Dzień 2: Praca z bazą danych - Część 2

Data: 03.02.2026 | **Godziny:** 14:00-17:00

Cele edukacyjne

Po ukończeniu tego modułu uczestnik będzie potrafił:

- [ ] Wyjaśnić zasady ACID i ich znaczenie dla integralności danych
- [ ] Prawidłowo używać `commit()` i `rollback()` do zarządzania transakcjami
- [ ] Wykonywać operacje INSERT, UPDATE, DELETE z obsługą błędów
- [ ] Używać `executemany()` do efektywnego wstawiania wielu rekordów
- [ ] Zdefiniować modele SQLAlchemy z relacjami
- [ ] Wykonywać operacje CRUD za pomocą SQLAlchemy ORM

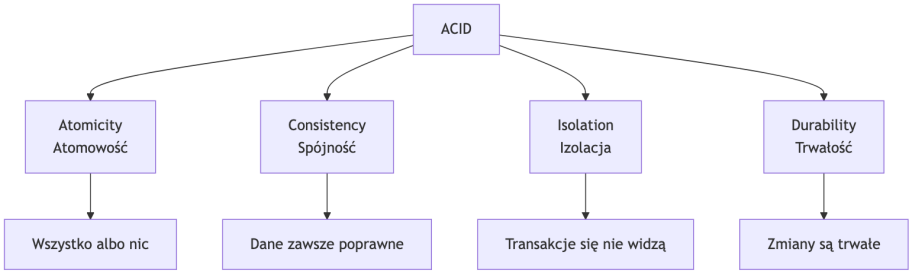
Wprowadzenie teoretyczne

Transakcje - fundament integralności danych

W branży ubezpieczeniowej integralność danych to podstawa. Wyobraź sobie scenariusz: klient kupuje polisę, system musi: 1. Zapisać dane polisy 2. Zaksięgować płatność 3. Wygenerować numer polisy 4. Wysłać potwierdzenie

Jeśli krok 3 się nie powiedzie, kroki 1-2 muszą zostać wycofane. Do tego służą transakcje.

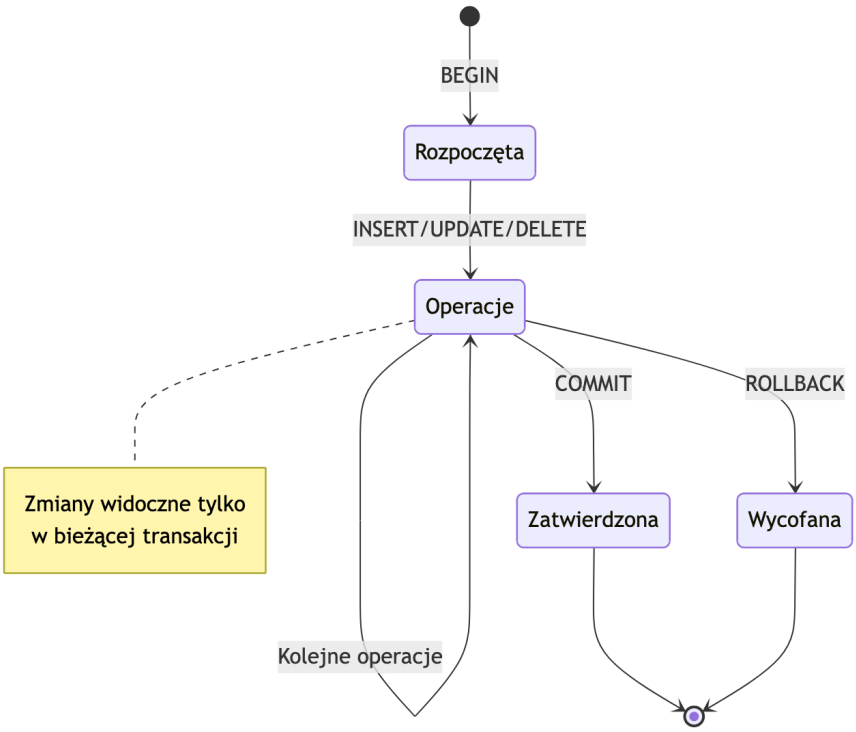
ACID - cztery filary transakcji



Diagram

Właściwość	Znaczenie	Przykład ubezpieczeniowy
Atomowość	Wszystkie operacje wykonują się lub żadna	Polisa + płatność = jedna operacja
Spójność	Baza zawsze w poprawnym stanie	Suma składek = suma na koncie
Izolacja	Transakcje nie widzą swoich zmian	Dwóch agentów nie sprzeda tej samej polisy
Trwałość	Po commit dane są bezpieczne	Awaria po zapisie nie usuwa danych

