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 *Робототехники и комплексной автоматизации*

КАФЕДРА *Системы автоматизированного проектирования (РК-6)*

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

Разработка подсистемы автоматического построения обобщающего документа о ходе научно-образовательных работ по различным направлениям

Студент РК6-82Б _____ **М.Т. Идрисов**
подпись, дата *фамилия, и.о.*

Руководитель курсового проекта _____ **А.П. Соколов**
подпись, дата *фамилия, и.о.*

Консультант _____ **А.Ю. Першин**
подпись, дата *фамилия, и.о.*

Москва, 2020 г.

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)

УТВЕРЖДАЮ

Заведующий кафедрой РК-6
(Индекс)

_____А.П. Карпенко_
(И.О.Фамилия)

«____» _____ 20___ г.

З А Д А Н И Е
на выполнение курсовой работы

по дисциплине __Технологии интернет_____

Студент группы __РК6-82Б_____

_____Идрисов Марат Тимурович_____
(Фамилия, имя, отчество)

Тема курсовой работы

Разработка подсистемы автоматического построения обобщающего документа о ходе научно-образовательных работ по различным направлениям

Направленность КП (учебная, исследовательская, практическая, производственная, др.)

учебная

Источник тематики (кафедра, предприятие, НИР) _____

График выполнения КР: 25% к *3* нед., 50% к *10* нед., 75% к *15* нед., 100% к *17* нед.

Техническое задание Необходимо разработать подсистему автоматического построения обобщающего документа о ходе научно-образовательных работ по различным направлениям

Оформление курсовой работы:

Расчетно-пояснительная записка на _19__ листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)

_____

_____

Дата выдачи задания «18» февраля 2020 г.

Руководитель курсовой работы _____ ___М.Т. Идрисов_____
(Подпись, дата) (И.О.Фамилия)

Студент _____ ___А.П. Соколов_____
(Подпись, дата) (И.О.Фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

Аннотация

Работа посвящена разработке подсистемы автоматического построения обобщающего документа о ходе научно-образовательных работ по различным направлениям.

Тип работы: курсовой проект.

Тема проекта: разработке подсистемы автоматического построения обобщающего документа о ходе научно-образовательных работ по различным направлениям.

Объект исследований: научно-исследовательские работы

Оглавление

1. Введение

В образовательной среде существует проблема на предмет загрузки своих проектов в систему Gitlab. В ввиду их большого количества не представляется возможным структурировать их надлежащим образом, что вызывает определенные трудности. Отчеты хаотично расположены в репозиториях, каждый формирует отчет согласно своим собственным представлениям. Наша задача состояла в том, чтобы привести все отчеты к единой структуре и свести их в единый документ. Это позволит значительно сократить время на создание итоговых отчетов и наглядно продемонстрировать этапы выполнения по каждому из научных направлений. Основная цель курсового проекта была направлена на создание и автоматизацию всего процесса.

2. Анализ работы

2.1. Анализ и выбор архитектуры приложения

Современная архитектура ПО начала отходить от крупных монолитных приложений [1]. Теперь основное внимание в вопросах архитектуры уделялось достижению высокого уровня масштабируемости без ущерба для производительности и доступности. Разбивая монолит на компоненты, инженерные организации предпринимали усилия по децентрализации управления изменениями, предоставляя командам больше контроля над тем, как функции вводятся в эксплуатацию. Повышая изолированность между компонентами, команды создателей ПО начали вступать в мир разработки распределенных систем, фокусируясь на написании менее крупных, более специализированных сервисов с независимыми циклами выпуска.

В приложениях, оптимизированных для выполнения в облаке, используется набор принципов, позволяющих командам более свободно оперировать способами ввода функций в эксплуатацию. По мере роста распределенности приложений (в результате повышения степени изолированности, необходимой для

предоставления большего контроля над ситуацией командам, владеющим приложением) возникает серьезная проблема, связанная с повышением вероятности сбоя при обмене данными между компонентами приложения. Неизбежным результатом превращения приложений в сложные распределенные системы становятся эксплуатационные сбои.

Архитектуры приложений, оптимизированных для работы в облачной среде, придают этим приложениям преимущества исключительно высокой масштабируемости, притом гарантируя их всеобщую доступность и высокий уровень производительности.

