**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ   
ІМЕНІ ТАРАСА ШЕВЧЕНКА**

Факультет комп’ютерних наук та кібернетики

Кафедра теоретичної кібернетики

**Звіт**

до лабораторної роботи на тему

**ФРАГМЕНТАРНА РЕАЛІЗАЦІЯ СИСТЕМИ УПРАВЛІННЯ ТАБЛИЧНИМИ БАЗАМИ ДАНИХ**

Виконав студент 4-го курсу   
Некряч Владислав Вадимович

Київ – 2022

# Постановка задачі

Створити декілька варіантів реалізації системи управління табличними базами даних з використанням запропонованих технологій.

Загальні вимоги:

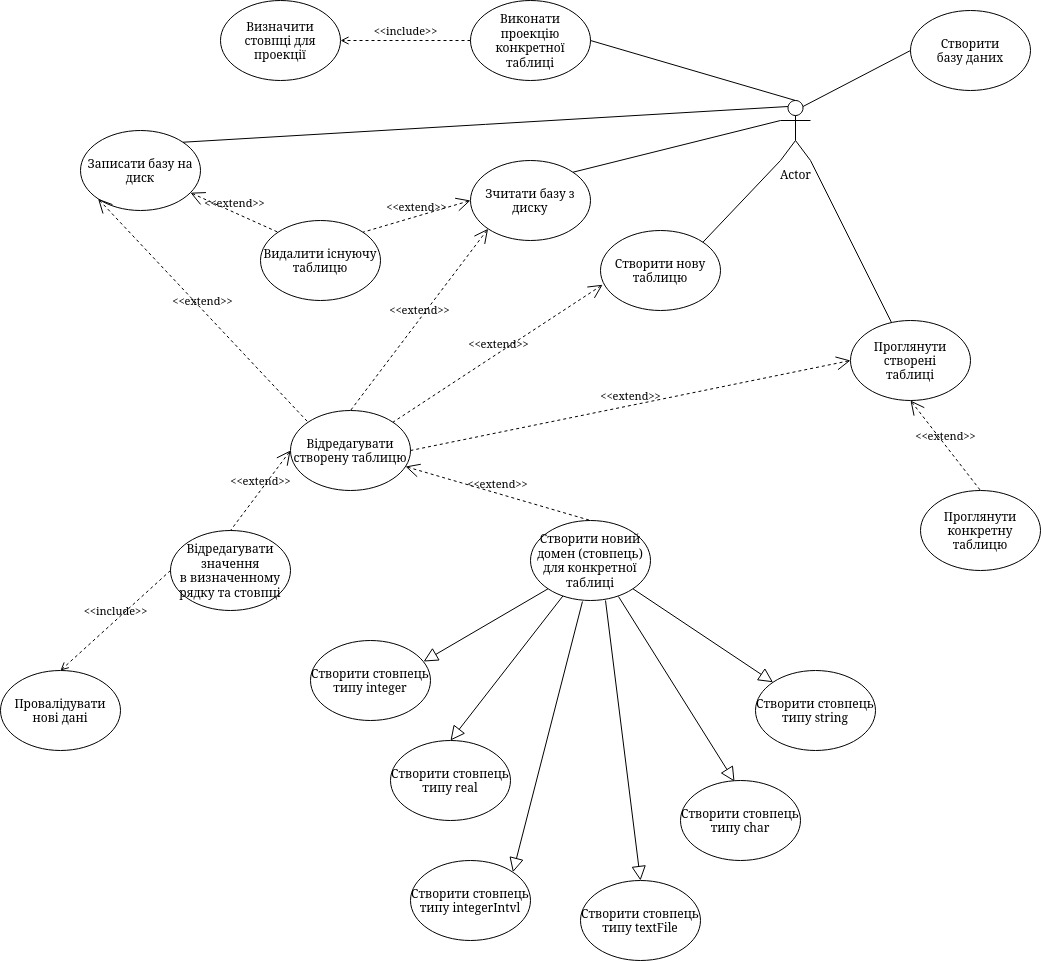
* кількість таблиць принципово не обмежена;
* кількість полів та записів у кожній таблиці принципово не обмежені;
* забезпечити підтримку типів integer, real, char, string
* реалізувати створення бази;
* реалізувати створення (з валідацією даних) та знищення таблиці з бази;
* реалізувати перегляд та редагування рядків таблиці;
* реалізувати збереження табличної бази на диску та, навпаки, її зчитування з диску.

Додаткові типи та операція над таблицями – варіант 17:

1) типи «текстовий файл» та «інтервал цілих чисел»

7) проекція таблиць

**0) Функціональна специфікація СУТБД – діаграма прецедентів UML**



**1) Використання UML при проектуванні та специфікації програмних систем**

• Діаграма прецедентів (1) – див. попередній етап.

**Потоки подій до 3 прецедентів**

Flow-of-Events

Flow 1: Creating a database

Preconditions: None

User clicks Create new database link.

They write the name of a database they want to create in the client. [A1]

User clicks "Create new database" button.

System creates new database with the specified name in case the name of the new database passed all validation checks. [E1]

Alternative flows:

A1. If the name doesn't pass validation checks, user gets to write a better name again.

Error flows:

E1. React Client will show an error message that a database was not created and the user will have the ability to go through the use case again.

Flow 2: Projecting a table

Preconditions: at least one database and one table should be created. (e.g., see Flow 1: creating a database)

User clicks on the database name and then the user clicks on the table name they want to project.

User selects the columns they want to project.

User clicks "Project" button.

System creates a projection of the specified table and shows the projection in the client. [E1]

Error flows:

E1. In case the projection was not created successfully, client shows the error and user will have the ability to go through the use case again.

