# ТЕХНОЛОГИИ АВТОМАТИЗАЦИИ ПРОЦЕССОВ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

# Лекция 1. Введение, постановка задачи DevOps и автоматическое тестирование.

## 1.1. Основные термины

*DevOps* – методология автоматизации технологических процессов сборки, настройки и развёртывания программного обеспечения, предполагающая активное взаимодействие специалистов по разработке со специалистами по информационно-технологическому обслуживанию и взаимную интеграцию их технологических процессов друг в друга для обеспечения высокого качества программного продукта.

*Continuous Integration* – сборка, тестирование и доставка приложения без участия человека.

*Continuous Testing* – выполнение автоматизированных тестов в рамках конвейера поставки ПО для получения немедленной обратной связи

*Continuous Delivery* – подход к разработке, в рамках которого приложение создаётся в короткие циклы, причём выпуск приложения надлежащего качества гарантирован в любое время.

*Continuous Deployment* – подход к разработке, при котором функциональные возможности предоставляются посредством автоматизированного развёртывания.

*Infrastructure as Code* — подход для управления и описания инфраструктуры через конфигурационные файлы или взаимодействие, а не через ручное редактирование конфигураций на серверах.

## 1.2. Жизненный цикл продукта и DevOps

На рис. 1 приведена схема внесения модификаций в программный продукт.



Рис. 1. Схема внесения модификации в программный продукт

Методология DevOps предполагает следующие изменения:

* На этапе **разработки** – ведение репозитория, автоматизация тестирования, сбор статистики, проверка корректности сборки ПО.
* На этапе **приемки** – автоматическое тестирование в пользовательских условиях, оценка качества работы.
* На этапе **развертывания** – автоматизированная настройка среды у пользователя.
* На этапе **релиза** – автоматизация рутинных действий (release notes, выставление тегов, подписание пакетов ЭЦП).

## 1.3. Системы контроля версий

Код должен быть:

* синхронизован
* наблюдаем
* отчуждаем

Самая популярная система контроля версий – Git.

Требуется внедрить:

* Соглашения о версиях языков программирования, сред разработки, стилей кодирования
* Правила ветвления
* Правила рецензирования и слияния веток
* Автоматические проверки по событиям в репозитории

## 1.4. Тестирование

*Ошибка* – несоответствие факта и желания в поведении программы. Тестирование не гарантирует отсутствие ошибок.

Виды ошибок

* Синтаксические (деление на ноль, некорректные имена функций)
* Runtime-ошибки (Segfault, NPE, OutOfMemory)
* Ошибки конкурентного доступа (дедлоки)
* Поведение не соответствует спецификации

Тестирование классифицируется по видам ошибок, которые оно может потенциально обнаружить:

* Статический анализ ПО – определение мест, где может быть ошибка, без реального выполнения исследуемых программ
* Юнит-тестирование – тестирование отдельных модулей продукта – пакета, функции, класса и т. д.
* Интеграционное тестирование – запуск продукта целиком в естественной среде выполнения, проверка общей корректности его работы.
* Функциональное тестирование – проверка соответствия продукта функциональным требованиям.
* Нагрузочное тестирование – проверка производительности системы под нагрузкой.

# Лекция 2. Docker. Введение.

## 2.1. Задача автоматизации среды

В рамках методологии DevOps автоматизируют:

* настройку среды
* сборку
* запуск
* развертывание

Это необходимо делать:

* стабильно
* повторимо
* управляемо

При этом возникают следующие проблемы:

* различные версии программных зависимостей, инструментов, ОС
* разное аппаратное обеспечение

Пути решения проблем:

* Документирование версии зависимостей, инструментов и ОС
* Инструменты автоматизации действий: Сhef, Puppet, Ansible
* Инструменты виртуализации и изоляции: Docker, VMWare, QEMU

## 2.2. Docker

*Docker* – это система изоляции процессов, версионирования и управления подобными процессами.

