



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
НА ТЕМУ:

«Метод автоматической генерации docker-файла для проекта на
языке python»

Студент группы ИУ7-84Б

(Подпись, дата)

Е.Д. Савинов

Руководитель ВКР

(Подпись, дата)

Э.С. Клышинский

Нормоконтролер

(Подпись, дата)

Д.Ю. Мальцева

2022 г.

РЕФЕРАТ

Расчетно-пояснительная записка 66 с., 20 рис., 2 табл., 19 источника.

КОНТЕЙНЕР, DOCKER, DOCKERFILE

Объектом работы является docker-файл

Целью выпускной квалификационной работы является разработка метода автоматической генерации docker-файла для проекта на языке Python.

Для достижения поставленной цели необходимо решить следующие задачи:

- проанализировать предметную область и существующие средства виртуализации;
- разработать концепцию метода автоматической генерации docker-файла для проекта на языке Python;
- спроектировать и реализовать метод автоматической генерации docker-файла для проекта на языке Python;
- экспериментальным путем проверить корректность работы разработанного метода.

СОДЕРЖАНИЕ

РЕФЕРАТ.....	5
ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ.....	9
ВВЕДЕНИЕ	10
1 Аналитический раздел.....	12
1.1 Аппаратная виртуализация	12
1.2 Виртуализация на уровне операционной системы	13
1.3 LXC	15
1.4 Docker.....	17
1.4.1 Объекты Docker	18
1.4.2 Docker-файл.....	20
1.5 Анализ средств для виртуализации.....	22
1.6 Анализ эффективности внедрения	22
1.6.1 Оценка, сравнение и планирование эффективности использования автоматизированных и человеческих ресурсов	22
1.6.2 Ценность специалиста на рынке труда	23
1.6.3 Временные и материальные затраты на подготовку специалиста начального уровня	26
1.6.4 Время развертывания проекта	26
1.6.5 Экономия при автоматическом развертывании проекта	27
1.7 Выводы из аналитического раздела	27
2 Конструкторский раздел	28
2.1 Требования к методу автоматической генерации docker-файла	28

2.2 IDEF0	28
2.3 Описание алгоритма автоматической генерации docker-файла.....	29
2.3.1 Поиск фреймворка Flask	29
2.3.2 Поиск фреймворка Django	30
2.3.3 Поиск конфигурации подключения баз данных фреймворка Django	31
2.3.4 Поиск конфигурации подключения баз данных фреймворка Flask .	32
2.3.5 Поиск React.....	33
2.3.6 Структура сети	34
2.4 Допустимые конфигурации подключения баз данных	35
2.4.1 Django.....	35
2.4.2 Flask.....	36
2.5 Требования к выходным данным	37
2.6 Выводы из конструкторского раздела	37
3 Технологический раздел	38
3.1 Выбор языка программирования.....	38
3.2 Выбор языка программирования.....	38
3.3 Реализация алгоритмов	39
3.4 Пользовательский интерфейс	53
3.5 Функциональное тестирование	54
3.6 Выводы из технологического раздела	56
4 Исследовательский раздел.....	58
4.1 Описание эксперимента	58
4.2 Результаты проведенного эксперимента	58

4.3 Выводы из исследовательского раздела	59
ЗАКЛЮЧЕНИЕ	61
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	62
ПРИЛОЖЕНИЕ А.....	64
ПРИЛОЖЕНИЕ Б	66

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ВМ — виртуальная машина.

ОС — операционная система.

БД — базы данных.

IT — Information Technology.

ПО — программное обеспечение.

ВВЕДЕНИЕ

В последние годы все чаще технологии виртуализации применяются в разработке ПО.

Эти технологии составляют основу облачных вычислений. Сервер не может существовать без виртуализации. Виртуализация позволяет запускать множество приложений на одном узле, разделяя ресурсы между приложениями. Технология позволяет решить проблему поддержки зависимостей различных версий как одной программы, так и библиотек. Также актуальна проблема поддержки объема legacy-кода.

При решении проблемы конфликта зависимостей, запуска приложений различных версий стоит вопрос об эффективном внедрении автоматического развертывания контейнеров с максимальным уменьшением человеческого ресурса.

Сегодня на рынке недостаточно специалистов уровня, способных решить эти вопросы. Обучение их требует большого количества времени и материальных затрат. Их содержание требует высоких зарплат, социальных гарантий, создание соответствующих условий труда и т. д. Также работа человека всегда по временным затратам гораздо выше работы машины. Человек может потратить на решение задачи неопределенное количество времени, тогда как у машины это время четко регламентировано на каждую операцию.

Целью данной работы является разработка метода автоматической генерации docker-файла для проекта на языке Python.

Для достижения поставленной цели необходимо решить следующие задачи:

- проанализировать предметную область и существующие средства виртуализации;
- разработать концепцию метода автоматической генерации docker-файла для проекта на языке Python;

- спроектировать и реализовать метод автоматической генерации docker-файла для проекта на языке Python;
- экспериментальным путем проверить корректность работы разработанного метода.

1 Аналитический раздел

1.1 Аппаратная виртуализация

Аппаратная виртуализация дает возможность запускать несколько виртуальных машин на одном компьютере. Можно создавать изолированные системы друг от друга.

Для создания виртуальных машин используется специальная технология – гипервизор. Гипервизор – это программный слой, который может контролировать и виртуализировать ресурсы хост-машины и позволяет виртуальным машинам выполняться одновременно на одной машине [1].

Существует два типа гипервизора [2]:

— гипервизор первого типа.

Гипервизор размещается на оборудовании и делает ресурсы доступными для гостевых операционных систем.

— гипервизор второго типа.

Гипервизор размещается в базовой операционной системе и поддерживает размещение других гостевых операционных систем. Существует риск того, что если базовая операционная система обнаружит какие-либо ошибки, то весь стек виртуализации (гостевые операционные системы) может выйти из строя.

На рисунке 1.1.1 представлена архитектура аппаратной виртуализации.

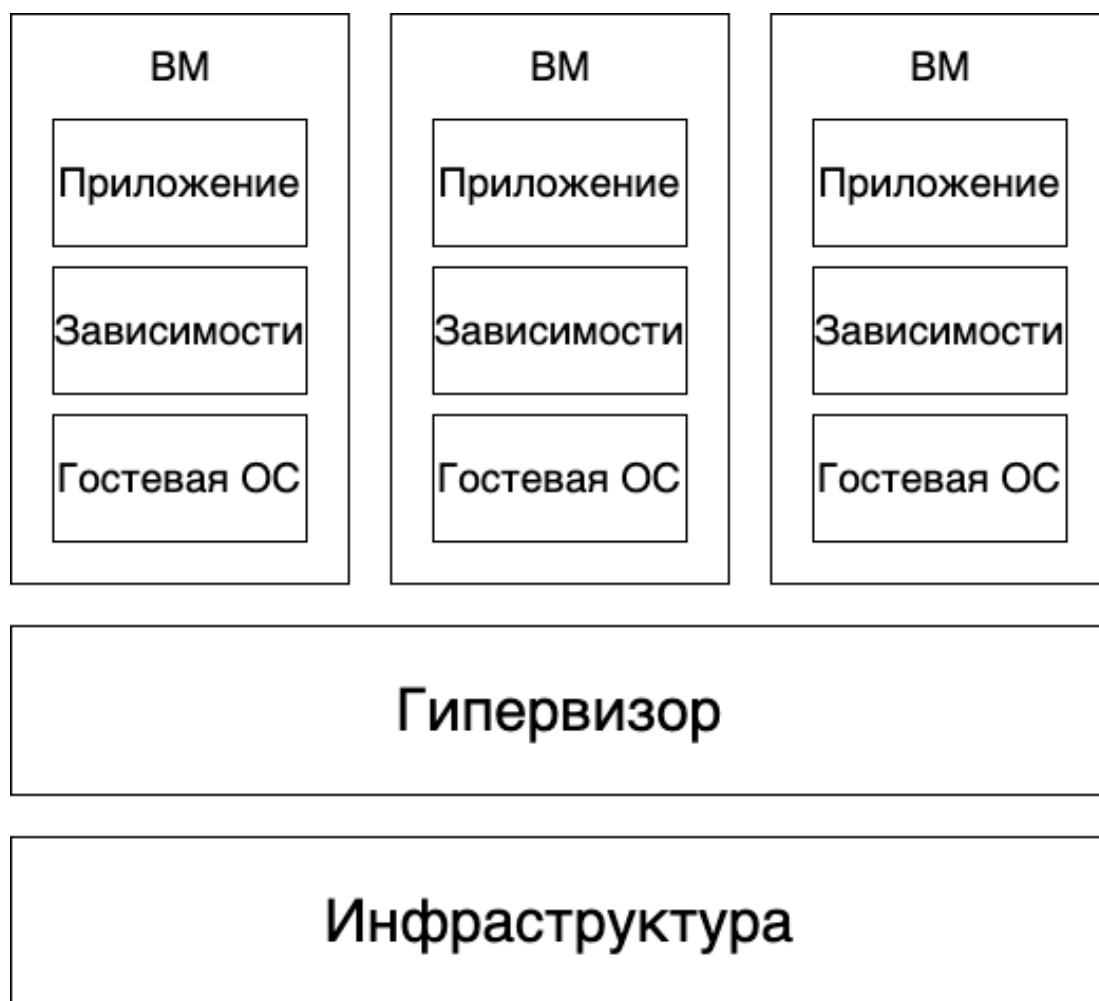


Рисунок 1.1.1 — Аппаратная виртуализация

За счет предоставления аппаратных интерфейсов виртуальные машины имеют упрощенное администрирование и поддержку. Повышенная безопасность достигается за счет полной изоляции виртуальной машины. Можно иметь только базовый образ, чтобы на его основе создавать среду для новых приложений. При смене инфраструктуры необходимо обновить только драйверы ОС хоста для дальнейшей работы виртуальных машин.

1.2 Виртуализация на уровне операционной системы

Виртуализация на уровне операционной системы позволяет виртуализировать над операционной системой. В отличие от аппаратной виртуализации, этот метод поддерживает только одну общую операционную систему.

На рисунке 1.2 представлена архитектура виртуализации на уровне операционной системы.

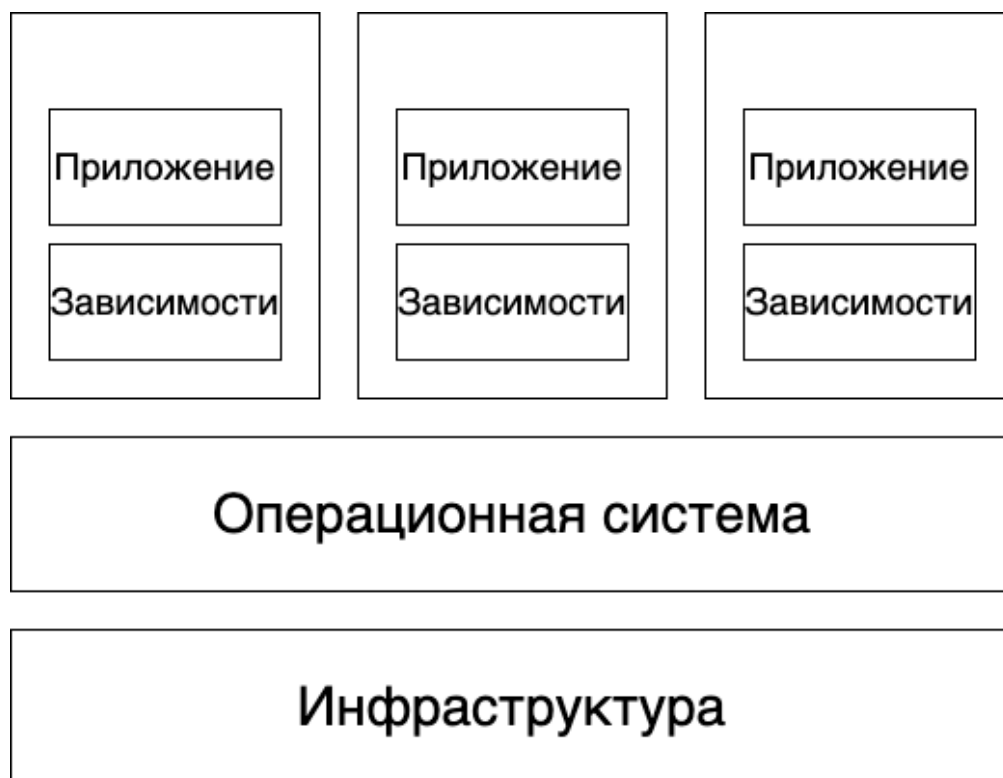


Рисунок 1.1.2 — Виртуализация на уровне операционной системы

Упакованные приложения представляют собой контейнеры. Контейнеризация — это подход к разработке программного обеспечения, при котором приложение, их зависимости и конфигурация упаковываются вместе в образ контейнера [3]. Это делается для того, чтобы программное обеспечение или приложение можно было запускать в любой среде и в любой архитектуре, независимо от этой среды или операционной системы инфраструктуры [4].

Каждый контейнер не нуждается в собственной гостевой операционной системе, поэтому требует значительно меньше ресурсов для развертывания. Можно быстро осуществлять горизонтальное масштабирование, например, для краткосрочных задач [4]. Данная технология позволяет настраивать производительность внутри контейнера для нужд каждого приложения.

1.3 LXC

LXC – это технология виртуализации на уровне операционной системы, поддерживаемая ядром Linux. Цель LXC - создать среду, приближенную к стандартной установке Linux, но без необходимости в отдельном ядре.

Контейнеры полагаются на следующие функции ядра Linux, чтобы получить изолированную область внутри хост-машины, без необходимости гипервизора [5]:

- control groups (cgroups);
- namespaces.

Технология контрольных групп используется для контроля над распределением, управлением и приоритизацией системных ресурсов.

