Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Inżynieria Oprogramowania dla Elektromobilności

Ćwiczenie nr 3. Implementacja aplikacji bazodanowej

Zagadnienia do opracowania:

- system kontroli wersji na przykładzie Git
- system budowania CMake
- diagram klas UML
- system zarządzania relacyjnymi bazami danych (RDBMS) na przykładzie SQLite: typy i spójność danych, operacje CRUD (create, read, update, delete), relacje między tabelami bazy danych, złączenia (*ang. joins*)
- paradygmat programowania obiektowego
- inteligentne wskaźniki (*ang. smart pointers*) języka C++; zastosowania klasy *std::unique_ptr*

Literatura

[1] L. Beighley. *Head First SQL*. Sebastopol, CA, USA: O'Reilly, 2007.

[2] *CMake. Documentation*. `https://cmake.org/documentation/`. [Online; dostęp 22.10.2022].

[3] W. Gajda. *Git. Rozproszony system kontroli wersji*. Gliwice, Polska: Helion, 2013.

[4] Kornelia Indykiewicz. *Wykład: Inżynieria Oprogramowania dla Elektromobilności*. 2022.

[5] S. Chacon; B. Straub. *Pro Git*. `https://git-scm.com/book/pl/v2/`. [Online; dostęp 22.10.2022]. 2014.

Spis treści

1. Cel ćwiczenia

Celem ćwiczenia jest opanowanie podstawowych umiejętności z zakresu projektowania i implementacji relacyjnej bazy danych.

2. Wprowadzenie

2.1. Aplikacja procesora danych

Aplikacja procesora danych (*data_processor*) stanowi podsystem realizujący dostęp do bazy danych. Jej zadaniem jest tłumaczenie wiadomości odbieranych przez serwer TCP na kwerendy języka SQL. Dostęp do pliku bazy danych realizowany jest przy użyciu systemu SQLite. Aplikacja korzysta z zewnętrznej biblioteki **SQLiteC++** do wywoływania przygotowanych poleceń SQL.

2.2. Budowanie aplikacji

W celu zbudowania pliku wykonywalnego aplikacji procesora danych należy kolejno:

1. utworzyć swój katalog roboczy wewnątrz folderu **Documents**. Nazwa katalogu powinna być unikalna (np. stanowić numer indeksu);

2. z poziomu utworzonego katalogu uruchomić konsolę systemową (terminal) i pobrać repozytorium projektu wywołując polecenie:

   git clone [adres_repozytorium]

   Adres repozytorium projektu można znaleźć w dołączonej specyfikacji bazy danych;

3. przenieść się do katalogu zawierającego pobrane repozytorium (**dataprocessor**) i za pomocą konsoli systemowej wywołać polecenie:

git submodule update --init --recursive

w celu zaimportowania zależności projektowych (podmodułów). Przechodzenie między katalogami systemu *Linux* z poziomu konsoli systemowej realizowane jest za pomocą polecenia **cd**:

cd dataprocessor

4. przenieść się do katalogu:

dataprocessor\third_parties\libipc\third_parties\paho.mqtt.c

i za pomocą konsoli systemowej wywołać kolejno polecenia:

cmake -Bbuild -H. -DPAHO_ENABLE_TESTING=OFF -DPAHO_BUILD_STATIC=ON -DPAHO_WITH_SSL=ON -DPAHO_HIGH_PERFORMANCE=ON

oraz

sudo cmake --build build/ --target install

5. wrócić do głównego katalogu repozytorium (**dataprocessor**) i utworzyć w nim katalog roboczy (np. **build**). W celu utworzenia nowego katalogu można skorzystać z konsoli systemowej:

mkdir build

6. przenieść się do utworzonego katalogu roboczego i za pomocą konsoli systemowej wywołać polecenie:

cmake ..

w celu wykonania skryptu **CMakeLists.txt**;

7. Za pomocą konsoli systemowej wywołać polecenie:

make

aby zbudować plik wykonywalny aplikacji *data_processor*.