При проектировании мы опирались на одни из главных принципов микросервисной архитектуры [2]:

- модули можно легко заменить в любое время: акцент на простоту, независимость развёртывания и обновления каждого из микросервисов;
- модули организованы вокруг функций: микросервис по возможности выполняет только одну достаточно элементарную функцию;
- модули могут быть реализованы с использованием различных языков программирования, фреймворков, связующего программного обеспечения, выполняться в различных средах контейнеризации, виртуализации, под управлением различных операционных систем на различных аппаратных платформах: приоритет отдаётся в пользу наибольшей эффективности для каждой конкретной функции, нежели стандартизации средств разработки и исполнения;
- архитектура симметричная, а не иерархическая: зависимости между микросервисами одноранговые.

Философия микросервисов фактически копирует философию Unix [3], согласно которой каждая программа должна «делать что-то одно, и делать это хорошо» и взаимодействовать с другими программами простыми средствами: микросервисы минимальны и предназначаются для единственной функции. Основные изменения, в связи с этим налагаются на организационную культуру,

которая должна включать автоматизацию разработки и тестирования, а также культуру проектирования, от которой требуется предусматривать обход прежних ошибок, исключение по возможности унаследованного кода.

Наиболее популярная среда для выполнения микросервисов - технология Docker [4], в этом случае каждый из микросервисов как правило изолируется в отдельный контейнер или небольшую группу контейнеров, доступную по сети другим микросервисам и внешним потребителям, и управляется средой оркестрации, обеспечивающей отказоустойчивость и балансировку нагрузки. Типовой практикой является включение в контур среды выполнения системы непрерывной интеграции [5].

2.2. Выбор фреймворка

При разработке приложения выбор пал на фреймворк Django [6], так как обладает существенными преимуществами, по сравнению с другими фреймворками, такие как Flask [7], Tornado [8]:

- ORM - Object-Relational Mapping или объектно-реляционное отображение — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных». Этим отображением в Django называются «модели».
- Миграции базы данных - переход от одной структуры базы данных к другой без потери косистентности.
- Автоматический интерфейс панели администратора. Используются мета-данные модели чтобы предоставить многофункциональный, готовый к использованию интерфейс для работы с содержимым.
- Стандартизированная структура - задаёт структуру проекта. Она помогает разработчикам понимать, где и как добавлять новую функциональность. Благодаря одинаковой для всех проектов структуре гораздо проще найти уже готовые решения или получить помощь от сообщества. Огромное

количество увлеченных разработчиков поможет справиться с любой задачей гораздо быстрее.

3. Программная реализация

3.1. Создание Docker образа

Когда мы создаем образ Docker для своего приложени, написанного на Python, мы должны построить его поверх существующего образа - и есть много возможных вариантов. Существуют образы ОС, такие как Ubuntu и CentOS, а также существует множество различных вариантов python образов.

Существует ряд общих критериев выбора базового образа, которыми мы должны руководствоваться:

- **Стабильность.** Мы хотим, чтобы сборка предоставляла тот же базовый набор библиотек, структуру каталогов и инфраструктуру, что и завтра, иначе приложение будет случайным образом ломаться.
- **Обновления безопасности:** мы хотитим, чтобы базовый образ был в хорошем состоянии, чтобы вы своевременно получали обновления безопасности для базовой операционной системы.
- **Современные зависимости:** мы создаем сложное приложение и будетем зависеть от установленных в операционной системе библиотек и приложений. Мы хотели бы, чтобы они не были слишком старыми.
- **Наиболе новая версия Python:** хотя это можно обойти, установив Python самостоятельно, наличие современного Python экономит наши усилия.
- **Небольшой размер образа:** при прочих равных условиях лучше иметь образ Docker меньшего размера, чем образ Docker большего размера.

Существуют три основные операционные системы, которые примерно соответствуют вышеуказанным критериям (даты и версии выпуска являются точными на момент написания; с течением времени может потребоваться несколько иной выбор).

- Ubuntu 18.04 (ubuntu:18.04 образ) был выпущен в апреле 2018 года, и, поскольку это релиз долгосрочной поддержки, он будет получать обновления безопасности до 2023 года [9].
- Ubuntu 20.04 (ubuntu:20.04 образ) был выпущен в конце апреля 2020 года, и, поскольку это релиз долгосрочной поддержки, он получит обновления безопасности до 2025 года [10].
- CentOS 8 (centos:8) был выпущен в 2019 году и будет иметь полные обновления до 2024 года и обновления до 2029 года [11].
- Debian 10 («Buster») был выпущен 6 июля 2019 года и будет поддерживаться до 2024 года [12].

