DOI 10.37882/2223-2966.2024.7-2.24

АВТОМАТИЗАЦИЯ ЖИЗНЕННОГО ЦИКЛА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ: РЕАЛИЗАЦИЯ БАЗОВОГО CI/CD КОНВЕЙЕРА

SOFTWARE LIFECYCLE AUTOMATION: IMPLEMENTATION OF THE BASIC CI/CD PIPELINE

A. Nikolaeva
A. Zabrodin

Summary. CI/CD pipelines are a relevant and convenient way to automate the process of building and deploying an application now.
The CI/CD concept includes not only basic stages, such as getting source code, its compilation and deployment, but can also be supplemented by various automated tests, static and dynamic code analysis, as well as other tools aimed at improving the quality and reliability of the developed software.
In this article, we will look at a practical example of implementing a basic CI/CD pipeline using popular and accessible tools Jenkins, Docker and Kubernetes.

Keywords: Continuous Integration (CI), Continuous Delivery (CD), Jenkins, Docker, Kubernetes.

Николаева Александра Ильинична
Петербургский государственный университет путей сообщения императора Александра I
sasha.fedotova.01@mail.ru
Забродин Андрей Владимирович
кандидат исторических наук, доцент,
Петербургский государственный университет путей сообщения императора Александра I
zabrodin@pgups.ru

Аннотация. На сегодняшний день использование CI/CD конвейеры является актуальным и удобным способом автоматизации процесса сборки и развертывания программного обеспечения (ПО).
Концепция CI/CD включает в себя не только базовые этапы, такие как получение исходного кода, его сборка и развертывание, а также может дополняться проведением различных автоматизированных тестов, статическим и динамическим анализом кода, а также другими инструментами, направленными на повышение качества и надежности разрабатываемого программного обеспечения.
В данной статье исследуется практический пример реализации базового CI/CD конвейера на популярных и доступных инструментах Jenkins, Docker и Kubernetes.

Ключевые слова: Continuous Integration (CI), Continuous Delivery (CD), Jenkins, Docker, Kubernetes.

Введение

Для понимания процесса реализации базового конвейера непрерывной интеграции и доставки необходимы как теоретические знания, так и практические навыки. В рамках данной статьи мы ставим перед собой задачу на практическом примере рассмотреть этапы этого процесса, изучить необходимые инструменты для их реализации и внедрить базовый конвейер. Дополнительно осуществим модернизацию в виде внедрения статического анализа кода, с целью обеспечения более надежной разработки и качества программного продукта

В классическом варианте процессы в CI/CD конвейере включают в себя два этапа: CI (непрерывная интеграция) и CD (непрерывная доставка), каждый из которых включает отдельные процессы и этапы, направленные на автоматизацию различных аспектов разработки и развертывания программного обеспечения. [1][2]

Определим исходные условия для нашего примера:

- код хранится в репозитории (GitHub),
- приложение написано на языке программирования Python,
- приложение работает в контейнере (Docker),
- используется публичное хранилище контейнеров (Docker Hub),
- приложение разворачивается в оркестраторе (Kubernetes).

Для реализации CI/CD конвейера будет использоваться Jenkins. Инструмент был выбран за его доступность (бесплатность и наличие нескольких вариантов разворачивания), хорошую документацию, большое сообщество, и как следствие, разнообразие расширений (плагинов) [3].

Сборка приложения с помощью контейнеризации

Контейнеризация — это упаковка программного кода с библиотеками операционной системы и всеми зависимостями, которые необходимы для выполнения кода. Контейнеры запускаются и работают абсолютно одинаково на любой инфраструктуре (Linux, Windows, кластер K8S и другое), при этом не важно, где он запускается — результат работы будет одинаковый.

Одним из наиболее популярных инструмент для контейнеризации приложений на данный момент — Docker.

Docker — это инструмент, помогающий разработчикам создавать, распространять и запускать контейнерные приложения.