Flow 3: Creating a database from dump

Preconditions: user should have access to a dump file that contains SQL commands to recreate the database

User clicks the button "Create database from dump".

User selects the dump file and clicks OK.

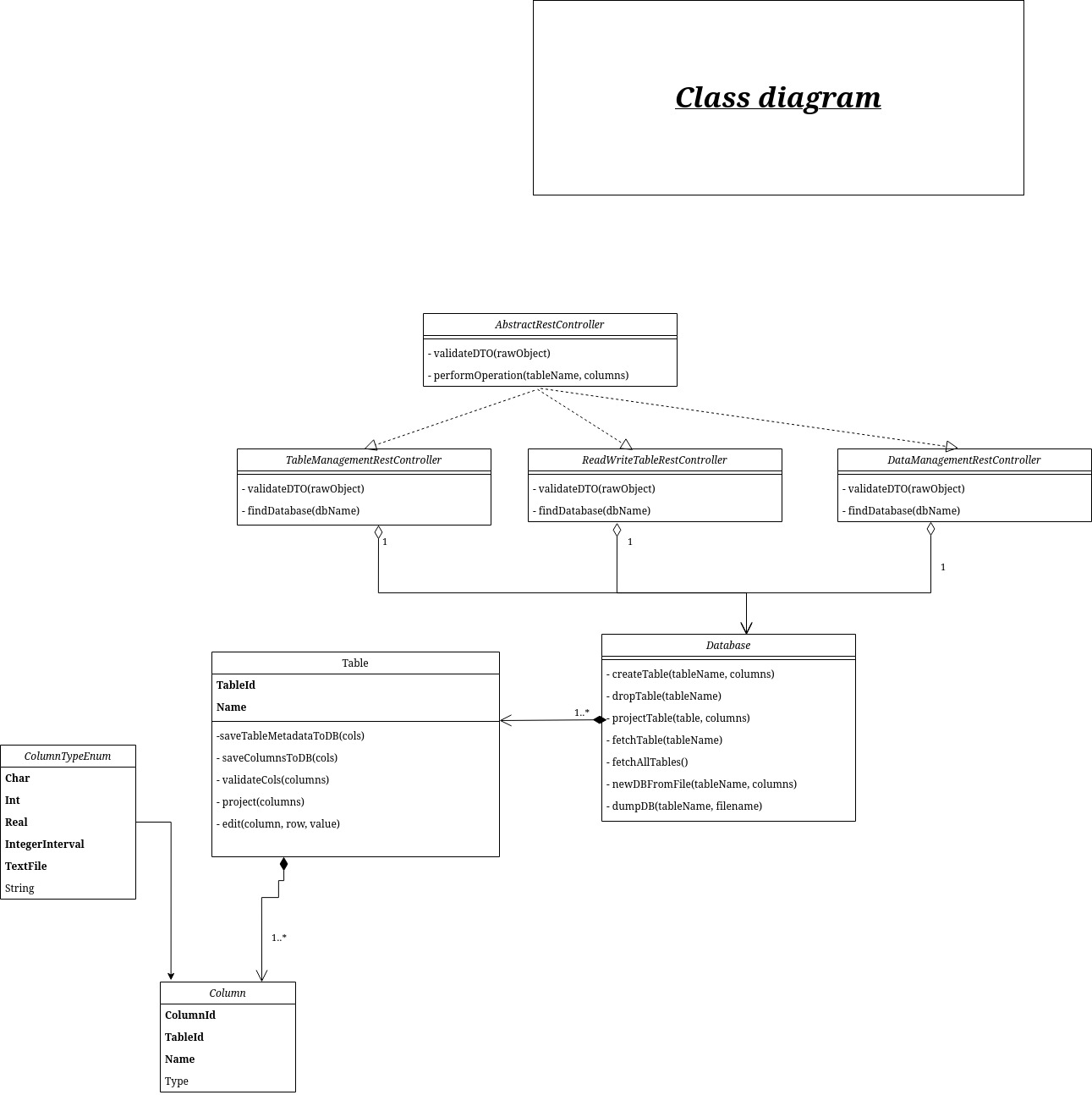
System creates a new database (in case one didn't exist before) using the dump file. [E1]

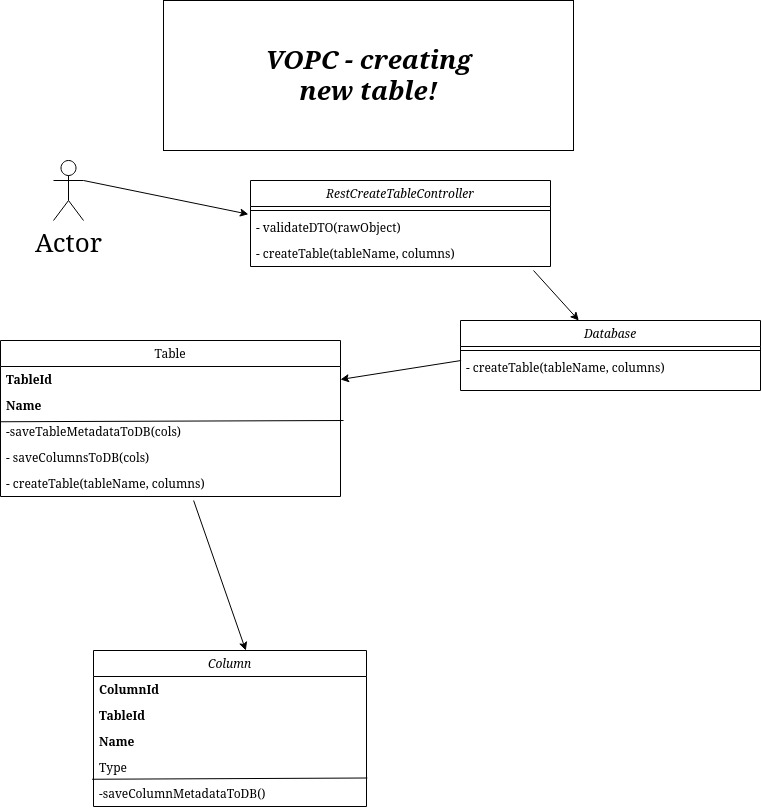
Error flows:

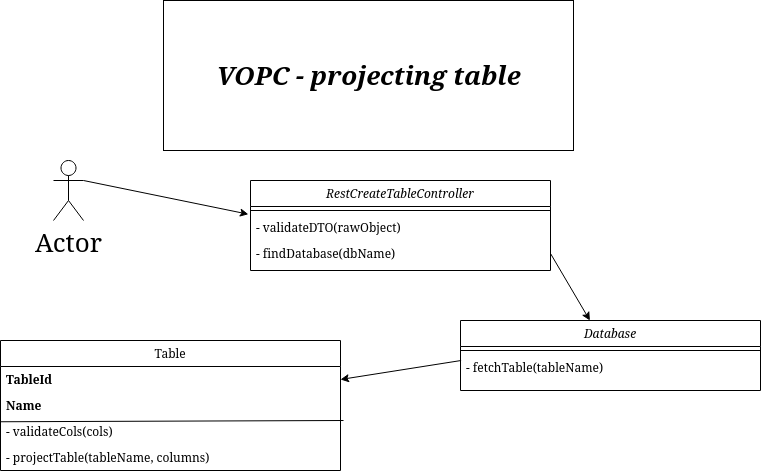
E1. In case the dump file is corrupted/not correct, user is shown an error message that the dump file is corrupted and database was not created.

In this case, user still has the ability to go through the use case again.

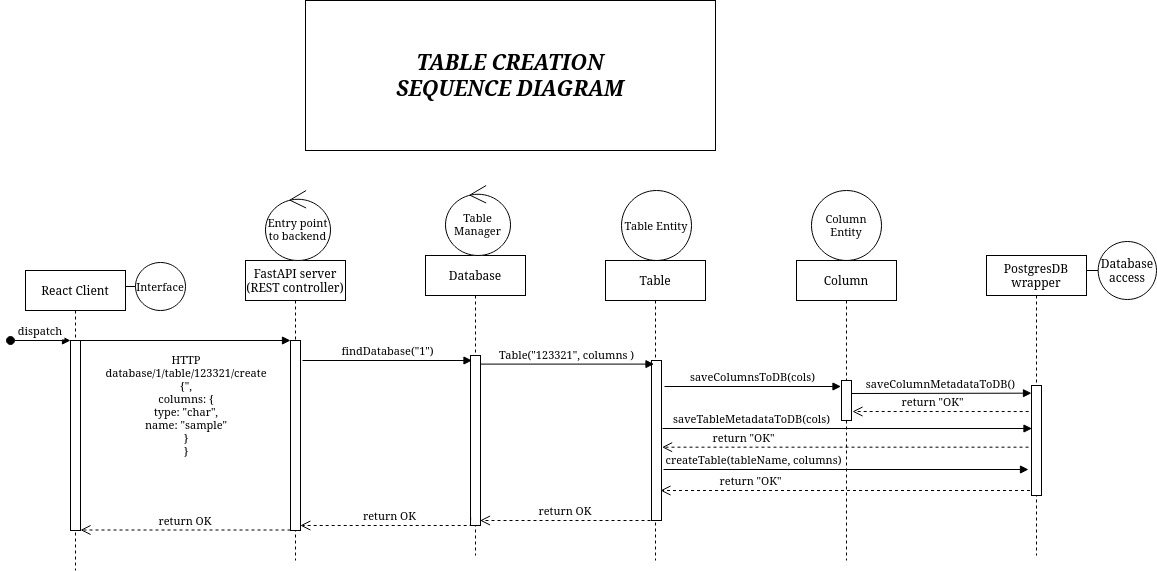
• Діаграми класів (3)

