**Доклад: Автоматизация тестирования с использованием GitHub Actions**

**1. Введение: Что такое GitHub Actions?**

GitHub Actions — это мощная платформа автоматизации, тесно интегрированная в экосистему GitHub. Она позволяет автоматизировать практически любые задачи в рамках жизненного цикла разработки программного обеспечения (ПО). По сути, это набор инструментов и сервисов, который дает возможность создавать собственные рабочие процессы (workflows) для CI/CD (непрерывной интеграции и непрерывной доставки/развертывания), автоматического тестирования, деплоймента приложений, отправки уведомлений и многого другого.

В современной разработке GitHub Actions играет ключевую роль. Платформа позволяет значительно увеличить скорость разработки за счет автоматизации рутинных задач. Это не только экономит время команды, но и повышает качество кода, так как ошибки могут быть выявлены на самых ранних этапах благодаря автоматическим проверкам и тестам.

Сервис GitHub Actions был анонсирован компанией GitHub в 2018 году. С тех пор он прошел большой путь развития, постоянно обогащаясь новыми функциями, интеграциями и становясь все более популярным инструментом среди разработчиков и DevOps-инженеров по всему миру.

Платформа предоставляет широкий спектр функций:

* Автоматизация процессов CI/CD: сборка, тестирование, развертывание.
* Раннее выявление проблем через автоматическое тестирование кода.
* Автоматизированный деплоймент в различные среды (разработка, тестирование, продакшен).
* Уведомления о результатах выполнения рабочих процессов.

**3. Начало работы: GitHub для тестировщиков**

* **Зачем тестировщику Git и GitHub?**
  + **Ручные тестировщики (QA Manual):** на первый взгляд, глубокое знание Git/GitHub может показаться необязательным. Тестирование проводится вручную, без программирования. Однако базовое понимание Git может быть полезным для:
    - Взаимодействия с командой разработчиков (понимание веток, коммитов).
    - Доступа к коду, документации, тест-кейсам, хранящимся в репозитории.
    - Понимания, какие изменения были внесены в той или иной версии.
    - Иногда знание Git указывается в вакансиях как плюс, даже если на практике оно используется редко.
  + **Автоматизаторы тестирования (QA Automation):** для автоматизаторов Git и GitHub – это практически незаменимые инструменты. Тесты – это код, и этот код нужно:
    - Хранить централизованно.
    - Версионировать (отслеживать изменения, откатываться к предыдущим версиям).
    - Предоставлять доступ коллегам для совместной работы.
    - Интегрировать с системами CI/CD (например, GitHub Actions) для автоматического запуска.

Чтобы эффективно использовать GitHub Actions, важно понимать основные концепции:

* **Workflows (Рабочие процессы):** Это автоматизированные процедуры, которые вы определяете в своем репозитории. Они состоят из одного или нескольких заданий (jobs) и могут запускаться по определенным событиям (events). Workflows определяются в YAML-файлах, хранящихся в директории .github/workflows вашего репозитория.
* **Events (События):** Это определенные действия в вашем репозитории, которые могут инициировать запуск workflow. Примеры: push (отправка коммитов), pull\_request (создание или обновление запроса на слияние), schedule (запуск по расписанию), ручной запуск (workflow\_dispatch) и многие другие.
* **Jobs (Задания):** Набор шагов (steps), выполняемых на одном и том же исполнителе (runner). Задания могут выполняться параллельно (по умолчанию) или последовательно, если настроить зависимости между ними. Каждое задание выполняется в свежей виртуальной среде.
* **Steps (Шаги):** Отдельная задача, выполняемая в рамках задания. Шаг может быть либо запуском команды оболочки (shell command), либо использованием готового действия (action). Шаги внутри одного задания выполняются последовательно.
* **Runners (Исполнители):** Это серверы, на которых выполняются ваши workflows. GitHub предоставляет готовых исполнителей под управлением Ubuntu Linux, Windows и macOS (GitHub-hosted runners). Также вы можете настроить собственные серверы в качестве исполнителей (self-hosted runners).
* **Actions (Действия):** Это переиспользуемые фрагменты кода, которые выполняют определенные задачи. Вы можете создавать свои собственные действия или использовать тысячи готовых действий, доступных в GitHub Marketplace, созданных сообществом или GitHub. Примеры популярных действий: actions/checkout (для получения кода из репозитория), actions/setup-node (для настройки среды Node.js), actions/setup-python (для настройки среды Python).
* **Структура Workflow: YAML-синтаксис**  
  Workflows описываются с использованием синтаксиса YAML (YAML Ain't Markup Language) – это удобный для чтения формат сериализации данных.
* **Базовая настройка Git и GitHub**  
  Прежде чем использовать GitHub Actions для автотестов, необходимо настроить Git и GitHub. В примере из документа используется Linux, но шаги аналогичны для других ОС (для Windows рекомендуется WSL - Windows Subsystem for Linux).
  + **Установите Git:** Загрузите и установите Git с официального сайта.
  + **Зарегистрируйтесь на GitHub:** Создайте аккаунт на github.com.
  + **Настройте Email Privacy (Рекомендуется):** В настройках GitHub (Emails) включите опции "Keep my email addresses private" и "Block command line pushes that expose my email". Это скроет ваш реальный email и предоставит приватный noreply-адрес.
  + **Настройте Git локально:** Откройте терминал и выполните команды, указав ваше имя и email (используйте приватный email с GitHub, если включили опцию выше):

git config --global user.name "ваше\_имя"

git config --global user.email "ваш\_приватный\_или\_реальный\_email"