Для формирования Docker-образа требуется: установить Docker, подготовить Dockerfile с инструкциями формирования образа и запустить процесс сборки через команду«dockerbuild»ссоответствующимипараметрами[4].

Основные инструкции Dockerfile:

- FROM — инструкция, указывающая базовый образ для всех последующих инструкций. Каждый Dockerfile должен начинаться с конструкции FROM.
- WORKDIR — инструкция, указывающая рабочую директорию для последующих команд RUN, CMD, ENTRYPOINT, COPY, ADD.
- ENV — инструкция, создающая в образе переменную среды и устанавливающая ей указанное значение.
- COPY — инструкция, копирующая файлы или директории из указанного источника на файловой системе хоста в образ по указанному пути.
- RUN — инструкция для выполнения указанной shell команды.
- CMD — инструкция, указывающая shell команду, которая выполнится при запуске контейнера из этого образа.
- EXPOSE — инструкция о том, что определенный порт должен быть доступен для обмена данными в контейнере.

Подробную информацию по каждой команде и остальные команды можно посмотреть в официальной документации Docker [4].

После сборки Docker-образа его можно использовать локально или разместить образ в хранилище образов (Registry) с помощью команды «docker push». Docker registry — это инструмент, обеспечивающий хранение и обмен Docker-образами. По умолчанию Docker использует Docker Hub (хранилище образов, созданное для разработчиков и участников с открытым исходным кодом), но можно использовать и частное хранилище образов или другое готовое хранилище, например, GitHub Packages.

После завершения процесса сборки приложения переходим к этапу тестирования.

Тестирование ПО

Тестирование ПО может проводиться на кодовой базе приложения, или же на работающем экземпляре приложения.

В рамках статьи рассматривается тестирование кода модульными тестами (unit-тесты). Модульные тесты позволяют проверить на корректность отдельные модули исходного кода программы в изоляции от других частей программы.

Результаты тестирования, как правило, отображаются в отчете, содержащем информацию о том, прошли ли все тесты, если не прошли — то какие и т. п. Также в таком отчете полезно увидеть какой объем кода покрыт тестами. Для этого могут использоваться встроенные функции в инструменты для тестирования (при их наличии), или же дополнительные инструменты.

Для тестирования приложений на Python может использоваться фреймворк pytest с расширение pytest-cov, позволяющим определить покрытие кода тестами. [5]

При запуске тестов Pytest позволяет определить также параметры для плагина pytest-cov, например:

- указание путей или пакетов для измерения покрытия,
- указание типа создаваемого отчета (xml/json, с указанием непокрытых тестами строк кода и другое) и др.

Следует отметить, что фреймворк pytest и его плагин pytest-cov позволяют проверить работу приложения и узнать какой объем кода проверяется, но только по заранее написанным тестам.

Для нахождения возможных уязвимостей в коде без написания тестов используется статический анализ кода. Выбор инструмента для проведения статического анализа кода также зависит от используемого языка программирования. Например, для Python часто используется инструмент Pylint. [6]

Если не использовать дополнительных параметров Pylint выводит в консоль сообщения о найденных недостатках и итоговую оценку качества кода по 10-бальной шкале. Сообщения делятся на 6 категорий: I (Informational) — информационные сообщения, R (Refactor) — сообщения о нарушениях метрики «хорошей практики», C (Convention) — сообщения о нарушениях стандартов кодирования, W (Warning) — предупреждения о стилистических проблемах или незначительных проблемах с программированием, E (Error) — сообщения об ошибках, связанных с важными проблемами программирования, F (Fatal) — сообщения об ошибках, препятствующих дальнейшей обработки данных pylint.

Pylint позволяет передавать множество опций, для настройки сканирования, например:

- настроить сканирование только на определение ошибок уровня Error и Fatal,

- пропустить сканирование указанных файлов,
- указать категории сообщений, при наличии которых сканирование должно завершаться ошибкой,
- настроить формат выводимого отчета и др.

В целом Pylint предназначен для более глубокого анализа кода, выявления потенциальных проблем и обеспечения высокого стандарта качества и читаемости кода.