Cykl życia transakcji



Diagram

Biblioteki i narzędzia

sqlite3 - zarządzanie transakcjami

```python
import sqlite3

# Domyślnie: isolation_level = '' (autocommit wyłączony)
conn = sqlite3.connect('baza.db')

# Włączenie autocommit
conn.isolation_level = None

# Poziomy izolacji SQLite
# '' – domyślny (DEFERRED)
# 'DEFERRED' – blokada przy pierwszym zapisie
# 'IMMEDIATE' – natychmiastowa blokada zapisu
# 'EXCLUSIVE' – pełna blokada
conn.isolation_level = 'IMMEDIATE'
```

psycopg2 - transakcje w PostgreSQL

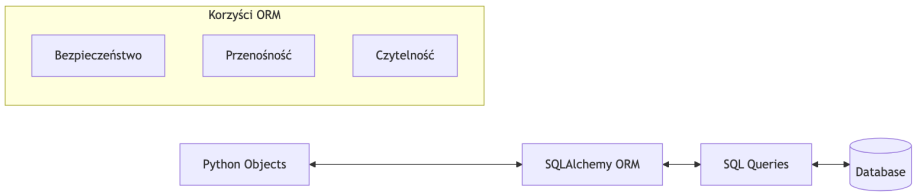
```python
import psycopg2

conn = psycopg2.connect(...)

# Autocommit
conn.autocommit = True

# Poziomy izolacji
from psycopg2.extensions import (
    ISOLATION_LEVEL_AUTOCOMMIT,
    ISOLATION_LEVEL_READ_COMMITTED,
    ISOLATION_LEVEL_REPEATABLE_READ,
    ISOLATION_LEVEL_SERIALIZABLE
)
conn.set_isolation_level(ISOLATION_LEVEL_SERIALIZABLE)
```

SQLAlchemy - ORM dla Pythona



Diagram

Przypadki użycia w ubezpieczeniach

Scenariusz 1: Sprzedaż polisy - transakcja złożona

```
import sqlite3
from datetime import datetime

def sprzedaj_polise(klient_id, typ_polisy, wartosc,
        metoda_platnosci):
    """
    Kompletny proces sprzedaży polisy.
    Wszystko albo nic - transakcja atomowa.
    """
    conn = sqlite3.connect('ergo_hestia.db')
    cursor = conn.cursor()

    try:
        # 1. Wygeneruj numer polisy
        cursor.execute("SELECT MAX(id) FROM polisy")
        max_id = cursor.fetchone()[0] or 0
        numer_polisy = f"POL-{datetime.now().year}-{max_id +
        1:05d}"

        # 2. Zapisz polisę
        cursor.execute("""
            INSERT INTO polisy (numer, typ, wartosc, klient_id,
        data_od, data_do)
            VALUES (?, ?, ?, ?, date('now'), date('now', '+1
        year'))
        """, (numer_polisy, typ_polisy, wartosc, klient_id))

        polisa_id = cursor.lastrowid

        # 3. Zapisz płatność
        cursor.execute("""
            INSERT INTO platnosci (polisa_id, kwota, data, metoda,
        status)
            VALUES (?, ?, datetime('now'), ?, 'ZAKSIEGOWANA')
        """, (polisa_id, wartosc, metoda_platnosci))

        # 4. Zaktualizuj historię klienta
        cursor.execute("""
            UPDATE klienci
            SET ostatnia_polisa = ?, liczba_polis = liczba_polis +
        1
            WHERE id = ?
        """, (numer_polisy, klient_id))

        # Wszystko OK - zatwierdź
        conn.commit()

        return {
            'sukces': True,
```

```
            'numer_polisy': numer_polisy,
            'polisa_id': polisa_id
        }

    except Exception as e:
        # Coś poszło nie tak - wycofaj wszystko
        conn.rollback()
        return {
            'sukces': False,
            'blad': str(e)
        }

    finally:
        conn.close()
```

