УДК 004

Разработка модуля компиляторов в рамках единого Docker-контейнера для работы с программами на различных языках программирования в веб-ориентированных средах

Фатеенков Данила Витальевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В данной статье описан процесс сборки Docker-контейнера с поддержкой различных языков программирования, который в дальнейшем может быть использован в качестве основного модуля при разработке онлайн-компиляторов, обучающих и тестирующих систем. Также представлены код для сборки языков программирования из Source архивов («исходники») и автоматизация тестирования установленных языков программирования в Docker-контейнере.
Ключевые слова: контейнеризация, Docker, Bash, сборка приложений, Linux, системное программирование, языки программирования

Development of a compiler module within a single Docker container for working with programs in different programming languages in web-oriented environments

Fateenkov Danila Vitalievich
Sholom-Aleichem Priamursky State University
Student

Abstract
This article describes the process of building a Docker container with support for various programming languages, which can be used as a basic module for developing online compilers, training and testing systems. The code for building programming languages from Source archives ("sources") and automation of testing of installed programming languages in Docker container are also presented.
Keywords: containerisation, Docker, Bash, application building, Linux, system programming, programming languages

1. Введение

1.1 Актуальность

Компиляция и запуск программ в веб-ориентированных средах (клиент-серверных приложениях, сайтах) в настоящее время не является новинкой в сфере разработки информационных систем, но при этом вопрос компоновки и подготовки модуля, в котором будут храниться все необходимые языки программирования для работы с кодом, который могут

отправить сторонние пользователи, остаётся актуальным. Актуальность обусловлена, в первую очередь, тем, что идея обработки пользовательского кода на стороне сервера может создавать большое количество проблем на этапе подготовки единого модуля, в котором будет записана вся необходимая информация. Для решения этой проблемы можно использовать инструменты контейнеризации приложений, одним из которых является Docker.

Правильно собранный модуль, который предоставит разработчику достаточную вариативность настроек компонентов, будет актуален и полезен и в настоящее время несмотря на то, что идея создания онлайн-компиляторов не является чем-то новым в настоящее время.

1.2 Обзор исследований

В своей научной работе, А.О. Кузовенков выделил критерии для систем, которые автоматизируют процесс проверки кода [1]. В статье рассмотрены требования к таким системам, а также рассмотрены уже существующие решения, автоматизирующие данный процесс.

О.Н. Половикова, А.С. Маничева и В.В. Журавлева провели анализ проблемы сбора и накопления готовых программных решений студентов с возможностью их непосредственного выполнения (тестирования) [2]. Для решения проблемы предлагается подход на основе технологии контейнерной виртуализации. Каждое программное решение автоматически разворачивается в изолированном Docker-контейнере. Подход апробирован на практике.

А.С. Бондаренко и К.С. Зайцев провели исследование возможности и способов применимости систем управления контейнерами для построения распределенных микросервисных архитектур [3].

M. Mareš рассмотрел проблемы безопасности систем автоматической проверки решений, которые представлены в виде кода на определённом языке программирования, в соревнованиях по программированию [4]. Автор статьи выделил 7 типов атак на проверяющие системы, которые направлены на эксплуатацию ошибок в различных модулях проверки.

1.3 Цель исследования

Цель – создать Docker-контейнер с поддержкой различных языков программирования для последующего внедрения в обучающую или тестирующую информационную систему.

2 Материалы и методы

Создание контейнера ведётся в ОС Debian 12. Для разработки используется Docker, а также Bash для подготовки скриптов. Для тестирования установленных языков программирования используются примеры программ, которые написаны на соответствующих языках программирования.

3 Результаты и обсуждения

Разработка модуля компиляторов представляет собой достаточно специфичную задачу и использование такого большого функционального контейнера может быть не оправдано в ряде случаев, но при этом такой контейнер может повысить безопасность и эффективность работы, например, онлайн-компиляторов или обучающих систем, где предполагается наличие практических заданий, связанных с написанием кода на различных языках программирования. Задача разработки такого модуля представляет собой обширное количество возможных путей достижения результата и в зависимости от требований инструменты разработки могут отличаться. Например, можно собрать единый контейнер на основе некоторой ОС (в качестве примера можно рассматривать Ubuntu или любую другую UNIX-подобную ОС), в котором будут устанавливаться все языки программирования через apt-get или yum. Также можно вручную собирать архивы с исходным кодом в компиляторы и интерпретаторы, что в ряде случаев значительно надёжнее, так как можно решить проблему совместимости и перехода между различными ОС. Также стоит учитывать, что репозитории операционных систем могут обновляться со значительной задержкой, что тоже ведёт к необходимости самостоятельной сборки исходного кода языка программирования (например, на момент написания и публикации данной работы в официальном репозитории Debian представлен Python версии 3.11.2, хотя наиболее актуальной является 3.12.4).