Zbudowany plik wykonywalny aplikacji zostanie umieszczony w nowo utworzonym katalogu **deploy** wewnątrz katalogu roboczego (tu: ***build***). Uruchomienie aplikacji (z poziomu katalogu **deploy**) za pomocą konsoli systemowej realizowane jest poleceniem:

./data_processor

Pomyślne uruchomienie aplikacji zostanie podsumowane na konsoli w postaci informacji o połączeniu z brokerem MQTT:

Application connected to MQTT broker

Błąd
sh: 1: sqlite3: not found
terminate called after throwing an instance of std::runtime_error
oznacza, że API **sqlite3** systemu **SQLite** nie jest zainstalowane w systemie operacyjnym. Instalację można przeprowadzić za pomocą konsoli systemowej, wywołując polecenie:

sudo apt install sqlite3

2.3. Zintegrowane środowisko programistyczne Qt Creator

Proponowanym zintegrowanym środowiskiem programistycznym (*IDE*) do pracy na zajęciach laboratoryjnych jest **Qt Creator**. Program można uruchomić za pomocą konsoli systemowej wywołując polecenie:

qtcreator

Otwarte zostanie okno główne programu. Aby zaimportować projekt *CMake* należy z paska menu wybrać **File** ⟶ **Open File or Project...** W oknie wyboru projektu należy przenieść się do katalogu repozytorium (**dataprocessor**) i zaznaczyć plik **CMakeLists.txt**, a następnie nacisnąć przycisk **Open** znajdujący się w prawym górnym rogu okna. W oknie konfiguracji projektu należy określić ustawienia narzędzi budowania aplikacji w środowisku **Qt Creator**. Zaznaczając opcję **Select all kits** wybieramy wszystkie dostępne konfiguracje. Kliknięcie przycisku **Configure Project** spowoduje wczytanie źródeł projektu i otwarcie okna edycji plików.

2.4. Konfiguracja aplikacji procesora danych

Katalog **data** znajdujący się w głównym katalogu repozytorium projektu (**dataprocessor**) zawiera dwa pliki: **app_config.json** oraz **init.sql**. Pierwszy z nich to plik konfiguracyjny aplikacji w formacie JSON (*JavaScript Object Notation*), składający się z trzech atrybutów:

- **enable_backup** – dopuszczalne wartości **true**/**false**; włącz albo wyłącz możliwość wykonywania kopii zapasowej bazy danych (przełączanie między klasami **BackupService** i **NullBackupService** na etapie uruchamiania aplikacji);
- **path_to_database_directory** – bezwzględna ścieżka do katalogu, w którym ma zostać umieszczony plik bazy danych;

- **path_to_database_backup_directory** – bezwzględna ścieżka do katalogu, w którym ma zostać umieszczony plik kopii zapasowej bazy danych.

Plik konfiguracyjny jest parsowany podczas uruchamiania aplikacji. W celu wczytania zmian w pliku należy ponownie uruchomić aplikację procesora danych.

Plik **init.sql** deklaruje transakcję, zapisaną w języku SQL, która ma zostać wykonana podczas tworzenia pliku bazy danych (**persistent_storage.db**). Można go wykorzystać do zdefiniowania tabel relacyjnej bazy danych, zapisując kolejne polecenia **CREATE TABLE** języka SQL między poleceniami **BEGIN TRANSACTION** oraz **COMMIT**. Plik parsowany jest podczas uruchamiania aplikacji.

2.5. Struktura aplikacji procesora danych

Katalog **src**, znajdujący się w głównym katalogu repozytorium projektu (**dataprocessor**), zawiera wszystkie pliki źródłowe aplikacji procesora danych. W pliku **main.cpp** znajduje się główna funkcja programu realizująca sekwencję inicjalizacji obiektów składających się na aplikację. Aplikacja korzysta z trzech bibliotek dołączanych jako podmoduły systemu kontroli wersji Git (*git submodules*): **SQLiteC++**, **common**, **ipc**.

Fabryka **dataprocessor::DatabaseFactory** tworzy obiekty bazy danych biblioteki **SQLiteC++** (**SQLite::Database**) za pomocą metody **dataprocessor::DatabaseFactory::create()**. Funkcja jako argument wywołania przyjmuje bezwzględną ścieżkę do katalogu, w którym ma zostać umieszczony plik bazy danych (**persistent_storage.db**). Ścieżka pobierana jest z pliku konfiguracyjnego **app_config.json**, znajdującego się w katalogu **data**.

