмотрены дополнительные техники: стиль кода, рефакторинг, код-ревью.

Также мы продолжим рассматривать инструменты автоматизации разработки и развертывания приложений. В дополнение к практике Continuous Integration, рассмотренной в прошлом семестре, мы изучим

практику Continuous Delivery, которая обеспечивает автоматическое развертывание приложения после внесения изменений в код с учетом соблюдения всех требований по обеспечению качества. Эти практики почти повсеместно используются совместно и сокращенно обозначаются CI/CD (Continuous Integration/Continuous Delivery). Такой подход к разработке позволяет очень быстро делать доступным пользователям новые возможности приложения.

Как и в предыдущем семестре, обучение организовано на основе проекта, который вы будете делать командами из трех-четырех человек. Можно будет продолжить работать над проектом из предыдущего семестра или выбрать новую тему. Например, темой проекта может стать создание чат-бота, отвечающий на часто задаваемые вопросы по заданной тематике на основе библиотеки Deep Pavlov (http://docs.deeppavlov.ai/en/master/features/skills/faq.html) с интеграцией с Telegram (http://docs.deeppavlov.ai/en/master/integrations/telegram.html) и Яндекс.Алисой (http://docs.deeppavlov.ai/en/master/integrations/yandex_alice.html). Работа над проектом будет реализована по шагам в течение всего курса: каждую неделю на практических занятиях вы будете выполнять небольшую часть проекта.

В конце семестра вы в команде разработаете приложение, использующее машинное обучение, и настроите его автоматическое развертывание на облачной платформе из репозитория GitHub с помощью CI/CD. При этом в процессе разработки вы будете использовать промышленные инструменты: продвинутые возможности git, линтеры для соблюдения стиля кода, рефакторинг, код ревью.

Модуль № 1. Юнит № 1. Стиль кода. Руководство по стилю кода в Python PEP 8.

Почему важно соблюдать стиль кода

Гвидо ван Россум, создатель Python, обратил внимание, что при промышленной разработке крупных программных систем разработчики тратят гораздо больше времени на чтение программного кода, чем на его написание. Поэтому очень важно уметь оформлять код таким образом,

чтобы его можно было легко понять. В связи с этим в Дзен Python (https://peps.python.org/pep-0020/) входит правило “Читаемость имеет значение” (Readability counts).

Однако что именно нужно делать, чтобы другим людям было просто понять ваш код? Рекомендации по улучшению читаемости собраны в руководствах по стилю кода на Python, самым известным из которых является PEP 8 – Style Guide for Python Code (https://peps.python.org/pep-0008/). Рекомендации в PEP 8 не являются жесткими правилами: вы можете разработать программу на Python, которая успешно запускается, корректно выполняет то, что требуется, но при этом не соответствует рекомендованному стилю кода. Однако соблюдение рекомендаций по стилю повысит читаемость вашего кода, в результате другим разработчикам из вашей команды будет проще его понять, модифицировать и поддерживать. В том числе таким другим разработчиком можете быть вы сами через полгода после написания кода.

Некоторые крупные компании создают собственные руководства по стилю кода на Python. В качестве примера можно привести Python Style Guide (https://google.github.io/styleguide/pyguide.html) от компании Google. В нем приведены правила стиля, которые разработчикам Google рекомендуется соблюдать в дополнение к PEP 8. Узнайте, если ли руководство по стилю кода на Python в компании, где вы работаете, или куда хотите устроиться.

Рекомендации по стилю кода на Python в PEP 8

Оформление кода

Отступы. Для отступов в Python рекомендуется использовать 4 пробела. В отступах не рекомендуется использовать символы табуляции, а также смешивать пробелы и табуляцию.

Максимальная длина строки. Для кода максимальная рекомендованная длина строки – 79 символов. Строки с комментариями и документацией рекомендуется ограничивать 72 символами.
Ограничение на максимальную длину строки делает удобным работу с несколькими файлами с кодом, которые открыты в расположенных рядом окнах. Например, при просмотре изменений или при проведении код-ревью.

Переносы строк. Строки, длина которых превышает 79 символов, рекомендуется разбивать на несколько используя неявное объединение строк в Python с использованием скобок (круглых, квадратных или фигурных) или с помощью висячего отступа (hanging indent). При висячем отступе только первая строка оформляется с отступом по требованиям Python, а остальные строки могут использовать другой отступ.

Рекомендуется использовать следующие варианты висячего отступа (этот и следующие примеры взяты из документа PEP 8 (https://peps.python.org/pep-0008/)):

```
 # Выравнивание по открывающемуся разделителю
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Добавление 4 пробелов (дополнительный уровень отступа) для отделения
# аргументов от всего остального
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Висячий отступ должен добавлять уровень
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

При использовании бинарных операций рекомендуется разрывать строку перед бинарной операцией.

```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Такой подход был предложен известным ученым в области компьютерных наук Дональдом Кнутом (https://ctan.org/tex-archive/systems/knuth/dist/tex/), автором книг по алгоритмам и издательской системы для математических текстов TeX. Определить, какая именно операция выполняется, получается проще и быстрее, если операция указывается вначале строки, а не в конце.

Пустые строки. Определения функций и классов рекомендуется отделять двумя пустыми строками в начале и конце. Определения методов внутри класса отделяются одной пустой строкой. Также пустые сроки рекомендуется использовать внутри функций для разделения логически независимых блоков.

Импорт. Подключение библиотек рекомендуется выполнять в отдельных строках:

```
# Правильно:
import os
import sys

# Неправильно:
import sys, os
```

Однако импорт нескольких сущностей из одного модуля вполне допускается в одной строке:

```
from subprocess import Popen, PIPE
```

Команды import рекомендуется располагать в начале файла, после комментариев к модулю и строк документации (docstrings).

Следует избегать использования * в импорте (from import *), так как в результате неясно, какие имена присутствуют в глобальном пространстве имен, что вводит в заблуждение как разработчиков, так и многие автоматизированные инструменты.

Строковые кавычки. В Python для строк можно использовать одинарные (') или двойные (") кавычки. Оба варианта равнозначны. Рекомендуется выбрать один вариант и использовать везде. Исключение составляют

строки, которые содержат одинарные или двойные кавычки. Для работы с такими строками рекомендуется использовать кавычки другого типа, в противном случае придется применять маскирующий символ (backslash). Для оформления строк с тремя кавычками рекомендуется использовать двойные кавычки (для совместимости с форматом docstrings, описанным в документе PEP 257 (https://peps.python.org/pep-0257/)), например:

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
```

Соглашения по наименованию. Существует большое количество стилей создания названий объектов в программе. Рекомендации по их использованию в Python приведены в таблице.

Объект	Рекомендации по наименованию	Пример
Переменная или функция	Короткое название из маленьких букв. Можно использовать нижнее подчеркивание для разделения слов.	*Переменные*: classifier status_code *Функции*: predict test_read_predict_negative
Пакет или модуль	Короткое название из маленьких букв. В названиях модулей можно использовать нижнее подчеркивание, а для пакетов это не рекомендуется.	tensorflow sklearn transformers
Класс	Стиль CapWords или CamelCase, при котором для отделения слов используются большие буквы.	BaseModel Item FastAPI
Константы	Название из больших букв, можно использовать нижнее	TOTAL MAX_OVERFLOW

	подчеркивание.	

Руководство по стилю кода PEP 8 содержит большое количество других рекомендаций, которые нужно изучить. Однако запомнить их все и научиться применять очень трудно. Для решения этой проблемы можно использовать специальные инструменты – линтеры, которые находят ошибки в стиле оформления кода. С одним из популярных линтеров, Flake8, мы познакомимся в следующем разделе.

Итоги

- Разработчик значительно больше времени тратит на чтение кода, чем на его разработку.
- Чтобы другие разработчики в команде могли быстро понять, что делает ваш код, нужно соблюдать общие для команды правила стиля кода.
- В Python рекомендации по стилю оформления кода приводятся в документе PEP 8.

Тест

Стиль кода необходимо соблюдать для того, чтобы:

1. **Другие разработчики могли быстро понять, что делает ваша программа.**
2. Можно было отличить код профессионального разработчика от начинающего.
3. Компилятор Python быстрее обрабатывал вашу программу.
4. Стиль кода соблюдать не нужно, это бесполезная трата времени.

Сколько пробелов нужно использовать в отступах, согласно рекомендациям PEP8?

1. 2 пробела.
2. PEP 8 рекомендует использовать символ табуляции вместо пробелов.
3. **4 пробела.**
4. PEP 8 не рекомендует использовать отступы в программе на Python.

В коде вы встретили название MIN_ACCURACY. Какой объект в программе может быть назван таким именем, при условии, что разработчик соблюдает рекомендации по стилю кода PEP 8:

1. Класс.
2. Переменная.
3. **Константа.**
4. Функция.

Модуль № 1. Юнит 2. Проверка стиля кода и наличия ошибок с помощью линтеров. Линтер flake8 для Python.

Выучить все рекомендации стиля оформления кода из PEP 8, а также из других руководств по стилю, которые вам нужно использовать, достаточно сложно. Еще сложнее постоянно соблюдать все эти рекомендации. К счастью, специалисты по программной инженерии постоянно ищут пути повышения эффективности разработчиков и автоматизации их работы. Линтеры — специальные инструменты, которые были созданы для автоматизации стиля кода.

Название линтер произошло от системы lint (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.1841), которая занималась проверкой корректности программ на языке C. Со временем все программы, которые решают похожие задачи, стали называться линтерами.

Для Python разработано несколько вариантов линтеров, самыми популярными из них являются pep8 (https://github.com/PyCQA/pycodestyle), Flake8 (https://github.com/pycqa/flake8), PyLint (https://pylint.org/). Мы подробно рассмотрим работу линтера Flake8. Однако все линтеры похожи друг на друга, поэтому, поняв принципы работы одного линтера, вы легко разберетесь с тем, как работают другие.

Линтер Flake8

Flake8 (https://github.com/pycqa/flake8) – это линтер для программ на Python с открытыми исходными кодами. Он позволяет находить ошибки в стиле оформления кода, а также нарушение других конвенций Python.

Flake8 не является независимым линтером, а позволяет запускать несколько популярных инструментов для одновременной проверки кода:

- PyFlakes (https://github.com/PyCQA/pyflakes)
- pycodestyle (https://github.com/PyCQA/pycodestyle)
- Ned Batchelder's McCabe script (https://github.com/PyCQA/mccabe)

Кроме перечисленных инструментов Flake8 может использовать плагины для выполнения дополнительных проверок.

Популярность Flake8 объясняется сочетанием просты в использовании и широких возможностей по проверке кода.

Установка Flake8

Установка flake8 выполняется с помощью команды:

```
pip install flake8
```

Запуск Flake8

В командной строке применить линтер Flake8 для проверки кода в файле можно следующим образом:

```
flake8 program.py
```

Также можно проверить все файлы проекта, для этого при запуске flake8 вместо имени файла с программой нужно указать название каталога с проектом:

```
flake8 project_folder
```

Пример использование Flake8

Давайте попробуем применить линтер Flake8 для поиска ошибок в файле со следующей программой, реализующей API для модели машинного обучения:

```
from fastapi import FastAPI
from transformers import pipeline
```

```
from pydantic import BaseModel
import json

class Item(BaseModel):
    text: str

app = FastAPI()
classifier = pipeline("sentiment-analysis")

@app.get("/")
def root():
    return {"message": "Hello World"}

@app.post("/predict/")
def predict(item: Item):
    return classifier(item.text)[0]
```

Программа находится в файле main.py. Перед запуском линтера попробуйте сами найти проблемы со стилем оформления кода в этой программе.

Чтобы найти нарушения стиля кода в этой программе нужно выполнить команду:

```
flake8 main.py
```

В результате запуска flake8 мы получим следующие результаты:

```
% flake8 main.py
main.py:4:1: F401 'json' imported but unused
main.py:6:1: E302 expected 2 blank lines, found 1
main.py:9:1: E305 expected 2 blank lines after class or function definition, found 1
main.py:10:44: W291 trailing whitespace
main.py:12:1: E302 expected 2 blank lines, found 1
main.py:16:1: E302 expected 2 blank lines, found 1
main.py:20:1: W391 blank line at end of file
```

Давайте рассмотрим формат вывода сообщений flake8 на примере первой строки:

```
main.py:4:1: F401 'json' imported but unused
```

Вывод flake8 включает четыре основные части:

- Имя файла с ошибкой, в данном примере 'main.py'. Имя файла полезно знать, когда мы запускаем линтер для проверки проекта, включающего несколько файлов.
- Строка и столбец в файле, на которых обнаружена ошибка. В примере '4:1' – четвертая строка, первый столбец.
- Код ошибки, в примере F401. Коды сообщений мы подробно рассмотрим далее.
- Поясняющее сообщение. В примере сообщение "'json' imported but unused" говорит нам о том, что импортированный модуль json не используется в программе.

Давайте рассмотрим, что означают другие ошибки. Начнем с тех ошибок, которые встречаются чаще всего:

```
main.py:6:1: E302 expected 2 blank lines, found 1
main.py:9:1: E305 expected 2 blank lines after class or function definition, found 1
main.py:12:1: E302 expected 2 blank lines, found 1
main.py:16:1: E302 expected 2 blank lines, found 1
```

Ошибки с кодами E302 и E305 говорят нам о том, что для отделения определения функций и классов используется одна строка, а не две, как рекомендуется в PEP8. Такие ошибки найдены в строках 6, 9, 12 и 16.

Следующее сообщение выглядит так:

```
main.py:10:44: W291 trailing whitespace
```

Код W291 с описанием 'trailing whitespace' означает, что в строке с кодом содержатся завершающие пробелы, что не рекомендуется. Проблема встречается в строке 10, столбец 44.

И последнее сообщение выглядит следующим образом:

```
main.py:20:1: W391 blank line at end of file
```

Сообщение 'blank line at end of file' говорит о том, что в конце файла есть пустая строка. Пустые строки в конце файла использовать не рекомендуется.

Чтобы исправить замечания линтера мы уберем импорт неиспользуемого модуля json, сделаем так, чтобы до и после объявления функций и классов было по две пустых строки, уберем пробелы в конце десятой строки и удалим пустую строку в конце файла. В результате программа будет выглядеть так:

```
from fastapi import FastAPI
from transformers import pipeline
from pydantic import BaseModel


class Item(BaseModel):
    text: str


app = FastAPI()
classifier = pipeline("sentiment-analysis")


@app.get("/")
def root():
    return {"message": "Hello World"}


@app.post("/predict/")
def predict(item: Item):
```

```
    return classifier(item.text)[0]
```

Давайте запустим Flake8 для измененной версии программы:

```
% flake8 main.py
```

В этот раз линтер завершит свою работу без выдачи каких-либо сообщений. Это означает, что ошибки в коде линтер не нашел.

Коды ошибок в Flake8

Flake8 в процессе работы выдает ошибки различных типов, получаемые от базовых инструментов, используемых для анализа кода: PyFlakes, pycodestyle и Ned Batchelder's McCabe script. Основные типы ошибок приведены в таблице.

Формат кода ошибки	Описание ошибки	Пример	Подробная информация
FXXX	Ошибка, найденная pyflakes	F401 'json' imported but unused	https://flake8.pycqa.org/en/latest/user/error-codes.html
EXXX	Ошибка, найденная pycodestyle	E302 expected 2 blank lines, found 1 E305 expected 2 blank lines after class or function definition, found 1	https://pycodestyle.pycqa.org/en/latest/intro.html#error-codes
WXXX	Предупреждение от pycodestyle	W291 trailing whitespace W391 blank line at end of file	https://pycodestyle.pycqa.org/en/latest/intro.html#error-codes
C901	Ошибка, найденная McCabe script	C901 Function is too complex	https://github.com/pycqa/mccabe

По ссылке в столбце таблицы с подробной информацией можно найти, что именно означает та или иная ошибка или предупреждение.

Итоги

- Линтер – это инструмент, который может автоматически проверять корректность программного кода.
- Flake8 – один из самых популярных и простых в использовании линтеров для Python.
- Flake8 позволяет найти ошибки в стиле оформления программы, а также некоторые другие проблемы в коде.

Тест

Для чего используется программа линтер?

1. Запуск тестов.
2. **Проверка соблюдения стиля кода в программе.**
3. Поиск ошибок компиляции программы.
4. Автоматическое исправления проблем со стилем кода.

Линтер Flake8 выдал следующее сообщение об ошибке: 'tensorflow' imported but unused.
Как исправить эту ошибку?

1. Заменить строку импорта на следующую:
   from tensorflow import *
2. Добавить две пустые строки до импорта модуля tensorflow и после него.
3. **Удалить импорт модуля tensorflow, т.к. он не используется в программе.**
4. Переместит импорт модуля tensorflow на первую строку в файле с кодом программы.

Модуль № 1. Юнит 3. Проверка стиля кода в среде разработки

Современные среды разработки на Python, такие как PyCharm и Visual Studio Code, обеспечивают широкие возможности по автоматизации соблюдения

руководств по стилю кода. Для этого можно использовать как собственные средства сред разработки, так и интегрировать внешние линтеры.

Исследования эффективности разработки показывают, что чем раньше будет замечена ошибка, тем быстрее и проще ее исправить. Поэтому обнаружение ошибок, связанных со стилем кода, в среде разработке в процессе кодирования полезнее, чем при запуске линтера после завершения очередного этапа разработки. В этом случае не нужно возвращаться от вывода линтера к исходному коду программы и искать, где ошибка. Исправления можно внести сразу в окне редактирования кода. Кроме того, современные среды разработки не просто находят ошибки в стиле кода, но и часто могут исправить их автоматически.

В этом разделе мы рассмотрим, как использовать средства работы со стилем кода в PyCharm, а также научимся интегрировать PyCharm и линтер Flake8.

Проверка стиля кода в среде разработки PyCharm

PyCharm (https://www.jetbrains.com/ru-ru/pycharm/) – это современная среда профессиональной разработки на Python, автоматизирующая многие процессы разработки, в том числе и соблюдение стиля кода. Давайте вставим код с нарушением стиля из предыдущего раздела в PyCharm и посмотрим, как работают встроенные в PyCharm инструменты работы со стилем кода.

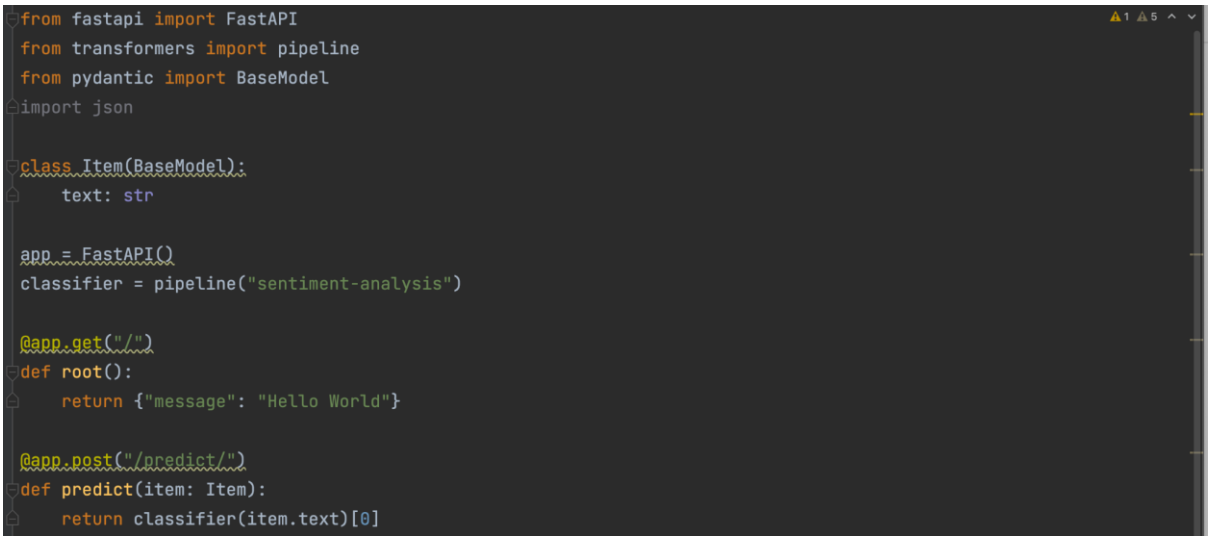


Рис. 1. Внешний вид кода программы с нарушением стиля в окне PyCharm

Нарушение стиля кода в данном файле показываются двумя способами:

- Неиспользуемый импорт модуля json выделен серым цветом.

- Ошибки в оформлении объявлении классов и функций подчеркнуты желтыми волнистыми линиями.

При подведении курсора к найденным PyCharm проблемным строкам кода появляется всплывающее окно с подробным описанием проблемы.

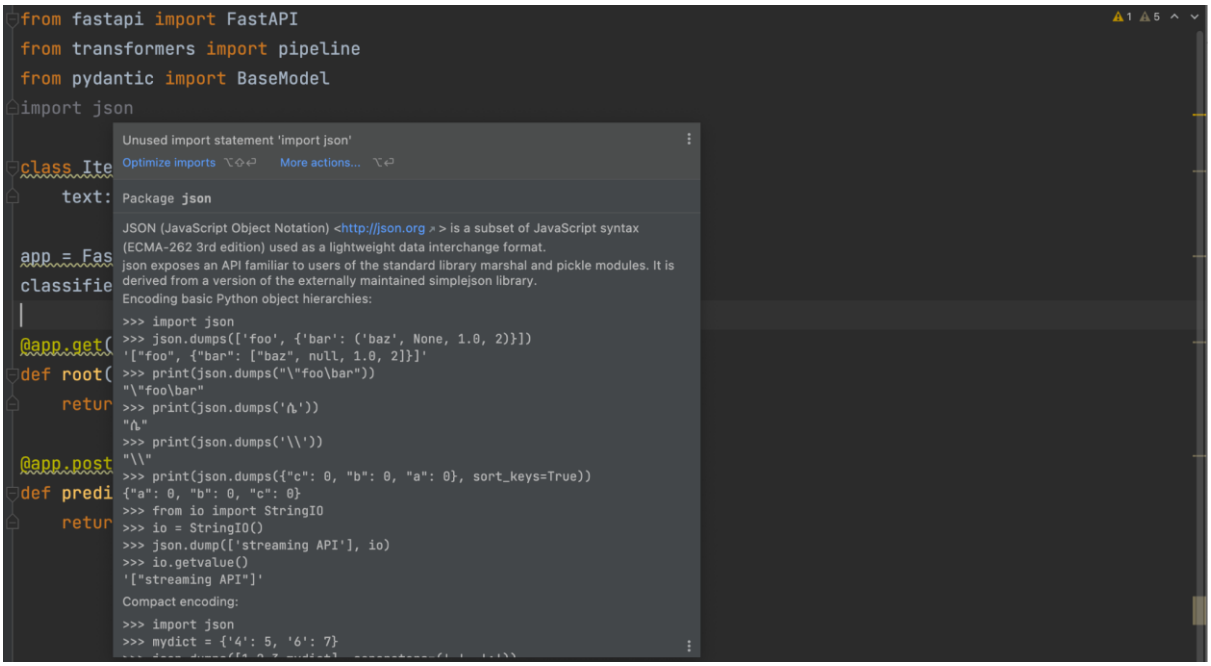


Рис. 2. Всплывающее окно с описанием проблемы импорта неиспользуемого модуля JSON

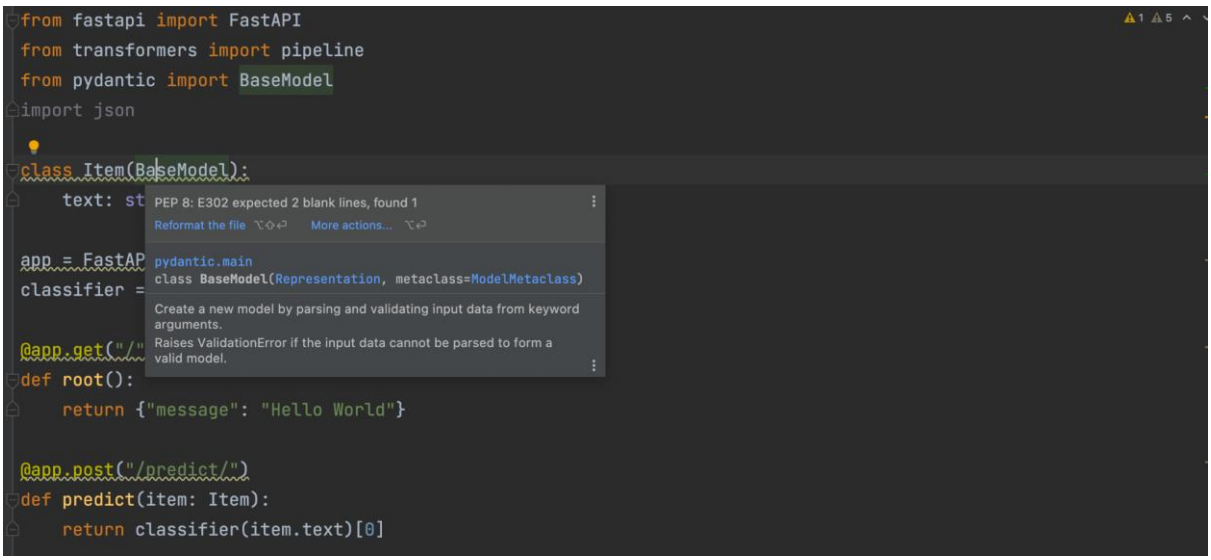


Рис. 3. Всплывающее окно с описанием проблемы нарушения оформления объявления класса

Общее количество найденных PyCharm проблем показывается в правом верхнем углу экрана. Если вы нажмете курсором в этой области, то в нижней части PyCharm откроется панель с общим перечнем ошибок.

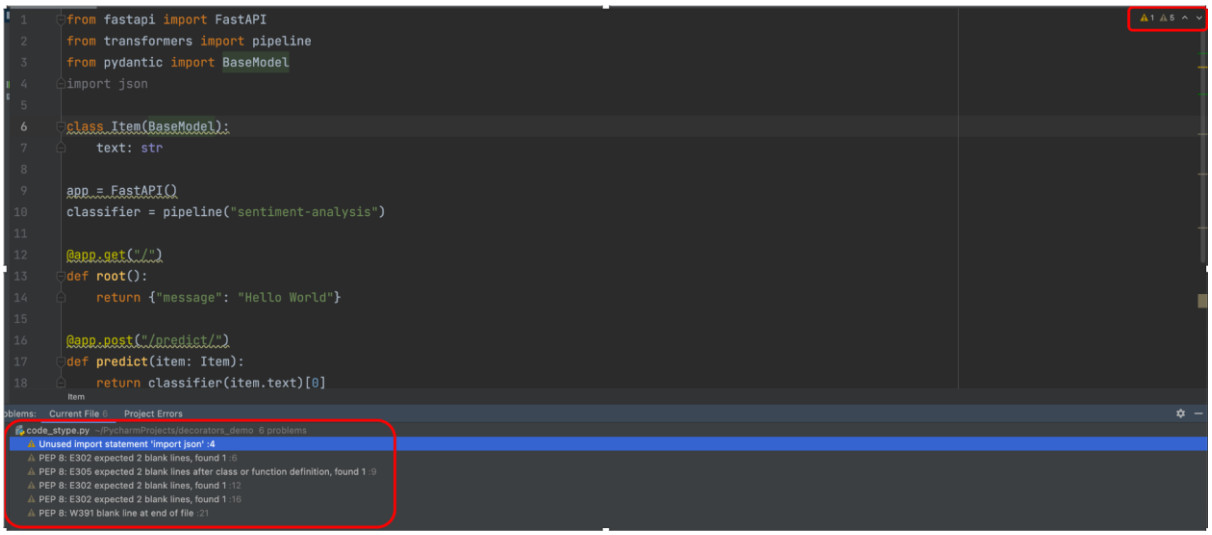


Рис. 4. Общий перечень проблем со стилем кода, которые нашел PyCharm

Однако достоинство среды разработки PyCharm заключается в том, что в ней существует возможность автоматически исправить найденные нарушения стиля. Для этого нужно подвести курсор к месту с найденной проблемой, нажать на правую кнопку мыши и выбрать пункт "Show Context Action".

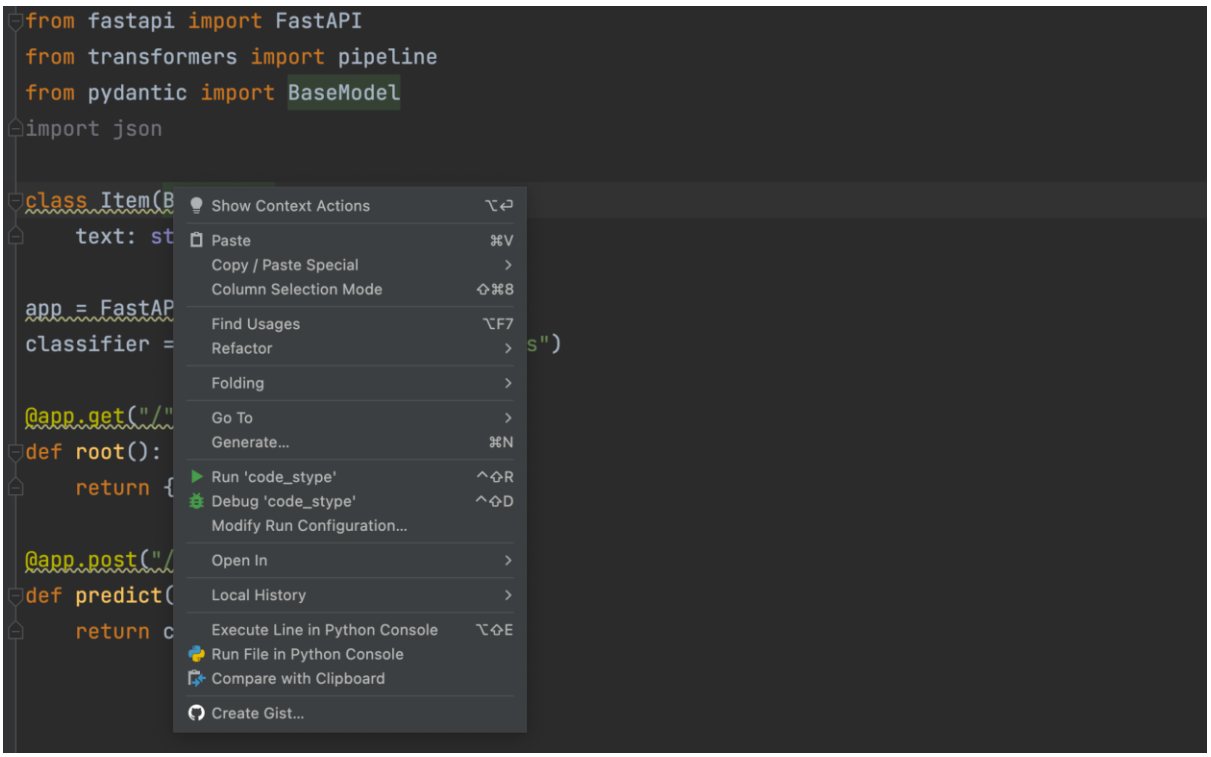


Рис. 5. Выбор действия, связанного с контекстом, в меню PyCharm

В результате откроется меню контекстных действий, которые PyCharm предлагает выполнить для устранения найденной проблемы.

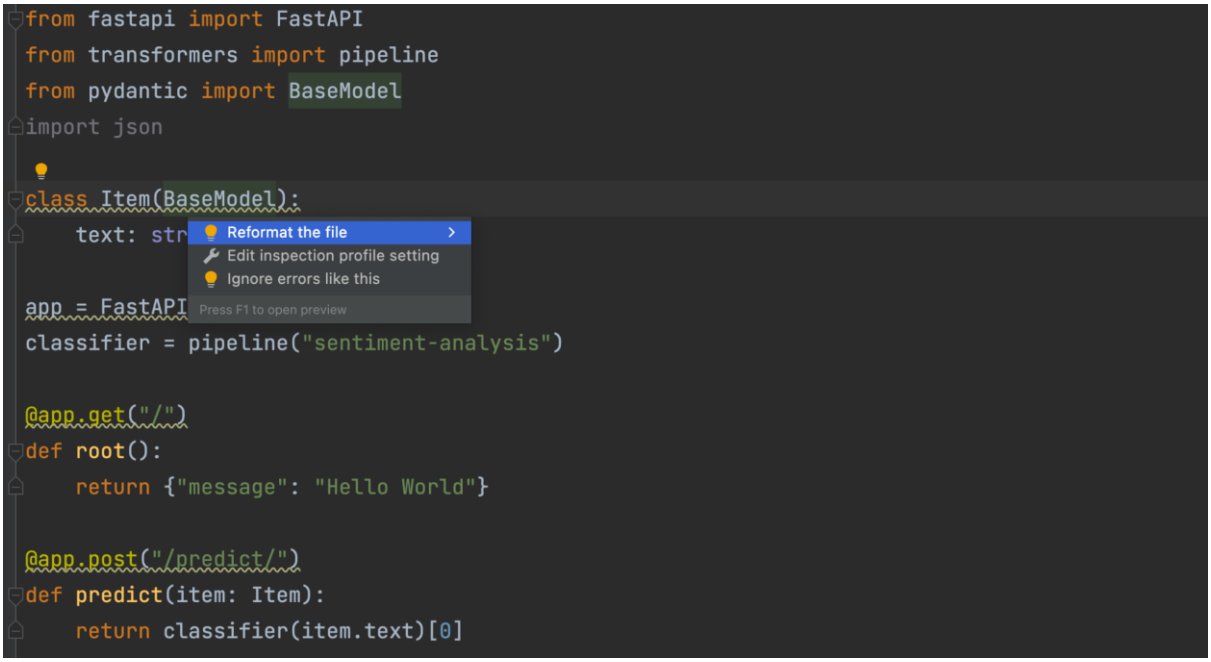


Рис. 6. Перечень контекстных действий PyCharm для устранения нарушений, связанных с форматированием кода

В случае с нарушением форматирования кода при объявлении классов и функций PyCHarm предлагает один из трех вариантов действий:

- Переформатировать файл в целях исправления найденной ошибки (Reformat the file).
- Изменить настройки поиска ошибок (Edit inspection profile setting).
- Игнорировать ошибки такого типа (Ignore errors like this).

Давайте выберем первый вариант и посмотрим, что произойдет.