В ходе данной работы будет рассмотрен процесс сборки языков программирования из архивов с исходным кодом. Для подготовки такого контейнера необходимо понимать Bash и уметь писать на нём скрипты. Все консольные команды, написанные в соответствующем проекту Dockerfile, будут учитывать ОС – Debian 12-ой версии.

Перед началом сборки первого языка программирования желательно подключить модули и библиотеки, которые используются в процессе сборки приложений (например, make или curl). В Docker уже предусмотрен такой контейнер, название которого “buildpack” [5], в котором есть всё необходимое для установки и настройки различных приложений: make, curl, unzip, zx-utils и др.

Чтобы лучше понимать работу контейнера, который представлен в данной статье, нужно разобрать процесс сборки одного из языков программирования.

В качестве примера далее будет описан процесс сборки языка программирования PHP. В первую очередь нужна ссылка на репозиторий языка, откуда можно будет получить нужную версию. У многих языков программирования такие репозитории открыты для всех желающих и в них представлены как уже архивы с готовыми исполняемыми файлами, так и с исходным кодом программ. С помощью curl можно скачать необходимую версию с репозитория ЯП (на момент написания и публикации статьи последняя версия PHP 8.3.8) и сохранить в необходимую директорию

(параметр -o, после которого следует путь, в который будет сохранён скачанный файл):

```
curl -fSsl https://www.php.net/distributions/php-8.3.8.tar.gz -o /tmp/php-8.3.8.tar.gz
```

Помимо параметра с указанием назначения в команде участвуют параметр -f и -Ssl (первый посылает сигнал прекращения работы команды при возникновении ошибки, второй скачивает только после проверки SSL сертификата).

С запакованным архивом сделать ничего нельзя, кроме распаковки, что можно сделать с помощью команды tar. Предварительно также создаётся директория, в которую будет распакован архив с помощью стандартной команды mkdir:

```
mkdir /tmp/php-8.3.8
tar -xf /tmp/php-8.3.8.tar.gz -C /tmp/php-8.3.8 --strip-components=1
```

При выполнении tar используются следующие опции: -x для указания команды на распаковку архива; -f для указания архива, который нужно распаковать; -C для указания пути назначения (куда будет распакован архив) и --strip-components=1 удаляет верхний уровень директорий из архива.