# Установка ветки по умолчанию (main вместо устаревшего master)

git config --global init.defaultBranch main

# Включение цветного вывода (опционально)

git config --global color.ui auto

# Проверка настроек

git config --list --global

* + **Создание и настройка SSH-ключа:** SSH-ключ позволяет безопасно подключаться к GitHub без ввода логина и пароля при каждой операции push или pull.
  + **Проверьте наличие ключа:** ls ~/.ssh/id\_ed25519.pub. Если файла нет, создайте его.
  + **Создайте ключ:** ssh-keygen -t ed25519 -C "ваш\_email@example.com". Нажимайте Enter на все запросы (куда сохранить, пароль – пароль можно оставить пустым для простоты, но для большей безопасности лучше задать).
  + **Скопируйте публичный ключ:** cat ~/.ssh/id\_ed25519.pub. Скопируйте все содержимое, начиная с ssh-ed25519 и заканчивая вашим email.
  + **Добавьте ключ в GitHub:** Зайдите в Настройки -> SSH and GPG keys -> New SSH key. Придумайте название (Title, например, "My Laptop Ubuntu"), вставьте скопированный ключ в поле "Key", тип ключа оставьте "Authentication Key" и нажмите "Add SSH key". (стр. 11-12)
  + **Проверка:** Теперь вы должны иметь возможность взаимодействовать с вашими репозиториями по SSH.

**Основные операции Git** (стр. 12-13)

**Репозиторий:** Хранилище кода и истории его изменений. Бывают локальные (на вашем компьютере) и удаленные (на серверах GitHub).

**Клонирование (git clone):** Создание локальной копии удаленного репозитория. Используйте SSH-ссылку (вида git@github.com:username/repo.git), а не HTTPS, чтобы работал SSH-ключ. (стр. 12)

git clone git@github.com:your\_username/your\_repository.git

cd your\_repository # Переход в папку репозитория

**Проверка статуса (git status):** Показывает состояние файлов (отслеживаемые, измененные, неотслеживаемые).

**Добавление в Staging Area (git add):** Подготовка файлов к коммиту.

git add имя\_файла # Добавить конкретный файл

git add . # Добавить все измененные и новые файлы в текущей папке

**Коммит (git commit):** Фиксация изменений в локальном репозитории с осмысленным сообщением.

git commit -m "Краткое описание изменений"

**Отправка на удаленный сервер (git push):** Загрузка ваших локальных коммитов в удаленный репозиторий (например, на GitHub).

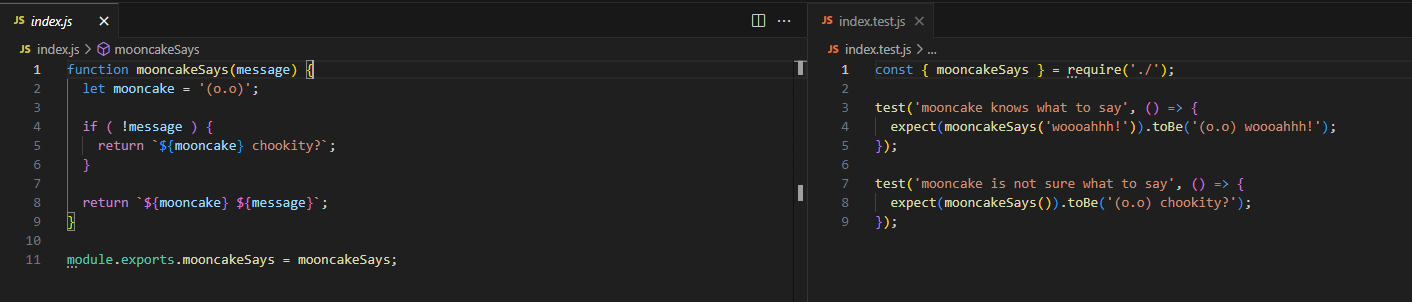
git push origin main # Отправить коммиты из локальной ветки main в удаленную main

**Получение изменений с сервера (git pull):** Загрузка изменений из удаленного репозитория в ваш локальный.

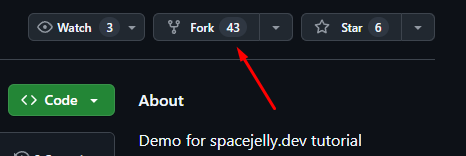
git pull origin main # Загрузить изменения из удаленной ветки main

**4. Практический пример 1: Автоматизация тестирования Node.js проекта**

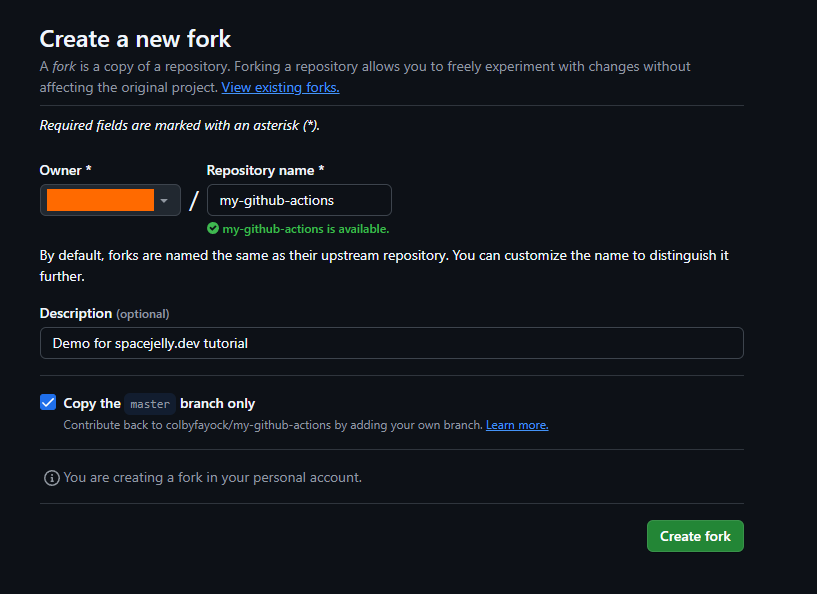
* **Подготовка проекта (Форк репозитория)**



