Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
Институт Вычислительной математики и информационных технологий

# ОТЧЕТ ПО ПРОИЗВДСТВЕННОЙ ПРАКТИКЕ
(технологическая (проектно-технологическая) практика)

Обучающийся Фаттахов Тимур Рамильевич гр.09-911 _____
(ФИО студента) (Группа) (Подпись)

Руководитель практики
от кафедры _____ доцент КТК Байрашева В.Р_____

Оценка за практику _____ _____
(Подпись)

Дата сдачи отчета 24.05.2023

Казань – 2023

# Оглавление

# Введение

Автоматизация процессов является неотъемлемой частью успеха современного бизнеса. Существует множество подходов и инструментов для повышения эффективности управления бизнес-процессами, одним из которых является внедрение автоматизированной системы управления проектами (АСУП).

Однако, на малом бизнесе, ресурсы для разработки и внедрения такой системы обычно ограничены. В этом случае, для успешной реализации проектов, необходимо разработать инновационный подход в управлении проектами, который позволит сократить бизнес-затраты и повысить эффективность управления проектами.

Цель данной работы - разработка концепции автоматизированной системы управления проектами для малого бизнеса, которая будет удобной и простой в использовании. В рамках работы будет проведен анализ существующих АСУП и выбран наиболее подходящий для решения поставленной задачи.

Также будут выявлены основные проблемы, с которыми сталкиваются представители малого бизнеса при управлении проектами, и предложены решения для их решения. Будет рассмотрена связь между проектным управлением и бизнес-стратегией, а также определены показатели проектной деятельности, которые позволят оценить эффективность работы АСУП.

Резюмируя, данная производственная практика будет посвящена разработке концепции автоматизированной системы управления проектами для малого бизнеса, что позволит повысить эффективность управления проектами, уменьшить затраты на управление, а также повысить качество выполняемых проектов.

# Глава I. Постановка задачи

## 1.1 Требования к реализации цели

Чтобы реализовать идеи, необходимо составить список требований к продукту.

Список требований:

1. Распределение ролей: администратор, пользователь, продажник, разработчик.
2. Возможность добавлять, редактировать, удалять, сдавать администратору задачи.
3. Возможность прикрепления файлов к задачам.
4. Администратор имеет возможность принимать или отправлять на доработку выполненные сотрудником задачи.

## 1.2 Техническое задание

Основной функционал приложения:

1 Регистрация и авторизация пользователя с помощью JWT токенов.
2 Возможность создавать задания продажником для разработчиков.
3 Возможность администратора вносить пометки к заданиям и устанавливать дедлайн.
4 Администратор имеет право распределять роли между пользователями.
5 Сотрудник после выполнения задания сдает его на проверку администратору. Администратор либо принимает работу, либо возвращает на доработку.

На рисунке 1.2.1 можем увидеть диаграмму прецедентов проекта:

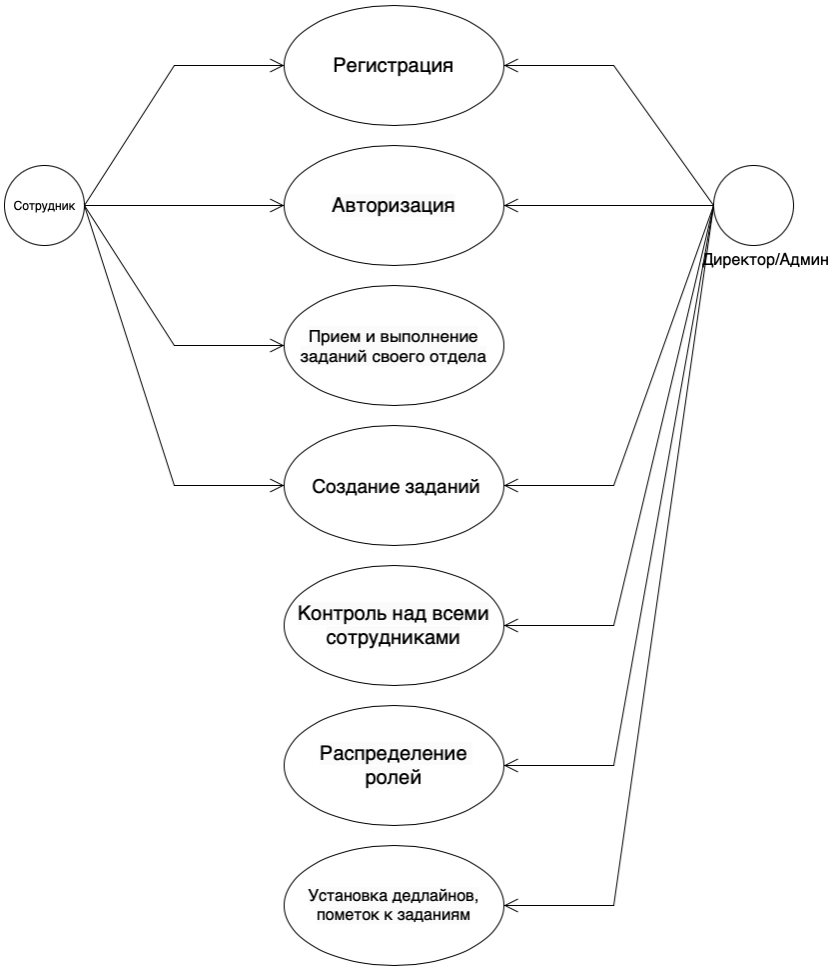


*Рисунок 1.2.1 Диаграмма прецедентов*

# Глава II. Выбор технологии для реализации проекта

## 2.1 Django Rest Framework

Для разработки web-сервиса был выбран язык программирования Python [1] и фреймворк Django.

Django[2] – это свободный фреймворк для веб приложений на языке Python.

Плюсы данного фреймворка:

- Стандартизированная структура
- Большинство инструментов для создания приложения уже присутствуют в данном фреймворке
- Приложения Django
- Безопасность
- Django Rest Framework[3] – библиотека для построения API

## 2.2 Vue.js

Для реализации клиентской части веб приложения было решено использовать Vue.js[4]. Это JavaScript – фреймворк, который подходит для создания пользовательских интерфейсов и сложных одностраничных приложений

Почему был выбран данный фреймворк:

- Подробная документация
- Высокая производительность
- Небольшой вес (около 20КБ)

## 2.3 СУБД PostgreSQL

PostgreSQL — это реляционная база данных с открытым кодом, которая является одной из наиболее известных среди всех существующих реляционных баз данных. База данных PostgreSQL является гибкой и

целостной. Например, база данных PostgreSQL поддерживает как реляционные, так и нереляционные запросы.

## 2.4 Postman

Данная программа используется для тестирования и обработки всех запросов. На стороне сервера без использования клиентской части.

# Глава III. Разработка веб-сайта

## 3.1 Структура проекта

Рассмотрим структуру серверной части проекта на рисунке 3.1.1

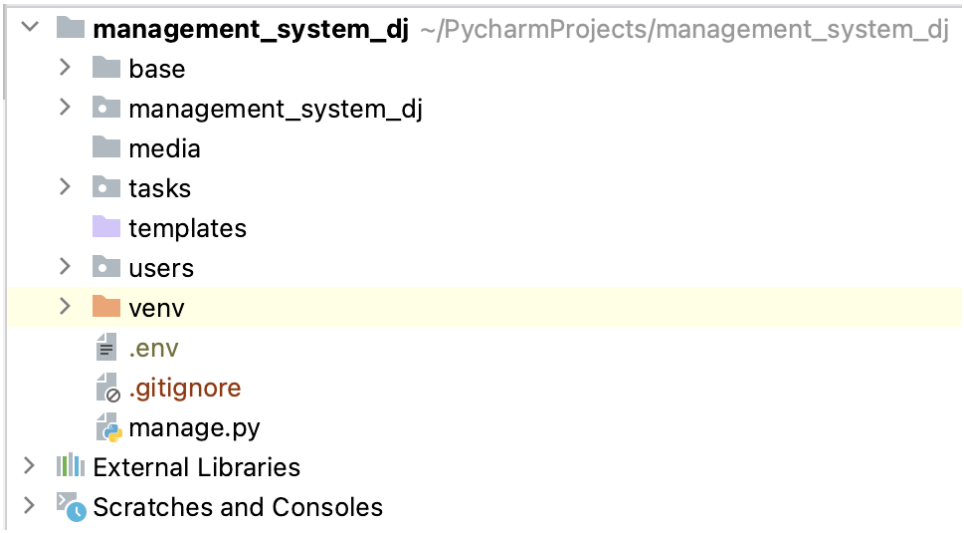


*Рисунок 3.1.1 Структура проекта*

Проект на Django по умолчанию состоит из нескольких базовых файлов. Папка management_system_dj внутри приложения – это, по сути, главная директория серверной части проекта. А директории tasks, users – это приложения проекта, которые не зависят друг от друга. manage.py используется для взаимодействия проекта с командной строкой (запуск/остановка сервера, создание суперпользователя, миграции).