Управление контрольными группами реализовано через systemd. Cgroups состоит из двух частей: ядра и подсистем. Рассмотрим 13 подсистем [6]:

- cpu.

Распределяет автоматические отчеты по использованию центрального процессора.

- cpuacct.

Создает автоматические отчеты по использованию ресурсов центрального процессора.

- cpuset.

Используется для привязки процессов в контрольной группе.

- memory.

Устанавливает лимиты и генерирует отчеты об использовании памяти задачами контрольной группы.

- devices.

Разрешает или блокирует доступ к устройствам.

- freezer.

Приостанавливает или возобновляет задачи в рамках контрольной группы.

- net_cls.

Устанавливает тег на сетевые пакеты, для идентификации пакетов, порождаемые определенной задачей в рамках контрольной группы.

— blkio.

Устанавливает лимиты на чтение и запись к блочным устройствам.

— net_prio.

Устанавливает динамические приоритеты по трафику.

— hugetlb.

Позволяет использовать большие страницы памяти для контрольных групп.

— pids.

Позволяет ограничить количество процессов, которые могут быть созданы в контрольной группе.

— rdma.

Позволяет ограничить использование rdma/ib для каждой группы.

Пространства имен (namespaces) отвечают за изоляцию контейнеров, гарантируют, что файловая система, имя хоста, пользователи, сетевая среда и процессы любого контейнера полностью отделены от остальной части системы.

Linux использует следующие пространства имен [7]:

— uts.

Позволяет процессу видеть отдельное имя хоста, отличное от фактического глобального пространства имен.

— pid.

Процессы в пространстве имен pid имеют другое дерево процессов. На уровне структуры данных процессы принадлежат одному глобальному дереву процессов, которое видно только на уровне хоста.

— mount.

Определяет какие точки монтирования должен видеть процесс. Если процесс находится в пространстве имен, то он будет видеть только подключения в этом пространстве имен.

— network.

Сетевое пространство имен предоставляет контейнеру отдельный набор сетевых подсистем. Отделяет контейнерную сеть от хост-сети.

— `ipcs`.

Пространство имен охватывает конструкции `ipcs`, такие как очереди сообщений POSIX.

1.4 Docker

Docker – это технология с открытым исходным кодом, которая решает проблемы развертывания и масштабирования путем отделения приложений от зависимой инфраструктуры. Она решает эти проблемы благодаря применению контейнеров, позволяющих упаковывать приложение со всеми зависимостями, включая структуру каталогов, метаданные, пространство процессов, номера сетевых портов и т.д. [8]

Docker использует архитектуру клиент (docker клиент) – сервер (docker демон). Клиент общается с так называемым демоном Docker, который берет на себя задачи создания, распределения и запуска контейнеров. Оба, клиент и сервер, могут работать в одной системе, также сервер может быть удаленным [9].

На рисунке 1.3 представлена архитектура Docker – клиент, демон и хост-компьютер.

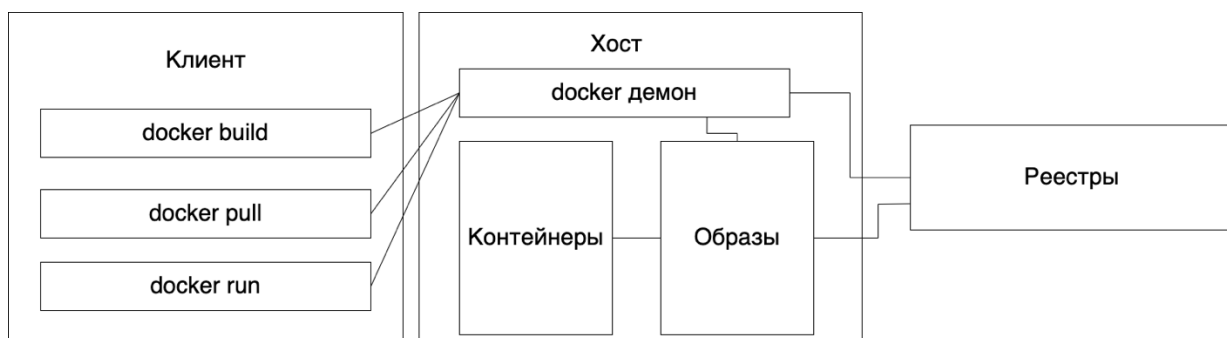


Рисунок 1.1.4 — Архитектура Docker

Docker Engine.

Является основной частью всей системы Docker. Механизм Docker Engine предоставляет эффективный и удобный интерфейс для запуска контейнеров,

который устанавливается на хост-компьютер и работает по архитектуре клиент-сервер. В него входит три основных компонента:

— server.

Выполняет инициализацию демона, который применяется для управления и модификации контейнеров, образов и томов.

— REST API;

Используется для взаимодействия клиента и демона.

— CLI;

Интерфейс командной строки используется для ввода команд Docker.

Docker Клиент.

Взаимодействует с демонами, получает или отдает информацию.

Docker Демон.

Docker Демон прослушивает запросы Docker API и управляет объектами Docker, такими как образы, контейнеры, сети и тома. Демон также может взаимодействовать с другими демонами для управления службами Docker.

Docker Реестры

В реестре хранятся образы Docker. Их можно запросить у любого демона, у которого есть соответствующий доступ. Docker Hub – общедоступный реестр и Docker по умолчанию настроен на поиск образов в нем.

1.4.1 Объекты Docker

Образы.

Docker-образ — это шаблон для создания Docker-контейнеров. Представляет собой исполняемый пакет, содержащий все необходимое для запуска приложения: код, среду выполнения, библиотеки, переменные окружения и файлы конфигурации [10].

Образ состоит из слоев. Каждое изменение записывается в новый слой. Благодаря такой организации можно распространять и развертывать только новые уровни, ускоряя процесс.

Контейнер.

Автономная виртуальная система, содержащая выполняющийся процесс, все файлы, зависимости, адресное пространство процесса и сетевые порты, необходимые приложению. Так как каждый контейнер имеет свое пространство портов, следует организовать их отображение в фактические порты на уровне Docker.

Тома.

Тома работают на контейнерах и предназначены для хранения данных. Другие программы получают доступ только через контейнер. Docker осуществляет управление томами через API Docker и CLI Docker. Один том может быть примонтирован к нескольким контейнерам.

Сети.

Сетевое пространство имен можно представить как стек с собственными сетевыми интерфейсами и соответствующими записями в таблице маршрутизации, действующими изолировано. Docker использует эту особенность для изоляции контейнеров и обеспечения безопасности. Docker предоставляет 4 сетевых драйвера:

— host.

Добавляет контейнер в сетевое пространство имен хоста. Контейнер и хост используют общее пространство имен. В этом режиме отсутствует возможность использования отображения портов.

— None.

Отсутствие поддержки сетевых взаимодействий.

— Bridge.

Используется по умолчанию, создает внутреннюю скрытую сеть для взаимодействия между контейнерами.

— Overlay.

Позволяет строить сети на нескольких хостах на Docker Swarm. У контейнеров есть свои адреса и подсети, могут напрямую обмениваться данными.

1.4.2 Docker-файл

Docker может автоматически создавать образы, читая инструкции из Dockerfile. Файл Dockerfile представляет из себя текстовый документ, содержащий все команды для сборки образа. С помощью команды Docker build пользователи могут производить автоматизированную сборку, которая выполняет последовательность инструкций в командной строке.

Dockerfile имеет следующий набор инструкций [11]:

— add.

Копирует файлы из контекста создания или из удаленных URL-ссылок в создаваемый образ. Если архивный файл добавляется из локального пути, то он будет автоматически распакован.

— cmd.

Запускает заданную инструкцию во время инициализации контейнера. Инструкция cmd замещается любыми аргументами, указанными в команде docker run после имени образа. В действительности выполняется только самая последняя инструкция cmd, а все предыдущие инструкции будут отменены.

— copy.

Используется для копирования файлов из контекста создания в образ. Имеет два формата: copy источник цель и copy [“источник”, “цель”].

— entrypoint.

Определяет выполняемый файл (программу) (и аргументы по умолчанию), запускаемый при инициализации контейнера. В эту выполняемую программу передаются как аргументы любые инструкции cmd или аргументы команды docker run, записанные после имени образа.

— env.

Определяет переменные среды внутри образа. На эти переменные можно ссылаться в последующих инструкциях.

— expose.

Сообщает механизму Docker о том, что в данном контейнере будет существовать процесс, прослушивающий заданный порт или несколько портов. Механизм Docker использует эту информацию при установлении соединения между контейнерами.

— `from`.

Определяет основной образ для Dockerfile. Все последующие инструкции выполняют операции создания поверх заданного образа. Эта инструкция обязательно должна быть самой первой в Dockerfile.

— `maintainer`.

Определяет метаданные об авторе “Author” для создаваемого образа в заданной строке. Обычно используется для записи имени автора образа и его контактных данных.

— `onbuild`.

Определяет инструкцию, которая должна выполняться позже, когда данный образ будет использоваться как основной уровень для другого образа. Это может оказаться полезным при обработке данных, добавляемых в образ-потомок.

— `run`.

Запускает заданную инструкцию внутри контейнера и сохраняет результат.

— `user`.

Задаёт пользователя для использования во всех последующих инструкциях `run`, `cmd`, `entrypoint`.

— `volume`.

Объявляет заданный файл или каталог как том. Если такой файл или каталог уже существует в образе, то он копируется в том при запуске контейнера.

— `workdir`.

Определяет рабочий каталог для всех последующих инструкций `run`, `cmd`, `entrypoint`, `add`, `copy`. Инструкцию можно использовать несколько раз.

Допускается указание относительных путей, при этом итоговый путь определяется относительно ранее указанного рабочего каталога workdir.

1.5 Анализ средств для виртуализации

Рассмотрим достоинства и недостатки средств для виртуализации:

Таблица 1.1.7 Сравнение средств для виртуализации

Название	Виртуализация с гипервизором	LXC	Docker
Безопасность	+	+	+
Отсутствие единой точки отказа	-	+	+
Упрощенное администрирование и поддержка	+	+	+
Отсутствие зависимости от платформы	+	-	+

Согласно таблице docker является наиболее подходящим для виртуализации, за счет безопасности контейнеров, отсутствия единой точки отказа, упрощенного администрирования и поддержки, отсутствия зависимости от платформы.

1.6 Анализ эффективности внедрения

1.6.1 Оценка, сравнение и планирование эффективности использования автоматизированных и человеческих ресурсов

Обеспечение эффективности внедрения информационных технологий стало особенно очевидной в период максимального подъёма их популярности и уверенности в безграничных возможностях их использования [12]. Около половины всех видов деятельности, которые люди занимают в мировой рабочей силе, потенциально могут быть автоматизированы путем адаптации имеющихся технологий [13].

Оценить эффективность информационных технологий и результат их внедрения в компанию непросто. Для оценки, сравнения и последующего выбора варианта для решения вопросов развертывания контейнеров автоматическим способом или с помощью человеческих ресурсов важно просчитать временные и материальные затраты на обучение и содержание специалиста данной сферы.

Вот лишь некоторые трудности, с которыми мы сталкиваемся:

- найти готового специалиста с требуемым опытом и удовлетворяющего запросам заказчика крайне сложно. Его зарплата должна быть не менее, чем средняя на рынке;
- существует другой путь.

Найти специалиста с базовым образованием. Обучить специфике работы в данной системе. Создать комфортные условия для долгосрочной работы. Но готовым специалистом он станет только через определенное время после высоких материальных затрат компании и потраченных временных ресурсов.

По самым приблизительным подсчетам, работа такого специалиста станет эффективной для компании только после 6-х месяцев нахождения в ней.

1.6.2 Ценность специалиста на рынке труда

Сегодня компании в сфере IT все чаще и чаще используют технологию контейнеризации при разработке своих приложений.

Исследование HeadHunter на основании описания более 300 тысяч IT-вакансий за 2016-2018 год показывает рынок спроса на специалистов в области DevOps-инженеров, которые специализируются на Docker [14].

DevOps – методика автоматизации рабочих процессов, существенно облегчающая задачи организации и способствующая действенным преобразованиям [15]. Это направление активно развивается, так как крупные компании нацелены на автоматизацию всех этапов разработки.

Самые востребованные ИТ-специальности

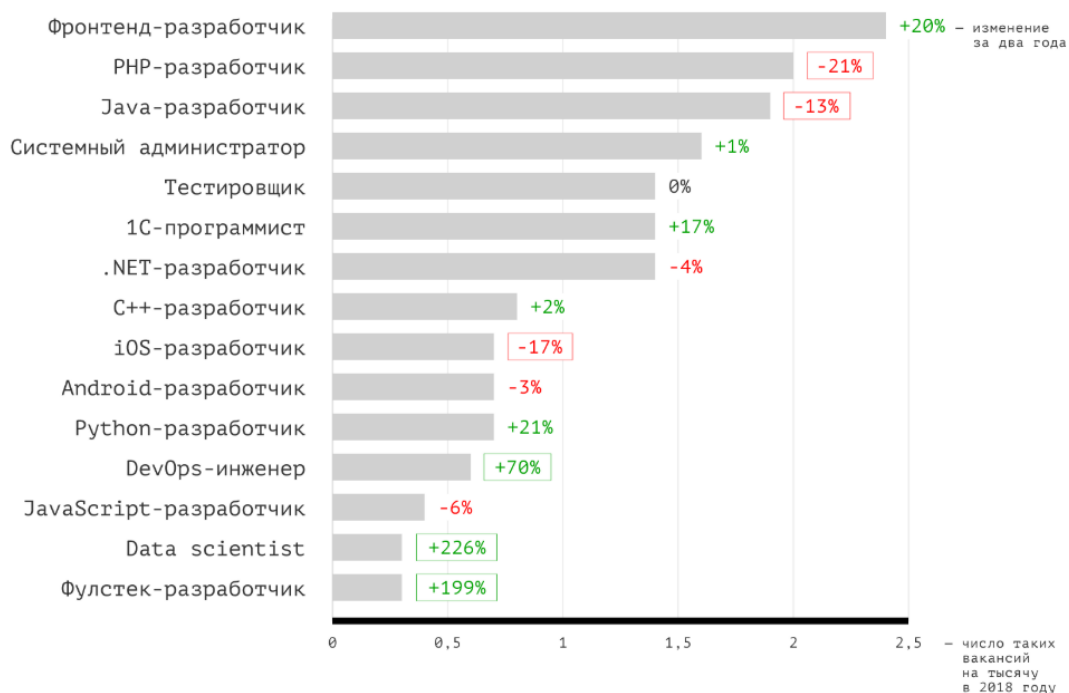


Рисунок 1.2.2 — Самые востребованные ИТ-специальности

Растет спрос на сотрудников. Как уже говорилось выше, обучение, внедрение и содержание подобного уровня специалистов очень затратное.

