Etap 5
Modelowanie Danych

Zespół nr 4

1 Role

- Kierownik - Maciej Prostak
- Właściciel zadania projektowego - Adrian Trąbka
- Projektant struktury relacyjnej - Jakub Niewiński
- Projektant XML - Mikołaj Kowalczewski
- Projektanci/programiści struktur programowych/nierelacyjnych, współodpowiedzialni także za implementację importu/eksportu danych:

  Export - Maciej Prostak

  Import - Adrian Trąbka

2 Opis programu do eksportu danych

2.1 schemas.py

W tym pliku umieszczone zostały schematy pomagające konwertować dane z relacyjnej bazy do danych w formacie XML. Każdy schemat opiera się na klasie typu Dataclass pozwalająca na łatwe serializowanie danych. Każda klasa posiada metodę create_xml_element, która tworzy odpowiednie elementy XML za pomocą modułu lxml.builde: Strona biblioteki lxml.

Przykład można zaobserwować na klasie Address:

```
self._maker = lxml.builder.ElementMaker()
...

def create_xml_element(self):
    m = self._maker
    return m.address(
        m.country(self.country),
        m.city(self.city),
        m.state(self.state),
        m.zip_code(self.zip_code),
        m.street_name(self.street_name),
        m.building_number(self.building_number),
    )
```

Który ostatecznie stworzy strukturę XML w następujący sposób:

```
<address>
  <country>Poland</country>
  <city>Krakow</city>
  <state>malopolskie</state>
  <zip_code>01-005</zip_code>
  <street_name>Wierzbowa</street_name>
  <building_number>15a</building_number>
</address>
```

Niektóre elementy XML jednak wymagają podania atrybutów. Rozwiązanie jest również proste:

```
def create_xml_element(self):
    m = self._maker
    return m.Deliverer({"user_id": str(self.user_id)})
```

Podając zmienną typu słownik do elementu lxml builder otrzymujemy końcowo:

```
<Deliverer user_id="2436" />
```

Realizacja zagnieżdżeń w lxml element builder polega na podaniu stworzonego elementu do nadrzędnego.

2.2 db_export.py

Wszystkie tworzone zapytania do bazy i budowanie schematu Order znajdują się w pliku db_export.py.

Za pomocą biblioteki cx_Oracle łączymy się z bazą Oracle i wykonujemy odpowiednie zapytania. Cursor jest klasą pomocniczą pomagająca wykonywać kwerendy oraz operować na danych zwracanych z bazy. Do metody execute podajemy zapytanie SQL wraz z parametrami. W przypadku poniżej jest to user_id. Metoda fetchone zwraca nam pierwszy element w wykonanym zapytaniu, która następnie konwertujemy na namedtuple (w celu łatwiejszego debugowania).

Przykład prostego zapytania:

```
def get_customer(connection, user_id: int):
    cursor = connection.cursor()
    cursor.execute(
        """
    SELECT user_id,
    zip_code,
    country,
    city,
    state,
    street_name,
    building_number,
    apartment_number,
    nip
    FROM users
    WHERE user_id = :user_id
    """,
        user_id=user_id,
    )
    row = cursor.fetchone()
    return namedtuple_factory(cursor, row)
```

Połączenie do bazy danych:

```
cx_Oracle.init_oracle_client(lib_dir=lib_dir)
dsn = cx_Oracle.makedsn(host, port, service_name=service_name)

with cx_Oracle.connect(
    user=user, password=password, dsn=dsn, encoding="UTF-8"
) as connection:
    logging.info("Created connection with database")
```

Wszystkie parametry są dość oczywiste poza lib_dir. Jest to ścieżka do rozpakowanej biblioteki instant client podaną na stronie: Oracle Instant Client W opisywanym pliku ostatecznie obiekty stworzone w trakcie zapytań konwertujemy do naszej wewnętrznego schematu (podaną w schemas.py).

2.3 helpers.py - Budowanie i walidacja xml

W pliku helpers.py znajdują się proste funkcje budujące ostateczne drzewo XML i walidujące stworzony XML z plikiem XSD. W przypadku niepowodzenia walidacji program się kończy i na wyjście zostaje wypisany dokładny błąd walidacji.

2.4 send_to_client.py - Zapytanie do klienta

Po walidacji naszej utworzonej struktury XML za pomocą biblioteki requests wysyłany jest zapytanie na serwer klienta. Content-type wysłanych danych jest typu application/xml.