На рисунке 3.1.2 представлена директория management_system_dj.

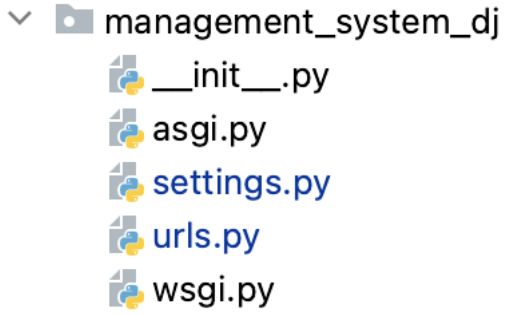


*Рисунок 3.1.2 Структура директории management_system_dj*

Рассмотрим его структуру поподробнее:

- __init.py__ - необходим для инициализации пакета.
- asgi.py – Данный Pyhon файл необходим для асинхронности, так как Django сам по себе синхронный.

- settings.py – Тут название говорит само за себя. В данном файле прописываются все настройки проекта. Например: секретный ключ проекта, тут регистрируются все наши приложения, подключаются все фреймворки, конфигурации базы данных и т.д.
- urls.py – Здесь прописываем все URL – адреса проекта
- wsgi.py – этот файл помогает взаимодействовать Django с веб-сервером.

Теперь давайте рассмотрим структуру отдельного приложения на рисунке 3.1.3:

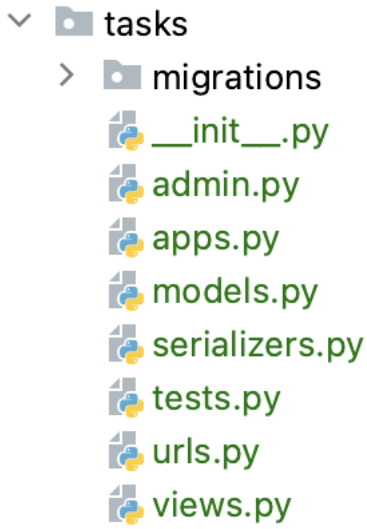


*Рисунок 3.1.3 Структура приложения tasks*

- Директория migrations – В данной директории хранятся все созданные модели и их связи
- __init.py__ - Необходим для инициализации пакета.
- admin.py - В данном файле настраивается административная панель проекта, для данного приложения
- apps.py – Именно с помощью данного файла приложение подключается к проекту
- models.py – С помощью них веб-приложения Django получают доступ и управляют базами данных
- serializers.py – Тут преобразовываем данные из базы данных в JSON формат
- tests.py – Данный файл необходим для написания тестов
- urls.py - Здесь прописываем все URL – адреса данного приложения

- views.py – Тут прописываем все наши представления, связанные с нашим приложением.

  В Django представления делятся на два типа:

  1. Представления на основе функции
  2. Представления на основе классов

С помощью рисунка 3.1.4 рассмотрим более подробно структуру клиентской части проекта:

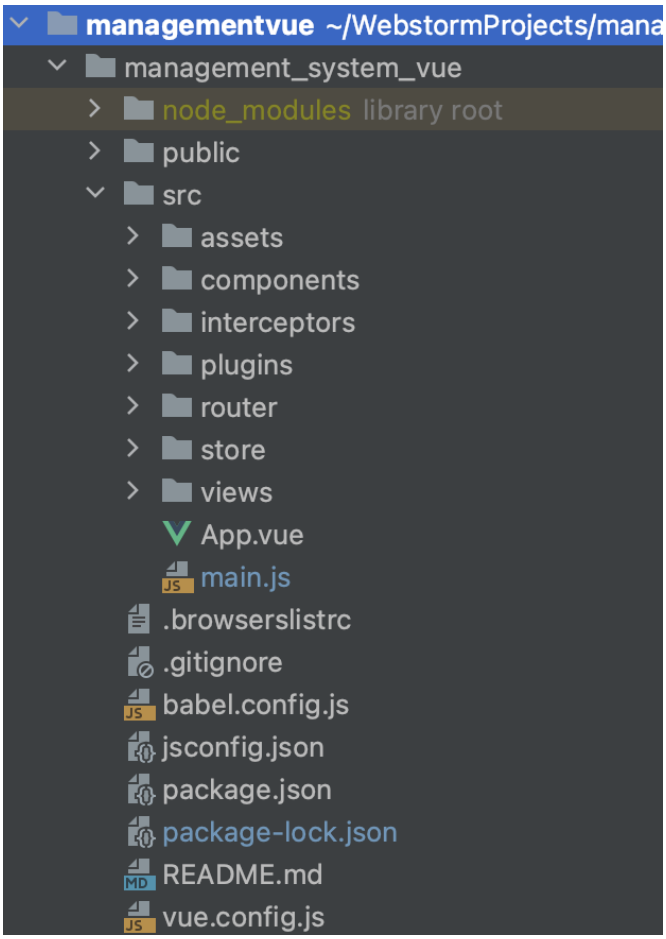


*Рисунок 3.1.4 Структура клиентской части проекта*

• public - Папка с чистым html, где указан тэг div с идентификатором app, которая служит ссылкой для vue.js

• src – Папка с vue содержимым:

1. assets – тут хранятся статические файлы, такие как: картинки, видео, шрифты, стили, скрипты.
2. components – здесь храним наши шаблонные vue компоненты.
3. router - здесь пишем маршруты и подключаем их к нашим компонентам пользовательского интерфейса.

4. store - данная директория отвечает за хранилище Vuex. Vuex – это паттерн управления состоянием.
5. views - тут хранятся все наши vue компоненты.
6. App.vue - это компонент точки входа, это основной компонент пользовательского интерфейса, в котором будут отображаться все остальные компоненты.
7. main.js - файл точки входа, который будет монтироваться App.vue - наш основной компонент пользовательского интерфейса.

## 3.2 Реализация серверной части проекта

### 3.2.1 Создание и настройка проекта

Для того, чтобы реализовать проект, для начала нам необходимо создать New Project Django на PyCharm. Далее нам необходимо установить с помощью pip Django Rest Framework. Для этого необходимо прописать следующее в командной строке:

```
pip3 install djangorestframework
```

Чтобы данный фреймворк работал с нашим веб-приложением, необходимо подключить его в настройках проекта. На рисунке 3.2.1.1 изображено, как это необходимо сделать:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'users.apps.UsersConfig',
    'rest_framework',
    'rest_framework_jwt',
    'djoser',
    'corsheaders',
    'tasks',
]
```

*Рисунок 3.2.1.1 Подключенные приложения к проекту*

В INSTALLED_APPS можем увидеть, что у нас подключены и другие сторонние приложения. Рассмотрим, для чего они нужны.

1. rest_framework_jwt нам необходим для авторизации по токенам

2. corsheaders необходим, чтобы разрешить доступ из нашего клиентского приложения к нашему REST API, так как они будут расположены на разных серверах
3. djoser отвечает за регистрацию и авторизацию пользователей по токенам[5]

После установки всех сторонних фреймворков можем создать свои приложения, которые будут нам необходимы. Для этого в командной строке необходимо прописать следующее:

```
pip3 manage.py startapp tasks
```

Наши приложения так же необходимо подключить в настройки проекта, как это мы можем увидеть на рисунке 3.2.1.1

### 3.2.2 Регистрация и Авторизация

В сервисе предусмотрена аутентификация по JWT токенам. Данная библиотека генерирует 2 токена для определенного пользователя. Перый токен – access token, второй – refresh token. Первый токен необходим для аутентификации пользователя. А второй, чтобы обновлять access token, так как они живут не долго. Для аутентификации пользователя в каждом заголовке запроса от него к серверу прописывается специальная строчка, например:
Authorization: Bearer 401f7ac837da42b97f613d789819ff93537bee6a

Сервер читает заголовок запроса, находит запись с access токеном, сверяет его по своей БД и если находит, то пользователь считается авторизованным.[5]

Для того, чтобы подтвердить почту при аутентификации, сделаем отправку писем с помощью SMTP-сервера. Для этого был выбран сервис Yandex Почта. Чтобы наш сервер отправлял письма зарегистрированному пользователю, необходимо прописать следующие настройки, которые указаны на рисунке 3.2.2.1:

```
DJOSER = {
    'SEND_ACTIVATION_EMAIL': True,
    'SEND_CONFIRMATION_EMAIL': True,
    'ACTIVATION_URL': 'activate/{uid}/{token}',
    'PASSWORD_RESET_SHOW_EMAIL_NOT_FOUND': True,
    'PASSWORD_RESET_CONFIRM_URL': '/password/reset/confirm/{uid}/{token}',
    'TOKEN_MODEL': None,
    'SERIALIZERS': {},
}

#SMTP
EMAIL_HOST = os.environ['EMAIL_HOST']
EMAIL_PORT = os.environ['EMAIL_PORT']
EMAIL_USE_TLS = False
EMAIL_USE_SSL = True
EMAIL_HOST_USER = os.environ['EMAIL_HOST_USER']
EMAIL_HOST_PASSWORD = os.environ['EMAIL_HOST_PASSWORD']
SERVER_EMAIL = EMAIL_HOST_USER
DEFAULT_FROM_EMAIL = EMAIL_HOST_USER
SITE_NAME = "WERTH"
DOMAIN = "127.0.0.1:8080"
```

*Рисунок 3.2.2.1 Настройка SMTP в серверной части*

Проверим работоспособность авторизации с помощью программы Postman.

Протестируем работу регистрации с помощью программы Postman. Для того, чтобы зарегистрироваться, необходимо отправить POST запрос на url *http://127.0.0.1:8000/auth/user/*

Регистрация так же поддерживает валидацию логина, пароля, e-mail и проверяет уникальность e-mail. Работоспособность можем увидеть на рисунках 3.2.2.2 и 3.2.2.3 :

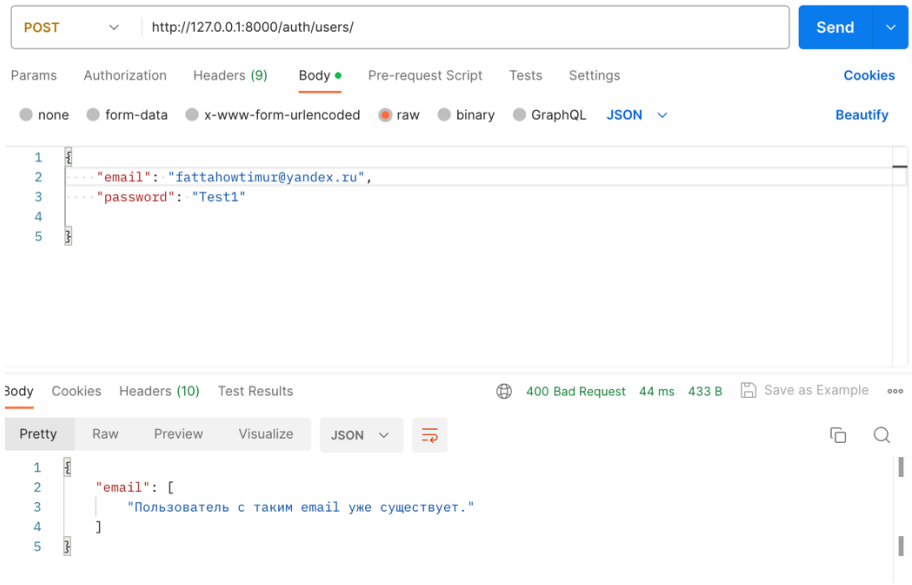


*Рисунок 3.2.2.2 Проверка email на уникальность*

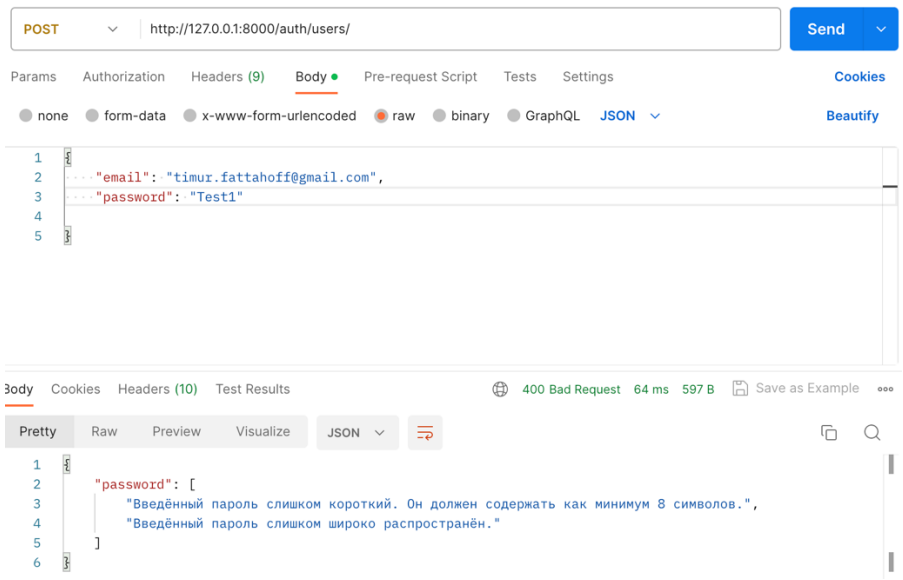


*Рисунок 3.2.2.3 Валидация пароля*

После ввода валидных данных аккаунт успешно проходит авторизацию, и данные пользователя записываются в базу данных и на e-mail отправляется письмо. Ниже, на рисунке 3.2.2.4, видим, как пользователь авторизовался.

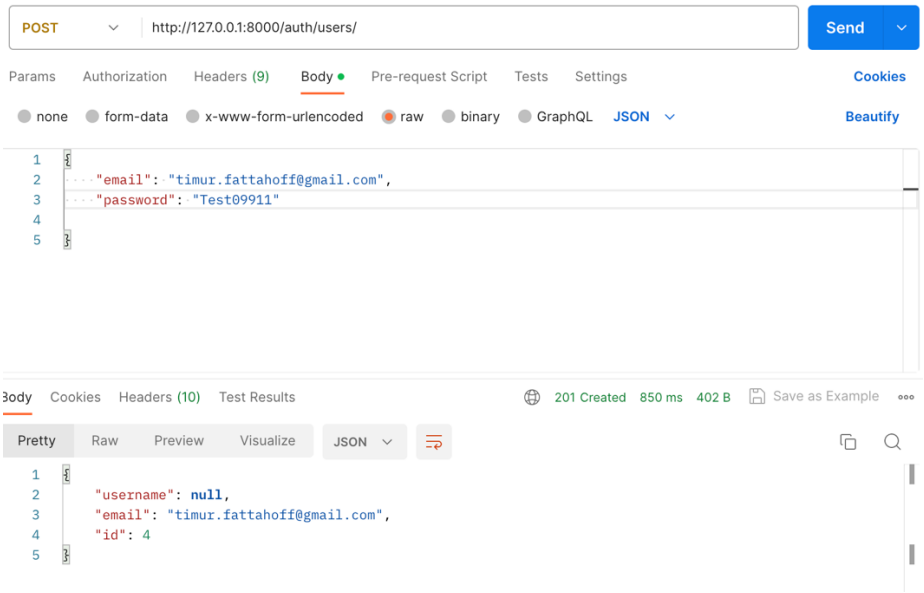


*Рисунок 3.2.2.4 Регистрация пользователя*

На рисунке 3.2.2.5 можем видеть, что пользователь получил активационное письмо на свой e-mail адрес.

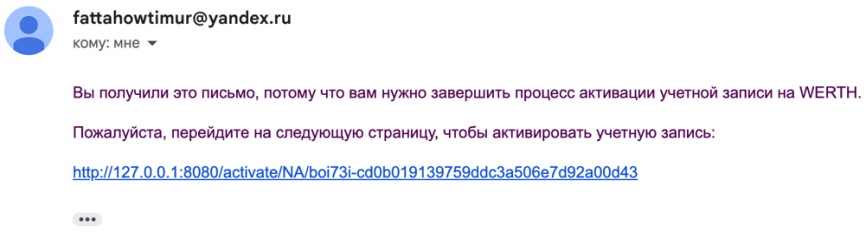


*Рисунок 3.2.2.5 Полученное письмо для активации аккаунта*

В случае, если не перейдем по ссылке и не активируем e-mail, пользователь не сможет пройти аутентификацию. Получим ошибку, который изображен на рисунке 3.2.2.6:

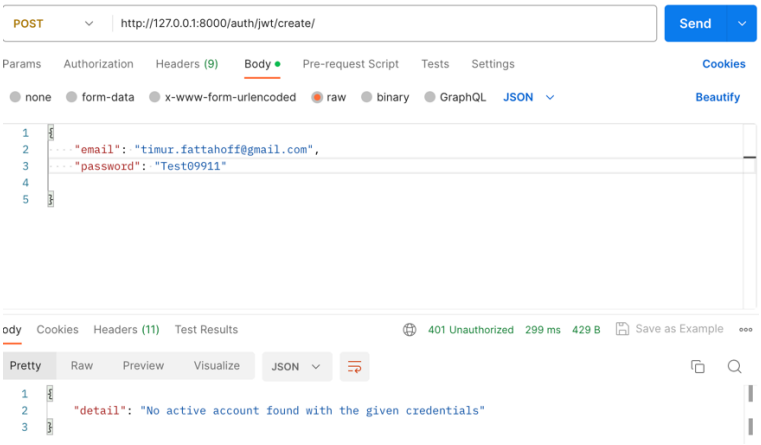


*Рисунок 3.2.2.6 Ошибка аутентификации при не активированном e-mail*

При авторизации на активированном e-mail получим два токена. Их можем увидеть на рисунке 3.2.2.7:

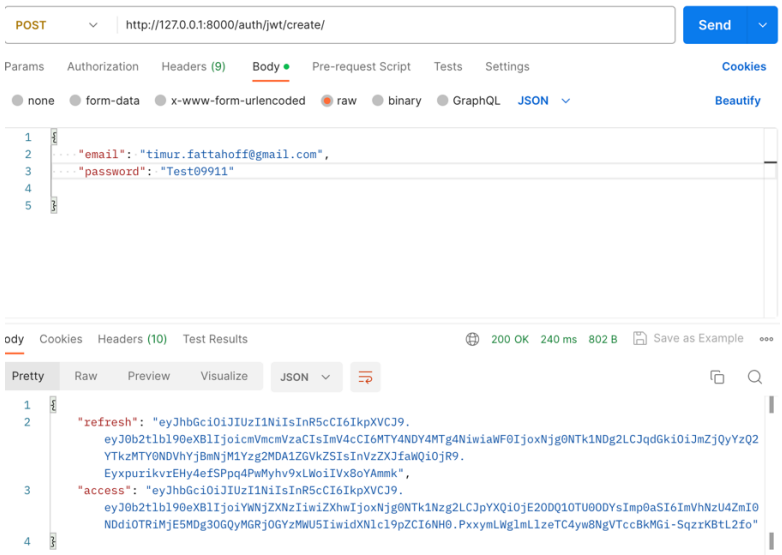


*Рисунок 3.2.2.7 Авторизация пользователя*

### 3.2.2 Функционал сайта

Рассмотрим функционал сайта, который реализуем в серверной части проекта:

• Описание пользователя

• Задачи (создание, редактирование, получение, удалениеq)

Данные об этих сущностях будут храниться в базе данных postgreSQL. Функционал был поделен на два приложения – это users, tasks. Приложения в Django – это отдельные web-приложения, которые предоставляют определенный функционал. Для отображения и создания таблиц в базе данных, не используя sql-запросы, в Django используются модели.

Рассмотрим создание моделей:

Класс CustomUser описывает сущность пользователя. В таблице будет храниться информация по всем пользователям(email, фамилия, имя, роль, фотография).

