

Hibernate i JPA

Wrocław, 04.04.2016,
Jakub Karbowski
Justyna Bytner

Agenda

- Relacyjne bazy danych a świat obiektowy
- Dostęp do danych za pomocą JDBC
- Odwzorowanie obiektowo-relacyjne
 - Hibernate
 - Mapowanie JPA
- Transakcje



Zadanie: Instalacja środowiska

- ☐ Pobrać projekt z GitHuba

<https://github.com/jkarbowiak/jpa-training>

- ☐ Można skorzystać z komendy **git clone**

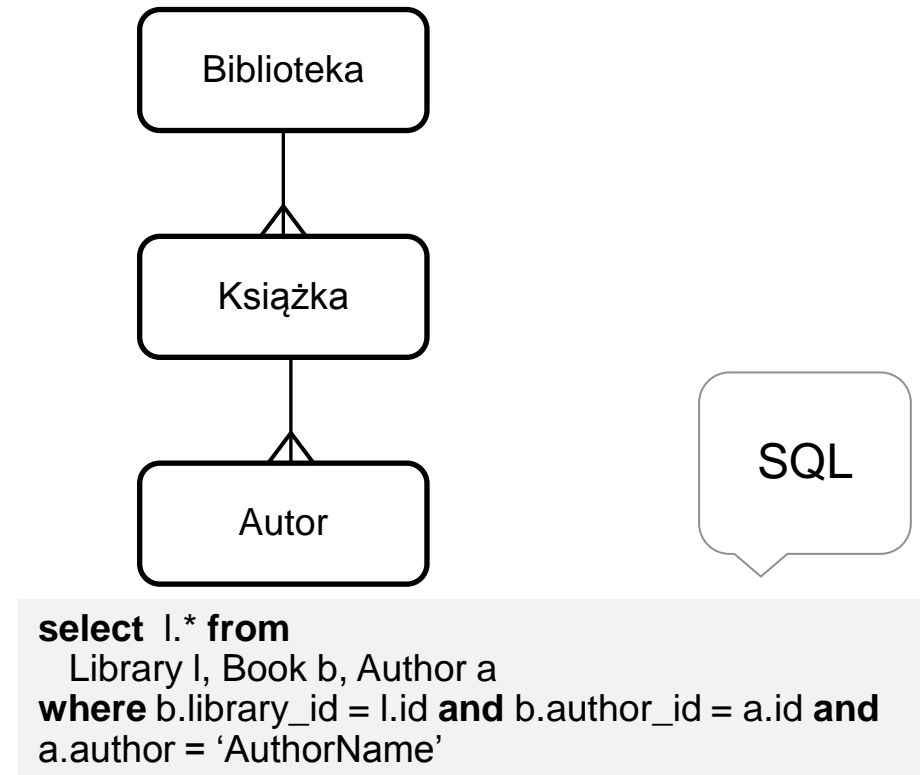
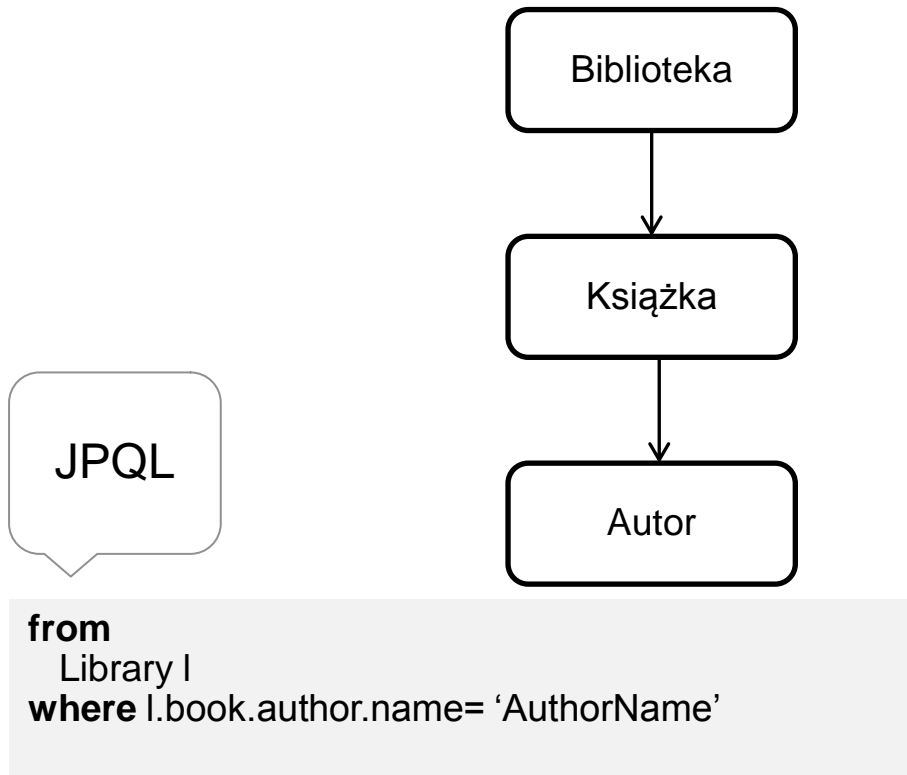
<https://github.com/jkarbowiak/jpa-training.git>