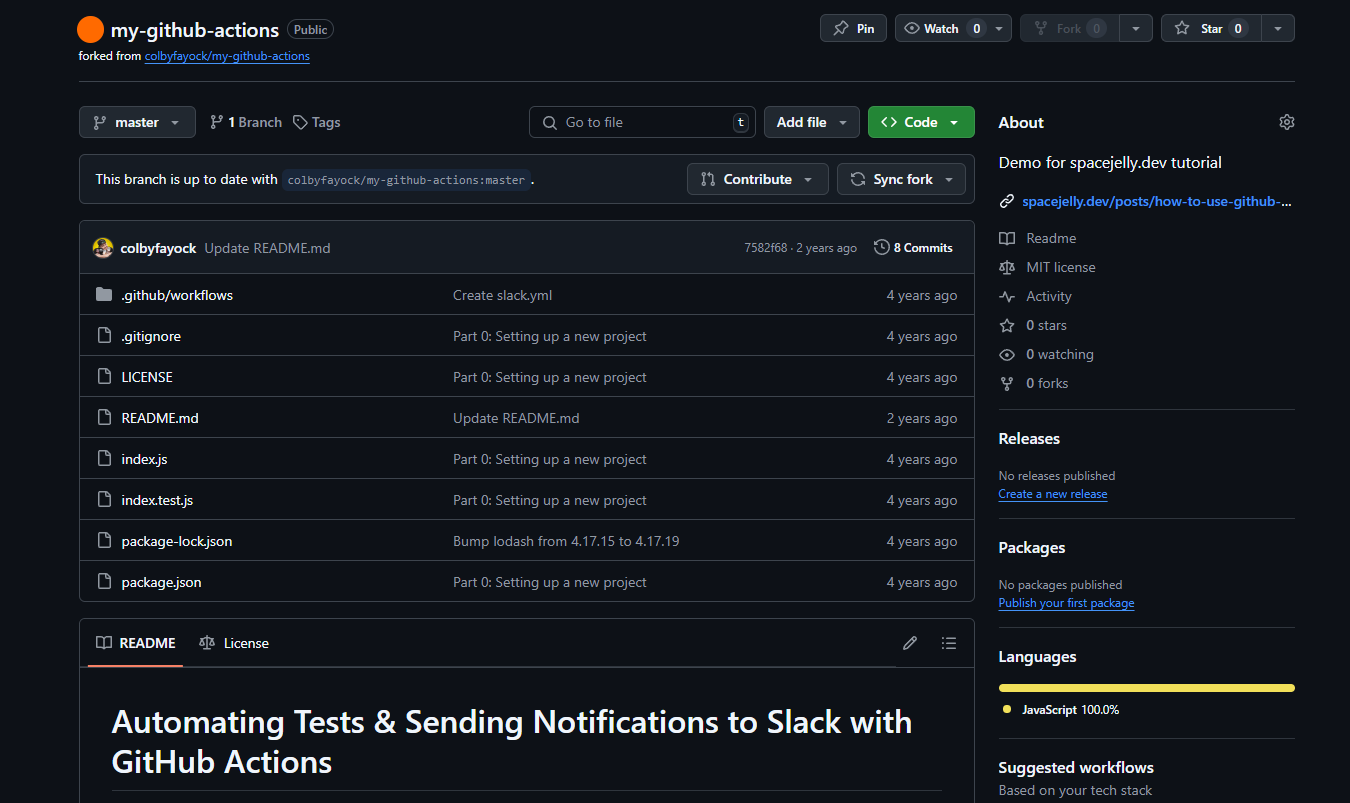
* 1. Перейдите к репозиторию.
  2. Нажмите кнопку "Fork" в правом верхнем углу. Это создаст копию репозитория в вашем GitHub аккаунте.



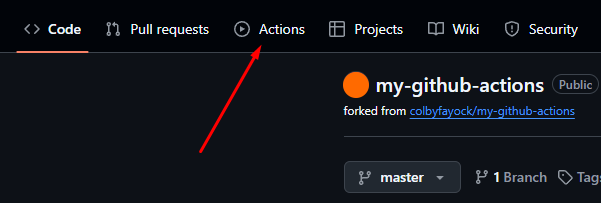
* 1. Можете оставить имя по умолчанию или изменить его. Убедитесь, что отмечена опция "Copy the master branch only". Нажмите "Create fork".



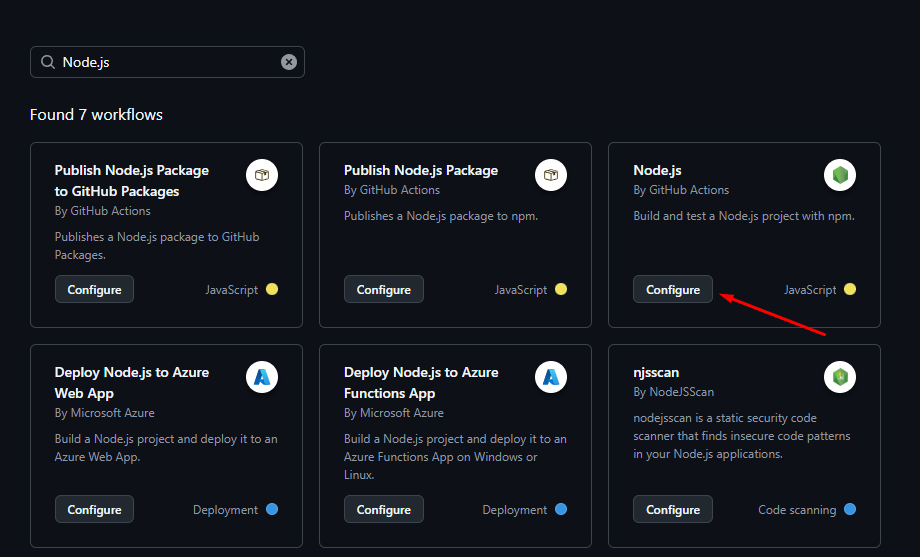
* 1. Вы будете перенаправлены на страницу вашего форка.