Scenariusz 2: Batch import szkód z systemu zewnętrznego

```
import sqlite3
import csv

def importuj_szkody_batch(plik_csv, batch_size=100):
    """
    Import szkód w paczkach z punktami kontrolnymi (savepoints).
    Pozwala na częściowy sukces przy dużych importach.
    """
    conn = sqlite3.connect('ergo_hestia.db')
    cursor = conn.cursor()

    zaimportowane = 0
    bledy = []

    try:
        with open(plik_csv, 'r', encoding='utf-8') as f:
            reader = csv.DictReader(f)
            batch = []

            for i, row in enumerate(reader, 1):
                batch.append(row)

                if len(batch) >= batch_size:
                    # Savepoint przed każdą paczką
                    cursor.execute(f"SAVEPOINT batch_{i}")

                    try:
                        for szkoda in batch:
                            cursor.execute("""
                                INSERT INTO szkody
                                (numer_szkody, polisa_numer,
        data_zgloszenia, opis, kwota)
                                VALUES (?, ?, ?, ?, ?)
                            """, (
                                szkoda['numer'],
```

```
                            szkoda['polisa'],
                            szkoda['data'],
                            szkoda['opis'],
                            float(szkoda['kwota'])
                        ))
                    zaimportowane += len(batch)

                except Exception as e:
                    # Błąd w paczce - wycofaj tylko tę paczkę
                    cursor.execute(f"ROLLBACK TO SAVEPOINT
    batch_{i}")
                    bledy.append({
                        'batch': i,
                        'error': str(e),
                        'records': len(batch)
                    })

                batch = []

        # Ostatnia niepełna paczka
        if batch:
            for szkoda in batch:
                cursor.execute("""
                    INSERT INTO szkody
                    (numer_szkody, polisa_numer,
    data_zgloszenia, opis, kwota)
                    VALUES (?, ?, ?, ?, ?)
                """, (
                    szkoda['numer'],
                    szkoda['polisa'],
                    szkoda['data'],
                    szkoda['opis'],
                    float(szkoda['kwota'])
                ))
            zaimportowane += len(batch)

    conn.commit()

except Exception as e:
    conn.rollback()
    raise

finally:
    conn.close()

return {
    'zaimportowane': zaimportowane,
    'bledy': bledy
}
```

Scenariusz 3: Przeliczenie składek rocznych (SQLAlchemy)

```
from sqlalchemy import create_engine, func
from sqlalchemy.orm import sessionmaker
from models import Polisa, Klient

def przelicz_skladki_roczne(wspolczynnik_inflacji=1.03):
    """
    Coroczne przeliczenie składek z użyciem SQLAlchemy.
    """
    engine = create_engine('sqlite:///ergo_hestia.db')
    Session = sessionmaker(bind=engine)
    session = Session()

    try:
        # Pobierz aktywne polisy
        aktywne_polisy = session.query(Polisa).filter(
            Polisa.data_do >= func.date('now')
        ).all()

        zaktualizowane = 0
        for polisa in aktywne_polisy:
            stara_wartosc = polisa.wartosc
            polisa.wartosc = round(polisa.wartosc *
        wspolczynnik_inflacji, 2)

            # Logowanie zmiany
            print(f"Polisa {polisa.numer}: {stara_wartosc} ->
        {polisa.wartosc}")
            zaktualizowane += 1

        session.commit()
        return zaktualizowane

    except Exception as e:
        session.rollback()
        raise

    finally:
        session.close()
```

Przykłady kodu

Transakcje - podstawy

```
import sqlite3

# Sposób 1: Ręczne zarządzanie
conn = sqlite3.connect('baza.db')
cursor = conn.cursor()
```

```python
try:
    cursor.execute("INSERT INTO klienci (imie) VALUES (?)",
        ('Jan',))
    cursor.execute("INSERT INTO klienci (imie) VALUES (?)",
        ('Anna',))
    conn.commit()  # Zatwierdź obie operacje
except Exception as e:
    conn.rollback()  # Wycofaj wszystko
    print(f"Błąd: {e}")
finally:
    conn.close()

# Sposób 2: Context manager (zalecany)
with sqlite3.connect('baza.db') as conn:
    cursor = conn.cursor()
    cursor.execute("INSERT INTO klienci (imie) VALUES (?)",
        ('Piotr',))
    # Automatyczny commit na końcu bloku
    # Automatyczny rollback przy wyjątku
```