- ☐ Uruchomić komendę **mvn clean install** w głównym folderze projektu

- ☐ Zaimportować projekt do IDE

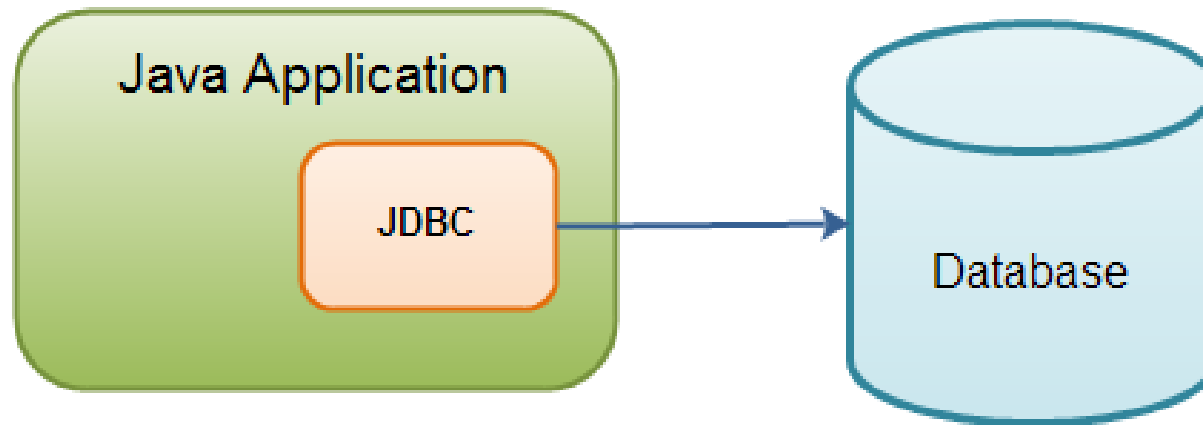
Relacyjne bazy danych a świat obiektowy

- ❑ Aplikacje tworzą, zarządzają i przechowują informacje strukturalne
- ❑ Programiści wybierają języki obiektowe
- ❑ Zapisywanie i pobieranie danych wymaga znajomości SQL



Dostęp do danych za pomocą JDBC

- ☐ Bardzo dobrze znane API
- ☐ Wymaga dobrej znajomości SQL
- ☐ Łączy do baz danych dla języka Java



JDBC – jak to działa

```
@Service
public class LibraryServiceImpl {

    private static final String FIND_ALL_LIBRARIES_IN_CITY_SQL =
        "select l.id, l.name, l.address_id from Library l, Address a where l.address_id = a.id and a.city = :city";

    @Autowired
    private NamedParameterJdbcOperations jdbcTemplate;
    @Autowired
    private LibraryRowMapper libraryRowMapper;

    public List<LibraryTo> findAllLibrariesInCity(String cityName) {
        SqlParameterSource params = new MapSqlParameterSource("city", cityName);
        return jdbcTemplate.query(FIND_ALL_LIBRARIES_IN_CITY_SQL, params, libraryRowMapper);
    }
}
```