Изоляция:

* похожа на виртуализацию, но имеет меньше ограничений
* общие ресурсы с ОС (ядро)
* свое окружение (файлы, переменные среды)
* свое дерево процессов
* ограничения на диск, ОЗУ, дерево процессов, сеть

Чем оперирует докер:

* Образы / Images (заготовки для контейнеров)
* Теги / Tags - версии образов
* Контейнеры / Containers (изолированные деревья процессов) - запущенный образ с дополнительными настройками
* Репозитории - опубликованные образы с тегами

Образы состоят из:

* Базового образа
* Слоев
* Слоя контейнера
* Манифеста

## 2.3. Как работают контейнеры

После вызова команды *docker run my\_image* docker-daemon считывает название образа, монтирует его файлы из хранилища образов. Затем docker-daemon создает новый процесс из команды docker run, ограничивая его с помощь linux namespaces:

* передает настройки ограничений работы контейнера (память, диск, процессор)
* передает кастомную среду (ENV, подключенные каталоги, сеть)
* монтирует к нему образ контейнера (writable)

Далее процесс начинает свою работу в неведении и думая, что он один, при этом дисковое пространство убывает только, если процесс что-то пишет на диск.

# Лекция 3. Docker. Работа с готовыми образами

Docker позволяет использовать множество готовых образов (напомним, что запущенный образ с дополнительными настройками – это контейнер). Готовые образы хранятся в реестре Docker Hub – хранилище образов. В большинстве случаев вы будуте использовать базовые образы без небходимости определять и настраивать свои собственные.

## 3.1. Просмотр образов и контейнеров

Для того, чтобы просмотреть доступные локально образы, можно использовать команду docker image list или docker images, которая позволяет вывести список образов вместе с вспомогательной информацией: tag, image\_id, датой создания и размером. Её синтаксис:

*docker images [OPTIONS] [REPOSITORY[:TAG]]*

Команда docker images принимает необязательный аргумент *[REPOSITORY[:TAG]]*, который ограничивает список образами, соответствующими аргументу. Если вы укажете *REPOSITORY*, но не *TAG*, команда выведет список всех образов в данном репозитории. Если указаны и *REPOSITORY*, и *TAG*, отображаются только образы, соответствующие заданным репозиторию и тегу. Если ничего не соответствует *REPOSITORY[:TAG]*, команда вернет пустой список. Кроме того, к команде могут применяться фильтры, например, по лэйблу или времени создания.



Для вывода списка контейнеров используется команда

*docker ps [OPTIONS]*

Она позволяет вывести все контейнеры, отфильтровать контейнеры, вывести последние контейнеры и т.д.



## 3.2. Загрузка образов

Если нужный образ отсутствует, его можно загрузить с помощью команды docker pull из репозитория. Её синтаксис:

*docker pull [OPTIONS] NAME[:TAG|@DIGEST]*

Если тег не указан, по умолчанию используется тег *:latest*. Образы могут состоять из нескольких слоев. Слои могут быть переиспользованы - уже скачанные слои не будут загружены повторно.

По умолчанию *docker pull* извлекает образы из Docker Hub. Кроме того, можно вручную указать путь к реестру, из которого следует извлечь данные. Например, если вы настроили локальный реестр, вы можете указать его путь и извлекать образы из него. Путь реестра аналогичен URL-адресу, но не содержит спецификатора протокола (https://). Docker использует протокол https:// для связи с реестром, если к реестру не разрешен доступ через небезопасное соединение.



## 3.3. Запуск контейнера

Для того, чтобы запустить контейнер, используется команда *docker run*. Её синтаксис:

*docker run [OPTIONS] IMAGE [COMMAND] [ARG...]*

Команда *docker run* сначала создает слой контейнера с возможностью записи поверх указанного образа, а затем запускает его. Остановленный контейнер можно перезапустить с сохранением всех предыдущих изменений.

Рассмотрим некоторые возможные опции:

|  |  |
| --- | --- |
| --detach , -d | запустить контейнер в фоновом режиме и вывести его идентификатор |
| --env , -e | установить переменные окружения |
| --expose | открыть порт или диапазон портов |
| --interactive , -i | держать STDIN открытым, даже если он не подключен |
| --label , -l | задать метаданные |
| --memory , -m | установить лимит используемой памяти |
| --mount | монтирование файловой системы к контейнеру |
| --name | присвоить контейнеру имя |

****

## 3.4. Остановка и удаление контейнеров и образов

Различные команды могут использоваться для того, чтобы остановить или удалить контейнер, образ или данные:

* docker kill [OPTIONS] CONTAINER [CONTAINER...] – остановить один или несколько запущенных контейнеров
* docker rm [OPTIONS] CONTAINER [CONTAINER...] – удалить один или несколько контейнеров
* docker rmi [OPTIONS] IMAGE [IMAGE...] – удалить один или несколько образов
* docker container prune [OPTIONS] – удалить все остановленные контейнеры
* docker system df [OPTIONS] – показать использование диска
* docker system prune [OPTIONS] – удалить неиспользуемые данные

# Лекция 4. Docker. Создание своего образа

Docker позволяет не только использовать готовые образы, но и создавать свои. Для этого используется dockerfile.