После прохождения модульных тестов и статического анализа кода можно разворачивать приложение.

Разворачивание приложения

Для запуска и управления приложениями в контейнерах обычно используют оркестраторы контейнеров. Такие инструменты помогают управлять большим количеством контейнеров и потребляемыми ресурсами, удобно масштабировать их, управлять передаваемыми секретами и другое.

Популярным инструментом для оркестрации является Kubernetes (K8s). Для работы с ним необходимо развернуть кластер, выбрать необходимые ресурсы и подготовить для них файлы разворачивания в Yaml формате и применить их.

В рамках данной статьи процесс развертывания K8s кластера подробно не рассматривается. При этом следует отметить, что Kubernetes кластер может быть создан следующими способами:

- развернут вручную на виртуальных машинах (хорошие инструкции можно посмотреть в источниках [7], [8]),
- выдан с помощью облачных провайдеров,
- создан с помощью minikube (инструмент для запуска Kubernetes кластера из одного узла).

Kubernetes предоставляет большое количество объектов для управления, для их описания можно воспользоваться конфигурационными файлами и создать буквально одной командой.

После создания объектов Kubernetes будет самостоятельно отслеживать их состояние и при необходимости править его для соответствия с заданной конфигурацией (например, если какой-нибудь контейнер завершил работу с ошибкой и под упал, Kubernetes пересоздаст его). K8s имеет широкие возможностей для управления разворачиваемыми приложениями, подробнее о них можно узнать в документации [7].

Перейдем к организации CI/CD конвейера, и описанию рассмотренных этапов в нём.

Реализация CI/CD конвейера

Для обеспечения непрерывной интеграции и доставки необходимо автоматизировать процессы сборки, тестирования и разворачивания приложения, чтобы они запускались при каждом изменении в репозитории. Для достижения этой цели воспользуемся инструментом Jenkins.

Процесс установки и настройки Jenkins вынесен за рамки данной статьи, однако вы можете найти соответствующие инструкции в официальной документации [9].

Определение конвейера записывается в текстовый файл (называемый Jenkinsfile), который может храниться в репозитории проекта или в самом Jenkins. Jenkinsfile может быть написан в 2-х вариантах синтаксиса — декларативный (Declarative) и скриптовой (Scripted).

Декларативные и скриптовые конвейеры устроены принципиально по-разному. Декларативный синтаксис появился позже скриптового, предоставляет больше синтаксический функций и разработан для упрощения написания и чтения кода конвейера. Рассмотрим подробнее структуру декларативного конвейера.

Jenkinsfile в декларативном варианте представляет собой многоуровневую структуру. На верхнем уровне — пайплайн (pipeline), который состоит из этапов (stages). Блок этапа определяет концептуально отдельное подмножество задач (например, сборка, тестирование и развертывание) и состоит из шагов (steps), шагами могут быть как отдельные операторы, вызовы функций, так и скрипты (script) или нестандартные конструкции-обертки для кода, предоставляемые различными плагинами.

После описания конвейера есть несколько способов запустить его работу: вручную, по расписанию или некоторым автоматизированным способом. Классический вариант автоматизированного запуска — через Webhook, например, из репозитория с кодом при добавлении изменений.

Webhook (вебхук) — это механизм, который позволяет веб-приложению автоматически отправлять HTTP-запросы на определенный URL-адрес при наступлении определенных событий. В контексте CI/CD, Webhook может использоваться для уведомления системы CI о том, что произошли изменения в репозитории кода, после чего запускается процесс автоматической сборки, тестирования и развертывания приложения.

Для демонстрации реализации CI/CD конвейера в Jenkins рассмотрим конкретный пример, представив для наглядности что у нас есть веб-приложение, реализующее мини-игру «камень-ножницы-бумага». Приложение не имеет аутентификации, нет базы данных или хранилища секретов. Оно ожидает на вход запрос с ходом пользователя («rock» — камень, «paper» — бумага,

«scissors» — ножницы), проверяет что ход корректный, случайным образом выбирает свой ход (из тех же вариантов) и определяет кто выиграл. Помимо основного функционала, также реализован метод для получения списка доступных ходов, для проверки работы приложения будут использоваться оба описанных метода.