INSERT - pojedynczy i batch

```python
import sqlite3

# Pojedynczy INSERT
with sqlite3.connect('baza.db') as conn:
    cursor = conn.cursor()
    cursor.execute(
        "INSERT INTO klienci (imie, nazwisko) VALUES (?, ?)",
        ('Jan', 'Kowalski')
    )
    print(f"Wstawiono rekord z ID: {cursor.lastrowid}")

# Batch INSERT - executemany (znacznie szybszy!)
nowi_klienci = [
    ('Anna', 'Nowak'),
    ('Piotr', 'Wiśniewski'),
    ('Maria', 'Dąbrowska'),
]

with sqlite3.connect('baza.db') as conn:
    cursor = conn.cursor()
    cursor.executemany(
        "INSERT INTO klienci (imie, nazwisko) VALUES (?, ?)",
        nowi_klienci
    )
    print(f"Wstawiono {cursor.rowcount} rekordów")
```

UPDATE i DELETE

```
import sqlite3

with sqlite3.connect('ergo_hestia.db') as conn:
    cursor = conn.cursor()

    # UPDATE z warunkiem
    cursor.execute("""
        UPDATE polisy
        SET wartosc = wartosc * 1.05
        WHERE typ = ? AND data_do > date('now')
    """, ('OC',))
    print(f"Zaktualizowano {cursor.rowcount} polis")

    # DELETE z warunkiem
    cursor.execute("""
        DELETE FROM polisy
        WHERE data_do < date('now', '-5 years')
    """)
    print(f"Usunięto {cursor.rowcount} starych polis")
```

INSERT OR REPLACE (Upsert)

```
import sqlite3

with sqlite3.connect('baza.db') as conn:
    cursor = conn.cursor()

    # Wariant 1: INSERT OR REPLACE
    cursor.execute("""
        INSERT OR REPLACE INTO klienci (id, imie, nazwisko, email)
        VALUES (?, ?, ?, ?)
    """, (1, 'Jan', 'Kowalski', 'jan.nowy@email.pl'))

    # Wariant 2: ON CONFLICT (SQLite 3.24+)
    cursor.execute("""
        INSERT INTO klienci (imie, nazwisko, email)
        VALUES (?, ?, ?)
        ON CONFLICT(email) DO UPDATE SET
            imie = excluded.imie,
            nazwisko = excluded.nazwisko
    """, ('Jan', 'Kowalski', 'jan@email.pl'))
```

SQLAlchemy - definiowanie modeli

```
from sqlalchemy import create_engine, Column, Integer, String,
        Float, Date, ForeignKey
from sqlalchemy.orm import declarative_base, relationship,
        sessionmaker

Base = declarative_base()
```

```python
class Klient(Base):
    __tablename__ = 'klienci'

    id = Column(Integer, primary_key=True)
    imie = Column(String(50), nullable=False)
    nazwisko = Column(String(50), nullable=False)
    email = Column(String(100), unique=True)
    telefon = Column(String(20))

    # Relacja jeden-do-wielu z polisami
    polisy = relationship("Polisa", back_populates="klient",
                          cascade="all, delete-orphan")

    def __repr__(self):
        return f"<Klient {self.imie} {self.nazwisko}>"

    @property
    def pelne_imie(self):
        return f"{self.imie} {self.nazwisko}"


class Polisa(Base):
    __tablename__ = 'polisy'

    id = Column(Integer, primary_key=True)
    numer = Column(String(20), unique=True, nullable=False)
    typ = Column(String(10), nullable=False)
    wartosc = Column(Float, default=0.0)
    klient_id = Column(Integer, ForeignKey('klienci.id'))
    data_od = Column(Date)
    data_do = Column(Date)

    # Relacja do klienta
    klient = relationship("Klient", back_populates="polisy")

    def __repr__(self):
        return f"<Polisa {self.numer} ({self.typ})>"

    @property
    def jest_aktywna(self):
        from datetime import date
        return self.data_od <= date.today() <= self.data_do
```