Fabryka **dataprocessor::BackupServiceFactory** tworzy obiekty serwisu obsługującego polecenie wykonania kopii zapasowej bazy danych. W zależności od konfiguracji aplikacji (włączony/wyłączony mechanizm kopii zapasowej) metoda **dataprocessor::BackupServiceFactory::create()** zwró-

ci odpowiednio instancję klasy **dataprocessor::BackupService** albo **dataprocessor::NullBackupService**. Wykonanie kopii zapasowej realizowane jest przez metodę **dataprocessor::IBackupService::makeBackup()**. Funkcja jako argument wywołania przyjmuje bezwzględną ścieżkę do katalogu, w którym ma zostać umieszczony plik kopii zapasowej bazy danych. Wywołanie metody na obiekcie klasy **dataprocessor::NullBackupService** spowoduje obsługę błędu odwołania do niewspieranej funkcjonalności. Implementacja metody w klasie **dataprocessor::BackupService** jest przedmiotem **Zadania 4**. W obecnej implementacji podsystemu wywołanie metody na obiekcie klasy **dataprocessor::BackupService** nie ma żadnego efektu.

Klasa **dataprocessor::QueryService** realizuje kwerendy CRUD na bazie danych za pomocą odpowiednio metod:

- **dataprocessor::QueryService::insert()**;
- **dataprocessor::QueryService::select()**;
- **dataprocessor::QueryService::update()**;
- **dataprocessor::QueryService::remove()**.

[*Dla dociekliwych: dlaczego metoda remove nie posiada nazwy odpowiadającej realizowanej kwerendzie SQL (DELETE)?*]
Wszystkie metody jako argument wywołania przyjmują obiekt klasy **dataprocessor::Record**, przechowujący nazwy i wartości kolejnych kolumn wybranej tabeli bazy danych. Klasa **dataprocessor::Record** umożliwia transfer danych między wiadomością IPC a bazą danych. Obecna implementacja aplikacji procesora danych wspiera kwerendy SQL o następującej składni:

1. dodawanie rekordów do bazy danych

   INSERT INTO [nazwa_tabeli] ([kolumna_1], [kolumna_2], ...)
   VALUES ([wartość_1], [wartość_2], ...);

2. pobieranie rekordów z bazy danych

SELECT * FROM [nazwa_tabeli] WHERE [kolumna] = [wartość];

3. aktualizowanie rekordów w bazie danych

UPDATE [nazwa_tabeli] SET [kolumna1] = [wartość1] WHERE [kolumna2] = [wartość2];