- ❑ Konieczność definiowania zapytań SQL
- ❑ Spring *NamedParameterJdbcOperations* do wykonywania instrukcji SQL
- ❑ Spring *RowMapper* do mapowania wyniku zapytania na obiekt

JDBC - wady

- ☐ Wymaga pisania dużej ilości dodatkowego kodu
- ☐ Trudne mapowanie wyniku SQL na obiekty biznesowe
- ☐ Kod związany z trwałością danych narażony na błędy programistów
- ☐ Brak przenośności kodu, implementacja zależna od bazy danych
- ☐ Implementacja bardzo trudna w utrzymaniu
- ☐ Ewentualne błędy w zapytaniach SQL widoczne dopiero w trakcie działania programu
- ☐ Kod nietestowalny

Odwzorowanie obiektowo-relacyjne

- ☐ Przekształcenie obiektów w encje bazy danych i odwrotnie
- ☐ Przekształcenie połączeń między obiektami na relacje bazy danych
- ☐ Przekształcenie obiektowego języka zapytań na SQL
- ☐ Spójny sposób obsługi różnych baz danych - przenośność
- ☐ Zapewnienie trwałości obiektów
- ☐ Ochrona programisty przed czasochłonnym SQL-em
- ☐ Eliminuje większość żmudnych zadań związanych z trwałością
 - pozwala skupić się na implementacji logiki biznesowej
- ☐ Zapewnienie stałych technik optymalizacyjnych
- ☐ Oddzielenie warstwy dostępu do danych od warstwy biznesu

Odwzorowanie obiektowo-relacyjne

❑ Niedopasowanie paradygmatów relacyjno-obiektowych

Podobieństwa	Różnice
<ul style="list-style-type: none">• Klasy i tabele• Właściwości i kolumny• Instancje i wiersze	<ul style="list-style-type: none">• Szczegółowość• Dziedziczenie (java)• Kolekcje (java)• Identyczność (equals vs PK)• Nawigacja po grafie obiektów

Podstawowym zadaniem ORM jest rozwiązanie wrodzonych niezgodności pomiędzy obiektami i bazami danych

Hibernate

- ❑ Gavin King rozpoczął prace nad biblioteką pod koniec 2001 roku
- ❑ Zespół na bieżąco realizował prośby użytkowników
- ❑ Zgodny ze standardem JPA
 - Możliwość traktowania Hibernate jako dostawcy trwałości
 - Możliwość używania bibliotek JPA, takich jak np. Spring-Data-Jpa
- ❑ Oddziela kod biznesowy od warstwy dostępu do danych

```
@Service
@Transactional(readOnly = true)
public class LibraryServiceImpl {

    @Autowired
    private LibraryRepository libraryRepository;
    @Autowired
    private MapperFacade mapper;

    public List<LibraryTo> findAllLibrariesByName(String name) {
        List<LibraryEntity> libraries = libraryRepository.findByName(name);
        return mapper.mapAsList(libraries, LibraryTo.class);
    }
}
```