По сравнению с 2016 годов востребованность DevOps-инженеров выросла на 70%. График показывает большую необходимость новых людей в отрасли. Чтобы конкурировать с международными работодателями и удерживать ценных сотрудников, российские работодатели готовы идти на условия соискателей и платить высокие зарплаты по сравнению с другими специальностями.

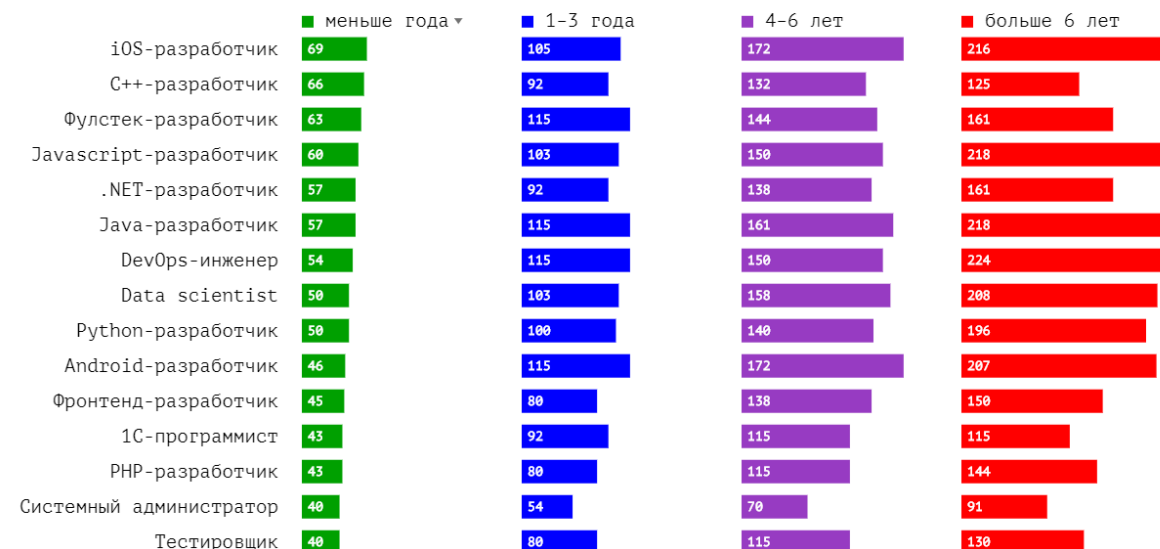
Рынок найма в ИТ перегрет. Появляются и растут экосистемы из-за чего компаниям нужно быстро масштабировать команды под амбициозные проекты. Поэтому они начинают агрессивно нанимать опытных специалистов, у которых растут зарплатные ожидания.

На рисунке 1.2.3 представлены зарплаты и требования к опыту работы в разных специальностях.

Зарплаты и требования к опыту работы в разных специальностях

ЗАРПЛАТА **ТРЕБУЕМЫЙ ОПЫТ**

Медиана зарплаты, тыс. руб.

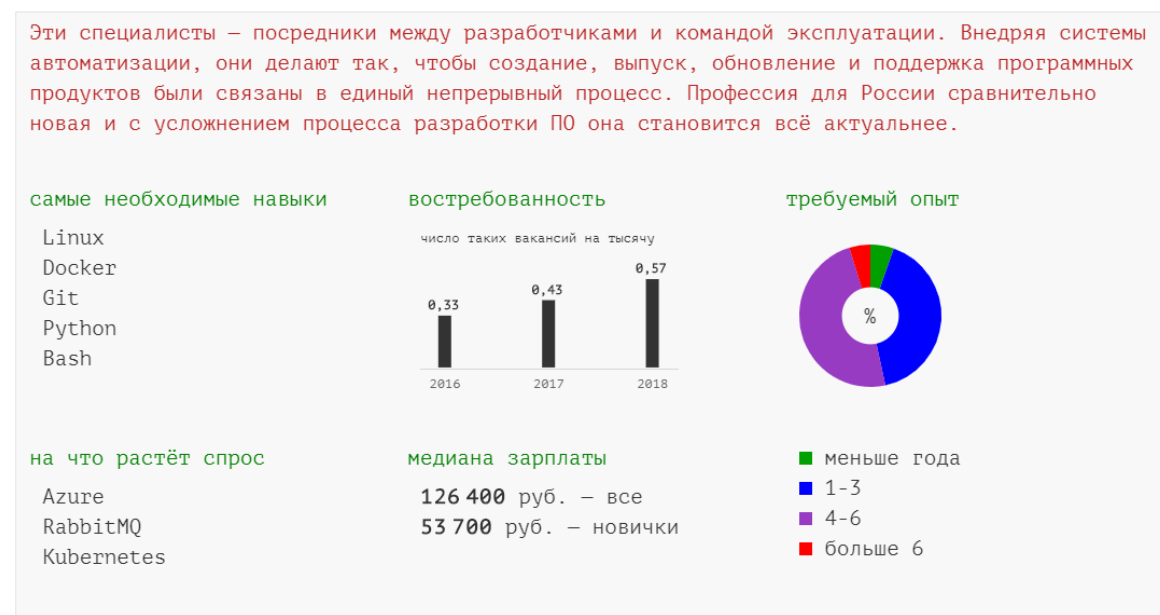


ПО ДАННЫМ ЯНДЕКС.ПРАКТИКУМА И АНАЛИТИЧЕСКОЙ СЛУЖБЫ HEADHUNTER, 2018

Рисунок 1.2.3 — Зарплаты и требования к опыту работы в разных специальностях

Можно заметить, что DevOps-инженеры входят в 10 самых оплачиваемых специалистов отрасли. Зарплаты колеблются от 54 до 224 тысяч рублей в месяц в зависимости от опыта работы.

На рисунке 1.2.4 представлен портрет представителя DevOps.



ПО ДАННЫМ ЯНДЕКС.ПРАКТИКУМА И АНАЛИТИЧЕСКОЙ СЛУЖБЫ HEADHUNTER, 2016–2018

Рисунок 1.2.4 — Портрет представителя DevOps

По данным на сентябрь 2021 года, зарплаты IT-специалистов выросли на 20-30% по сравнению с летом [16]. К концу года дефицит специалистов усугубился, это отмечают руководителей компаний. Количество вакансий только увеличивается – на 85% по сравнению с осенью 2020 года: в конце ноября 2021 года компании на hh.ru предлагали около 140 тыс. рабочих мест в сфере IT, интернета и телекома [17].

Можно заключить, что профессия становится все актуальнее для российских IT компаний. Рынок автоматизации проектов растет. Требуются все новые и новые специалисты для покрытия интересов компаний.

Поэтому при самых минимальных расчетах становится понятно, что автоматизация может экономить средства компании.

1.6.3 Временные и материальные затраты на подготовку специалиста начального уровня

Современный бизнес ожидает от IT все больше. Требования к производству программного обеспечения становятся жестче, планка качества – выше, времени и ресурсов – меньше. Отрасль требует новых специалистов, поэтому многие компании берут все новых и новых разработчиков начального уровня, чтобы в будущем покрыть потребность в качественном персонале. Чтобы получить стоящего разработчика компании необходимо от 3 до 6 месяцев на его обучение. В долгосрочной перспективе компания возымеет результат, но может нести издержки в краткосрочной перспективе.

1.6.4 Время развертывания проекта

Время на формирование `dockerfile` можно значительно сократить, если автоматизировать процесс развертывания проекта.

При выполнении задания человеком весь цикл развертывания одного контейнера занимает от 30 мин до нескольких дней. Время развертывания проекта зависит от наличия сложных зависимостей и от того, надо ли развернуть базу данных внутри контейнера или подключиться к имеющейся. Чем запутаннее зависимости между файлами, тем их сложнее описать для эффективной работы

При автоматизации процесса цикл создания контейнера будет занимать соответственно гораздо меньшее время.

1.6.5 Экономия при автоматическом развертывании проекта

При автоматической развертке проекта компания может сэкономить на DevOps-инженерах. Сократится время развертывания контейнеров за счет автоматизации процесса.

Поскольку процессы преобразуются путем автоматизации отдельных видов деятельности, люди будут выполнять действия, которые поддерживают работу машин. То есть дефицит рабочей силы в скором времени будет удовлетворен не подбором персонала, а подбором ИТ.

1.7 Выводы из аналитического раздела

При данном разделе были рассмотрены и проанализированы существующие средства виртуализации, проанализирована эффективность использования автоматизированных и человеческих ресурсов. Из рассмотренных средств виртуализации наиболее подходящим для виртуализации является Docker, за счет безопасности контейнеров, отсутствия единой точки отказа, упрощенного администрирования и поддержки, отсутствия зависимости от платформы.

2 Конструкторский раздел

2.1 Требования к методу автоматической генерации docker-файла

Ниже приведен список требований к методу генерации docker-файла:

- метод должен определять в директории проекта фреймворки Django и Flask;
- метод должен работать с библиотекой для создания пользовательских интерфейсов - React и фреймворком Django;
- метод должен считывать конфигурацию подключения баз данных postgresql и sqlite3 в фреймворках Django и Flask по определенному шаблону;
- метод должен создавать образ контейнера проекта;
- метод должен запускать контейнер проекта.

2.2 IDEF0

Для генерации docker-файла и запуска контейнера необходимо директория проекта и точка монтирования каталога баз данных, если база данных не расположена в директории самого проекта.

На рисунках 2.2.1 – 2.2.2 представлена IDEF0 диаграмма.

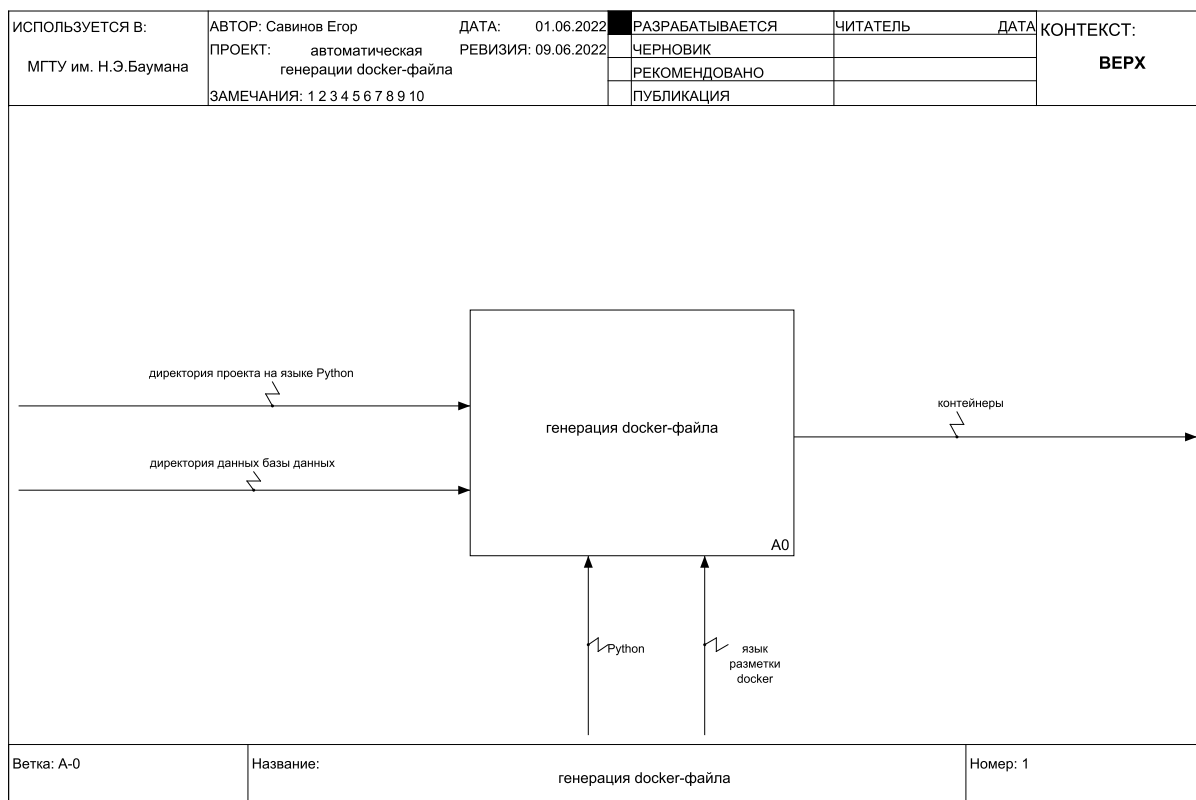


Рисунок 2.2.1 — IDEF0 диаграмма ветки A-0

Для решения поставленной задачи необходимо:

- обнаружить поддерживаемый фреймворк Django или Flask;
- найти поддерживаемые базы данных postgresql, sqlite3;
- найти версии зависимостей, которые используются в проекте;
- сформировать docker-файл для проекта;
- создать сеть для взаимодействия контейнеров;
- запустить контейнер проекта, базы данных;
- перенести данные баз данных в контейнер баз данных.