## 4.1. Dockerfile

Dockerfile — это текстовый документ, содержащий все команды, которые пользователь может вызвать в командной строке для сборки образа. Docker создает образ из файла Dockerfile и контекста. Контекст — это набор файлов в указанном месте PATH или URL: PATH — это каталог в вашей локальной файловой системе, URL-адрес — это расположение репозитория Git.

Dockerfile создается по определенным правилам. Комментарии начинаются с символа #. Инструкции не чувствительны к регистру, однако принято писать команды заглавными буквами, чтобы их было легче отличить от аргументов. Docker выполняет инструкции по порядку. Dockerfile должен начинаться с инструкции FROM, задающей базовый (родительский) образ. Дальше могут следовать следующие инструкции (но они не обязательны):

1. LABEL — описывает метаданные. Например — сведения о том, кто создал и поддерживает образ.
2. ENV — устанавливает постоянные переменные среды.
3. RUN — выполняет команду и создаёт слой образа. Используется для установки в контейнер пакетов.
4. COPY — копирует в контейнер файлы и папки.
5. ADD — копирует файлы и папки в контейнер, может распаковывать локальные .tar-файлы.
6. CMD — описывает команду с аргументами, которую нужно выполнить когда контейнер будет запущен. Аргументы могут быть переопределены при запуске контейнера. В файле может присутствовать лишь одна инструкция CMD.
7. WORKDIR — задаёт рабочую директорию для следующей инструкции.
8. ARG — задаёт переменные для передачи Docker во время сборки образа.
9. ENTRYPOINT — предоставляет команду с аргументами для вызова во время выполнения контейнера. Аргументы не переопределяются.
10. EXPOSE — указывает на необходимость открыть порт.
11. VOLUME — создаёт точку монтирования для работы с постоянным хранилищем.

Готовый dockerfile может выглядеть, например, так:

#Задаем базовый образ

FROM python:3.6-alpine

#Задаем рабочую директорию для следующих команд

WORKDIR /app

#Копируем файлы в контейнер

COPY ./requirements.txt ./

COPY ./server.py ./

COPY ./do\_db\_stuff.py ./

#Устанавливаем в контейнере пакеты

RUN python -m pip install -r ./requirements.txt

RUN date -R > ./version.txt

#выполяем в контейнере команду

CMD python -m http.server 8000

## 4.2. Сборка образа

Сборка образа выполняется с помощью команды *docker build*. Сборка выполняется демоном Docker, а не интерфейсом командной строки. Первое, что делает процесс сборки, это отправляет весь контекст (рекурсивно) демону. Традиционно Dockerfile называется Dockerfile и располагается в корне контекста. Однако вы можете использовать флаг -f, чтобы указать Dockerfile в любом месте вашей файловой системы.

*docker build -f /path/to/a/Dockerfile .*

Вы можете указать репозиторий и тег, в котором будет сохранен новый образ, если сборка пройдет успешно:

*docker build -t shykes/myapp .*

Прежде чем демон Docker выполнит инструкции в Dockerfile, он выполняет предварительную проверку Dockerfile и возвращает ошибку, если синтаксис неверен. Успещно собранный образ можно использовать для запуска контейнеров также, как и готовый образ.

# Лекция 5. Docker. Работа с docker-compose

## 5.1. Создание многоконтейнерных приложений

Docker Compose — это инструмент для создания и запуска многоконтейнерных приложений. Для задания конфигурации используется файл YAML, а затем с помощью одной команды запускаются, управляются и масштабируются все сервисы. Docker Compose позволяет запустить или остановить все контейнеры сразу, настроить сеть, порты и имена в одном месте, а также связать контейнеры между собой.

Использование Compose в основном состоит из трех шагов:

* Определение среды приложения с помощью Dockerfile, чтобы его можно было воспроизвести где угодно.
* Определение служб, из которых состоит приложение, в файле docker-compose.yml, чтобы их можно было запускать вместе в изолированной среде.
* Выполнение команды *docker compose up*, запускающей приложение.