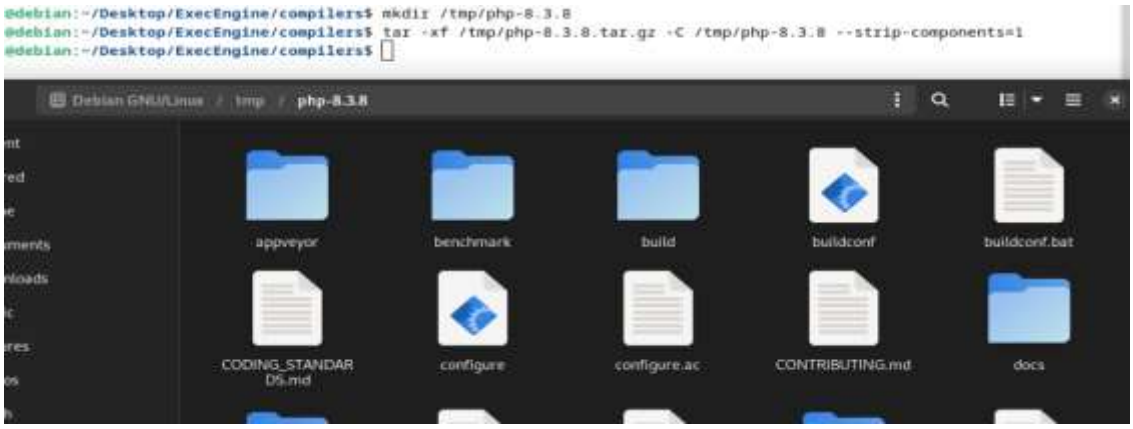

Рисунок 1. Результат скачивания и распаковки архива с исходным кодом PHP

Затем необходимо перейти в директорию с исходным кодом языка программирования. На данном этапе необходимо внимательно изучить структуру проекта перед сборкой его в исполняемые файлы. Если в корне проекта есть buildconf, то этот скрипт необходимо выполнить в первую очередь, так как данный скрипт генерирует или обновляет файлы configure и config.m4 на основе настроек и зависимостей, указанных в исходном коде программы. Скрипт подготавливает проект к запуску скрипта configure, который проверяет наличие необходимых библиотек и зависимостей в системе, а также генерирует Makefile для сборки. Также buildconf автоматизирует запуск всех необходимых утилит автоконфигурации, таких

как autoconf, autoheader, libtoolize, и других. Для запуска buildconf используется следующая команда:

```
./buildconf -f
```

В этой части кода также добавлена опция -f, которая нужна для принудительной сборки всех необходимых конфигурационных файлов.

```
@debian:/tmp/php-8.3.8$ ./buildconf -f
buildconf: Checking installation
buildconf: autoconf version 2.71 (ok)
buildconf: Forcing buildconf. The configure files will be regenerated.
buildconf: Cleaning cache and configure files
buildconf: Rebuilding configure
buildconf: Rebuilding main/php_config.h.in
buildconf: Run ./configure to proceed with customizing the PHP build.
@debian:/tmp/php-8.3.8$
```

Рисунок 2. Процесс генерации конфигурационных файлов

После выполнения данного скрипта стоит также указать путь установки программы (в данном случае языка программирования): обычно это /usr/local. Путь можно указать в конфигурационном файле configure. Для этого в файле предусмотрена опция --prefix:

```
./configure --prefix=/usr/local/php-8.3.8
```

После настройки окружения можно запускать процесс установки. Для этого должен быть установлен модуль make (который есть в buildpack). В качестве дополнительной опции можно передать количество ядер процессора, которое может быть задействовано в процессе сборки. Для указания всех ядер можно использовать макрос nproc:

```
make -j$(nproc)
make -j$(nproc) install
```

```
@debian:/tmp/php-8.3.8$ ./configure --prefix=/usr/local/php-8.3.8
checking for grep that handles long lines and -e... /usr/bin/grep
checking for egrep... /usr/bin/grep -E
checking for a sed that does not truncate output... /usr/bin/sed
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking target system type... x86_64-pc-linux-gnu
checking for pkg-config... no
checking for cc... cc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether the compiler supports GNU C... yes
checking whether cc accepts -g... yes
checking for cc option to enable C11 features... none needed
checking how to run the C preprocessor... cc -E
checking for icc... no
checking for suncc... no
```

Рисунок 3. Процесс подготовки исходного кода к сборке и генерация Makefile

После сборки приложения остаётся очистить директорию /tmp/ от архива и папки с исходным кодом языка программирования. Проверить работоспособность языка можно стандартной командой вывода версии в консоль (см. рис. 4).

```
@debian:~/Desktop/ExecEngine/compilers$ sudo docker run -it compiler
root@d96ad1c3d3a0:/# /usr/local/php-8.3.8/bin/php -v
PHP 8.3.8 (cli) (built: Jun 23 2024 13:54:03) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.3.8, Copyright (c) Zend Technologies
```

Рисунок 4. Результат сборки PHP из архива с исходным кодом языка внутри контейнера

С помощью команды RUN можно передать Dockerfile информацию о скрипте, который необходимо выполнить при подготовке контейнера, что позволит установить PHP до первого запуска самого контейнера, если в файле указать все команды, описанные выше.

