# ZODB – obiektowa baza danych w Pythonie

OPIS ŚRODOWISKA, ASPEKT OBIEKTOWY, METODY, OPERACJE CRUD

BOCAK MATEUSZ
GABRIELA BIENIEK

# Wprowadzenie do ZODB

- Obiektowa baza danych dla Pythona – przechowuje obiekty bez użycia SQL
- Zmiany w obiektach są buforowane i zapisywane tylko przy `commit` (zapewnia ACID)
- Dawniej „Zope Object Database”, dziś samodzielny projekt („Z Object DB” niezależny od Zope)
- Instalacja: `pip install ZODB` (Python 2.7, 3.4+; dostępne też na PyPI)

# Zalety ZODB

- Brak oddzielnego języka zapytań – operujemy kodem Pythona bezpośrednio na obiektach
- Przejrzysta integracja z kodem - obiekty zachowują się jak zwykłe obiekty Pythona, wystarczy dziedziczyć po klasie Persistent
- Naturalne relacje - obiekty odnoszą się bezpośrednio do innych obiektów (brak złożonych JOIN-ów)
- Transakcje ACID (atomowość, spójność, izolacja, trwałość) – wszystkie zmiany zatwierdzane są razem (lub wycofywane)
- Dzięki lokalnej pamięci podręcznej (cache) odczyty są szybkie, zwłaszcza w architekturze klient-serwer (ZEO)

# Historia i tło

- ZODB powstał jako część frameworka Zope (stąd pierwotna nazwa) i był rozwijany przez społeczność Zope
- Obecnie działa niezależnie od Zope, używany samodzielnie w wielu projektach Pythonowych
- Wersje - kompatybilny z Python 2.7 oraz 3.4 i nowszymi, dostępny także dla PyPy
- Aktywnie rozwijany przez społeczność – repozytoria (ZODB, persistent, transaction itp.) na GitHub, lista dyskusyjna i dokumentacja online

# Instalacja i konfiguracja

- Instalacja – `pip install ZODB` – instaluje ZODB oraz związane pakiety (`persistent, transaction, BTrees`)
- Wymagania – Python ≥ 2.7/3.4+, kompilator C do modułów rozszerzonych (lub gotowe binarki, zwłaszcza na Windows)
- Po instalacji można tworzyć bazę plikową (FileStorage) lub stosować bazę w pamięci (`DB(None)`) dla testów
- Konfiguracja opcjonalna – plik konfiguracyjny ZODB.conf lub konfiguracja w kodzie Pythona (określenie storage, ZEO, RelStorage itp.)

# Podstawowe pojęcia: obiekty trwałe

- Klasa persistent.Persistent – dziedzicząc po niej, obiekt staje się „śledzony” przez ZODB (zmiany ustawiają `_p_changed=True`)
- Obiekty są „leniwo” wczytywane z dysku – przy pierwszym dostępie pobierane z magazynu, dalej trzymane w pamięci podręcznej
- *Root* bazy – specjalny obiekt działający jak słownik (mapa) – punkt wejścia do przechowywanych danych
- Wystarczy przypisywać obiekty do `root[…]`, aby stały się trwałe (np. `root['konto']=konto`)
- Wszystkie obiekty muszą być picklowane – listy, słowniki itp. działają, ale otwarte pliki, sockety czy obiekty kodu nie mogą być zapisane

# Przykład użycia (kod Python)

KOD: UTWORZENIE BAZY I POŁĄCZENIA (FILESTORAGE)

```
from ZODB import DB, FileStorage

import transaction

storage = FileStorage.FileStorage('data.fs’)

db = DB(storage)

conn = db.open()

root = conn.root()
```

KOD: DEFINICJA KLASY I DODANIE OBIEKTU DO BAZY

```
from persistent import Persistent

class Osoba(Persistent):

    def __init__(self, imie):

        self.imie = imie

root['jan'] = Osoba('Jan Kowalski')
```

# Mutowalne atrybuty i trwałość

- ZODB nie wykrywa automatycznie zmian wewnątrz zwykłych mutowalnych typów (np. zmiana listy jako `lista.append(x)` )
- Rozwiązania – używać typów trwałych (`PersistentList, PersistentMapping` lub BTrees) lub ręcznie oznaczać zmianę: `obj._p_change = True`
- Przykład – jeśli `self.lista` to zwykła lista, po modyfikacji należy ustawić `self._p_changed = True`, aby ZODB zapisał zmiany
- Lepsze praktyki – wszystkie duże lub często zmieniane kolekcje trzymać w trwałych strukturach (z modułu `persistent` lub `BTrees`)