```python
from fastapi import FastAPI
from transformers import pipeline
from pydantic import BaseModel
import json


class Item(BaseModel):
    text: str


app = FastAPI()
classifier = pipeline("sentiment-analysis")


@app.get("/")
def root():
    return {"message": "Hello World"}


@app.post("/predict/")
def predict(item: Item):
    return classifier(item.text)[0]
```

Рис. 7. Внешний вид окна PyCharm с исправленным форматированием кода

Можно видеть, что исправлено форматирование во всем файле: объявление всех классов и функций отделено двумя пустыми строками в начале и в конце. Таким образом, PyCharm сделал именно то, что требуется.

К сожалению, не всегда проблемы со стилем кода можно решить автоматически средствами PyCharm. Поэтому используйте эти средства с осторожностью, чтобы не повредить код.

PyCharm находит проблемы, вызванные несоблюдением общепринятых руководств по стилю кода, например, PEP 8. Если же в вашей компании или проекте есть дополнительное руководство по стилю, расширяющее PEP 8, то PyCharm может не найти ошибки, связанные с этим дополнительным руководством. Их придется искать самостоятельно, или подключить внешний линтер, который настроен на поиск таких ошибок.

Использование линтера Flake8 в PyCharm

К PyCharm, как и к другим средам разработки на Python, можно подключить внешний линтер. Это полезно в тех случаях, когда линтер умеет находить ошибки, про которые не знает PyCharm. Давайте рассмотрим, как подключить к PyCharm линтер Flake8, с которым мы познакомились ранее.

В PyCharm линтер Flake8 подключается через плагин "File Watcher". Этот плагин позволяет следить за изменениями в файлах проекта и при необходимости предпринимать какие-то действия, например, выводить информацию в окне редактирования кода. Таким образом будет возможность получать информацию об ошибках, найденных Flake8, в процессе редактирования кода в среде разработки.

На первом этапе установим плагин "File Watcher" для PyCharm. Для этого нужно зайти в меню "Preferences" -> "Plugins", в окне установки плагинов найти "File Watcher" и нажать кнопку "Install"

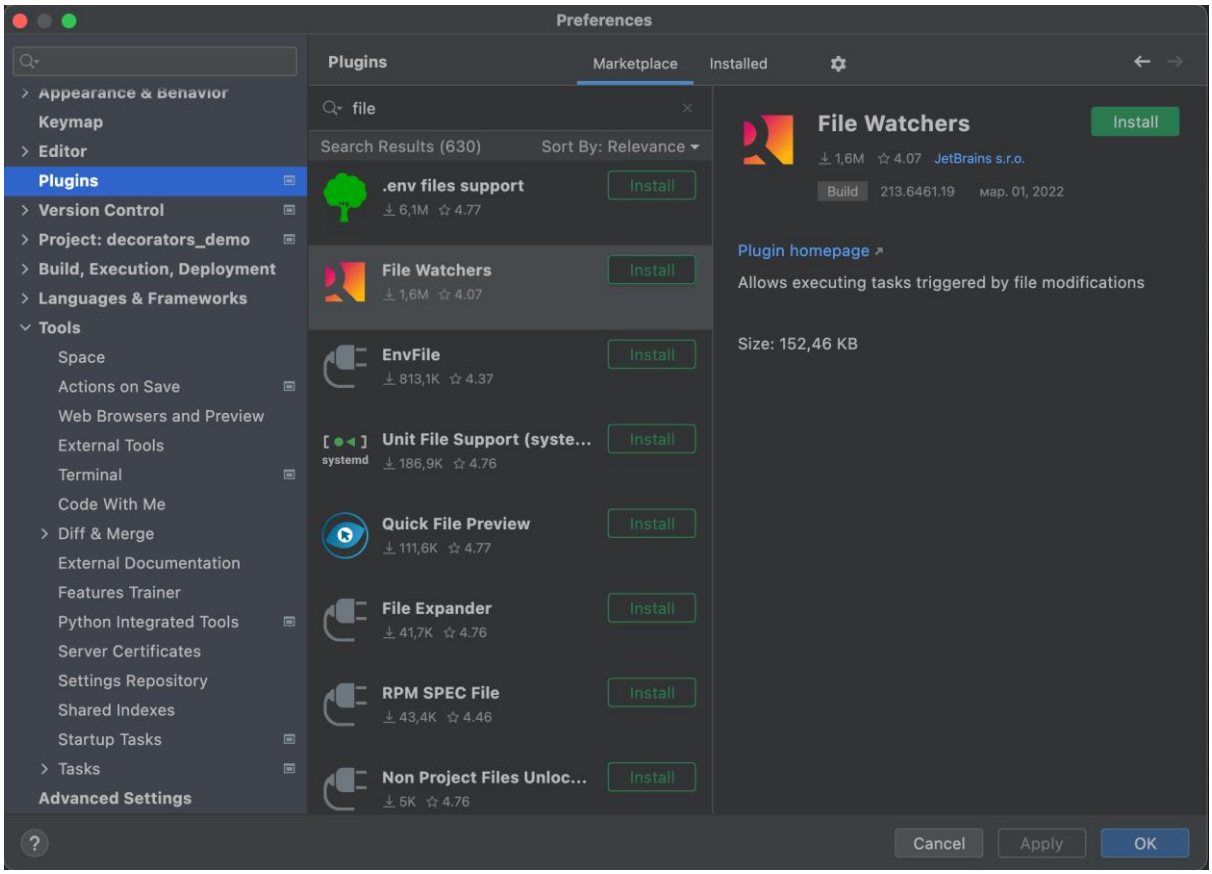


Рис. 8. Установка плагина "File Watcher" в PyCharm

После установки плагина "File Watcher" нужно перезапустить PyCharm, чтобы изменения, связанные с установкой плагина вступили в силу. После этого нужно снова зайти в меню "Preferences" -> "Tools" и выбрать появившийся после установки плагина пункт меню "File Watchers".

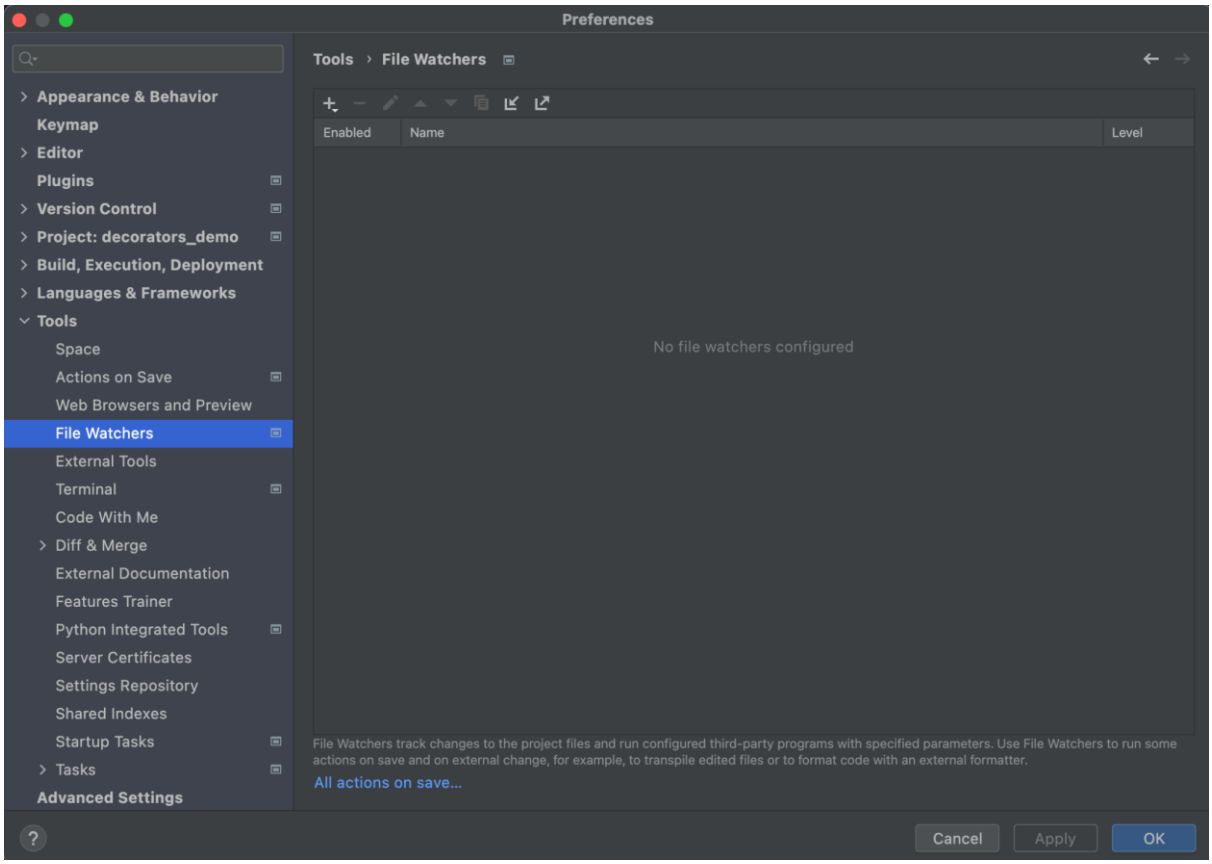


Рис. 9. Настройка File Watchers в PyCharm

Пока таблица на странице конфигурации "File Watchers" пуста. Нужно нажать на кнопку "+", чтобы создать новую запись для Flake8. Появится окно редактирования "File Watcher".

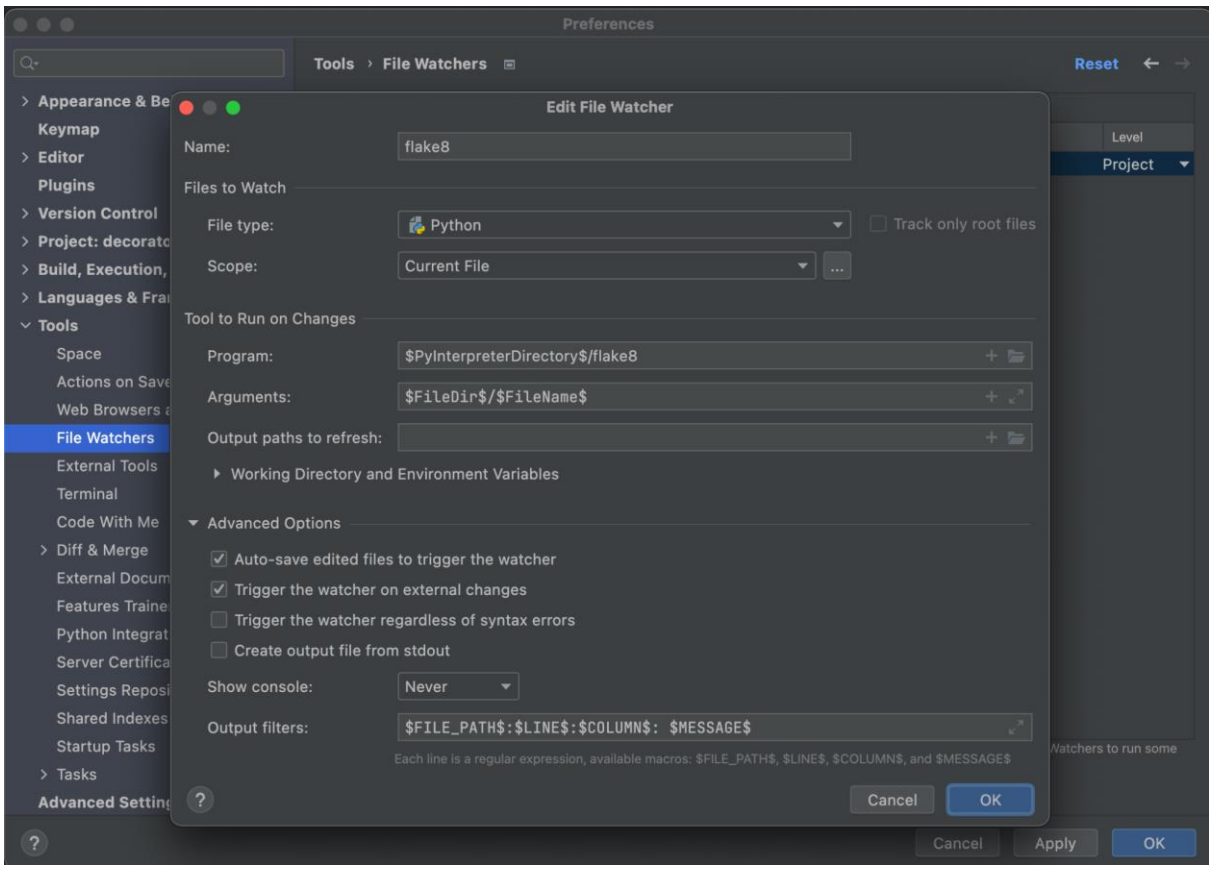


Рис. 10. Настройка File Watcher для Flake8 в PyCharm

Значения полей настройки File Watcher для Flake8 в PyCharm указаны в таблице.

Название поля	**Значение**	**Описание**
Name	flake8	Название File Watcher
File Type	Python	Тип файлов, для которых будет запускаться flake8
Scope	Current File	Область проверки – только для текущего файла, редактирование которого выполняется
Program	$PyInterpreterDirectory$/flake8	Путь к flake8 с использованием макроса PyCharm $PyInterpreterDirectory$ (каталог с интерпретатором

		Python)
Arguments	$FileDir$/$FileName$	Аргументы для запуска flake8 с использованием макросов PyCharm – имя файла и путь к нему.
Show console	Never	Сообщение от линтера не будут показываться в консоли. Мы настроим вывод в окно редактора.
Output Filters	$FILE_PATH$:$LINE$:$COLUMN$:$MESSAGE$	Формат сообщения от линтера (не рекомендуется изменять).

После заполнения всех полей нужно нажать кнопку "ОК" и нужный нам File Watcher, запускающий линтер Flake8 будет создан.

Следующий шаг – настройка вывода информации о проблемах, найденных Flake8 в окно среды разработки. Для этого необходимо настроить правила в пункте меню "Preferences" -> "Editor" -> "Inspections". В окне в середине экрана нужно выбрать пункт "File Watcher" и для него в разделе "Severity" установить уровень предупреждений "Error".

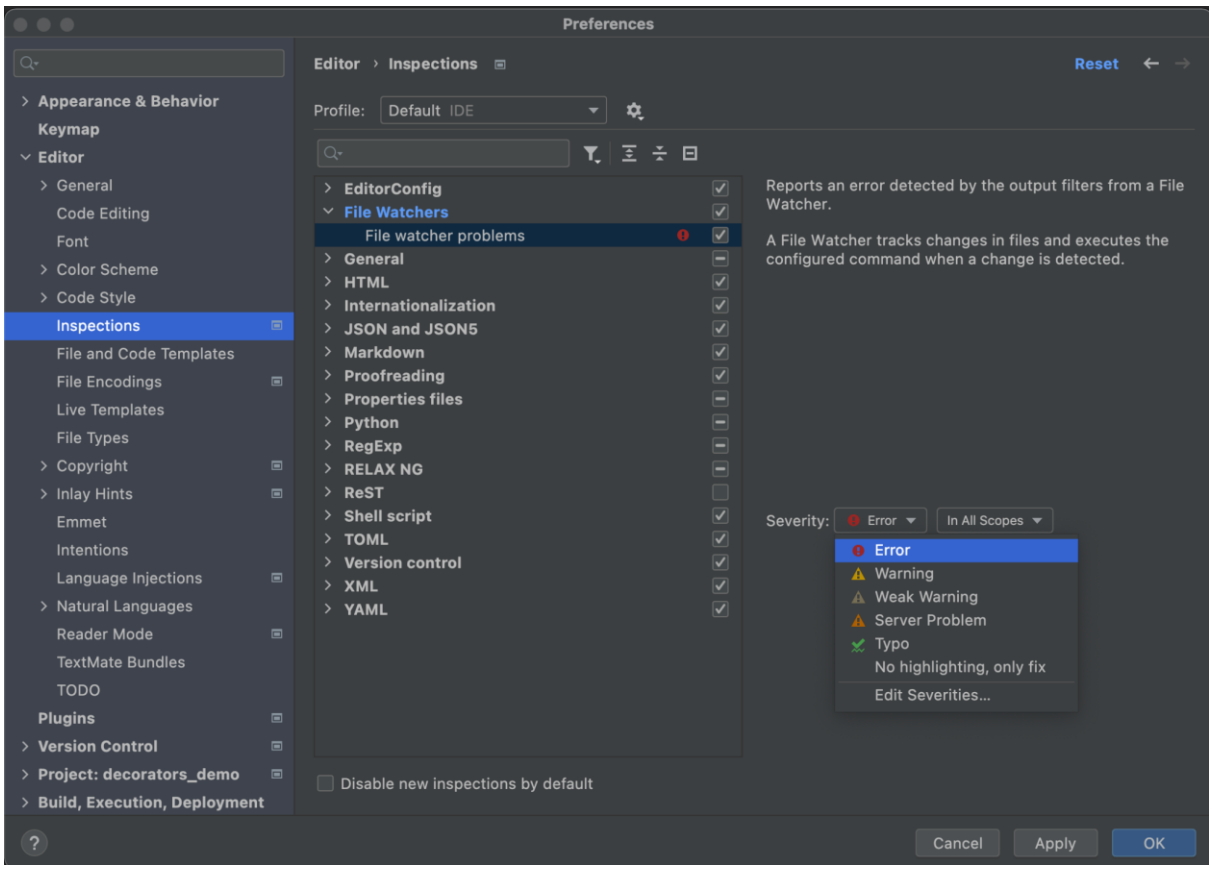


Рис. 11. Настройка правил показа информации от File Watchers в редакторе PyCharm

Проблемы уровня "Error" в редакторе PyCharm выделяются красной волнистой линией. Получить подробную информацию о проблеме можно, подведя к месту проблемы курсор. Описание появится во всплывающем окне, как показано на рисунке.

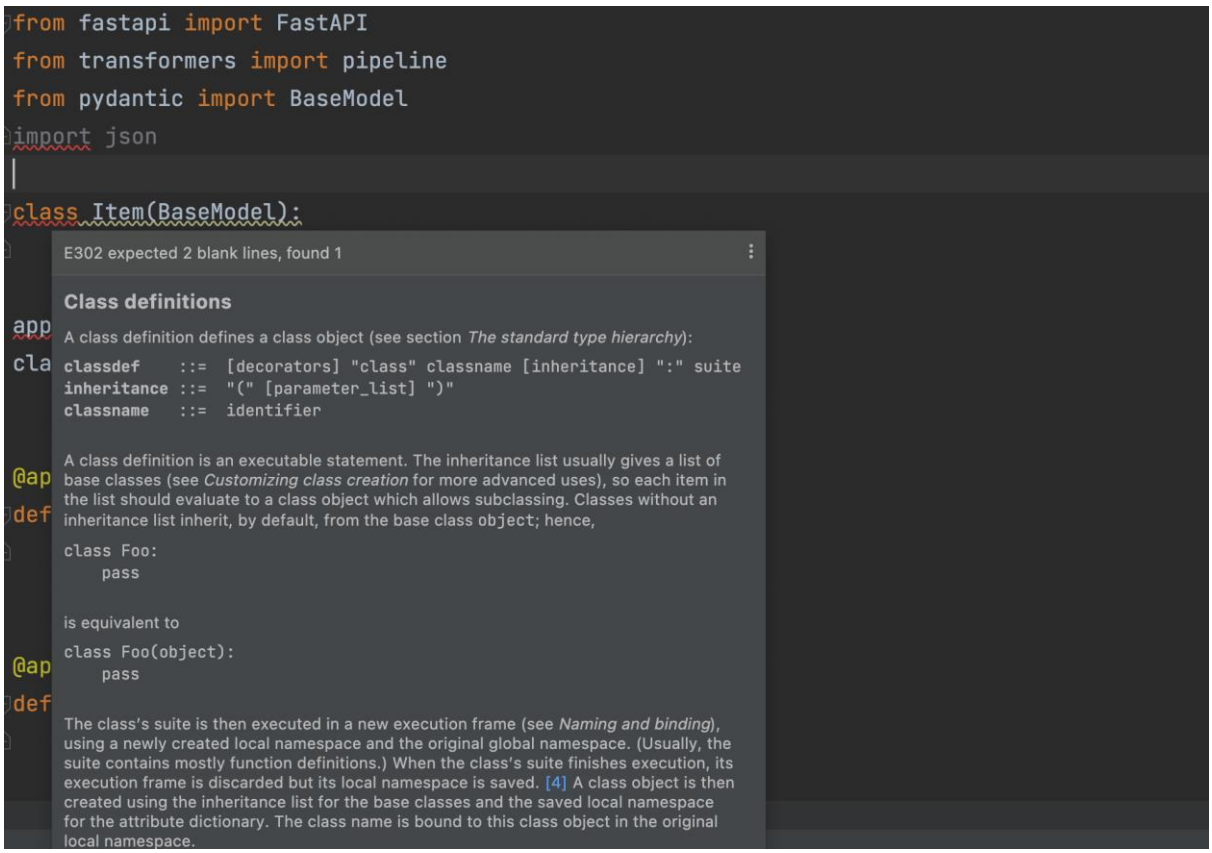


Рис. 12. Сообщение в интерфейсе PyCharm о проблеме, найденной линтером Flake8

Итоги

- Использование среды разработки для поиска проблем со стилем кода позволяет быстро найти и устранить такие проблемы.
- Среда разработки PyCharm обладает встроенными возможностями по поиску и автоматическому исправлению нарушений общепринятых руководств по стилю, включая PEP 8.
- В случае, когда встроенных возможностей среды разработки не хватает, можно подключить внешний линтер и настроить отображение информации от него в окне разработки программы.

Модуль № 1. Юнит 4. Использование линтера Flake8 в GitHub Actions

Вы изучили руководства по стилю кода в Python и соблюдаете их при разработке программ. Но как убедиться в том, что это делают также и все члены вашей команды разработки? Для этой цели можно воспользоваться инструментами Continuous Integration, которые мы изучали в прошлом семестре.

Проверку соблюдения стиля кода, так же как и запуск тестов, можно включить в поток работ, который запускается при выполнении commit в репозиторий с исходным кодом. Давайте рассмотрим, как это делать с использованием инструментов GitHub Actions (https://github.com/features/actions).

В прошлом семестре мы рассматривали, как использовать GitHub Actions для автоматического запуска тестов на GitHub. Для этого мы создавали поток работ (WorkFlow) с использованием шаблона "Python Application".

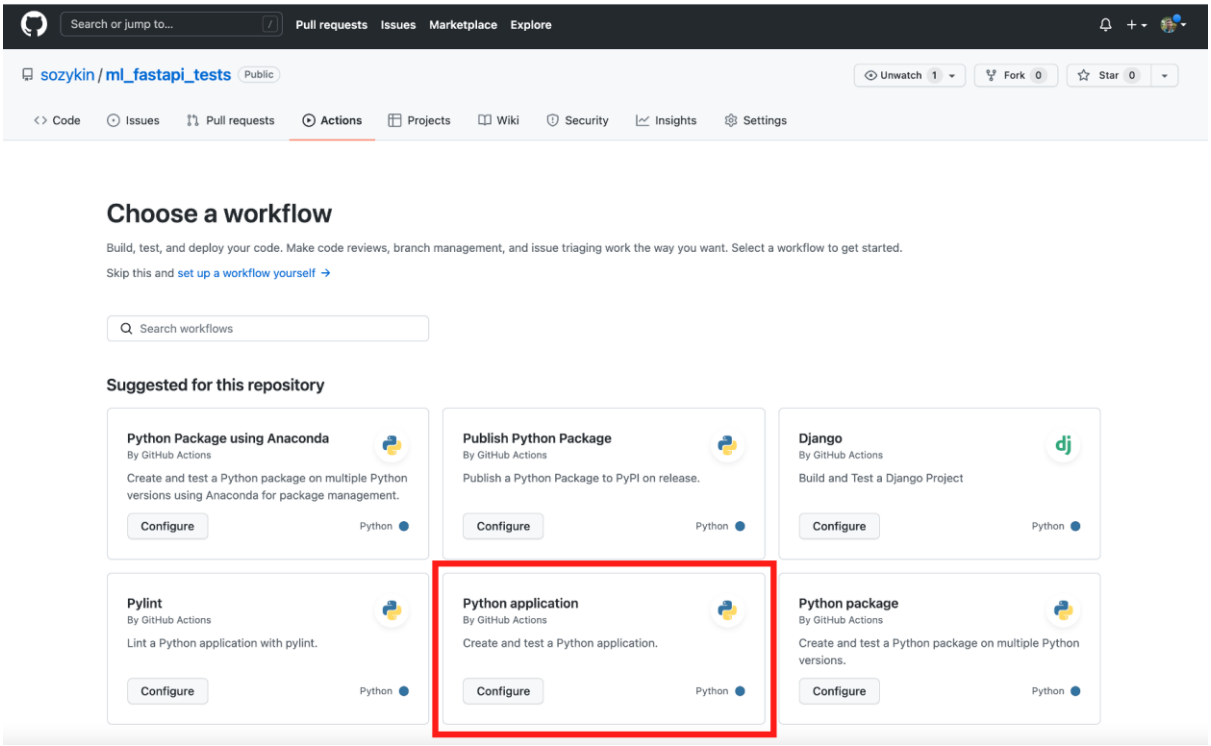


Рис. 13. Выбор типа потока работ GitHub Actions для приложения на Python

В результате создавался следующий файл с описанием потока работ:

```
# This workflow will install Python dependencies, run tests and lint with a single version of Python
# For more information see: https://help.github.com/actions/language-and-framework-guides/using-python-with-github-actions

name: Python application
```

```
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.9
      uses: actions/setup-python@v2
      with:
        python-version: "3.9"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars
wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 -
-statistics
    - name: Test with pytest
      run: |
        pytest
```

Ранее в этом файле мы подробно рассматривали секцию, посвященную тестирования кода с использованием pytest. Однако шаблон GitHub Actions для Python Applications включает также и раздел для проверки стиля кода с использованием Flake8:

```
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars
wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 -
-statistics
```

Таким образом, созданный нами поток работ GitHub Action уже включает запуск линтера Flake8 для проверки стиля кода. Давайте подробно рассмотрим, как это делается.

В потоке работ команда flake8 вызывается два раза с разными параметрами. Параметры используются для того, чтобы ограничить, какой тип ошибок анализируется (с помощью ключа --select).

Первая команда, как можно понять из комментария перед ней, используется для того, чтобы остановит поток работ в случае наличия синтаксических ошибок в коде на Python или неопределенных переменных:

```
flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
```

С помощью ключа --select выбраны коды ошибок, с которыми производится работа: они должны начинаться на E9, F63, F7 или F82. Если будет обнаружена одна из таких ошибок (или несколько таких ошибок), то поток работ завершится неудачно.

Вторая строка проверяет все остальные ошибки. При этом с помощью ключа --exit-zero все ошибки трактуются как предупреждения. Поэтому, в случае обнаружения ошибок, остановки потока работ не происходит.

Сообщения о выявленных ошибках, которые нашел линтер, но которые не привели к остановке потока работ, можно увидеть в журнале сборки проекта в разделе "Lint with flake8"

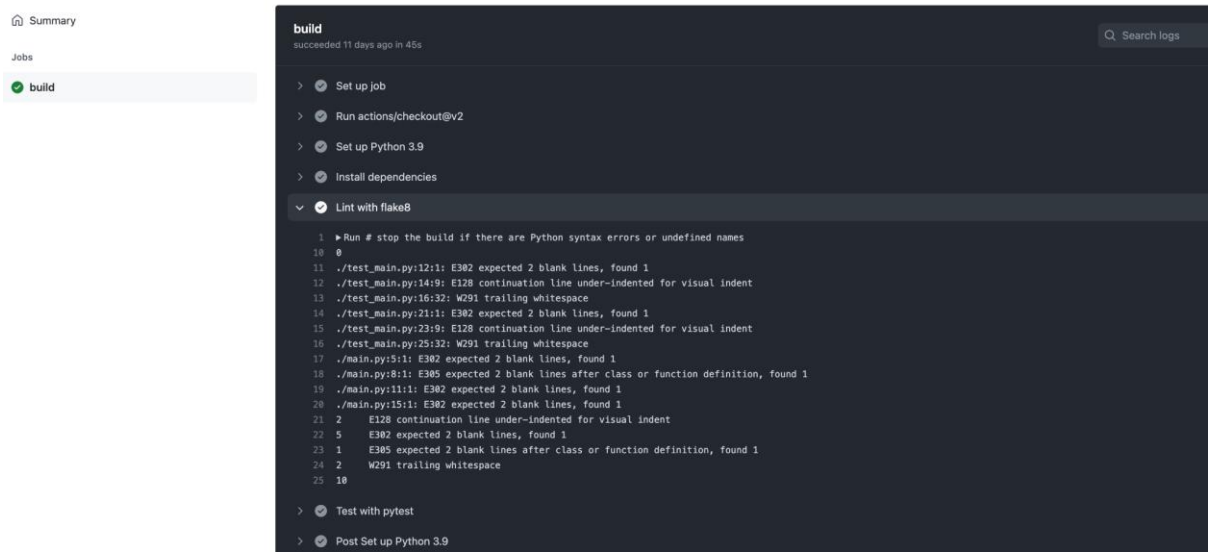


Рис. 15. Информация об ошибках, выявленных линтеров, в журнале потока работ GitHub Actions

Итоги:

- Инструменты Continuous Integration позволяют автоматически запускать линтеры чтобы убедиться в соблюдении стиля кода программы на Python.
- В шаблон приложения Python на GitHub Actions входит запуск линтера Flake8.
- Линтер flake8 запускается два раза:
    - Первый запуск обнаруживает синтаксические ошибки Python и использование необъявленных переменных. В случае обнаружения таких ошибок поток работ Continuous Integration останавливается.
    - Второй запуск ищет все остальные ошибки, причем при обнаружении ошибок поток работ не останавливается. Сообщения об ошибках можно найти в журнале GitHub Actions.
- Использование линтера в инструментах Continuous Integration позволяют автоматизировать соблюдение правил стиля оформления кода в команде разработчиков.

Модуль № 1. Юнит 5. Автоматическое форматирование кода на Python в соответствии с руководством по стилю

Линтеры позволяют находить проблемы, связанные с соблюдением стиля кода, в том числе в автоматическом режиме в инструментах Continuous Integration. Однако, для повышения качества кода и снижения трудозатрат

разработчиков, хотелось бы сделать следующий шаг: автоматически оформлять код программы в соответствии с руководством по стилю. Подобные возможности есть в средах разработки PyCharm и Visual Studio Code, однако не все разработчики могут их применять.

Для того, чтобы решить эту проблему, можно использовать средства автоматического форматирования кода на Python, которые могут исправить ошибки стиля кода. Среди таких форматтеров для Python можно назвать Black (https://github.com/psf/black), autopep8 (https://github.com/hhatto/autopep8), YAPF (https://github.com/google/yapf). Все они имеют открытые исходные коды и распространяются бесплатно. В этом разделе мы рассмотрим, как использовать один из самых популярных форматтеров для Python – Black.

Установка Black

Установка форматтера Black выполняется с помощью команды:

```
pip install black
```

Если вы планируете использовать Black в Jupyter Notebook, то установку нужно выполнять с помощью следующей команды:

```
pip install black[jupyter]
```

Запуск Black

Запуск форматтера Black выполняется похожим образом, как и линтера Flake8. При запуске black можно указать либо файл, который нужно отформатировать, либо каталог, в котором будут отформатированы все подходящие файлы.

Пример запуска форматтера Black:

```
black main.py
```

Пример использования Black

Давайте рассмотрим пример использования форматтера black для исправления ошибок стиля в коде, который мы рассматривали ранее:

```
from fastapi import FastAPI
from transformers import pipeline
from pydantic import BaseModel
import json

class Item(BaseModel):
    text: str

app = FastAPI()
classifier = pipeline("sentiment-analysis")

@app.get("/")
def root():
    return {"message": "Hello World"}

@app.post("/predict/")
def predict(item: Item):
    return classifier(item.text)[0]
```

Запуск форматтера Black для этого файла выглядит следующим образом:

```
 % black main.py
reformatted main.py

All done! ✨□ ✨
1 file reformatted.
```

Форматтер Black показывает, что файл main.py успешно переформатирован. После обработки форматтером файл выглядит следующим образом:

```
from fastapi import FastAPI
from transformers import pipeline
```

```
from pydantic import BaseModel
import json


class Item(BaseModel):
    text: str


app = FastAPI()
classifier = pipeline("sentiment-analysis")


@app.get("/")
def root():
    return {"message": "Hello World"}


@app.post("/predict/")
def predict(item: Item):
    return classifier(item.text)[0]
```

Как можно видеть, исправлены проблемы с форматированием, но импорт не используемого модуля json остался. Таким образом, использование форматтера не отменяет необходимость запускать линтер, т.к. форматтер исправляет не все проблемы.

Просмотр изменений, выполняемых форматтером Black

Чтобы узнать, какие изменения форматтер Black вносит в файл, можно вызвать Black с параметров --diff. В результате будут показаны предлагаемые изменения:

```
python3.9 -m black --diff main.py
--- main.py   2022-03-14 11:08:45.510365 +0000
+++ main.py          2022-03-14 11:08:55.785187 +0000
@@ -1,20 +1,22 @@
 from fastapi import FastAPI
```

```
 from transformers import pipeline
 from pydantic import BaseModel
 import json

+
 class Item(BaseModel):
     text: str

+
 app = FastAPI()
-classifier = pipeline("sentiment-analysis")
+classifier = pipeline("sentiment-analysis")
+

 @app.get("/")
 def root():
     return {"message": "Hello World"}

+
 @app.post("/predict/")
 def predict(item: Item):
     return classifier(item.text)[0]
-
-
would reformat main.py

All done! ✨□ ✨
1 file would be reformatted.
```

Параметр --diff только показывает предлагаемые изменения, исходный файл при этом не изменяется.

Использование форматтера Black в GitHub Actions

Форматтеры, также как и линтеры, эффективно использовать в инструментах Continuous Integration. Black можно интегрировать с GitHub Actions. Для этого в репозитории GitHub нужно создать файл

.github/workflows/black.yml со следующим содержимым (https://black.readthedocs.io/en/stable/integrations/github_actions.html):

