🧪 Лабораторная работа №5

**«Слоистая архитектура: проектирование многоуровневой системы с PostgreSQL»**

**🔹 Цель работы**

1. Изучить принципы слоистой (layered) архитектуры в разработке программных систем.
2. Реализовать приложение с чётким разделением на Presentation → Business Logic → Data Access.
3. Использовать PostgreSQL как хранилище данных.
4. Применить инверсию зависимостей и интерфейсы для гибкости и тестируемости.
5. Оценить преимущества модульности при замене компонентов.

**📚 Теоретическая справка**

**Слоистая архитектура**

Система разбивается на горизонтальные слои, каждый из которых:

* Выполняет строго определённую функцию,
* Зависит только от нижележащих слоёв,
* Не знает о деталях реализации других слоёв.

**Типичные слои:**

|  |  |  |
| --- | --- | --- |
| Presentation | Обработка запросов, формирование ответа | Flask, FastAPI, CLI |
| Business Logic | Бизнес-правила, валидация, оркестрация | Сервисы, use cases |
| Data Access | Работа с БД, файлами, API внешних систем | Репозитории, ORM |

✅ Преимущества:

* Лёгкость тестирования (можно мокать репозиторий),
* Замена БД без изменения логики,
* Поддержка командной разработки.

**⚙️ Подготовка к работе**

**Требуемое ПО:**

* Python 3.9+
* Git
* PostgreSQL (локально или в облаке)
* (Опционально) VS Code / PyCharm

**Варианты PostgreSQL:**

1. Локально: установите [PostgreSQL](https://www.postgresql.org/download/)
2. Бесплатно в облаке:
   * [Render.com → PostgreSQL](https://render.com/docs/databases) (Free tier, без карты)
   * [Supabase](https://supabase.com/) (Free tier, PostgreSQL + Auth)

💡 Для лабораторной подойдёт любой вариант. Ниже — пример с локальной БД, но легко адаптируется под облако.

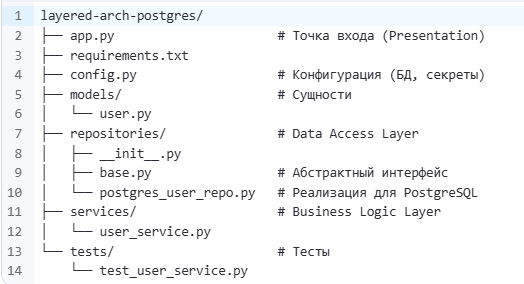
**🛠️ Задание 1: Проектирование структуры проекта**

**Цель**

Создать структуру проекта с чётким разделением слоёв.

**Шаги**

1. Создайте папку: layered-arch-postgres
2. Внутри создайте структуру:



✍️ Отчёт: диаграмма слоёв (текст или рисунок), описание обязанностей каждого слоя.

**🛠️ Задание 2: Настройка PostgreSQL и конфигурации**

**Шаги**

1. Установите PostgreSQL и создайте БД:

|  |
| --- |
| sql |
| CREATE DATABASE layered\_lab;  CREATE USER lab\_user WITH PASSWORD 'secure\_password';  GRANT ALL PRIVILEGES ON DATABASE layered\_lab TO lab\_user; |

1. Создайте таблицу:

|  |
| --- |
| sql |
| \c layered\_lab  CREATE TABLE users (  id SERIAL PRIMARY KEY,  name VARCHAR(100) NOT NULL,  email VARCHAR(100) UNIQUE NOT NULL,  created\_at TIMESTAMP DEFAULT NOW()  ); |

1. Создайте файл config.py:

|  |
| --- |
| python |
| import os  # Для локальной разработки  DB\_HOST = os.getenv("DB\_HOST", "localhost")  DB\_PORT = os.getenv("DB\_PORT", "5432")  DB\_NAME = os.getenv("DB\_NAME", "layered\_lab")  DB\_USER = os.getenv("DB\_USER", "lab\_user")  DB\_PASSWORD = os.getenv("DB\_PASSWORD", "secure\_password")  DATABASE\_URL = f"postgresql://{DB\_USER}:{DB\_PASSWORD}@{DB\_HOST}:{DB\_PORT}/{DB\_NAME}" |

💡 В облаке (например, Render) вы просто укажете DATABASE\_URL как переменную окружения.

**🛠️ Задание 3: Реализация Data Access Layer**

**Шаги**

1. Установите зависимости

|  |
| --- |
| Командная строка |
| pip install psycopg2-binary |

1. Файл models/user.py

|  |
| --- |
| python |
| from dataclasses import dataclass  from datetime import datetime  @dataclass  class User:  id: int  name: str  email: str  created\_at: datetime |

1. Файл repositories/base.py (абстрактный интерфейс):

|  |
| --- |
| python |
| from abc import ABC, abstractmethod  from typing import List, Optional  from models.user import User  class UserRepository(ABC):  @abstractmethod  def create\_user(self, name: str, email: str) -> User:  pass  @abstractmethod  def get\_user\_by\_email(self, email: str) -> Optional[User]:  pass  @abstractmethod  def get\_all\_users(self) -> List[User]:  pass |