2.3 Описание алгоритма автоматической генерации docker-файла

2.3.1 Поиск фреймворка Flask

Для поиска фреймворка Flask необходимо:

- перейти на директорию проекта;
- прочитать построчно файлы директории проекта;

- найти зависимость Flask;
- если зависимость Flask найдена, то вернуть True, иначе перейти на следующий файл, а если все файлы в директории просмотрены, то перейти в следующую поддиректорию проекта.

Схема алгоритма изображена на рисунке 2.3.1.

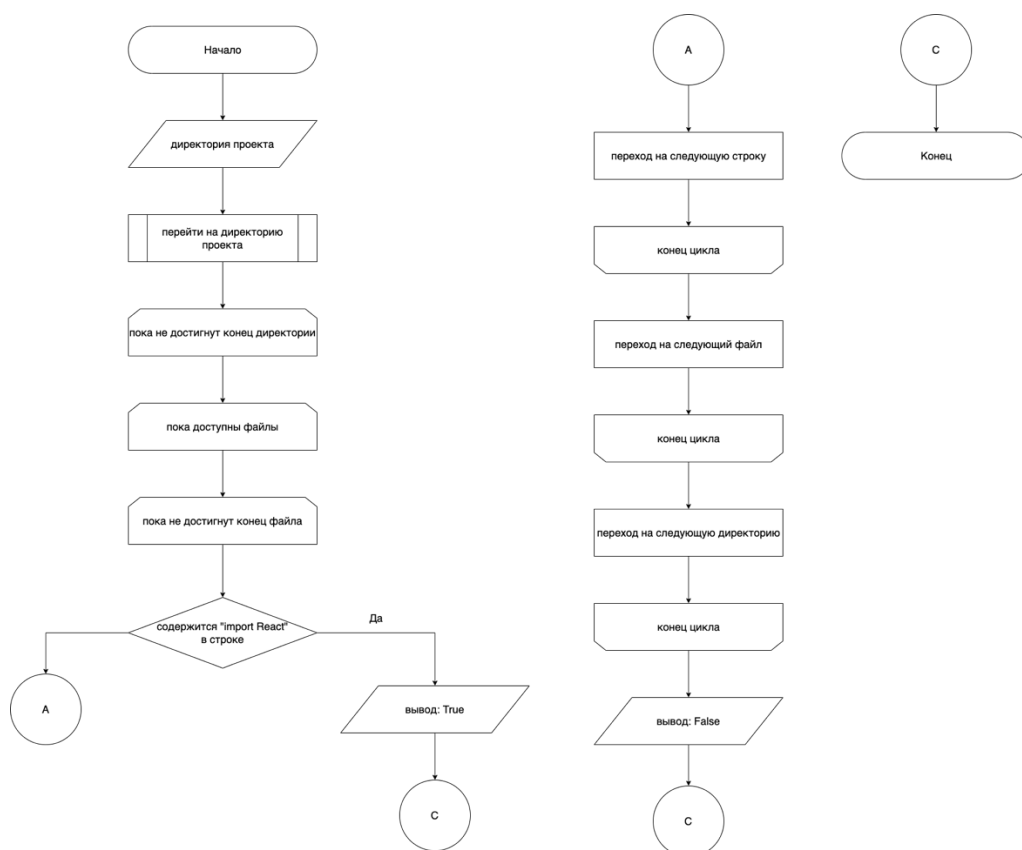


Рисунок 2.3.1 — Схема алгоритма поиска фреймворка Flask

2.3.2 Поиск фреймворка Django

Для поиска фреймворка Django необходимо:

- перейти на директорию проекта;
- прочитать файлы директории проекта;
- найти базовый файл проекта Django `manage.py`.

Содержимое файла `manage.py` используется как инструмент командной строки для управления проектами. Файл создается при инициализации проекта и не может быть удален.

Схема алгоритма изображена на рисунке 2.3.2.

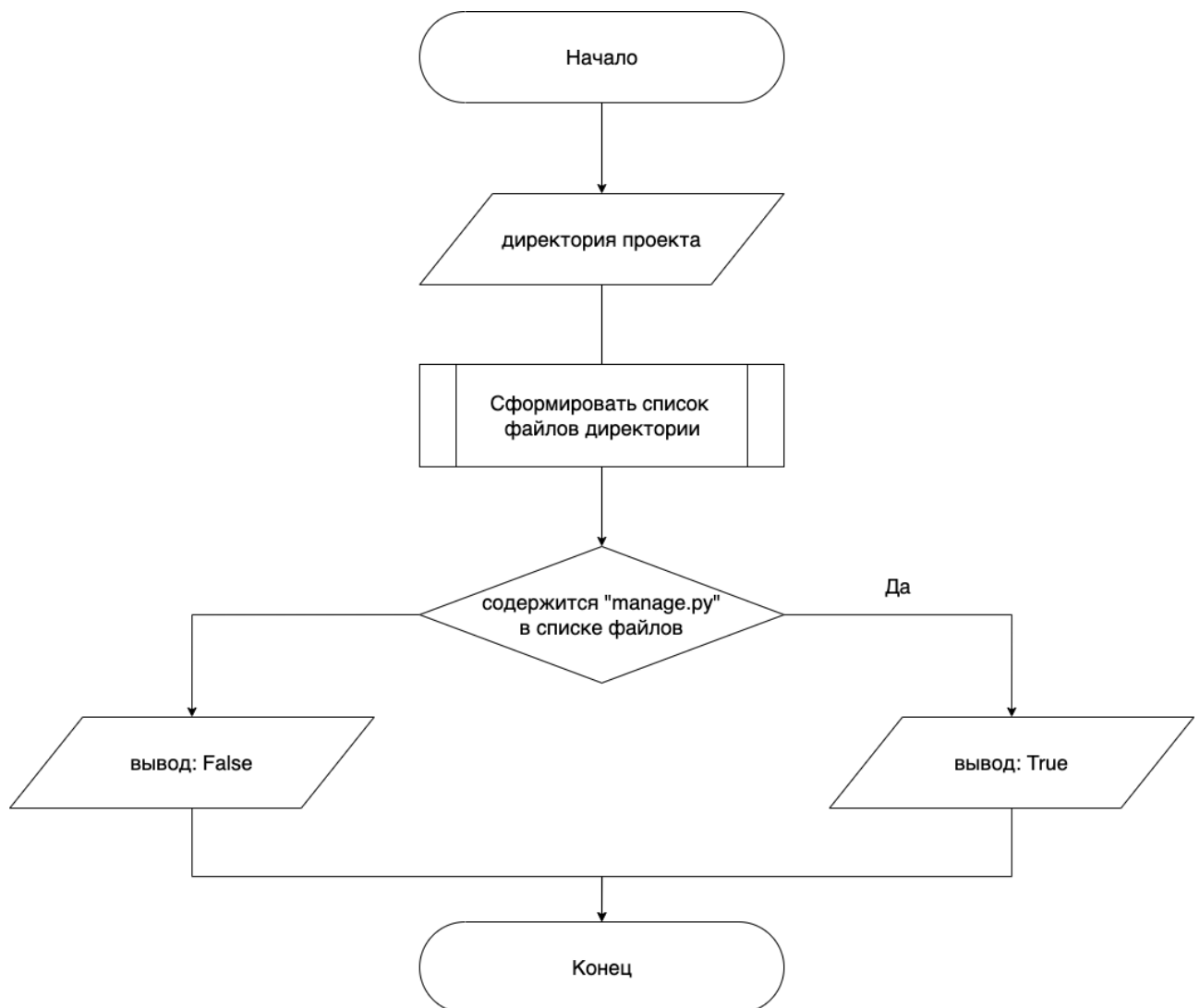


Рисунок 2.3.2 — Схема алгоритма поиска фреймворка Django

2.3.3 Поиск конфигурации подключения баз данных фреймворка Django

При поиске конфигурации подключения баз данных читается файл настроек проекта построчно и с помощью регулярного выражения ищутся соответствия настроек подключения баз данных, если считана вся конфигурация подключения базы данных, то словарь с данными добавляется в общий список хранения баз данных и ищется следующая. Схема алгоритма изображена на рисунке 2.3.3.

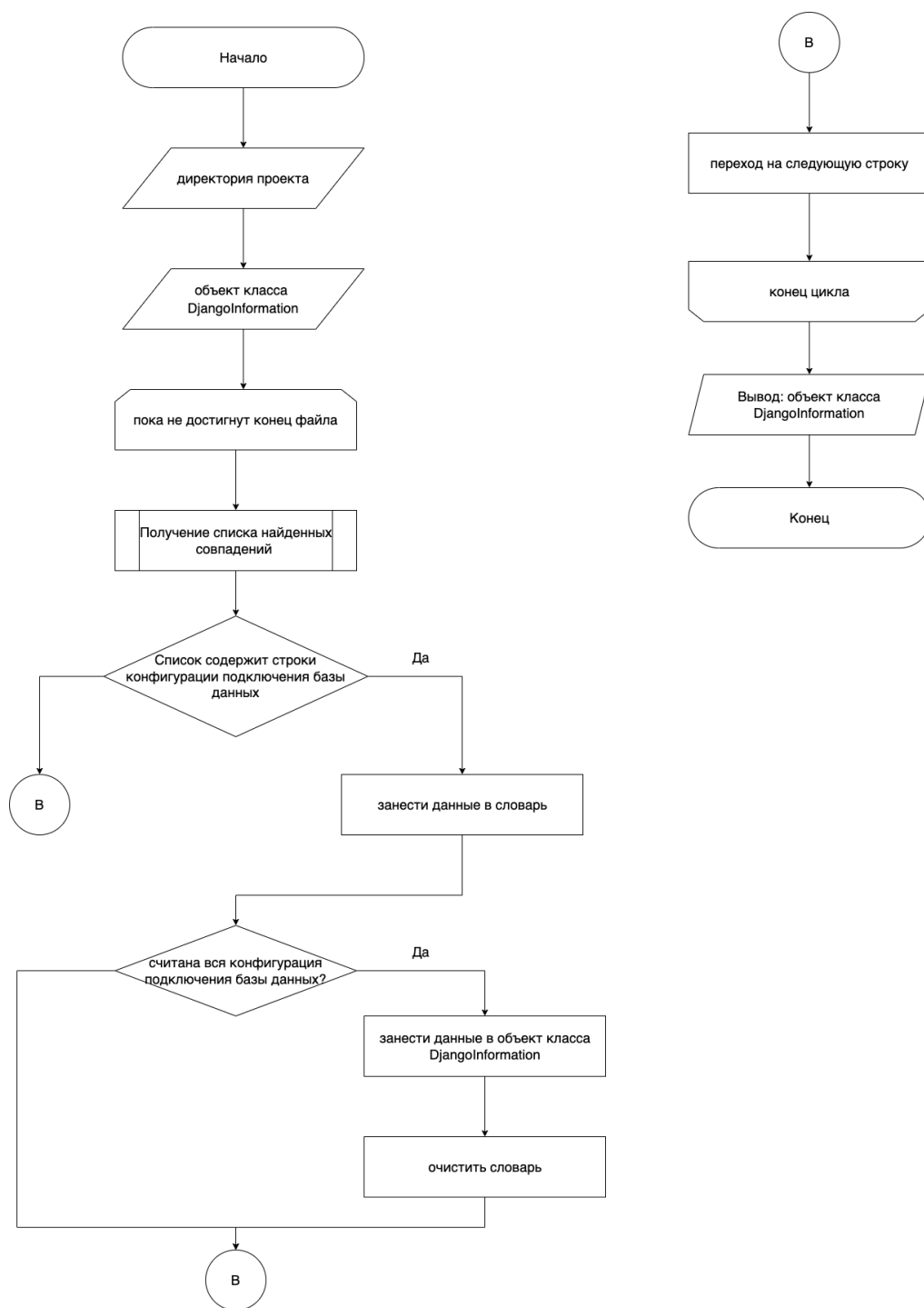


Рисунок 2.3.3 — Схема алгоритма поиска конфигурации подключения баз данных фреймворка Django

2.3.4 Поиск конфигурации подключения баз данных фреймворка Flask

В отличие от поиска баз данных в приложении, построенном на фреймворке Django, сначала необходимо найти файл, где хранится конфигурация

подключения к базам данным. Далее аналогично читается файл настроек проекта построчно и с помощью регулярного выражения ищутся соответствия настроек подключения баз данных, если считана вся конфигурация подключения базы данных, то словарь с данными добавляется в общий список хранения баз данных и ищется следующая. Схема алгоритма изображена на рисунке 2.3.4.