Но может возникнуть ситуация, при которой разработчику может потребоваться использование нескольких версий одного языка программирования. Например, Python подразделяется на 2 версии: 2.7 и 3.12. Эти версии имеют значительные различия в синтаксисе, что делает вторую версию актуальной и в настоящее время. Установку нескольких версий можно организовать через цикл, который в Bash также предусмотрен. Перед этим в Dockerfile необходимо указать версии, которые будут установлены. Для этого необходимо использовать команду ENV:

```
ENV PYTHON_VERSIONS \
  2.7.17 \
  3.12.4
```

Затем в цикле можно повторить шаги по установке, которые были описаны выше. При этом в исходном коде Python отсутствует buildconf, но есть configure, в котором нужно указать через префикс путь, куда будет установлен язык программирования. Также нужно учитывать, что при наличии нескольких версий будет меняться и путь к скачиванию файлов, и путь к расположению языка программирования. Самый лёгкий способ разделения языков программирования по версиями это указание значения переменной, в которую сохранено значение версии на итерации цикла, в путях к файлам.

Итоговый вариант скрипта выглядит следующим образом:

```
RUN set -xe && \
  for VERSION in $PYTHON_VERSIONS; do \
    curl                                                -fSsl
"https://www.python.org/ftp/python/$VERSION/Python-
$VERSION.tar.xz" -o /tmp/python-$VERSION.tar.xz && \
    mkdir /tmp/python-$VERSION && \
```

```
        tar -xf /tmp/python-$VERSION.tar.xz -C /tmp/python-
$VERSION --strip-components=1 && \
        rm /tmp/python-$VERSION.tar.xz && \
        cd /tmp/python-$VERSION && \
        ./configure \
          --prefix=/usr/local/python-$VERSION && \
        make -j$(nproc) && \
        make -j$(nproc) install && \
        rm -rf /tmp/*; \
      done
```

Весь процесс установки автоматизирован, но при этом он может занять некоторое время, а дополнительного взаимодействия с make не требуется. В итоге внутри контейнера будут установлены 2 различные версии языка программирования Python (см. рис. 5).

```
root@d96ad1c3d3a0:/# /usr/local/python-2.7.17/bin/python --version
Python 2.7.17
root@d96ad1c3d3a0:/# /usr/local/python-3.12.4/bin/python3 --version
Python 3.12.4
```

Рисунок 5. Результат установки Python внутри контейнера

Можно также допустить следующий вариант установки: сначала разработчик скачивает архив с уже собранным языком программирования, затем его распаковывает в необходимую директорию, а после он может уже компилировать и выполнять код, написанный на этом языке программирования. Такой подход может быть особенно полезен, когда язык программирования требует специфичные модули для сборки в исполняемые файлы, либо если исходниками не предусмотрена сборка языка в Linux.

В качестве примера можно рассмотреть язык программирования Go, который требует наличия установленных git и cgo (компилятор C). На официальном сайте представлены архивы языка под Linux, которые можно скачать также через curl и распаковать сразу в /usr/local/:

```
    ENV GO_VERSIONS \
      1.22.4
    RUN set -xe && \
      for VERSION in $GO_VERSIONS; do \
        curl -fSsl "https://dl.google.com/go/go$VERSION.linux-
amd64.tar.gz" -o /tmp/golang-$VERSION.tar.gz && \
        mkdir /usr/local/go-$VERSION && \
        tar -xf /tmp/golang-$VERSION.tar.gz -C /usr/local/go-
$VERSION --strip-components=1 && \
        ln -s /usr/local/golang-$VERSION/bin/go /usr/bin/go && \
        rm -rf /tmp/*; \
      done
```

В данном случае также появляется новая команда ls, которая создаёт ярлык исполняемого файла Go. Данная команда опциональна и

компиляция/запуск программ, написанных на этом языке программирования, будет работать и без создания ссылки.

```
root@d96ad1c3d3a0:/# /usr/local/go-1.22.4/bin/go version
go version go1.22.4 linux/amd64
root@d96ad1c3d3a0:/# S
```

Рисунок 6. Результат установки собранного языка программирования внутри контейнера

Для проверки работоспособности модуля можно выполнить любой код на загруженных языках программирования. Самый стандартный способ проверки: вывод какого-либо сообщения в консоль (например, классический «Hello, world!»). Для выполнения данной задачи был сначала подготовлен перечень языков программирования, которые были предварительно загружены в контейнер:

1. Assembly (NASM v2.16.03)
2. Bash (v5.2.21)
3. C, C++ (GNU GCC v13.3.0)
4. Dart (v3.4.4)
5. Go (v1.22.4)
6. PHP (v8.3.8)
7. Java (OpenJDK 22)
8. JavaScript (NodeJS v20.15.0)
9. Kotlin (v2.0.0)
10. Pascal (FPC v3.2.2)
11. Python (v2.7.17 & v3.12.4)
12. PyPy (v7.3.16, Python 2.7 & 3.10)
13. Ruby (v3.3.3)
14. Rust (v1.79.0)
15. Swift (v5.10.1)