SQLAlchemy - operacje CRUD

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

# Utworzenie silnika i sesji
engine = create_engine('sqlite:///ergo_hestia.db', echo=False)
Base.metadata.create_all(engine)
```

```
Session = sessionmaker(bind=engine)
session = Session()

# CREATE
nowy_klient = Klient(imie='Jan', nazwisko='Kowalski',
        email='jan@email.pl')
session.add(nowy_klient)
session.commit()

# READ
klient =
        session.query(Klient).filter_by(email='jan@email.pl').first()
wszyscy = session.query(Klient).all()
kowalscy =
        session.query(Klient).filter(Klient.nazwisko.like('Kowal%')).all()

# UPDATE
klient.telefon = '500600700'
session.commit()

# DELETE
session.delete(klient)
session.commit()

# Zawsze zamknij sesję
session.close()
```

SQLAlchemy - zapytania zaawansowane

```
from sqlalchemy import and_, or_, func, desc

# Złożone warunki
polisy = session.query(Polisa).filter(
    and_(
        Polisa.typ == 'OC',
        Polisa.wartosc > 500,
        Polisa.data_do >= func.date('now')
    )
).all()

# OR
klienci = session.query(Klient).filter(
    or_(
        Klient.imie == 'Jan',
        Klient.imie == 'Anna'
    )
).all()

# Sortowanie i limit
top_polisy = session.query(Polisa)\
    .order_by(desc(Polisa.wartosc))\
    .limit(10)\
    .all()
```

```python
# Agregacje
suma_oc = session.query(func.sum(Polisa.wartosc))\
    .filter(Polisa.typ == 'OC')\
    .scalar()

# Grupowanie
statystyki = session.query(
    Polisa.typ,
    func.count(Polisa.id).label('liczba'),
    func.sum(Polisa.wartosc).label('suma')
).group_by(Polisa.typ).all()

# JOIN przez relację (automatyczny)
klient = session.query(Klient).filter_by(id=1).first()
for polisa in klient.polisy:  # Lazy loading
    print(polisa.numer)

# JOIN jawny
from sqlalchemy.orm import joinedload
klient = session.query(Klient)\
    .options(joinedload(Klient.polisy))\
    .filter_by(id=1)\
    .first()  # Eager loading - jeden query
```

Częste błędy i antywzorce

1. Brak obsługi transakcji

```python
# ZŁE - brak rollback przy błędzie
def zapisz_dane_zle(dane):
    conn = sqlite3.connect('baza.db')
    cursor = conn.cursor()
    cursor.execute("INSERT INTO tabela VALUES (?)", dane)
    conn.commit()  # Co jeśli commit się nie powiedzie?
    conn.close()

# DOBRE - pełna obsługa transakcji
def zapisz_dane_dobrze(dane):
    conn = sqlite3.connect('baza.db')
    try:
        cursor = conn.cursor()
        cursor.execute("INSERT INTO tabela VALUES (?)", dane)
        conn.commit()
    except Exception as e:
        conn.rollback()
        raise
    finally:
        conn.close()
```

```python
# NAJLEPSZE – context manager
def zapisz_dane_najlepiej(dane):
    with sqlite3.connect('baza.db') as conn:
        cursor = conn.cursor()
        cursor.execute("INSERT INTO tabela VALUES (?)", dane)
```

2. Wiele pojedynczych INSERT zamiast batch

```python
# ZŁE – bardzo wolne (1000 commitów!)
for klient in lista_1000_klientow:
    cursor.execute("INSERT INTO klienci VALUES (?)", (klient,))
    conn.commit()

# DOBRE – jeden commit
for klient in lista_1000_klientow:
    cursor.execute("INSERT INTO klienci VALUES (?)", (klient,))
conn.commit()

# NAJLEPSZE – executemany
cursor.executemany("INSERT INTO klienci VALUES (?)",
                   [(k,) for k in lista_1000_klientow])
conn.commit()
```

3. Nieprawidłowe użycie sesji SQLAlchemy

```python
# ZŁE – nigdy nie zamykamy sesji
def pobierz_klienta_zle(id):
    session = Session()
    return session.query(Klient).filter_by(id=id).first()
    # Session leak!