Код приложения и дополнительные файлы можно посмотреть в репозитории [10].

Ниже представлен реализованный Jenkinsfile, этапы которого рассмотрим подробнее.

```
    def app
pipeline {
  agent {label 'python && kubectl'}

    // Параметры конвейера, позволяют передать значения через UI Jenkins или иным способом для управления работой конвейера parameters {
    string(name: 'docker_username', defaultValue: 'alexandrafedotova', description: 'Username for Docker Hub')
    string(name: 'app_name', defaultValue: 'rps_game', description: 'Docker image name')
    string(name: 'branch', defaultValue: 'master', description: 'Project branch name')
    string(name: 'repo_url', defaultValue: 'https://github.com/AlexandraFedotova/RPS_game.git', description: 'Git repository url')
    string(name: 'k8s_namespace', defaultValue: 'rps-game-develop', description: 'K8s namespaces for resource creation')
  }

    stages {
        // Получение исходного кода приложения из репозитория, код берется из указанного в параметрах репозитория и ветки
     stage('Get source code'){
       steps {
         checkout scmGit(branches: [[name: «${branch}»]], userRemoteConfigs: [[url: «${repo_url}»]])
       }
     }
     // Сборка приложения с помощью Docker, ожидается что в проекте находится корректный Dockerfile
     stage('Build image') {
       steps {
         echo «Build app image: ${docker_username}/${app_name}:${env.BUILD_TAG}»
         script {
           app = docker.build(«${docker_username}/${app_name}:${env.BUILD_TAG}»)
         }
       }
     }
     // Прохождение тестов и статического анализа кода, команды для установки зависимостей, прохождение тестов и
     // сканирования имеют в себе указания на файлы и директории в проекте. Если их не будет команды завершатся с ошибкой
     stage('Run tests and SAST') {
       steps {
         withPythonEnv('python3'){
           // Install tests requirements
           sh 'pip install -r requirements-tests.txt'
           // Run pytest
           catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {
             sh 'pytest --cov-fail-under=70 --cov=game --cov-report term-missing tests'
           }
           // Run pylint
           catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {
             sh 'pylint --fail-under 7 --fail-on E --output-format json2 game'
           }
         }
       }
     }
     // Отправка образа в хранилище образов Docker Hub (ожидается наличие credentials в Jenkins для подключения к Docker Hub)
     stage('Push image') {
       steps {
         script {
           withDockerRegistry(credentialsId: 'DockerCreds') {
             app.push()
             app.push('latest')
           }
         }
       }
     }
     // Разворачивание приложения в Kubernetes, используются файлы конфигурации из проекта, переданное пространство имен K8s,
     // Для подключения к кластеру используется kubeconfig файл, который должен быть заранее добавлен в Jenkins credentials
     stage('Deploy app') {
       steps {
         withKubeConfig([credentialsId: 'KubeConfig', namespace: «${k8s_namespace}»]) {
           sh 'kubectl apply -f k8s/game-service.yaml -f k8s/game-deployment.yaml'
         }
       }
     }
   }
```

```
  post {
    always {
      cleanWs()
    }
  }
}
```

В разделе pipeline Jenkinsfile-а 4 блока:

1. agent — указание о том на каком узле Jenkins-а должен выполняться конвейер. Для всех узлов в Jenkins можно задавать лейблы, любые описания узлов, по которым с ними можно работать.
2. parameters — объявленные параметры конвейера, они могут использоваться в ходе работы конвейера. Для всех параметров определены значения по умолчанию, также их можно переопределить при запуске конвейера.
3. stages — этапы конвейера,
4. post — описание действий, выполняющихся после завершения основных этапов конвейера.