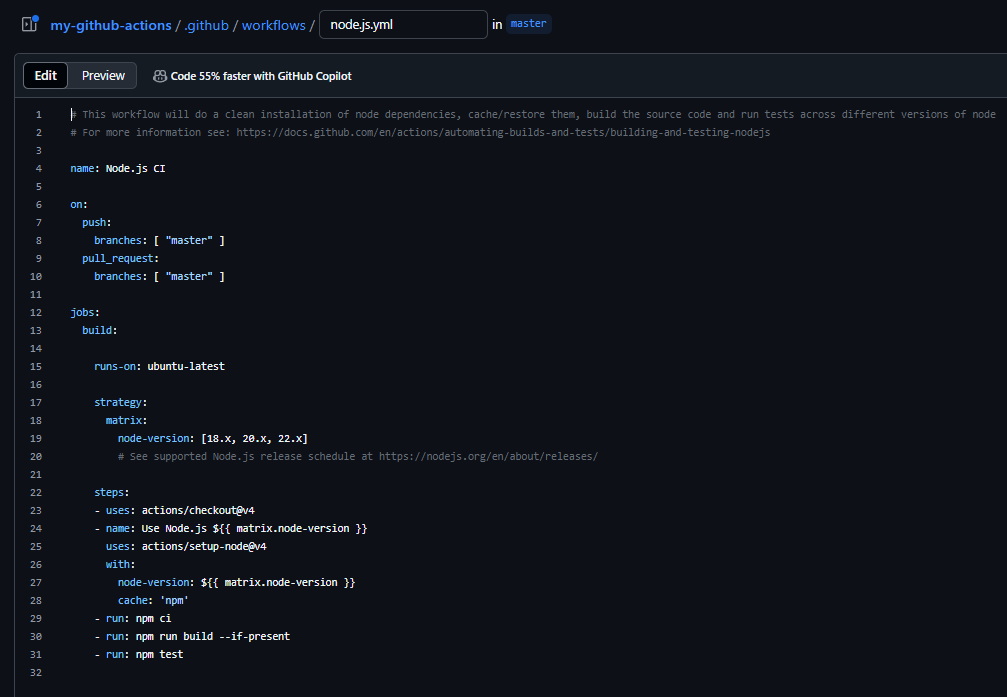
* **Создание Workflow файла (.github/workflows/tests.yml)**
  1. Перейдите на главную страницу вашего форка.
  2. Перейдите на вкладку "Actions".



* 1. GitHub может предложить готовые шаблоны. Нам нужен шаблон для Node.js. Если он не виден сразу, найдите его поиском или нажмите "set up a workflow yourself".



* 1. Откроется редактор с предзаполненным YAML-файлом node.js.yml.

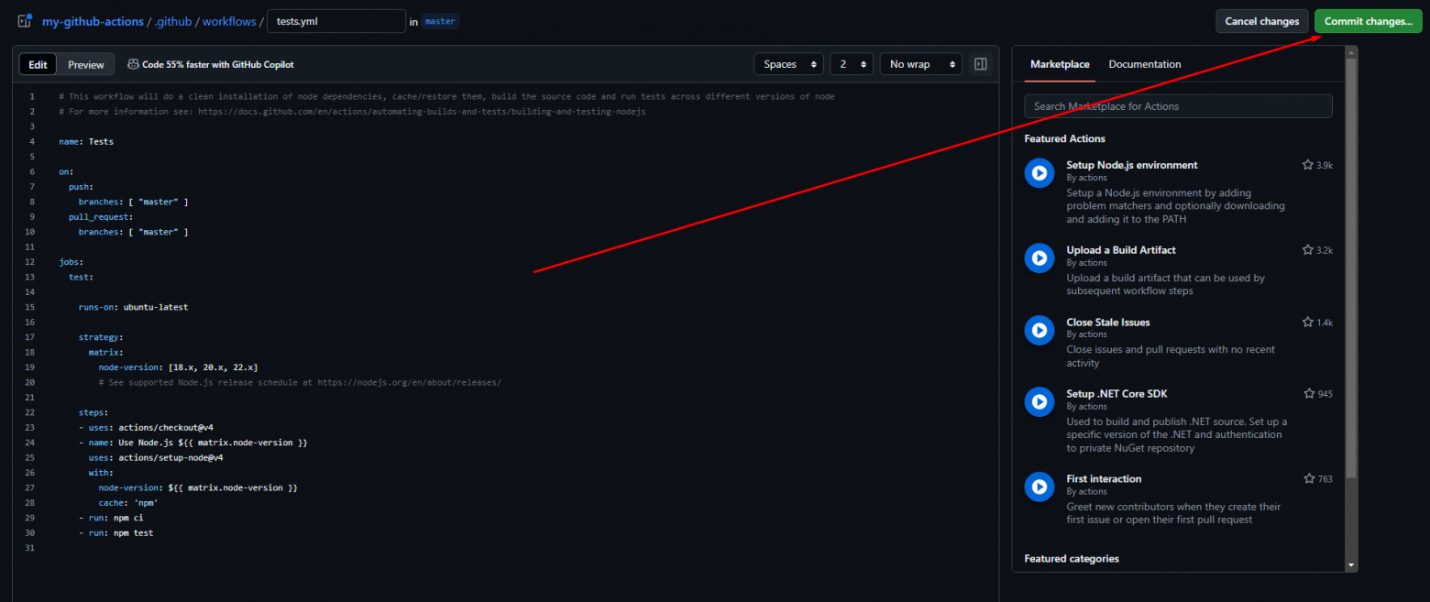


* **Разбор шагов Workflow (checkout, setup-node, npm ci, npm test)**

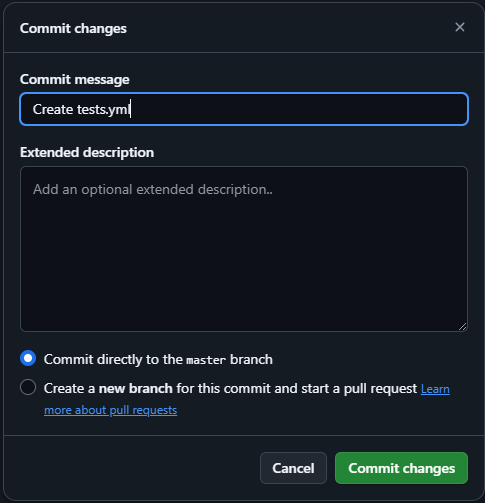
Давайте адаптируем предложенный шаблон и разберем его шаги:

1. **name: Tests**
   * Это **отображаемое имя** всего рабочего процесса. Оно будет показано в интерфейсе GitHub на вкладке "Actions" репозитория.
2. **on:**
   * Этот блок определяет **события (events)**, которые будут **запускать (триггерить)** этот рабочий процесс.
   * **push:**
     + Запускает workflow, когда кто-то делает git push (отправляет коммиты) в репозиторий.
     + **branches: [ "master" ]**: Уточняет, что workflow будет запускаться *только* при push в ветку master.
   * **pull\_request:**
     + Запускает workflow, когда кто-то создает или обновляет Pull Request (запрос на слияние).
     + **branches: [ "master" ]**: Уточняет, что workflow будет запускаться *только* для Pull Request'ов, которые нацелены на слияние в ветку master.
3. **jobs:**
   * Этот блок определяет одну или несколько **задач (jobs)**, которые будут выполняться в рамках workflow. Задачи могут выполняться параллельно (по умолчанию) или последовательно. В данном случае у нас одна задача.
4. **build:**
   * Это **идентификатор** задачи. Ты можешь назвать его как угодно.
5. **runs-on: ubuntu-latest**
   * Указывает, на какой **операционной системе** будет выполняться эта задача. GitHub предоставляет виртуальные машины (runners) с разными ОС.
   * ubuntu-latest: Означает, что будет использоваться последняя стабильная версия Ubuntu Linux, предоставляемая GitHub. Это самый распространенный выбор для многих проектов, особенно Node.js. Другие варианты включают windows-latest и macos-latest.
6. **strategy:**
   * Позволяет определить **стратегию** выполнения задачи. Часто используется для создания **матрицы сборок (build matrix)**.
   * **matrix:**
     + Определяет набор различных конфигураций, с которыми задача будет запущена. Задача будет выполнена *несколько раз*, по одному разу для каждой комбинации в матрице.
     + **node-version: [18.x, 20.x, 22.x]**:
       - Определяет переменную node-version внутри матрицы.
       - Список [18.x, 20.x, 22.x] означает, что задача test будет запущена **трижды**:
         1. С установленной последней версией Node.js 18.x.
         2. С установленной последней версией Node.js 20.x.
         3. С установленной последней версией Node.js 22.x.
       - x в версии (например, 18.x) означает "использовать последнюю доступную патч-версию в этой мажорной/минорной ветке".
7. **steps:**
   * Определяет **последовательность шагов**, которые будут выполняться в рамках *каждого* запуска задачи (т.е. для каждой версии Node.js из матрицы). Шаги выполняются по порядку.
8. **- uses: actions/checkout@v4** (Шаг 1)
   * uses:: Означает, что этот шаг использует готовое **действие (action)**, предоставленное GitHub.
   * actions/checkout@v4: Это стандартное действие от GitHub, которое **скачивает (клонирует) код твоего репозитория** на виртуальную машину (runner), где выполняется задача.
9. **- name: Use Node.js ${{ matrix.node-version }}** (Шаг 2)
   * name:: Дает шагу читаемое имя, которое будет отображаться в логах GitHub Actions.
   * ${{ matrix.node-version }}: Это синтаксис для **вставки значения переменной из матрицы**. На каждом запуске задачи сюда будет подставлена конкретная версия Node.js (18.x, 20.x или 22.x). Например, имя шага будет "Use Node.js 18.x", "Use Node.js 20.x" и т.д.
   * uses: actions/setup-node@v4: Использует другое стандартное действие от GitHub, которое **устанавливает указанную версию Node.js** на runner.
   * with:: Позволяет передать **параметры (входные данные)** в используемое действие (actions/setup-node).
   * node-version: ${{ matrix.node-version }}: Передает в action setup-node нужную версию Node.js, взятую из текущего прогона матрицы.
   * cache: 'npm': **Очень важный параметр для производительности!** Он включает **кэширование зависимостей**, установленных через npm. При первом запуске зависимости скачиваются и кэшируются. При последующих запусках action попытается восстановить зависимости из кэша, что значительно ускоряет шаг установки зависимостей.
10. **- run: npm ci** (Шаг 3)
    * run:: Означает, что этот шаг выполняет **команду в командной строке** runner'а.
    * npm ci: Это команда пакетного менеджера npm.
      + Она устанавливает зависимости проекта **строго** на основе файла package-lock.json. Это гарантирует, что в CI будут установлены *те же самые* версии зависимостей, которые зафиксированы в lock-файле.
      + Перед установкой она удаляет папку node\_modules (если она есть), обеспечивая чистую установку.
      + **Почему npm ci, а не npm install?** В CI/CD окружениях npm ci предпочтительнее, так как она быстрее и обеспечивает большую воспроизводимость сборок по сравнению с npm install, которая может обновлять минорные версии/патчи зависимостей в package-lock.json, если они соответствуют диапазонам в package.json.
11. **# - run: npm run build --if-present** (Шаг 4)
    * run: npm run build: Эта команда запустила бы скрипт с именем build, определенный в секции "scripts" твоего файла package.json. Обычно скрипт build используется для компиляции кода (например, TypeScript в JavaScript), сборки бандлов (Webpack, Rollup) и т.п.
    * --if-present: Это флаг для npm, который говорит: "выполни npm run build только в том случае, если скрипт build действительно существует в package.json". Если скрипта нет, команда просто завершится успешно, не вызвав ошибки.
12. **- run: npm test** (Шаг 5)
    * run: npm test: Выполняет стандартную команду npm для запуска тестов. Эта команда, в свою очередь, запускает скрипт с именем test, определенный в секции "scripts" твоего файла package.json. Это **ключевой шаг** данного workflow. Если эта команда завершится с ошибкой (т.е. тесты не пройдут), весь job (и, следовательно, весь workflow для этой версии Node.js) будет помечен как неудавшийся (failed).