# DOBRE – zawsze zamykaj sesję
def pobierz_klienta_dobrze(id):
    session = Session()
    try:
        return session.query(Klient).filter_by(id=id).first()
    finally:
        session.close()

# NAJLEPSZE – context manager
from contextlib import contextmanager

@contextmanager
def get_session():
    session = Session()
    try:
        yield session
        session.commit()
    except:
        session.rollback()
        raise
    finally:
```

```
        session.close()

def pobierz_klienta_najlepiej(id):
    with get_session() as session:
        return session.query(Klient).filter_by(id=id).first()
```

4. N+1 Problem w ORM

```
# ZŁE - N+1 queries
klienci = session.query(Klient).all()  # 1 query
for klient in klienci:
    print(klient.polisy)  # N queries (lazy loading)

# DOBRE - eager loading
from sqlalchemy.orm import joinedload

klienci = session.query(Klient)\
    .options(joinedload(Klient.polisy))\
    .all()  # 1 query z JOIN

for klient in klienci:
    print(klient.polisy)  # Dane już załadowane
```

Podsumowanie dnia

Wzorzec do zapamiętania

```
# Raw SQL z transakcją
with sqlite3.connect('baza.db') as conn:
    cursor = conn.cursor()
    try:
        cursor.execute("INSERT ...")
        cursor.execute("UPDATE ...")
        # Auto-commit na końcu with
    except Exception:
        conn.rollback()
        raise

# SQLAlchemy ORM
with get_session() as session:
    klient = Klient(imie='Jan', nazwisko='Kowalski')
    session.add(klient)
    # Auto-commit i close przez context manager
```

Checklista

Po dzisiejszych zajęciach powinieneś umieć:

- [ ] Wyjaśnić zasady ACID
- [ ] Używać `commit()` i `rollback()` do zarządzania transakcjami
- [ ] Stosować context manager dla bezpiecznych operacji
- [ ] Wykonywać INSERT, UPDATE, DELETE z parametryzacją
- [ ] Używać `executemany()` do batch operations
- [ ] Definiować modele SQLAlchemy z kolumnami i relacjami
- [ ] Wykonywać CRUD w SQLAlchemy
- [ ] Stosować filtry i agregacje w SQLAlchemy

Przygotowanie do następnych zajęć

Do zrobienia przed Dniem 3:

1. Zainstaluj NumPy: `pip install numpy`
2. Przejrzyj dokumentację: https://numpy.org/doc/

Czym zajmiemy się na Dniu 3:

- NumPy - podstawy obliczeń numerycznych
- Tablice wielowymiarowe
- Operacje wektorowe

Materiały dodatkowe

Linki

- SQLAlchemy Documentation
- SQLite Transactions
- ACID Properties

Cheat Sheet: Transakcje i SQLAlchemy

```
TRANSAKCJE (raw SQL)
--------------------
conn.commit()        # Zatwierdź zmiany
conn.rollback()      # Wycofaj zmiany
conn.autocommit      # Tryb auto-commit
SAVEPOINT nazwa      # Punkt kontrolny
ROLLBACK TO nazwa    # Wycofaj do punktu

SQLALCHEMY - SILNIK
-------------------
engine = create_engine('sqlite:///baza.db')
engine = create_engine('postgresql://user:pass@host/db')

SQLALCHEMY - MODEL
------------------
class Model(Base):
    __tablename__ = 'tabela'
    id = Column(Integer, primary_key=True)
    nazwa = Column(String(50), nullable=False)
    fk = Column(Integer, ForeignKey('inna.id'))
    relacja = relationship("Inna", back_populates="...")

SQLALCHEMY - SESJA
------------------
Session = sessionmaker(bind=engine)
session = Session()
session.add(obj)
session.add_all([obj1, obj2])
session.commit()
session.rollback()
session.close()

SQLALCHEMY - QUERY
------------------
session.query(Model).all()
session.query(Model).first()
session.query(Model).filter_by(id=1)
session.query(Model).filter(Model.id == 1)
session.query(Model).order_by(Model.nazwa)
session.query(func.count(Model.id))
```