## 5.2. Задание конфигурации

Файл с конфигурацией может содержать следующие директивы (в таблице перечислены не все возможные варианты):

|  |  |
| --- | --- |
| build | Задаёт параметры конфигурации, которые применяются во время сборки |
| configs | Предоставляет доступ к конфигурациям для каждого сервиса |
| depends\_on | Задаёт зависимость между сервисами |
| deploy | Задаёт конфигурацию, связанную с развертыванием и запуском сервисов |
| devices | Задаёт список сопоставлений устройств |
| entrypoint | Переопределяет точку входа |
| environment | Задаёт переменные окружения |
| expose | Открывает порты, не публикуя их на хост-компьютере — они будут доступны только связанным контейнерам |
| healthcheck | Настраивает проверку работоспособности контейнеров |
| image | Указывает образ для запуска контейнера |
| logging | Настраивает ведение журнала событий |
| ports | Открывает порты |
| services | Задаёт список контейнеров  |
| version | Задаёт версию приложения |
| volumes | Монтирует диски |

Пример:

#Задаётся версия приложения

*version: ‘2’*

#Дальше будет список сервисов

*services:*

#Первый сервис под названием app, который собирается в текущей папке, имеет #открытый порт 8080 и зависит от сервиса mongo

*app:*

 *build: .*

 *ports:*

* *“8000:8000”*

 *depends on:*

* *mongo*

#Второй сервис под названием mongo, который собирается из заданного образа mongo

*mongo:*

*image: “mongo: 4.0.17-xenial”*

# Лекция 6. Отладка контейнеров

## 6.1. Логирование и выполнение команд в контейнере

Для отладки контейнеров могут использоваться команды *logs* и *exec*. Команда docker logs извлекает логи на момент выполнения команды. Её синтаксис:

*docker logs [OPTIONS] CONTAINER*

Команда docker logs может иметь следующие флаги:

|  |  |
| --- | --- |
| --details | Показывает дополнительные сведения |
| --follow , -f | Позволяет следить за логами |
| --since | Показывает логи с заданного момента времени |
| --tail , -n | Задаёт количество отображаемых строк логов (с конца файла) |
| --until | Показывает логи до заданного момента времени |

Команда *docker exec* запускает новую команду в работающем контейнере. Её синтаксис:

*docker exec [OPTIONS] CONTAINER COMMAND [ARG...]*

Команда, запущенная с помощью docker exec, выполняется только во время работы основного процесса контейнера (PID 1) и не перезапускается при перезапуске контейнера. Команда будет выполняться в каталоге контейнера по умолчанию.

## 6.2. Отладка производительности

Для отладки контейнеров могут использоваться утилиты *htop* и *ctop*. *htop* выводит список запущенных процессов и информацию о них. *htop* выводит в частности следующие столбцы:

|  |  |
| --- | --- |
| PID | Идентификатор процесса |
| USER | Имя пользователя владельца процесса |
| RES | Размер используемой физической памяти процесса |
| VIRT | Размер виртуальной памяти процесса |
| S | Состояние процесса |
| CPU% | Процент процессорного времени, которое процесс использует в данный момент |
| MEM% | Процент памяти, используемой процессом в данный момент |
| TIME+ | Время работы процесса |
| Command | Командная строка процесса |

Утилита *ctop* предоставляет краткий обзор метрик для нескольких контейнеров. Позволяет вывести имена контейнеров, их CID, нагрузку на процессор и память, информацию о передаче и получении данных.

# Лекция 7. Jenkins. Введение и основные сущности

Jenkins — это сервер автоматизации с открытым исходным кодом. Он помогает автоматизировать сборку, тестирование и развертывание, облегчая процесс CI/CD. Основное приложение Jenkins с базовыми функциями называется ядром Jenkins (core). Также Jenkins легко расширяется с помощью множества плагинов, позволяющих, например, настраивать мониторинг системы.

## 7.1. Инфраструктура Jenkins

Структурной единицей Jenkins является нода (node) – машина, являющаяся частью среды Jenkins, способная выполнять проекты. Нодой является главный сервер Jenkins (controller) – центральный координирующий процесс, который хранит конфигурацию, загружает плагины и отображает различные пользовательские интерфейсы для Jenkins. Также нодами являются агенты (agents) – машины или контейнеры, которые подключаются к controller и выполняют задачи по его указанию. Нода может иметь 0 и больше экзекьюторов (executors), определяющих сколько проектов может одновременно исполняться на ноде. Проект (project, job) – это задаваемое пользователем описание работы, которую должен выполнить Jenkins, например, создание части программного обеспечения.