Только Ubuntu 20.04 включает в себя последнюю версию Python.

Также существую «официальные python образы» [13], который поставляется с предварительно установленной с несколькими версиями Python (3.5, 3.6, 3.7, 3.8 бета и т.д.), и имеет несколько вариантов:

- Debian Buster, с множеством установленных пакетов. Сам образ является большим, но теория состоит в том, что эти пакеты устанавливаются через общие слои образов, которые будут использоваться другими официальными образами Docker, поэтому общее использование диска будет низким.
- slim вариант Debian Buster. В нем отсутствуют слои общих пакетов, поэтому сам образ намного меньше.

В итоге выбор пал на образ **python:3.8-slim-buster**. Он актуальнее, чем ubuntu:18.04, стабилен, не будет иметь значительных изменений в библиотеке и меньше шансов получения ошибок производства, чем в образе Alpine. 60 МБ при загрузке, 180 МБ без сжатия на диске [14], он предоставляет последнюю версию Python и обладает всеми преимуществами Debian Buster.

Для построения отчетов используется язык компьютерной верстки TeX, компилятор для которого не установлен в базовом образе **python:3.8-slim-buster.**

Требуется дополнить базовый образ недостающими пакетами для решения этой проблемы. Был написан Dockerfile для установки недостающих пакетов:

Листинг 1. Dockerfile с пакетами для компиляции TeX файлов

```
FROM python:3.8-slim-buster
COPY /tmp /tmp

RUN apt-get update \
&& apt-get install latexmk texlive-lang-cyrillic texlive-latex-recommended
texlive-pictures texlive-latex-extra -y \
&& cd "$_" \
&& latex hyphenat.ins \
&& mkdir -p /usr/share/texlive/texmf-dist/tex/latex/hyphenat \
&& mv hyphenat.sty /usr/share/texlive/texmf-dist/tex/latex/hyphenat
```

Базовый образ состоит из стека слоев, которые доступны только для чтения (иммутабельны) (рисунок 1), а все изменения происходят в верхнем слое стека

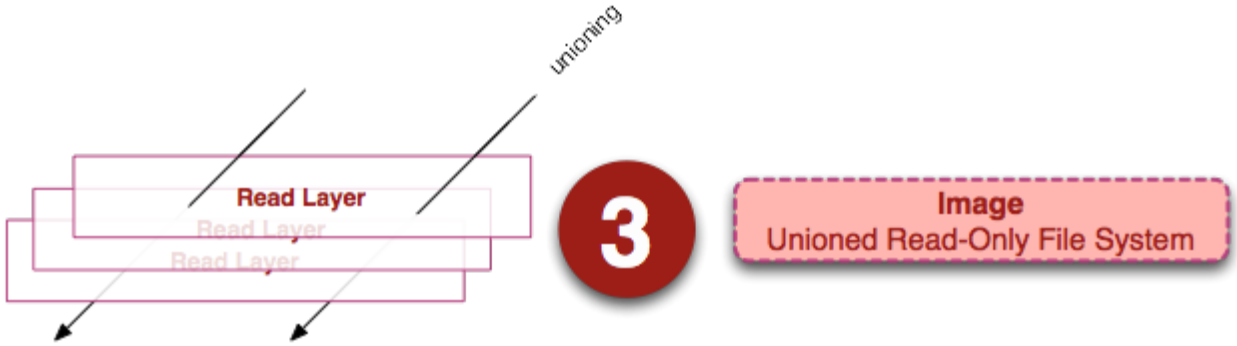

Рисунок 1 Общая структура базового Docker образа

Для собственного базового образа нужно добавить верхний слой для записи наверх стека слоев (рисунок 2) и записать изменения и превратить верхний слой в слой для чтения (рисунок 3).