Затем были подготовлены примеры программ, которые выводят сообщение «Hello, world!» и в скобках после этого сообщения название языка программирования. Были созданы отдельные каталоги, в которые также сохранены конфигурации (файлы под названием «properties») каждого языка программирования, чтобы их легче было идентифицировать в дальнейшем. Файл «properties» содержит следующую информацию об языке программирования:

1. NAME – название языка.
2. VERSIONS – версии ЯП, разделённые пробелом.
3. SOURCE_FILE – название файла, в котором представлен код программы.
4. COMPILED_FILE – название скомпилированного файла.
5. COMPILE_CMD – команда для компиляции программы.
6. RUN_CMD – команда для запуска программы.

Пример конфигурации для Python представлен ниже:

```
    NAME="Python"
    VERSIONS="2.7.17 3.12.4"
    SOURCE_FILE="script.py"
    COMPILED_FILE=""
    COMPILE_CMD=""
    RUN_CMD="/usr/local/python-
${version}/bin/python${version%%.*}
/usr/local/tests/python/script-${version}.py"
```

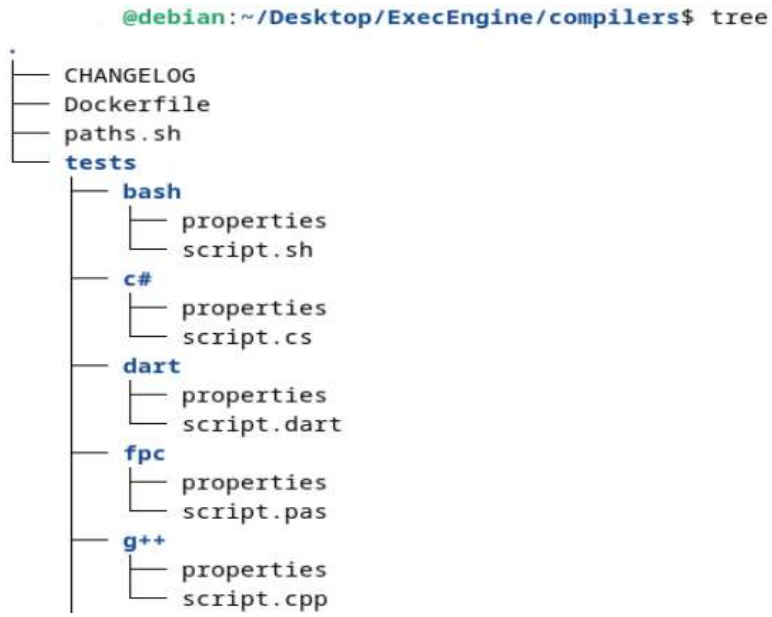

Рисунок 7. Структура проекта на этапе тестирования загруженных в контейнер ЯП

Следующий этап проведения тестирования: подготовка Bash-скрипта, который будет считывать файлы конфигураций и запускать команды на компиляцию и выполнение программ.

Данный скрипт разделён на 3 функции:

1. run_test – функция, в которой выполняется компиляция и запуск программ.

2. read_properties – функция, в которой происходит считывание конфигураций языков программирования.

3. main – основная функция, в которой происходит перебор языков программирования и проводится тестирование.

Функция run_test в качестве аргументов получает команды для компиляции и запуска программ, а также версию языка программирования. Выполнение команд компиляции и запуска осуществляются с помощью встроенной функции eval. При возникновении ошибки выводится соответствующее сообщение:

```
run_test() {
    local version="$1"
    local compile_cmd="$2"
```

```
        local run_cmd="$3"

        if [[ -n "$compile_cmd" ]]; then
            eval  "${COMPILE_CMD//\{VERSION\}/$version}"  ||  {
printf "Compilation failed! \n"; }
        fi
        eval  "${RUN_CMD//\{VERSION\}/$version}"  ||  {  printf
"Execution failed! \n"; }
    }
```