Результат запуска проекта называется билдом (build). В результате билда могут создаваться артифакты – неизменяемые файлы, которые могут использоваться в дальнейшем. Билд завершается с одним из следующих статусов:

|  |  |
| --- | --- |
| Aborted | Билд был прерван до завершения |
| Failed | Во время работы билда произошла фатальная ошибка |
| Stable | Билд был успешен и нет информации о нефатальных ошибках |
| Successful | Во время билда не произошло ошибок компиляции |
| Unstable | Во время билда произошли ошибки, но не фатальные |

## 7.2. Проекты в Jenkins

Проекты бывают двух типов: Freestyle и Pipeline. Freestyle проекты настраиваются с помощью графического интерфейса, путём добавления шагов в сборку. Шаг (step) – это атомарная задача, которая указывает Jenkins, что именно необходимо сделать. Freestyle проекты достаточно легко использовать, кроме того они обладают большой гибкостью. Однако они плохо применимы для больших и сложных проектов.

Pipeline проекты позволяют задавать набор инструкций с помощью кода. Концептуально части проекта могут быть разбиты на стадии (stages), объединяющие шаги, например, “деплой” или “тестирование”. Для написания pipeline проектов используется язык програмирования Groovy.

Существует два синтаксиса pipeline проектов - декларативный (declarative pipeline) и скриптовый (scripted pipeline). Декларативный синтаксис ограничивает то, что доступно пользователю, более строгой и заранее определенной структурой, что делает его идеальным выбором для простых проектов. Скриптовый синтаксис имеет очень мало ограничений, которые как правило, определяются самим Groovy, что делает его идеальным выбором для опытных пользователей и тех, у кого более сложные задачи. Как следует из названия, декларативный синтаксис предполагает декларативную модель программирования, а скриптовый – императивную.

# Лекция 8. Jenkins. Настройка Job, анализ сборок, особенности эксплуатации

## 8.1. Настройка проектов

Проект в Jenkins — это последовательность шагов, которые поддерживают реализацию и интеграцию CI/CD в ваш процесс разработки программного обеспечения. Pipeline проекты создаются на языке программирования Groovy, в FreeStyle проектах те же шаги задаются с помощью графического интерфейса. Далее будут подробнее рассмотрены различные возможности проектов в Jenkins.

Шаг можно рассматривать как одну команду, которая выполняет одно действие. Если шаг успешно выполнен, Jenkins переходит к следующему шагу, если шаг неуспешен, процесс прерывается. Такими шагами могут быть, например, тестирование, деплой или выполнение любой консольной программы.

Также существуют шаги, которые «обертывают» другие шаги и позволяют, например, повторять шаг до тех пор, пока он не будет успешными, или наоборот завершить процесс, если шаг занимает слишком много времени. После выполнения проекта может потребоваться постобработка, например, очистка рабочего пространства, сохранение артефактов, логирование или отправление уведомления со статусом проекта. Кроме того, Jenkins позволяет задавать переменные окружения как для одного шага так и для всего проекта. Например, так можно задавать данные для авторизации, которые не стоит напрямую прописывать в проекте, или указывать системные требования к ноде, на которой будет выполняться проект.

Jenkins позволяет не только автоматизированно выполнять шаги, но и может ожидать реакции человека. Например, человек может быть необходим, чтобы ввести данные или подтвердить допустимость следующего шага.

## 8.2. Пример проекта

 Рассмотрим пример проекта, содержащего некоторые из описанных выше шагов.

pipeline {

 /\*проект может выполняться на любом доступном агенте\*/

 agent any

 stages {

 /\* Деплой в тестовой среде \*/

 stage('Deploy - Staging') {

 /\*Запуск двух скриптов\*/

 steps {

 sh './deploy staging'

 sh './run-smoke-tests'

 }

 }

 /\*Шаг, ожидающий подтверждения от человека \*/

 stage('Human check') {

 steps {

 input "Does the staging environment look ok?"