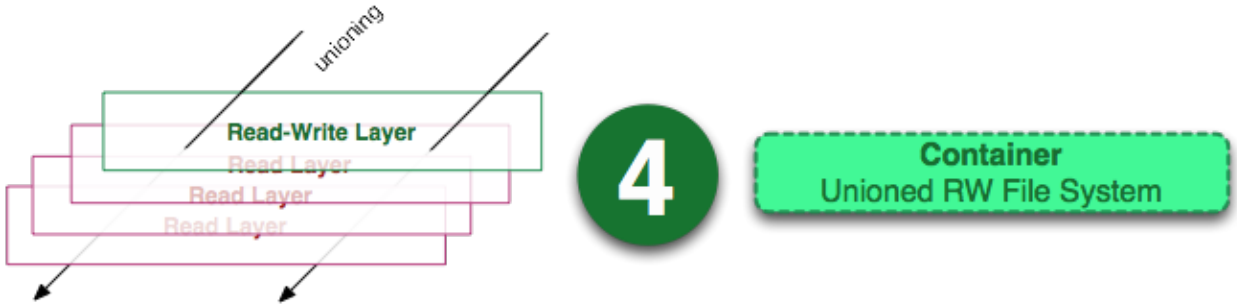

Рисунок 2 Добавление в стек верхнего слоя для записи

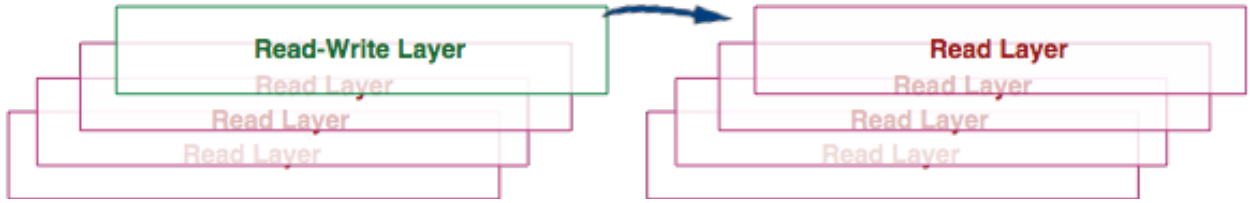

Рисунок 3 Преобразование верхнего слоя в слой для записи

Данную цепочку преобразований, которая выполняет команда docker build представлена на рисунке 4.

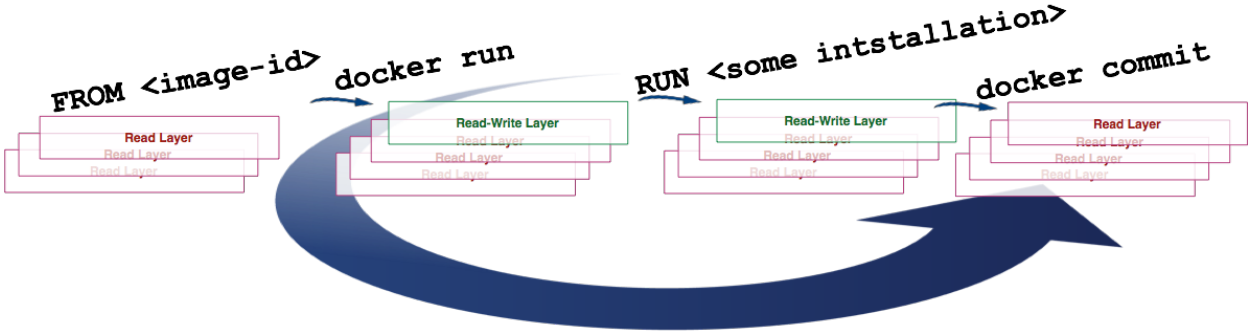

Рисунок 4 Цепочка преобразований, выполненная docker build

Команда build использует значение инструкции FROM из файла Dockerfile как базовый образ после чего:

1) запускает контейнер (create и start)
2) изменяет слой для записи
3) делает commit

Мы получили базовый образ, который был назван **python-latexmk:3.7.6-slim-buster.**

С помощью команды **docker pull** мы разместили данный образ в Docker Hub - крупнейшую в мире библиотеку для контейнерных образов [15], для возможности его использования в основном проекте.

3.2. Реализация основных компонентов

Отчеты располагаются в репозитории GitLab. Требуется API-интерфейс, позволяющий получать данные и оперировать ими. Gitlab предоставляет API-интерфейс вместе с подробной документацией [16]. Чтобы не посылать get/post запросы, был использован модуль python-gitlab [17], предоставляя возможность работать с сущностями GitLab как python-объектами.

Как было замечено ранее, Django включает в себя ORM, позволяя оперировать не запросами к базе данных, а методами и классами python.