```
name: Lint

on: [push, pull_request]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: psf/black@stable
```

Итоги

- Форматтеры позволяют автоматически исправлять ошибки стиля кода в программах
- Один из популярных форматтеров на Python – Black.
- Форматтер Black можно использовать как отдельно, так и в составе инструментов Continuous Integration, включая GitHub Actions.

Практическое задание

Цель задания: научиться использовать линтер flake8 для автоматической проверки качества кода в процессе Continuous Integration.

Задание выполняется в командах из 2-4 человек.

Задание:

1. Оформите стиль кода проекта предыдущего семестра в соответствии с PEP 8.
2. Настройте линтер flake8 в GitHub репозитории проекта предыдущего семестра в Continuous Integration на GitHub Actions.
3. Обеспечьте, что проверка линтером в GitHub Action выполняется без ошибок и предупреждений.

Критерии: - В поток работ GitHub Actions в репозитории с кодом приложения должен быть включен линтер flake8

- Поток работ GitHub Actions должен завершаться успешно.
- Запуск линтера flake8 в потоке работ GitHub Actions должен заершаться без предупреждений (как на рис.15).

Форма сдачи: через платформу, прислать ссылку на репозиторий GitHub с кодом своего приложения.

Дополнительное задание со звездочкой:
Настройте в репозитории GitHub запуск автоформаттера black. Пришлите ссылку на репозиторий GitHub с кодом своего приложения.

Список источников

- The Zen of Python – https://peps.python.org/pep-0020/
- PEP 8 – Style Guide for Python Code – https://peps.python.org/pep-0008/
- Google Python Style Guide – https://google.github.io/styleguide/pyguide.html
- Donald Knuth. The TeXBook – https://ctan.org/tex-archive/systems/knuth/dist/tex/
- PEP 257 – Docstring Conventions – https://peps.python.org/pep-0257/
- Flake8 – https://github.com/pycqa/flake8
- PyLint – https://pylint.org/
- Линтер – это чистый код для ленивых – https://thecode.media/linter/
- Стильный код на Python, или учимся использовать Flake8 – https://habr.com/ru/company/dataart/blog/318776/
- PyCharm IDE для профессиональной разработки на Python – https://www.jetbrains.com/ru-ru/pycharm/
- GitHub Actions – https://github.com/features/actions
- Black Python Code Formatter – https://github.com/psf/black
- autopep8 – https://github.com/hhatto/autopep8
- YAPF – https://github.com/google/yapf
- Black GitHub Actions integration – https://black.readthedocs.io/en/stable/integrations/github_actions.html

Модуль № 2
Название: Продвинутый уровень командной разработки

Образовательные результаты:

- Студент может использовать ветки (branches) в репозитории Git для разработки новых возможностей приложения.
- Студент может объединять (merge) код из ветки с основным кодом приложения в Git репозитории.
- Студент может использовать pull request для внесения изменений в Git репозиторий.

В этом модуле:

В прошлом семестре вы познакомились с системой контроля версий Git, которую применяют для организации командной разработки. Также вы научились использовать сервис GitHub для создания репозиториев Git с открытым доступом.

В этом модуле мы рассмотрим продвинутые возможности системы Git, которые используются для организации промышленной разработки в крупных компаниях.

В больших проектах, над которыми работают крупные команды, изменения редко вносятся в основную версию кода в репозитории. Ведь если над одним и тем же кодом одновременно работают несколько человек, они могут внести изменения, которые противоречат друг другу. В результате приложение окажется неработоспособным. Для того, чтобы решить эту проблему, создаются отдельные независимые версии кода, которые в Git называются ветками (brahches). Для реализации новой возможности в приложении, разработчик создает новую ветку, пишет код для новой возможности в этой ветке, а после завершения разработки объединяет ветку с кодом в основной версии репозитория. В модуле мы подробно рассмотрим, как работать с ветками.

Когда над кодом работает большая команда бывает сложно определить, какие именно изменения привели к потери работоспособности приложения. В этом модуле мы рассмотрим, как искать коммиты, в которых были

выполнены интересующие вас изменения, а также как выполнять отмену изменений, если это требуется.

В модуле вы научитесь работать с ветками в репозиториях на GitHub, а также искать и отменять изменения в таких репозиториях.

Модуль № 2. Юнит № 1. Ветки (branches) в git. Назначение и варианты использования.

Основы работы с ветками в Git

Ветка (branch) в системах контроля версий – это отдельная версия кода в репозитории, не зависимая от основной версии. В некоторых системах контроля версий для создания веток приходилось копировать полностью весь код в репозитории, поэтому создание веток было ресурсозатратной операцией. Однако в Git ветки реализованы легковесно, операции создания веток и переключения между ними выполняются очень быстро. Поэтому в Git ветки используются почти повсеместно.

Ветки удобно создавать, когда над проектом работает несколько человек. Предположим, что один разработчик реализует новую возможность (feature) в приложении, второй исправляет ошибку (bug), найденную тестировщиками, а третий – занимается оптимизацией приложения для повышения производительности. Если всем троим необходимо будет изменить код в одном и том же файле, то могут возникнуть проблемы.

Рассмотрим сценарий, при котором все три разработчика скопировали с помощью Git текущую версию кода из репозитория GitHub на свои локальные компьютеры и работают с локальными копиями репозитория. Пусть первым свою работу завершил разработчик, исправляющий ошибку. Он выполняет коммит своих изменений и отправляет их в репозиторий на GitHub. После этого очередной этап разработки завершил программист, работающий над новой возможностью приложения. Он также выполняет коммит в локальной версии репозитория и отправляет изменения на GitHub. Но так как он отредактировал тот же самый файл, что и разработчик, исправляющий ошибку, то эти изменения очень непросто внести. Требуется внимательно просмотреть изменения, выполненные каждым разработчиком, определить, как они согласуются друг с другом, и решить, как должна

выглядеть итоговая версия кода. Такая операция называется **объединение (merge).** Выполнить объединение автоматически получается не всегда, часто требуется вмешательство человека. Ситуация станет еще хуже, если свои изменения отправит третий разработчик, занимающийся оптимизацией. Придется объединять не две, а три версии одного файла.

Решить проблему с необходимостью объединения различных вариантов изменения одного файла можно несколькими способами.
Первый вариант – каждый разработчик может регулярно проверять обновления в централизованном репозитории на GitHub и изменять свой код в соответствии с полученными обновлениями. Но если над проектом работает достаточно много человек и они регулярно выполняют коммиты в репозиторий в соответсвии с методологией Continuous Integration, то изменений даже в течение одного дня может быть очень много. В результате придется постоянно выполнять обновления кода в соответствии с изменениями в репозитории и времени на разработку может не остаться.

Второй вариант – использование отдельных веток для реализации изменений. В этом случае для новой разработки создается отдельная ветка. В нашем примере можно создать три ветки: *ветку для новой возможности приложения, ветку для исправления ошибки и ветку для оптимизации производительности.* Каждый программист вносит изменения только в свою ветку, они не влияют на работу других программистов. Объединение веток происходит не после каждого коммита, а в конце работы над задачей в целом, что значительно проще реализовать.

Реализация веток в Git

В Git для хранения различных версий данных используется подход на основе серии снимков (snapshot). При выполнении коммита Git запоминает, как выглядит состояние файлов на этот момент времени. В отличие от других систем контроля версий, Git сохраняет не изменения по сравнению с предыдущей версией файла, а полную копию измененного файла. Набор таких измененных файлов и составляет снимок.

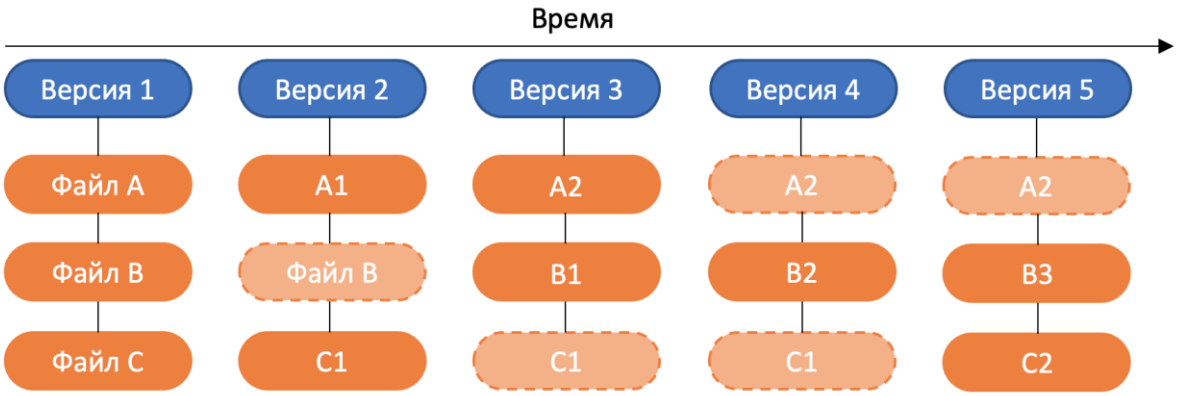


Рис. 1. Хранение данных в Git

Давайте рассмотрим пример изменений трех файлов ("Файл A", "Файл B" и "Файл C"), показанный на рис. 1. В первую версии ("Версия 1" на рисунке) входят исходные файлы A, B и C. После этого программист вносит изменения в файлы A и C и выполняет коммит. Git сохраняет новую версию файлов ("Версия 2" на рисунке). В эту версию входят измененные варианты файлов A и C, обозначенные A1 и C1. Для повышения эффективности хранения, копии файлов, которые не были изменены в коммите, не включаются в снимок. Вместо этого сохраняется ссылка на предыдущий снимок. В примере Файл B не был изменен, поэтому снимок Версии 2 вместо копии Файла B содержит ссылку на Файл B в Версии 1. Снимок версии 3 включает измененные версии файлов A2 и B1 и ссылку на Файл C1 из предыдущей версии. Подобным образом организуются снимки измененных файлов в Версии 4 и Версии 5.

Таким образом, каждый коммит в Git содержит снимок измененных в нем файлов и указатель на предыдущий коммит (рис. 2).

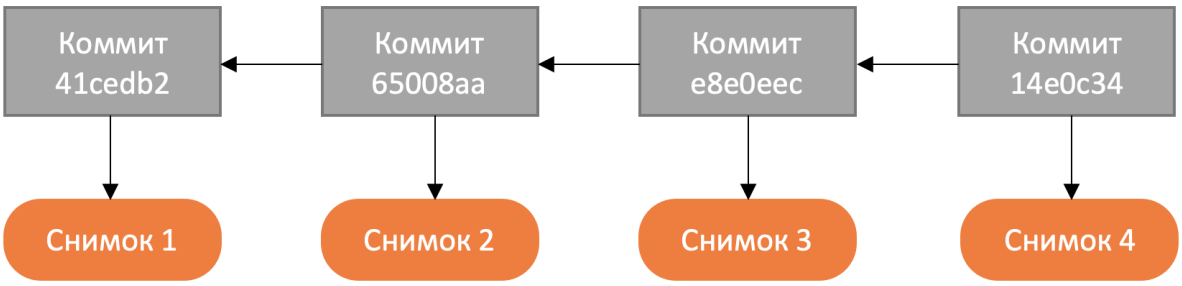


Рис. 2. История коммитов в Git

Ветка в Git – это просто указатель на один из коммитов в истории. По умолчанию в Git используется ветка с названием **main**, которая указывает на последний коммит (рис. 3).

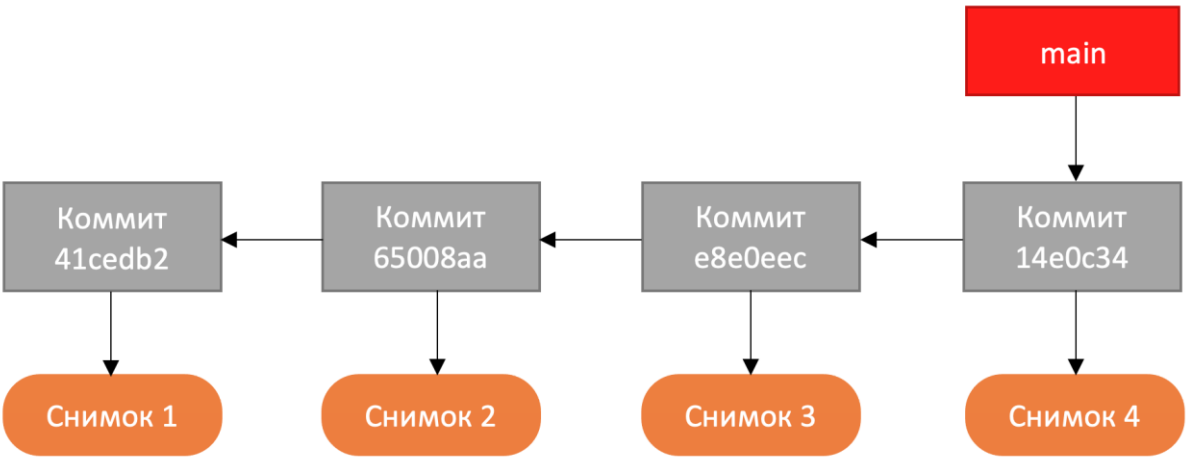


Рис. 3. Ветка main в Git

В ветке main нет ничего необычного, она может использоваться так же, как и все другие ветки. Ранее ветка, которая создается по умолчанию, называлась master.

Можно создавать ветки с различными именами. Например, в дополнение к ветке main можно создать ветку testing. Указатель этой ветки может ссылаться на другой коммит (рис. 4).

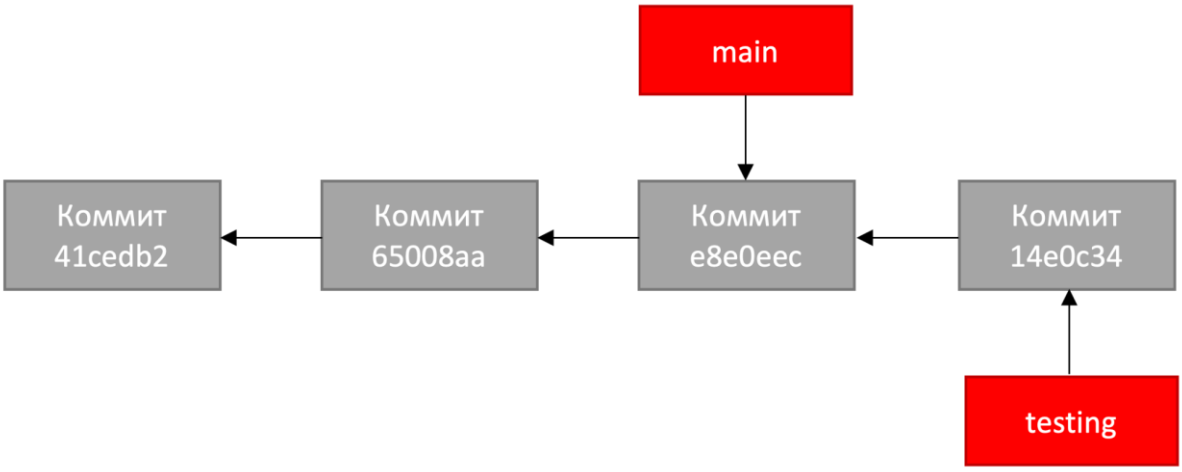


Рис. 4. Указатели веток main и testing

Для того, чтобы определить, в какой ветке ведется работа, Git использует еще один указатель, который называется **HEAD**. Он указывает на ветку, активную в данный момент (рис.5).

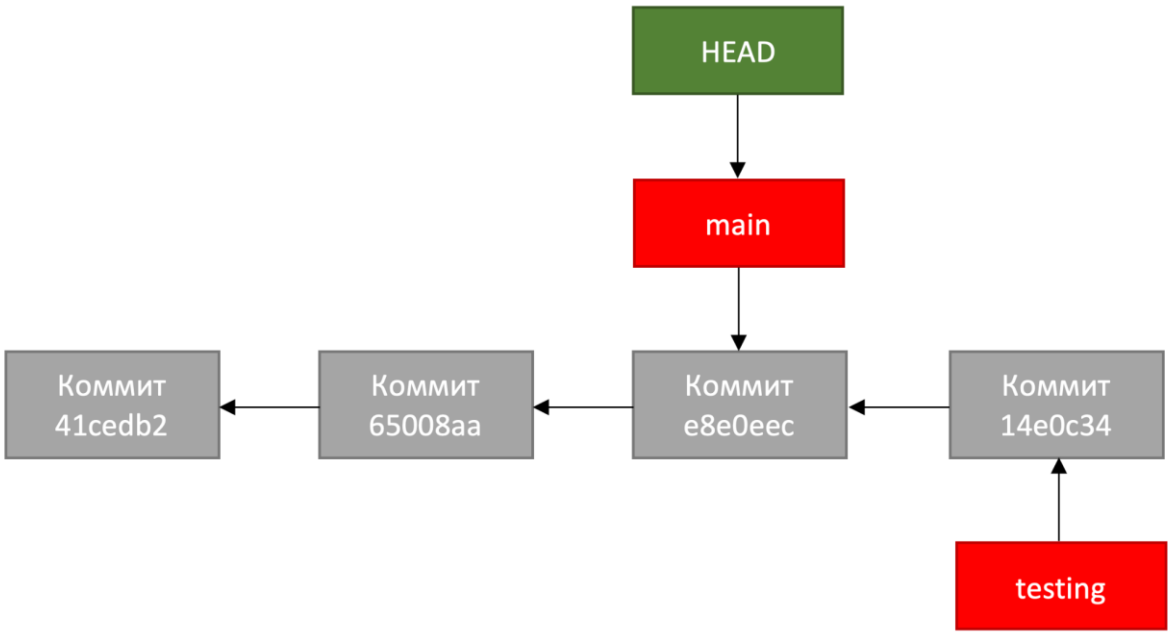


Рис. 5. Указатель на текущую ветку HEAD

Разветвление в Git

Когда мы используем несколько веток в Git, они не обязательно указывают на различные участки одной и той же последовательности кода, как показано на рис. 5. История изменений в разных ветках может разойтись. Например, если мы внесем изменения в ветке main, то ее состояние будет отличаться от состояния ветки testing (рис. 6).

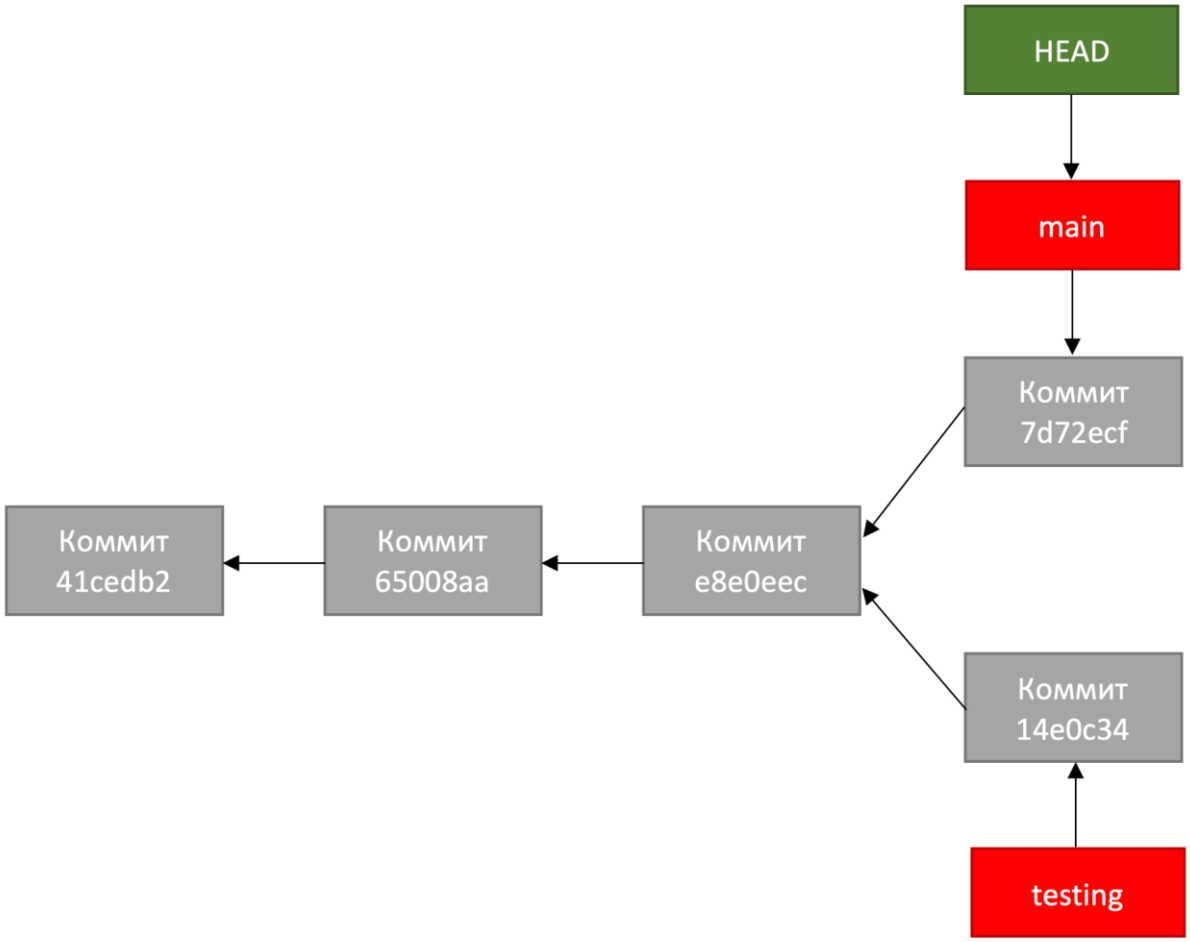


Рис 6. Разветвление истории изменений в Git

После разветвления код в ветках main и testing можно развивать независимо друг от друга. Для этого нужно переключать указатель HEAD на ветку main или testing.

Слияние и перебазирование в Git

Для того, чтобы объединить ветки с разными вариантами кода в Git используются два типа процедур: **слияние (merging)** и **перебазирование (rebasing)**.

При слиянии веток выполняется объединение кода из двух различных версий и создается новый коммит, содержащий это объединение (рис. 7). При этом коммит, созданный в результате объединения, содержит ссылки на два предыдущих коммита из различных веток.

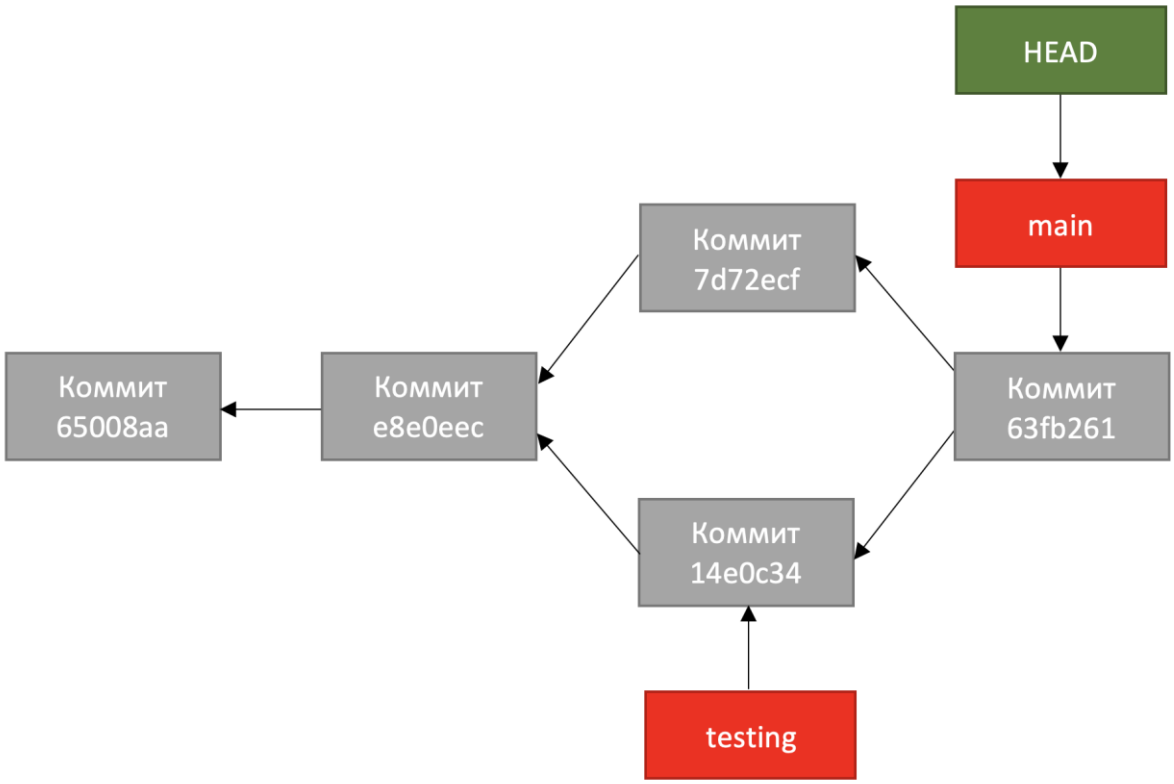


Рис. 7. Слияние веток в Git

Перебазирование устроено более сложным образом. При перебазировании в Git все изменения в одной ветке применяются к другой ветке. В примере перебазирования на рис. 8 после разветвления (коммит e8e0eec) сначала применяются коммиты из ветки main, а после них коммиты из ветки testing.

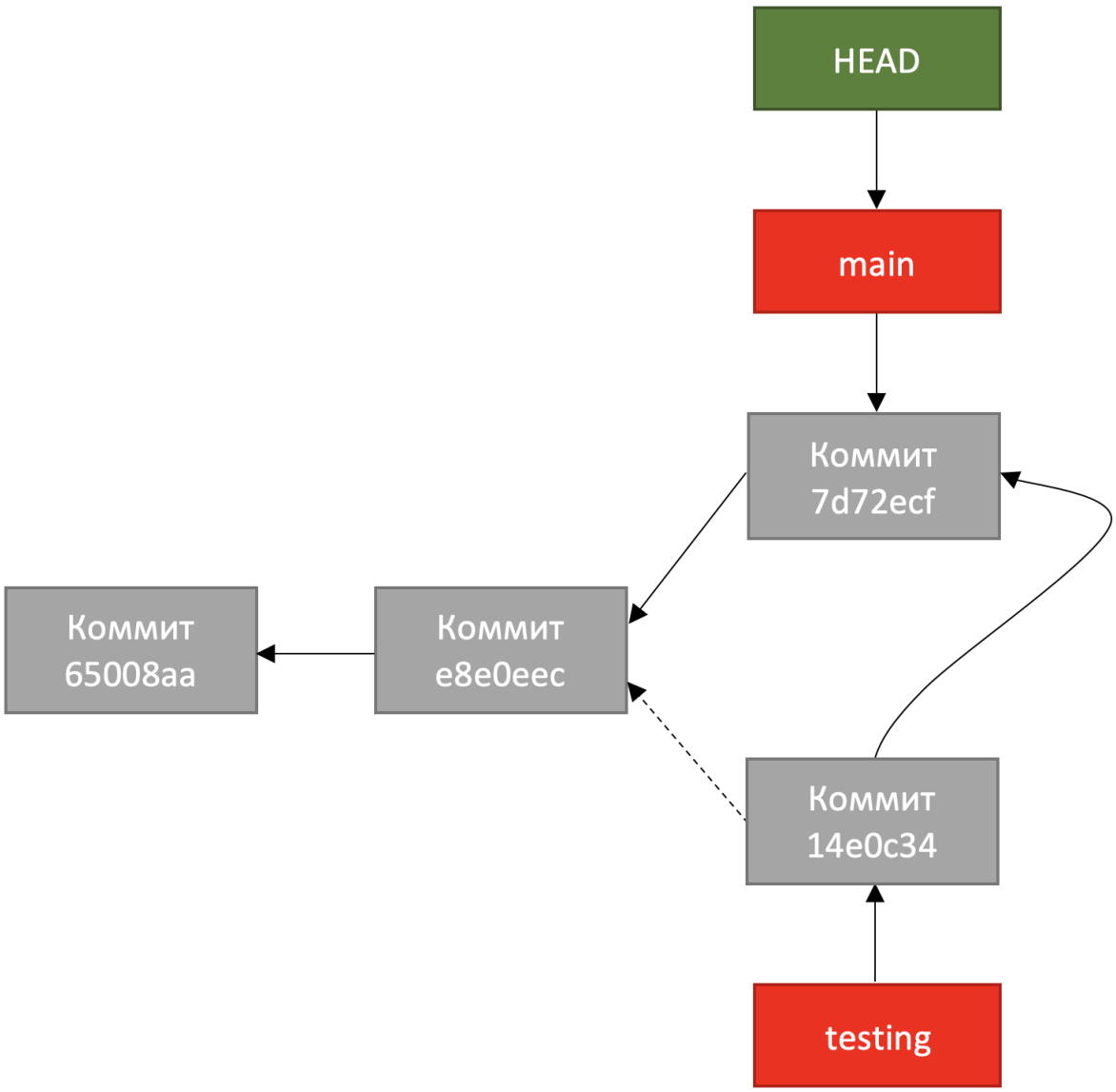


Рис. 8. Перебазирование в Git.

Итоги

- Ветки в системах контроля версий применяются для изоляции изменений, вносимых в код.
- В Git ветки – это указатели на один из коммитов в истории изменений кода в репозитории.
- По умолчанию в репозитории Git создается ветка main, другие ветки нужно создавать самостоятельно.
- Чтобы определить, в какой ветке производится работа, Git использует указатель HEAD.
- История изменений в разных ветках может отличаться друг от друга. В этом случае говорят, что произошло разветвление.

- Объединение изменений их различных веток можно реализовать с помощью слияния или перебазирования.

Тест

Что такое ветка в Git:

1. Указатель на HEAD.
2. **Указатель на один из коммитов в истории изменений.**
3. Точка разветвления истории изменений в репозитории.
4. Инструмент реализации слияния.

Как называется ветка в Git, которая создается по умолчанию:

1. master.
2. testing.
3. **main.**
4. trash.

Какие процедуры используются в Git для объединения изменений из разных веток:

1. **Слияние**.
2. Поглощение.
3. Тестирование.
4. Перебазирование.

Модуль № 2. Юнит № 2. Команды для работы с ветками в Git.

Теперь, когда вы знаете, что такое ветки в Git, и для чего их использовать, давайте рассмотрим, какие команды есть в Git для работы с ветками.

Создание и переключение веток

Имя ветки, с которой выполняется текущая работа, выдается в команде git status:

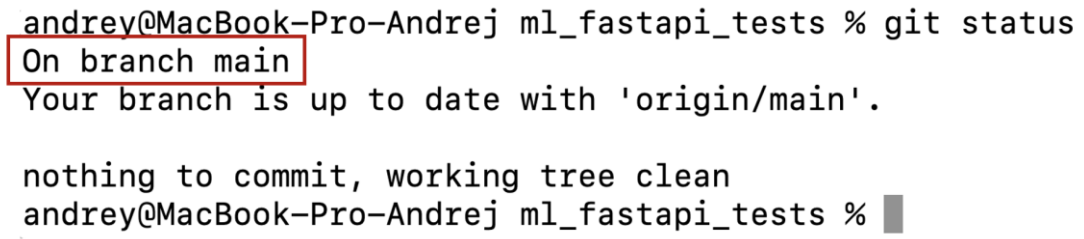


Рис. 9. Имя текущей ветки в git status

Для создания новой ветки используется команда git branch. Например, создать ветку testing можно следующей командой:

```
% git branch testing
```

При создании ветки автоматического переключения на нее не происходит. Переключится на ветку можно с помощью команды git checkout:

```
% git checkout testing
Switched to branch 'testing'
% git status
On branch testing
nothing to commit, working tree clean
```

Как мы рассматривали ранее, чтобы определить, в какой ветке производится работа, в Git применяется указатель HEAD. Посмотреть, куда он указывает, можно с помощью команды git log:

```
% git log --oneline
f5f429c (HEAD -> testing, origin/main, origin/HEAD, main) added
test_main.py
44dd632 Create main.py
14e0c34 Create Procfile
e8e0eec Create requirements.txt
```

Если мы переключимся на ветку main, то в выводе git log можно будет увидеть, что указатель HEAD стал показывать на main:

```
% git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
% git log --oneline
f5f429c (HEAD -> main, origin/main, origin/HEAD, testing) added
test_main.py
44dd632 Create main.py
14e0c34 Create Procfile
e8e0eec Create requirements.txt
```

Для переключения между ветками также можно использовать команду git switch, которая работает аналогично git checkout:

```
% git switch testing
Switched to branch 'testing'
```

Команды git switch и git checkout выполняют два действия:

- Перемещают указатель HEAD на нужную ветку.
- Меняют состояние файлов в каталоге в соответствии с их состоянием в ветке.

Команда git branch без параметров показывает имеющиеся ветки:

```
% git branch
  main
* testing
```

Звездочкой выделена текущая ветка testing

Создание разветвления

Давайте попробуем создать разветвления: внесем независимые изменения в две ветки: main и testing. В качестве примера давайте создадим по пустому файлу в каждой ветке. Файл в ветке main будет называться new_main.py, а в ветке testing: new_testing.py.

Создаем файл в ветке main:

```
% git switch main
Switched to branch 'main'
% touch new_main.py
% git add new_main.py
% git commit -m "Added new_main.py"
[main 422d559] Added new_main.py
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 new_main.py
```

Создаем файл в ветке testing:

```
% git switch testing
Switched to branch 'testing'
% touch new_testing.py
% git add new_testing.py
% git commit -m "Added new_testing.py"
[testing d1f79f0] Added new_testing.py
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 new_testing.py
```

Разветвление можно увидеть в выводе команды git log. Для этого нужно указать два дополнительных параметра:

- --all - показывать информацию обо всех ветках, а не только о текущей
- --graph - показывать граф изменений

Вывод git log для нашего примера будет выглядеть следующим образом:

```
% git log --oneline --graph --all
* 422d559 (main) Added new_main.py
| * d1f79f0 (HEAD -> testing) Added new_testing.py
|/
* f5f429c (origin/main, origin/HEAD) added test_main.py
* 44dd632 Create main.py
* 14e0c34 Create Procfile
* e8e0eec Create requirements.txt
```

В истории изменений есть разветвление после коммита f5f429c:

- Ветка main включает коммит 422d559, в котором добавлен файл new_main.py
- Ветка testing включает коммит d1f79f0, в котором добавлен файл new_testing.py

Указатель HEAD ссылается на ветку testing.

Объединение веток

Для слияния (merge) необходимо перейти в ветку, в которой мы хотим получить объединенную версию кода (в нашем примере это ветка main) и выполнить команду git merge:

```
% git switch main
Switched to branch 'main'
% git merge testing
Merge made by the 'recursive' strategy.
 new_testing.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 new_testing.py
```

Слияние выполнено успешно. Давайте посмотрим граф изменений:

```
% git log --oneline --graph --all
*   981050b (HEAD -> main) Merge branch 'testing'
|\
| * d1f79f0 (testing) Added new_testing.py
* | 422d559 Added new_main.py
|/
* f5f429c (origin/main, origin/HEAD) added test_main.py
* 44dd632 Create main.py
* 14e0c34 Create Procfile
* e8e0eec Create requirements.txt
```

В выводе команды git log мы можем увидеть, что создан новый коммит 981050b, в котором выполнено слияние предыдущих коммитов из двух

разных веток: коммита 422d559 из ветки main и коммита d1f79f0 из ветки testing.

В результате работы команды git merge выполнено объединения разветвления в истории коммитов. Конечный результат слияние содержится в ветке main, на нее же указывает HEAD.

Альтернативный вариант объединения веток: перебазирование с помощью команды git rebase. Для этого нужно перейти в ветку testing и выполнить перебазирование:

```
% git switch testing
% git rebase main
Successfully rebased and updated refs/heads/testing.
```

Теперь все коммиты ветки testing находятся после коммитов ветки main. Можно перенести ветку main на текущее состояние ветки testing с помощью команды слияния:

```
% git switch main
% git merge testing
Fast-forward (no commit created)
```

Команда git merge в данном случае не создает коммита для слияния веток, т.к. история изменений линейная и можно просто передвинуть указатель ветки main на тот же самый коммит, на который указывает ветка testing.

Удаление веток

После объединения, а также в других случаях, когда ветки становятся не нужны, их можно удалять. Для этого нужно вызвать команду git branch с параметром -d и указать название ветки:

```
% git branch -d testing
Deleted branch testing (was 85b95f1).
```

При этом, если вы не выполнили слияние ветки, то при попытке удалить ее вы получите сообщение об ошибке:

```
% git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Как можно понять из сообщения, чтобы удалить ветку, которая не была слита, нужно использовать команду:

```
git branch -D testing
```

Узнать, какие ветки были слиты, можно следующей командой:

```
git branch --merged
```

Как мы уже рассматривали ранее, ветка в Git – это просто указатель на один из коммитов в истории изменений. Поэтому когда вы удаляете ветку, то вы удаляете такой указатель, а не сами коммиты с изменениями. Поэтому слитую ветку можно удалять, даже если в дереве коммитов было разветвление. Коммиты, созданные ранее в ветке, останутся, удалится только указатель.

Итоги

В этом юните вы научились работать с ветками:

- Создавать новые ветки с помощью команды git branch.
- Переключаться между ветками с помощью команд git checkout или git switch
- Объединять ветки командами git merge или git rebase

Однако объединение веток – это не такая простая задача, как кажется на первый взгляд. Если в разных ветках изменялись одни и те же файлы, то выполнить объединение в автоматическом режиме не получится. Что делать в этом случае, мы рассмотрим в следующем модуле.

Тест

Какая команда в Git используется для создания новой ветки featureA?

1. **git branch featureA**
2. git branch -D featureA
3. git branch
4. git switch featureA

Какая команда в Git используется для переключения на ветку featureA?

1. git branch featureA
2. git merge featureA
3. git rebase featureA
4. **git switch featureA**

Какая команда в Git используется для слияния ветки featureA с основной веткой main?

1. git branch main
2. **git merge featureA**
3. git rebase featureA
4. git merge main

Модуль № 2. Юнит № 3. Устранение конфликтов при слиянии веток

Изменения в разных ветках выполняются независимо друг от друга. Поэтому может возникнуть ситуация, когда изменения, выполненные в этих ветках, противоречат друг другу. Например, изменения вносятся в один файл. В этом случае при попытке слияния веток появится сообщение об ошибке.

```
% git merge testing
Auto-merging file.py
CONFLICT (content): Merge conflict in file.py
Automatic merge failed; fix conflicts and then commit the result.
```

В примере мы пытаемся слить ветки main и testing, но это сделать нельзя, т.к. есть конфликт с содержимым (content) файла file.py.

Команда git status будет выдавать сообщение об ошибке слияния и даст подсказку, что нужно изменить, чтобы слияние было выполнено успешно.

```
% git status
On branch main

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   file.py

no changes added to commit (use "git add" and/or "git commit -a")
```

В выводе git status видно, что есть не объединенные ветки (unmerged paths). Команда предлагает два варианта решения этой проблемы:

- Разрешение конфликта и последующий запуск команды git commit для фиксации изменений.
- Отмена объединения веток с помощью команды git merge --abort

Ниже выводится информация, где именно произошел конфликт: обе ветки изменили один и тот же файл (both modified: file.py).

В результате попытки объединения Git подготовил для нас файл вариантами изменений, выполненными в разных ветках.

```
% cat file.py
<<<<<<< HEAD
import tensorflow as tf
=======
import torch
>>>>>>> testing
```

В нашем простом примере файл file.py содержит только одну строки с импортом библиотеки для работы с нейронными сетями. В ветке main, на

которую указывает HEAD, выполняется импорт библиотеки tensorflow. В ветке testing импортируется библиотека torch. Два варианта изменений отделены друг от друга символами "=======".

Нам нужно выбрать один из вариантов. Давайте выберем первый вариант с импортом tensorflow, а остальные строки удалим. Получившийся файл сохраним.