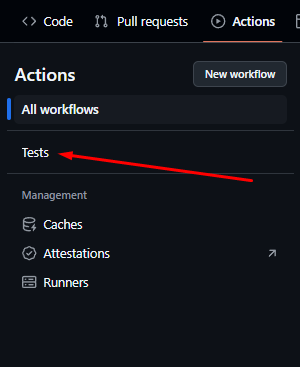
* **Запуск и анализ результатов (Успешное выполнение)**
  1. После редактирования YAML-файла нажмите "Commit changes..." в правом верхнем углу редактора.



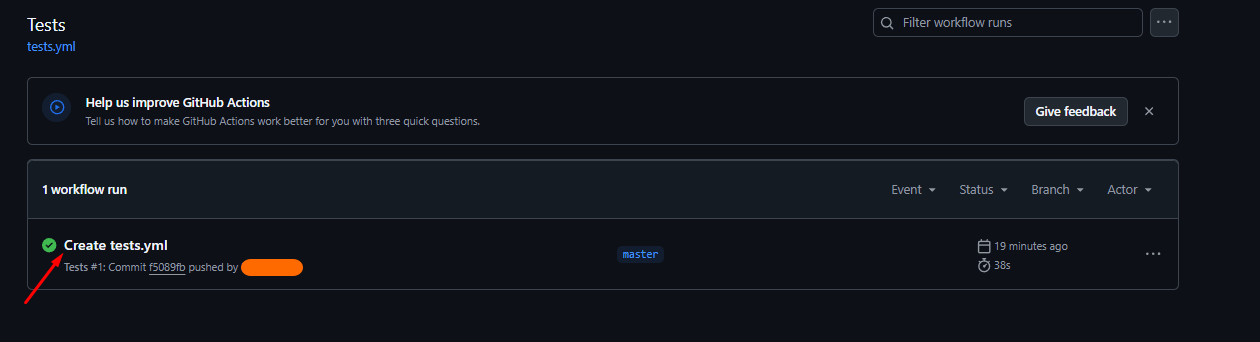
* 1. Оставьте сообщение коммита по умолчанию ("Create tests.yml") или измените его. Убедитесь, что выбрано "Commit directly to the master branch". Нажмите "Commit changes".



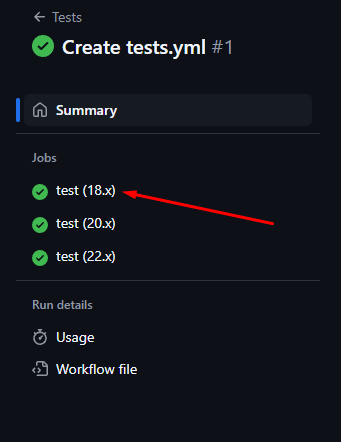
* 1. Перейдите снова на вкладку "Actions". Вы увидите ваш новый workflow "Tests" в списке слева. В основной части появится запущенный процесс.



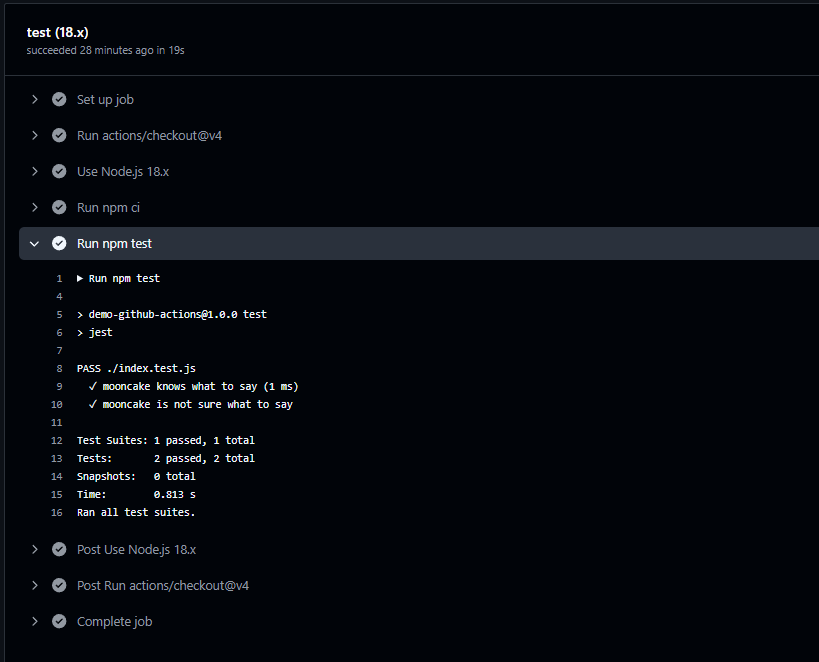
* 1. Кликните на название коммита ("Create tests.yml") в списке запусков.