```
class CustomUser(AbstractUser):
    ADMIN = 1
    SELLER = 2
    DEVELOP = 3
    EMPLOYEE = 4
    ROLES = (
        (ADMIN, 'Admin'),
        (SELLER, 'Seller'),
        (DEVELOP, 'Developer'),
        (EMPLOYEE, 'Employee'),
    )
    class Meta:
        verbose_name = 'Пользователь'
        verbose_name_plural = 'Пользователи'
    username = None
    email = models.EmailField(unique=True)
    first_name = models.CharField(max_length=30, blank=True)
    last_name = models.CharField(max_length=40, blank=True)
    role = models.PositiveSmallIntegerField(choices=ROLES, blank=True,
null=True, default=4)
    avatar = models.ImageField(
        upload_to=get_path_avatar,
        blank=True,
        null=True,
        validators=[FileExtensionValidator(allowed_extensions=['jpg']),
validate_size_image]
    )
    USERNAME_FIELD = "email"
    REQUIRED_FIELDS = ['username']
    objects = CustomUserManager()
    def __str__(self):
        return self.email
```

Класс Task описывает сущность задачи. В таблице будет храниться информация по всем задачам(тема, описание, исполнители, количество исполнителей для конкретной задачи, описание, дедлайн задачи, статус, пометки, а так же файл, прикрепленный к задаче).

```
class Task(models.Model):
    WAITING = 1
    ATWORK = 2
    ONINSPECTION = 3
    ACCEPTED = 4
    STATUS = (
        (WAITING, 'Ожидает исполнителя'),
        (ATWORK, 'В работе'),
        (ONINSPECTION, 'На проверке'),
        (ACCEPTED, 'Принят'),
    )

    task_topic = models.CharField(max_length=50)
    task_executors = models.ManyToManyField(CustomUser, default=None)
    task_number_of_performing = models.IntegerField()
    task_count_of_performing = models.IntegerField(default=0)
    task_description = models.CharField(max_length=255)
    task_deadline = models.DateTimeField()
    task_status = models.PositiveSmallIntegerField(choices=STATUS, blank=True,
null=True, default=1)
    task_flag = models.CharField(max_length=50)
    task_file = models.FileField(upload_to= get_path_file, blank=True,
null=True)

    class Meta:
        verbose_name = 'Задача'
        verbose_name_plural = 'Задачи'
```

Каждый класс представляет собой одну таблицу в базе данных. Внутри класса объявляются переменные – это названия колонок в таблице.

Базовым функционалом приложения будут являться CRUD операции над моделями данных. CRUD – расшифровывается как create, read, update, delete, что обозначает основные функции при работе с базой данных – создание, чтение, обновление, удаление сущностей.

API. Бэкенд приложения представляет собой WebAPI – способ построения приложения, работающий в стиле REST – архитектуры, использующейся для взаимодействия с сервером через HTTP запросы: Get, Post, Put, Delete. Для того, чтобы обрабатывать запросы, необходимо преобразовать данные из БД в JSON формат. Это реализуется в python файле serializers.py. Сериализаторы также обеспечивают десериализацию, позволяя

преобразовать обработанные данные обратно в сложные типы после предварительной проверки входящих данных.

Пример сериализатора для создания задачи изображен на рисунке 3.2.3.1.