После устранения конфликта в файле file.py нужно зафиксировать изменения в нем:

```
% git add file.py
% git commit
[main 6709f4e] Merge branch 'testing'
```

Команда git commit завершает объединение веток. В результате создан коммит 6709f4e, который содержит результаты слияния ветки testing с веткой main.

Итоги

- При объединение веток могут возникнуть конфликты, например, если в разных ветках внесены изменения в один и тот же файл.
- В случае наличия конфликтов автоматическое объединение веток невозможно. Git останавливает объединение.
- Для устранения конфликтов необходимо вручную выбрать нужные варианты изменений в файлах с конфликтами.
- После устранения конфликтов нужно зафиксировать изменения командой git commit.
- Объединение веток завершается после устранения конфликтов и фиксации изменений.

Модуль № 2. Юнит № 4. Поиск и отмена изменений в Git

Преимущества работы с репозиторием Git заключается в том, что все изменения, включенные в коммиты, сохраняются. Это очень полезно в случае, если потеряна работоспособность приложения в результате

изменений в коде. Всегда есть возможность вернуться к предыдущей работоспособной версии кода.

Поиск изменений

Команда git grep выполняет поиск в текущем каталоге и в истории изменений заданной строки или с помощью регулярного выражения. Например, чтобы найти все файлы со строкой test можно воспользоваться следующей командой:

```
% git grep test
test_main.py:from fastapi.testclient import TestClient
test_main.py:def test_read_main():
test_main.py:def test_read_predict_positive():
test_main.py:def test_read_predict_negative():
```

Вывод команды содержит имя файла и текст в нем, который содержит интересующую нас строку test. В нашем проекте все тесты находятся в одном файле test_main.py.

Если интересует не место расположение строки, а время изменения, то можно использовать команду git log с параметром -S:

```
git log -S test --oneline
f5f429c added test_main.py
```

Изменения со строкой test были внесены в коммите f5f429c. Сообщение коммита – "added test_main.py" – добавлен файл test_main.py. Именно этот файл и содержит тесты.

Более продвинутый вариант: искать историю изменений конкретной функции. Для этого нужно вызвать команду git log с параметром -L :название_функции:название_файла, например:

```
% git log -L :test_read_predict_negative:test_main.py --oneline
f5f429c (origin/main, origin/HEAD) added test_main.py
```

```
diff --git a/test_main.py b/test_main.py
--- /dev/null
+++ b/test_main.py
@@ -0,0 +22,8 @@
+def test_read_predict_negative():
+    response = client.post("/predict/",
+        json={"text": "I hate machine learning"}
+    )
+    json_data = response.json()
+
+    assert response.status_code == 200
+    assert json_data['label'] == 'NEGATIVE'
```

Функция test_read_predict_negative из файла test_main.py была изменена в коммите f5f429c. Именно в этом коммите функция была создана, как можно понять из описания изменений коммита, показанных командой git log.

Отмена определенного коммита

Предположим, что вы нашли коммит, в котором были внесены изменения, приведшие к неработоспособности приложения. Для того, чтобы отменить этот коммит, можно использовать команду git revert:

```
% git revert f5f429c
Removing test_main.py
[main 9ee46ae] Revert "added test_main.py"
 1 file changed, 29 deletions(-)
 delete mode 100644 test_main.py
```

Команда в примере отменяет коммит f5f429c, в котором был добавлен файл с тестами для приложения test_main.py (в реальности тесты вряд ли могут нарушить работоспособность приложения, так что отменять добавление тестов приходится очень редко).

На самом деле удалить коммит из истории изменений в репозитории Git нельзя. Поэтому команда git revert создает еще один коммит, который выполняет противоположные действия для коммита, который требуется отменить. В примере это коммит 9ee46ae в ветке main.

Отмена последнего коммита в истории изменений

В Git есть возможность удалить один или несколько последних коммитов из истории изменений с помощью команды git reset:

```
% git reset HEAD~
```

Ключевое слово HEAD~ говорит о том, что удаляется один коммит от последнего коммита. Если нужно удалить два коммита, то можно использовать HEAD~2.

Отмена незафиксированных изменений

Незафиксированные изменения в файле можно отменить с помощью команды git restore:

```
% git restore README.md
```

Эта команда восстановит файл README.md из последней зафиксированной в коммите версии. В результате все изменения, внесенные в локальную версию и не зафиксированные, будут потеряны.

Итоги

В Git большую часть изменений, которые привели к неработоспособности приложения, можно отменить.

Для поиска требуемых изменений можно использовать команды:

- git grep – ищет строку в файле в репозитории или в истории.
- grep log – ищет, когда были выполнены те или иные изменения.

Отменить изменения можно следующими командами:

- git revert – отменяет коммит в любом месте истории изменений.
- git reset – удаляет один или несколько коммитов в конце истории изменений.
- git restore – отменяет незафиксированные изменения.

Тест

Какая команда Git отменяет коммит с заданным номером в середине истории изменений:

1. git rollback
2. git restore
3. **git revert**
4. git reset

Какая команда Git позволяет найти коммит, в котором внесены изменения в функцию predict в файлу main.py:

1. git grep main.py --log
2. **git log -L :predict:main.py --oneline**
3. git log -S predict --oneline
4. git search :predict:main.py

Модуль № 2. Юнит № 5. Рекомендации по хорошему стилю работы с репозиторием git

В небольших проектах, в которых участвует несколько человек, Git можно использовать просто как технический инструмент для работы над кодом. Но в крупных проектах с большим количеством участников, часто распределенных по разным территориям и часовым поясам, Git становится средством коммуникации. Когда у вас нет возможности быстро подойти лично к другому разработчику в команде и объяснить, что вы хотите сделать в своем коде, эффективное использование Git для коммуникация становится очень важным практическим навыком.

Сообщения коммитов

В этом модуле мы рассмотрели много возможностей Git по работе с историей изменений, поиску и отмене нежелательных действий. Большая часть инструментов Git, предназначенных для этих целей, показывают историю изменений в виде коммитов с сообщениями коммитов (commit message). Давайте еще раз посмотрим на пример графа истории изменений:

```
% git log --oneline --graph --all
*   981050b (HEAD -> main) Merge branch 'testing'
|\
| * d1f79f0 (testing) Added new_testing.py
* | 422d559 Added new_main.py
|/
* f5f429c (origin/main, origin/HEAD) added test_main.py
* 44dd632 Create main.py
* 14e0c34 Create Procfile
* e8e0eec Create requirements.txt
```

В каждой строке вывода команды git log показывается идентификатор коммита и сообщение коммита. Поэтому очень важно писать сообщения коммитов таким образом, чтобы другие разработчики могли быстро и просто понять, что именно делает данный коммит.
Рекомендации по составлению сообщений коммитов есть в книге Pro Git (https://git-scm.com/book/ru/v2/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9-Git-%D0%A3%D1%87%D0%B0%D1%81%D1%82%D0%B8%D0%B5-%D0%B2-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B5), а также в различных статьях (https://cbea.ms/git-commit/, http://who-t.blogspot.com/2009/12/on-commit-messages.html). Здесь мы приведем базовые рекомендации, а с остальными можете познакомиться самостоятельно.

1. Используйте повелительное наклонение, которое выражает распоряжение или просьбу. Несколько примеров:

Update README.md
Merge branch 'testing'
Add main_test.py

Именно такой стиль при автоматическом создании сообщений коммитов используют системы GitHub, GitLab и аналогичные.

Иногда сообщения коммитов пишут в прошедшем времени ('Added main_test.py' вместо 'Add main_test.py') или в сообщение включают простое описание изменений без глагола ('New API method'). Там можно делать в небольших командах, но в крупных проектах желательно использовать стиль сообщений коммитов, который ожидает увидеть большая часть разработчиков.

Чтобы проверить, написано ли сообщение коммит в рекомендуемом стиле, можно попробовать продолжить с его помощью следующее предложение:

If applied, this commit will …

Примеры для правильных сообщений коммит:

If applied, this commit will Update README.md.
If applied, this commit will Merge branch 'testing'.
If applied, this commit will Add main_test.py.

Если же сообщение коммит будет написано в нерекомендованном стиле, то полученное предложение будет неправильным. Например:

If applied, this commit will Added main_test.py.
If applied, this commit will New API method.

2. Сообщение коммит может содержать не только одну строку, но и подробное описание изменений, внесенных в коммит. Такое описание полезно, например, при проведении код ревью, которое мы будем изучать в следующих разделах курса.

Ранее мы рассматривали, как создавать сообщение коммит с использованием опции -m в команде git commit:

```
git commit -m "Add new_main.py"
```

Использование опции -m удобно для создания коротких сообщений о коммитах. Если нужно написать длинное сообщение, то опцию -m можно пропустить:

```
git commit
```

В этом случае после запуска команды откроется окно редактора, в котором можно написать подробное сообщение коммита. В качестве шаблона сообщения можно использовать пример из книги Pro Git:

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body.  The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase will confuse you if you run the
two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
or "Fixes bug."  This convention matches up with commit messages generated
by commands like git merge and git revert.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, followed by a
  single space, with blank lines in between, but conventions vary here


- Use a hanging indent
```

3.Рекомендуется писать сообщения коммитов на английском языке. Это особенно важно делать для проектов, размещающихся в открытых репозиториях, таких как GtiHub. Для небольших проектов, разрабатываемых внутри одной компании, можно использовать русский язык в сообщениях коммитов, если всех программистов команды устраивает такой подход.

Ветки и изменения

Ветки удобно использовать для управления изменениями. Но при коллективной разработке это нужно делать с осторожностью. Не рекомендуется удалять или изменять ветки, которые уже направлены в удаленный репозиторий, доступный другим участникам команды. Например, если вы переименуете ветку в удаленном репозитории, то она переименуется у всех остальных участников, которые могут этого не ожидать. Кроме того, изменения веток могут привести к проблемам в работе инструментов CI/CD, которые настроены на работу с ветками с определенными именами. Особенно это касается основной ветки main: ее удаление или переименование может привести к большим проблемам как у разработчиков, так и у инструментов CI/CD.

Удалять с помощью команды git reset рекомендуется только те коммиты, которые не были отправлены на удаленный репозиторий. В противном случае удаление этих коммитов может оказаться неожиданным для других разработчиков, которые уже получили их из репозитория.

В целом рекомендации по веткам и коммитам такие: потенциально деструктивные действия (удаление, переименование и т.п.) рекомендуется выполнять только в локальной копии репозитория. Если ветка или коммит стали доступны другим разработчикам, изменять их крайне нежелательно.

Модель взаимодействия "Fork and pull"

В крупных проектах редко используется модель разработки, при которой все изменения вносятся в единый репозиторий. В случае большого количества разработчиков и большого числа ежедневных изменений, организовать работу в таком режиме затруднительно. Вместо этого используются более сложные модели, при которых не применяется единый репозиторий. Одной из таких моделей является "Fork and pull" (копирование и слияние).

Модель работы "Fork and pull" состоит из следующих шагов:

1. Создание копии репозитория интересующего проекта (fork).
2. Создание ветки для предлагаемых изменений в собственной копии репозитория. Это позволяет заниматься разработкой без необходимости получать права доступа к репозиторию проекта и другой дополнительной координации.

3. После завершения разработки предлагаемых изменений можно отправить запрос на объединение (pull request) в оригинальный репозиторий проекта.
4. Pull request рассматривается администраторами проекта, а также другими разработчиками. Если доступ к репозиторию открытый, то рассматривать и комментировать pull request могут все желающие.
5. В случае одобрения pull request его изменения вливаются в какую-то ветку кода в оригинальном репозитории.

Модель "Fork and pull" особо популярна для проектов с открытыми исходными кодами, которые размещаются на общедоступных платформах, таких как GitHub или GitLab. Однако и для крупных закрытых проектов эту модель применяют все чаще и чаще.

Итоги

В этом модуле мы рассмотрели инструменты продвинутой командной разработки в Git:

- Организацию изменений в коде в репозитории с помощью веток (branches).
- Объединение веток с помощью слияния или перебазирования.
- Поиск интересующих нас изменений в истории коммитов.
- Отмена изменений, приводящих к неработоспособности приложения.

Также мы рассмотрели рекомендации по хорошему стилю работы с Git в крупных проектах, когда Git становится инструментов коммуникации:

- Рекомендации по написанию сообщений коммитов таким образом, чтобы другие разработчики проекта могли быстро понять, что делает этот коммит.
- Рекомендации по работе с ветками и отмене изменений.
- Использование модели совместной работы "Fork and pull" для крупных проектов.

Практическое задание

Студенты начинают выполнять задание под руководством преподавателя на асинхронном практическом занятии. Завершают выполнение самостоятельно.

Цель практического задания: научиться создавать pull request для проектов других разработчиков.

1. Создайте копию (fork) репозитория на GitHub, содержащий пример проекта прошлого семестра курса программной инженерии – https://github.com/sozykin/ml_fastapi_tests
2. Подумайте, как можно улучшить приложение. Возможные варианты:
    - Разработка нового метода API.
    - Создание нового теста.
    - Создание документации (на приложение, на методы API и т.п.).
    - Исправление стиля кода.
3. В своей копии создайте новую ветку для реализации придуманных вами улучшений.
4. Реализуйте предложенные улучшения в новой ветке.
5. Создайте pull request для включения ваших изменений в основной репозиторий проекта.
6. В случае необходимости ответьте на вопросы преподавателя по pull request в интерфейсе GitHub.
7. Если необходимо, то измените код pull request, чтобы его можно было объединить с кодом в основном репозитории.
8. Убедитесь, что преподаватель слил ваш pull request с основной версией кода в репозитории.

Студенты отправляют решение через интерфейс GitHub. Решение проверяет преподаватель.
Решение считается успешным, если изменения из pull request студента включены в основную версию кода. В противном случае решение неуспешно.

Список источников

- Scott Chacon, Ben Straub. Pro Git – https://git-scm.com/book/ru/v2
- How to Write a Git Commit Message – https://cbea.ms/git-commit/

Модуль № 3
Название: Методика Continuous Delivery

Образовательный результат:

- Студент может перечислить особенности методики Continuous Delivery.
- Студент может назвать способы совместного использования Continuous Integration и Continuous Delivery (CI/CD).
- Студент умеет настраивать CI/CD на GitHub и Heroku.

В этом модуле:

В прошлом семестре мы рассматривали методику повышения эффективности разработки и качества работы приложения Continuous Integration, которая заключается в том, что разработка выполняется небольшими порциями, изменения постоянно записываются (коммитятся) в репозиторий и при каждом изменении запускаются тесты (также может запускаться сборка приложений, что нужно в некоторых языках программирования, таких как C++ и Java).

В этом модуле мы рассмотрим дополняющую методику, которая называется Continuous Delivery, что переводится как непрерывная доставка. Методика Continuous Delivery заключается в том, что приложение должно быть постоянно готово к передаче в практическое использование, а развертывание приложения для продуктивной эксплуатации выполняется автоматически.

Методики Continuous Integration и Continuous Delivery часто используются совместно и обозначаются Continuous Integration/Continuous Delivery, сокращенно CI/CD. В этом случае при выполнении коммита в репозиторий выполняется автоматический запуск тестов, сборка приложения (при необходимости) и автоматическое развертывание приложения. Развернутое приложение, в случае успешной работы, может автоматически быть переведено в практическое использование.

Модуль № 3. Юнит № 1. Методика Continuous Delivery

В прошлом семестре вы научились автоматически развертывать приложение на облачной платформе Heroku с помощью инструментов GitHub Actions. Причем, как вы узнали, есть возможность развертывать приложение только в случае успешного прохождения тестов (в более общем виде, успешного завершения пайплайна CI).

Однако многие из вас успели на практике убедиться, что успешное прохождение тестов не является гарантией работоспособности приложения. Не все ошибки могут быть определены тестами. Особенно это касается ошибок работы моделей машинного обучения. Ведь модель машинного обучения всегда выдает ответ, и очень сложно разработать тест, который сможет отличить правильный ответ модели машинного обучения от неправильного.

Кроме того, автоматическое развертывание приложения для продуктивной эксплуатации может быть плохой идеей, даже если приложение полностью работоспособно. Ведь приложение может делать не то, что нужно пользователю, или не таким образом, как это нужно. Или все может работать правильно, но интерфейс взаимодействия с пользователем очень не удобный.

Методика Continuous Delivery была разработана для устранения описанных выше проблем, связанных с автоматическим развертыванием приложения. В Continuous Delivery определяется несколько стадий работы над приложением. В различных подходах к Continuous Delivery и реализующих их инструментах стадии могут незначительно отличаться друг от друга. Приведем стадии, которые используются чаще всего:

1. **Development** – стадия разработки приложения. На этом этапе производится разработка приложения.
2. **Review** – стадия проверки новых разработок приложения. Используется, например, перед тем, как выполнить слияние веток в репозитории. Или при слиянии Pull Request.
3. **Testing –** стадия тестирования приложения. На этом оценивается работоспособность приложения в целом. Например, выполняется интеграционное тестирование, при котором проверяется, как приложение взаимодействует с базой данных, сторонними микросервисами и т.п. Также может проводится тестирование производительности, при котором оценивается, как приложение ведет себя под нагрузкой. Тестирование на этой стадии выполняется разработчиками и техническими специалистами.
4. **Staging** – еще одна стадия, на которой выполняется проверка работоспособности приложения. На этой стадии к тестированию привлекаются аналитики, менеджеры продуктов, специалисты по взаимодействию с пользователями. Выполняется оценка соответствия работы приложения требованиям бизнеса и ожиданиям пользователя.

5. **Production** – продуктивное использование приложения пользователями.

Для реализации таких стадий создается несколько окружений (environment) для работы приложения. Такие окружения включают набор серверов (или виртуальных машин в облаке), сетевого оборудования и других необходимых компонентов. Например, отдельные серверы могут использоваться для тестового окружения, для Staging и для продуктива. На таких серверах могут работать разные версии приложения. Поэтому остановка приложения на тестовом сервере из-за бага в коде новой версии не приводит к проблемам на продуктивном сервере и не сказывается на работе пользователей.

Разделение развертывания приложения на несколько этапов обеспечивает повышение качества его работы у пользователей, ведь перед этим приложение проходит несколько этапов проверки. С другой стороны, каждый из таких этапов проверки увеличивает время доставки приложения пользователям.

Методика Continuous Delivery направлена на автоматизацию развертывания приложения с учетом нескольких стадий его разработки. Как правило, в инструментах Continuous Delivery создаются пайплайны для автоматизации развертывания приложения в разных окружениях (testing, staging, production). Как именно это делается, зависит от конкретного инструмента. В следующем разделе мы рассмотрим возможности Continuous Delivery в GitHub Actions и Heroku. Другие инструменты работают аналогичным образом.

Итоги:

- Continuous Delivery – методика автоматизации развертывания приложения для повышения скорости доставки приложения пользователям и качества работы приложения.
- Continuous Delivery предполагает, что приложение проходит несколько этапов от разработки до развертывания. Чаще всего используются следующие этапы: development, review, testing, staging, production.
- Инструменты Continuous Delivery позволяют создавать пайплайны для автоматизации развертывания приложения на разных стадиях.

Модуль № 3. Юнит № 2. Настройка Continuous Delivery на платформе Heroku

В предыдущем семестре мы использовали платформу Heroku для автоматического развертывания приложения из репозитория на GitHub (можно дать ссылку на модуль 4 курса Программная инженерия предыдущего семестра). Однако возможности этой платформы не ограничиваются автоматическим развертыванием, Heroku позволяет создавать полноценную инфраструктуру для Continuous Delivery с использованием пайплайнов.

Пайплайны в Heroku представляют собой набор связанных приложений, использующих один и тот же исходный код (из одного репозитория). В пайплайне можно создать несколько стадий в соответствии с методикой Continuous Delivery, и перемещать приложения между стадиями.

Для создания пайплайна нужно зайти в консоль управления Heroku, в правой верхней части консоли нажать “New” и в появившемся меню выбрать “Create new pipeline” (рис.1).

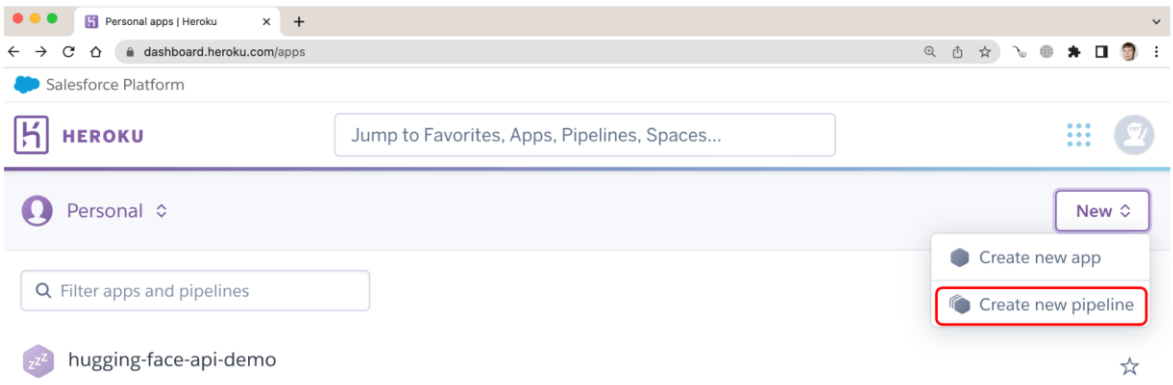


Рис.1. Создание пайплайна в консоли управления Heroku

После этого откроется окно создания пайплайна, показанное на рис. 2.

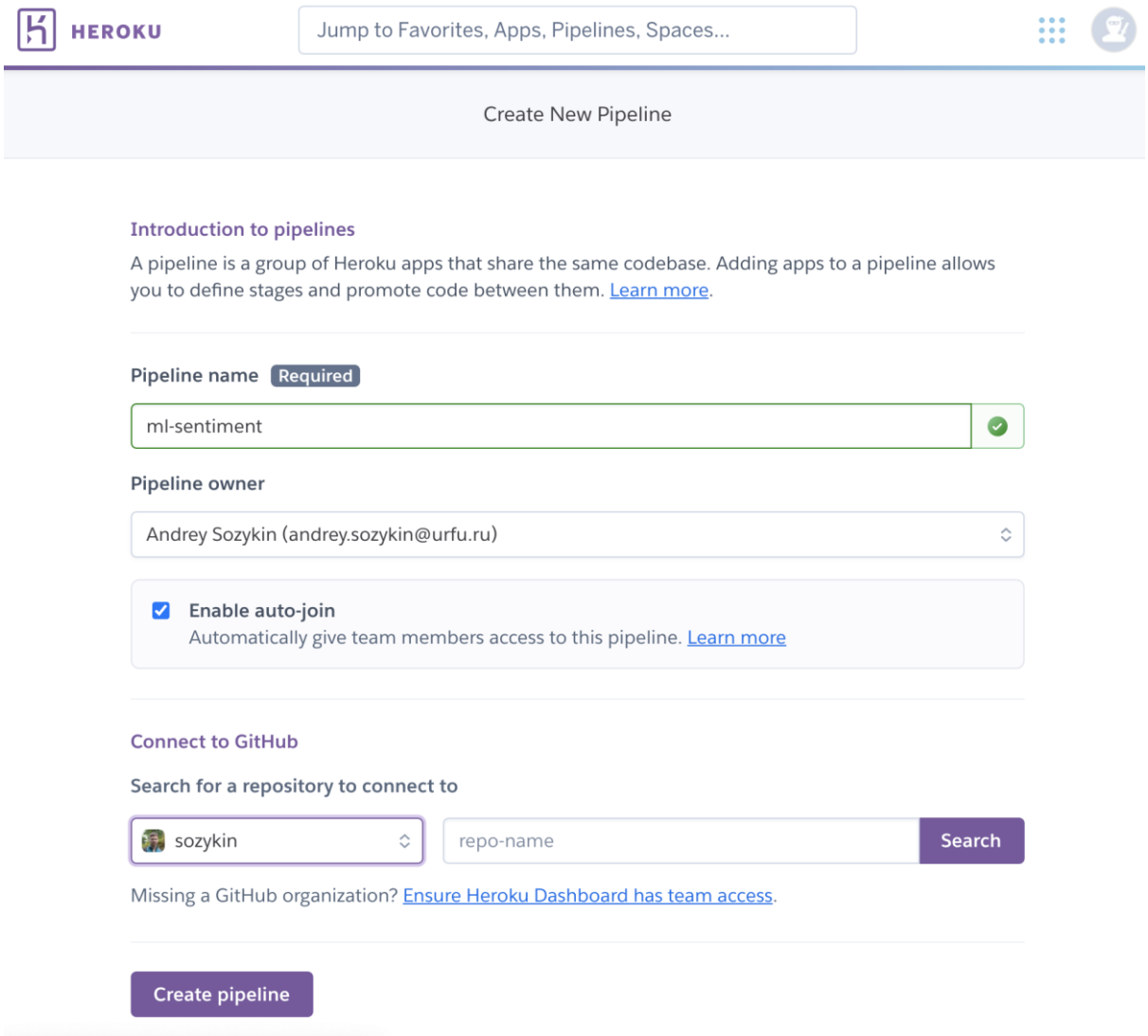


Рис. 2. Окно создания пайплайна в Heroku

В окне создания пайплайна нужно выполнить три действия:

1. Заполнить имя пайплайна (Pipeline name). Имя должно быть уникальным. Если имя не уникальное, то Heroku выдаст сообщение об этом рядом с окном ввода.
2. Выбрать владельца пайплайна (Pipeline owner). Как правило, это ваш пользователь на Heroku.
3. Подключить репозиторий GitHub, из которого будет выполняться развертывание приложения в пайплайне. Подключение репозитория выполняется таким же образом, как подключение к обычному приложению (можно дать ссылку на Модуль 4, Юнит 3 предыдущего семестра курса Программной инженерии).

После подключения репозитория (рис. 3) нужно нажать кнопку “Create Pipeline” для создания пайплайна.

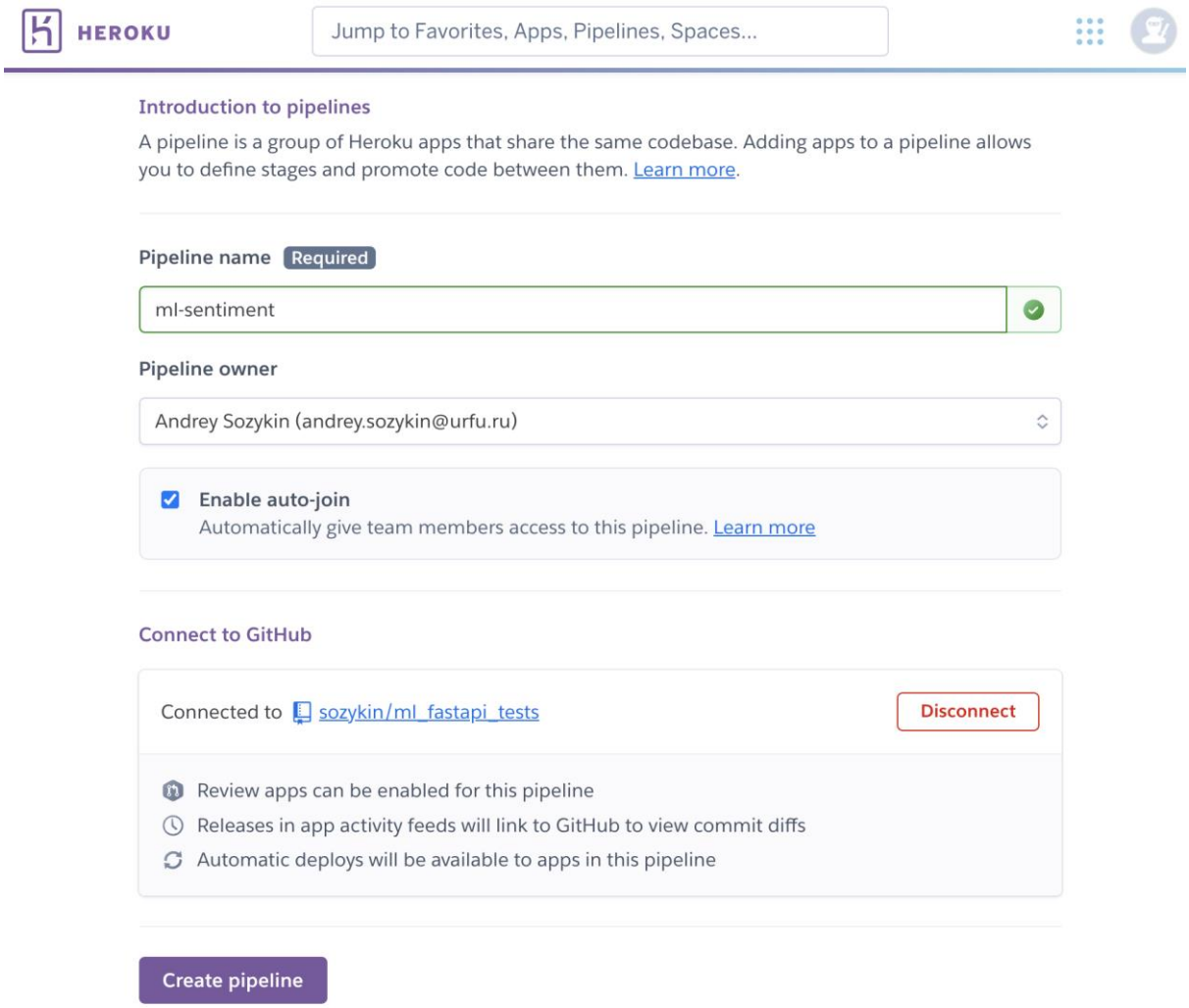


Рис. 3. Окно создания пайплайна на Heroku с подключенным репозиторием приложения на GitHub

После создания пайплайн в Heroku включает три стадии: Review, Staging и Production (рис. 4).

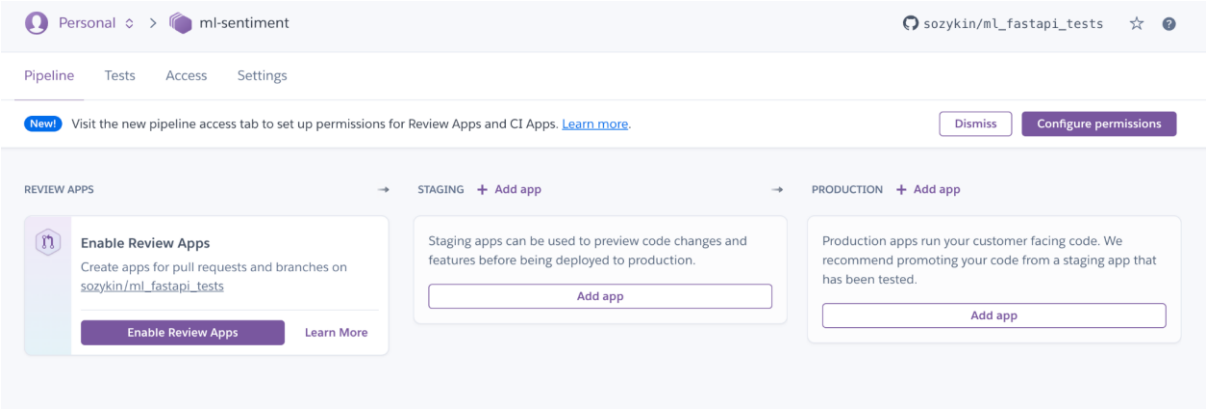


Рис. 4. Стадии пайплайна в Heroku

Стадия Review предназначена для ревью изменений в приложении, предлагаемых в Pull requests к репозиторию на GitHub. Мы подробно рассмотрим, как она работает, позднее. А сейчас начнем со стадии Staging.

Первое, что нам нужно сделать – это добавить приложение на стадию Staging. Для этого нажимаем на кнопку “Add app” и затем, так как у нас нет еще ни одного приложения, на кнопку “Create new app…” (рис. 5).

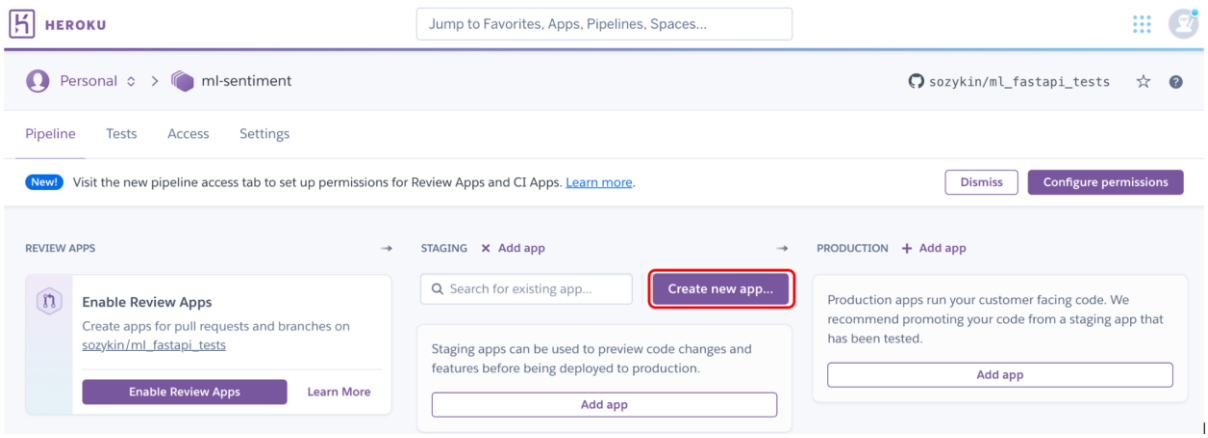


Рис. 5. Создание нового приложения в пайплайне на стадии Staging

После этого в правой части консоли управления откроется окно создания приложения. Нужно ввести название приложения, выбрать регион дата центра, в котором будет развернуто приложение, и нажать на кнопку “Create app”.

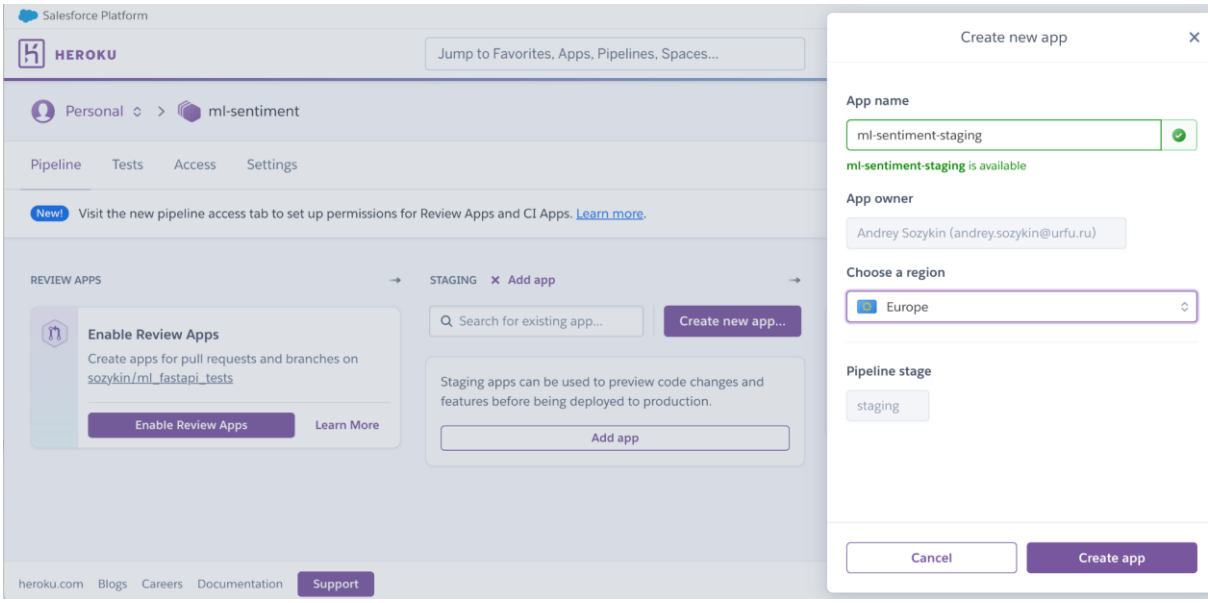


Рис. 6. Окно создания нового приложения в пайплайне

В результате новое приложение будет создано, подключено к репозиторию пайплайна и добавлено на стадию Staging пайплайна.

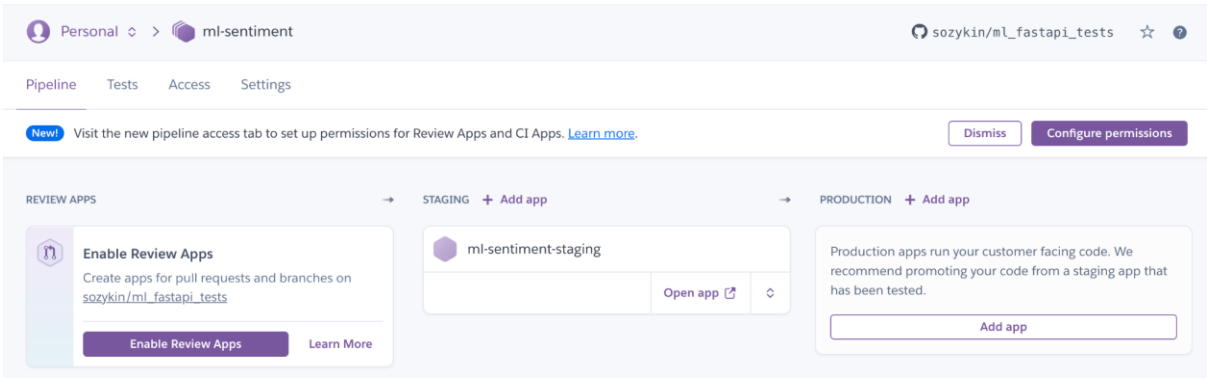


Рис. 7. Новое приложение добавлено на стадию Staging пайплайна

После создания с новым приложением в пайплайне можно работать как с обычным приложением на Heroku. Чтобы развернуть приложение из репозитория на GitHub, нужно зайти в раздел Deploy на странице приложения и нажать на кнопку “Deploy Branch” в разделе “Manual Deploy” (рис. 8).

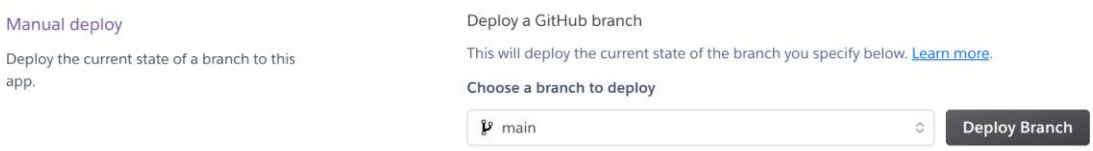


Рис. 8. Ручное развертывание приложения из репозитория на GitHub

В случае успешного развертывания приложения информация об этом появится в пайплайне, также будет доступна кнопка “Open app”, нажав на которую можно перейти в приложение (рис. 9).

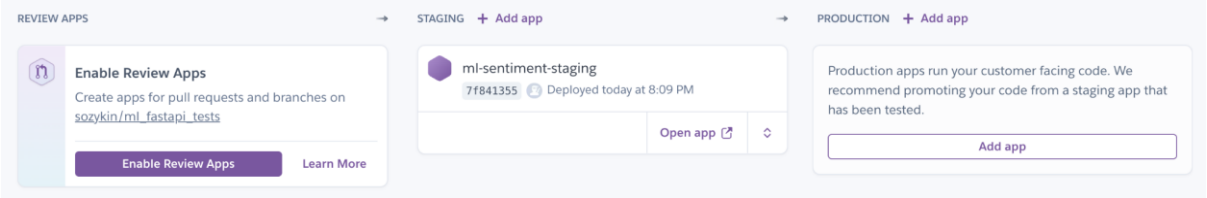


Рис. 9. Развернутое приложение в пайплайне

Аналогичным образом можно создать новое приложение для стадии Production (рис. 10).

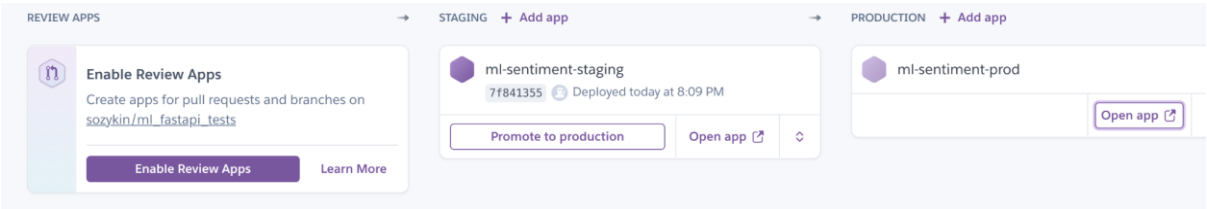


Рис. 10. Пайплайн с приложениями на стадиях Staging и Production

Приложение на стадии Staging можно перевести на стадию Production, нажав на кнопку “Promote to production” рядом с приложением. Откроется окно переноса приложения (рис. 11).

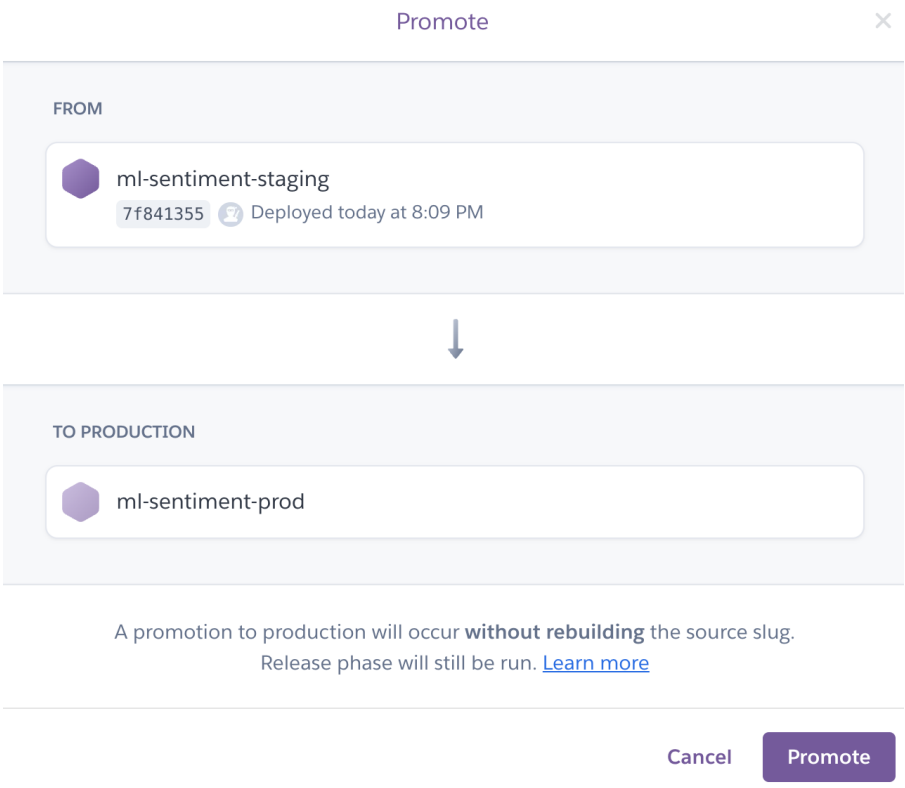


Рис. 11. Окно переноса приложения со стадии Staging на стадию Production

В окне переноса можно выбрать, какое приложение со стадии Staging вы хотите перенести на стадию Production, и в какое приложение хотите его поместить. После выбора нужных приложений нажмите на кнопку “Promote”.

При переносе приложения на Heroku используется тот же самый slug, что и для приложения на стадии Staging. Он разворачивается в отдельном контейнере (Dyno в терминологии Heroku), предназначенном для стадии Production. Из-за использования готового образа развертывание приложения в Production выполняется быстрее, чем на стадии Staging.

После переноса приложения в Production пайплайн содержит два независимых друг от друга контейнера, один из которых выполняет приложение на стадии Staging, а другой на стадии Production. Доступ к этим приложениям выполняется по разным ссылкам. Таким образом, после того, как выполнен перенос приложения в Production, можно заниматься развертыванием новой версии приложения на стадии Staging. Версия приложения в Production при этом не изменится, пока вы явно не дадите команду на перенос.

Итоги

- На облачной платформе Heroku для реализации методики Continuous Delivery используются пайплайны.
- Пайплайны в Heroku включают несколько стадий работы одного приложения.
- Приложения, работающие на различных стадиях пайплайна, независимы друг от друга. Изменения в приложении на одной стадии не влияет на приложения на других стадиях.
- Перенос приложения с одной стадии на другую выполняется автоматически в консоли Heroku.

Модуль № 3. Юнит № 3. Совместное использование Continuous Integration и Continuous Delivery (CI/CD)

Continuous Integration/Continuous Delivery

Наибольшие возможности дает совместное использование методик Continuous Integration и Continuous Delivery. Общая методика называется

называется Continuous Integration/Continuous Delivery или сокращенно CI/CD. При этом сначала выполняются процессы Continuous Integration для тестирования и сборки приложения, а затем Continuous Delivery для его развертывания.

Рассмотрим пример реализации процесса CI/CD:

1. Разработчик реализует новую возможность в приложении и сохраняет ее в репозиторий (выполняет коммит).
2. В репозитории стартует процесс CI, включая запуск тестов, линтера, и других инструментов.
3. В случае успешного завершения процесса CI запускается процесс CD, реализующий автоматическое развертывание приложения из репозитория в окружения стадий Testing или Staging.
4. Развернутое приложение тестируется в окружения Testing или Staging.
5. В случае успешного тестирования приложение переносится в окружение стадии Production.

Настройка CI/CD в GitHub Actions и на платформе Heroku

Инструменты GitHub Actions и Heroku можно использовать для того, чтобы реализовать методику CI/CD. Ранее мы уже рассматривали все отдельные шаги, которые необходимо выполнить. В этом разделе мы рассмотрим пример процесса в целом.

1. **Создание репозитория с кодом приложения на GitHub.**
2. **Настройка CI в GitHub Actions.** Как мы рассматривали в предыдущем семестре (дать ссылку на Модуль 5, Юнит 3 первого семестра курса Программной инженерии), для реализации CI для приложения Python в репозитории GitHub нужно создать файл .github/workflows/python-app.yml со следующим содержанием:

```
# This workflow will install Python dependencies, run tests and lint with a single version of Python
# For more information see: https://help.github.com/actions/language-and-framework-guides/using-python-with-github-actions

name: Python application
```

```
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Python 3.10
      uses: actions/setup-python@v2
      with:
        python-version: "3.10"
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install flake8 pytest
        if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
    - name: Lint with flake8
      run: |
        # stop the build if there are Python syntax errors or undefined names
        flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
        # exit-zero treats all errors as warnings. The GitHub editor is 127 chars
wide
        flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 -
-statistics
    - name: Test with pytest
      run: |
        pytest
```

3. **Создание пайплайна на Heroku.** При создании пайплайна к нему необходимо подключить репозиторий, созданный на первом шаге.
4. **Создание приложений на стадиях Staging и Production.** Как создавать приложения мы рассмотрели в предыдущем разделе.

5. **Настройка автоматического развертывания приложения на стадию Staging из репозитория на GitHub.** Для этого на закладке “Deploy” приложения стадии Staging нужно перейти к разделу “Automatic Deploys”, в нем выбрать ветку, из которой будет выполняться развертывание приложения, поставить галочку напротив пункта “Wait for CI to pass before deploy” и нажать кнопку “Enable automatic deploy”.

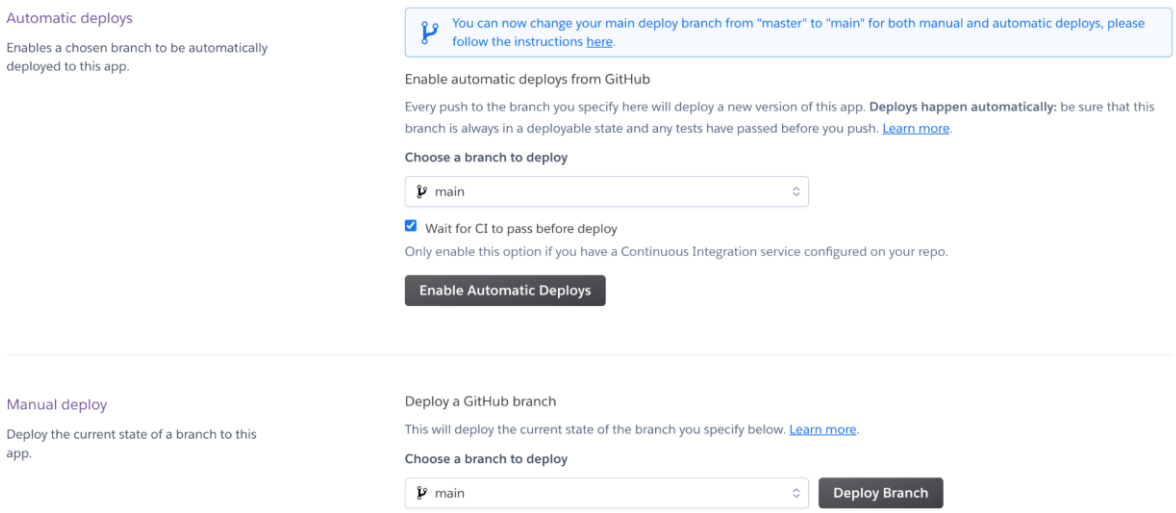


Рис. 12. Настройка автоматического развертывания приложения из репозитория на GitHub после завершения процесса CI

После успешных настроек CI/CD на GitHub и Heroku, выполнение коммит в репозиторий GitHub запустит процесс CI в GitHub Actions, в котором работают тесты на основе pytest и линтер flake8. В случае успешного выполнения тестов и проверок линтера, приложение из репозитория будет автоматически развернуто на Heroku в окружение Staging. Работу приложения можно проверить и в случае успеха перенести приложения из окружения Staging в окружение Production с помощью консоли управления Heroku.

Динамические окружения для стадии Review

Ранее мы рассматривали, как работать в пайплайне на облачной платформе Heroku с окружениями Staging и Production. Для приложений, работающих на этих стадиях, создаются постоянно работающие контейнеры. При этом количество приложений в окружениях Staging и Production ограничено: как

правило используется по одному приложению на каждой стадии, или по нескольку приложения, отличающихся параметрами.

Стадия Review в Continuous Delivery работает по другому. Эта стадия предназначена для оценки изменений в приложении (review). Как правило, изменения реализуются в отдельных ветках репозитория и для их применения создаются pull request. Перед тем, как принять решение, безопасно ли объединять код в pull request с основным кодом приложения, выполняется проверка на стадии Review. При этом в крупном проекте, над которым работает большое количество разработчиков, на рассмотрении может одновременно находится несколько pull request. Поэтому для стадии Review невозможно создать постоянно действующее окружение, как для стадий Staging и Production.

На платформе Heroku для стадии Review реализованы динамические окружения. Как только в репозитории GitHub, к которому подключен пайплайн, создается pull request, Heroku создает новое приложение, в котором развертывается код из ветки pull request. Приложение работает определенное время, по умолчанию 5 дней, за которые нужно проверить его работоспособность. После этого приложение удаляется. Также приложение удаляется после закрытия pull request.

Чтобы включить работу с динамическими приложениями на стадии Review, нужно нажать на кнопку “Enable Review Apps” в пайплайне (рис. 13).

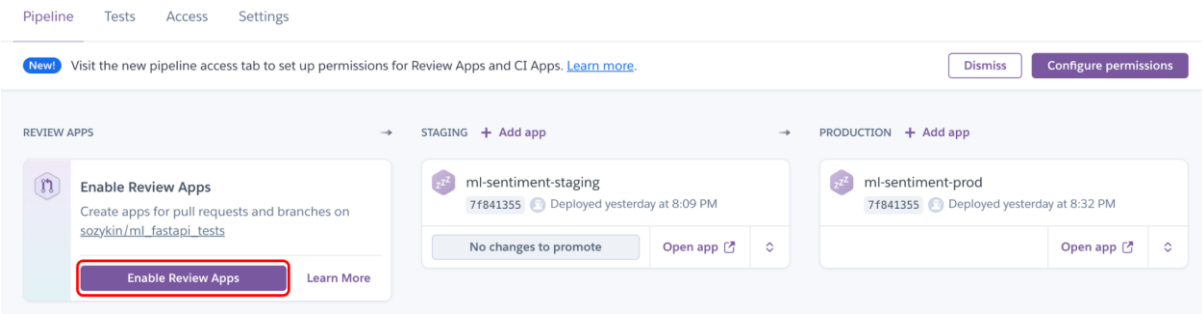


Рис. 13. Включение динамических приложений для стадии Review

После нажатия кнопки “Enable Review Apps” в правой части экрана откроется панель с настройками приложения стадии Review (рис 14). В ней нужно поставить следующие галочки:

- “Create new review apps for new pull requests automatically”
- “Wait for CI to pass”

- “Destroy stale review apps automatically”

Также в панели нужно выбрать регион для приложения и нажать на кнопку “Enable Review Apps” (рис 14).

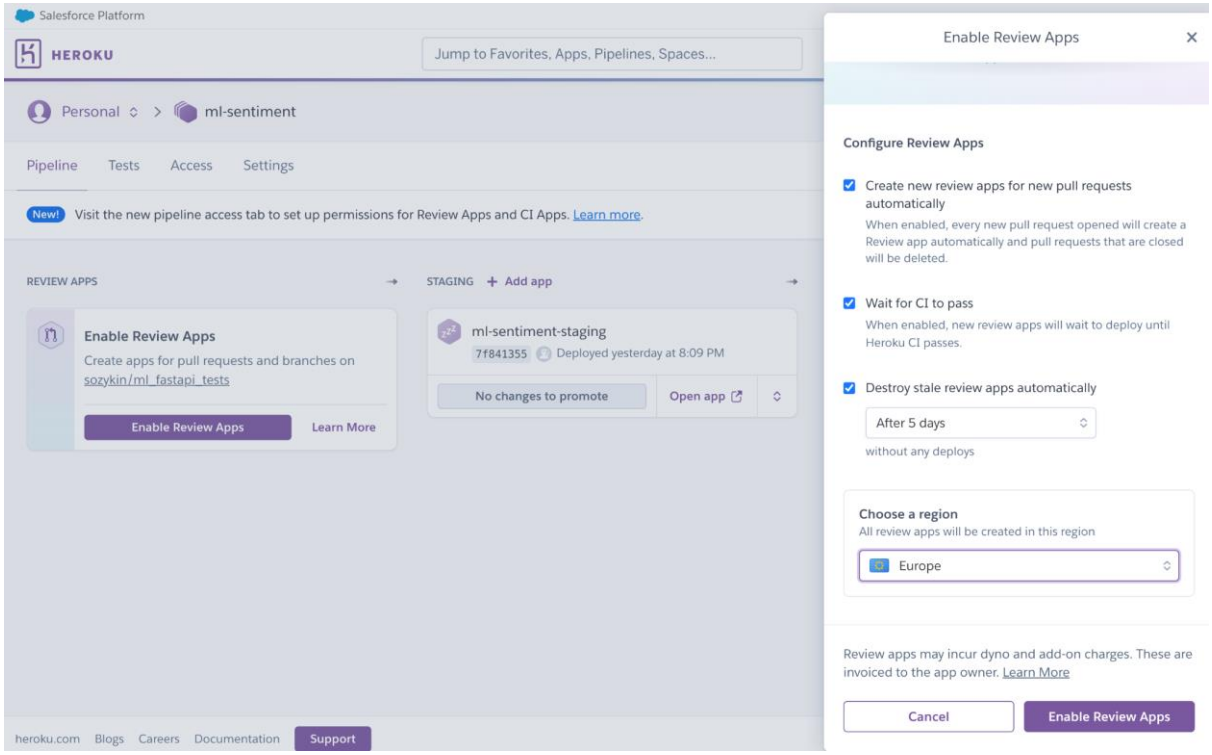


Рис. 14. Настройка динамических приложений для стадии Review на Heroku

В результате динамические приложения для стадии Review в пайплайне будут включены. Но так как пока в репозитории нет pull request, то и динамических приложений на этой стадии пайплайна нет (рис. 15).

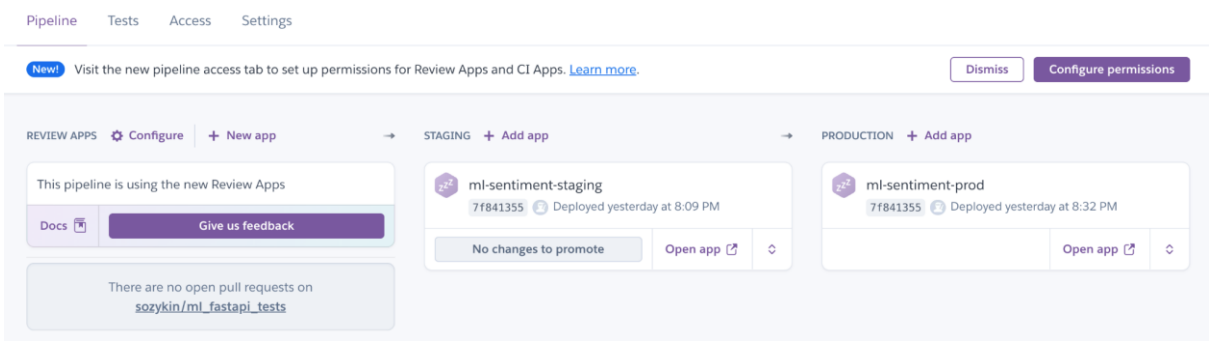


Рис. 15. Пайплайн с включенными динамическими приложениями для стадии Review

Если в репозитории GitHub создать новую ветку и pull request для объединения этой ветки с основной версией кода, то на стадии Review

пайплайна появится новое приложение, в которое будет автоматически развернут код из новой ветки (рис. 16).

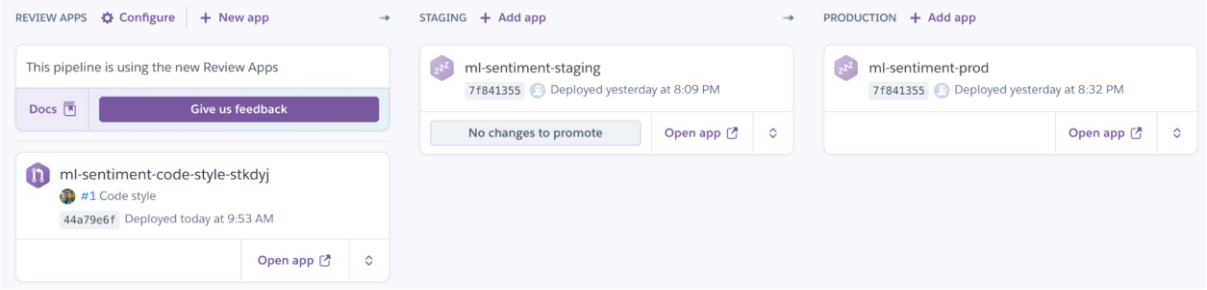


Рис. 16. Пайплайн с развернутым динамическим приложением для стадии Review из pull request в репозитории GitHub

Если в репозитории будет несколько pull request, то на стадии Review пайплайна будет создано отдельное приложение для каждой ветки, планируемой к объединению.

Итоги

- Continuous Delivery – методика автоматизации развертывания приложения для повышения скорости доставки приложения пользователям и качества работы приложения.
- Согласно методике Continuous Delivery, приложение проходит несколько этапов от разработки до развертывания. Наиболее популярные этапы: development, review, testing, staging, production.
- Наибольшие возможности дает совместное использование методик Continuous Integration и Continuous Delivery: Continuous Integration/Continuous Delivery или сокращенно CI/CD. При этом сначала выполняются процессы Continuous Integration для тестирования и сборки приложения, а затем Continuous Delivery для его развертывания.
- На облачной платформе Heroku для реализации Continuous Delivery используются пайплайны.
- Возможна интеграция Continuous Delivery в пайплайнах на Heroku с Continuous Integration на GitHub Actions.
- Для стадий Staging и Production на Heroku создаются постоянно работающие контейнеры.
- Для стадии Review на Heroku используются динамические приложения, которые запускаются автоматически при создании pull request на GitHub.

- Более подробно с особенностями применения CI/CD для автоматизации задач машинного обучения вы узнаете в курсе MLOps.

Итоговый тест

Выстройте в правильной последовательности стадии, которые проходит приложение, в методике Continuous Delivery:

1. Development
2. Review
3. Testing
4. Staging
5. Production

Какие стадии Continuous Delivery создаются в пайплайне на облачной платформе Heroku автоматически?

1. Development
2. **Review**
3. Testing
4. **Staging**
5. **Production**

На какой стадии пайплайна Heroku создаются динамические приложения:

1. Development
2. **Review**
3. Staging
4. Production

Когда создаются динамическое приложения в пайплайне Heroku?

1. При выполнении commit в репозитории на GitHub
2. При создании новой ветки в репозитории на GitHub
3. При нажатии кнопки “Promote” в приложении на стадии Staging
4. **При создании pull request в репозитории на GitHub**

Практическое задание

На практическом вебинаре с преподавателем вы начали выполнять задание по настройке Continuous Delivery на Heroku из репозитория на GitHub. Завершите и приложите ссылку на выполненное задание на платформу.

Напомним, что задание выполняется в командах из 2-4 человек. Рекомендуется выполнять задание с репозиторием проекта, который команда будет сдавать в текущем семестре. Но допускается работать и с репозиторием из предыдущего семестра.

1. Создайте пайплайн на Heroku и подключите его к репозиторию GitHub.
2. Создайте Pull Request для репозитория на GitHub и проверьте, что на Heroku создалась виртуальная машина для запуска приложения из этого Pull Request.
3. Проверьте работоспособность версии приложений из Pull Request на Heroku.
4. Соедините Pull Request с общей версией кода.
5. Проверьте работоспособность приложения на виртуальной машине стадии staging на Heroku.
6. В случае успешной проверки работоспособности разверните код приложения на продуктивной виртуальной машине на Heroku.

Для сдачи задания пришлите три ссылки на платформе:
1. Ссылку на репозиторий GitHub.
2. Ссылку на приложение в окружении Staging.
3. Ссылку на приложение в окружении Production.

Список источников

- Heroku Pipelines – https://devcenter.heroku.com/articles/pipelines
- GitHub Actions – https://github.com/features/actions
- How to make your first pull request on GitHub – https://www.freecodecamp.org/news/how-to-make-your-first-pull-request-on-github-3/

Модуль № 4

Название: Качество кода

Образовательный результат:

- Студент может объяснять важность обеспечение качества кода при разработке программного обеспечения
- Студент может выполнять рефакторинг кода на Python.

В этом модуле:

Эффективность команды, разрабатывающей крупные программные системы, как правило, со временем сокращается (рис. 1). Первый год или два команда работает очень быстро и новые функции системы постоянно выходят в релиз. Но со временем объем кода растет и вносить изменения становится все сложнее. Даже незначительное изменение может привести к большому количеству ошибок в непредсказуемых местах. Поэтому новые функции получается выводить в продуктивное использование все реже и реже. А со временем наступает такой момент, когда изменить что-то в коде практически невозможно. В этом случае команда разработки предлагает остановить развитие текущей системы и реализовать новую версию с нуля. Однако для новой версии картина оказывается похожей: после года-двух быстрой разработки эффективность снова снижается.

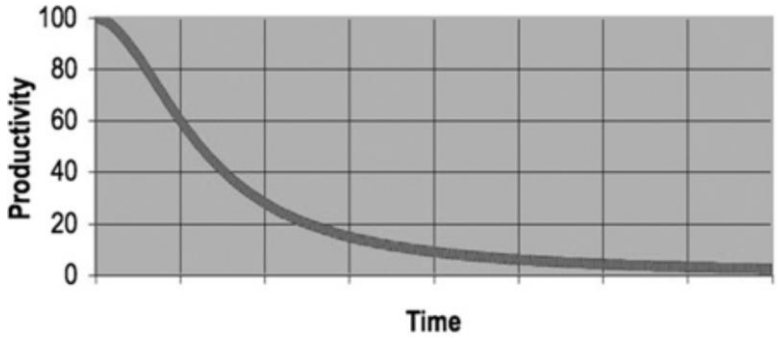


Рис 1. Эффективность команды разработчиков [1]

Причина проблем с продуктивностью в качестве кода, который создает команда. Даже очень плохо написанный код может выполнять то, что нужно пользователям. На начальном этапе это не создает проблем. Но со временем изменить что-то в плохо написанном коде становится очень сложно.

Для решения указанных проблем в программной инженерии были разработаны методы и инструменты для создания качественного кода. Один из основателей данного направления – Роберт Мартин, который написал книгу “Чистый код”. Он предложил использовать в разработке программного обеспечения подход японских производителей автомобилей, которые, в отличие от американских или европейских, концентрировались

на обеспечение работоспособности автомобиля на протяжении всего срока его эксплуатации, а не только на производстве новых продуктов. **Для этого при разработке программного обеспечения необходимо уделять пристальное внимание не только реализации необходимых функций, но и обеспечению возможности поддерживать и изменять код на протяжении длительного времени.**

Роберт Мартин предложил подход к определению, что можно считать чистым кодом, а также описал принципы, техники и инструменты, с помощью которых можно добиться нужной чистоты кода. Один из таких инструментов, стиль оформления кода, мы уже подробно рассмотрели в первом модуле (дать ссылку на первый модуль этого семестра по Стилю кода). В этом модуле мы познакомимся с другими принципами, техниками и инструментами.

Однако однозначного определения, что такое чистый код, до сих пор не существует. Кроме того, подходы к созданию качественного кода в разных языках программирования отличаются. В частности, в Python существует рекомендованный подход к решению некоторых задач, который называется идиоматическим, или **Pythonic**. В этом модуле мы познакомимся с распространенными идиомами Python, которые сильнее всего отличаются от других языков программирования.

На практике всегда создавать только чистый код не получается, т.к. в противном случае не хватит времени для реализации бизнес-требований к приложению. Необходимо соблюдать баланс между реализацией новых функций и обеспечением качества кода. Для этого используется метафора **“технического долга”: сознательного принятия решения об использовании не очень качественного кода для ускорения реализации бизнес требований.** Однако в будущем необходимо запланировать время для того, чтобы вернуть “технический долг” и повысить чистоту кода, чтобы его можно было беспрепятственно поддерживать и развивать в будущем.

Модуль № 4. Юнит № 1. Качество кода. Понятие “чистый код”. Преимущества использования чистого кода.

Сейчас отрасль разработки программного обеспечения сконцентрирована на скорейшем выводе на рынок приложений, которые решают задачи

пользователей. Для этого была разработана методология Agile и созданы несколько фреймворков управления проектами, включая Scrum и Kanban.

Однако подход к разработке программного обеспечения, при котором приоритизируются только новые функции приложения, которые нужно вывести на рынок как можно быстрее, приводит к созданию большого количества кода плохого качества. Такой код сложно поддерживать и расширять, а это означает, что приложение будет работать нестабильно и в него нельзя будет добавлять новые функции. Со временем пользователи откажутся от такого приложения в пользу более стабильного и активно развивающегося.
Таким образом, некачественный код приводит к существенным проблемам, как техническим, так и финансовым.

Чистый код

Роберт Мартин в книге “Чистый код” утверждает, что создание чистого кода – это искусство. Сформулировать однозначное определение, что такое чистый код, невозможно. Однако когда мы смотрим на код, то почти всегда можем определить, чистый он, или нет. Это работает также, как оценка искусства: мы можем отличить хорошую картину от плохой, но сформулировать почему эта картина хороша, а другая плоха очень сложно. Однако способность отличить хорошую картину от плохой не означает, что мы можем написать хорошую картину. Аналогичным образом обстоит ситуация с разработкой качественного кода.

Роберт Мартин провел опрос экспертов в разработке программного обеспечения, что они считают чистым кодом. В качестве основных характеристик можно выделить следующие:

- Код должен быть элегантным, удобным для чтения и понятным.
- Код должен решать одну задачу качественно.
- Код должен быть эффективным и высокопроизводительным.
- Имена в коде должны быть осмысленными.
- Код предоставляет один путь, а не несколько, чтобы решить одну задачу.
- Код не должен содержать повторений.
- Количество сущностей в коде (таких как классы, методы, функции и т.п.) должно быть минимальным.

- Зависимости в коде должны быть минимальными чтобы обеспечить возможность поддержки.
- Ошибки в коде должны обрабатываться полностью.
- Код должен содержать тесты и все тесты должны завершаться успешно.
- Код должен быть пригоден для чтения и расширения не только его автором, но и другими разработчиками в команде.

Наверняка вы обратили внимание, что многие принципы чистого кода включены в Zen of Python. Например:

- Красивое лучше, чем уродливое.
- Простое лучше, чем сложное.
- Читаемость имеет значение.
- Ошибки никогда не должны замалчиваться.
- Если они не замалчиваются явно.
- Должен существовать один и, желательно, только один очевидный способ сделать это.
- Если реализацию сложно объяснить — идея плоха.
- Если реализацию легко объяснить — идея, возможно, хороша.

Таким образом, соблюдение философии Python помогает вам писать чистый код!

Действия по повышению качества кода

Написание чистого кода – это искусство, добиться совершенства в котором достаточно сложно. Мы здесь приведем базовые рекомендации, которые нужно соблюдать, чтобы код был чистым. В дальнейшем вам придется разбираться с каждым направлением более подробно. А также придется выяснять, что является “чистым кодом” для команды, в которой вы работаете.

1. **Используйте осмысленные имена.** В коде мы постоянно используем имена: мы называем переменные, функции, аргументы, классы, пакеты и т.п. Хорошо выбранное имя позволяет быстро понять, для чего нужна переменная или что делает функция.
   Сложно понять, для чего нужна переменная, которая называется одной буквой *x* или словосочетанием *the_list*. А если переменная называется *x_train*, то можно определить, что в этой переменной находится набор

данных для обучения. Функция *train_test_split*, как понятно из названия, разделяет набор данных на две части: для обучения и для тестирования.

Рекомендованные правила выбора имен зависят от языка программирования, который вы используйте. Для языка Python такие рекомендации есть в документе PEP8.

Выбор осмысленных и понятных другим разработчика имен – сложная тема. Изучите рекомендации по выбору имен для вашего языка программирования, рекомендации вашей компании или проекта, в котором вы работаете. Также можно спросить других членов команды о рекомендованных подходах выбору осмысленных имен.

2. **Функции и методы,** которые вы создаете, должны быть небольшими и решать только одну задачу. В идеале код функции или метода должен помещаться на один экран. Тогда можно будет смотреть на код целиком, и в этом случае понять, что код делает, будет проще и быстрее. Сложность функции или метода может оценить линтер flake8 с использованием McCabe complexity checker (https://github.com/PyCQA/mccabe).

   Функция не должна иметь побочных эффектов. Например, функция не должна изменять значения своих аргументов или глобальных переменных. Ожидаемое действие от функции – возврат значения. Именно этого ожидают от функции другие разработчики в вашей команде. Если ваша функция в дополнение к возврату значения изменяет значение глобальной переменной, то другие разработчики могут не догадаться об этом, в результате могут возникнуть ошибки, причину появления которых найти будет очень сложно.
3. **Форматируйте код в соответствии с рекомендованным стилем**. Стиль кода мы подробно рассматривали в первом модуле этого семестра.
4. **Не копируйте код.** Если одинаковые действия нужно выполнять с различными данными или в различных частях системы, то рекомендуется создать функцию или метод, которые будут содержать единственную копию кода. Эту функцию уже нужно вызывать из разных мест системы. В противном случае, когда нужно будет вносить изменения в код, это придется делать во всех копиях одновременно. При этом легко ошибиться и выполнить несогласованные изменения. В результате могут возникать ошибки, причину которых очень сложно найти.

   В дополнение представьте ситуацию, когда копию кода создавали не вы, а другой разработчик. В этом случае вам особенно сложно будет найти причину ошибки, т.к. вы ничего не знаете о наличии этих копий и

необходимости согласованных изменений. Постарайтесь не ставить коллег по команде в такую ситуацию.

5. **Обрабатывайте ошибки при работе программы.** Входные данные могут быть неправильный или использовать неверный формат. Оборудование может выходить из строя. Сетевое соединение может быть потеряно. Ваша программа должна сохранять работоспособность при возникновении подобных ситуаций, а не останавливаться. Кроме того, очень полезно выдавать осмысленные сообщения об ошибках. При этом желательно использовать подходы к обработке ошибок, рекомендуемые для языка программирования, на котором ведется разработка. Рекомендации для Python мы рассмотрим в следующем разделе.
6. **Разрабатывайте тесты.** Именно тесты позволят вам быть уверенными в том, что ваш кода работает успешно. Создание тестов с помощью pytest для программ на Python мы подробно рассматривали в пятом модуле прошлого семестра.

Здесь мы не приводим рекомендаций по разработке чистого объектно-ориентированного кода, т.к. это отдельная большая и сложная тема, которая выходит за границы данного курса. На практике инженер машинного обучения обычно может решать большую часть практических задач без создания сложного объектно-ориентированного кода. Для простых случаев приведенных рекомендаций будет достаточно.

Дисциплины для создания чистого кода

Каким образом можно реализовать рекомендации по созданию чистого кода наиболее эффективным образом? Роберт Мартин в книге “Clean Craftsmanship: Disciplines, Standards, and Ethics” приводит четыре дисциплины, которые можно использовать для этой цели:

1. **Test-Driven Development (разработка через тестирование)**. Подход к разработке программного обеспечения, при котором сначала пишется тест для функции, а затем уже реализуется сама функция. Если бы мы использовали разработку через тестирование в прошлом семестре для создания проекта по определению тональности текста, то на первом этапе мы бы написали тесты для проверки работоспособности системы. Как проверить, что система определения тональности текста работает? Минимальный вариант тестов: передать системе тексты с положительной и отрицательной тональностью и

проверить, что система определит тональность правильно. Тесты могу выглядеть следующим образом:

```
def test_predict_positive():
    response = client.post("/predict/",
        json={"text": "I like machine learning!"}
    )
    json_data = response.json()

    assert response.status_code == 200
    assert json_data['label'] == 'POSITIVE'

def test_predict_negative():
    response = client.post("/predict/",
        json={"text": "I hate machine learning!"}
    )
    json_data = response.json()

    assert response.status_code == 200
    assert json_data['label'] == 'NEGATIVE'
```

В первом тесте вы передаем в систему предложение с положительной эмоциональной окраской и проверяем, что в поле 'label' ответа содержится значение 'POSITIVE'. Во втором тесте мы передаем предложение с отрицательной эмоциональной окраской и ожидаем, что поле 'label' ответа содержится значение 'NEGATIVE'.

Обратите внимание, что при написании тестов мы также используем правила чистого кода. Функции, которые мы создали для тестирования, небольшого размера (каждая помещается на один экран) и делают только одну вещь. Мы не создали одну функцию, которая проверяет работоспособность на предложениях как с положительной, так и с отрицательной эмоциональной окраской, так как это две разные задачи. По названия функций легко понять, что они делают: test_predict_positive тестирует распознавание (predict) предложения с позитивной эмоциональной окраской, а test_predict_negative – с отрицательной.

Теперь, когда у нас готовы тесты, мы можем перейти к написании основной функции predict:

```
class Item(BaseModel):
    text: str


app = FastAPI()
classifier = pipeline("sentiment-analysis")


@app.post("/predict/")
def predict(item: Item):
    return classifier(item.text)[0]
```

Достаточно ли разработанных нами тестов для проверки работоспособности приложения? Что будет, если в функцию predict передать предложение с нейтральной эмоциональной окраской? Что произойдет, если текст вообще не передавать? Или вместо текста будут передано числи? Или список текстов? Для каждого из перечисленных случаев, а также других подобных случаев, которые потенциально могут привести к проблемам, нужно написать отдельный тест. А затем переписать функцию predict таким образом, чтобы все тесты проходили успешно.

Разработка через тестирование является базовой дисциплиной для всех остальных дисциплин. Именно наличие тестов позволяет нам быть уверенными, что в результате наших попыток повысить качество кода мы не нарушим работоспособность приложения.
В настоящее время разработка через тестирование является достаточно спорной дисциплиной и на практике ее используют редко. Однако необходимость разработки тестов никто не ставит под сомнение. Попробуйте в проекте по Программной инженерии этого семестра сначала писать тесты, а потом уже код основного приложения, и посмотреть на результаты. Возможно, вам понравится и вы в дальнейшем будете применять такую практику.

2. **Рефакторинг** – это изменение кода с целью повышения его качества без изменения функциональности и поведения.
   Для успешного проведения рефакторинга обязательно нужны тесты. В противном случае нам сложно понять, сохранилась ли работоспособность системы после изменений, направленных на повышение качества кода. Именно из-за этого многие разработчики боятся улучшать работающий код, т.к. такие улучшения могут привести к изменениям в непредсказуемым местам.
   Очень важно, чтобы изменения, выполняемые в процессе рефакторинга, не затрагивали функциональности кода. Если изменится поведение приложения, то мы не сможем оценить его работоспособность с использованием имеющегося набора тестов.
   Рефакторинг мы подробно рассмотрим в одном из следующих разделов.

3. **Collaborative Programming** – дисциплина разработки программного обеспечения в команде. Может быть реализована в виде различных механик:
   - *Парное программирование* (pair programming) – подход к разработке программного обеспечения, при котором за одним компьютером работает два человека. Один из них пишет код, а другой проверяет на наличие ошибок как в самом коде, так и в стиле, качестве, архитектурных решениях и т.п. В результате количество ошибок в коде снижается и время, необходимое на отладку, сокращается.
     Недостаток парного программирования заключается в том, что для реализации заданного объема работы требуется в два раза больше людей. При этом такие вложения не всегда оправдываются сокращением времени на отладку и исправление ошибок.
     С другой стороны, парное программирование решает проблему распространения знаний: в любой момент времени в команде есть минимум два человека, которые понимают, как устроен любой участок кода. Поэтому если один из них заболеет или уйдет в отпуск, то ничего критичного не случится.
     В последнее время программирование парами выходит на новый уровень, при котором одним из партнеров является искусственный интеллект, как, например, в GitHub Copilot (https://copilot.github.com/). Это позволяет решить проблемы с

неэффективным использованием человеческих ресурсов в проекте.

Расширенный вариант программирования парами – *программирование толпой (mob programming).* В этом случае за одним компьютером находятся не два, а несколько человек.

- *Код-ревью (code review)* – это еще один инструмент организации командной работы над кодом. В этом случае код, в отличие от парного программирования, разрабатывает один человек. Но перед тем, как производится объединение кода в общую ветку (или релиз кода, если ветки не используются), код проверяют другие разработчики в команде. Один или несколько других членов команды должны проверить и одобрить предлагаемый код, после этого выполняется объединение.

  Преимуществом код-ревью перед парным программированием заключается в снижении трудозатрат на разработку основной версии кода: его создает один человек, а не два ли несколько. При этом код-ревью также помогает решить проблему распространения знаний: во время проведения ревью участники команды разбираются в коде друг друга.

  Однако у код-ревью есть существенный недостаток: если при парном программировании оба человека, создающих код, являются его авторами, то при проведении ревью кода теряется его авторство. Поэтому к коду могут относиться менее внимательно и конструктивно. Кроме того, иногда в результате проверки кода замечания формулируются в обидной или даже оскорбительной манере. В результате польза от такого процесса снижается.

  Мы подробно рассмотрим код-ревью в следующем модуле.
- *Мозговой штурм.* Традиционная техника совместной работы может применяться и для решения задач программирования. Команда может проводить мозговые штурмы для обсуждения архитектуры, общего API, технических особенностей реализации кода и других проблем, которые затрагивают работу всех участников.

4. **Простая архитектура.** Рекомендуется выбирать минимальный вариант архитектуры для приложения, который обеспечивает реализацию требований. Такая архитектура должна включать минимальный набор компонентов с минимальными связями между ними.

Часто архитектурные решения усложняют для того, чтобы в будущем была возможность развития: добавления новых функций, масштабирования и т.п. Однако из-за постоянно меняющихся требований к программному обеспечению часто оказывается, что предположения о будущем развитии, сделанные в начале проекта, оказываются неправильными. Поэтому заложенный в архитектуру приложения возможности для масштабирования оказываются не востребованными, но вместо них нужны другие, соответствующие новым требованиям. Именно поэтому не рекомендуется усложнять архитектуру без особой необходимости.

Итоги:

- Качественный код помогает избежать проблем с эффективностью работы команды программистов, создающих крупные программные системы.
- Однозначного определения, что такое чистый код, не существует.
- Первые понятие чистого кода и подходы к нему описал Роберт Мартин в книге “Чистый код”.
- Для создания чистого кода рекомендуется выполнять следующие основные действия: использовать осмысленные имена, создавать небольшие функции, не копировать код, форматировать код в соответствии с рекомендованным стилем, обрабатывать все возможные ошибки и писать тесты.
- Создавать качественный код помогают четыре дисциплины: разработка через тестирование, рефакторинг, командное программирование, простая архитектура.

Модуль № 4. Юнит № 2. Особенности чистого кода на Python

Создатели языка Python при его проектировании специально приняли ряд проектных решений, которые помогают писать на Python именно чистый код. Например, в Python обязательно использовать отступы для выделения блоков кода, из-за чего приходится форматировать код. Философия Python содержит многие принципы чистого кода, которые мы рассматривали в предыдущем разделе, также в Python есть общепринятый стиль кода, оформленный в документе PEP8. Еще одна особенность Python, облегчающая написание чистого кода – это идиомы Python.

Идиомы в Python – это набор рекомендаций по решению тех или иных задач именно таким образом, чтобы полностью использовались возможности Python и при этом писать качественный код. Такой код называется Pythonic. В этом разделе мы рассмотрим наиболее востребованные идиомы Python.

Итераторы

Во многих языках программирования для создания циклов используются счетчики. В Python также можно создавать такие циклы:

```
optimizers = ['sgd', 'adam', 'rmsprop']

for i in range(len(optimizers)):
    print(optimizers[i])
```

Однако в Python идиоматическим является подход к созданию циклов с использованием итераторов:

```
optimizers = ['sgd', 'adam', 'rmsprop']

for optimizer in optimizers:
    print(optimizer)
```

List Comprehension

List Comprehension – это специфический для Python механизм создания списков на основе существующих списков. Предположим, что мы создаем код для системы обработки естественного языка, одним из этапов которого является приведение всех слов к нижнему регистру. Традиционным подходом будет использовать для этой цели цикл:

```
words = ['Тестируем', 'List', 'Comprehensions', 'в', 'Python']
words_lower = []
for word in words:
    words_lower.append(word.lower())
print(words_lower)
```

```
['тестируем', 'list', 'comprehensions', 'в', 'python']
```

Однако List Comprehension предоставляют для этой цели более простой и элегантный механизм:

```
words = ['Тестируем', 'List', 'Comprehensions', 'в', 'Python']
words_lower = [word.lower() for word in words]
print(words_lower)
['тестируем', 'list', 'comprehensions', 'в', 'python']
```

Именно List Comprehension является идиоматическим в Python и позволяет создавать более Pythonic код. Использовать циклы для этой цели не рекомендуется.

В дополнение к List Comprehension в Python также есть Set Comprehension и Dictionary Comprehension для работы со словарями и списками.

Встроенные функции для структур данных

Во многих случаях, для которых в традиционных языках программирования используются циклы, в Python можно применить встроенные функции или методы структур данных. Вот пример цикла для поиска максимального значения в списке:

```
output = [0.1, 0.15, 0.2, 0.55] # Выходные данные модели машинного обучения
max_value = 0
for value in output:
    if value > max_value:
        max_value=value
print(max_value)
```

Вместо этого цикла в Python рекомендуется использовать встроенную функцию max:

```
output = [0.1, 0.15, 0.2, 0.55] # Выходные данные модели машинного обучения
```

```
print(max(output))
```

Номер максимального элемента в списке можно определить с помощью функции index списка:

```
output = [0.1, 0.15, 0.2, 0.55] # Выходные данные модели машинного обучения
print(output.index(max(output)))
```

А если вы работаете не со списками, а с массивами numpy, что обычно бывает при решении задач машинного обучения, то для этой цели можно использовать встроенную в numpy функцию argmax:

```
import numpy as np
output = [0.1, 0.15, 0.2, 0.55] # Выходные данные модели машинного обучения
print(np.argmax(output))
```

Таким образом, в целом в Python использование циклов для обработки элементов структур данных не является предпочтительным действием. На первом этапе попробуйте найти функцию, которую можно применить для реализации вашей задачи, в том числе в популярных библиотеках. И только если таких функций нет, используйте циклы.

Менеджеры контекста

Часто возникает ситуация, когда нужно обязательно выполнить определенные действия, даже если в процессе выполнения программы возникнут какие-то ошибки. Например, когда мы работаем с файлом, его всегда нужно закрывать после обработки. Традиционно для этой цели используют секцию finally в блоке try:

```
f = open(filename)
try:
    process_file(f)
finally:
```

```
    f.close()
```

Однако в Python решить такую задачу можно более элегантно с помощью менеджера контекста, который создается с помощью ключевого слова with (https://peps.python.org/pep-0343/):

```
with open(filename) as f:
    process_file(f)
```

В этом случае менеджер контекста, созданный с помощью оператора with, обеспечит, что файл будет закрыт после завершения блока кода, даже если в процессе обработки произойдет исключение.

Общие исключения

В Python при обработке исключений обычно указывается, какое именно исключение вы хотите перехватить. Однако желаемый тип исключений указывают не всегда, что может привести к проблемам. Например, для открытия файла можно написать такой код:

```
try:
    f = opn(filename)
except:
    sys.exit("Не могу открыть файл!")
```

При выполнении этого кода мы получим сообщение об ошибке "Не могу открыть файл!". Однако реальная ошибка связана с тем, что неправильно написано названия функции открытия файла: opn вместо open. Чтобы избежать получения сбивающих с толку сообщений об ошибках, нужно явно указывать, какой тип исключения вы планируете обрабатывать. В данном примере это IOError:

```
try:
    f = open(filename)
except IOError:
    sys.exit("Не могу открыть файл!")
```

Более подробно идиомы Python можно изучить в книгах Pythonic Programming (https://pragprog.com/titles/dzpythonic/pythonic-programming/),

Fluent Python (https://www.oreilly.com/library/view/fluent-python-2nd/9781492056348) и Clean Code in Python (https://www.oreilly.com/library/view/clean-code-in/9781788835831/).

Итоги

- Python спроектирован таким образом, чтобы помогать разработчику создавать чистый код.
- Рекомендованный подход к написанию кода на Python называется идиоматическим.
- Код, разработанный с использованием идиом Python, называется Pythonic код.
- Идиомы Python часто отличаются от идиом разработки кода на других языках программирования.
- Использование идиом Python сделает ваш код более элегантным, понятным для других Python разработчиков, поможет избежать ошибок, а также в некоторых случаях повысить производительность.

Модуль № 4. Юнит № 3. Рефакторинг.

Что такое рефакторинг

Рефакторинг, совместно с тестами, является одним из инструментов устранения проблем с качеством кода.

Как мы уже рассматривали ранее, **рефакторинг – это изменение кода с целью повышения его качества без изменения функциональности и поведения.**

Один из первых и самых крупных наборов рекомендаций по рефакторингу представлен в книге Мартина Фаулера “Рефакторинг. Улучшение существующего кода” [8]. Все рекомендуемые методы в этой книге достаточно простые и большая часть из них не требуют высоких затрат. Кроме того, методы рефакторинга обычно совпадают с рекомендациями по обеспечения высокого качества код. Среди наиболее эффективных действий для проведения рефакторинга можно отметить:

1. **Изменение имен переменных, классов, объектов на осмысленные.**

2. **Устранение дублирования кода.** Если код повторяется больше одного раза, то нужно вынести его в отдельную функцию или метод.
3. **Разделение больших функций.** Если функция или метод слишком большие и не помещаются на одном экране, рекомендуется разбить их на две части (или больше, если это требуется).
4. **Разделение больших файлов.** По аналогии с большими функциями, которые сложно понять, крупные файлы тоже рекомендуется разделять на несколько более маленьких. Отдельные файлы затем подключаются к основному файлу с кодом.
5. **Удаление мертвого кода.** Когда программа изменяется со временем, некоторые ее участки перестают использоваться. Они могут быть закомментированы, или просто никогда не вызываться. Аналогичная ситуация возможна с аргументами функций: когда-то они были полезные но со временем перестали использоваться.

   Такие участки мертвого кода рекомендуется удалять. Во-первых, наличие мертвого кода затрудняет понимание, что делает та или иная функция. Во-вторых, изменяя код функции можно случайно сделать так, чтобы мертвый код снова заработал, что может привести к ошибкам.

Другие методы рефакторинга можно найти в каталоге, созданном Мартином Фаулером – https://www.refactoring.com/catalog/

Рефакторинг и тесты

Успешное проведение рефакторинга невозможно без наличия тестов. Многие опытные разработчики боятся вносить какие-либо изменения в крупные программные системы из-за того, что могут возникнуть ошибки. Тесты позволяют справиться с этой проблемой.

Так как в процессе рефакторинга изменяется только структура кода, но не его функциональность, то успешный запуск тестов после рефакторинга позволяет подтвердить его безопасность.

Рекомендуется разделять процесс рефакторинга на отдельные небольшие шаги (примеры таких шагов приведены в предыдущем разделе) и после каждого шага запускать тесты, чтобы убедиться в успешности изменений. Кроме того, можно создавать отдельный коммит для каждого шага

рефакторинга. Таким образом можно будет быстро и просто откатить изменения в случае, если на тестах будет обнаружена проблема.

Рефакторинг в среде разработки

Несмотря на то, что большая часть методов рефакторинга содержит набор простых действий, на практике выполнять рефакторинг нелегко. Например, вы обнаружили в коде функцию с неинформативным именем и решили это исправить. Однако для этого недостаточно изменить имя в определении функции. Вам придется искать все места в коде, где используется эта функция, и менять ее название. Если функция используется часто, то вручную внести все изменения достаточно сложно.

Многие современные среды разработки содержат инструменты автоматизации рефакторинга. Пример внешнего вида окна PyCharm с вызванным меню рефакторинга показан на рис. 2. Чтобы вызвать это меню, нужно открыть контекстное меню, нажав правую клавишу мыши, и в контекстном меню выбрать пункт “Refactoring”. Альтернативный вариант – нажать сочетание клавиш Shift+T.

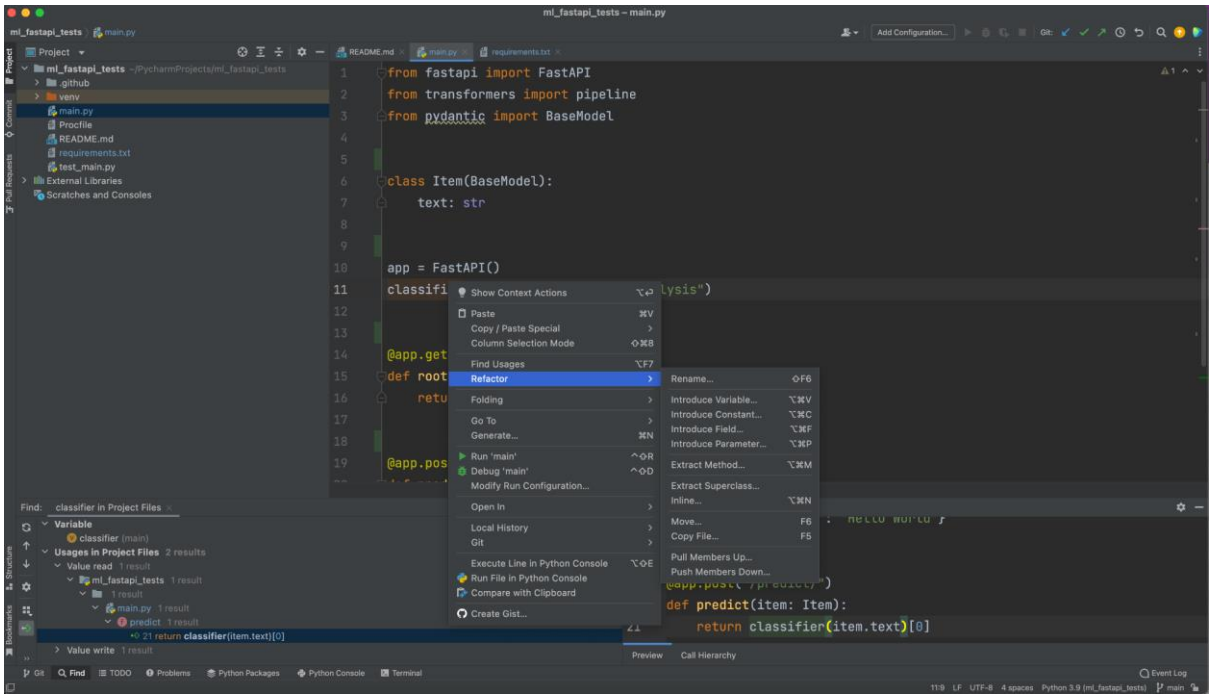


Рис. 2. Рефакторинг в PyCharm

Полезные возможности рефакторинга, который умеет делать PyCharm:

- Rename… – переименование переменной, функции, класса или объекта.

- Extract Method… – вынесение выделенного кода в отдельную функцию или метод и вставка вызова этой функции.
- Inline… – действие противоположное "Extract Method…" – вставка копии кода из функции.

Описание других полезных методов рефакторинга можно найти по ссылкам:

- PyCharm – https://www.jetbrains.com/help/pycharm/refactoring-source-code.html#popular-refactorings
- VS Code – https://code.visualstudio.com/docs/editor/refactoring

Итоги

- Рефакторинг – это изменение кода с целью повышения его качества без изменения функциональности и поведения.
- Для успешного проведения рефакторинга необходимы тесты, чтобы быть уверенными в безопасности изменений.
- Рекомендованные методы рефакторинга собраны в каталоге Мартина Фаулера – https://www.refactoring.com/catalog/
- Эффективнее всего выполнять рефакторинг используя среду разработки, такую как PyCharm или VS Code.

Модуль № 4. Юнит № 4. Технический долг в программной инженерии.

Метафора технического долга

В реальных проектах не всегда удается поддерживать качество кода на требуемом уровне. Популярные сейчас подходы к управлению программными проектами приоритезируют работу над созданием новых функций, полезных пользователю. Проблемы же с качеством кода на работе пользователей могут не отражаться, по крайне мере в первое время после их возникновения. Поэтому менеджерам часто сложно принять решение о выделении времени для работы над качеством кода, ведь это означает, что нужно отвлечь разработчиков от реализации полезных для пользователя функций.

Таким образом, приходится находить компромисс между разработкой новых функций и совершенствованием кода. При этом иногда требуется быстро реализовать нужную функциональность, не обращая внимания на чистоту кода, а затем уже постепенно улучшать качество. В этом нет ничего плохого, главное, необходимо иметь инструмент, который позволяет не

забыть обо всех проблемах в коде со временем их исправить. Именно таким инструментом и является технический долг.

Технический долг позволяет мыслить о качестве кода в финансовых терминах. Предположим, что берем ипотечный кредит для того, чтобы купить квартиру. Затем нам нужно платить по этому кредиту каждый месяц часть суммы задолженности и проценты. Если мы не будем платить, то со временем банк квартиру у нас заберет.

Похожим образом работает и технический долг. Чтобы быстро реализовать нужную пользователям функциональность, мы берем кредит: используем некачественный код. Затем нам нужно запланировать выплаты по этому долгу: регулярно часть рабочего времени посвящать тому, чтобы повысить качество кода. Делать это придется в течение нескольких итераций разработки, пока все проблемы с качеством не будут исправлены. Тогда технический долг будет возвращен.

Если же выплат по техническому долгу не будет, то проблемы, связанные с качеством кода будут нарастать. Когда мы не платим по ипотечному кредиту, то банк забирает нашу квартиру, а когда мы не платим по техническому долгу, то у нас снижается эффективность разработки и со временем мы потеряем возможность вносить любые изменения в код без возникновения большого количества ошибок.

Технический долг в системах машинного обучения

В машинном обучении традиционно основной акцент делался на разработке моделей, позволяющим обеспечить лучшее качество решения задачи (State-Of-The-Art). Однако в практических системах машинного обучения код, относящийся к собственно машинному обучению, занимает относительно небольшой объем. Для обеспечения практического использования этого кода нужно большое количество других компонентов. Анализ таких компонентов приведен в статье “Скрытый технический долг в системах машинного обучения”. Архитектура типичного приложения машинного обучения приведена на рис. 3.

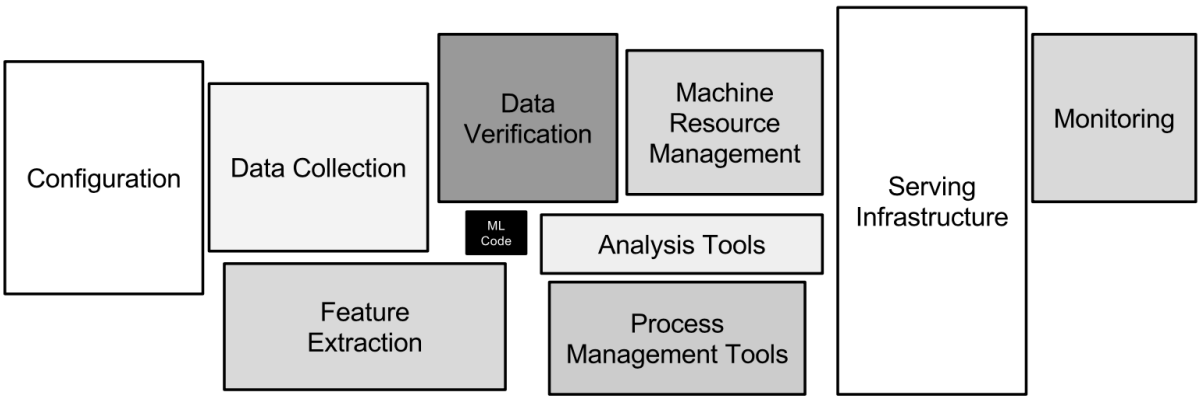


Рис. 3. Скрытый технический долг в системах машинного обучения [10]

Кроме кода, относящегося к собственно машинному обучению, система должна включать следующие компоненты:

- Управление конфигурациями (Configuration) – определяет, какие признаки используются, какие данные применяются, какие гиперпараметры передаются в алгоритмы и т.п.
- Сбор данных (Data Collection) – код для сбора и подготовки данных, которые используются для обучения модели.
- Выбор признаков (Feature Extraction) – код, который выделяет из данных наиболее полезные признаки. В том числе могут генерироваться новые признаки на основе данных, если это полезно для качества работы модели машинного обучения.
- Проверка данных (Data Verification) – код, которые проверяет качество данных и делает так, чтобы неправильные данные не использовались для обучения модели, а также не передавались в модель на этапе предсказания.
- Менеджер ресурсов (Machine Resource Management) – система распределения ресурсов вычислительного оборудования между потребителями: виртуальными машинами, контейнерами или приложениями. Для крупных систем менеджер ресурсов должен уметь управлять не одной машиной, а кластером из серверов.
- Менеджер процессов (Process Management Tools) – инструменты для организации процессов обработки данных, например, пайплайна машинного обучения.
- Инфраструктура для развертывания (Serving Infrastructure) – вычислительное и сетевое оборудование, на котором развертывается продуктивная версия системы машинного обучения.

- Мониторинг (Monitoring) – система мониторинга работоспособности инфраструктуры, в которой работает приложение машинного обучения, а также качества работы модели.

Если большая часть времени выделяется только на совершенствование модели, а остальные компоненты системы остаются почти без внимания, то обеспечить качественный и надежный сервис для пользователей такая система не в состоянии.

Управление техническим долгом

Чтобы технический долг не остался красивой метафорой для описания плохого кода необходимы действенные инструменты по управлению техническим долгом. В качестве наиболее важных инструментов можно выделить следующие:

1. **Определение источника технического долга.** Технический долг может быть двух типов:
    - Осознанный, при котором решение о снижении качества кода принималось осмысленно в качестве компромисса для повышения скорости разработки.
    - Неосознанный – технический долг возникший вследствие низкой квалификации разработчиков, недостатка планирования, нехватки времени или по каким-то другим причинам. Обнаружить неосознанный технический долг помогают инструменты программной инженерии (статический анализатор кода, линтер и т.п.). Также неосознанный технический долг может быть обнаружен в процессе исправления ошибок в коде.

    В качестве примера давайте рассмотрим ситуацию, при которой статический анализатор обнаружил 5 копий одно и того же кода в различных частях системы. Несогласованное изменение копий приводило к ошибкам, причину которых было очень трудно определить.
2. **Оценка технического долга.** Несмотря на название *Технический*, технический долг является экономической проблемой. Первым шагом по управлению техническим долгом является оценка:
    - Объема ресурсов (количество людей и их время), которые нужны для устранения технического долга.

- Эффекта, который будет получен в результате устранения технического долга.

Если потенциальный эффект выше, чем объем требуемых ресурсов, то имеет смысл планировать работы по устранению проблем с качеством кода.

Продолжим рассматривать наш пример с техническим долгом, вызванным наличием пяти копий кода. Команда предложила вместо копий кода создать одну функцию и вызвать ее во всех необходимых местах. Трудозатраты на это составят ориентировочно час времени. К этому нужно прибавить время на запуск всех необходимых тестов. Кроме того, нужно учесть время, необходимое на отладку, т.к. изменения будут вноситься в 5 разных частей системы. Итого, для решения проблемы нужен ориентировочно день работы одного специалиста.

Эффект от решения можно оценить на основе результатов исправления ошибки, вызванной согласованным изменением копий в коде. На поиск причины ошибки и ее устранение у разработчика ушло половина рабочего дня.

Если в будущем будет выполнено хотя бы две модификации дублированных участков кода, то эффект от устранения проблемы с качеством кода можно считать превышающим трудозатраты.

3. **Планирование устранения технического долга.** Оценив объем и эффекты от устранения технического долга можно переходить к планированию.

   Рекомендуемый подход: выделять от 10 до 20% времени на устранение технического долга. В нашем примере один рабочий день разработчика составляет как раз 10% от двухнедельного спринта, которые используются в команде. Таким образом, эту задачу можно включить в бэклог спринта для разработчика.

   Если объем трудозатрат существенно превышает 20% от времени разработчиков, то имеет смысл разделить устранение технического долга на несколько этапов. Как мы рассматривали в разделе про рефакторинг, каждый отдельный рекомендованный метод прост и не трудозатратен. Поэтому, как правило, устранение большей части проблем с качеством кода можно выполнить за несколько шагов рефакторинга.

Итоги:

- Эффективность команд, разрабатывающих крупные программные системы, снижается со временем.
- Плохое качества кода приводит к тому, что крупную программную систему сложно поддерживать и невозможно развивать.
- Чтобы обеспечить жизнеспособность системы на длительном промежутке времени (несколько лет), нужно сместить акцент с реализации новых функций на обеспечение возможности поддержки системы в течение всего периода ее жизни.
- Чистый код позволяет легко поддерживать и развивать крупную программную систему.
- Ключевые инструменты поддержания высокого качества кода: разработка тестов и рефакторинг.
- На практике нет возможности всегда обеспечивать чистый код. Для управление качества кода на протяжении длительного времени используется метафора технического долга.
- Для программных систем, использующих машинное обучение, характерно пристальное внимание только к коду, реализующему машинное обучение. В результате возникает скрытый технический долг, связанный со всеми остальными компонентами системы.

Итоговый тест

Как изменяется эффективность работы команд, создающих крупные программные системы, со временем:

1. Увеличивается
2. **Уменьшается**
3. Остается без изменений
4. Движется в разных направлениях

Какие последствия может иметь длительное использование плохого кода в приложении:

1. **Любые изменения в коде, даже небольшие, приводят к возникновению ошибок, причину которых сложно найти.**
2. Пользователи заметят плохой код и перестанут работать с приложением.
3. Приложение легко поддерживать и расширять.

4. Снижение стоимости разработки приложения, т.к. писать плохой код можно быстро и дешево.

Что из нижеперечисленного является признаками чистого кода

1. **Осмысленные имена переменных.**
2. **Короткие функции, помещающиеся на один экран.**
3. Дублирование кода в разных частях системы.
4. Отсутствие обработки ошибок.
5. **Оформление кода в соответствии с руководством по стилю PEP8.**
6. Отсутствие тестов.

Какой фрагмент использует наиболее идиоматический для Python код.

1.

```
f = open(filename)
try:
    process_file(f)
finally:
    f.close()
```

2.

```
f = open(filename)
try:
    process_file(f)
except:
    sys.exit("Ошибка при обработке файла!")
finally:
    f.close()
```

3. (Правильный вариант)

```
with open(filename) as f:
    process_file(f)
```

4.