2.5 main.py

Jest to główny plik wykonujące kolejne operacje eksportu danych:

1. Pobranie z bazy zamówienia i konwersja na wewnętrzny schemat.
2. Skonwertowanie Order do struktury XML.
3. Walidacja stworzonej struktury XML.
4. Wysłanie do klienta struktury.

Za pomocą biblioteki load_dotenv wszystkie dane do połączenia z bazą zostają pobrane ze zmiennych znajdujących się w pliku .env.

3 Opis programu do importu danych

W pliku order.schemas.ts umieszczone zostały schematy pozwalające walidować poprawność danych przed umieszczeniem ich w nierelacyjnej bazie MongoDB. Użyto w tym celu gotowej klasy bibliotecznej Schema, która w konstruktorze przyjmuje definicję schematu. Przykładowy obiekt znajduje się w poniższym listingu.

```
export const OrderSchema: Schema = new Schema({
  id: { type: String, required: true },
  created_date: { type: String, required: true },
  total_cost: { type: String, required: true },
  state: { type: String, required: true },
  destination_address: { type: AddressSchema, required: true },
  restaurant: { type: RestaurantSchema, required: true },
  ordered_dishes: { type: [OrderedDishSchema], required: true },
  deliverer: { type: DelivererSchema, required: true },
  customer: { type: CustomerSchema, required: true },
  user_note: { type: String, required: true },
  external_invoice_id: { type: String, required: true },
  external_payment_id: { type: String, required: true },
});
```

Na podstawie OrderSchema utworzono model pozwalający w łatwy sposób wykonywać operację na kolekcji orders.

```
export const OrderMongoModel: Model<OrderDocument> = model<OrderDocument>(
  "orders",
  OrderSchema
);
```

Do poprawnego importu danych należało skorzystać z walidatora XML. W tym celu skorzystano z gotowej biblioteki ”xsd-schema-validator”, która zawiera funkcje odpowiedzialną za ustalenie zgodności wejściowego XML z XML Schema zapisanego w folderze resources. Funkcja ta po zwalidowaniu uruchamia funkcję zwrotną podaną w argumencie.

```
export const validateXML = async (xmlString: string) => {
  return new Promise(
    (resolve, reject) => {
      xsdSchemaValidator.validateXML(
        xmlString,
        xmlSchemaPath,
        (error, result) => {
          if (error || !result.valid) reject(error);
          else resolve(result);
        }
      );
    }
  );
};
```

Kolejną funkcją pomocniczą jest funkcja odpowiedzialna za parsowanie danych XML na obiekt JSON. W tym przypadku również skorzystano z gotwego rozwiązania jakie oferuje biblioteka fast-xml-parser.

```
export const parseXML = (xmlString: string): JSONObject => {
  const xmlParser = new XMLParser({
    ignoreAttributes: false,
    attributeNamePrefix: "",
  });
  const xmlData = xmlParser.parse(xmlString);
  return Array.isArray(xmlData.Orders.Order)
    ? xmlData.Orders.Order[0]
    : xmlData.Orders.Order;
};
```

Po sparsowaniu należy przekonwertować obiekt JSON na taki, aby był w pełni zgodny ze schematem Mongo. Do tego została zaimplementowana własna funkcja.

Wszystkie powyższe funkcje zostały użyte do utworzenia kontrolera dla punktu końcowego /POST /orders/append. Punkt końcowy przyjmuje żądanie i wyciąga z niego przesłany przez użytkownika XML. W dalszym etapie przesłany XML jest walidowany, parsowany i konwertowany na obiekt zgodny ze schematem Mongo. Ostatecznie tworzony jest dokument w bazie, a użytkownik otrzymuje w odpowiedzi utworzony dokument.

```
@Controller
 async append(req: Request, res: Response, next: NextFunction) {
   const { rawBody: xmlOrder } = req;
   if (!xmlOrder) throw new Error("request body is not valid");
   const xmlValidationResult = await validateXML(xmlOrder);
   if (!xmlValidationResult.valid) throw new Error("xml is not valid");

   const jsonOrder = parseXML(xmlOrder);
   const order: Order = convertToOrder(jsonOrder);

   const orderDocument = new OrderMongoModel(order);
   await orderDocument.save();

   res.status(201).json(success({ orderDocument }));
 }
```