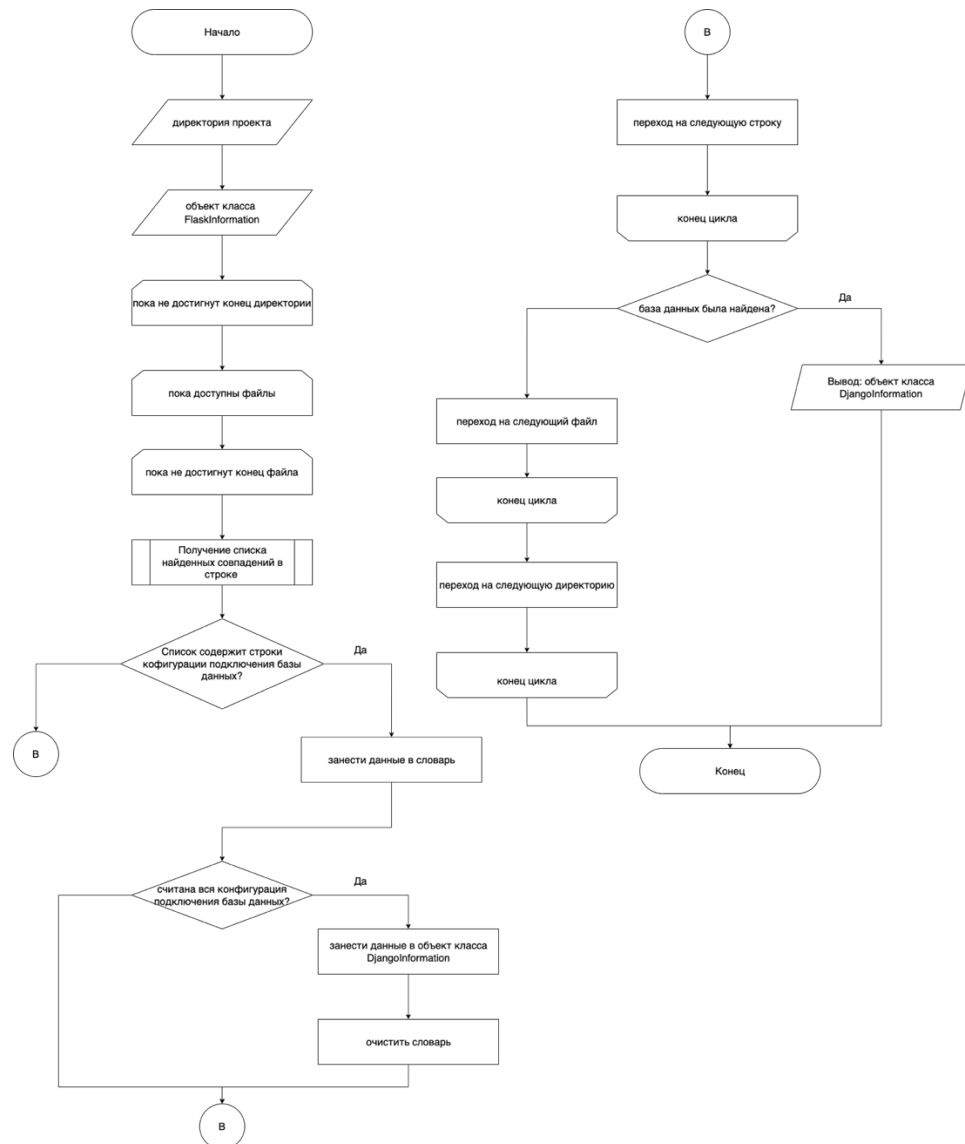


Рисунок 2.3.4 — Схема алгоритма поиска конфигурации подключения баз данных фреймворка Flask

2.3.5 Поиск React

Для поиска библиотеки для создания пользовательских интерфейсов React необходимо:

- перейти на директорию проекта;
- прочитать построчно файлы директории проекта;
- найти зависимость React;
- если зависимость React найдена, то вернуть True, иначе перейти на следующий файл, а если все файлы в директории просмотрены, то перейти в следующую поддиректорию проекта.

Схема алгоритма изображена на рисунке 2.3.5.

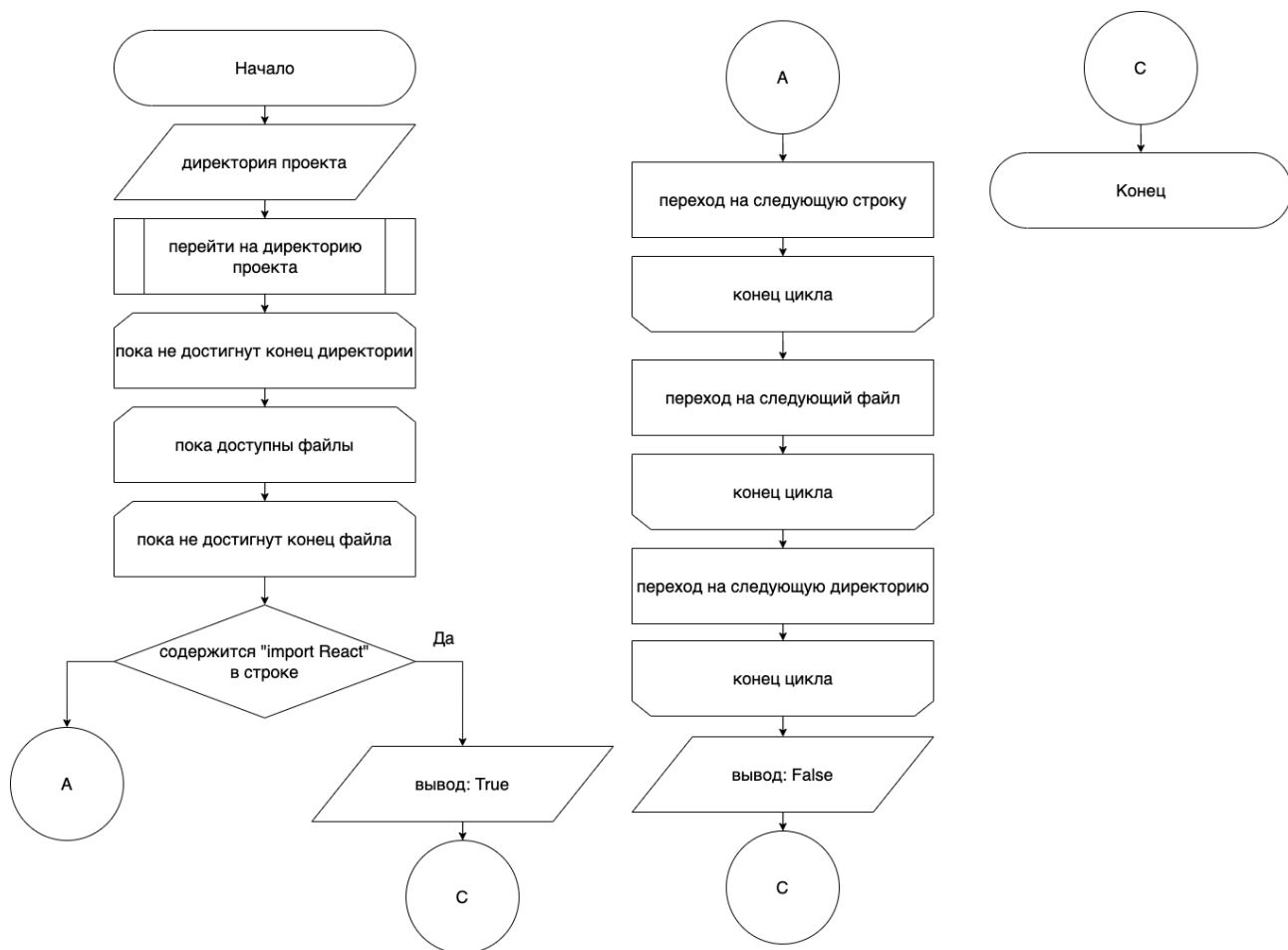


Рисунок 2.3.5 — Схема алгоритма поиска React

2.3.6 Структура сети

На рисунке 2.3.6 наглядно как образы взаимодействуют между собой. Мостовая сеть назначает контейнерам внутри нее IP-адреса в диапазоне 172.0.X.X и подключает к одной мостовой сети, обеспечивая при этом изоляцию от контейнеров, которые не подключены к этой мостовой сети. Контейнеры,

подключенные к одной и той же определяемой пользователем мостовой сети, эффективно открывают друг другу все порты.

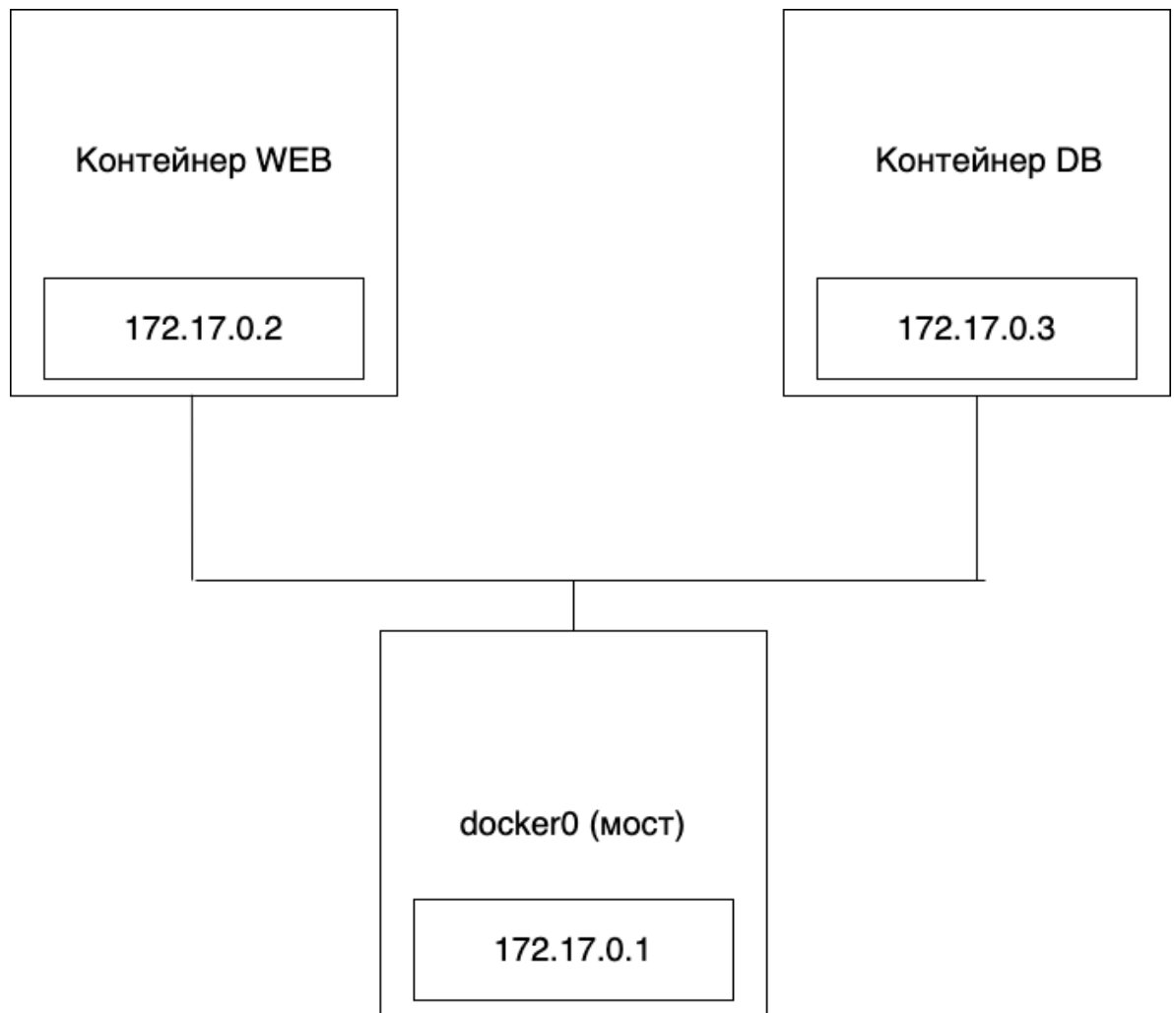


Рисунок 2.3.6 — Структура сети

2.4 Допустимые конфигурации подключения баз данных

2.4.1 Django

По умолчанию Django настроен для использования SQLite в качестве серверной части. Из-за того, что база данных хранится локально, не нужно создавать дополнительный контейнер для взаимодействия между приложением и базой данных. На рисунке 2.4.1 представлена конфигурация подключения SQLite.


```

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'mydatabase'
    }
}

```

Рисунок 2.4.1 — конфигурация подключения SQLite

При подключении к серверу локальному или удаленному Postgres конфигурация подключения изменяется, необходимо указать пользователя, пароль, название базы данных, хост и порт. На рисунке 2.4.2 представлена конфигурация подключения PostgreSQL.

```

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'test123',
        'USER': 'postgres',
        'PASSWORD': '2468',
        'HOST': 'db0',
        'PORT': 5432
    }
}

```

Рисунок 2.4.2 — конфигурация подключения PostgreSQL

2.4.2 Flask

Для подключения баз данных в фреймворке Flask используется программная библиотека на языке Python для работы с реляционными СУБД с применением технологии ORM Flask-SQLAlchemy. Конфигурация для подключения имеет две разновидности. Для PostgreSQL указывается имя пользователя, пароль, название базы данных, порт и хост, а в SQLite указывается название базы

данных. На рисунке 2.4.3 представлена конфигурация подключения для SQLite и PostgreSQL.

```
app.config['SQLALCHEMY_DATABASE_URI'] = "postgres://postgres:2468@db0:5432/testtest"
app.config["SQLALCHEMY_BINDS"] = {
    "users": "sqlite:///database.db"
}
```

Рисунок 2.4.3 — конфигурация подключения PostgreSQL и SQLite

2.5 Требования к выходным данным

Пользователю будет предоставлено работающее приложение, состоящее из одного или двух контейнеров в Docker. Количество контейнеров будет зависеть от способа подключения баз данных. Если база данных расположена в директории проекта, то создание дополнительного контейнера не потребуется, а если база данных расположена на сервере, то создается сеть для взаимодействия контейнеров между собой и дополнительный контейнер для базы данных.

2.6 Выводы из конструкторского раздела

В конструкторском разделе были рассмотрены требования к методу, представлены IDEF0 диаграммы, приведены схемы алгоритмов поиска фреймворков Django и React и библиотеки React, конфигураций подключения баз данных PostgreSQL и SQLite3.

В программе должен быть реализован графический интерфейс, позволяющий пользователю взаимодействовать с ПО, в интерфейсе должны быть поля для ввода директории проекта, ввода директории данных базы данных.

3 Технологический раздел

3.1 Выбор языка программирования

В качестве языка программирования был выбран Python [18] так как:

- достаточный опыт в программировании на языке Python, позволяет сократить время на написание программы;
- Python – объектно-ориентированный язык программирования. В связи с этим позволяет использовать классы, наследование;
- Python – кроссплатформенный, что позволяет сократить время разработки программы;
- Не нужно беспокоиться о работе с памятью.