# Obiektowość w ZODB i porównanie z innymi OODB

- Pełne przechowywanie obiektów Pythona – ZODB zapisuje instancje klas bezpośrednio, z zachowaniem stanu i relacji referencyjnych
- Brak mapowania do schematów czy tabel – nie ma potrzeby definicji osobnych struktur danych –wystarczy klasa dziedzicząca po `Persistent`
- Transparentne zarządzanie relacjami – referencje między obiektami są zapisywane automatycznie (bez JOINów czy kluczy obcych)
- Porównanie z db4o / Object DB
  - db4o (Java/.NET) i ObjectDB (Java) wymagają generowania plików schematu lub konfiguracji klas, zewnętrznych narzędzi do migracji danych
  - ZODB działa „naturalnie” w Pythonie – bez dodatkowych narzędzi czy konfiguratorów

# Tworzenie i użycie metod obiektów

- Metody jak w standardowej klasie Pythona – tworzenie w klasie dziedziczącej po `Persistent` (bez dodatkowych dekoratorów czy zależności)
- Logika i trwałość w jednym miejscu – kod metody (np. `def zmniejsz_cene(self, pct):`) modyfikuje atrybuty obiektu – po transaction.commit zmiany są zapisane na dysku
- Automatyczne śledzenie zmian – gdy metoda zmieni atrybut obiektu, ZODB ustawia `_p_changed = True` i uwzględnia modyfikację przy commitowaniu
- Mutowalne struktury – jeśli modyfikacja dotyczy listy lub słownika, lepiej skorzystać z `PersistentList/PersistentMapping` lub `BTrees` (eliminacja potrzeby ręcznego oznaczania zmiany)

```
class Produkt(Persistent):
    def zmniejsz_cene(self, procent):
        self.price *= (1 - procent/100)
```

# Transakcje

- Zmiany w bazie są grupowane w transakcje; trzeba je jawnie wykonać `transaction.commit()` żeby trwale zapisać zmodyfikowane zasoby
- Jeśli zmiany mają zostać odrzucone, można wywołać `transaction.abort()` – stan bazy wraca do momentu przed transakcją
- Właściwości ACID – wszystkie zmiany w transakcji zostaną zapisane albo żadne – chroni to integralność danych przy błędach
- ZODB wspiera również dwufazowy commit (2PC) – transakcję można koordynować między ZODB a innymi zasobami (np. relacyjnymi bazami danych)

# ZEO (klient-serwer)

- ZEO (Zope Enterprise Objects) to serwer TCP/IP dla ZODB – pozwala wielu procesom współdzielić jedną wspólną bazę
- Każdy klient tworzy `ClientStorage`, który buforuje obiekty lokalnie i komunikuje się z serwerem (magazyn serwera to np. FileStorage)
- Przy zapisie klienta serwer wysyła do pozostałych powiadomienie (invalidate) z listą zmodyfikowanych obiektów – powoduje to odświeżenie ich w lokalnych cache
- Architektura klient-serwer najbardziej efektywna przy dominującym odczycie – lokalny cache znacząco przyśpiesza operacje odczytu (ale przy intensywnym zapisywaniu może pojawić się nadmiar komunikatów invalidate)

# BTrees

- BTrees to trwałe, zrównoważone drzewa przechowujące pary klucz-wartość – zoptymalizowane do bardzo dużych zbiorów (więcej danych niż RAM)
- Typy: `OOBTree` (dowolne typy kluczy i wartości), `IOBTree` (klucze int, wartości obiekty), `OIBTree` ( opt. dla int-int)
- Zaletami BTrees są szybkie wyszukiwanie (O(log N)) i integracja z cache ZODB – często używane elementy są trzymane w pamięci dla szybkości
- Przykład użycia:

```
from BTrees.OOBTree import OOBTree

root.dane = OOBTree()

root.dane['liczba'] = 42
```

# Indeksowanie i wyszukiwanie

- ZODB nie ma wbudowanej „silni” zapytań jak SQL – odwołujemy się bezpośrednio do obiektów i ich atrybutów (przechodzimy przez strukturę obiektów)
- Jeżeli potrzebujemy wyszukiwania według klucza, można samodzielnie zbudować indeksy: np. `index = OOBTree()`, gdzie klucz to wartość atrybutu, a wartość to lista referencji do obiektów
- W Zope istnieje „ZCatalog” do indeksowania treści, ale czysty ZODB wymaga ręcznej obsługi – można użyć zewnętrznych bibliotek (fulltext, Whoosh itp.)
- Uwaga:  dla aplikacji mocno opartych na złożonych zapytaniach (np. reporty, filtrowanie), może być efektywniejszy RDBMS lub dedykowana baza danych