На основе схемы таблиц были сформированы соответствующие Django модели:

Листинг 2. Django-модели базы данных.

```
from django.db import models


class Solun(models.Model):
    slnid = models.CharField(primary_key=True, max_length=3)
    dscra = models.CharField(max_length=50)
    slnna = models.CharField(unique=True, max_length=20)
    cpxid = models.ForeignKey('Cmplx', models.DO_NOTHING, db_column='cpxid',
blank=True, null=True)
    dscrb = models.TextField(blank=True, null=True)
    dscrc = models.TextField(blank=True, null=True)
    dirna = models.CharField(max_length=70, blank=True, null=True)
    rlvnc = models.CharField(max_length=3, blank=True, null=True)
```

```
    class Meta:
        db_table = 'solun'
        unique_together = (('slnid', 'cpxid'),)

class Tmpls(models.Model):
    tmlid = models.CharField(primary_key=True, max_length=3)
    catid = models.ForeignKey('Tmcat', models.DO_NOTHING, db_column='catid',
blank=True, null=True)
    activ = models.NullBooleanField()
    rlpth = models.TextField(blank=True, null=True)
    dscra = models.TextField(blank=True, null=True)
    versi = models.CharField(max_length=14, blank=True, null=True)
    foldr = models.CharField(max_length=35, blank=True, null=True)
    rlvnc = models.CharField(max_length=3, blank=True, null=True)

    class Meta:
        db_table = 'tmpls'

class Cmplx(models.Model):
    cpxid = models.CharField(primary_key=True, max_length=3)
    dscra = models.CharField(max_length=50, blank=True, null=True)
    cpxna = models.CharField(unique=True, max_length=20)
    pcpxi = models.ForeignKey('self', models.DO_NOTHING, db_column='pcpxi',
blank=True, null=True)
    dscrc = models.TextField(blank=True, null=True)
    dirna = models.CharField(max_length=70, blank=True, null=True)
    rlvnc = models.CharField(max_length=3, blank=True, null=True)

    class Meta:
        db_table = 'cmplx'

class Tmcat(models.Model):
    catid = models.CharField(primary_key=True, max_length=3)
    dscra = models.TextField(blank=True, null=True)
    foldr = models.CharField(max_length=35, blank=True, null=True)
    rlvnc = models.CharField(max_length=3, blank=True, null=True)

    class Meta:
        db_table = 'tmcat'
```

Основой цикл работы приложения представлен реализован двумя классами: GitLabExtractor и ReportCreator:

- Классе GitLabExtractor отвечает за всю работу с сущностями GitLab (получение, загрузка отчетов)
- Класс ReportCreator реализует всю работу по созданию промежуточных и итогового отчета. Блок-схема программы представлена на рисунке 5.