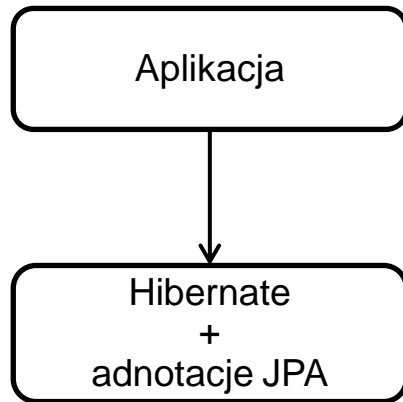
```
public interface LibraryRepository extends JpaRepository<LibraryEntity, Long> {

    @Query("from LibraryEntity l where l.name like :name")
    List<LibraryEntity> findByName(@Param("name") String name);

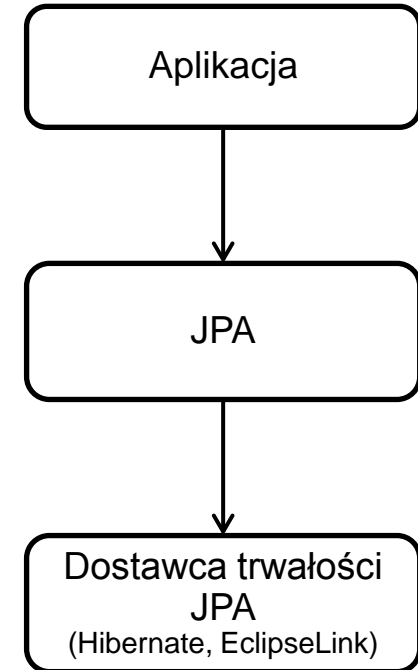
    @Query("from LibraryEntity l where l.address.city = :cityName")
    List<LibraryEntity> findByCity(@Param("cityName") String cityName);
}
```

Hibernate i JPA

Hibernate vs JPA



- ☐ SessionFactory
- ☐ Session
- ☐ Hibernate Query



- ☐ EntityManagerFactory
- ☐ EntityManager
- ☐ JPA Query

JPA konfiguracja

```
<jpa:repositories base-package="pl.spring.demo.repository" />

<util:properties id="hsqldbJpaProps" location="classpath:/test_data_access/hsqldb_jpa.properties"/>

<tx:annotation-driven transaction-manager="transactionManager"/>

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<jdbc:embedded-database id="dataSource" type="HSQL" />

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean" depends-on="transactionManager">
  <property name="persistenceUnitName" value="hsql"/>
  <property name="dataSource" ref="dataSource"/>
  <property name="jpaProperties" ref="hsqldbJpaProps"/>
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
  </property>
  <property name="packagesToScan">
    <array>
      <value>pl.spring.demo.entity</value>
    </array>
  </property>
</bean>
```

```
hibernate.ejb.naming_strategy org.hibernate.cfg.ImprovedNamingStrategy
hibernate.dialect org.hibernate.dialect.HSQLDialect
hibernate.show_sql true
hibernate.format_sql true
hibernate.hbm2ddl.auto create
```

Encje

```
public class LibraryEntity {  
    private String name;  
    private String domain;  
  
    public LibraryEntity () {  
    }  
  
    public String getName() { return name; }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getDomain() { return domain; }  
  
    public void setDomain(String domain) {  
        this.domain = domain;  
    }  
}
```

Zwykła klasa
POJO

```
@Entity  
public class LibraryEntity {  
    @Id  
    private Long id;  
    @Column(name = „name”, length = 30, nullable = false)  
    private String name;  
    @Column(name = „domain”, length = 5, nullable = true)  
    private String domain;  
  
    public LibraryEntity () {  
    }  
  
    // getters and setters  
}
```

Encja

Adnotacje Encji

```
@Entity
@Table(name = "LIBRARY", schema = "public")
@Access(AccessType.FIELD)
public class LibraryEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = „name”, length = 30, nullable = false)
    private String name;
    @Lob
    @Column(nullable = false)
    private String description;
    private String city;

    public LibraryEntity () {
    }
    @Access(AccessType.PROPERTY)
    public String getCity () {
    }
}
```

- ❑ **@Entity** - oznacza klasę, odzwierciedla tabelę BD
- ❑ **@Entity(name="...")** – nigdy nie używać
- ❑ **@Access** – decyduje gdzie umieszczać adnotacje
- ❑ **@Table** – zmienia domyślną nazwę tabeli BD
- ❑ **@Id** – oznacza atrybut jako klucz główny
- ❑ **@GeneratedValue** – auto generacja wartości PK
- ❑ **@Column** – pozwala ustawić wartości kolumny
- ❑ **@Lob** – typ dla dużych danych tekstowych
- ❑ **@Enumerated(EnumType.STRING)** – enumy
- ❑ **@Transient** – wyłączenie pola z persystencji
- ❑ **@MappedSuperclas** – zwykłe dziedziczenie