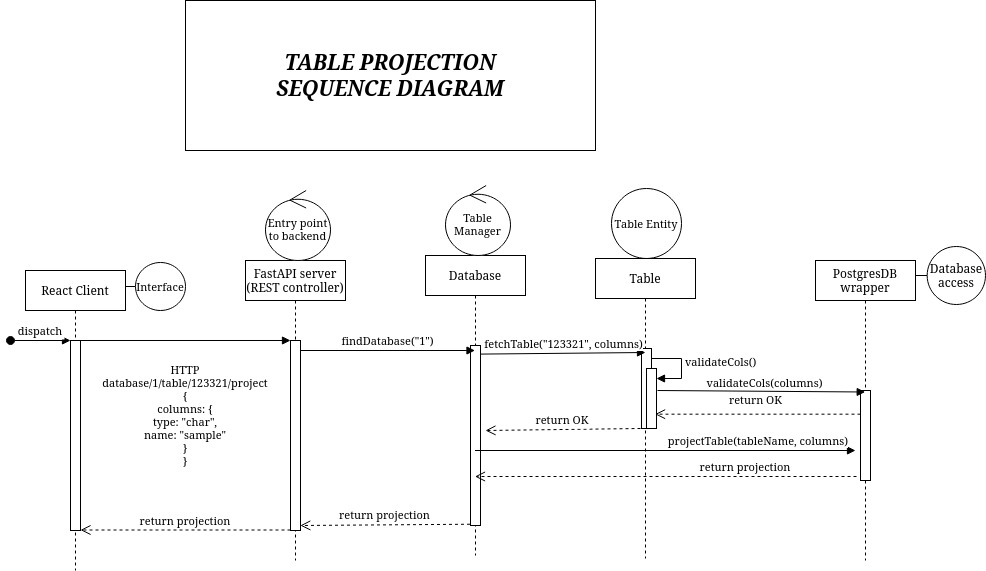


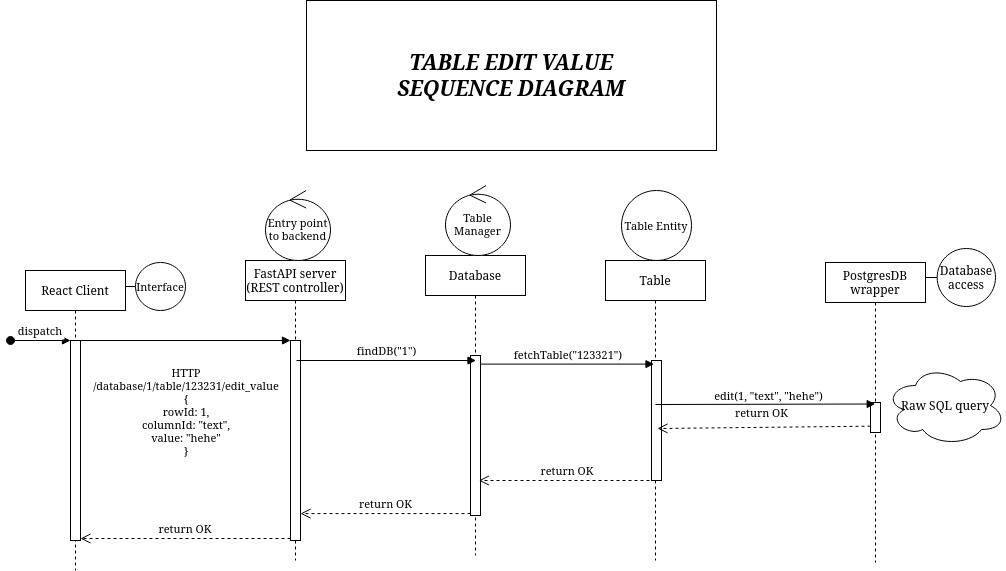


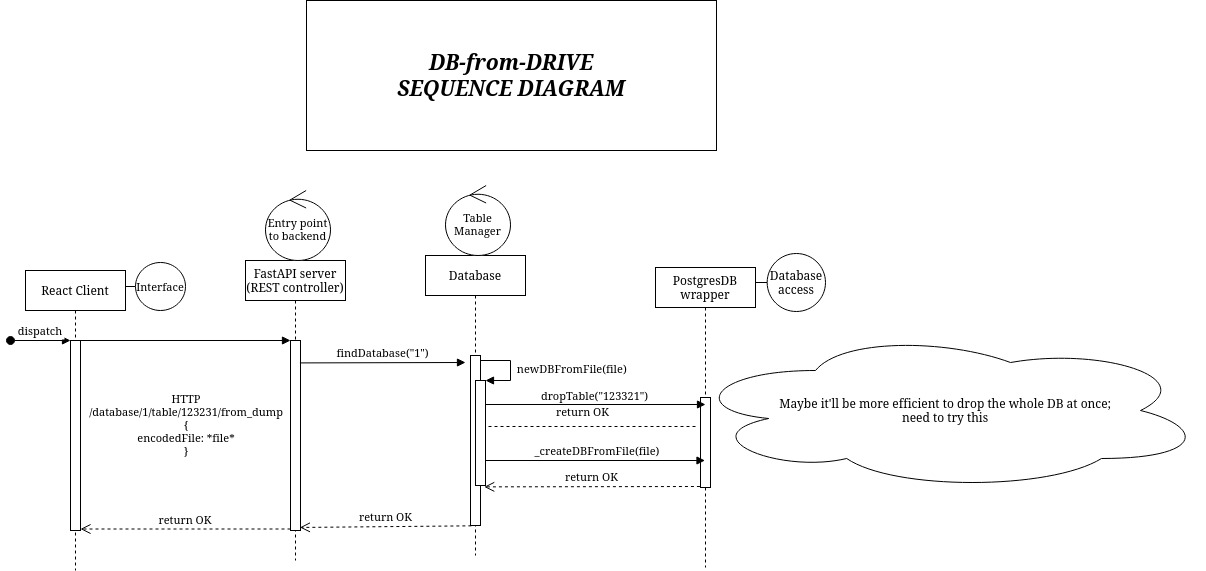


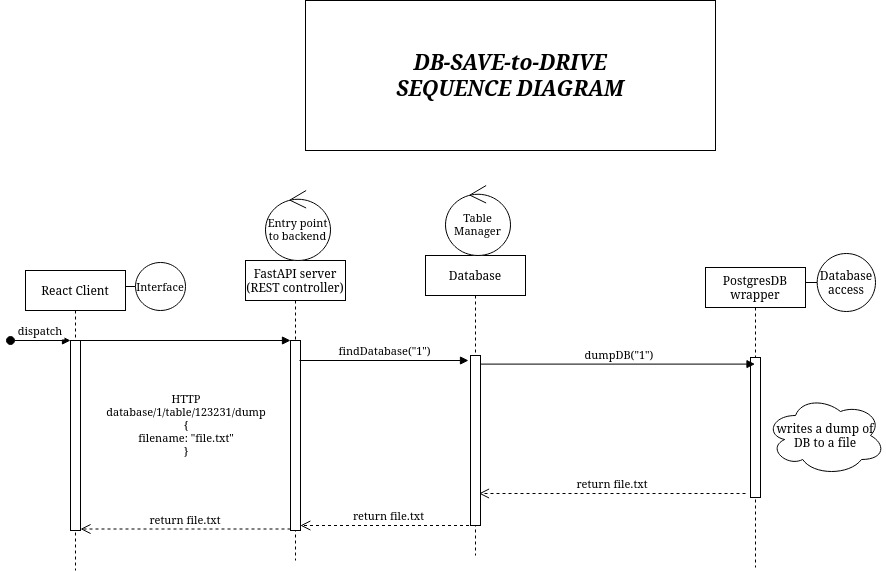
• Діаграми взаємодії (6)