4. usuwanie rekordów z bazy danych

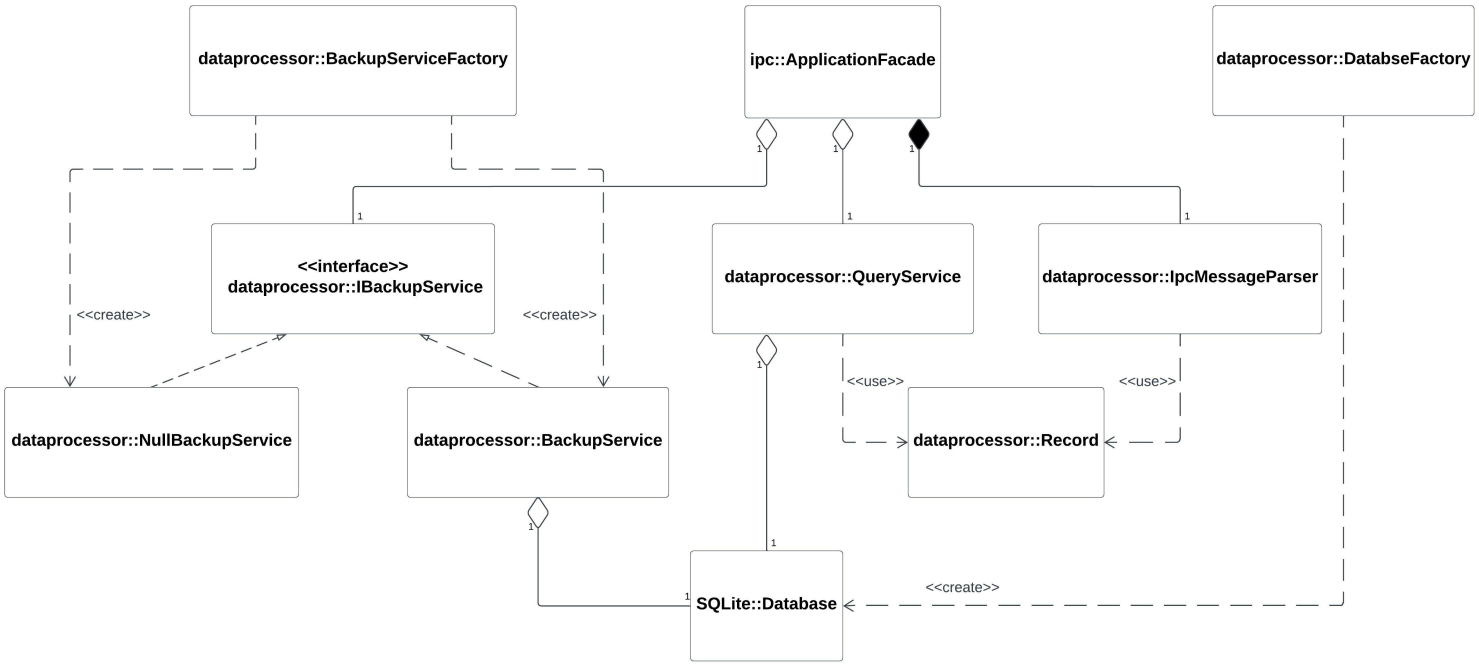
DELETE FROM [nazwa_tabeli] WHERE [kolumna] = [wartość];

Nazwy kolejnych kolumn i ich wartości w kwerendach SQL pobierane są z obiektu klasy dataprocessor::Record w kolejności w jakiej zostały przesłane w wiadomości IPC! (patrz: rozdział 2.6)

Klasa **ipc::ApplicationFacade** umożliwia konfigurację i obsługę komunikacji międzyprocesowej za pośrednictwem brokera komunikatów (tu: brokera MQTT). Z kolei klasa **dataprocessor::IpcMessageParser** realizuje translację wiadomości IPC na obiekt klasy **dataprocessor::Record**. Strukturę aplikacji w postaci diagramu klas UML przedstawiono na rys. 2.1.

2.6. Komunikacja międzyprocesowa (IPC)

Komunikacja z aplikacją procesora danych odbywa się za pośrednictwem protokołu MQTT (*ang. Message Queue Telemetry Transport*) realizującego wzorzec publikacja–subskrypcja. System wykorzystuje broker komunikatów (tu: Eclipse Moquitto) do wymiany danych między aplikacjami. Klienci subskrybują się do brokera na konkretne tematy wiadomości, które chcą odbierać (*topics*). Wiele aplikacji może jednocześnie być zasubskrybowanych, jak również publikować wiadomości na ten sam temat. Broker odbiera publikowane wiadomości i filtruje je na podstawie tematu, przesyłając do zasubskrybowanych klientów. Tematy wiadomości są strukturyzowane wg schematu