```
class CreateTaskSerializer(serializers.ModelSerializer):
    timurfattahov
    class Meta:
        model = Task
        fields = (
            "task_topic",
            "task_count_of_performing",
            "task_description",
            "task_deadline",
            "task_status",
            "task_file",
        )
```

*Рисунок 3.2.3.1 Листинг сериализатора*

Рассмотрим на примере создания задачи реализацию запросов CRUD, для остальных сущностей применяется такой же принцип.

Метод создания задачи можем увидеть на рисунке 3.2.3.2:

```
timurfattahov
@api_view(['POST'])
@permission_classes([IsAuthenticated])
def CreateTaskView(request):
    serializer = CreateTaskSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data)
    return Response(status=status.HTTP_400_BAD_REQUEST)
```

*Рисунок 3.2.3.2 Листинг метода создания задачи*

Входной параметр преобразовывается в JSON формат, проверяется на валидность и сохраняется в таблице базы данных. Если все хорошо, функция возвращает созданный объект, иначе ошибку HTTP_400_BAD_REQUEST.

С помощью программы Postman обработаем этот запрос и проверим его на работоспособность.

Результат POST запроса для создания задачи можем увидеть на рисунке 3.2.3.3:

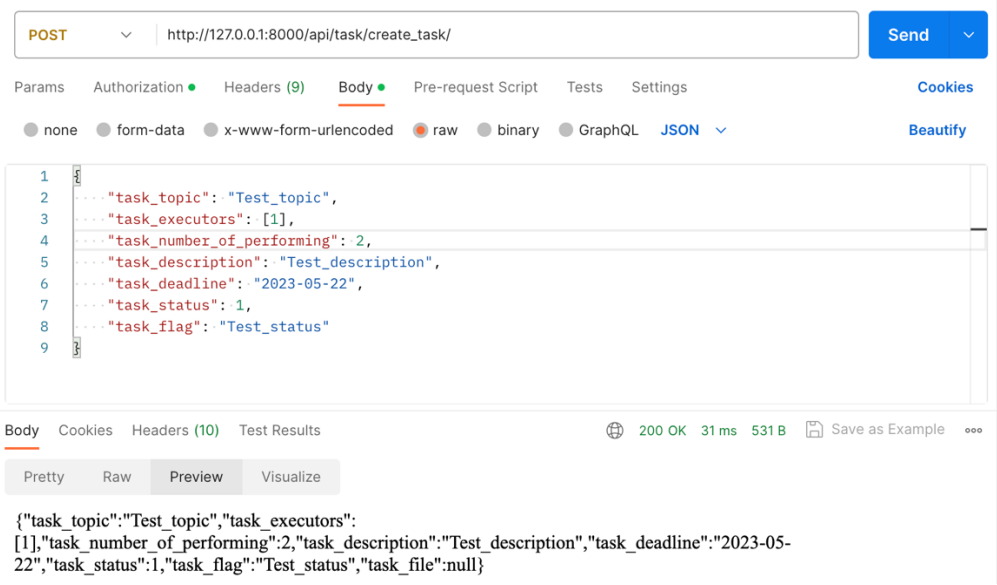


*Рисунок 3.2.3.3 Результат POST запроса*

## 3.3 Реализация клиентской части проекта

### 3.3.1 Создание приложения

Для того, чтобы создать проект на Vue, необходимо установить Vue CLI. Vue CLI – это интерфейс командной строки. Чтобы его установить, необходимо ввести в терминал следующую команду:

```
npm install -g vue-cli
```

После установки Vue CLI можем приступить к созданию проекта. Для этого в консоли переходим в директорию нашего проекта и запускаем команду:

```
vue create management_system_vue
```

Проект создан, можем приступить к написанию кода для клиентской части.

### 3.3.2 Разработка пользовательского интерфейса

Для реализации пользовательского интерфейса были использованы следующие технологии:

1) HTML
2) CSS
3) JavaScript
4) Vue.js
5) Vuetify

Страницы сайта на Vue.js создаются с помощью Vue компонентов. Данная компонента состоит из трех блоков. Первый блок – template. Здесь расположен весь наш HTML страницы. Второй – это script. В данном блоке расположены все наши методы, объекты и реализации запросов на языке JavaScript. И третий блок – это style. Все наши стили пропишем именно в данном блоке.

Так как наши страницы будут содержать много одинакового HTML кода, например заголовок и футер, создадим для них отдельные компоненты в директории components. И при необходимости просто вызывать в блоке template нужную компоненту.

Рассмотрим, как выглядит компонента Vue, с помощью рисунка 3.3.2.1:

```
<template>
<navbar></navbar>

</template>

<script>
import Navbar from "@/components/Navbar";
export default {
name: "TestComponent",
  components: {Navbar}
}
</script>

<style scoped>

</style>
```

*Рисунок 3.3.2.1 Пустая компонента Vue.js*

Для того, чтобы перемещаться из одной страницы в другую, нам необходимо проложить маршруты между ними, как указано на рисунке 3.3.2.2. Делается это в директории router в файле main.js и выглядит следующим образом:

```
import { createRouter, createWebHistory } from 'vue-router'
import Company from "@/views/Company.vue";
import PageNotFound from "@/views/PageNotFound.vue";
import Registration from "@/views/Registration.vue";
import EmailVerifPage from "@/views/EmailVerifPage.vue";
import Activate from "@/views/Activate.vue";
import LogInPage from "@/views/LogInPage.vue";
import FillUserInfo from "@/views/FillUserInfo.vue";
const routes = [
  {path: '/', name: 'home', component: Company},
  {path: '/:pathMatch(.*)*', name: 'Error', component: PageNotFound},
  {path: '/registration', name: 'registration', component: Registration},
  {path: '/activate/:uid/:token', name: 'activate', component: Activate},
  {path: '/email_send', name: 'email_send', component: EmailVerifPage},
  {path: '/log_in', name: 'LogInPage', component: LogInPage},
  {path: '/fill_user_info', name: 'FillUserInfo', component: FillUserInfo}
]

const router = createRouter( options: {
  history: createWebHistory(process.env.BASE_URL),
  routes
})
```

*Рисунок 3.3.2.2 Листинг маршрутов сайта*

Рассмотрим подробнее структуру компонентов на примере одной страницы, для остальных страниц применяется такой же принцип.

Ниже можем увидеть листинг блока template, реализованный на HTML5 и стилей vuetify:

```
<template>
<NavBar></NavBar>
  <div class="main">
    <v-card
        class="mx-auto"
        max-width="344"
        title="Заполните информацию о себе"
        color=""
    >
      <v-container
          style="align-items: center;"
      >
        <v-text-field
            v-model="first_name"
            color="black"
            label="Имя"
            variant="underlined"
        ></v-text-field>

        <v-text-field
            v-model="last_name"
```

```
            color="black"
            label="Фамилия"
            variant="underlined"
          ></v-text-field>

          <v-file-input
            :rules="rules"
            color="black"
            accept="image/png, image/jpeg, image/bmp"
            placeholder="Загрузите аватар"
            prepend-icon="mdi-camera"
            label="Аватар"
            variant="underlined"
          ></v-file-input>

        </v-container>
        <v-list lines="one">
          <v-list-item
            v-for="(values, name) in errors"
            :key="name"
            :title="values"
          ></v-list-item>
        </v-list>
        <v-divider></v-divider>
        <v-card-actions>
          <v-spacer></v-spacer>
          <v-btn type="submit" color="dark" @click="submitform">
            Сохранить
            <v-icon icon="mdi-chevron-right" end></v-icon>
          </v-btn>
        </v-card-actions>
      </v-card>
    </div>
  </template>
```

Обработка CRUD запросов из серверной части был реализован в блоке script с помощью axios. Axios – это HTTP-клиент, основанный на промисах и предназначенный для браузеров и для Node.js. С его помощью сможем реализовать HTTP запросы. Рассмотрим пример запроса на стороне клиента, который изображен на рисунке 3.3.2.3.