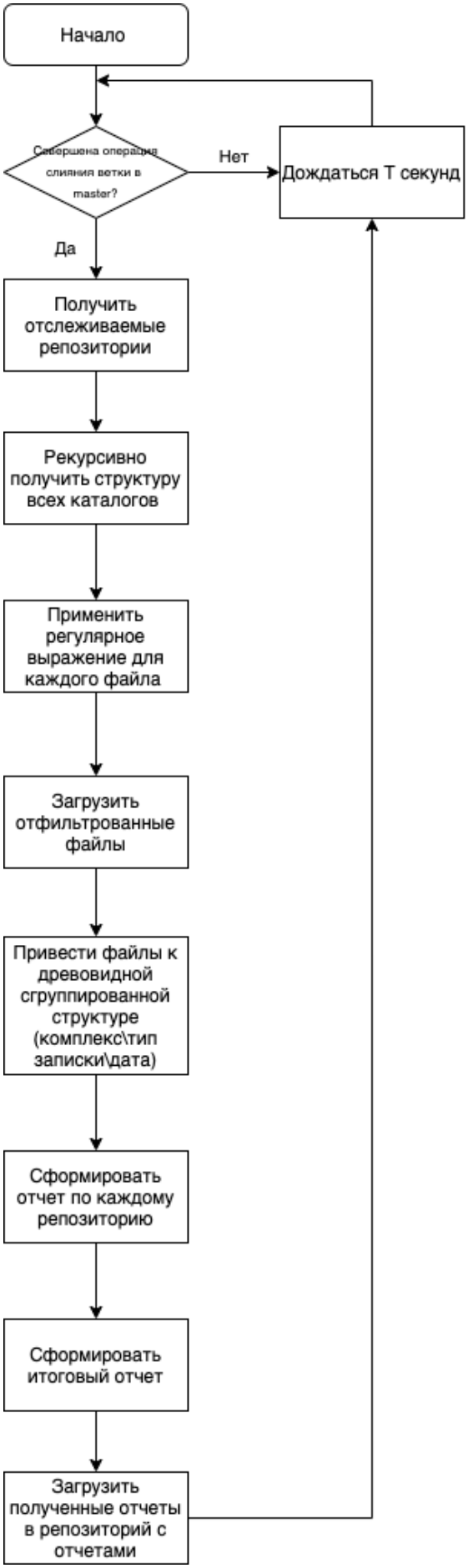

Рисунок 5. Блок-схема программы

4. Тестирование программы

Для тестирования применялся метод модульного тестирования, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными. Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

За модульное тестирование отвечает модуль pytest [18]. Для тестирования Django-моделей были созданы фикстуры – тестовые данные, сформированные из базы данных по таблицам solun, tmpls, cmplx и tmcat. В итоге получилось 326 тестовых объектов. Ниже представлен пример тестирования функционала группировки списка отчетов:

Листинг 3. Пример модульного теста

```
def test_group_by_type(self):
    notes = [
        {'id': 'f0b4b21c607290d52c35af3ca6b36270a4d92fc9', 'name':
'rndhpc_rem_2020_04_09_n01.tex', 'type': 'rem',
         'path': 'rnddoc/rndhpc_rem_2020_04_09_n01.tex', 'mode': '100644',
'complex': 'rndhpc', 'year': '20',
         'month': '04', 'day': '09', 'title': 'n01'},
        {'id': '78e41117f4663ac886c01c4790ccf5524e194fc9', 'name':
'rndhpc_rem_2025_12_10_214124.tex',
         'type': 'rem', 'path': 'rnddoc/rndhpc_rem_2025_12_10_214124.tex',
'mode': '100644', 'complex': 'rndhpc',
         'year': '25', 'month': '12', 'day': '10', 'title': '214124'}]

    grouped_notes = self.worker.group_by_type(notes)

    assert grouped_notes == {
        'Разработка ресурсоемкого ПО инж.анализа.          ': {'Заметка общего
назначения': [{'complex': 'rndhpc',

'day': '10',

'id': '78e41117f4663ac886c01c4790ccf5524e194fc9',

'mode': '100644',

'month': '12',

'name': 'rndhpc_rem_2025_12_10_214124.tex',

'path': 'rnddoc/rndhpc_rem_2025_12_10_214124.tex',

'title': '214124',

'type': 'rem',

'year': '25'},

{'complex': 'rndhpc',

'day': '09',

'id': 'f0b4b21c607290d52c35af3ca6b36270a4d92fc9',

'mode': '100644',

'month': '04',

'name': 'rndhpc_rem_2020_04_09_n01.tex',

'path': 'rnddoc/rndhpc_rem_2020_04_09_n01.tex',

'title': 'n01',

'type': 'rem',

'year': '20'}]}}
```

5. Результат работы программы

Для демонстрации результата работы в репозитории были добавлены несколько фиктивных отчетов. Получившийся отчет представлен на рисунке 6.