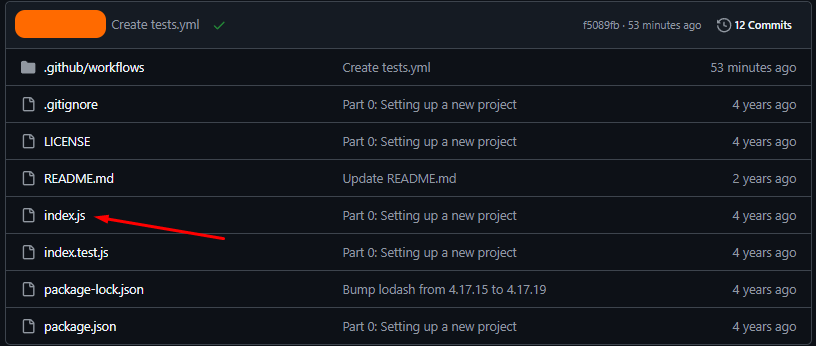
* 1. Вы увидите детали выполнения. Слева будут перечислены jobs (у нас один test), а под ним – запуски для каждой версии Node.js из матрицы (test (18.x), test (20.x), test (22.x)). Все должны быть с зелеными галочками, если тесты прошли успешно.



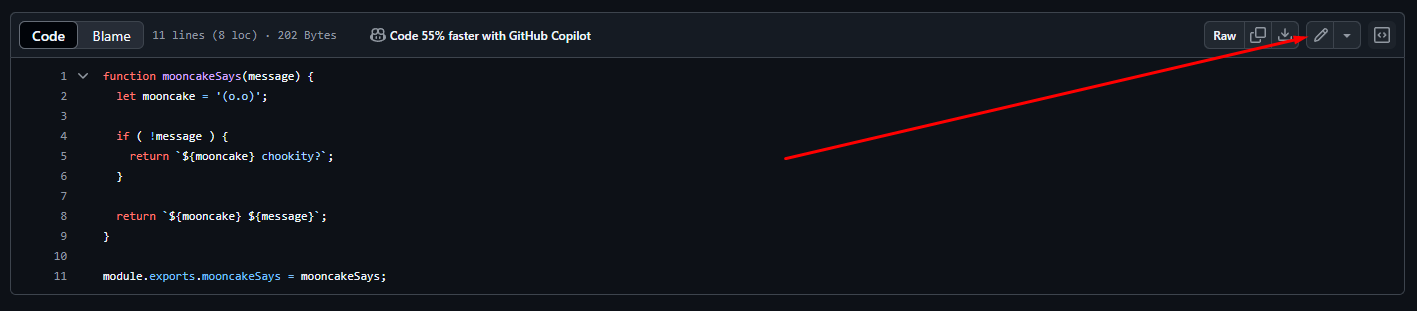
* 1. Кликните на любой из запусков, например, test (18.x).



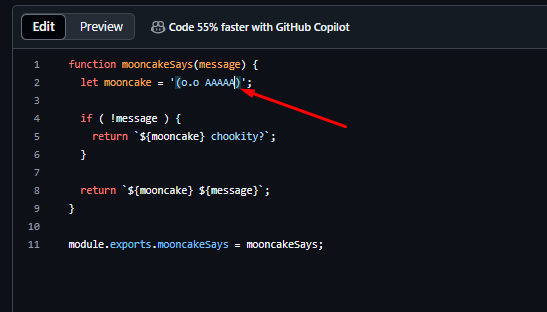
* 1. Разверните шаг "Run npm test". Вы увидите вывод команды npm test, включая информацию о пройденных тестах.
* **Пример отладки: Имитация ошибки и её исправление**
  1. Вернитесь на вкладку "Code".
  2. Откройте файл index.js.



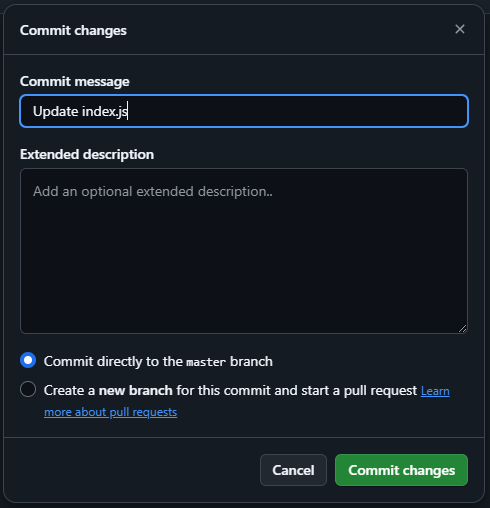
* 1. Нажмите на значок карандаша ("Edit this file") в правом верхнем углу.



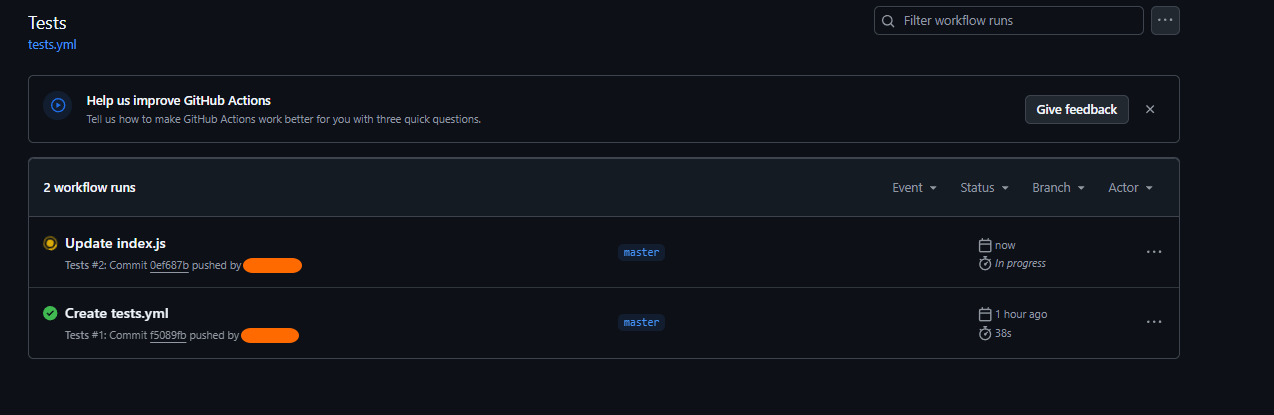
* 1. Сломайте код намеренно. Например, измените строку let mooncake = '(o.o)'; на let mooncake = '(o.o AAAAA)';