```
f = open(filename)
try:
    process_file(f)
except IOError:
    sys.exit("Ошибка при обработке файла!")
```

```
finally:
    f.close()
```

Что такое рефакторинг?

1. Реализация новых функций в коде.
2. Изменение кода с целью повышения его производительности без изменения функциональности и поведения.
3. Механизм автоматического улучшения качества кода в методике CI/CD.
4. **Изменение кода с целью повышения его качества без изменения функциональности и поведения.**

Для каких целей используется технический долг?

1. Для получения финансовых ресурсов для повышения качества кода.
2. Для увеличения сроков разработки крупных программных систем.
3. **Для нахождения компромисса между разработкой новых функций и совершенствованием кода.**
4. Технический долг в крупных программных системах использовать категорически недопустимо.

Практическое задание

В командах по 2-4 человека проведите рефакторинг кода своего проекта. Для этого:

1. Проанализируйте код в репозитории GitHub своего проекта и оцените, как можно улучшить его качество.
2. Запишите все идеи по улучшению качества кода, которые имеет смысл реализовать.
3. Выберите и реализуют 2-3 предложения по улучшению качества кода.
4. Каждое улучшение должно быть оформлено в виде отдельного коммита с описанием смысла и содержания улучшения.
5. Для сдачи задания пришлите на платформу ссылку на репозиторий с кодом, в котором есть минимум 2 коммита, посвященных рефакторингу.

Список источников

1. Роберт Мартин. Чистый код. Создание, анализ и рефакторинг. 2010.
2. PEP 20 – The Zen of Python. – https://peps.python.org/pep-0020/
3. Robert Martin. Clean Craftsmanship: Disciplines, Standards, and Ethics. 2021.
4. Kent Beck. Test Driven Development: By Example. 2002.
5. Mariano Anaya. Clean Code in Python: Refactor your legacy code base. 2018. https://www.oreilly.com/library/view/clean-code-in/9781788835831/
6. Pythonic – https://habr.com/ru/post/114731/
7. PEP 343 – The “with” Statement – https://peps.python.org/pep-0343/
8. Martin Fowler. Refactoring: Improving the Design of Existing Code. – https://martinfowler.com/books/refactoring.html
9. Philippe Kruchten, Robert Nord, Ipek Ozkaya. Managing Technical Debt: Reducing Friction in Software Development – https://learning.oreilly.com/library/view/managing-technical-debt/9780135646052/
10. D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, Dan Dennison. Hidden Technical Debt in Machine Learning Systems // NIPS'15: Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2 December 2015 Pages 2503–2511– https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems

Рекомендуемая литература

- Write More Pythonic Code – https://realpython.com/learning-paths/writing-pythonic-code/
- PEP 8 – Style Guide for Python Code – https://peps.python.org/pep-0008/
- Cyclomatic complexity (McCabe complexity) – https://en.wikipedia.org/wiki/Cyclomatic_complexity
- McCabe complexity checker – https://github.com/PyCQA/mccabe
- GitHub Copilot – https://copilot.github.com/
- Dmitry Zinoviev. Pythonic Programming – https://pragprog.com/titles/dzpythonic/pythonic-programming/

- Luciano Ramalho. Fluent Python – https://www.oreilly.com/library/view/fluent-python-2nd/9781492056348/
- Каталог методов рефакторинга от Мартина Фаулера - https://www.refactoring.com/catalog/

Модуль № 5
Название: Код-ревью

Образовательный результат:

- Студент может объяснять важность выполнения код-ревью при внесении изменений в программное обеспечение.
- Студент может выполнять код-ревью на платформе GitHub.

В этом модуле:

В предыдущем модуле мы рассмотрели различные варианты организации совместной разработки (collaborative programming), использование которых позволяет повысить качество кода и повысить стабильность работы приложения. В этом модуле мы подробно рассмотрим один из таких методов, который широко используется на практике многими командами в большом количестве компаний – **код-ревью**.

Вы научитесь проводить код-ревью с помощью бесплатных инструментов GitHub, доступных всем разработчикам, а также узнаете, как правильно организовать код-ревью в команде.

Также мы рассмотрим, какое место занимает код-ревью в процессе CI/CD в целом.

Модуль № 5. Юнит № 1. Понятие код-ревью

Код-ревью – это одна их техник совместной разработки, которая широко применяется на практике многими компаниями.

Техника код-ревью предполагает, что код разрабатывает один человек, но после того, как работа закончена, код проверяют другие участники команды – **ревьюеры**. Они оценивают разработанный код по нескольким параметрам, среди которых:

- Наличие ошибок в коде.
- Стиль кода.
- Качество кода.
- Архитектурные решения.
- Наличие тестов и их адекватность.
- Эффективность использованных в коде решений.

В случае успешной оценки ревьюер одобряет код, в противном случае – отправляет код на доработку автору. Автор должен исправить все замечания к коду от ревьюера, или убедить его, что принятые решения обоснованы т.к. ревьюер, возможно, не учел какие-то важные факторы при проведении оценки.

Хорошая практика состоит в том, что каждый код проверяет минимум два человека. Тогда выше вероятность, что они заметят ошибки или какие-то другие проблемы с кодом. Объединение предлагаемого кода с основной версией рекомендуется выполнять только после того, как будет получено два или больше одобрений от ревьюеров.

Проведение код-ревью обеспечивает следующие преимущества:

- **Повышение стабильности работы приложения.** Ревьюеры могут увидеть ошибки, которые не заметил автор кода. Также ревьюеры оценивают наличие и адекватность тестов и в случае сомнений могут направить код на доработку с целью дописать необходимые тесты. В этом случае возможные ошибки будут обнаружены новыми тестами. Особенно этот эффект от код-ревью проявляется, если его проводят опытные разработчики.
- **Распространение знаний в команде.** Каждый разработчик в команде участвует в разработке только части кода в системе. В результате может оказаться, что в некоторых частях кода разбирается только один человек. И если этот человек заболеет, уедет в отпуск или уволится, то изменить или модернизировать эту часть кода будет проблематично. Регулярное проведение код-ревью всеми участниками команды позволяет изучить различные части кода и обсудить с авторами, почему тот или иной фрагмент программной системы устроен именно таким образом. В результате все участники команды как минимум имеют общее представление о системе в целом и достаточно глубоко разбираются в большей части кода. Для

достижения этой цели очень важно, чтобы ревью кода проводили все участники команды, а не только опытные.

- **Повышение качества кода.** Код плохого качества почти невозможно определить тестами, т.к. плохой код не влияет на результаты работы программы, видные из внешнего мира. Зато опытный разработчик во время ревью может обнаружить плохой код достаточно быстро и сформулировать предложения по улучшению его качества. Кроме того, ревьюеры во время анализа кода могут предложить решения по улучшению качества, которые не пришли в голову автору. Наример, ревьюер может знать, что для реализации задачи, которая стояла перед разработчиком, соседняя команда уже написала достаточно хорошую библиотеку. Поэтому он может порекомендовать использовать эту библиотеку вместо того, чтобы реализовывать все самостоятельно. В результате не только увеличится скорость разработки, но и повысится переиспользуемость кода.
- **Обучение начинающих разработчиков.** Начинающим разработчикам, особенно новичкам в команде, очень сложно быстро разобраться с архитектурой приложения, используемым в команде подходам к хорошему и плохому коду, стилю кода, выбору архитектурных решений. Поэтому ревью для кода новичков может проводиться более внимательно, чем ревью кода более опытных разработчиков. Кроме того, во время ревью кода новичка более опытный разработчик может не просто указать на ошибки, который совершил новичок, но и дать рекомендации, как такие ошибки лучше исправить. А также объяснить, почему предлагаются именно такие подходы к исправлению проблем, в чем их достоинства и недостатки.

Как и у любого инструмента программной инженерии, у код-ревью есть недостатки:

- **Проведение код-ревью требует времени**. В зависимости от качества анализа, от код-ревью может занимать от 5 до 20% рабочего времени программиста.
- **Код-ревью снижает скорость разработки**. Код не может быть использован сразу после завершения его разработки. Необходимо время на то, чтобы несколько человек провели проверку кода. Если в команде не оговорены правила регулярного выделения времени для код-ревью, то задержка с готовностью кода может составить несколько дней (в худшем случае недель).

- **Возврат кода на доработку в результате код-ревью эмоционально неприятен авторам.** Как правило, большая часть авторов гордятся результатами своего труда и не любят, когда другие люди указывают им на ошибки. Даже если такие ошибки действительно существуют и рациональное решение заключается в том, что их нужно исправить, на эмоциональном уровне получать замечания от других людей к своему коду очень неприятно.
- **Код-ревью может приводить к конфликтам в команде.** Некоторые разработчики формулируют замечания к коду в некорректной форме. К сожалению, для российских ИТ-шников в целом характерна токсичная манера общения. Такое общение приводит к конфликтам в команде. Также ревьюеры могут давать замечания по спорным вопросам, по которым нет компромисса в команде, и требовать их обязательного исполнения. В результате вместо продуктивной работы начинаются религиозные войны (holy wars).

Итоги:

- Код-ревью – это одна их техник совместной разработки, при которой один человек разрабатывает код, а другой (или другие) проверяет его качество.
- В результате проведения код-ревью может быть принято решение об одобрении кода или возвращении его на доработку.
- Использование код-ревью снижает количество ошибок, повышает качество кода, обеспечивает возможность распространения знаний и обучения новичков.
- Однако проведение код-ревью требует времени и может привести к конфликтам в команде.
- На практике положительные стороны код-ревью чаще всего перевешивают отрицательные, поэтому код-ревью применяется в большом количестве команд во многих компаниях.

Модуль № 5. Юнит № 2. Инструменты для проведения Code Review в GitHub

GitHub, как и многие другие системы организации командной разработки программного обеспечения, содержит встроенные инструменты для проведения код-ревью. Эти инструменты доступны бесплатно всем заинтересованным разработчикам.

В GitHub код-ревью запускается после создания pull request (дать ссылку на Модуль № 2. Юнит № 5.). Перед тем, как pull request будет объединен с общей версией программы, для его кода может быть проведено ревью. И только после того, как один или несколько других разработчиков одобрят предлагаемые изменения, будет выполнено объединение.

При настройках по умолчанию код-ревью может выполнятся по желанию. Однако есть возможность настроить работу GitHub таким образом, чтобы для некоторых веток код-ревью проводилось обязательно. Для этого используется механизм **защищенных веток** (https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/defining-the-mergeability-of-pull-requests/about-protected-branches).

Запрос код-ревью для pull request

После создания pull request вы можете запросить своих коллег, у которых есть доступ к репозиторию, провести код-ревью. Для этого в правой части экрана pull request нужно выбрать поле ввода “Reviewers”, начать вводить имя интересующего вас пользователя и затем выбрать его в выпадающем списке (рис. 1).

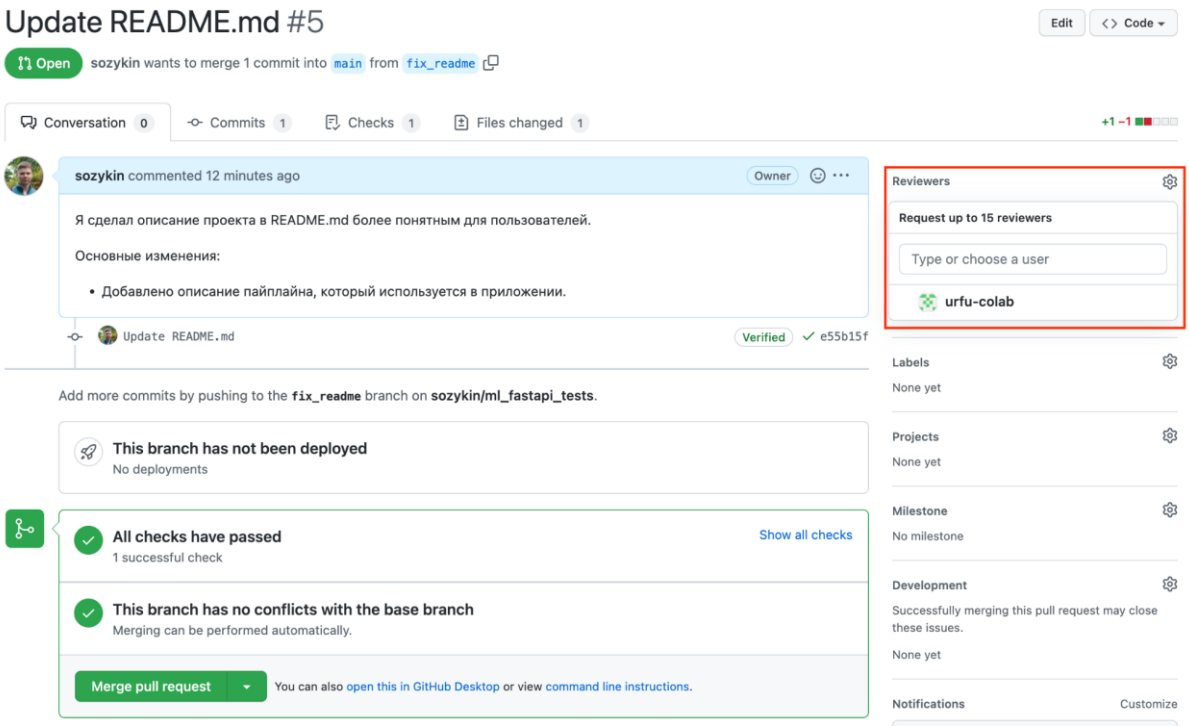


Рис. 1. Приглашение провести ревью кода в pull request

После выбора пользователя для приглашения в ревьюеры, информация о нем появится в разделе “Review Requested” (рис. 2).

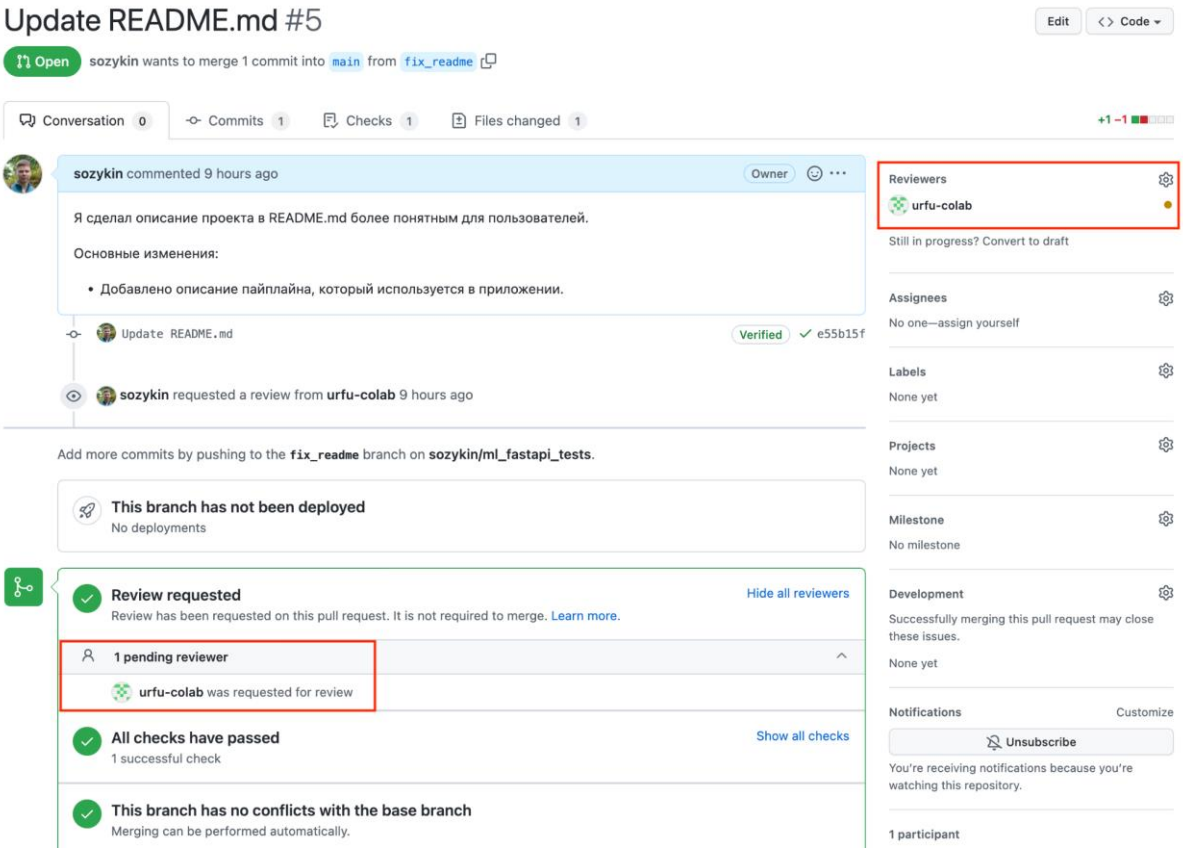


Рис. 2. Информация о приглашенном ревьюере

Пользователь, у которого попросили провести ревью, увидит этот запрос в верхней части экрана интерфейса GitHub (рис. 3). Чтобы начать ревью, нужно нажать кнопку “Add your review”.

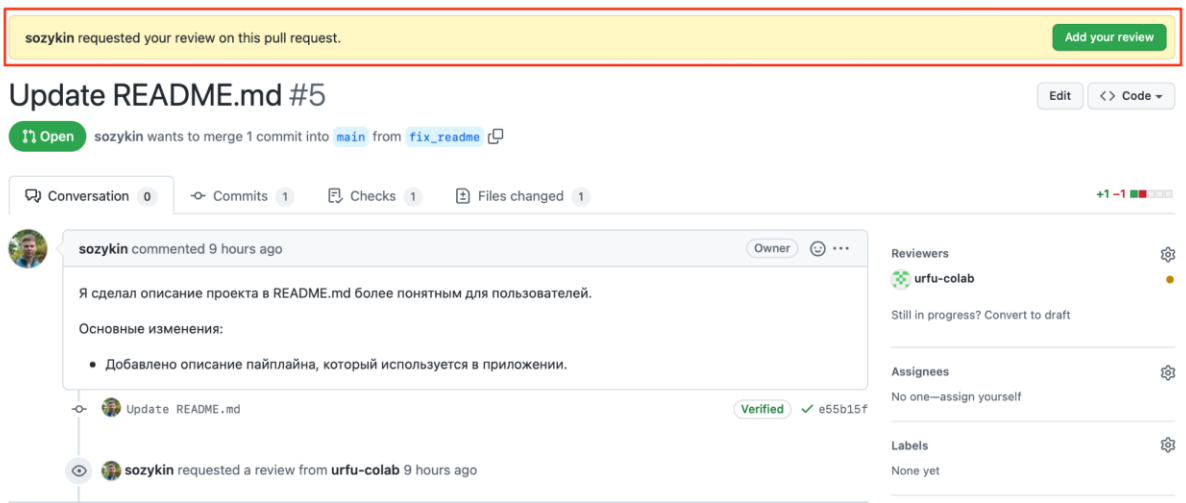


Рис. 3. Запрос на проведение код-ревью

Проведение код-ревью

В результате проведения обзора кода ревьюер должен принять одно из двух решения:

- **Одобрить** предлагаемые изменения, если он считает, что в коде нет существенных ошибок и проблем. В этом случае ревьюер соглашается с тем, что pull request может быть объединен с основной версией кода.
- **Запросить изменения в код** в случае, если ревьюер считает, что в коде есть ошибки и в текущем виде pull request не может быть объединен с основной версией кода. В этом случае разработчику нужно переработать код для устранения замечаний, который сформулировал ревьюер.

В интерфейсе проведение код-ревью на GitHub доступны следующие действия (рис. 4):

1. Комментирование кода в pull request. Комментарии могут быть привязаны к конкретным строкам, к которым у ревьюера возникли вопросы или замечания.
2. Формулировка итогового результата проведения ревью:
    - **Одобрить (Approve)** – одобрить pull request.
    - **Запросить изменения (Request change) –** отправить замечания к коду, которые должны быть устранены.
    - **Комментировать** – отправить отзыва о коде без высказывания четкого решения: одобрен ли код, или нужно запросит изменения.

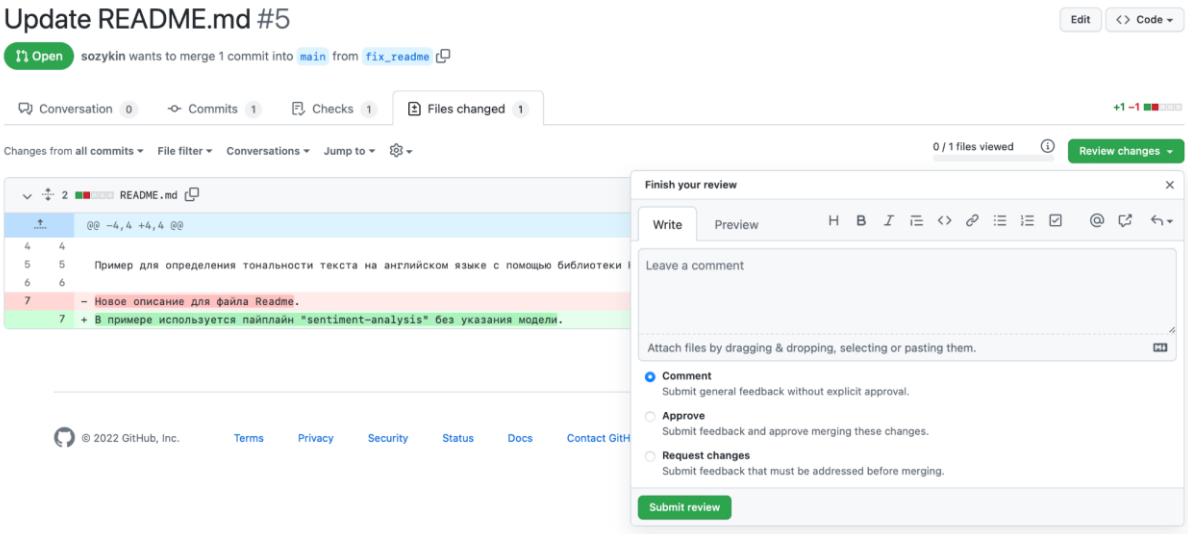


Рис. 4. Интерфейс проведения код-ревью

Комментирование кода

Комментарии к коду в процессе проведения код-ревью позволяют сделать процесс исправления изменений в коде более удобным и быстрым. Комментарий привязывается к конкретной строке кода, в которой ревьюер нашел проблему, поэтому разработчик сразу может определить, где именно находится проблема.

Для того, чтобы создать комментарий, нужно выбрать строку кода, которая вызывает вопросы, и нажать на синюю кнопку с символом “+” (рис. 5).

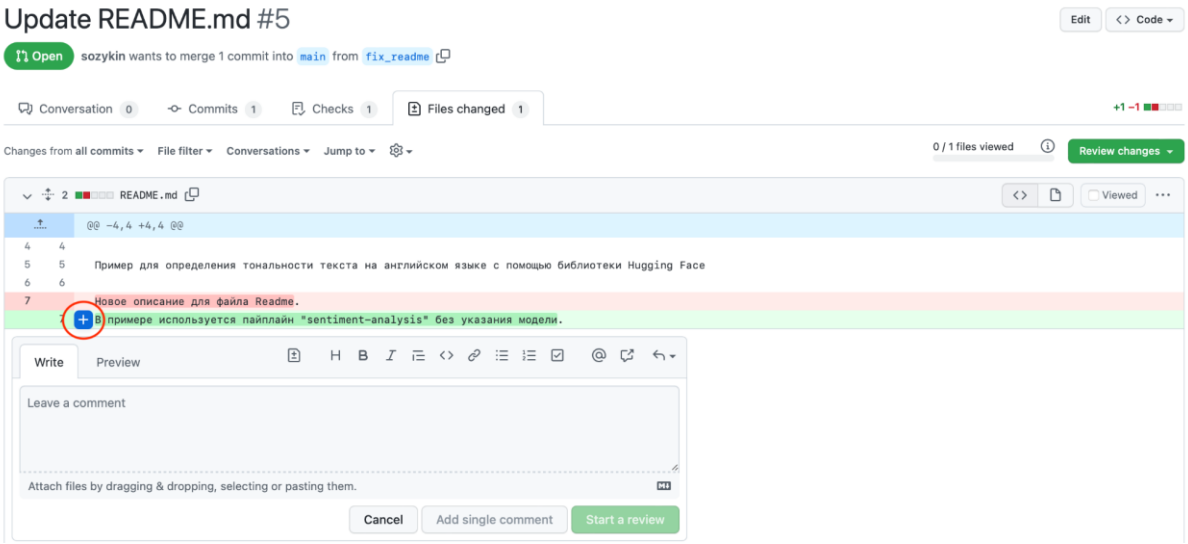


Рис. 5. Создание комментария к коду

Текст нужно написать в появившемся ниже окне (рис. 6). Отправит комментарий можно нажав одну из двух кнопок:

- Start a review – добавить комментарий к коду, а затем перейти к написанию полного ревью.
- Add single comment – добавить комментарий, но при этом не начинать процесс написания ревью.

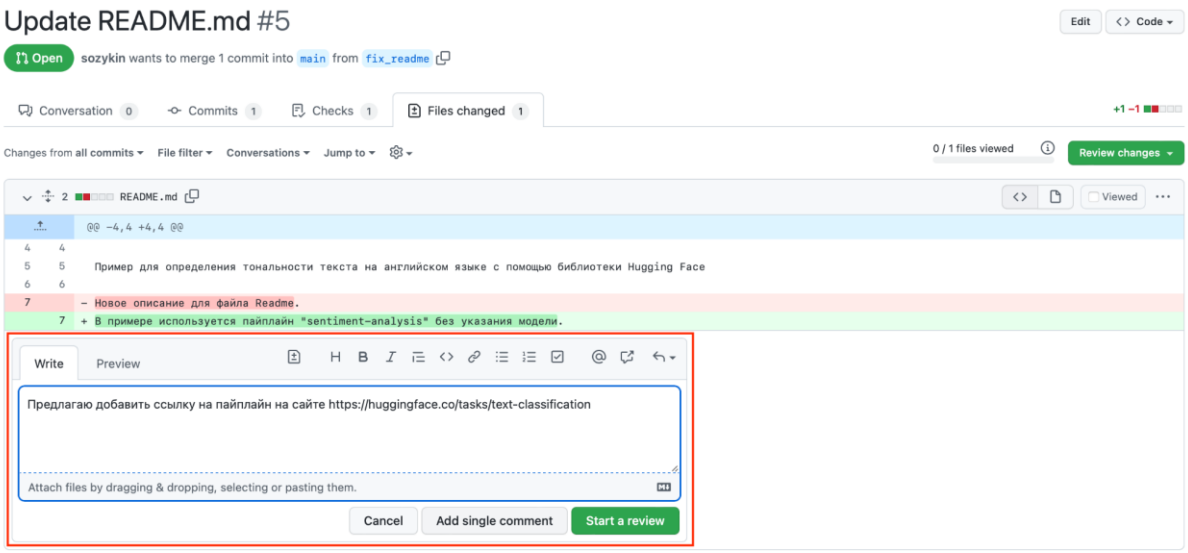


Рис. 6. Создание комментария к коду

Создатель pull request увидит комментарий к коду в ленте обсуждения pull request (рис. 7). Он может внести изменения в код, при необходимости, и написать ответ на комментарий.

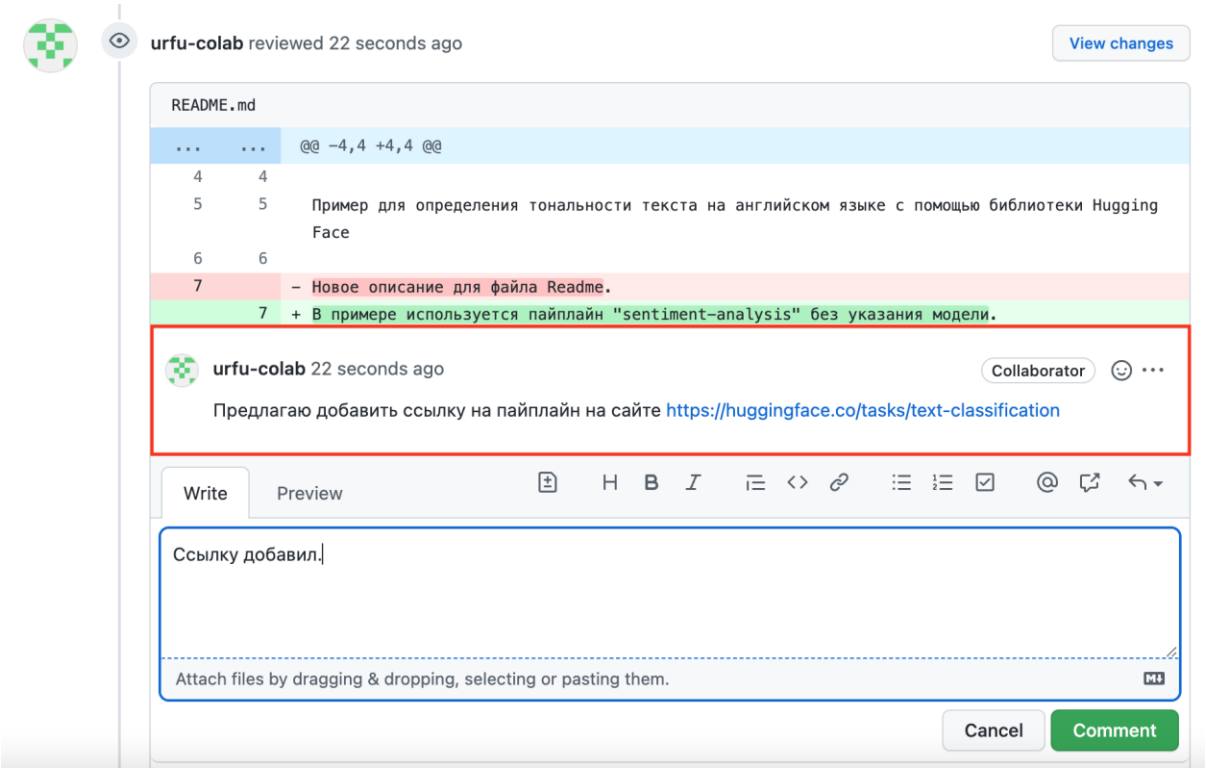


Рис. 7. Ответ на комментарий от создателя pull request

Ревьюер увидит ответ на комментарий от автора pull request рядом с текстом комментария (рис. 8). Если ответ автора достаточен для того, чтобы устранить замечания, то ревьюер может нажать кнопку “Resolve

conversation”, чтобы закончить обсуждение. В противном случае можно написать еще один комментарий и продолжить диалог.

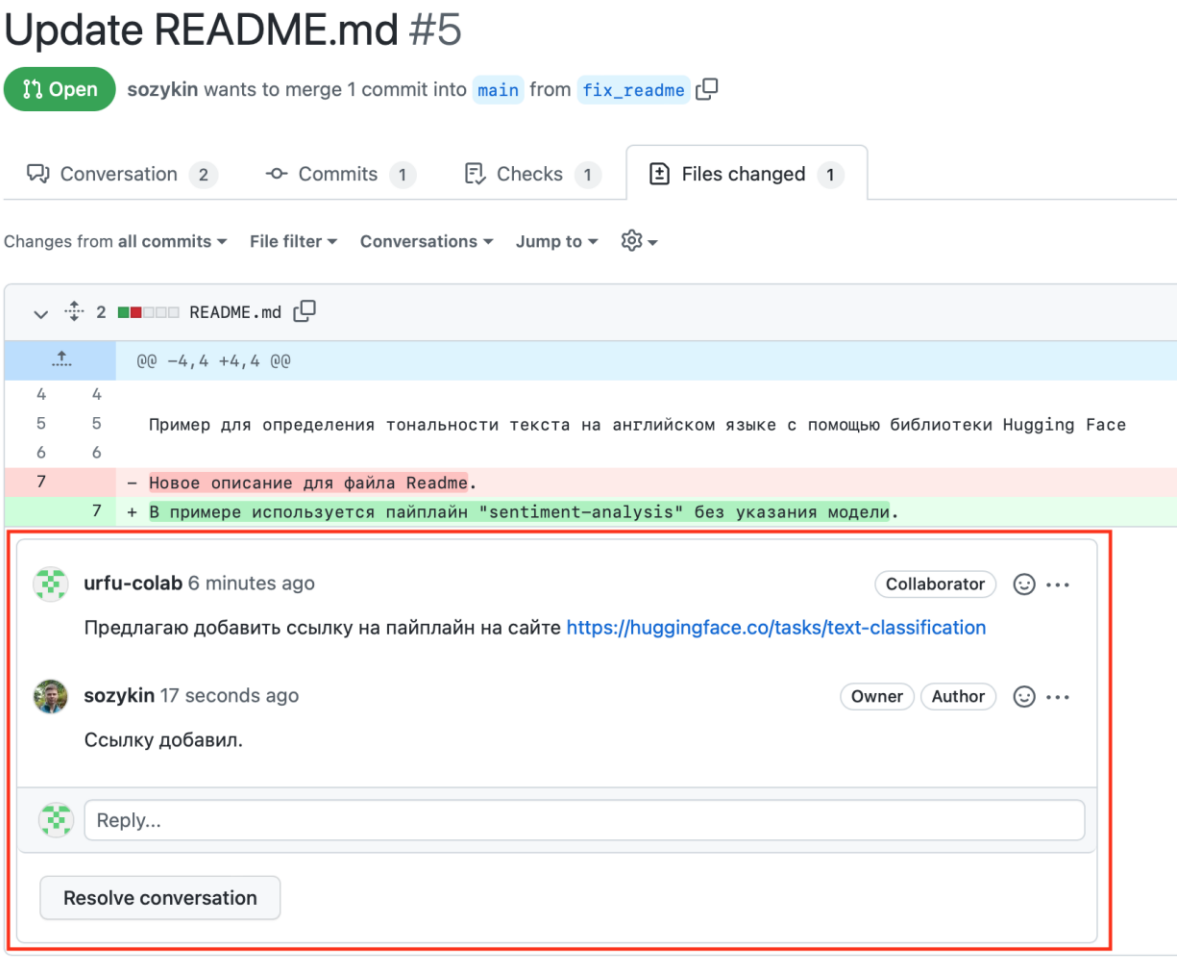


Рис. 8. Просмотр ответа на комментарий от создателя pull request

Одобрение кода

Если вы, как ревьюер, просмотрели код и считаете, что в нем все хорошо написано и pull request может быть объединен с основной версией кода, то можно одобрить код. Для этого нужно выполнить следующие действия (рис. 9):

- Нажать на кнопку “Review changes”.
- В появившемся окне написать текст сообщения о том, что с кодом все хорошо.
- Под полем ввода сообщения выбрать результат проведения ревью: Approve.
- Нажать кнопку “Submit review” для завершения процесса ревью.

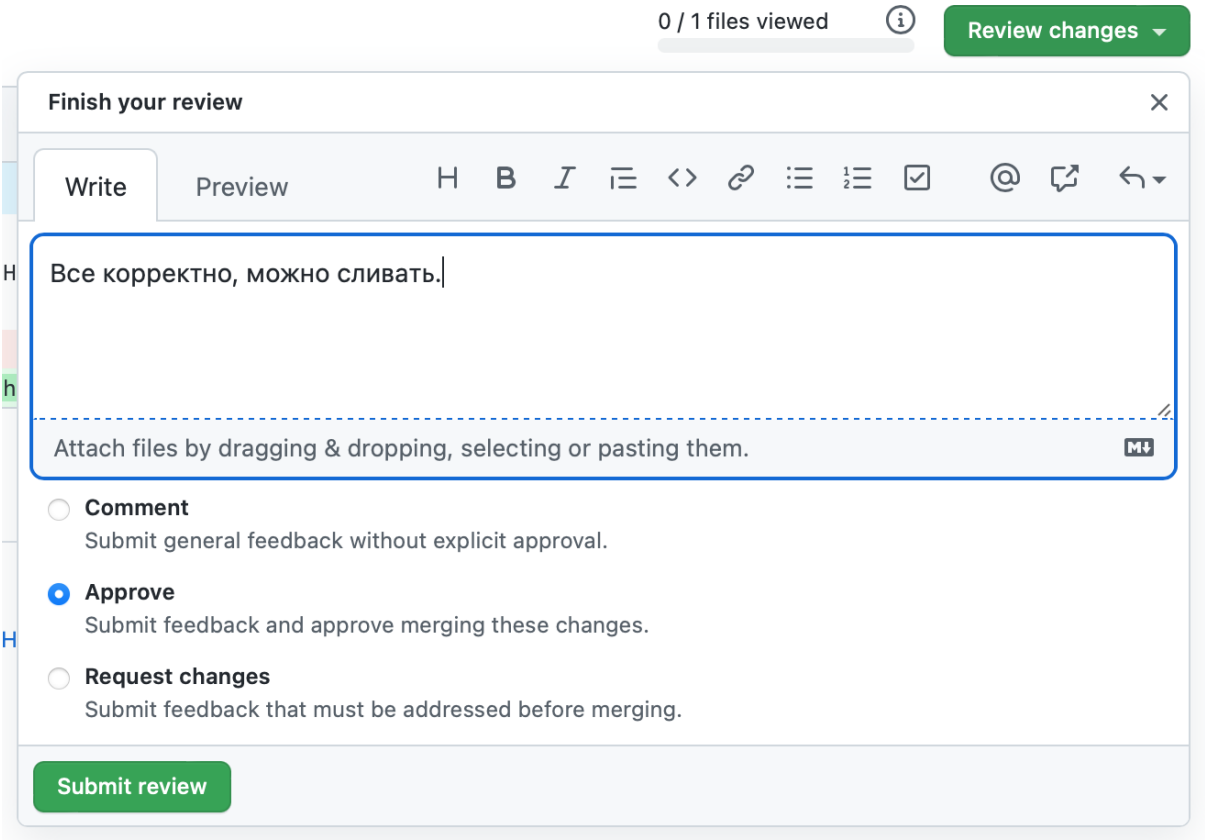


Рис. 9. Одобрение изменений в коде

Автор pull request увидит, что ревьюер его одобрил, в правой верхней части экрана обсуждения pull request, где указан список ревьюеров (рис. 10), а также в нижней части обсуждения перед итоговой информацией о том, готов ли pull request к объединению (рис. 11).

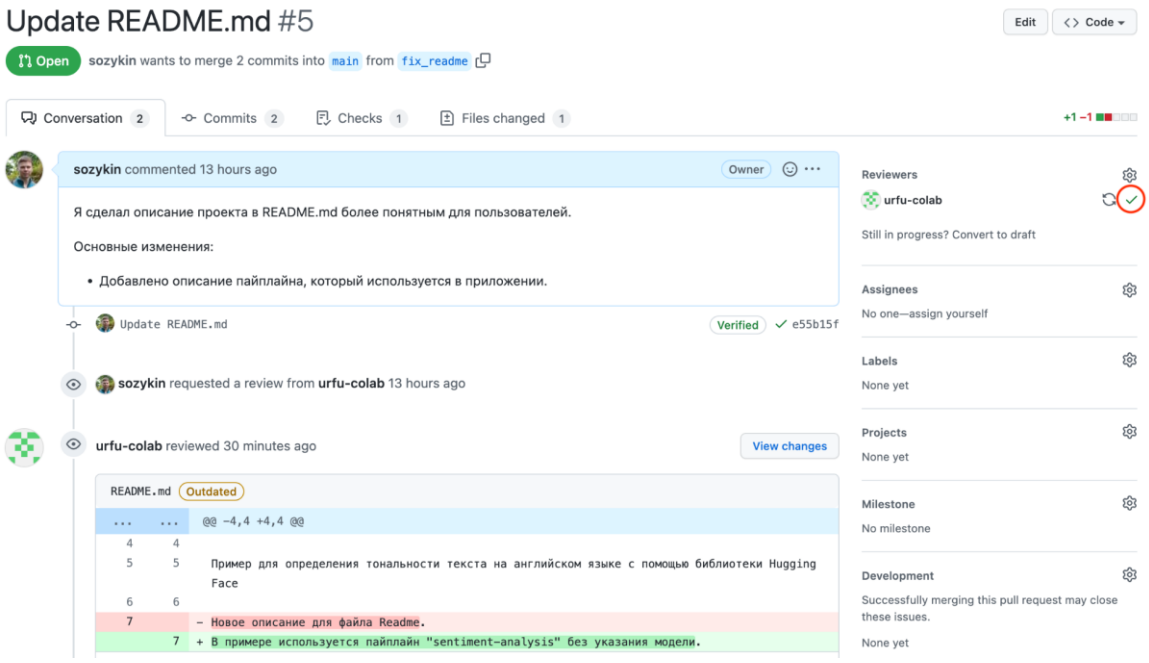


Рис. 10. Информаций о проведенном ревью с одобрением pull request

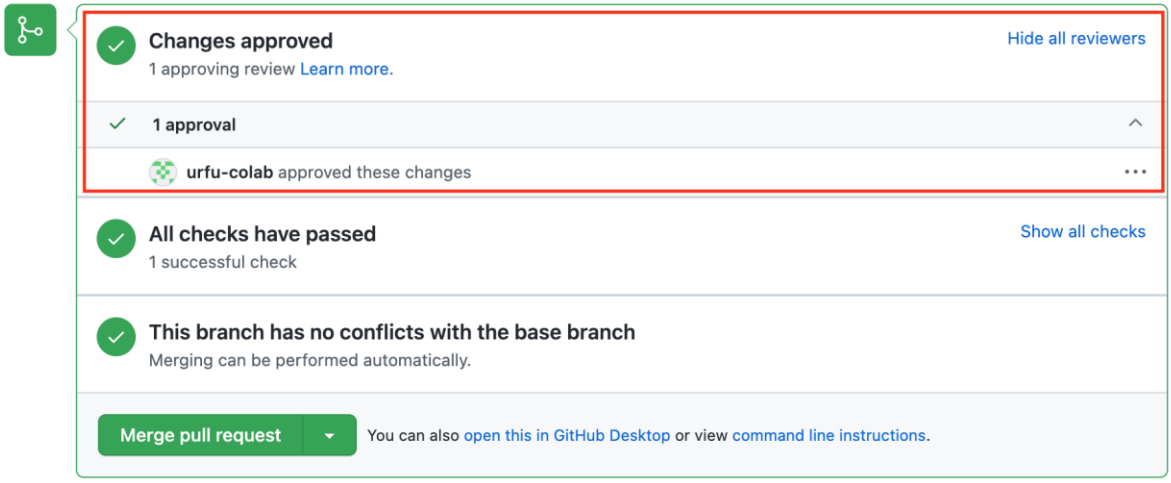


Рис. 11. Итоговая информация обсуждения pull request

При наличии нужного количество одобрений может быть выполнено объединение pull request в общую версию кода. Для этого нужно нажать кнопку “Merge pull request”.

Запрос на изменения

В случае, если в коде, предложенном в pull request есть ошибки или неточности, то ревьюер может указать на них и попросить исправить, отправив запрос на изменение как результат ревью. Для этого нужно выполнить следующие действия (рис. 12):

- Нажать на кнопку “Review changes”.
- В появившемся окне написать свои замечания к коду.
- Под полем ввода сообщения выбрать результат проведения ревью: Request Change.
- Нажать кнопку “Submit review” для завершения процесса ревью.

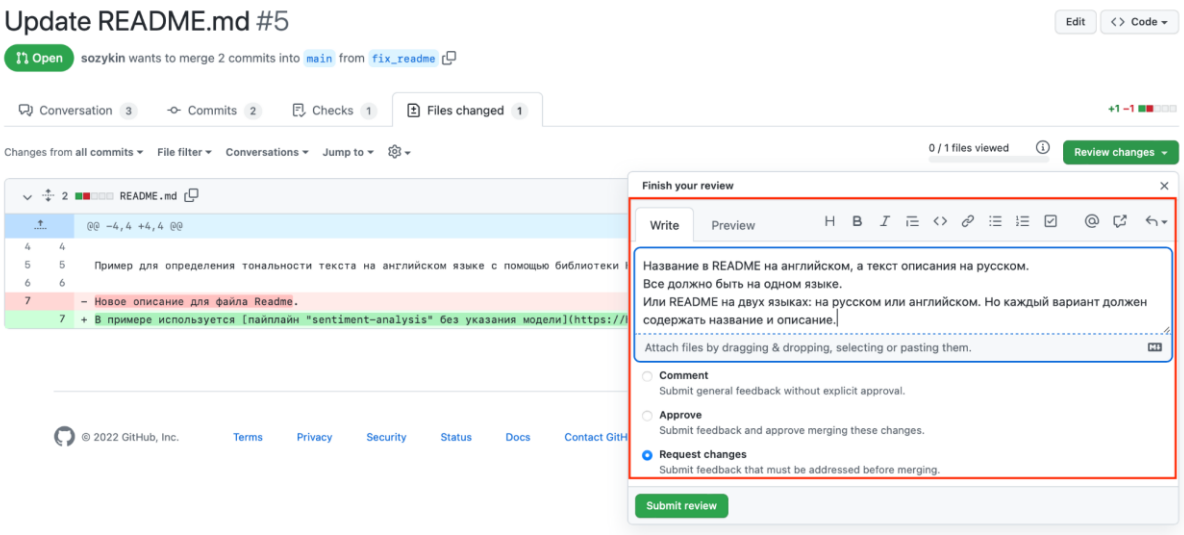


Рис. 12. Отправка запроса на изменение в код-ревью

Запрос на изменение также будет виден в правой верхней части экрана обсуждения pull request, где указан список ревьюеров (рис. 13), и в нижней части обсуждения перед итоговой информацией о том, готов ли pull request к объединению (рис. 14).

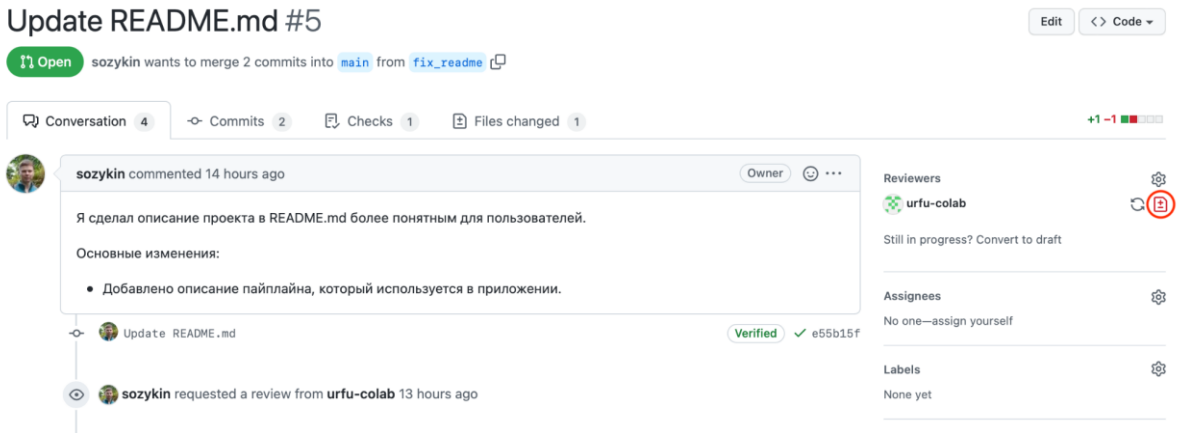


Рис. 13. Информаций о проведенном ревью с запросом изменений

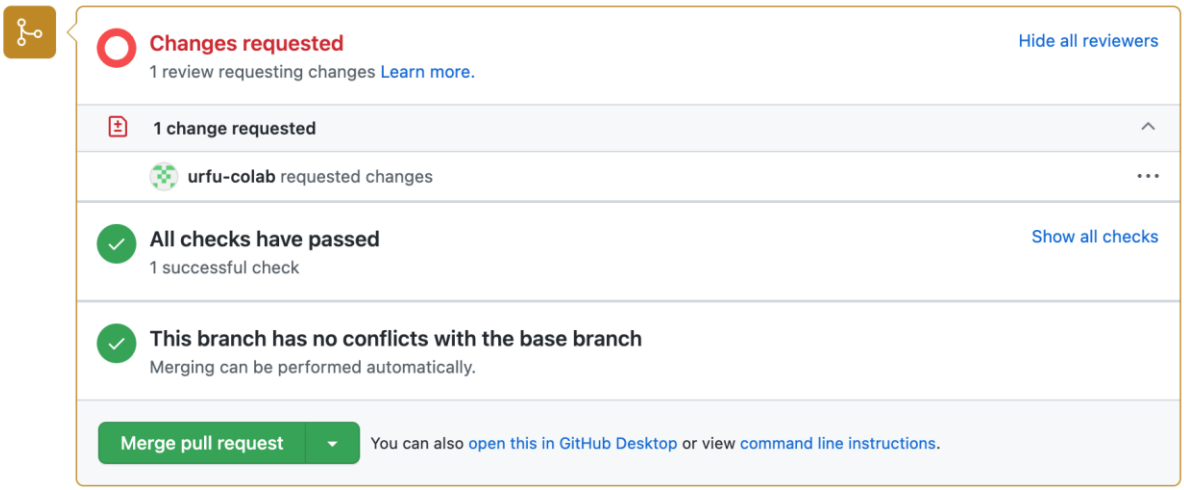


Рис. 14. Итоговая информация обсуждения pull request с запросом на изменения

Как вы можете видеть на рис. 14, наличие запроса на изменения в результате код-ревью не мешает провести объединение pull request и основной версии кода. Запретить объединять pull request, к которым имеются замечания ревьюеров, можно с помощью механизма защищенных веток.

Защищенные ветки

GitHub, как и другие подобные системы, позволяет ограничить некоторые операции, которые можно выполнять с определенными, наиболее критичными ветками в репозитории (main, master и др.). Это делается для того, чтобы защититься от потери работоспособности приложения из-за непродуманных изменений в коде. Вот примеры ограничений, которые поддерживает GitHub:

- Запрет выполнения commit в ветку (для внесения изменений необходимо создавать pull request).
- Запрет объединения кода в pull request и кода в ветке без получения одобрения на код-ревью.
- Запрет объединения кода в pull request и кода в ветке без успешного прохождения проверок (тесты, линтер и т.п.).
- Запрет на объединения кода в pull request и кода в ветке в случае проблем при развертывании приложения в заданное окружение (development, test и т.п.).

Чтобы включить защиту веток нужно выбрать закладку “Settings”, в левом меню выбрать пункт “Branches”, и нажать кнопку “Add rule” (рис. 15).

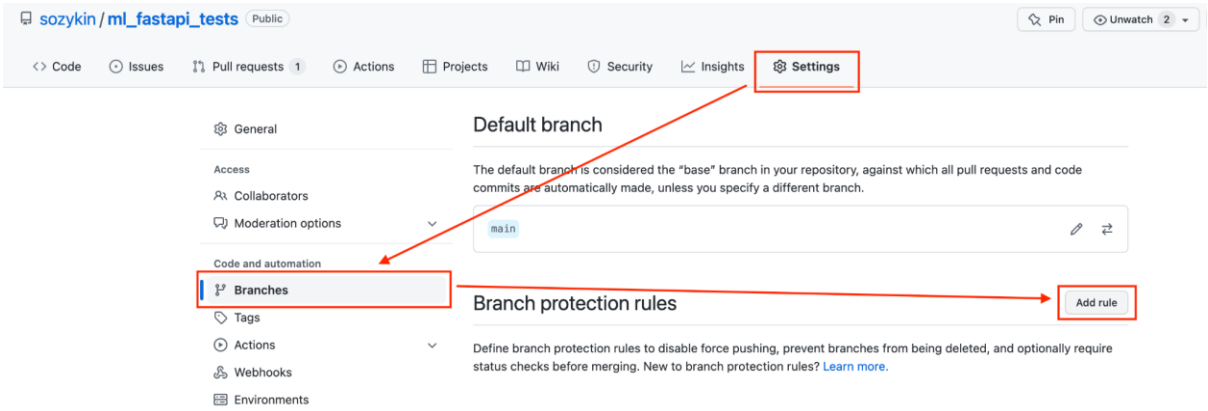


Рис. 15. Создание правил защиты веток

Давайте создадим правило, которое позволит изменять ветку main только после создания pull request, проведения код-ревью и получения одобрения от двух ревьюеров. Для этого нужно:

1. Нажать на кнопку “Add rule” в окне создания правил защиты веток (рис. 15).
2. В появившемся окне (рис. 16), написать название ветки, для которой мы хотим создать правило: main.
3. Поставить галочку напротив пункта “Require a pull request before merging” (перед объединением требуется pull request).
4. Поставить галочку напротив пункта “Require approvals” (требуется одобрение на код-ревью).
5. Из выпадающего списка в пункте “Required number of approvals before merging” (необходимое количество одобрение перед объединением) выбрать значение 2.
6. Нажать на кнопку “Create”. Правило будет создано.