3.2 Выбор языка программирования

В качестве среды разработки было выбрано программное обеспечение от JetBrains – PyCharm [19] так как:

- достаточный опыт программирования в данной среде разработки;
- автодополнение кода и анализ кода;
- подсветка ошибок;
- автоматический рефакторинг, позволяет эффективно редактировать код;
- удобная навигация по проекту;
- наличие полнофункционального встроенного терминала и инструментов для работы с базами данных;
- интеграция с Docker.

3.3 Реализация алгоритмов

Для хранения информации о подключении к базам данных был создан базовый класс. В листинге 3.3.1 представлены класс для хранения информации о базах данных.

Листинг 3.3.1 — класс для хранения информации о базах данных

```
1. class DBInformation(object):
2.     def __init__(self, dbs=None):
3.         if dbs is None:
4.             dbs = []
5.             self.dbs = dbs
6.
7.     def display(self):
8.         print(self.dbs)
9.
10.    def get_dbs(self):
11.        return self.dbs
12.
13.    def update_dbs(self, db):
14.        self.dbs.append(db)
```

Для Django приложения необходимо знать информацию о порте подключения, директории проекта. Класс наследуется от базового класса для хранения информации о базах данных.

В листинге 3.3.2 представлены класс для хранения информации о Django приложении.

Листинг 3.3.2 — класс для хранения информации о Django приложении

```
1. class DjangoInformation(DBInformation):
2.     def __init__(self, workdir=None, port=8000):
3.         super().__init__()
4.         self.workdir = workdir
```

```

5.         self.port = port
6.
7.     def update_dbs(self, db):
8.         super().update_dbs(db)
9.
10.    def get_workdir(self):
11.        return self.workdir
12.
13.    def set_workdir(self, workdir):
14.        self.workdir = workdir
15.
16.    def get_port(self):
17.        return self.port
18.
19.    def set_port(self, port):
20.        self.workdir = port

```

Для создания docker-файла Django приложения необходимы базовый образ будущего контейнера и параметры переменных среды. В листинге 3.3.2 представлены класс для docker-файла Django приложения.

Листинг 3.3.3 — класс для docker-файла Django приложения

```

1.     class Docker_Information_django(object):
2.         def __init__(self, basicimage='python:3', pythonun-
3. buffered='1', pythondontwritebytecode='1'):
4.             self.basicimage = basicimage
5.             self.pythonunbuffered = pythonunbuffered
6.             self.pythondontwritebytecode = python-
7. dontwritebytecode
8.
9.         def get_basicimage(self):
10.            return self.basicimage
11.
12.        def get_pythonunbuffered(self):

```

```
13.         return self.pythonunbuffered
14.
15.         def get_pythondontwritebytecode(self):
16.             return self.pythondontwritebytecode
```

Для Flask приложения также необходимо знать информацию о порте подключения, директории проекта. Класс наследуется от базового класса для хранения информации о базах данных. В листинге 3.3.4 представлены класс для хранения информации о Flask приложении.

Листинг 3.3.4 — класс для хранения информации о Flask приложении

```
1.         class FlaskInformation(DBInformation):
2.             def __init__(self, workdir=None, port=5000):
3.                 super().__init__()
4.                 self.workdir = workdir
5.                 self.port = port
6.
7.             def update_dbs(self, db):
8.                 super().update_dbs(db)
9.
10.            def get_workdir(self):
11.                return self.workdir
12.
13.            def set_workdir(self, workdir):
14.                self.workdir = workdir
15.
16.            def get_port(self):
17.                return self.port
18.
19.            def set_port(self, port):
20.                self.workdir = port
```

Для создания docker-файла Flask приложения необходимы базовый образ будущего контейнера. В листинге 3.3.5 представлены класс для docker-файла Flask приложения.

Листинг 3.3.5 — класс для docker-файла Flask приложения

```
1.         class Docker_Information_flask(object):
2.             def __init__(self, basicimage='python:3'):
3.                 self.basicimage = basicimage
4.
5.             def get_basicimage(self):
6.                 return self.basicimage
```

Для React необходимо знать информацию о порте подключения, версии node js, о директории хранения package_json, рабочей директории. В листинге 3.3.6 представлены класс для хранения информации о React приложении.

Листинг 3.3.6 — класс для хранения информации о React приложении

```
1.         class ReactInformation(object):
2.             def __init__(self, node_version=None, pack-
3. age_json=None, workdir=None, port=3000):
4.                 self.node_version = node_version
5.                 self.package_json = package_json
6.                 self.workdir = workdir
7.                 self.port = port
8.
9.             def get_node_version(self):
10.                 return self.node_version
11.
12.             def set_node_version(self, node_version):
13.                 self.node_version = node_version
14.
15.             def get_package_json(self):
16.                 return self.package_json
```

```

17.
18.         def set_package_json(self, package_json):
19.             self.package_json = package_json
20.
21.         def get_workdir(self):
22.             return self.workdir
23.
24.         def set_workdir(self, workdir):
25.             self.workdir = workdir
26.
27.         def get_port(self):
28.             return self.port
29.
30.         def set_port(self, port):
31.             self.workdir = port

```

Для создания docker-файла React необходимы базовый образ будущего контейнера. В листинге 3.3.7 представлены класс для docker-файла React.

Листинг 3.3.7 — класс для docker-файла React

```

1.         class Docker_Information_react(object):
2.             def __init__(self, basicimage='node:'):
3.                 self.basicimage = basicimage
4.
5.             def get_basicimage(self):
6.                 return self.basicimage

```

В листингах 3.3.8 - 3.3.9 представлены реализации модулей поиска фреймворков.

На вход подается директория проекта, в ней в папках и подпапках ищется файл по умолчанию для Django. Для Flask ищется в файлах подключённый модуль Flask.

Листинг 3.3.8 — Поиск фреймворка Django

```
1.         def detector_django_project(dir_project):
2.             try:
3.                 file_list = os.listdir(dir_project)
4.             except FileNotFoundError:
5.                 print("Нет такой директории: " + dir_project)
6.                 exit(0)
7.             else:
8.                 if "manage.py" in file_list:
9.                     return True
10.                else:
11.                    return False
```

Листинг 3.3.9 — Поиск фреймворка Flask

```
1.         def detector_flask_project(dir_project):
2.             try:
3.                 os.listdir(dir_project)
4.             except FileNotFoundError:
5.                 print("Нет такой директории: " + dir_project)
6.                 exit(0)
7.             else:
8.                 for root, dirs, files in os.walk(dir_project):
9.                     for file in files:
10.                        try:
11.                            for line in open(root + '/' + file):
12.                                if 'import Flask' in line:
13.                                    return True
14.                        except:
15.                            continue
16.                    return False
```

На листинге 3.3.10 – 3.3.11 представлена реализация поиска конфигурации для баз данных.

Каждая из баз данных помещается в структуру данных Python – словарь, далее в массив объекта баз данных.

В файле основных настроек Django по установленному шаблону ищутся такие настройки баз данных как:

- ENGINE – адаптер для базы данных;
- NAME – название базы данных;
- USER – пользователь базы данных;
- PASSWORD – пароль для доступа к базе данных.

Если база данных расположена в директории приложения, то для доступа необходимы только ENGINE и NAME.

Для поиска конфигурации баз данных используется регулярное выражение: `'(\w+)':s*([0-9]+|'(.+)')`.

Где регулярное выражение расшифровывается как:

- `.+` – любой символ один или более раз, кроме символа новой строки;
- `[0-9]+` – любая цифра от 0 до 9 один или более раз;
- `\s*` – символ пробел один или более раз;
- `\w+` – любая цифра или буква один или более раз;
- `()` – группирует выражение и возвращает найденный текст.

Листинг 3.3.10 — Поиск конфигурации баз данных для фреймворка Django

```
1.     def detect_databases_django(dir_project, django_str):
2.         settingspy = ut.find('settings.py', dir_project)
3.
4.         if settingspy is None:
5.             print("Не найден файл: " + dir_project + '/set-
6. tings.py')
7.             exit(1)
8.
9.         db = {}
10.        flag_add = False
11.        flag_sqlite3 = False
12.        pattern = re.compile("'(\w+)':s*([0-9]+|'(.+)')")
13.        with open(settingspy) as f:
```

```

14.         for line in f.readlines():
15.             match = re.findall(pattern, line)
16.             if match and 'django.contrib.auth.pass-
17. word_validation.' not in match[0][1] and \
18.                 'BACKEND' not in match[0][0]:
19.                 if 'django.db.backends.post-
20. gresql_psycopg2' in match[0][1]:
21.                     db['ENGINE'] = 'postgres'
22.                     continue
23.                 if 'django.db.backends.sqlite3' in
24. match[0][1]:
25.                     db['ENGINE'] = 'sqlite3'
26.                     django_str.update_dbs(db)
27.                     db = {}
28.                     flag_add = True
29.                     flag_sqlite3 = True
30.                     continue
31.                 if 'django.db.backends.mysql' in
32. match[0][1]:
33.                     db['ENGINE'] = 'mysql'
34.                     continue
35.
36.                 if flag_sqlite3:
37.                     continue
38.
39.                 db[match[0][0]] = match[0][1].re-
40. place("'", "")
41.
42.                 if match[0][0] == 'PORT':
43.                     django_str.update_dbs(db)
44.                     db = {}
45.                     flag_add = True
46.
47.             if flag_add is False:
48.                 print("База данных не найдена")

```

```
49.             exit(1)
50.
51.             return django_str
```

В файле основных настроек Flask по установленному шаблону ищутся такие настройки баз данных как:

- ENGINE – адаптер для базы данных;
- NAME – название базы данных;
- USER – пользователь базы данных;
- PASSWORD – пароль для доступа к базе данных.

В отличии от Django конфигурация настроек подключения к базам данных представлена в форме строки.

Если база данных расположена в директории приложения, то для доступа необходимы ENGINE, NAME и иной шаблон.

Для поиска конфигурации баз данных используются два регулярные выражения: `(\w+)://(.+):(.)+@(.+):(.)+/(.)\"` для PostgreSQL и `("(.+)=\s'(.+)://(.)'` для SQLite.

Где регулярные выражения расшифровывается как:

- `.+` – любой символ один или более раз, кроме символа новой строки;
- `[0-9]+` - любая цифра от 0 до 9 один или более раз;
- `\s*` - символ пробел один или более раз;
- `\w+` - любая цифра или буква один или более раз;
- `()` – группирует выражение и возвращает найденный текст;

Листинг 3.3.11 — Поиск конфигурации баз данных для фреймворка Flask

```
1.     def detect_databases_flask(dir_project, flask_str):
2.         path = os.path.exists(dir_project)
3.
4.         if path is False:
5.             # print("Не найдена директория: " + dir_project)
6.             exit(1)
7.
8.         db = {}
9.         flag_add = False
10.        pattern = re.com-
11.        pile("(\\w+):/(.+):(.)+@(.+):(.)+/(.+)"")
12.        for root, dirs, files in os.walk(dir_project):
13.            for file in files:
14.                try:
15.                    with open(root + '/' + file) as f:
16.                        for line in f.readlines():
17.                            if flag_add:
18.                                line = line.split(" ")
19.                                line = line[len(line) - 1]
20.                                match = re.findall(pattern,
21. line)
22.                                if match:
23.                                    db['ENGINE'] = match[0][0]
24.                                    db['USER'] = match[0][1]
25.                                    db['PASSWORD'] = match[0][2]
26.                                    db['HOST'] = match[0][3]
27.                                    db['PORT'] = match[0][4]
28.                                    db['NAME'] = match[0][5]
29.                                    flag_add = True
30.                                    flask_str.update_dbs(db)
31.                                    db = {}
32.                                except UnicodeDecodeError:
33.                                    continue
34.                                if flag_add is True:
```

```

35.             break
36.
37.         pattern = re.compile("(.)=\s'(.+)'://(.)'")
38.         for root, dirs, files in os.walk(dir_project):
39.             for file in files:
40.                 try:
41.                     with open(root + '/' + file) as f:
42.                         for line in f.readlines():
43.                             match = re.findall(pattern,
44. line)
45.                             if match:
46.                                 db['ENGINE'] = match[0][1]
47.                                 db['NAME'] = match[0][2]
48.                                 flag_add = True
49.                                 flask_str.update_dbs(db)
50.                                 db = {}
51.                             except UnicodeDecodeError:
52.                                 continue
53.                             if flag_add is True:
54.                                 break
55.
56.         if flag_add is False:
57.             #print("База данных не найдена")
58.             exit(1)
59.
60.         return flask_str

```

На листинге 3.3.12 – 3.3.13 представлена реализация функции формирования docker-файла для Django и Flask.

Для Django задается базовый образ, переменные окружения, рабочая директория, файл с зависимостями, которые нужно установить, папка содержимое которой нужно скопировать в контейнер, порт и параметры запуска контейнера. Если присутствует в проекте React, то параметры запуска используются отличные от Django.

Листинг 3.3.12 — формирование docker-файла для фреймворка Django

```
1.     def createdockerfile(docker_str, django_str, dir_pro-
2.     ject, react_flag):
3.         os.chdir(dir_project)
4.         f = open('dockerfile', 'w')
5.         f.write('FROM ' + docker_str.get_basicimage() +
6.                 '\nENV      PYTHONDONTWRITEBYTECODE='      +
7.         docker_str.get_pythonunbuffered() +
8.                 '\nENV      PYTHONUNBUFFERED='              +
9.         docker_str.get_pythondontwritebytecode() +
10.                '\nWORKDIR /' + django_str.get_workdir() +
11.                '\nCOPY      requirements.txt                /'      +
12.         django_str.get_workdir() + '/' +
13.                '\nRUN pip3 install -r requirements.txt' +
14.                '\nCOPY . /' + django_str.get_workdir() + '/'
15. +
16.                '\nEXPOSE ' + str(django_str.get_port()))
17.
18.         if react_flag is False:
19.             f.write('\nCMD      ["python",      "manage.py",
20.             "runserver", "0.0.0.0:" + str(django_str.get_port()) + "']')
21.
22.         f.close()
```

Для Flask задается базовый образ, рабочая директория, файл с зависимостями, которые нужно установить, папка содержимое которой нужно скопировать в контейнер.