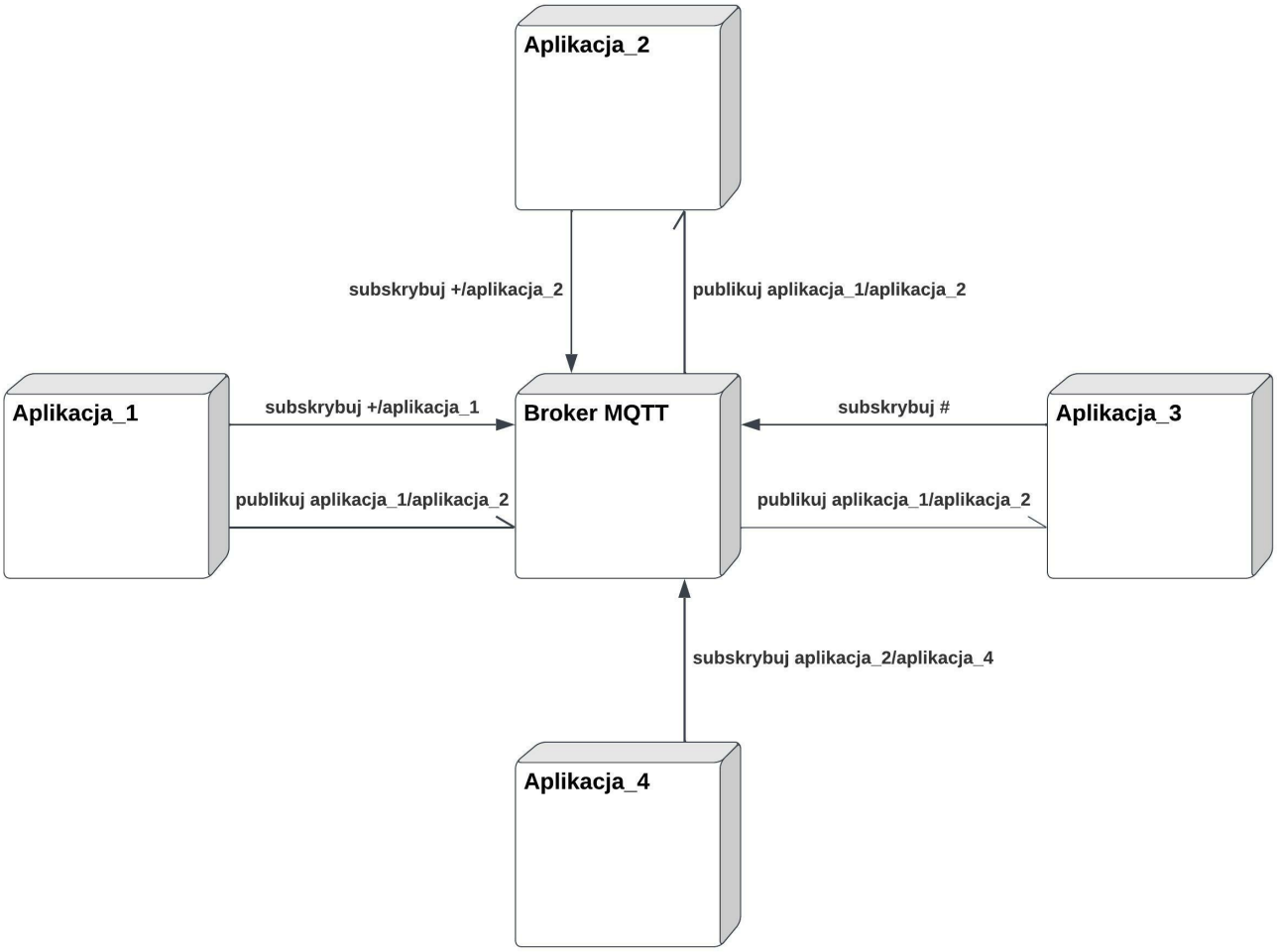


Rys. 2.1. Diagram klas UML aplikacji *data_processor*

człon_1/człon_2/.../człon_n, gdzie ukośnik stanowi separator, a poszczególne człony określane są przez użytkownika (aplikację). W implementacji biblioteki **ipc** ograniczono długość tematów wiadomości do dwóch członów wg schematu **nadawca/odbiorca**. Dodatkowo, w ramach procesu subskrypcji obsługiwane są dwa znaki specjalne:

- + – zastępuje pojedynczy człon tematu (dowolny odbiorca lub dowolny nadawca);
- # – zastępuje wszystkie człony tematu (dowolny temat).

Aplikacja procesora danych subskrybuje się do brokera MQTT na tematy wiadomości **+/data_processor** (dowolny nadawca do aplikacji o nazwie data_processor). Schemat komunikacji międzyprocesowej przedstawiono na rys. 2.2.