 }

 }

 /\*Деплой в продуктовой среде \*/

 stage('Deploy - Production') {

 /\*Запуск скрипта \*/

 steps {

 sh './deploy production'

 }

 }

 }

 /\* Постобработка \*/

 post {

 /\* При любом завершении проекта очистить рабочее пространство \*/

always {

 echo 'One way or another, I have finished'

 deleteDir()

 }

/\* Если проект завершился ошибкой, отправить письмо \*/

 failure {

 mail to: 'team@example.com',

 subject: "Failed Pipeline: ${currentBuild.fullDisplayName}",

 body: "Something is wrong with ${env.BUILD\_URL}"

 }

}

}

# Лекция 9. Jenkins. Ноды и дополнительные вопросы

## 9.1. Компоненты системы распределенных сборок

Распределенная архитектура Jenkins состоит из нод, агентов и исполнителей (экзекьюторов), а также самого главного контроллера Jenkins.

Главный контроллер Jenkins — это веб-сервер, который действует как «мозг», решающий, как, когда и где выполнять поставленные задачи. На главном контролере также выполняются все задачи управления (например, настройка, авторизация и аутентификация). Создаваемые в процессе выполнения задач файлы записываются в файловую систему контроллера или выгружаются в репозиторий артефактов.

Ноды – это «машины», на которых работают агенты. Jenkins отслеживает каждую подключенную ноду, оценивая дисковое пространство, загруженность, время отклика и т.д. Нода отключается, если любое из этих значений выходит за пределы настроенного порога. На главном контроллере Jenkins также могут выполняться задачи проектов, но это может снизить производительность, уменьшить масштабируемость и создать серьезные проблемы с безопасностью.

Агенты управляют выполнением задачи от имени контроллера Jenkins с помощью исполнителей. Агент на самом деле представляет собой небольшой клиентский процесс Java. Инструменты, необходимые для сборки и тестирования, устанавливаются на ноде, где работает агент; их можно установить напрямую или в контейнере. Каждый агент фактически представляет собой процесс со своим собственным PID на хост-компьютере. Исполнитель (экзекьютор) — это слот для выполнения задач; по сути, это поток в агенте. Количество исполнителей на ноде определяет количество задач, которые могут выполняться одновременно.

# Лекция 10. GitHub actions

## 10.1. Зачем нужен GitHub actions?

GitHub Actions — это платформа позволяющая автоматизировать процесс сборки, тестирования и развертывания. Обрабатываться может как каждый отдельный pull request, так и готовые к развертыванию в продуктовой среде части кода. GitHub Actions позволяет автоматизировать и другие действия, например, добавление меток к веткам в репозиториии.

Единственное, что вам понадобится для того, чтобы использовать GitHub Actions – это репозиторий GitHub. GitHub предоставляет виртуальные машины Linux, Windows и macOS для запуска ваших процессов, вы также можете самостоятельно развернуть и поддерживать локальный сервис.

## 10.2. Компоненты GitHub Actions

Автоматизация в GitHub Actions состоит из нескольких компонентов.

 Процесс (workflow) — это настраиваемая автоматизированная последовательность действий, в рамках который выполняется одно или несколько заданий (job). Процессы определяются в файле YAML и запускаются после определенного события (event) в репозитории, вручную или по расписанию. Например, можно задать процесс для тестирования pull requests или для развертывания приложения.

 Задание состоит из шагов, которые выполняются в рамках одного процесса. Каждый шаг - это shell скрипт или действие (то есть заранее заданный набор сложных, но часто повторяющихся шагов). Шаги выполняются последовательно и зависят один от другого, а также могут использовать общие данные и обмениваться ими. Кроме того, одна задача может зависеть от другой, в таком случае зависимая задача начнет выполняться после успешного завершения предшествующей задачи.

## 10.3. Пример использования GitHub Actions

 GitHub Actions использует YAML файлы для определения процессов. Каждый процесс хранится в виде отдельного файла YAML в каталоге с именем .github/workflows.

Например, YAML файл может выглядеть так:

#Название процесса

name: learn-github-actions

#Триггер, то есть событие, при котором процесс запустится

on: [push]

#Начало списка задач

jobs:

#Название задачи

 check-bats-version:

#Задание ОС, на которой задача будет выполняться

 runs-on: ubuntu-latest

#Шаги задачи, содержащие 2 заранее заданных action и две команды

 steps:

 - uses: actions/checkout@v3

 - uses: actions/setup-node@v3

 with:

 node-version: '14'

 - run: npm install -g bats

 - run: bats -v