```
await axios
    .get( url: 'auth/users/me')
    .then(response => {
        user_id  = response.data.id
        sessionStorage.setItem('user_id', response.data.id)
    })

await axios
    .get( url: 'api/user/info/' + user_id + '/')
    .then(response => {
        first_name = response.data.first_name
        last_name = response.data.last_name
        role = response.data.role
    })
```

*Рисунок 3.3.2.3 GET запрос на стороне клиента*

В данном методе реализовано два get запроса. Первый запрос получает из сервера id пользователя. Второй запрос с помощью id пользователя получает Фамилию, Имя и роль пользователя. Рассмотрим так же запросы post и delete. Запрос на обновление будет работать по такому же принципу как post запрос, только url- адрес будет содержать еще идентификационный параметр объекта, который хотим обновить.

На рисунке 3.3.2.4 изображен фрагмент кода, который реализует post запрос:

```
submitform(){
  axios.defaults.headers.common['Authorization'] = 'Bearer ' + localStorage.getItem( key: 'JWT')
  axios
      .post( url: 'api/user/fillinfo/'+ this.$store.getters.user_id +'/',  data: {
        first_name: this.first_name,
        last_name: this.last_name,
      })
      .then(response => {
        const payload = {'first_name': this.first_name, 'last_name': this.last_name}
        this.$store.commit('getuserfullname', payload)
        this.$router.push('/')
      })
      .catch(error => {
        console.log(error)
        this.errors = error.data
      })
}
```

*Рисунок 3.3.2.4 POST запрос на стороне клиента*

Данный метод реализует заполнение информации о пользователе в базе данных, а так же хранение этой информации в vuex. Для того, чтобы пользователь смог отправить запрос, он должен быть авторизованным. Чтобы сервер смог понять, что пользователь авторизован, необходимо в заголовок Authorization передать токен пользователя. Запрос принимает на вход два параметра. Первый – это url адрес запроса в серверной части. Второй – данные, которые необходимо передать в базу данных. В случае, если post запрос обработается, данные сохранятся во vuex и сайт перенаправит пользователя на главную страницу. Vuex - паттерн управления состоянием.[6]

# Заключение

В данной работе была разработана серверная и клиентская части веб – приложения для сайта разработки автоматизированной системы управления проектами для малого бизнеса.

Для выполнения поставленной цели была проделана следующая работа

1. Определены требования к реализации поставленной задачи
2. Составлено техническое задание
3. Выбраны и изучены технологии для разработки
4. Реализована авторизация с помощью JWT токенов
5. Реализован функционал с помощью CRUD операциями над моделями данных

С помощью данного веб-сервиса компании смогут автоматизировать свои процессы. После создания задачи разработчики сами выбирают задачи, которые они будут выполнять. Руководители смогут следить за своими сотрудниками, проверять их выполненные работы и вносить туда правки.

Разработка такой системы способствует увеличению конкурентоспособности малых бизнесов и повышению качества услуг и продуктов. Проект является перспективным, так как современные технологии и методы разработки обеспечивают возможность дальнейшего улучшения и развития системы.

Таким образом, были проделаны все запланированные на производственную практику работы. В таблице 1. указаны следующие приобретенные компетенции:

Таблица 1.

| Компетенция и расшифровка | Освоенные навыки |
|---|---|
| ПК-1<br>Способен осуществлять проведение работ по обработке, анализу научно-технической информации и результатов исследований | Были изучены основы разработки REST приложений на Python и Vue.js |

| | |
|---|---|
| ПК-2<br>Способен осуществлять выполнение экспериментов и оформлять результаты исследований и разработок | Были выявлены основные потребности в разработке сайта, сформирована структура проекта. |
| ПК-3<br>Способен владеть методами математической физики и прикладной математики | Спроектирована база данных в СУБД PostgreSQL. |
| ПК-4<br>Способен применять методы и принципы построения физических и математических моделей, их применимости к конкретным процессам и явлениям | Созданы представления главной страницы сайта и страницы регистрации и авторизации. |
| ПК-5<br>Способен проводить математическое и компьютерное моделирование, в соответствии с исходными данными задач исследуемой области | Реализована система регистрации и авторизации на back-end с помощью JWT токенов. Реализован процесс создания, редактирования, чтения и удаления задач на back-end. Протестирован функционал сайта. |

# Список использованной литературы


1. Python documentation. [Электронный ресурс]. - Режим доступа: https://docs.python.org/3/index.html (дата обращения: 05.04.2023).
2. Django documentation. [Электронный ресурс]. - Режим доступа: https://docs.djangoproject.com/en/4.1/ (дата обращения: 05.04.2023).
3. Django Rest Framework documentation. [Электронный ресурс]. - Режим доступа: https://www.django-rest-framework.org/topics/documenting-your-api/ (дата обращения: 07.04.2023).
4. Документация Vue.js [Электронный ресурс]. - Режим доступа: https://vueframework.com/docs/v3/ru/ru/guide/introduction.html (дата обращения: 15.04.2023).
5. Djoser documentation. [Электронный ресурс]. - Режим доступа: https://djoser.readthedocs.io/en/latest/ (дата обращения: 07.05.2022).
6. Форум разработчиков: Освоение Vuex — с нуля до героя. [Электронный ресурс]. - Режим доступа: https://habr.com/ru/post/421551/ (дата обращения: 12.05.2022).

# Приложение

## Листинг серверной части проекта

### models.py

```
from django.db import models
from users.models import CustomUser
from base.services import get_path_file


class Task(models.Model):
    WAITING = 1
    ATWORK = 2
    ONINSPECTION = 3
    ACCEPTED = 4
    STATUS = (
        (WAITING, 'Ожидает исполнителя'),
        (ATWORK, 'В работе'),
        (ONINSPECTION, 'На проверке'),
        (ACCEPTED, 'Принят'),
    )

    task_topic = models.CharField(max_length=50)
    task_executors = models.ManyToManyField(CustomUser, default=None)
    task_number_of_performing = models.IntegerField()
    task_count_of_performing = models.IntegerField(default=0)
    task_description = models.CharField(max_length=255)
    task_deadline = models.DateField()
    task_status = models.PositiveSmallIntegerField(choices=STATUS,
blank=True, null=True, default=1)
    task_flag = models.CharField(max_length=50)
    task_file = models.FileField(upload_to= get_path_file, blank=True,
null=True)

    class Meta:
        verbose_name = 'Задача'
        verbose_name_plural = 'Задачи'


class CustomUser(AbstractUser):

    ADMIN = 1
    SELLER = 2
    DEVELOP = 3
    EMPLOYEE = 4
    ROLES = (
        (ADMIN, 'Admin'),
        (SELLER, 'Seller'),
        (DEVELOP, 'Developer'),
        (EMPLOYEE, 'Employee'),
    )

    class Meta:
        verbose_name = 'Пользователь'
        verbose_name_plural = 'Пользователи'
    username = None
    email = models.EmailField(unique=True)
    first_name = models.CharField(max_length=30, blank=True)
    last_name = models.CharField(max_length=40, blank=True)
    role = models.PositiveSmallIntegerField(choices=ROLES, blank=True,
```

```
null=True, default=4)
    avatar = models.ImageField(
        upload_to=get_path_avatar,
        blank=True,
        null=True,
        validators=[FileExtensionValidator(allowed_extensions=['jpg']),
validate_size_image]
    )

    USERNAME_FIELD = "email"
    REQUIRED_FIELDS = ['username']

    objects = CustomUserManager()

    def __str__(self):
        return self.email
```

## serializers.py

```
from rest_framework import serializers
from .models import CustomUser


class UserInfoSerializer(serializers.ModelSerializer):
    class Meta:
        model = CustomUser
        fields = [
            "first_name",
            "last_name",
            "role"
        ]


class UserFillInfoSerializer(serializers.ModelSerializer):
    class Meta:
        model = CustomUser
        fields = [
            "first_name",
            "last_name",
        ]

class CreateTaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = (
            "task_topic",
            "task_executors",
            "task_number_of_performing",
            "task_description",
            "task_deadline",
            "task_status",
            "task_flag",
            "task_file",
        )

class GetTaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = (
            "id",
            "task_topic",
            "task_count_of_performing",
            "task_description",
```

