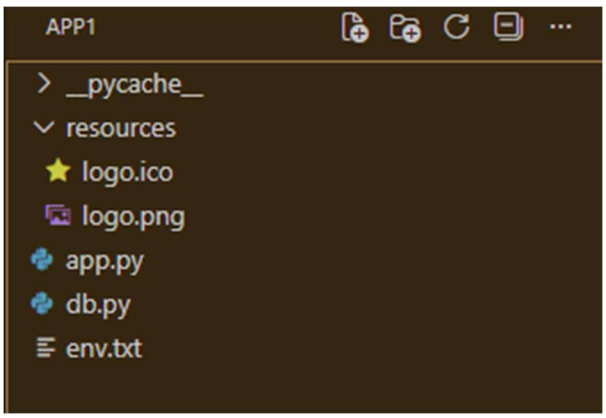
Отчёт по проделанной работе. Этап 2

1) Инициализация фреймворка для разработки бэкенд. В качестве основы я использовала фреймворк Streamlit на Python. Хотя он обычно относится к фронтенд-инструментам для data apps, в данном проекте Streamlit выступает как «тонкий сервер», который напрямую работает с базой данных через собственный модуль db.py. Логика работы с данными (чтение, добавление, обновление, удаление) вынесена в отдельный модуль, что позволяет рассматривать его как бэкенд-слой.

Я создала структуру проекта:

- app.py — основной модуль приложения
- db.py — модуль работы с базой данных
- resources/ — иконка и логотип компании
- .env — настройки подключения к PostgreSQL



2) Подключение библиотек

- **streamlit** — интерфейс и взаимодействие с пользователем.
- **pandas** — работа с табличными данными.
- **psycopg2 / sqlalchemy** — взаимодействие с PostgreSQL.
- **dotenv** — загрузка конфигурации БД из .env.
- **base64** — кодирование логотипа для вставки в хедер.

3) Подключение базы данных к бэкенду

```python
class DatabaseConnection:
    """
    класс для управления подключением к PostgreSQL через SQLAlchemy
    """

    def __init__(self):
        # данные бд
        self.host = os.getenv('DB_HOST', 'localhost')
        self.port = os.getenv('DB_PORT', '5432')
        self.database = os.getenv('DB_NAME', 'postgres')
        self.user = os.getenv('DB_USER', 'postgres')
        self.password = os.getenv('DB_PASSWORD', '0909')

        # строка для подключения SQLAlchemy (с использованием psycopg2)
        self.connection_string = (
            f"postgresql+psycopg2://{self.user}:{self.password}"
            f"@{self.host}:{self.port}/{self.database}"
        )
```

4) Выбор архитектуры REST или GraphQL

В рамках разработки подсистемы было принято решение не использовать отдельный REST- или GraphQL-сервер, поскольку приложение построено на фреймворке Streamlit, который позволяет напрямую взаимодействовать с модулем работы с базой данных (db.py) без промежуточного HTTP-API. Тем не менее, архитектурно модуль db.py реализует REST-подобный подход, где каждая функция соответствует типичному REST-эндпоинту

REST-операция	Аналог в проекте
GET /products	`get_products()`
GET /products/{id}	`get_product_by_id()`
POST /products	`add_product()`
PUT /products/{id}	`update_product()`
DELETE /products/{id}	`delete_product()`
GET /workshops	`get_workshops()`
GET /products/{name}/production-time	`get_production_time_for_product()`

5) Разработка модулей для передачи и мутации информации из базы данных

Для взаимодействия с PostgreSQL был разработан отдельный модуль db.py, содержащий функции для чтения, изменения и удаления данных. Модуль реализует полный набор операций CRUD (Create, Read, Update, Delete).

6) Создание отчёта

Отчёт успешно сформулирован и оформлен