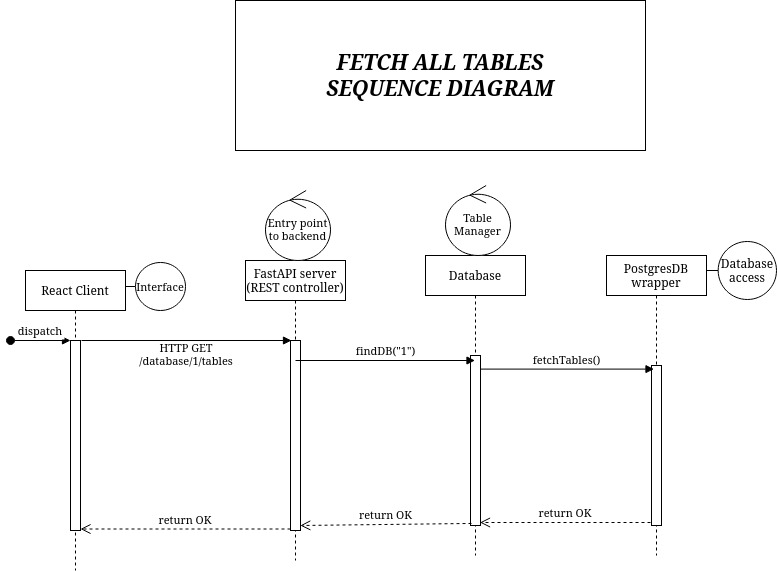




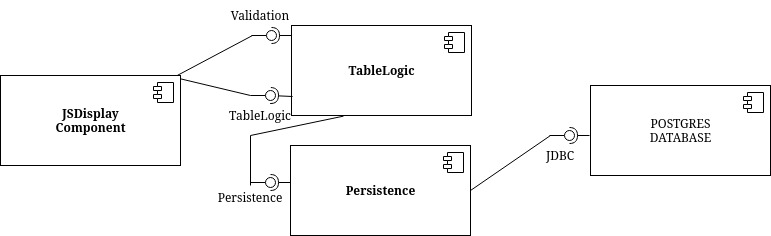




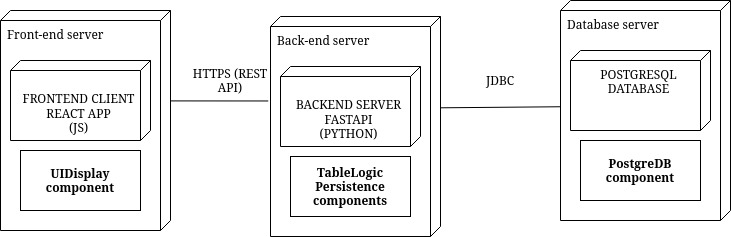




• Діаграми компонентів



• Діаграма розгортання



**2-3) Розробка локальної версії СУТБД**

• Розробка класів

**Database** – клас бази даних, містить поле назви *name* та колекцію таблиць *tables*:

class Database(BaseORMModel):  
 \_\_tablename\_\_ = "databases"  
  
 id = Column(Integer, primary\_key=True, index=True)  
 name = Column(String, unique=True)  
 tables = relationship("Table", back\_populates="database", cascade="all, delete")

**Table** – клас таблиці, містить поле назви *name,* посилання на базу даних до якої відноситься ця таблиця та настовпчики *columns*:

class Table(BaseORMModel):  
 \_\_tablename\_\_ = "tables"  
  
 id = Column(Integer, primary\_key=True, index=True)  
 name = Column(String)  
 database\_id = Column(Integer, ForeignKey("databases.id"))  
  
 database = relationship("Database", back\_populates="tables")  
 columns = relationship("TableColumn", back\_populates="table", cascade="all, delete")

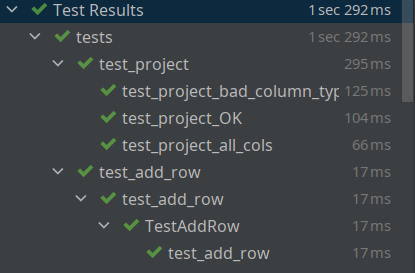
**TableColumn** – клас стовпчика. Містить поле назви *name*, поле типу *type* та посилання на відповідну таблицю.

class TableColumn(BaseORMModel):  
 \_\_tablename\_\_ = "table\_columns"  
  
 id = Column(Integer, primary\_key=True, index=True)  
 name = Column(String)  
 type = Column(Enum(ColumnType))  
 table\_id = Column(Integer, ForeignKey("tables.id"))  
  
 table = relationship("Table", back\_populates="columns")

• Створення UML-діаграми класів – див. етап 1

• Проведення unit-тестування

Зробимо unit-тести на валідацію та варіантну операцію за допомогою фреймворку xUnit:



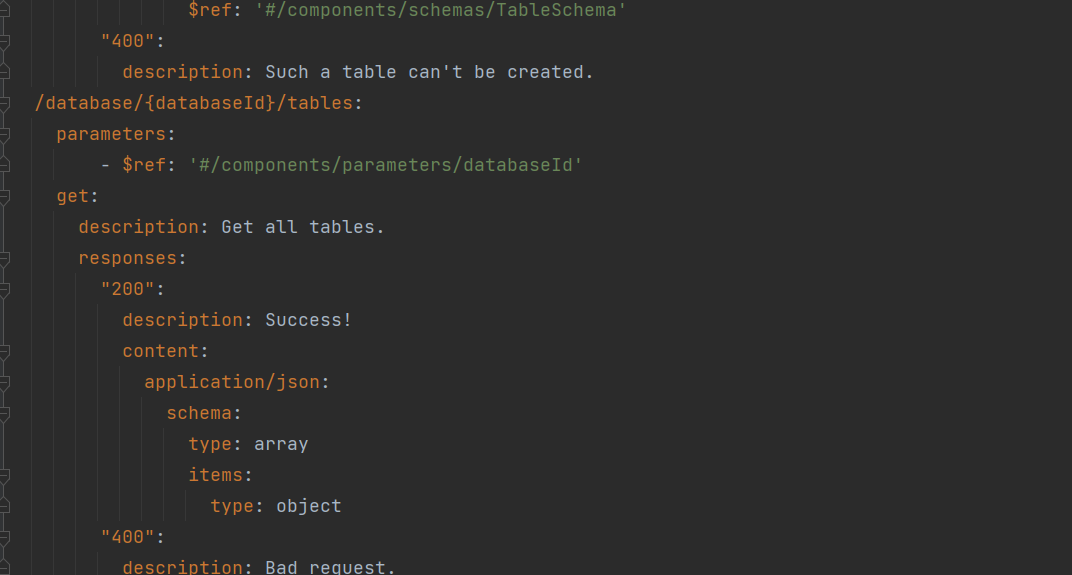
**10) REST web-сервіси**

**11) REST web-сервіси + HATEOAS**

**12) REST web-сервіси. Розробка OpenAPI Specification**

**13) REST web-сервіси. Реалізація серверного проєкту, використовуючи кодогенерацію стабу за OpenAPI Specification.**

Спочатку була написана OpenAPI специфікація стандарту 3.0.



За цією специфікацією був згенерований стаб серверної частини проекту.



Згідно з концепцією HATEOAS, при поверненні результату роботи запиту над певним об’єктом ми маємо включити також набір посилань на інші запити, що відповідають діям, які можна виконати над даним об’єктом або його складовими. Ми створили окрему функцію hateoas\_links, яка повертає можливі дії в залежності від поточного стану додатку:

def hateoas\_links(  
 database: str = None,  
 table: str = None,  
):  
 link\_dictionary = {  
 "read\_from\_dump": "/database/read\_dump",  
 }  
 if database:  
 link\_dictionary["get\_dump"] = "/database/{database}/get\_dump"  
 link\_dictionary["create\_table"] = "/database/{database}/table/create"  
 link\_dictionary["get\_all\_tables"] = "/database/{databaseId}/tables"  
 if table:  
 link\_dictionary["add\_row"] = "/database/{databaseId}/table/{table\_id}/add\_row"  
 link\_dictionary["delete\_table"] = "/database/{databaseId}/table/{tableId}"  
 link\_dictionary["project\_table"] = "/database/{databaseId}/table/project"  
 link\_dictionary["get\_data\_from\_table"] = "/database/{databaseId}/table/{tableId}"  
 if database and table and table\_has\_data(database, table):  
 link\_dictionary["edit\_value"] = "/database/{databaseId}/table/{tableId}/edit\_value"  
 return link\_dictionary

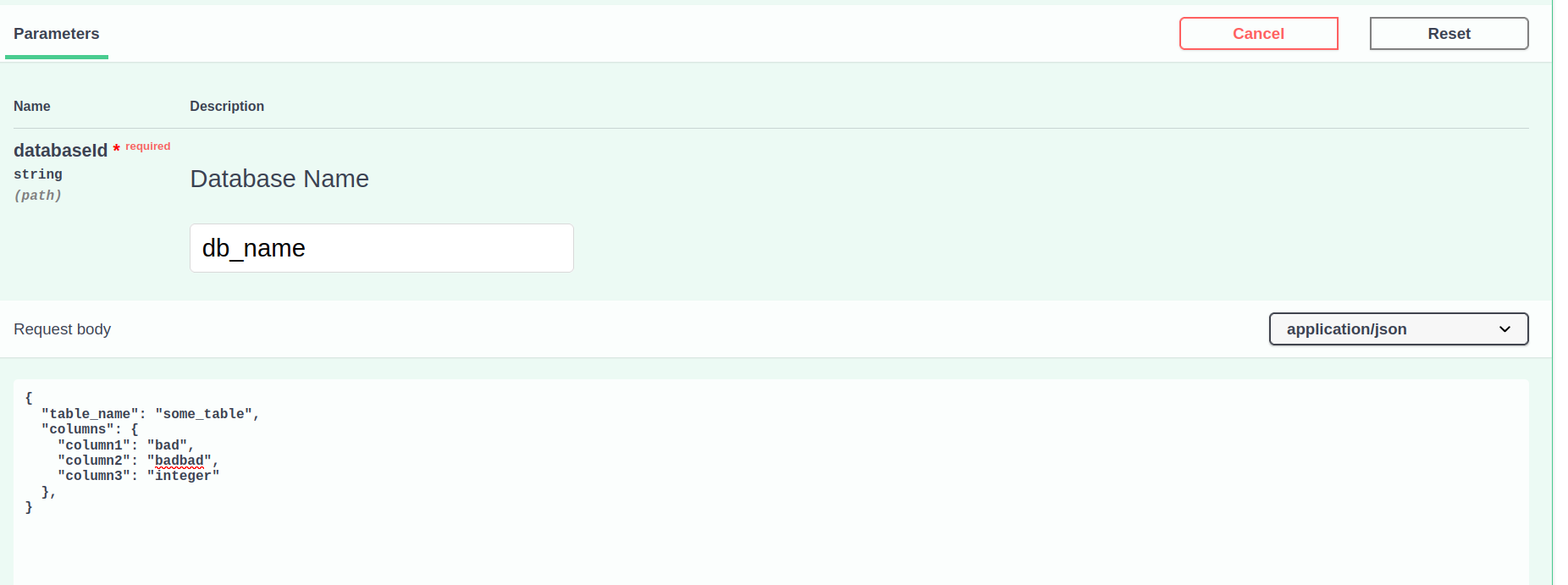
Тепер наведемо приклад методу, що відповідає запиту на створення нової БД.