```
            "task_deadline",
            "task_status",
            "task_executors",
            "task_flag",
            "task_file",
        )
```

views.py

```
from django.shortcuts import render
from rest_framework import status
from rest_framework.decorators import api_view, permission_classes
from rest_framework.permissions import IsAuthenticated, AllowAny
from rest_framework.response import Response

from .models import Task
from .serializers import CreateTaskSerializer, GetTaskSerializer


@api_view(['POST'])
@permission_classes([AllowAny])
def CreateTaskView(request):
    serializer = CreateTaskSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data)
    return Response(status=status.HTTP_400_BAD_REQUEST)


@api_view(['GET'])
@permission_classes([AllowAny])
def GetAllTasksView(request):
    item = Task.objects.all()
    serializer = GetTaskSerializer(item, many=True)
    return Response(serializer.data)


@api_view(['GET'])
@permission_classes([AllowAny])
def GetTaskView(request, task_id):
    item = Task.objects.all.filter(id=task_id)
    serializer = GetTaskSerializer(item, many=False)
    return Response(serializer.data)


@api_view(['POST'])
@permission_classes([IsAuthenticated])
def PutTaskView(request, task_id):
    item = Task.objects.get(id=task_id)
    serializer = CreateTaskSerializer(instance=item, data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data)
    return Response(status=status.HTTP_400_BAD_REQUEST)


@api_view(['DELETE'])
@permission_classes([IsAuthenticated])
def DeleteTaskView(request, task_id):
    try:
        item = Task.objects.get(id=task_id)
    except Task.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)
```

```
    if request.method == "DELETE":
        item.delete()
    return Response(status=status.HTTP_204_NO_CONTENT)


@api_view(['GET'])
@permission_classes([IsAuthenticated])
def userinfo(request, user_id):
    item = CustomUser.objects.get(id=user_id)
    serializer = UserInfoSerializer(item, many=False)
    return Response(serializer.data)

@api_view(['POST'])
@permission_classes([IsAuthenticated])
def filluserinfo(request, user_id):
    item = CustomUser.objects.get(id=user_id)
    serializer = UserFillInfoSerializer(instance=item, data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data)
    return Response(status=status.HTTP_400_BAD_REQUEST)
```

## Листинг клиентской части проекта

### Компонента Navbar.vue

```
<template>
 <div>
  <v-toolbar
    dark
    prominent
    color="white"
  >
   <v-toolbar-title>Werth</v-toolbar-title>
   <v-spacer></v-spacer>
   <v-btn variant="outlined" v-if="!isAuthenticated">
    <router-link to="/log_in" style="color: black; text-decoration: none"> Вход </router-link>
   </v-btn>
   <v-btn icon v-if="isAuthenticated" @click="LogOut()">
    <v-icon>mdi-export</v-icon>
   </v-btn>
  </v-toolbar>
  <v-divider></v-divider>
 </div>
</template>

<script>
import {mapGetters} from "vuex";

export default {
 name: "NavBar",
 data(){
  return{
   // drawer: false,
   // links: [
   //   {text: 'Мои задачи', route: '/'},
   //   {text: 'Все задачи', route: '/'},
   //   {text: 'Проекты', route: '/'},
   // ],
```

```
    // chats:[
    //   {chat: 'Менеджеры', route: '/'},
    //   {chat: 'Продажники', route: '/'},
    // ]
   }
 },
 mounted() {
  this.$store.commit('initializeStore')
 },
 methods:{
  LogOut(){
   this.$store.commit('removeToken')
   this.$router.push('/log_in')
  },
 },
 computed:{
  ...mapGetters({
   isAuthenticated: "isAuthenticated",
   email: "email",
  }),
 },
}
</script>
<style scoped>
</style>
```

## Компонента SideBar.vue

```
<template>
 <v-navigation-drawer
    expand-on-hover
    rail
 >
  <v-list>
   <v-list-item
      prepend-avatar="https://cdn.vuetifyjs.com/images/john.png"
      :title="first_name+' '+last_name"
      subtitle="role"
   ></v-list-item>
  </v-list>
  <v-divider></v-divider>
  <v-list density="compact" nav>
   <v-list-item prepend-icon="mdi-folder" title="My Files" value="myfiles"></v-list-item>
   <v-list-item prepend-icon="mdi-account-multiple" title="Shared with me" value="shared"></v-list-item>
   <v-list-item prepend-icon="mdi-star" title="Starred" value="starred"></v-list-item>
  </v-list>
 </v-navigation-drawer>
 <v-main style="height: 250px"></v-main>
</template>

<script>
import axios from "axios";
import {th} from "vuetify/locale";
import {mapGetters} from "vuex";

export default {
 name: "SideBar",
 computed: mapGetters(['first_name','last_name', 'role'])
}
</script>
<style scoped>
</style>
```

## Компонента Company.vue

```
<template>
  <NavBar></NavBar>
  <SideBar></SideBar>
</template>

<script>
import NavBar from "@/components/NavBar.vue";
import SideBar from "@/components/SideBar.vue";
import {mapActions, mapGetters, mapState} from "vuex";
import axios from "axios";
import {th, tr} from "vuetify/locale";
export default {
  name: "Company",
  components: {SideBar, NavBar},
  data(){
    return{
      errors: {},
      isAuthenticated: this.$store.state.isAuthenticated,
    }
  },
  computed: {
    ...mapGetters(['first_name', 'last_name'])
  },
  async mounted() {
    await this.user_id_func()
    if(this.$store.getters.first_name === "" && this.$store.getters.last_name === "" &&
this.$store.getters.isAuthenticated === true){
      console.log('OK')
      this.$router.push("/fill_user_info")
    }
    else {
      console.log("No OK")
      console.log(this.$store.state.email)
      console.log(this.$store.state.JWT)
      console.log(this.isAuthenticated)
    }
  },
  methods: mapActions(['user_id_func']),
}
</script>

<style scoped>
</style>
```

## Компонента FillUserInfo.vue

```
<template>
  <NavBar></NavBar>
  <div class="main">
    <v-card
        class="mx-auto"
        max-width="344"
        title="Заполните информацию о себе"
        color=""
    >
      <v-container
          style="align-items: center;"
```

```
    >
      <v-text-field
        v-model="first_name"
        color="black"
        label="Имя"
        variant="underlined"
      ></v-text-field>

      <v-text-field
        v-model="last_name"
        color="black"
        label="Фамилия"
        variant="underlined"
      ></v-text-field>

      <v-file-input
        :rules="rules"
        color="black"
        accept="image/png, image/jpeg, image/bmp"
        placeholder="Загрузите аватар"
        prepend-icon="mdi-camera"
        label="Аватар"
        variant="underlined"
      ></v-file-input>

    </v-container>
    <v-list lines="one">
      <v-list-item
        v-for="(values, name) in errors"
        :key="name"
        :title="values"
      ></v-list-item>
    </v-list>
    <v-divider></v-divider>
    <v-card-actions>
      <v-spacer></v-spacer>
      <v-btn type="submit" color="dark" @click="submitform">
        Сохранить
        <v-icon icon="mdi-chevron-right" end></v-icon>
      </v-btn>
    </v-card-actions>
   </v-card>
  </div>
</template>

<script>
import NavBar from "@/components/NavBar.vue";
import axios from "axios";

export default {
  name: "FillUserInfo",
  components:{NavBar},
  data(){
    return{
      first_name: '',
```

```
    last_name: '',
    rules: '',
    errors: {},
   }
  },
  mounted() {
  },
  methods:{
   submitform(){
    axios.defaults.headers.common['Authorization'] = 'Bearer ' + localStorage.getItem('JWT')
    axios
      .post('api/user/fillinfo/'+ this.$store.getters.user_id +'/', {
       first_name: this.first_name,
       last_name: this.last_name,
      })
      .then(response => {
       const payload = {'first_name': this.first_name, 'last_name': this.last_name}
       this.$store.commit('getuserfullname', payload)
       this.$router.push('/')
      })
      .catch(error => {
       console.log(error)
       this.errors = error.data
      })
   }
  }
}
</script>