Relacje

- ☐ **@OneToOne** – encja A może mieć relację do dokładnie jednej encji B
- ☐ **@OneToMany** – encja A może mieć relację do kilku encji B
- ☐ **@ManyToOne** – wiele encji A może mieć relację do dokładnie jednej encji B
- ☐ **@ManyToMany** – wiele encji A może mieć relację do wielu encji B
 - Tworzy tabelę asocjacyjną

Wyróżniamy relacje:

- ☐ jednokierunkowe
- ☐ dwukierunkowe

Relacja @OneToOne

Jednokierunkowa

```
@Entity
public class User {


    @OneToOne(
        cascade = CascadeType.ALL, // default: empty
        fetch = FetchType.LAZY, // default: EAGER
        optional = false) // default: true
    private Address address;
}
```

```
@Entity
public class Address {
}
```

Dwukierunkowa

```
@Entity
public class User {

    @OneToOne
    @JoinColumn(name = „ADDRESS_FK”)
    private Address address;
}
```

- 
1. mappedBy określa właściciela relacji
 2. użytkownik ma klucz obcy do adresu
 3. Bez mappedBy klucz obcy po obu stronach

```
@Entity
public class Address {

    @OneToOne(mappedBy = „address”)
    private User user;
}
```


Relacja @OneToMany / @ManyToOne

Jednokierunkowa

```
@Entity
public class User {

    @OneToMany(
        cascade = CascadeType.ALL, // default: empty
        fetch = FetchType.EAGER) // default: LAZY
    @JoinColumn(name = „user_id”)
    private Collection<Address> addresses;
}
```

bez @JoinColumn utworzona zostanie tabela asocjacyjna

```
@Entity
public class Address {
}
```

Dwukierunkowa

```
@Entity
public class User {

    @OneToMany(mappedBy = „user”)
    private Collection<Address> addresses;
}
```

mappedBy tak samo jak @JoinColumn usuwa tabelę asocjacyjną

```
@Entity
public class Address {

    @ManyToOne
    @JoinColumn(name = „USER_FK", nullable = false)
    private User user;
}
```

Relacja @ManyToMany

Jednokierunkowa

```
@Entity
public class User {
    @ManyToMany(
        cascade = CascadeType.ALL, // default: empty
        fetch = FetchType.LAZY) // default: EAGER
    @JoinTable(name = "USER_ADDRESS",
        joinColumns = {@JoinColumn(name = „USER_ID”, nullable = false, updatable = false)},
        inverseJoinColumns = {@JoinColumn(name = „ADDRESS_ID”, nullable = false, updatable = false)})
    private Collection<Address> addresses;
}
```

Dwukierunkowa

```
@Entity
public class Address {
    @ManyToMany(mappedBy = „user”)
    private Collection<User> users;
}
```

← bez mappedBy dwie tabele
asocjacyjne zostaną stworzone

Kaskady (Cascade)

- ❑ Entity Manager dokonuje zmian na danej encji, np. persist
- ❑ Kaskady umożliwiają operację na encji połączonej relacją
 - PERSIST
 - MERGE
 - REMOVE
 - ALL

```
@Entity
@Table(name = "BOOK")
public class BookEntity {

    @OneToOne(cascade = CascadeType.ALL, mappedBy = "book")
    private BookSpoilerEntity bookSpoiler;
```

OrphanRemoval

- ❑ Usuwa sieroty
- ❑ Na encji głównej musi być ustawiona kaskada (CascadeType.UPDATE)
- ❑ Po usunięciu elementów z kolekcji i zapisaniu encji głównej, z bazy danych usuwane są encje połączone relacją

```
@Entity
@Table(name = "BOOK")
public class BookEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @OneToMany(mappedBy = "book", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<BookExemplarEntity> bookExemplars = new HashSet<>();
```

Dziedziczenie

- ❑ Naturalna właściwość obiektów
- ❑ Nie ma zastosowania w relacyjnych bazach danych
- ❑ Rozwiązaniem Hibernate są trzy strategie:
 - SINGLE_TABLE
 - TABLE_PER_CLASS
 - JOINED
- ❑ Możliwość wykonywania zapytań polimorficznych

Strategia SINGLE_TABLE

- ☐ Podobne obiekty przechowywane są w jednej tabeli
- ☐ Rekordy rozróżniane są przez tzw. Dyskryminator
- ☐ Wydajne wyszukiwanie – brak złączeń
- ☐ Puste kolumny (różnice między obiektami)
- ☐ Problem z warunkiem Not-Null.