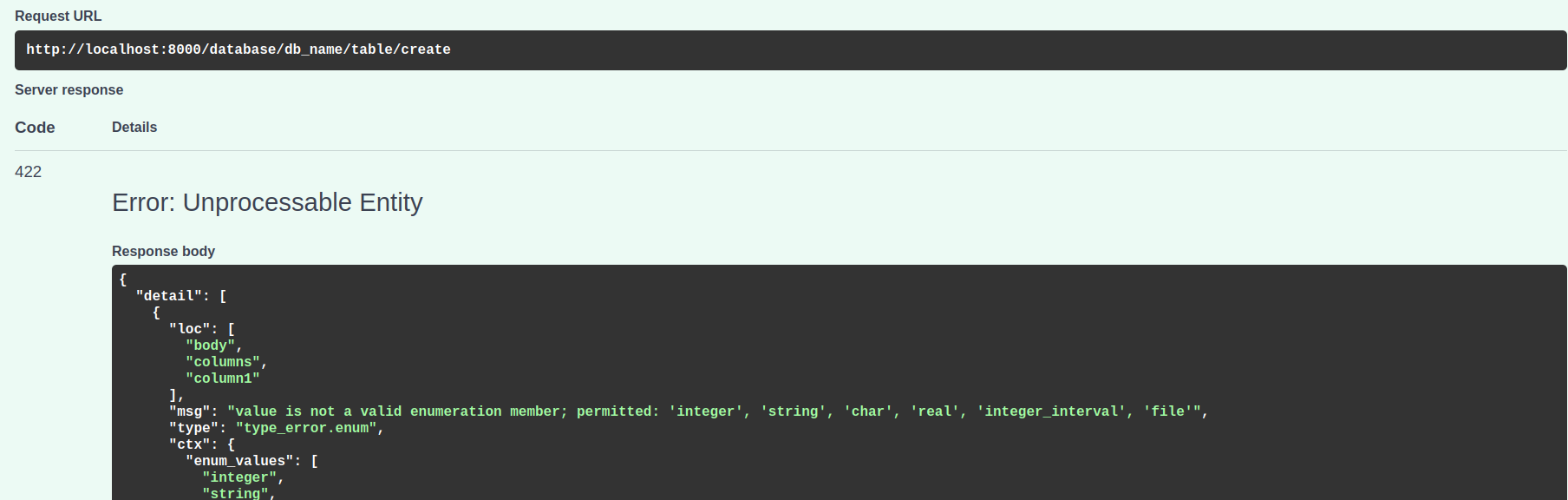
@router.post(  
 "/database/create",  
 responses={  
 201: {"model": DatabaseCreatePost201Response, "description": "Database has been created"},  
 400: {"description": "An error occured during database creation."},  
 },  
 tags=["default"],  
 response\_model\_by\_alias=True,  
)  
async def create\_database(  
 response: Response,  
 database\_create\_post\_request: DatabaseCreatePostRequest = Body(None, description=""),  
 meta\_database: Session = Depends(get\_meta\_db),  
) -> DatabaseCreatePost201Response:  
 *"""Create an empty database"""*  
database\_name = database\_create\_post\_request.database\_name  
 pattern = re.compile("[a-zA-Z]")  
 if not pattern.match(database\_name):  
 raise HTTPException(status\_code=400, detail="Bad request!")  
 connection, cursor = get\_db\_connection()  
 try:  
 cursor.execute(f"CREATE DATABASE {database\_name};")  
 logger.info(f"Created database {database\_name}")  
 save\_database\_metadata(database\_name=database\_name, session=meta\_database)  
 except Exception as e:  
 raise HTTPException(status\_code=400, detail=str(e))  
 finally:  
 connection.close()  
  
 response.status\_code = 201  
 response\_data = DatabaseCreatePost201Response(  
 database\_name=database\_create\_post\_request.database\_name,  
 links=hateoas\_links(database=database\_name)  
 )  
 return response\_data

Нижче наведено приклад виконання POST-запиту створення бази даних:



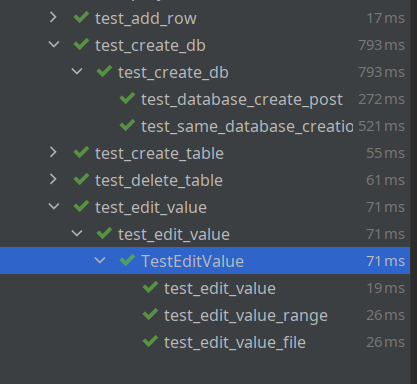
У запитах також враховані можливі помилкові ситуації: наприклад, якщо ми спробуємо створити таблицю з неіснуючим типом колонки, то буде виведено повідомлення про помилку:





**27) Інтегроване (Mock-) тестування у проєктах, що використовують реальні СУБД (реляційні чи ні) для збереження даних (етапи 24, 25).**

Було написано 11 тестів, з яких 8 є інтеграційними.