<style scoped>
.main{
  height: calc(100% - 70px);
  padding-top: 15vh;
}
</style>
```

## Компонента LogInPage.vue

```
<template>
  <NavBar></NavBar>
  <div class="main">
    <v-card
      class="mx-auto"
      max-width="344"
      title="Авторизация"
      color=""
    >
    <v-container
      style="align-items: center;"
    >
     <v-text-field
      type="email"
      v-model="email"
      color="black"
      label="Email"
```

```
          variant="underlined"
        ></v-text-field>
        <v-text-field
          type="password"
          v-model="password"
          color="black"
          label="Пароль"
          placeholder="Введите пароль"
          variant="underlined"
        ></v-text-field>
      </v-container>
      <v-list lines="one">
        <v-list-item
          v-for="(values, name) in errors"
          :key="name"
          :title="values"
        ></v-list-item>
      </v-list>
      <v-divider></v-divider>
        <v-card-actions>
          <v-spacer></v-spacer>
          <v-btn type="submit" color="dark" @click="SignForm">
            Войти
            <v-icon icon="mdi-chevron-right" end></v-icon>
          </v-btn>
        </v-card-actions>
      </v-card>
      <br>
      <div class="d-flex justify-center mb-6">
        <p class="mx-auto" >Не зарегистрированы? <router-link to="/registration" style="color: black; text-
decoration: none"> Зарегистрироваться </router-link></p>
      </div>
  </div>
</template>

<script>
import axios from "axios";
import {th} from "vuetify/locale";
import NavBar from "@/components/NavBar.vue";
export default {
  name: "LogInPage",
  components: {NavBar},
  data(){
    return{
      email: "",
      password: "",
      errors: {},
    }
  },
  mounted() {
    document.title = 'Вход | Werth'
  },
  methods:{
    SignForm(){
      axios.defaults.headers.common['Authorization'] = ""
```

```
    const formData = {
      email: this.email,
      password: this.password
    }
    axios
      .post("auth/jwt/create", formData)
      .then(response => {
        sessionStorage.setItem("email", this.email)
        localStorage.setItem('isAuthenticated', 'true')
        this.$store.commit('setToken', response.data.access)
        this.$store.commit('setRefreshToken', response.data.refresh)
        axios.defaults.headers.common['Authorization'] = "Bearer " + response.data.access
        this.$router.push("/")
      })
      .catch(error => {
        this.errors.push('Проверьте корректность введенных данных.')
      })
  }
 }
}
</script>

<style scoped>
.main{
 height: calc(100% - 70px);
 padding-top: 20vh;
}
</style>
```

## Компонента Registration.vue

```
<template>
 <NavBar></NavBar>
 <div class="main">
    <v-card
      class="mx-auto"
      max-width="344"
      title="Регистрация"
      color=""
    >
     <v-container
       style="align-items: center;"
     >
      <v-text-field
        v-model="email"
        color="black"
        label="Email"
        variant="underlined"
      ></v-text-field>
      <v-text-field
        type="password"
        v-model="password"
        color="black"
        label="Пароль"
```

```
            placeholder="Введите пароль"
            variant="underlined"
          ></v-text-field>
          <v-text-field
            type="password"
            v-model="password2"
            color="black"
            label="Подтвердите пароль"
            placeholder="Подтвердите пароль"
            variant="underlined"
          ></v-text-field>

        </v-container>
        <v-list lines="one">
          <v-list-item
            v-for="(values, name) in errors"
            :key="name"
            :title="values"
          ></v-list-item>
        </v-list>
        <v-divider></v-divider>
        <v-card-actions>
          <v-spacer></v-spacer>
          <v-btn type="submit" color="dark" @click="submitForm">
            Зарегистрироваться
            <v-icon icon="mdi-chevron-right" end></v-icon>
          </v-btn>
        </v-card-actions>
      </v-card>
      <br>
      <div class="d-flex justify-center mb-6">
        <p class="mx-auto" >Уже зарегистрированы? <router-link to="/log_in" style="color: black; text-decoration:
none"> Войти </router-link></p>
      </div>
    </div>
  </template>

  <script>
  import axios from 'axios'
  import NavBar from "@/components/NavBar.vue";
  import {th} from "vuetify/locale";
  export default {
    name: "Registration",
    components: {NavBar},
    data(){
      return{
        email: "",
        password: "",
        password2: "",
        errors: {},
      }
    },
    mounted() {
      document.title = 'Регистрация | Werth'
    },
```

```
  methods:{
    submitForm(){
      this.errors = []
      if (this.email ===")
        this.errors.push('Введите e-mail')
      if (this.password ===")
        this.errors.push('Введите пароль')
      if (this.password !== this.password2)
        this.errors.push('Пароли не совпадают')
      // if (this.rules === ")
      //   this.errors.push('Загрузите аватар')
      if (!this.errors.length) {
        axios
            .post('auth/users/', {
          email: this.email,
          password: this.password,
        })
            .then(response => {
              localStorage.setItem("email",this.email)
              this.$router.push('email_send')
            })
            .catch(error => {
              this.errors = error.response.data;
            })
      }
    }
  },
}
</script>

<style scoped>
.main{
  height: calc(100% - 70px);
  padding-top: 15vh;
}
</style>
```

## Файл index.js в директории store:

```
import axios from "axios";

export default {
    state:{
        JWT: '',
        JWT_Refresh: '',
        email: '',
        isAuthenticated: true,
        user_id: null,
        first_name: '',
        last_name: '',
        role: null,
    },
    actions:{
        async user_id_func(ctx){
            axios.defaults.headers.common['Authorization'] = "Bearer " + localStorage.getItem('JWT')
```

```
        let user_id = null
        let first_name = ''
        let last_name = ''
        let role = ''
        await axios
            .get('auth/users/me')
            .then(response => {
                user_id  = response.data.id
                sessionStorage.setItem('user_id', response.data.id)
            })

        await axios
            .get('api/user/info/' + user_id + '/')
            .then(response => {
                first_name = response.data.first_name
                last_name = response.data.last_name
                role = response.data.role
            })
        const payload = {'user_id': user_id, 'first_name': first_name, 'last_name':last_name, 'role': role}
        ctx.commit('getuserid', payload)
    },
},
mutations:{
    initializeStore(state){
        if (localStorage.getItem("JWT")){
            state.JWT = localStorage.getItem("JWT")
            state.isAuthenticated = true
            state.JWT_Refresh = sessionStorage.getItem('JWT_Refresh')
            state.email = sessionStorage.getItem('email')
        } else {
            state.JWT = ""
            state.isAuthenticated = false
            state.JWT_Refresh = ""
            state.email = ""
        }
    },
    setToken(state, JWT){
        state.JWT = JWT
        state.isAuthenticated = true
        localStorage.setItem("JWT", state.JWT)
    },
    setRefreshToken(state, JWT_Refresh){
        sessionStorage.setItem("JWT_Refresh", JWT_Refresh)
    },
    removeToken(state){
        state.JWT = ""
        state.JWT_Refresh = ""
        state.email = ""
        state.role = null
        state.first_name = ""
        state.last_name = ""
        state.user_id = null
        state.isAuthenticated = false
        localStorage.removeItem("JWT")
        sessionStorage.removeItem("JWT_Refresh")
        localStorage.removeItem("isAuthenticated")
    },
    getuserid(state, payload){
        state.user_id = payload.user_id
        state.first_name = payload.first_name
        state.last_name = payload.last_name
        state.role = payload.role
    },
```

```
        getuserfullname(state, payload){
            state.first_name = payload.first_name
            state.last_name = payload.last_name
        }
    },
    getters:{
        JWT: (state) => state.JWT,
        JWT_Refresh: (state) => state.JWT_Refresh,
        email: state => state.email,
        isAuthenticated: (state) => state.isAuthenticated,
        user_id: (state) => state.user_id,
        first_name: (state) => state.first_name,
        last_name: (state) => state.last_name,
        role: (state) => state.role,
    },
}
```