В реализованном конвейере 5 основных этапов (смотри рис. 1):

1. получение кода из git репозитория,
2. сборка приложения,
3. прохождение модульных тестов и статического анализа кода,
4. отправка собранного артефакта (Docker-образа) в Docker Hub,
5. разворачивание приложения в K8s кластере.

Помимо 5 основных этапов можно заметить 2 не описанных прямо в блоке stages в Jenkinsfile-е: Declarative: Checkout SCM, Declarative: Post Actions. Первый этап отвечает за получение Jenkinsfile-а из репозитория, второй описан в блоке post и в данном случае — очищает рабочую область на агенте, на котором выполнялся конвейер.

Получение кода из репозитория реализовано с помощью расширения Jenkins Pipeline: SCM Step и кроме получения данных репозитория и нужной ветки ничего не делает.

При сборке приложения использовался плагин Jenkins-а: Docker Pipeline. Плагин предоставляет удобный способ выполнить наиболее частые операции Docker, например, собрать образ, отправить образ в хранилище, выполнить код внутри образа и другое.

Для сборки Docker образа использовался следующий Dockerfile:

```
#Dockerfile
FROM python:3.12-rc
WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
COPY main.py .
COPY game.py .
CMD python main.py
EXPOSE 5000
```

Далее полученный образ будет использоваться на этапе отправки в хранилище, реализованного с помощью того же плагина.

На этапе тестирования создается виртуальное окружения в Python, в которое устанавливаются зависимости (помимо основных зависимостей добавлены pytest, pytest-cov и pylint). После чего в блоках catchError по отдельности выполняются команды по модульному тестированию и статическому анализу кода. Блок catchError используется для того, чтобы даже при упавших тестах и сканировании конвейер продолжил работу и дальнейшие шаги по отправке образа в хранилище и разворачивании были пройдены. Такое поведение допустимо в некоторых случаях (например, в среде разработки) и может контролироваться далее при желании. Однако, данный этап будет подсвечен в UI Jenkins-а как упавший (при упавших результатах тестирования и статического анализа кода).

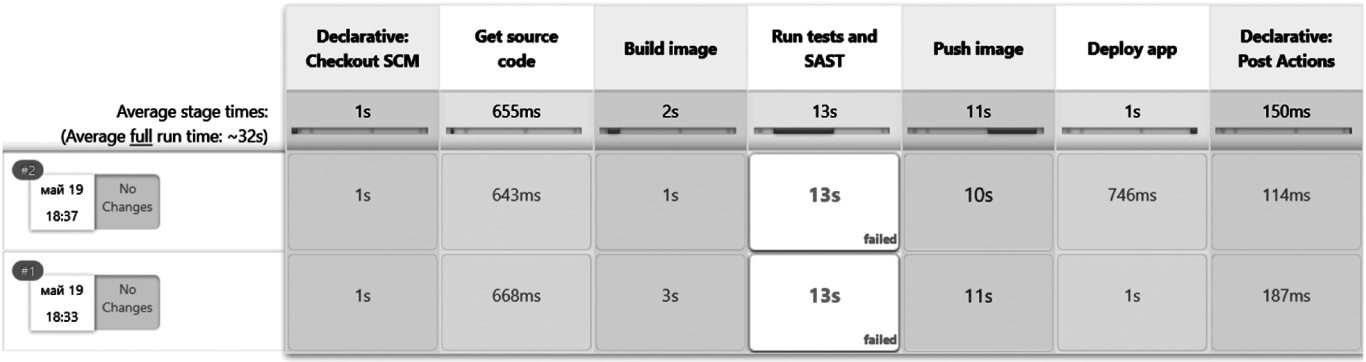

Рис. 1. Результаты выполнения конвейера в пользовательском интерфейсе Jenkins-а

```
+ pytest --cov-fail-under=70 --cov=game --cov-report term-missing tests
...                                                                  [100%]

---------- coverage: platform linux, python 3.10.12-final-0 ----------
Name               Stmts   Miss  Cover   Missing
------------------------------------------------
game/__init__.py       2      0   100%
game/main.py          19     19     0%   1-30
game/rps_game.py      24      9    62%   21, 27-34
------------------------------------------------
TOTAL                 45     28    38%

FAIL Required test coverage of 70% not reached. Total coverage: 37.78%
3 passed in 0.03s
```