Функция read_properties считывает содержимое файла, путь к которому передаётся из функции main в качестве входного параметра. Вся информация из файлов сохраняется в соответствующие переменные и после этого функция прекращает свою работу на данной итерации цикла в функции main:

```
    read_properties() {
        local properties_file="$1"
        while IFS="=" read -r key value; do
            case "$key" in
                NAME) NAME="${value//\"/}" ;;
                VERSIONS) VERSIONS="${value//\"/}" ;;
                SOURCE_FILE) SOURCE_FILE="${value//\"/}" ;;
                COMPILED_FILE) COMPILED_FILE="${value//\"/}" ;;
                COMPILE_CMD) COMPILE_CMD="${value//\"/}" ;;
                RUN_CMD) RUN_CMD="${value//\"/}" ;;
            esac
        done < "$properties_file"
    }
```

В функции main происходит перебор всех директорий внутри каталога tests (в котором хранятся тесты для всех языков программирования) и пытается считать properties в папке, которая соответствует некоторому языку программирования. Если такого файла нет, то язык программирования будет пропущен.

Список директорий можно получить с помощью функции find, которая определена в заголовке цикла for:

```
    main() {
        local base_dir="."

        for dir in $(find "$base_dir" -type d); do
            local properties_file="${dir}/properties"
            if [[ -f "$properties_file" ]]; then
                read_properties "$properties_file"

                for version in $VERSIONS; do
                    printf "Running test for $NAME (v$version)
\n"
                    local
compile_cmd="${COMPILE_CMD//\{VERSION\}/$version}"
```

```
                    local
run_cmd="${RUN_CMD//\{VERSION\}/$version}"
                    run_test      "$version"      "$compile_cmd"
"$run_cmd"
                done

                printf "\n"
            fi
        done
    }
```

Следующим этапом развития данного модуля может стать интеграция изолирующей среды в контейнер, которая обеспечит дополнительную защиту информационной системы, а также будет замерять время и память, которые приходятся на выполнение той или иной программы.

Созданный каталог с тестами достаточно скопировать в контейнер с помощью команды COPY в Dockerfile, а также вызвать команду для запуска скрипта «run_tests.sh»:

```
COPY tests /usr/local/tests
CMD ["/usr/local/tests/run_tests.sh"]
```

```
@debian:~/Desktop/ExecEngine/compilers$ sudo docker run -it compiler_test
Running test for Go (v1.22.4)
Hello, world! (Go)

Running test for PHP (v20.15.0)
Hello, world! (PHP)
Running test for C (v13.3.0)
Hello, world! (C)

Running test for Java (v22)
Hello, world! (Java)

Running test for NodeJS (v20.15.0)
Hello, world! (NodeJS)

Running test for NASM (v2.16.03)
Hello, world!
Running test for Kotlin (v2.0.0)
Hello, world! (Kotlin)
```

Рисунок 8. Результат проведения тестирования работоспособности установленных языков программирования

Таким образом, в рамках данной статьи был представлен контейнер с поддержкой компиляции и запуска программ на 15-и языках программирования. Данный способ сборки приложений позволяет легко менять список версий для каждого языка, а также добавлять новые языки программирования, что обеспечивает расширяемость и гибкость настройки модуля. Данный контейнер можно использовать в разработке онлайн-компиляторов, обучающих и тестирующих систем. Также в рамках данной

статьи был продемонстрирован процесс тестирования установленных языков программирования в рамках созданного контейнера.

Библиографический список

1. Кузовенков А.О. Совершенствование процесса автоматической проверки задач по программированию // В сборнике: Прикладная математика и информатика: современные исследования в области естественных и технических наук. Материалы VI Международной научно-практической конфе2020. С. 864-869.
2. Половикова О.Н., Маничева А.С., Журавлева В.В. Разработка программной платформы для тестирования прикладных решений на основе технологии контейнерной виртуализации // КИО. 2023. №3.
3. Бондаренко А.С., Зайцев К.С. Управление контейнерами при построении распределенных систем с микросервисной архитектурой // International Journal of Open Information Technologies. 2023. №8.
4. Mareš M. Security of Grading Systems // Olympiads in informatics. 2021. Т.15. С. 37-52.
5. Docker BuildPack | GitHub URL: https://github.com/docker-library/docs/tree/master/buildpack-deps. (Дата обращения: 26.06.2024)