Листинг 3.3.13 — формирование docker-файла для фреймворка Flask

```

1.         def createdockerfile(docker_str, flask_str, dir_pro-
2.     ject):
3.         os.chdir(dir_project)
4.         f = open('dockerfile', 'w')
5.         f.write('FROM ' + docker_str.get_basicimage() +
6.             '\nWORKDIR /' + flask_str.get_workdir() +
7.             '\nCOPY      requirements.txt      /'      +
8.     flask_str.get_workdir() + '/' +
9.             '\nRUN pip3 install -r requirements.txt' +
10.            '\nCOPY . /' + flask_str.get_workdir() + '/'
11. +
12.            '\nEXPOSE ' + str(flask_str.get_port()) +
13.            '\nCMD ["python3", "app.py"]')
14.
15.         f.close()

```

Для React задается базовый образ, рабочая директория, файл с зависимостями, которые нужно установить, папка с фронтендом которую нужно скопировать в контейнер, порт и параметры запуска контейнера.

Листинг 3.3.14 — формирование docker-файла для React

```

1.         def createdockerfile(docker_str, react_str, dir_pro-
2.     ject):
3.         os.chdir(dir_project)
4.         f = open('dockerfile', 'w+')
5.         f.write('\n\nFROM ' + docker_str.get_basicimage() +
6.     react_str.get_node_version() +
7.             '\nRUN mkdir /' + react_str.get_workdir() +
8.             '\nWORKDIR /' + react_str.get_workdir() +
9.             '\nCOPY    /'    + react_str.get_workdir() +
10.     '/package.json /' + react_str.get_workdir() +
11.             '\nCOPY /' + react_str.get_workdir() + ' /'
12. + react_str.get_workdir() +
13.             '\nRUN npm install' +

```



```

14.         '\nEXPOSE ' + str(react_str.get_port()) +
15.         '\nCMD ["npm", "start"]')
16.
17.         f.close()

```

На листинге 3.3.15 представлена реализация функции для создания контейнера для базы данных PostgreSQL.

Перед тем как запустить контейнер, генерируется резервная копия базы данных приложения. Затем создается контейнер с параметрами, которые были определены в приложении. Спустя некоторое время после запуска контейнера резервная копия копируется в директорию данных базы данных и восстанавливает базу данных в контейнере.

Для создания и восстановления резервных копий PostgreSQL используется программа `pg_dump` и `pg_restore` соответственно.

Листинг 3.3.15 — формирование контейнера для базы данных PostgreSQL

```

1.     def launch_db_container_postgres(db, dir_db, i):
2.         os.chdir(dir_db)
3.         os.system("pg_dump -Fc " + db['NAME'] + " > db.dump")
4.
5.
6.         os.system("docker run -d --name db" + str(i) + " -p
7. 543" + str(i) + ":5432"
8.                 " -e POSTGRES_USER=" + db['USER'] + "
9. -e "
10.                 "POSTGRES_PASSWORD=" + db['PASSWORD']
11. + " -e "
12.                 "POSTGRES_DB=" + db['NAME'] +
13.                 " -v " + dir_db + " --network=mynetwork
14. postgres:" +
15.                 os.popen('postgres --ver-
16. sion').read().split(" ")[2].split("\n")[0])

```

```

17.
18.         time.sleep(10)
19.         os.system("docker cp db.dump db" + str(i) + ":" +
20. dir_db)
21.         os.system("docker exec db" + str(i) + " pg_restore -
22. U " + db['USER'] +
23.         " -d " + db['NAME'] + " " + dir_db +
24. "/db.dump")

```

3.4 Пользовательский интерфейс

Интерфейс программы реализован с помощью фреймворка PySimpleGUI.

На рисунке 3.4.1 показан графический интерфейс программы. Пользователю предлагается указать директорию проекта и директорию хранения данных базы данных. Если не нужно создавать контейнер необходимо нажать на соответствующую кнопку.

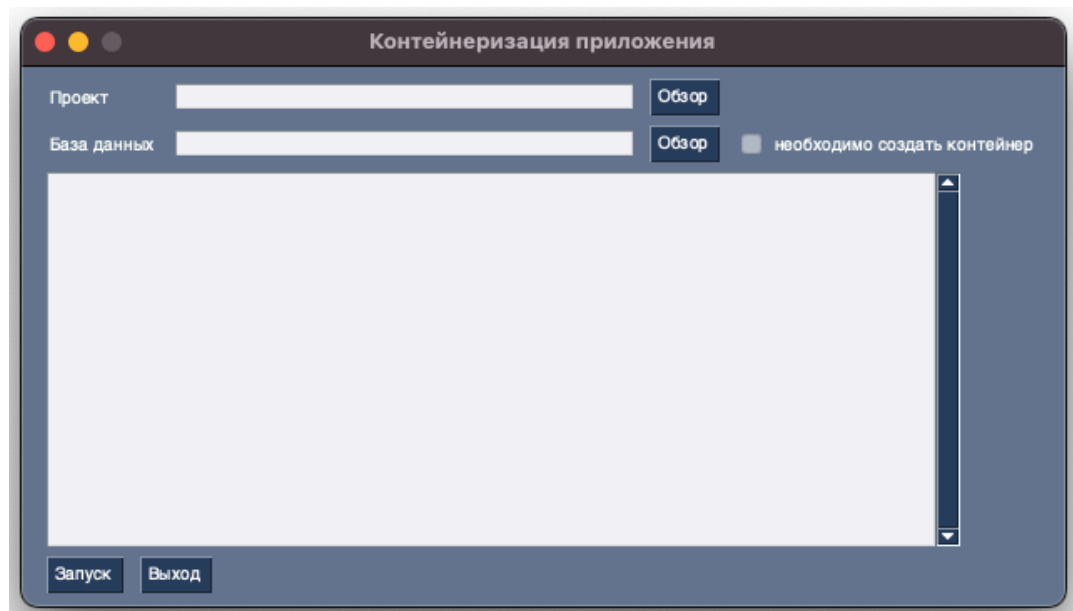


Рисунок 3.4.1 — Окно программы

Во время работы программы в окно вывода выводится название найденного фреймворк и баз данных и порт контейнера приложения (рисунок 3.4.2).

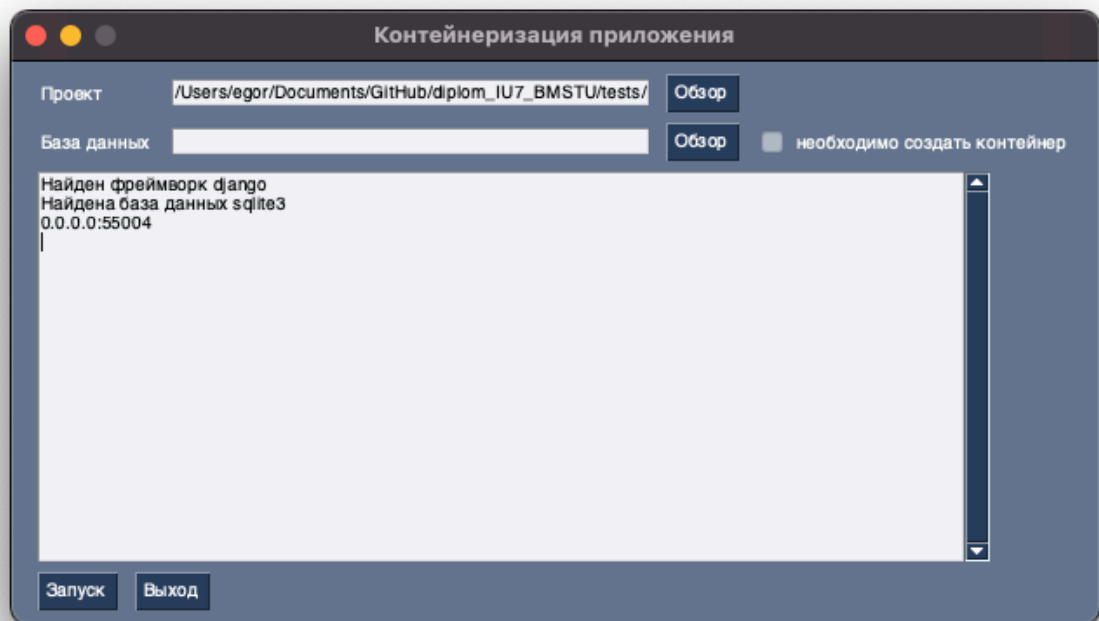


Рисунок 3.4.2 — Окно программы после запуска

3.5 Функциональное тестирование

Функциональное тестирование проводится методом «черного ящика». Функциональность исследуется без знаний о внутреннем устройстве, без учета деталей реализации программного продукта. В листинге 3.7.1 представлен пример тестирования с помощью фреймворка unittest функции поиска фреймворка Django и поиска конфигураций подключения баз данных фреймворка Django.

Листинг 3.7.1 — функциональное тестирование функций поиска

```
1. class TestDjangoMethods(unittest.TestCase):
2.     def test_no_directory(self):
3.         with self.assertRaises(SystemExit) as cm:
4.             dt.detector_django_project('')
5.             self.assertEqual(cm.exception.code, 1)
6.
7.     def test_no_directory_databases(self):
```

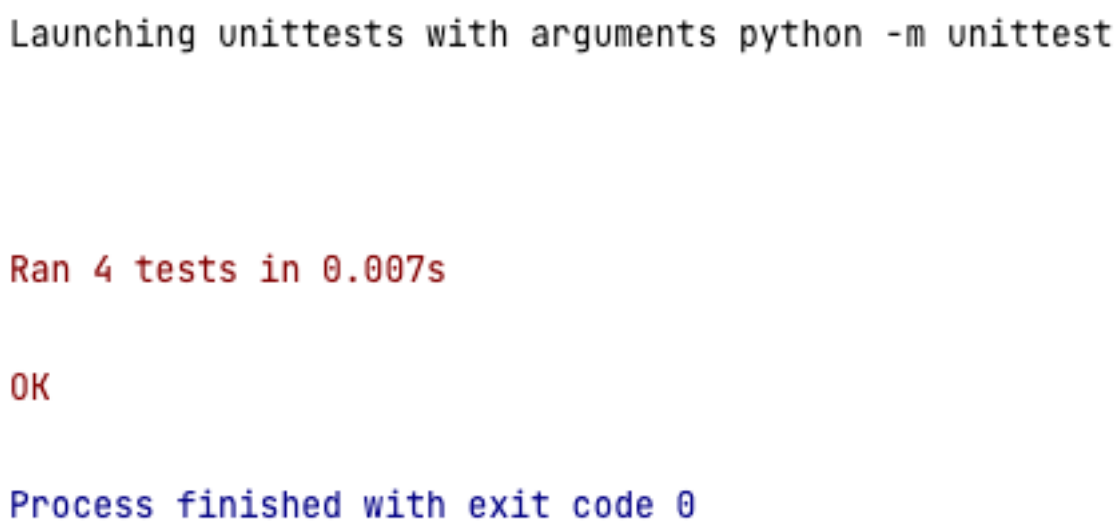
```

8.         with self.assertRaises(SystemExit) as cm:
9.             dt.detect_databases_django('', None)
10.            self.assertEqual(cm.exception.code, 1)
11.
12.        def test_no_database(self):
13.            django_str = django_structure()
14.            dir_project = '/Users/egor/Documents/GitHub/' \
15.                'diplom_IU7_BMSTU/tests/de-
16. modjango\ copy/mysite'
17.            django_str.set_workdir(dir_project.split("/")
18.                [len(dir_pro-
19. ject.split("/")) - 1])
20.            with self.assertRaises(SystemExit) as cm:
21.                dt.detect_databases_django(dir_project,
22. django_str)
23.            self.assertEqual(cm.exception.code, 1)
24.
25.        def test_add_database(self):
26.            message = "Базы данных отличаются"
27.            django_str = django_structure()
28.            dir_project = '/Users/egor/Documents/GitHub/' \
29.                'diplom_IU7_BMSTU/tests/de-
30. modjango/mysite'
31.            django_str.set_workdir(dir_project.split("/")
32.                [len(dir_pro-
33. ject.split("/")) - 1])
34.            django_str_correct = django_structure()
35.            django_str_correct.set_workdir(dir_pro-
36. ject.split("/"))
37.                [len(dir_pro-
38. ject.split("/")) - 1])
39.
40.            struct = dt.detect_databases_django(dir_project,
41. django_str)
42.

```

```
43.         django_str_correct.update_dbs({'ENGINE': 'post-
44. gres', 'NAME': "testdb", 'USER': "postgres",
45.                                         'PASSWORD':
46. "2468", 'HOST': "db", 'PORT': '5432'})
47.
48.         self.assertEqual(struct.dbs,         django_str_cor-
49. rect.dbs)
```

На рисунке 3.7.1 изображен результат выполнения функциональных тестов.



```
Launching unittests with arguments python -m unittest

Ran 4 tests in 0.007s

OK

Process finished with exit code 0
```

Рисунок 3.7.1 — Результат функциональных тестов

Функциональные тесты позволяют своевременно выявить ошибки и, тем самым избежать множества проблем при работе с ним в дальнейшем.

3.6 Выводы из технологического раздела

В технологическом разделе были выбраны язык программирования и среда разработки. Приложение реализовано на языке программирования Python, в качестве среды разработки был выбран PyCharm. Описаны структура данных для Flask, Django, Docker и баз данных, описана реализация поиска фреймворка и конфигураций баз данных, создания docker-файла и запуска контейнеров.