Рис. 2. Результаты выполнения модульных тестов

Для запуска модульных тестов использовалась команда:

pytest --cov-fail-under=70 --cov=game –-cov-report term-missing tests

Команда запускает прохождение тестов, находящихся в директории tests и подсчет покрытия кода тестами. В параметрах указан модуль для вычисления покрытия кода — game, а также дополнительные параметры для отчета по покрытию, а именно: добавить в отчет указания о непокрытых участках кода и добавить условия для успешного завершения команды — покрытия кода тестами должно быть не менее 70 % (см. рис. 2).

Для запуска статического анализа кода использовалась команда:

pylint --fail-under 7 --fail-on E --output-format json2 game

В данной команде указаны модуль для сканирования (game), минимальная оценка для успешного прохождения анализа (7), а также добавлено ограничение — если будут найдены ошибки (Error) — сканирование будет считаться неуспешным. Для удобочитаемости результаты сканирования выводятся в json формате. Часть отчета Pylint представлена на рисунке 3.

```
"statistics": {
    "messageTypeCount": {
        "fatal": 0,
        "error": 0,
        "warning": 0,
        "refactor": 2,
        "convention": 11,
        "info": 0
    },
    "modulesLinted": 5,
    "score": 6.75
}
```

Рис. 3. Блок statistics из отчета Pylint

Последний этап — разворачивание приложение. Для этого используем утилиту kubectl и конфигурационные файлы, хранящиеся в репозитории. Для подключения к нужному кластеру используется плагин Kubernetes CLI и kubeconfig файл.

Kubeconfig файл — это файл, использующийся для настройки доступа к кластерам. С этим файлом любой может получить доступ к кластеру, поэтому он является секретом и хранится в Jenkins credentials.

Jenkins credentials — это способ для хранения секретных данных в Jenkins-е и их использование в конвейерах.

Файлы, используемые для разворачивания приложения в Kubernetes представлены ниже:

```
# game-deployment.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
 name  : rps-game-deployment
 labels:
   app: rps-game-app
spec:
 replicas: 2
 selector:
  matchLabels:
    app: rps-game-app
 template:
  metadata:
   labels:
     app: rps-game-app
  spec:
   containers:
    — name : rps-game
      image: alexandrafedotova/rps_game_test:latest
      ports:
```

```
        — containerPort: 5000

# game-service.yaml
---
apiVersion: v1
kind: Service
metadata:
  name  : rps-game-service
  labels:
    owner: alexandrafedotova
spec:
  selector:
    app: rps-game-app
  ports:
    — port     : 80
      targetPort: 5000
  type: LoadBalancer
```

В результате работы конвейера:

1. собран Docker-образ и размещен в Docker Hub под названием «alexandrafedotova/rps_game_test: latest»,
2. проведено модульное тестирование приложения (покрытия кода составило 38 %, поэтому тесты считаются «упавшими»),
3. проведён статический анализ кода (найдено 2 уязвимости уровня R и 11 уровня C, общая оценка — 6.75, поэтому статический анализ кода также не считается успешно пройденным),
4. приложение развернуто в кластере Kubernetes в пространстве имен «rps-game-develop»: создан 1 деплоймент, 2 пода и 1 сервис (см. рисунок 4).

Развернутое приложение доступно по внешнему адресу сервиса (95.174.89.32), смотри рисунок 5.

Несомненно, что предпочтительным подходом является настройка доступа по DNS-именам, а не по IP-адресам, но это оставляет пространство для последующих улучшений и доработок.

Итоги

CI/CD процессы полностью охватывают цикл разработки, начиная с написания кода и заканчивая его разворачиванием в рабочей среде. От того, насколько они правильно организованы, зависит удобство его использования и эффективность.