Strategia SINGLE_TABLE

```
@Entity
@Table(name = "AUTHOR")
@DiscriminatorColumn(name = "TYPE", length = 6, discriminatorType = DiscriminatorType.STRING)
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class AuthorEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    protected Long id;
    @Column(nullable = true, length = 30)
    protected String nickName;
```

```
@Entity
@DiscriminatorValue("WRITER")
public class WriterEntity extends AuthorEntity {

    @Enumerated(EnumType.STRING)
    private LiteraryGenre literaryGenre;
```

```
@Entity
@DiscriminatorValue("PROFES")
public class ProfessorEntity extends AuthorEntity {

    @Column(nullable = true)
    private String university;
```

	TYPE	ID	NICK_NAME	BIRTH_DATE (yyyy-MM-dd)	FIRST_NAME	LAST_NAME	VERSION	LITERARY_GENRE	UNIVERSITY
1	PROFES	13	<null>	1949-06-08	Janusz	Ratajczak	0	<null>	PWR
2	WRITER	2	Molier	1622-01-15	Jean	Poquelin	0	COMEDY	<null>

Strategia TABLE_PER_CLASS

- ☐ Oddzielna tabela bazy danych na jedną encję
- ☐ Brak problemów z warunkami Not-Null
- ☐ Redundancja wspólnych atrybótów
- ☐ Niewydajne zapytania SQL przy zapytaniach polimorficznych

Strategia JOINED

- ☐ Bazowe obiekty zapisane są w jednej tabeli
- ☐ Różnice między obiektami zapisywane są w oddzielnych tabelach
- ☐ Znormalizowana baza danych
- ☐ Brak redundancji
- ☐ Wymagane złączenie tabel przy zapytaniach polimorficznych

Strategia JOINED

```
@Entity
@Table(name = "BOOK_EXEMPLAR")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BookExemplarEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    protected Long id;
    @Column(nullable = false, length = 15, unique = true)
    protected String serialNumber;
```

	ID	SERIAL_NUMBER
1	1	ISBN-0001
2	2	ISBN-0002
3	3	ISBN-0003

```
@Entity
@Table(name = "PAPER_BOOK")
@PrimaryKeyJoinColumn(name = "book_ex_id", referencedColumnName = "id")
public class PaperBookExemplarEntity extends BookExemplarEntity {
    private int pageCount;
    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private PaperSize paperSize;
    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private BookCover bookCover;
```

	BOOK_COVER	PAGES_COUNT	PAPER_SIZE	BOOK_EX_ID
1	HARD	322	A_5	1
2	HARD	254	B_5	2
3	SOFT	443	A_5	3

Typy Embedded

@Embeddable

```
public class PersonalData {
```

```
    private String firstName;
```

```
    private String lastName;
```

```
    @Column (columnDefinition=" DATE", nullable = false)
```

```
    private Date birthDate;
```

```
    public PersonalData() {
```

```
    }
```

```
    // getters & setters
```

```
}
```

@Entity

```
public class AuthorEntity {
```

```
    @Embedded
```

```
    @AttributeOverrides({
```

```
        @AttributeOverride(name = "firstName", column = @Column(name = "FIRST_NAME", nullable = false)),
```

```
        @AttributeOverride(name = "lastName", column = @Column(name = "LAST_NAME", nullable = false)))
```

```
    private PersonalData personalData;
```