# Magazyny danych

- `FileStorage`: domyślny magazyn zapisujący dane do pliku na dysku (zazwyczaj `.fs` i `.index`) – łatwy w użyciu, ale plik może rosnąć do dużych rozmiarów
- `MappingStorage`: magazyn w pamięci (zwracany przy `DB(None)`) – do testów, nie trwały (dane znikają po wyłączeniu programu)
- `RelStorage`: rozszerzenie zapisujące obiekty ZODB w relacyjnej bazie danych (PostgreSQL, MySQL, Oracle itp.)
- `RelStorage` pozwala stosować narzędzia RDBMS (backup, replikacja) do ochrony danych – obiekty są trzymane w tabelach jako pickle lub BLOB
- Inne magazyny: `ClientStorage` (klient ZEO), `BTreeFolderStorage` (Zope), można tworzyć własne przez implementację interfejsu `IStorage`

# Konfilkty i rozwiązywanie

- ZODB stosuje optymistyczną kontrolę współbieżności – konflikt zachodzi, gdy dwa procesy/wiele wątków próbują zapisać ten sam obiekt jednocześnie
- Domyślne zachowanie – jedna z transakcji zgłasza `ConflitError` i jest wycofywana, zachęcając do ponowienia operacji
- Można napisać własną metodę `_p_resolveConflict(self, oldState, savedState, newState)` , która łączy zmiany z dwóch operacji (np. sumowanie liczników)
- W Zope/Transakcjach domyślnie próbuje się powtórzyć konfliktującą operację do 3 razy (jeżeli konflikt się potwórzy, rzucony jest błąd)

# Porównanie: pickle / shelve

- `pickle` – moduł Pythona do serializacji pojedyńczego obiektu do bajtów. Brak transakcji – programista sam zapisuje i odczytuje plik
- `shelve` – moduł standardowy implementujący prosty magazyn klucz-wartość (używa pickle i bazy dbm). Działa jak dyskowy słownik, ale wymaga ręcznego otwierania/zamykania pliku i nie obsługuje współbieżnych zasobów
- W pickle/shelve programista musi sam zarządzać odczytem i zapisem (np. `open(), close()),` nie ma izolacji transakcji i bufora
- ZODB w przeciwieństwie do shelve – automatycznie zarządza cache'owaniem i transakcjami – `transaction.commit()` zapisuje całą grupę zmian naraz
- Dodatkowo – ZEO umożliwa wielosesyjne użycie ZODB, czego pickle/shelve nie obsługują

# Porównanie: relacyjne bazy/ORM

- Relacyjne bazy SQL – silne ACID, język SQL, wymagana normalizacja danych i z góry zdefiniowany schemat tabel
- ORM (np. SQLAlchemy, DjangoORM) – umożliwiają pracę na obiektach, ale pod spodem generują zapytania SQL i wymagają mapowania klasy ↔ tabela
- ZODB – nie wymaga osobnego schematu, struktura bazy wynika bezpośrednio z klas i atrybutów w kodzie
- Relacje – w ZODB obiekty wskazują na inne obiekty bezpośrednio (normalnie realizowane jest to przez referencje). Nie ma konieczności złożonych joinów czy relacyjnych tabel pośrednich
- Zaleta – naturalny model hierarchiczny i obiektowy (brak niezbędnych rekurencyjnych JOIN-ów)
- Wada – brak wygodnych zapytań typu SQL - trzeba traversować obiekty lub budować własne indeksy

# Wydajność i skalowalność

- Buforowanie w pamięci – ZODB trzyma ostatnio używane obiekty w cache każdego połączenia, co przyspiesza ponowne odczyty tego samego obiektu
- W architekturze ZEO każdy klient ma swój lokalny cache – znacznie przyspiesza odczyty w wielu procesach (ale intensywne zapisy generują dodatkowe komunikaty invalidate)
- BTrees – pozwalają efektywnie przechowywać duże mapy – operacje O(log N) dzięki drzewiastej strukturze
- Funkcja `pack()` – po pewnym czasie baza rośnie (trzyma stare wersje obiektów); `db.pack()` usuwa przestarzałe wersje (odzyskuje miejsce na dysku)
- Przy dużej liczbie jednoczesnych zapisów wydajność może spaść (koszty synchronizacji cache) – zaleca się unikać „gorących” obiektów/mutexów.