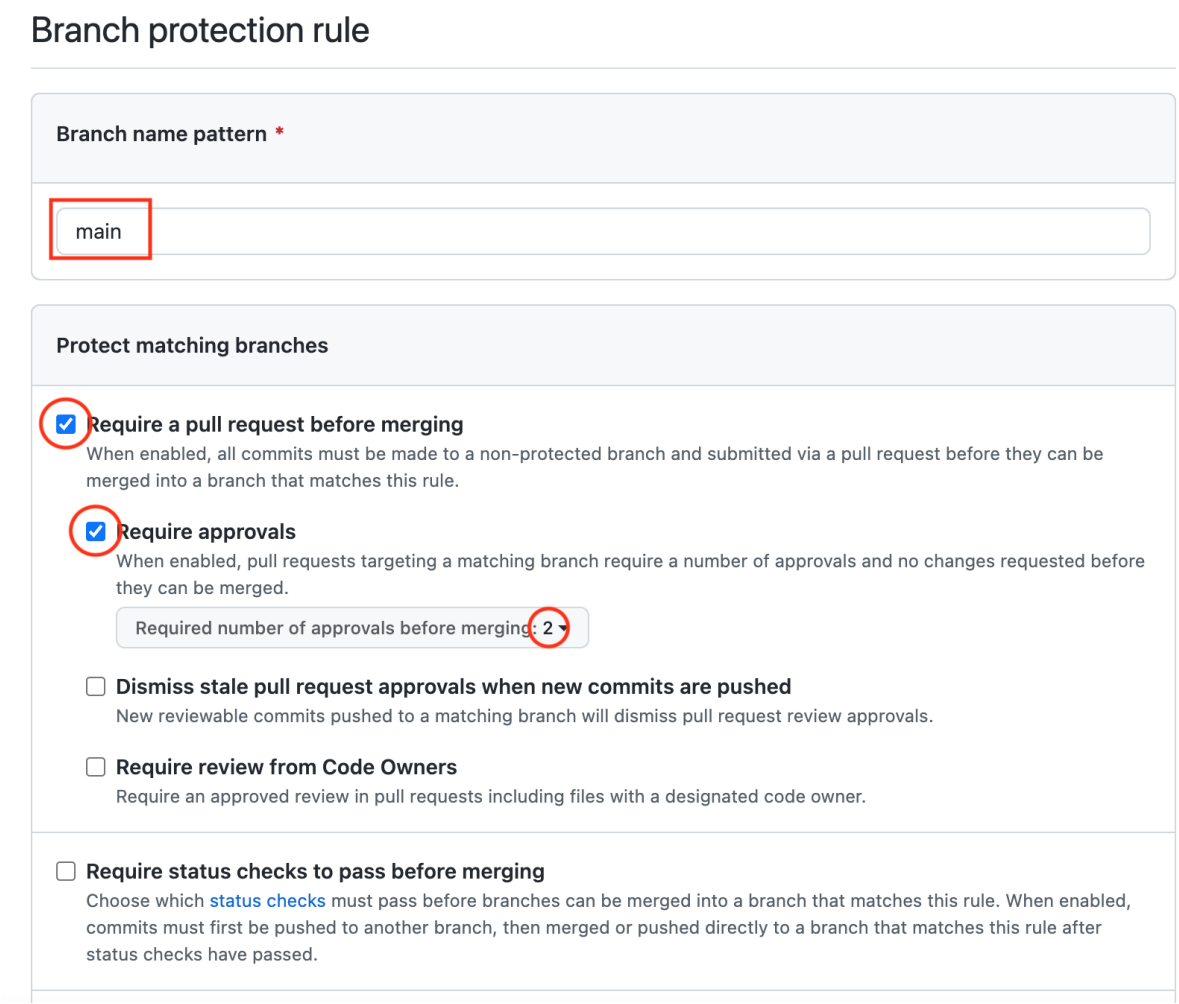


Рис. 16. Правило защиты ветки, для изменения которой необходим pull request с одобрением двух ревьюеров

После создания правила защиты ветки мы можем перейти в созданный ранее pull request и увидим, что объединение заблокировано из-за наличия запроса на изменения по результатам код-ревью (рис. 17). Также появилась подсказка, что объединение может быть выполнено автоматически, когда запрос на изменение будет обработан и удастся получить одобрение от ревьюера.

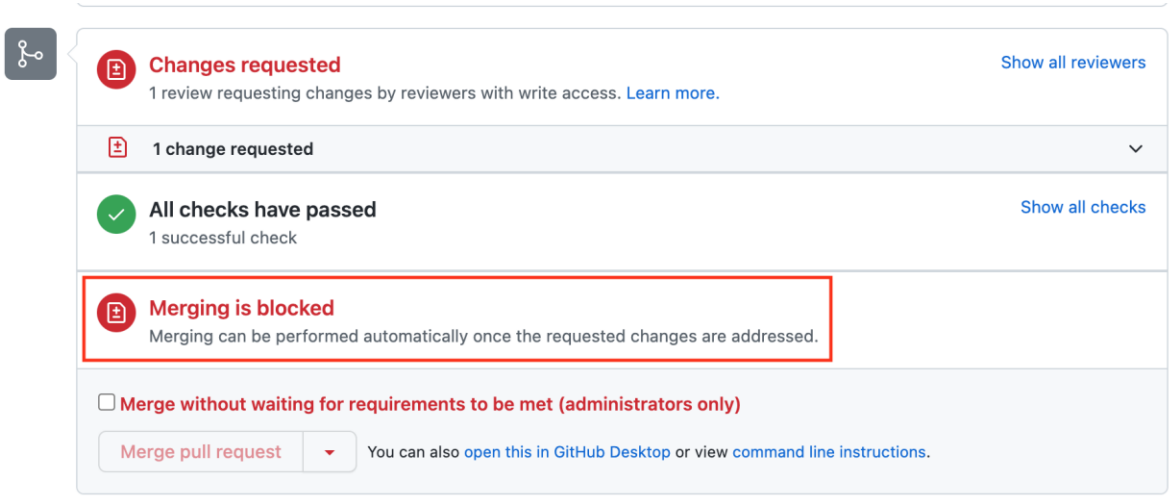


Рис. 17. Объединение кода в pull request с кодом в ветке заблокировано из-за наличия запроса на изменения по результатам код-ревью.

Итоги:

- GitHub содержит бесплатные инструменты для проведения код-ревью.
- Код-ревью на GitHub проводиться для pull request.
- Ревьюер может написать комментарии к отдельным строкам кода, а также одобрить код в целом или запросить изменение.
- При одобрении кода он может быть объединен с основной версией кода.
- При запросе на изменение объединение выполнять не желательно, пока все замечания к коду не будут устранены.
- Механизм защищенных веток в GitHub позволяет запретить объединение кода в pull request с основной версией кода до получения нужного количества одобрений от ревьюеров.

Модуль № 5. Юнит № 3. Организация процесса код-ревью в команде разработки

GitHub и другие похожие системы предоставляют нам удобные инструменты для проведения код-ревью. Однако их можно использовать различными способами. В этом разделе мы рассмотрим, как можно эффективно организовать процесс код-ревью в команде разработки.

Кто проводит код-ревью

Один из важнейших вопросов при организации командной работы: кто в команде проводит код-ревью и выполняет объединение pull request с основной версией кода? Есть несколько распространенных вариантов.

1. **Код-ревью проводит тимлид.** Он же принимает решение, какой pull request может быть объединен с основной версией кода и выполняет это объединение. Как правило тимлид один из самых сильных разработчиков в команде, поэтому он может качественно провести ревью кода, созданного остальными разработчиками. При этом в процессе код-ревью он может выступать в роли наставника других разработчиков в команде. Так как тимлид отвечает за качество работы команды в целом, то он будет очень ответственно принимать решение о готовности кода.
   Такой подход хорошо работает для небольших команд, но с ростом команды и проекта начинают возникать трудности.
   - Тимлид оказывается единственным человеком, который в курсе того, как работает система в целом. Если он по какой-то причине не сможет работать, то проект может остановиться.
   - Забирая всю ответственность за качество кода себе, тимлид лишает инициативы и ответственности остальных участников команды. Для творческой деятельности, к которой относится разработка программного обеспечения, это не самые эффективные условия работы. В результате моральное состояние остальных членов команды может снизиться.
   - С ростом проекта тимлид все больше времени будет вынужден уделять код-ревью и у него может не остаться времени для выполнения других дел.

   Один из вариантов реализации такого подхода: код-ревью в дополнение к тим-лиду проводит еще один или несколько ведущих разработчиков. Это может решить проблемы с нагрузкой, и частично с распространением знаний, но не решает проблему морали. Большая часть разработчиков в команде по прежнему будут чувствовать себя винтиками в машине с минимальными возможностями повлиять на результат.
2. **Код-ревью проводят все участники команды.** Эта ситуация характерна для больших команд, работающих над крупными проектами. Вовлечение всей команды в процесс код-ревью помогает распространить знания в команде и избавиться от рисков ухода единственного разработчика, который разбирается в определенной части системы. При этом нагрузка на проведение код-ревью

равномерно распределяется между всеми участниками. С другой стороны, так как код проверяют не только опытные специалисты, но и разработчики среднего и начального уровня, то качество проверки может снизиться. В результате в продуктивное использование будут попадать больше кода с ошибками, который будут вызывать проблемы у пользователей.

Принятие решения о том, что pull request может быть объединен с основной версией кода может быть организовано двумя основными способами:

- Один из ведущих разработчиков, включая тимлида, выполняет объединение pull request вручную. Такой разработчики увидит, что pull request набрал достаточное количество одобрений на код-ревью, и выполнит объединение.
- Автоматическое объединение pull request и основной версии кода после того, как pull request набрал достаточное количество одобрений на код-ревью. Такую возможность сейчас предоставляют многие

3. **“Умный” выбор ревьюеров.** Этот вариант похож на предыдущий, при котором ревью кода выполняет вся команда, но только ревьюер выбирается не случайным образом и не по выбору автора pull request, а в соответствии с вкладом участников команды именно в тот фрагмент кода, который был изменен. Современные системы командной разработки умеют определять, кто был автором файла с кодом, кто вносил в него изменения чаще всего и кто вносил самые существенные изменения. Используя такие инструменты можно подобрать ревьюера, который лучше всего разбирается в том участке кода, в котором выполняется модификация, и поэтому лучше других может оценить изменения. В GitHub для этой цели можно использовать раздел “Blame view”. Более того, некоторые инструменты могут автоматически подбирать наиболее вовлеченных в нужный участок кода ревьюеров.

   При реализации такой стратегии код-ревью не нужно забывать о необходимости распространения знаний в команде. Поэтому в дополнение к одному или двум разработчикам, которые хорошо разбираются в изменяемом участке кода, хорошо бы добавить в ревьюеры человека, который не знаком с ним. Тогда у него появится возможность разобраться в новом для себя участке кода системы и в будущем поддержать коллег при необходимости работать с этим кодом.

Что делать в процессе код-ревью

Вас назначили проводить код-ревью для pull request, вы знаете, как применять необходимые инструменты, но что именно вы будете проверять в коде? Какие замечания вы будете писать и на какие темы? Вот краткие рекомендации:

1. **Оцените, что делает изменяемый код в целом.** Это особенно полезно делать, если вы не очень хорошо разбираетесь в данном фрагменте системы. Часто бывает очень сложно адекватно оценить предлагаемые изменения без понимания контекста.
2. **Прочитайте описание изменений в pull request** и сравните с тем, как эти изменения реализованы в коде. Все ли изменения совпадают? Нет ли ошибки при реализации изменений?
3. **Проверьте, есть ли тесты для новой функциональности.** Оцените, достаточно ли их, не упустил ли разработчик каких-либо крайних случаев? Если в вашей системе тесты не запускаются автоматически при создании pull request, то запустите эти тесты вручную (здесь нужно задать себе вопрос, почему запуск тестов не автоматизирован и попытаться решить эту проблему).
4. **Оцените качество кода** согласно рекомендациям, рассмотренным в предыдущем разделе. Как мы уже обсуждали, качество кода не всегда должно быть совершенным. Технический долг вполне допустим, но он должен быть осознанным. В идеальном случае технический долг должен быть отражен в файле с кодом и в системе отслеживания задач.
5. **Оцените архитектуру решения** и соответствие ее принятым в команде подходам.
6. **Если вы можете это сделать,** то дайте советы по повторному использованию существующего кода, о котором может не знать автор, а также о наличии более эффективных методов реализации той задачи, которую решал автор кода.

Не рекомендуется писать замечания по вещам, которые могут быть обнаружены и исправлены автоматически, например, с помощью линтеров или форматтеров. Все, что может быть автоматизировано, лучше автоматизировать, чтобы снижать нагрузку на разработчиков. Тогда они смогут сконцентрироваться на реализации полезных для пользователя функций, а не на борьбу с процессом разработки.

Как писать конструктивное ревью

Мы уже рассматривали ранее, что код-ревью может вызывать конфликты в команде. Такое может произойти, если ревью будет написано в неуважительном для автора кода формате. Причем даже один человек в команде, который пишет ревью в таком стиле, может свести на нет весь полезный эффект от код ревью. К сожалению, в России подобный стиль общения у разработчиков встречается достаточно часто.

При написании ревью рекомендуется исходить из следующих предпосылок:

- Возврат кода на доработку, даже абсолютно обоснованный, эмоционально очень неприятен для любого автора кода.
- Члены вашей команды предпринимают максимальные усилия для того, чтобы написать максимально качественный код (если это не так, то эту проблему необходимо решать другими средствами, код-ревью в таких ситуациях бесполезно).

Исходя из этих предпосылок, базовое правило составления конструктивного ревью: обсуждайте проблему с кодом, а не личность человека, который написал этот код.

Вот несколько рекомендаций по тому, как составить эффективное ревью, которое поможет вашему коллеге улучшить свой код:

- Замечания к коду должны быть четкими и понятными. Представьте, что в качестве ревью вам прислали три эмодзи с веселыми какашками. Как вы поймете, что именно нужно изменить в коде и каким образом? Лучше всего создавать комментарии, привязанные к конкретным строкам кода, в комментариях писать, что не так с данным кодом и предлагать возможные варианты решения.
- Если вы видите потенциальную проблему в коде, но сомневаетесь в ее актуальности, то вы можете в тексте ревью сформулировать свои опасения в виде вопроса автору. Не забыл ли он о чем-либо, что кажется вам важным? Не повлияет ли выбранное решение на используемый объем памяти? Вполне вероятно, что если проблема действительно есть, то автор кода быстрее найдет ее причину и сможет исправить.
- Замечания пишите в уважительной и спокойной манере.
- Пишите замечания так, как будто адресуете их равному себе. Даже если вы проверяете код разработчика среднего уровня или новичка.

- Помните, что у всех собственный подход к разработке. Если вы видите в коде работающее решение, но понимаете, что вы бы реализовали это по другому, то не нужно настаивать на переработке только по тому, что решение отличается от вашего (если, конечно, качество решений при этом примерно одинаковое).

Итоги:

- Важный вопрос при организации процесса разработки: кто именно выполняет код-ревью в команде. Это может быть тимлид (или ограниченное число ведущих разработчиков), вся команда, или специальным образом выбранные ревьюеры, наиболее подходящие именно для данного pull request.
- Второй важный вопрос: кто и каким образом принимает решение о возможности объединения pull request с основной версией кода. Это может делать вручную тимлид (ограниченное число ведущих разработчиков) или автоматически с помощью инструментов командной разработки и наличии определенного количества одобрений от ревьюеров.
- В процессе код-ревью попробуйте найти ошибки в коде, оцените объем и качество тестов, качество кода и архитектурных решений.
- Возвращение кода на доработку эмоционально очень неприятно автору. Постарайтесь написать ревью конструктивно и понятно, чтобы автор быстро смог исправить все замечания. Этим вы не просто сделаете человеку приятное, но и повысите скорость разработки.

Модуль № 5. Юнит № 4. Код-ревью в процессе CI/CD

Код-ревью – это один из важных этапов процесса CI/CD, который мы рассматриваем в курсе программной инженерии. Теперь мы изучили все отдельные этапы и готовы взглянуть на процесс в целом. Итак, процесс разработки с помощью методики CI/CD выглядит следующим образом:

1. Разработчик создает локальную копию репозитория с кодом проекта. При необходимости можно сделать fork интересующего репозитория и уже для него создавать локальную копию.
2. В локальном репозитории создается новая ветка для реализации нужной функциональности.

3. Разработчик создает код, реализующий нужную функциональность, в новой ветке.
4. Разработчик пишет тесты для новой функциональности и убеждается, что тесты проходят успешно. Этот шаг и предыдущий могут быть поменяны местами если используется разработка через тестирование (Test Driven Development).
5. Выполняется локальный запуск линтера для проверки стиля и качества кода.
6. Внесенные изменения в код коммитятся в локальном репозитории и отправляются в центральный репозиторий (при наличии прав доступа у разработчика).
7. Когда разработчик завершил работу над новой функциональность и проверил корректность кода локально на тестах и с помощью линтера, может быть создан запрос на объединения кода в новой ветке с основной версией кода – pull request.
8. Для pull request выполняются автоматические проверки: запускаются тесты и линтер в централизованном репозитории.
9. В случае успешного выполнения проверок выполняется развертывание кода из pull request в Review окружении.
10. Другие разработчики из команды проводят ревью кода. Они могут просматривать результаты запуска тестов и линтера, а также работать с предлагаемой версией приложения в ревью окружении.
11. При наличии замечаний к коду, ревьюеры отправляют запрос на изменение с указанием, что и каким образом нужно исправить. Разработчик видит эти запросы и устраняет замечания.
12. После устранения всех замечаний ревьюеры одобряют изменения в коде.
13. Код из новой ветки в pull request объединяется с основной версией кода в приложении. При этом снова запускаются тесты и линтер. В случае успеха, pull request закрывается. Развернутая версия приложения в окружении Review удаляется.
14. Объединенная версия кода разворачивается в окружении Testing или Staging.
15. В случае успешного тестирования приложение может быть перенесено в продуктивное окружение.

Таким образом, вы можете видеть, что промышленная разработка крупных программных систем – сложный многоэтапный процесс, в котором участвует вся команда разработки. Разработка включает не только создание

кода, но и написание тестов, проведение код-ревью, настройку автоматического запуска тестов и линтера в процессе CI и развертывания в различные окружения в процессе CD. Такой подход позволяет поставлять пользователям протестированное программное обеспечение с высоким качеством кода и делать это быстро за счет автоматизации рутинных операций.

Итоги

- Код-ревью – это одна их техник совместной разработки, при которой один человек разрабатывает код, а другой (или другие) проверяет его качество.
- В результате проведения код-ревью может быть принято решение об одобрении кода или возвращении его на доработку.
- Использование код-ревью снижает количество ошибок, повышает качество кода, обеспечивает возможность распространения знаний и обучения новичков.
- Современные системы системы разработки, такие как GitHub, содержат инструменты для проведения код-ревью.
- Процесс проведения код-ревью может быть организован различными способами. Код-ревью может проводить тимлид или ведущие разработчики, а также все члены команды. Разные сценарии организации процесса приводят к разным положительным и отрицательным эффектам.
- Возвращение кода на доработку в результате ревью является эмоционально неприятным для автора кода.
- Неконструктивный подход к написанию ревью может приводить к конфликтам в команде.

Итоговый тест

Что такое код-ревью:

1. **Одна из техник совместной разработки, при которой один человек разрабатывает код, а другой проверяет его качество.**
2. Одна из техник совместной разработки, при которой два или больше человек совместно за одним компьютером разрабатывают код.

3. Одна из техник совместной разработки, при которой на первом этапе разработчик создает тесты, и уже потом код, который реализует необходимую функциональность.
4. Одна из техник совместной разработки, при которой изменяется внутренняя структура кода, но не функциональность, которую он реализует.

Какие преимущества обеспечивает код-ревью:

1. Повышение скорости разработки.
2. **Распространение знаний в команде.**
3. Отсутствие необходимости тратить время на написание тестов.
4. **Повышение стабильности работы приложения и качества кода.**

Какие недостатки у код-ревью:

1. Распространение знаний снижает защищенность рабочего места разработчика.
2. Повышение качества кода.
3. **Скорость разработки снижается.**
4. **В команде могут возникнуть конфликты.**

Ранее вы не использовали в команде код-ревью и хотите его внедрить. Однако ваш менеджер выступает против. Какие аргументы можно использовать, чтобы убедить менеджера в необходимости код-ревью?

1. Код-ревью будет занимать от 5 до 20% времени разработчиков.
2. **Время, выделенное на проведение код-ревью, позволит сократить время на поиск и исправление ошибок в приложении, которые дошли до пользователей.**
3. В ревью очень удобно писать гадости разработчикам, которые тебе не нравятся.
4. **Распространение знаний в процессе код-ревью позволит обеспечить взаимозаменяемость людей в команде.**

Новый начинающий разработчик в команде отказывается проводить код-ревью. Он объясняет свою позицию тем, что у него слишком мало опыта чтобы оценивать код ведущих разработчиков в команде. Какие доводы

можно привести, чтобы убедить начинающего разработчика проводить ревью?

1. **Проводя код-ревью новый разработчик может познакомиться с разделами системы, в создании которых он сам не принимает участия.**
2. Проводя код-ревью новый разработчик может постоянно отправлять запросы на изменения тем разработчикам, которые пишут много замечаний ему.
3. Если начинающий разработчик не будет проводить код-ревью, то его могут уволить.
4. **Проводя код-ревью новый разработчик может повысить свою значимость в команде, так как берет на себя часть ответственности за качество кода.**

Практическое задание

В команде из 2-4 человек внесите изменения в репозиторий на GitHub и проведите код-ревью изменений. Для этого:

1. Выберите репозиторий на GitHub, с которым вы будете работать.
2. Добавьте всех участников команды в качестве коллабораторов в этот репозиторий.
3. Каждый участник команды должен создать новую ветку в репозитории и внести в нее изменения.
4. Каждый участник команды должен создать pull request для объединения кода из новой ветки в ветку main.
5. Для каждого pull request минимум два коллаборатора должны провести код-ревью.
6. На основе результатов проведения код-ревью хозяин репозитория выполняет объединение pull request.

   Для сдачи задания нужно прислать на платформу ссылку на репозиторий GitHub, в котором выполнено код-ревью.

Список источников

1. GitHub. Code Review – https://github.com/features/code-review

2. Github. About protected branches – https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/defining-the-mergeability-of-pull-requests/about-protected-branches
3. Giuliana Carullo. Implementing Effective Code Reviews. How to Build and Maintain Clean Code – https://link.springer.com/book/10.1007/978-1-4842-6162-0

Рекомендуемая литература

- Reviewing pull requests – https://lab.github.com/githubtraining/reviewing-pull-requests
- Code review: вы делаете это неправильно – https://habr.com/ru/company/badoo/blog/413965/
- Что такое холивар: 6 ярких примеров из мира технологий и разработки – https://ru.hexlet.io/blog/posts/chto-takoe-holivar-7-yarkih-primerov-iz-mira-tehnologiy-i-razrabotki