Rys. 2.2. Komunikacja międzyprocesowa (IPC) z wykorzystaniem brokera komunikatów MQTT

Treść wiadomości IPC jest ustandaryzowana wykorzystując format JSON. Biblioteka **ipc** wymaga, aby każda z parsowanych wiadomości zawierała obowiązkowo następujące atrybuty:

- **command** – polecenie do obsłużenia przez odbiorcę (w formacie tekstowym);

- **id** – identyfikator wiadomości wykorzystywany w komunikacji asynchronicznej (liczba całkowita w zakresie od 0 do 18446744073709551615 przyjęta arbitralnie);
- **responseExpected** – flaga informująca czy nadawca oczekuje otrzymania odpowiedzi na przesłaną wiadomość (wartość logiczna true/false).

Aplikacja procesora danych wspiera następujące polecenia realizujące odpowiadające im operacje CRUD oraz operację wykonania kopii zapasowej bazy danych:

- **insert**
  Przykład wiadomości IPC w formacie JSON:
  {
  ”command” : ”insert”,
  ”id” : 12345,
  ”responseExpected” : true,
  ”table” : ”students”,
  ”record” : [
  {”first_name” : ”Adam”},
  {”last_name” : ”Kowalski”},
  {”index” : ”123456”}
  ]
  }
- **select**
  Przykład wiadomości IPC w formacie JSON:
  {
  ”command” : ”select”,
  ”id” : 54,
  ”responseExpected” : true,
  ”table” : ”teachers”,
  ”record” : [

```
            {"last_name" : "Nowak"}
        ]
}
```

- **update**

  Przykład wiadomości IPC w formacie JSON:

```
{
    "command" : "update",
    "id" : 3,
    "responseExpected" : true,
    "table" : "books",
    "record" : [
        {"genre" : "fantasy"},
        {"book_id" : "4512"}
    ]
}
```

- **delete**

  Przykład wiadomości IPC w formacie JSON:

```
{
    "command" : "delete",
    "id" : 4329,
    "responseExpected" : true,
    "table" : "suppliers",
    "record" : [
        {"phone" : "660912553"}
    ]
}
```

- **backup**

  Przykład wiadomości IPC w formacie JSON:

```
{
    "command" : "backup",
```

”id” : 666,
”responseExpected” : true
}

W przypadku obsługi poleceń CRUD aplikacja procesora danych wymaga przekazania w wiadomości IPC dwóch dodatkowych atrybutów:

- **table** – nazwa tabeli bazy danych, na której ma zostać wykonana kwerenda (w formacie tekstowym);
- **record** – tablica obiektów JSON opisujących pary {nazwa kolumny–wartość} **w kolejności przekazywanej do kwerendy SQL** (zarówna nazwa kolumny, jak i wartość muszą zostać zapisane w formacie tekstowym).

Komunikację z brokerem MQTT (wysyłanie i odbieranie wiadomości o danym temacie) można realizować również z wykorzystaniem aplikacji zainstalowanych w ramach pakietu **mosquitto–clients**:

- **mosquitto_sub** – aplikacja nasłuchująca komunikacji z wykorzystaniem protokołu MQTT; w celu śledzenia wszystkich wiadomości przychodzących i wychodzących z brokera można uruchomić aplikację za pomocą konsoli systemowej wywołując polecenie:

  mosquitto_sub -t \# --pretty -v

- **mosquitto_pub** – aplikacja publikująca wiadomość z wykorzystaniem protokołu MQTT; aplikację można uruchomić za pomocą konsoli systemowej wywołując polecenie:

  mosquitto_pub -t [temat_wiadomości] -m [treść_wiadomości]

  Przykład:
  mosquitto_pub -t ”test/data_processor”
  -m ”{\”command\”:\”backup\”,\”id\”:123,\”responseExpected\”:false}”

Należy zwrócić uwagę, że wszystkie cudzysłowy zawarte w treści wiadomości muszą zostać poprzedzone ukośnikiem: \”.

3. Program ćwiczenia

Uwaga: Warunkiem dopuszczenia do realizacji laboratorium jest przedstawienie rozwiązania Zadania 1. na początku zajęć.

Uwaga: Zadania należy realizować w kolejności numerycznej.