Продемонстрирован интерфейс программы. Проведено функциональное тестирование.

4 Исследовательский раздел

4.1 Описание эксперимента

Для проведения экспериментов было выбрано несколько проектов:

- приложение 1 - проект с фреймворком Django и React и с базой данных SQLite3;
- приложение 2 - проект с фреймворком Django и с базой данных SQLite3;
- приложение 3 - проект с фреймворком Django с пользователями базы данных SQLite3 и PostgreSQL;
- приложение 4 - проект с фреймворком Flask и с базой данных SQLite.
- приложение 5 - проект с фреймворком Flask и с несколькими базами данных PostgreSQL.

В данном разделе будет приведено сравнение времени генерации программы и пользователем контейнеров приложений.

4.2 Результаты проведенного эксперимента

Таблица 4.2.1 Сравнение времени развертывания

	Программа	Пользователь
Приложение 1	30.55	938.43
Приложение 2	8.72	382.34
Приложение 3	35.78	653.13
Приложение 4	5.90	531.53
Приложение 5	50.82	856.12

Графическое представление результатов исследования представлены на рисунке 4.2.1.

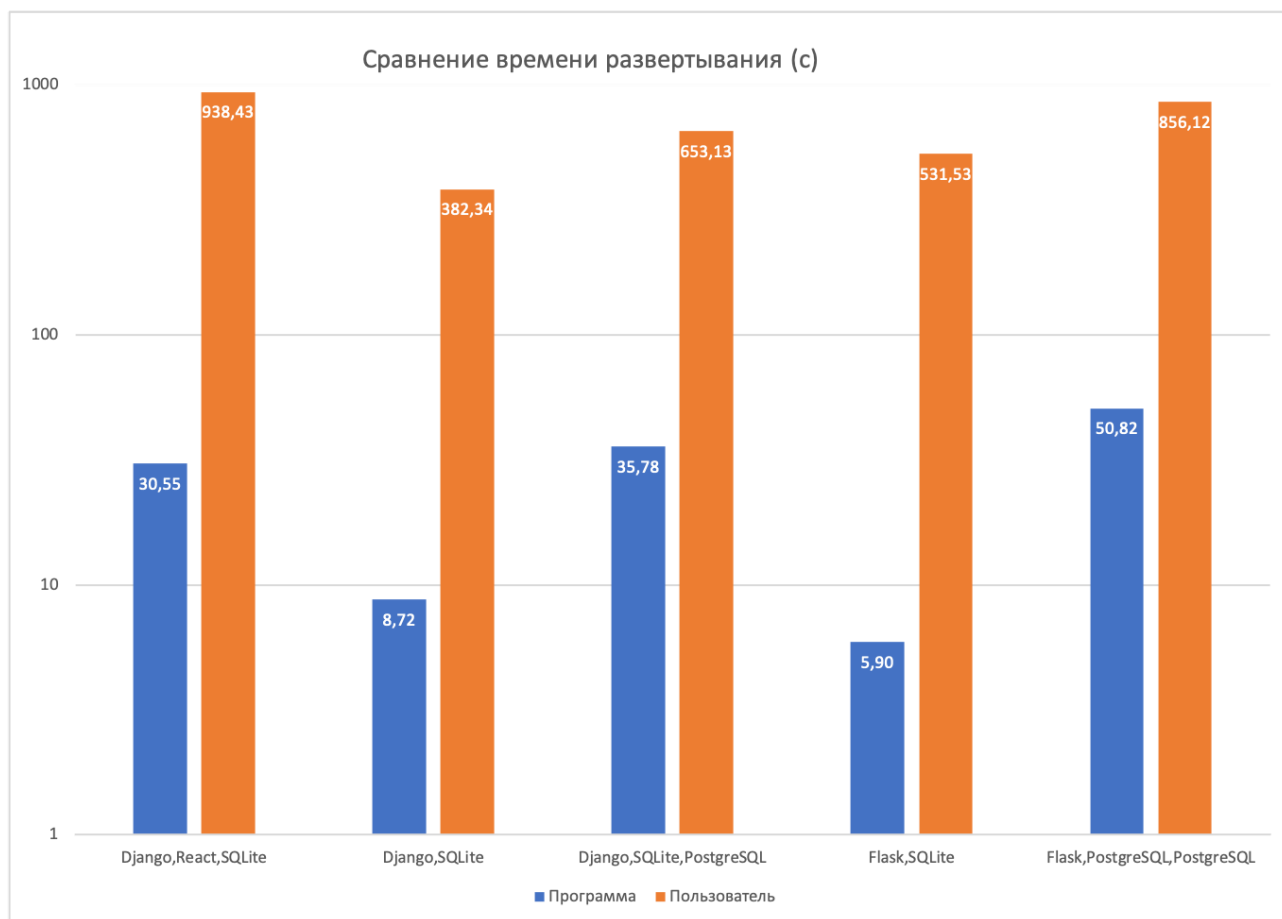


Рисунок 4.2.1 — Результат функциональных тестов

По результатам сравнения можно заметить, что программа справляется более чем в 100 раз быстрее, чем если бы генерировал контейнеры пользователь вручную.

По таблице сравнения времени можно увидеть, что приложения, где присутствовала база данных PostgreSQL, обрабатывались в 3 и более раз дольше, чем приложения с SQLite. Это связано с тем, что необходимо было сгенерировать дополнительный контейнер, скопировать базу данных и настроить сеть для взаимодействия контейнеров.

4.3 Выводы из исследовательского раздела

В результате исследовательской работы была подтверждена корректная работа метода генерации docker-файла. В ходе исследовательской работы было установлено, что метод корректно работает с фреймворками Django и Flask и

React, с базами данных PostgreSQL и SQLite, время развертывания контейнеров значительно сократилось за счет автоматизации процесса.

ЗАКЛЮЧЕНИЕ

Во время выполнения выпускной квалификационной работы была достигнута поставленная цель: разработан метод генерации docker-файла для проекта на языке Python. В ходе выполнения поставленной цели, были решены следующие задачи:

- проанализирована предметная область и существующие средства виртуализации;
- проанализирована эффективность метода внедрения автоматического развертывания контейнеров;
- разработана концепция метода;
- спроектирован и реализован метод генерации docker-файла для проекта на языке Python;
- экспериментальным путем проверена корректность работы разработанного метода.

Преимуществом метода является:

- высокая скорость генерации docker-файла и контейнеризации;
- поддерживаются различные проекты на языке Python;

Однако у метода есть недостатки:

- нет поддержки Windows;
- данные должны быть записаны по определенному шаблону для корректного считывания.
- Требуется менять хост в конфигурации подключения для корректной работы приложения и баз данных в контейнере.

Пути развития работы:

- реализация поддержки Windows;
- реализация большего числа шаблонов для корректного считывания данных различных проектов на языке Python.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Sane, B.O. Niang, I. Fall, D. // A Review of Virtualization, Hypervisor and VW Allocation Security: Threats, Vulnerabilities, and Countermeasures - 2018. – pp. 1318
2. Sagar Ajay Rahalkar // Virtualization and Cloud Basics - 2016. – pp. 71-72
3. Общие сведения о контейнерах и Docker [Электронный ресурс]. – Режим доступа: <https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/container-docker-introduction/>. – (13.02.22);
4. What is containerization? [Электронный ресурс]. – Режим доступа: <https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization#overview/>. – (14.02.22);
5. Kumaran S., Senthil // Practical LXC and LXD: Linux Containers for Virtualization and Orchestration - Apress - 2017. – pp. 4.
6. Cgroups [Электронный ресурс]. – Режим доступа: <https://man7.org/linux/man-pages/man7/cgroups.7.html/>. – (14.02.22);
7. Shashank Mohan Jain // Linux Containers and Virtualization: A Kernel Perspective - 2020. – pp. 32-33
8. Кочер П.С. // Микросервисы и контейнеры Docker – 2019. – С. 54;
9. Васильев Петр Алексеевич // Развертывание сервера с помощью технологии docker – 2016. – 1 с.
10. Docker-образ [Электронный ресурс]. – Режим доступа: <https://cloud.yandex.ru/docs/container-registry/concepts/docker-image>. – (13.02.22);
11. Моуэт Э. // Использование Docker - 2017. – 62-64 с.
12. А.П. Табурчак, П.П. Табурчак, А.А. Севергина // Анализ внедрения и использования информационных технологий – Экономический вектор - 2016. – 1 с.
13. Автоматизация vs Человеческий труд – наступление продолжается [Электронный ресурс]. – Режим доступа: <https://hr-portal.ru/article/avtomatizaciya-vs-chelovecheskiy-trud-nastuplenie-prodolzhaetsya>. – (14.02.22);

14. ИТ: обзор рынка вакансий и топ-15 специальностей [Электронный ресурс]. – Режим доступа: <https://hh.ru/article/24562>. – (13.02.22);
15. Херинг М. // DevOps для современного предприятия - 2020. – 5 с.
16. Зарплаты ИТ-специалистов выросли в среднем на 20% за 3 месяца [Электронный ресурс]. – Режим доступа: <https://press.rabota.ru/zarplaty-it-spetsialistov-vyrosli-v-srednem-na-20-za-3-mesyatsa/>. – (14.02.22);
17. Дефицит в ИТ: за год половина российских компаний заметили эту тенденцию [Электронный ресурс]. – Режим доступа: https://www.cnews.ru/news/line/2021-12-08_defitsit_v_it_za_god_polovina/. – (14.02.22);
18. Python [Электронный ресурс]. – Режим доступа: <https://www.python.org/about/>. – (15.02.22);
19. PyCharm [Электронный ресурс]. – Режим доступа: <https://www.jetbrains.com/ru-ru/pycharm/>. – (16.02.22);

ПРИЛОЖЕНИЕ А

Поиск баз данных для фреймворка Flask

Листинг А.1 – поиск баз данных для фреймворка Flask

```
1.     def detect_databases_flask(dir_project, flask_str):
2.         path = os.path.exists(dir_project)
3.
4.         if path is False:
5.             print("Не найдена директория: " + dir_project)
6.             exit(1)
7.
8.         db = {}
9.         flag_add = False
10.        pattern = re.com-
11.        pile("(\\w+):/(.+):(.)+@(.):(.+)/(.+)\\")
12.        for root, dirs, files in os.walk(dir_project):
13.            for file in files:
14.                try:
15.                    with open(root + '/' + file) as f:
16.                        for line in f.readlines():
17.                            if flag_add:
18.                                line = line.split(" ")
19.                                line = line[len(line) - 1]
20.                                match = re.findall(pattern,
21. line)
22.                                if match:
23.                                    db['ENGINE'] = match[0][0]
24.                                    db['USER'] = match[0][1]
25.                                    db['PASSWORD'] = match[0][2]
26.                                    db['HOST'] = match[0][3]
27.                                    db['PORT'] = match[0][4]
28.                                    db['NAME'] = match[0][5]
29.                                    flag_add = True
30.                                    flask_str.update_dbs(db)
31.                                    db = {}
```

```

32.             except UnicodeDecodeError:
33.                 continue
34.             if flag_add is True:
35.                 break
36.
37.             pattern = re.compile("(.)=\s'(.+):///(.+)'"
38.             for root, dirs, files in os.walk(dir_project):
39.                 for file in files:
40.                     try:
41.                         with open(root + '/' + file) as f:
42.                             for line in f.readlines():
43.                                 match = re.findall(pattern,
44. line)
45.                                 if match:
46.                                     db['ENGINE'] = match[0][1]
47.                                     db['NAME'] = match[0][2]
48.                                     flag_add = True
49.                                     flask_str.update_dbs(db)
50.                                     db = {}
51.             except UnicodeDecodeError:
52.                 continue
53.             if flag_add is True:
54.                 break
55.
56.         if flag_add is False:
57.             print("База данных не найдена")
58.             exit(1)
59.
60.         return flask_str

```

ПРИЛОЖЕНИЕ Б

Поиск баз данных для фреймворка Django

Листинг Б.1 – поиск баз данных для фреймворка Django

```
1.         def detect_databases_django(dir_project, django_str):
2.             settingspy = ut.find('settings.py', dir_project)
3.
4.             if settingspy is None:
5.                 print("Не найден файл: " + dir_project + '/set-
6. tings.py')
7.                 exit(1)
8.
9.                 db = {}
10.                flag_add = False
11.                flag_sqlite3 = False
12.                pattern = re.compile("'(\w+)':\s*([0-9]+|'(.+)')")
13.                with open(settingspy) as f:
14.                    for line in f.readlines():
15.                        match = re.findall(pattern, line)
16.                        if match and 'django.contrib.auth.pass-
17. word_validation.' not in match[0][1] and \
18.                            'BACKEND' not in match[0][0]:
19.                            if
20.                                'django.db.backends.post-
21. gresql_psycopg2' in match[0][1]:
22.                                    db['ENGINE'] = 'postgres'
23.                                    continue
24.                                if
25.                                    'django.db.backends.sqlite3' in
26. match[0][1]:
27.                                        db['ENGINE'] = 'sqlite3'
28.                                        django_str.update_dbs(db)
29.                                        db = {}
30.                                        flag_add = True
31.                                        flag_sqlite3 = True
32.                                        continue
```

```

32.             if 'django.db.backends.mysql' in
33. match[0][1]:
34.                 db['ENGINE'] = 'mysql'
35.                 continue
36.
37.             if flag_sqlite3:
38.                 continue
39.
40.             db[match[0][0]] = match[0][1].re-
41. place("'", "")
42.
43.             if match[0][0] == 'PORT':
44.                 django_str.update_dbs(db)
45.                 db = {}
46.                 flag_add = True
47.
48.             if flag_add is False:
49.                 print("База данных не найдена")
50.                 exit(1)
51.
49.         return django_str

```