1. Файл repositories/postgres\_user\_repo.py:

|  |
| --- |
| python |
| import psycopg2  from psycopg2.extras import RealDictCursor  from typing import List, Optional  from models.user import User  from repositories.base import UserRepository  from config import DATABASE\_URL  class PostgresUserRepository(UserRepository):  def \_\_init\_\_(self):  self.connection\_params = DATABASE\_URL  def \_get\_connection(self):  return psycopg2.connect(self.connection\_params)  def create\_user(self, name: str, email: str) -> User:  with self.\_get\_connection() as conn:  with conn.cursor(cursor\_factory=RealDictCursor) as cur:  cur.execute(  "INSERT INTO users (name, email) VALUES (%s, %s) RETURNING id, created\_at",  (name, email)  )  result = cur.fetchone()  return User(id=result['id'], name=name, email=email, created\_at=result['created\_at'])  def get\_user\_by\_email(self, email: str) -> Optional[User]:  with self.\_get\_connection() as conn:  with conn.cursor(cursor\_factory=RealDictCursor) as cur:  cur.execute("SELECT \* FROM users WHERE email = %s", (email,))  row = cur.fetchone()  if row:  return User(\*\*row)  return None  def get\_all\_users(self) -> List[User]:  with self.\_get\_connection() as conn:  with conn.cursor(cursor\_factory=RealDictCursor) as cur:  cur.execute("SELECT \* FROM users")  rows = cur.fetchall()  return [User(\*\*row) for row in rows] |

✍️ **Отчёт**: скриншот кода репозитория, объяснение использования RealDictCursor.

## 🛠️ Задание 4: Реализация Business Logic Layer

### Файл services/user\_service.py:

|  |
| --- |
| python |
| from typing import List, Optional  from models.user import User  from repositories.base import UserRepository  class UserService:  def \_\_init\_\_(self, user\_repo: UserRepository):  self.user\_repo = user\_repo  def register\_user(self, name: str, email: str) -> User:  if not email or "@" not in email:  raise ValueError("Некорректный email")  if self.user\_repo.get\_user\_by\_email(email):  raise ValueError("Пользователь с таким email уже существует")  return self.user\_repo.create\_user(name, email)  def find\_user\_by\_email(self, email: str) -> Optional[User]:  return self.user\_repo.get\_user\_by\_email(email)  def list\_all\_users(self) -> List[User]:  return self.user\_repo.get\_all\_users() |

💡 Обратите внимание: сервис **принимает репозиторий через конструктор** — это **инверсия зависимостей**. Вы можете передать любой репозиторий, реализующий интерфейс.

## 🛠️ Задание 5: Presentation Layer (Flask API)

### Файл app.py:

|  |
| --- |
| python |
| from flask import Flask, request, jsonify  from services.user\_service import UserService  from repositories.postgres\_user\_repo import PostgresUserRepository  app = Flask(\_\_name\_\_)  # Инициализация зависимостей  repo = PostgresUserRepository()  service = UserService(repo)  @app.route('/health')  def health():  return {"status": "ok"}  @app.route('/users', methods=['POST'])  def register\_user():  data = request.get\_json()  try:  user = service.register\_user(data['name'], data['email'])  return jsonify({  "id": user.id,  "name": user.name,  "email": user.email,  "created\_at": user.created\_at.isoformat()  }), 201  except ValueError as e:  return jsonify({"error": str(e)}), 400  except Exception as e:  return jsonify({"error": "Internal error"}), 500  @app.route('/users', methods=['GET'])  def get\_users():  users = service.list\_all\_users()  return jsonify([{  "id": u.id,  "name": u.name,  "email": u.email,  "created\_at": u.created\_at.isoformat()  } for u in users])  if \_\_name\_\_ == '\_\_main\_\_':  app.run(host='0.0.0.0', port=5000) |

### Файл requirements.txt:

|  |
| --- |
|  |
| Flask==3.0.0  psycopg2-binary==2.9.7 |

## 🛠️ Задание 6: Тестирование и демонстрация слоистости

### 1. Запуск приложения

|  |
| --- |
| Командная строка |
| python app.py |

### 2. Тест через curl

|  |
| --- |
| Командная строка |
| curl -X POST http://localhost:5000/users \  -H "Content-Type: application/json" \  -d '{"name": "Иван", "email": "ivan@example.com"}'  curl http://localhost:5000/users |

### 3. Unit-тест (файл tests/test\_user\_service.py)

|  |
| --- |
| python |
| import unittest  from unittest.mock import Mock  from models.user import User  from services.user\_service import UserService  from repositories.base import UserRepository  class TestUserService(unittest.TestCase):  def setUp(self):  self.mock\_repo = Mock(spec=UserRepository)  self.service = UserService(self.mock\_repo)  def test\_register\_new\_user(self):  self.mock\_repo.get\_user\_by\_email.return\_value = None  fake\_user = User(1, "Анна", "anna@example.com", None)  self.mock\_repo.create\_user.return\_value = fake\_user  user = self.service.register\_user("Анна", "anna@example.com")  self.assertEqual(user.name, "Анна")  self.mock\_repo.create\_user.assert\_called\_once()  def test\_email\_already\_exists(self):  existing\_user = User(1, "Борис", "boris@example.com", None)  self.mock\_repo.get\_user\_by\_email.return\_value = existing\_user  with self.assertRaises(ValueError):  self.service.register\_user("Борис", "boris@example.com")  if \_\_name\_\_ == '\_\_main\_\_':  unittest.main() |

Запуск тестов:

|  |
| --- |
| Командная строка |
| python -m unittest tests/test\_user\_service.py -v |

✍️ Отчёт: скриншоты тестов, вывод curl, объяснение, как мокинг помогает тестировать логику без БД.

**📊 Отчёт по лабораторной работе**

Студент оформляет отчёт, включающий:

1. Титульный лист.
2. Цель и задачи.
3. Теоретическая часть (слоистая архитектура, инверсия зависимостей).
4. Ход выполнения:
   * Структура проекта,
   * Код ключевых компонентов,
   * Скриншоты: создание БД, запросы curl, результаты тестов.
5. Выводы:
   * Как слоистая архитектура упростила разработку?
   * Можно ли заменить PostgreSQL на другую БД? Как?
   * Какие сложности возникли?