Zadanie 1. (Zadanie domowe) Na podstawie załączonej specyfikacji MVP zaprojektuj tabele bazy danych. W tym celu:

- nazwij tabele bazy danych według ich przeznaczenia;
- określ relacje między tabelami (jeden–do–jednego / jeden–do–wielu / wiele–do–wielu);
- nazwij kolumny w każdej z wyróżnionych tabel;
- określ typ danych w każdej z kolumn poszczególnych tabel (**INTEGER** / **REAL** / **TEXT** / **BLOB**);
- określ ograniczenia wartości przyjmowanych przez dane w poszczególnych kolumnach;
- zidentyfikuj kolumny stanowiące klucze główne (**PRIMARY KEY**) i klucze obce (**FOREIGN KEY**).

Zapisz polecenie SQL **CREATE TABLE** dla każdej z tabel projektowanej bazy danych. Zapewnij integralność referencyjną (*ang. referential integrity*) między tabelami przez powiązanie klucza obcego z kluczem głównym.

Zadanie 2. (na ocenę 3.0) Zbuduj, a następnie skonfiguruj aplikację procesora danych. W tym celu:

- w pliku **data/app_config.json** uzupełnij bezwzględne ścieżki do katalogów bazy danych oraz kopii zapasowej bazy danych (arbitralnie wybrane katalogi w obrębie swojego katalogu roboczego);

- w pliku **data/init.sql**, w ramach transakcji SQL, zapisz polecenia SQL z **Zadania 1**. służące do utworzenia tabel bazy danych.

Następnie przetestuj działanie aplikacji:

- weryfikując każde z poleceń CRUD;
- przesyłając do aplikacji nieobsługiwaną komendę;
- przesyłając do aplikacji niepoprawnie sformatowaną wiadomość.

Przesyłane wiadomości IPC, wraz z otrzymanymi odpowiedziami zapisz w protokole z przebiegu ćwiczenia.

Zadanie 3. (na ocenę 4.0) Obecna implementacja procesora danych nie realizuje dwóch wymagań funkcjonalnych ze specyfikacji MVP. Jednym z nich jest tworzenie kopii zapasowej bazy danych. Przeanalizuj kod aplikacji oraz diagram klas UML (rys. 2.1) i zidentyfikuj drugie niezrealizowane wymaganie funkcjonalne. W protokole z przebiegu ćwiczenia zapisz kwerendę SQL, za pomocą której można obsłużyć brakujący scenariusz użycia aplikacji.

Zadanie 4. (na ocenę 5.0) Korzystając z klasy **SQLite::Backup** biblioteki **SQLiteC++** (`https://github.com/SRombauts/SQLiteCpp/blob/master/include/SQLiteCpp/Backup.h`) zaimplementuj mechanizm tworzenia kopii zapasowej pliku bazy danych. Implementację umieść wewnątrz metody **dataprocessor::BackupService::makeBackup()** (plik **BackupService.cpp**). Przetestuj działanie wdrożonego mechanizmu. W protokole z przebiegu ćwiczenia zapisz przesłane i odebrane wiadomości IPC.

W sprawozdaniu zawrzeć:

- protokół z przebiegu ćwiczenia;
- informację o przeznaczeniu obsługiwanej bazy danych;
- informację o zastosowanym systemie RDBMS;
- strukturę tabel bazy danych (nazwy tabel, nazwy kolumn, typy danych, ograniczenia wartości danych, klucze główne i obce, relacje między tabelami) wraz z poleceniami SQL umożliwiającymi utworzenie tych tabel;
- informację o relacjach między klasami podsystemu procesora danych (dziedziczenie / realizacja / asocjacja / agregacja / kompozycja), zidentyfikowanych na podstawie diagramu klas UML (rys. 2.1);
- wnioski z realizacji **Zadania 2.** (przeprowadzone testy, wysłane i odebrane wiadomości IPC, ocena poprawności działania podsystemu procesora danych);
- informację o niespełnionych wymaganiach funkcjonalnych w ramach MVP oraz kwerendę SQL, za pomocą której można obsłużyć brakujący scenariusz [*jeśli zostało zrealizowane* ***Zadanie 3.***];
- wnioski z realizacji **Zadania 4.** (przeprowadzone testy, wysłane i odebrane wiadomości IPC, ocena poprawności działania mechanizmu kopii zapasowej) [*jeśli zostało zrealizowane* ***Zadanie 4.***].