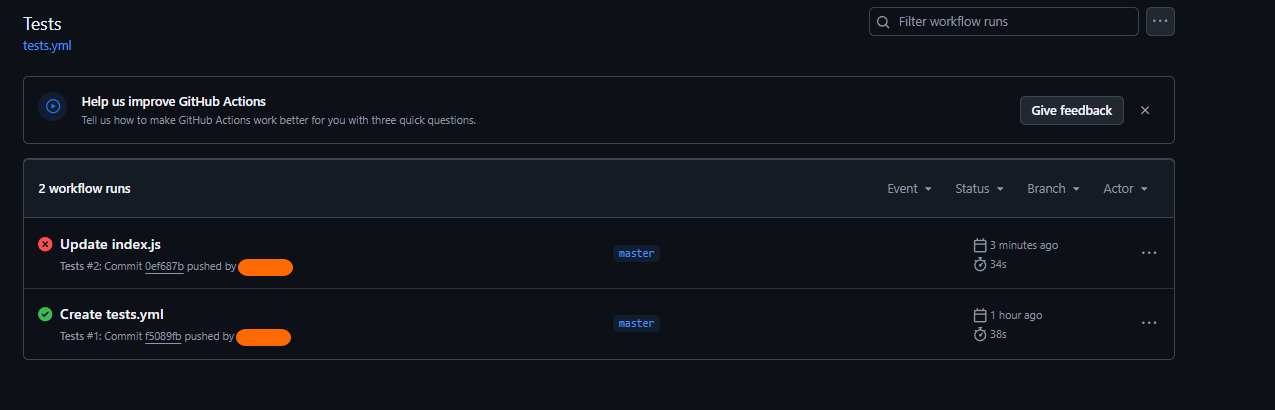
* 1. Нажмите "Commit changes...".
  2. Введите сообщение коммита, например, "Update index.js" (чтобы отличать от предыдущего). Убедитесь, что коммит идет в master branch. Нажмите "Commit changes".



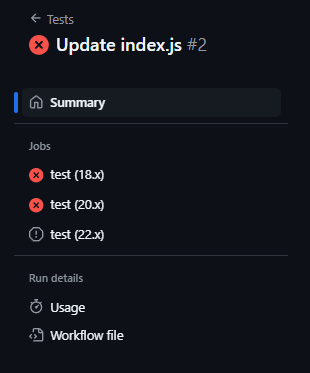
* 1. Снова перейдите на вкладку "Actions". Вы увидите новый запуск workflow. Сначала он будет "желтым" (в процессе выполнения).



* 1. Через некоторое время он станет "красным", так как тесты не пройдут из-за внесенной ошибки.



* 1. Кликните на "красный" коммит ("Update index.js").



* 1. В боковой панели слева все версии тестов (test (18.x), test (20.x), test (22.x)) будут помечены красными крестиками.
  2. Кликните на любой из них, например, test (18.x).



* 1. Разверните шаг "Run npm test". Вы увидите детальный вывод ошибки, показывающий, какой тест не прошел и почему (ожидаемое значение не совпало с полученным).

**Продвинутые возможности и лучшие практики**

* **Интеграция с внешними сервисами** (стр. 4-5)  
  GitHub Actions легко интегрируется с:
  + **Облачными платформами:** AWS, Google Cloud, Microsoft Azure (есть официальные Actions от провайдеров).
  + **Контейнеризацией:** Docker, Kubernetes.
  + **Системами управления версиями:** GitLab, Bitbucket (для синхронизации или миграции).
  + **Мониторингом:** Sentry, New Relic.
  + **Чатами:** Slack, Discord (для отправки уведомлений).  
    *Пример деплоя в Amazon ECS:* Показывает шаги логина в AWS ECR (реестр контейнеров), сборки и пуша Docker-образа, и деплоя задачи в ECS с использованием готовых aws-actions.
* **GitHub Marketplace: Поиск и использование готовых Actions**
  + Marketplace содержит тысячи готовых Actions для различных задач.
  + **Популярные Actions:** actions/checkout, actions/setup-\*, actions/cache, actions/upload-artifact, actions/download-artifact.

**9. Будущее GitHub Actions**

* **Текущие тенденции:** Расширение экосистемы готовых Actions, углубление интеграции с облачными сервисами (AWS, Azure, GCP), постоянное улучшение инструментов CI/CD, безопасности (например, Trusted Publishing для PyPI - стр. 52), и поддержки DevOps практик. (стр. 8)
* **Потенциальное влияние:** GitHub Actions продолжит изменять процессы разработки, становясь еще более важным инструментом. Улучшенная поддержка языков, интеграция с AI-инструментами (как GitHub Copilot в редакторе workflow - стр. 20), и фокус на безопасности повысят скорость и надежность разработки, улучшая конкурентоспособность бизнеса.

**10. Заключение**

GitHub Actions – это универсальная и мощная платформа для автоматизации всего цикла разработки ПО, включая тестирование. Она позволяет командам повысить эффективность, улучшить качество кода и ускорить доставку продуктов пользователям. Для тестировщиков, особенно занимающихся автоматизацией, освоение GitHub Actions открывает широкие возможности для интеграции тестов в CI/CD конвейер, обеспечения стабильности продукта и более тесного взаимодействия с командой разработки. Начиная с простых workflow для запуска тестов и постепенно переходя к более сложным сценариям с использованием матриц, кэширования, секретов и интеграций, вы можете значительно повысить ценность тестирования в ваших проектах.