В данной статье на практическом примере было рассмотрено построение базового конвейера с использованием популярных и бесплатных инструментов. В дополнении к этапам сборки и развертывания, в конвейер включены шаги тестирования (только модульные тесты) и прохождения статического анализа кода с помощью Pylint.

Полученный конвейер достаточно легко модифицировать: можно добавить дополнительные шаги и проверки, например сбор метрик кода, а также вынести запуск тестов и анализа кода в отдельный скрипт, что позволит сделать процесс более унифицированным.

```
alex@DESKTOP-E0JHEA3:~$ kubectl get pods -n rps-game-develop
NAME                                  READY   STATUS    RESTARTS   AGE
rps-game-deployment-765655b4-x9vq8    1/1     Running   0          15m
rps-game-deployment-765655b4-xgdn2    1/1     Running   0          15m
alex@DESKTOP-E0JHEA3:~$ kubectl get deployments -n rps-game-develop
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
rps-game-deployment   2/2     2            2           15m
alex@DESKTOP-E0JHEA3:~$ kubectl get services -n rps-game-develop
NAME               TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)        AGE
rps-game-service   LoadBalancer   10.97.156.191   95.174.89.32   80:32593/TCP   15m
alex@DESKTOP-E0JHEA3:~$
```

Рис. 4. Созданные ресурсы K8s

```
alex@DESKTOP-E0JHEA3:~$ curl 'http://95.174.89.32/paper'
"Bot move is scissors. Bot win"alex@DESKTOP-E0JHEA3:~$
alex@DESKTOP-E0JHEA3:~$ curl 'http://95.174.89.32/paper'
"Bot move is paper. No one win"alex@DESKTOP-E0JHEA3:~$
alex@DESKTOP-E0JHEA3:~$ curl 'http://95.174.89.32/moves'
["rock","paper","scissors"]alex@DESKTOP-E0JHEA3:~$
alex@DESKTOP-E0JHEA3:~$
```

Рис. 5. Запросы к развернутому приложению

ЛИТЕРАТУРА

1. Вехен Джульен. Безопасный DevOps. Эффективная эксплуатация систем. — Санкт-Петербург: Питер, 2020. — 432 с. — ISBN 978-5-4461-1336-1. — URL: https://ibooks.ru/bookshelf/365290/reading (дата обращения: 06.05.2024). — Текст: электронный.
2. Джин Ким, Патрик Дебуа, Джон Уиллис, Джез Хамбл. Руководство по DevOps. — Манн, Иванов и Фербер (МИФ), 2018. — 542 с — ISBN 978-5-00100-750-0. — Текст: непосредственный.
3. Рафал Лешко. Continuous Delivery with Docker and Jenkins. — Packt Publishing, 2024 — 332 с. — ISBN 9781787125230 — Текст: электронный.
4. Docker docs URL: https://docs.docker.com/ Режим доступа: свободный. [дата обращения: 17.04.2024]
5. Pytest: документация на русском языке URL: https://pytest-docs-ru.readthedocs.io/ru/latest/ Режим доступа: свободный. [дата обращения: 12.05.2024]
6. Pylint 3.1.0 documentation URL: https://pylint.readthedocs.io/en/stable/ Режим доступа: свободный. [дата обращения: 12.05.2024]
7. Документация по Kubernetes URL: https://kubernetes.io/ru/docs/home/ [дата обращения: 17.04.2024]
8. Установка и настройка кластера Kubernetes на Ubuntu Server URL: https://www.dmosk.ru/instruktions.php?object=kubernetes-ubuntu Режим доступа: свободный. [дата обращения: 17.04.2024]
9. Jenkins User Documentation URL: https://www.jenkins.io/doc/ Режим доступа: свободный. [дата обращения: 18.04.2024]
10. Репозиторий RPS_game https://github.com/AlexandraFedotova/RPS_game Режим доступа: свободный. [дата обращения: 19.04.2024]