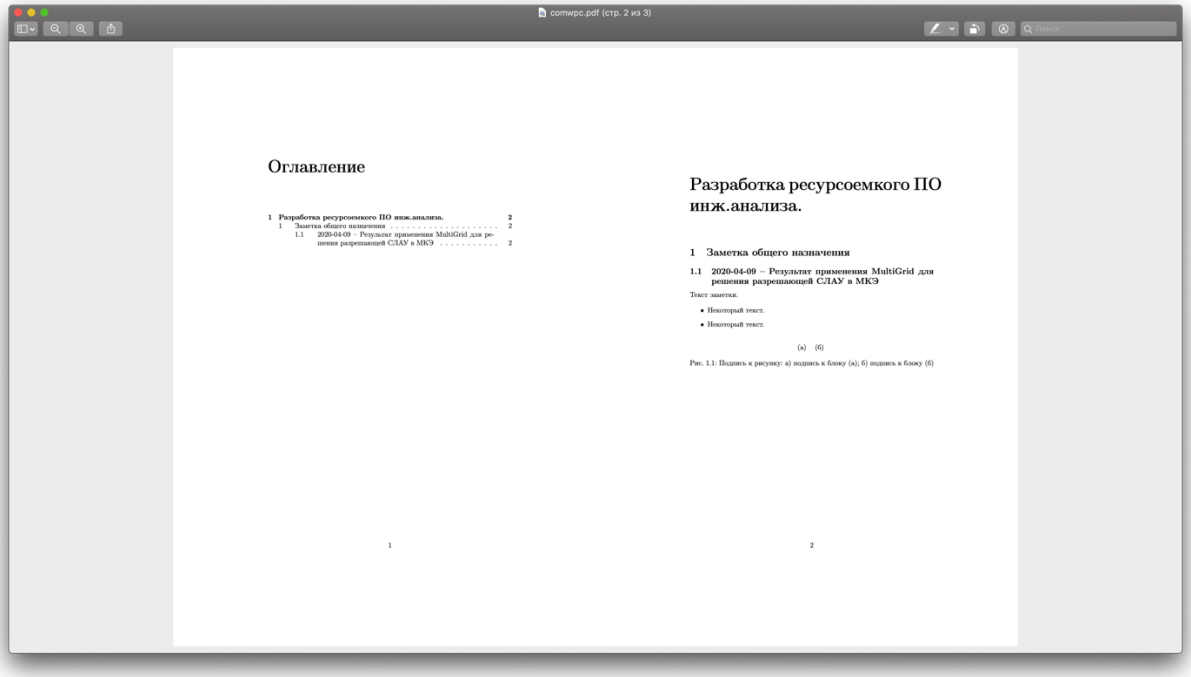

Рисунок 6. Пример отчета

6. Заключение

В данной работе были описаны принципы микросервисной архитектуры приложений, которые являются основой разработки ПО в современных реалиях. Приведена аргументация, в пользу выбора фреймворка Django.

Было разработано приложение, автоматизирующее процесс создания отчетов.

7. Список литературы

1. Конференция ИМПОРТОЗАМЕЩЕНИЕ 2020: РЕАЛЬНЫЙ ОПЫТ [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: http://www.tadviser.ru/index.php/%D0%9A%D0%BE%D0%BD%D1%84%D0%B5%D1%80%D0%B5%D0%BD%D1%86%D0%B8%D1%8F:%D0%98%D0%BC%D0%BF%D0%BE%D1%80%D1%82%D0%BE%D0%B7%D0%B0%D0%BC%D0%B5%D1%89%D0%B5%D0%BD%D0%B8%D0%B5_2020:_%D1%80%D0%B5%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BE%D0%BF%D1%8B%D1%82
2. Microservices [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://en.wikipedia.org/wiki/Microservices
3. Basics of the Unix Philosophy [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: http://www.catb.org/esr/writings/taoup/html/ch01s06.html
4. Docker overview [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://docs.docker.com/get-started/overview/
5. Continuous Integration [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: http://wiki.c2.com/?ContinuousIntegration
6. Django [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://www.djangoproject.com/
7. Welcome to Flask’s documentation [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://flask.palletsprojects.com/en/1.1.x/
8. Tornado is a Python web framework [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://github.com/tornadoweb/tornado
9. Ubuntu 18.04.4 LTS (Bionic Beaver) [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://releases.ubuntu.com/18.04.4/
10. Ubuntu 20.04 LTS (Focal Fossa) [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://releases.ubuntu.com/20.04/
11. CentOS-8 (1911) Release Notes [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа:

https://wiki.centos.org/action/show/Manuals/ReleaseNotes/CentOS8.1911?action=show&redirect=Manuals%2FReleaseNotes%2FCentOSLinux8

12. Выпущен Debian 10 "buster" [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://www.debian.org/News/2019/20190706.ru.html
13. Python - Docker Hub [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://hub.docker.com/_/python
14. python:3.8-slim-buster [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://hub.docker.com/layers/python/library/python/3.8-slim-buster/images/sha256-527bd4f643ae1885abeaa483cc675e2cee5b958612012d60ec10455ac5405270?context=explore
15. Docker Hub is the world's easiest way to create, manage, and deliver your teams' container applications [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://hub.docker.com/
16. Automate GitLab via a simple and powerful API [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://docs.gitlab.com/ee/api/README.html
17. Python wrapper for the GitLab API [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://github.com/python-gitlab/python-gitlab
18. Full pytest documentation [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://docs.pytest.org/en/latest/contents.html