# Ograniczenia i dobre praktyki

- Zawsze wywołuj `transaction.commit()` – bez zatwierdzenia zmiany nie zostaną zapisane na dysku
- Nie współdziel połączeń między wątkami – każdy wątek (lub proces) powinien używać własnej instancji `Connection` (np. `db.open()`)
- Używaj klas dziedziczących po `persistent.Persistent` oraz trwałych typów BTrees/PersistentList/PersistentMapping – zapewnia wykrywanie zmian
- Po edycji mutowalnych atrybutów (listy, słowniki) oznacz obiekt jako zmieniony: `oj._p_changed = True` (jeśli nie używasz trwałych kontenerów)
- Regularnie wykonuj `db.pack()` lub narzędzie takie jak `zodbpack`, aby usunąć nieużywane rewizje i kontrolować rozmiar pliku
- Unikaj przechowywania dużych danych binarnych bezpośrednio w ZODB (lepiej system plików lub RelStorage z BLOB).

# CRUD w ZODB

- Tworzenie (`Create`) – Utworzenie instancji klasy dziedziczącej po `persistent.Persistent`, dodanie jej do trwałego kontenera (np. `PersistentMapping` lub `BTrees`) na obiekcie root (np. `root[,klucz’] = obiekt`) i zatwierdzenie transakcji przez `transaction.commit()`
- Odczyt (`Read`) – pobranie obiektu z kontenera jak ze słownika `obj – root[,klucz’]`. Obiekt jest już trwały – sam odczyt nie wymaga dodatkowego commita (dostęp nic nie zmienia)
- Aktualizacja (`Update`) – modyfikacja atrybutu obiektu lub elementu kontenera (np. `obj.atr = wartość`). Następnie wykonanie `transaction.commit()` dla zapisu. Jeśli jest zmiana mutowalnego atrybutu (lista, dict) należy ręcznie ustawić `obj._p_changed = True` po modyfikacji lub – wygodniej – użyć `PersistentMapping/BTrees` (same rejestrują zmiany)
- Usuwanie (`Delete`) – usunięcie obiektu z kontenera (np. `del root[,klucz’]` ) i wykonanie `transaction.commit().` Operacja jest transakcyjna – po commicie obiekt znika trwale z bazy

# CRUD w ZODB

- Transakcje - ZODB stosuje transakcje ACID. Wywołanie transaction.commit() zatwierdza wszystkie zmiany w bieżącej transakcji (stają się trwałe), a transaction.abort() cofa zmiany z tej transakcji (przywracając poprzedni stan obiektów). Po abort _p_changed obiektów zmienia się na None, a po commit na False.
  Dzięki transakcjom operacje CRUD są atomowe.

# CRUD w ZODB

```
# CREATE – tworzenie i commit
root['persons'] = PersistentMapping()
root['persons']['p1'] = Person('Alice')
transaction.commit()

# READ – odczyt obiektu
p = root['persons']['p1']
print(p.name)  # Alice
```

```
# UPDATE – modyfikacja atrybutu i commit
p.name = 'Alice Smith'
transaction.commit()

# DELETE – usunięcie i commit
del root['persons']['p1']
transaction.commit()
```

# Przykład użycia

- ZODB jest szeroko stosowany w systemach CMS (np. Plone, Zope) – naturalny do hierarchicznej treści i obiektów dynamicznych.
- Przykład integracji – repoze.zodbconn – middleware pozwalający używać ZODB w dowolnej aplikacji WSGI (np. repoze.tm2)
- Przydatny do przechowywania stanu aplikacji, konfiguracji, sesji użytkowników itp., gdzie ważne jest przechowywanie obiektów Pythona
- Elastyczność modelu – łatwo rozbudować dane (zmiana klasy/atrybutu bez migracji schematu)

# Zasoby i narzędzia

- Dokumentacja – oficjalna strona i tutoriale ZODB (zodb.org), artykuły (np. Michel Pelletier, Jim Fulton)
- Narzędzia – zodb (kompresja bazy), fsrefs (sprawdzanie spójności struktur), zodbinfo/zodbdump
- Repozytoria – kod źródłowy ZODB, persistent, BTrees, transaction na GitHub (ZopeFoundation)
- Wsparcie – lista dyskusyjna ZODB, forum Zope, GitHub issues. Wiele przykładów i FAQ dostępnych online.

# Podsumowanie

- ZODB to pełnoprawna obiektowa baza danych w Pythonie, oferująca przezroczystą trwałość obiektów i transakcje ACID
- Ułatwia modelowanie złożonych struktur obiektowych, ale wymaga innego podejścia niż relacyjne bazy (brak SQL, bezpośrednie odwołania między obiektami)
- Kluczowe koncepcje – obiekty Persistent, transkacje (commit/abort), struktury BTrees, współdzielenie przez ZEO
- Dobre praktyki – korzystaj z trwałych typów kolekcji, stosuj transakcje, regularnie packuj bazę, testuj scenariusze konfliktów