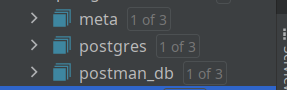
Всі тести після закінчення виконання видаляють тестову базу даних, тим самим роблячи процес тестування зручним та швидким для розробника.

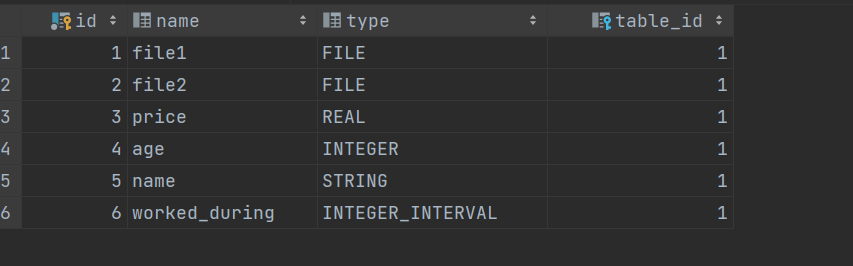
Приклад тесту для створення БД:

def test\_database\_create\_post(  
 client: TestClient,  
 test\_meta\_db\_connection: psycopg2.extensions.cursor,  
 test\_database\_name: str  
):  
 *"""Test case for database\_create\_post*  
 *"""*  
  
database\_create\_post\_request = DatabaseCreatePostRequest(database\_name=test\_database\_name)  
 headers = {  
 }  
 response = client.request(  
 "POST",  
 "/database/create",  
 headers=headers,  
 json=dict(database\_create\_post\_request),  
 )  
 response\_content = json.loads(response.content)  
  
 try:  
 assert db\_exists(database=test\_database\_name, connection=test\_meta\_db\_connection)  
 assert metadata\_about\_database\_was\_saved(  
 database=test\_database\_name,  
 connection=test\_meta\_db\_connection  
 )  
 assert response\_content.get("database\_name") == test\_database\_name  
 assert response.status\_code == 201  
 finally:  
 test\_meta\_db\_connection.execute(f"DROP DATABASE IF EXISTS {test\_database\_name}")

**24) Варіант проекту з використанням реляційної СУБД**

В якості СУБД візьмемо PostgreSQL Server. Проект ділиться на 2 частини – статичну та динамічну частини. В статичній частині (БД meta) зберігаються метадані про створені БД – інформація про колонки, таблиці та власне БД. В динамічній частині (як приклад показана створена БД postman\_db) зберігаються створені БД та їх дані.



Приклад таблиці з метаданими про колонки таблиць:

Для збереження даних в додаток виконується маппінг типів в типи PostgreSQL:

datatypes\_mapping = {  
 ColumnType.STRING: "VARCHAR",  
 ColumnType.REAL: "NUMERIC",  
 ColumnType.CHAR: "VARCHAR(1)",  
 ColumnType.INTEGER: "INTEGER",  
 ColumnType.INTEGER\_INTERVAL: "int4range",  
 ColumnType.FILE: "BYTEA"  
}

Кожного разу, коли ми оновлюємо структуру динамічної частини додатку (створення нової БД, таблиці, видалення таблиці, тощо), ми оновлюємо метадані. Як приклад розглянемо метод для видалення існуючої таблиці.

@router.delete(  
 "/database/{databaseId}/table/{tableId}",  
 responses={  
 200: {"description": "Deletion succesful"},  
 400: {"description": "An error occured during deletion."},  
 },  
 tags=["default"],  
 response\_model\_by\_alias=True,  
)  
async def drop\_database\_table(  
 response: Response,  
 databaseId: str = Path(None, description="Database name"),  
 tableId: str = Path(None, description="Table name"),  
 meta\_database: Session = Depends(get\_meta\_db),  
) -> dict:  
 *"""Drop table from database"""*  
connection, cursor = get\_db\_connection(database=databaseId)  
 try:  
 cursor.execute(f"DROP TABLE {tableId};")  
 delete\_table\_metadata(database=databaseId, table=tableId, session=meta\_database)  
 except Exception as e:  
 raise HTTPException(status\_code=400, detail=str(e))  
 finally:  
 connection.close()  
 response.status\_code = 200  
 return hateoas\_links(database=databaseId)

Визначення delete\_database\_metadata():

def delete\_table\_metadata(database: str, table: str, session: Session):  
 database = session.query(Database) \  
 .filter(Database.name == database) \  
 .first()  
 table = session.query(Table)\  
 .filter(  
 Table.database\_id == database.id,  
 Table.name == table,  
 ) \  
 .first()  
 session.delete(table)  
 session.commit()

**\*) Створення Докер-файлів до проекту**

Було також створено докер-файли для контейнеризації проекту.

**Dockerfile**

FROM python:3.7 AS *builder*  
  
WORKDIR /usr/src/app  
  
RUN python3 -m venv /venv  
ENV *PATH*="/venv/bin:$*PATH*"  
  
RUN pip install --upgrade pip  
  
COPY . .  
RUN pip install --no-cache-dir .  
  
  
FROM python:3.7 AS *test\_runner*  
WORKDIR /tmp  
COPY --from=*builder* /venv /venv  
COPY --from=*builder* /usr/src/app/tests tests  
ENV *PATH*=/venv/bin:$*PATH*  
  
# install test dependencies  
RUN pip install pytest  
  
# run tests  
RUN pytest tests  
  
  
FROM python:3.7 AS *service*  
WORKDIR /root/app/site-packages  
COPY --from=*test\_runner* /venv /venv  
ENV *PATH*=/venv/bin:$*PATH*

**docker-compose.yaml:**

version: '3.6'  
services:  
 service:  
 build:  
 context: .  
 target: service  
 ports:  
 - "8080:8080"  
 command: uvicorn openapi\_server.main:app --host 0.0.0.0 --port 8080