Generowanie kluczy głównych

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

```
@Id
@TableGenerator(
    name="bookGen",
    table="ID_GEN", // opcjonalnie
    pkColumnName="GEN_KEY", // opcjonalnie
    valueColumnName="GEN_VALUE", // opcjonalnie
    pkColumnValue="BOOK_ID_GEN") // opcjonalnie
@GeneratedValue(strategy = GenerationType.TABLE, generator = "bookGen")
private Long id;
```

```
@Id
@SequenceGenerator(name = "bookGen", sequenceName = "BOOK_SEQ")
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "bookGen")
private Long id;
```

Encje

- ❑ Adnotacje jako metadane opisujące mapowanie pomiędzy obiektem a bazą
- ❑ Cykle życia encji:
 - **nowy** (*new*) – brak identyfikatora i powiązania z kontekstem persystencji
 - **zarządzany** (*managed*) – encja posiada ID i jest powiązana z kontekstem persystencji
 - **odłączony** (*detached*) – encja posiada ID ale nie jest powiązana z kontekstem pers.
 - **usunięty** (*removed*) – tak jak zarządzany, ale oznaczona jako „do usunięcia”

Listenery

- ☐ **@PrePersist**
- ☐ **@PostPersist**
- ☐ **@PreUpdate**
- ☐ **@PostUpdate**
- ☐ **@PostLoad**
- ☐ **@PreRemove**
- ☐ **@PostRemove**

Listenery wewnątrz encji

```
@Entity
@Table(name = "CUSTOMER_CARD")
public class CustomerCardEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(nullable = false)
    private String serialNumber;

    @PrePersist
    public void generateDefaultSerialNumber() {
        serialNumber = new SerialNumberGenerator().generate();
    }
}
```

EntityManager

- ❑ Podstawowy element całej warstwy persystencji
- ❑ Zarządza transakcjami i encjami

```
String persistenceUnitName = "MyPersistenceUnit";
```

```
// utwórz
```

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnitName);
```

```
EntityManager em = emf.createEntityManager();
```

```
// zrób co masz do zrobienia
```

```
zrobCos(em);
```

```
// zamknij
```

```
em.close();
```

```
emf.close()
```


Entity Manager – dostęp do danych

// zapis

```
Product banan = new Product(1, „banan”, „owoce”);  
em.persist(banan);
```

// odczyt

```
Product bananFromDB = em.find(Product.class, 1);
```

// usunięcie

```
Product bananFromDB = em.find(Product.class, 1);  
em.remove(bananFromDB);
```

// zapytanie

```
Product product = em.createQuery(  
    "SELECT p FROM Product p WHERE p.category = :cat_param", Product.class)  
    .setParameter("cat_param", „owoce”)  
    .getSingleResult();
```

Transakcje

- ☐ Zbiór operacji na bazie danych stanowiących jedną całość
- ☐ Zmieniają spójny stan bazy danych w inny spójny stan
- ☐ Cechy transakcji:
 - Atomowość (atomicity)
 - Spójność (consistency)
 - Izolacja (isolation)
 - Trwałość (durability)

Transakcje – EntityManager

```
// rozpoczyna transakcję  
em.getTransaction().begin();  
// wykonanie operacji  
Product prodFromDb = em.find(Product.class, 1);  
prodFromDb.setCategory(„newCategory”);  
// zatwierdzenie transakcji  
em.getTransaction().commit();  
// ewentualne wycofanie transakcji  
em.getTransaction().rollback();
```

Transakcje Spring @Transactional

- ❑ Najlepszy sposób budowania transakcyjnych aplikacji Spring

```
@Service
@Transactional(readOnly = true)
public class LibraryServiceImpl implements LibraryService {

    @Autowired
    private LibraryRepository libraryRepository;
    @Autowired
    private MapperFacade mapper;

    @Override
    public List<LibraryTo> findAllLibraries() {
        List<LibraryEntity> libraries = libraryRepository.findAll();
        return mapper.mapAsList(libraries, LibraryTo.class);
    }
}
```

- ❑ Brak powielonego kodu
- ❑ Bardzo łatwa konfiguracja

- *Propagacja, izolacja, timeout, readOnly, rollbackFor itp.*

- ❑ Możliwość zdefiniowania na całej klasie i metodzie

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
```

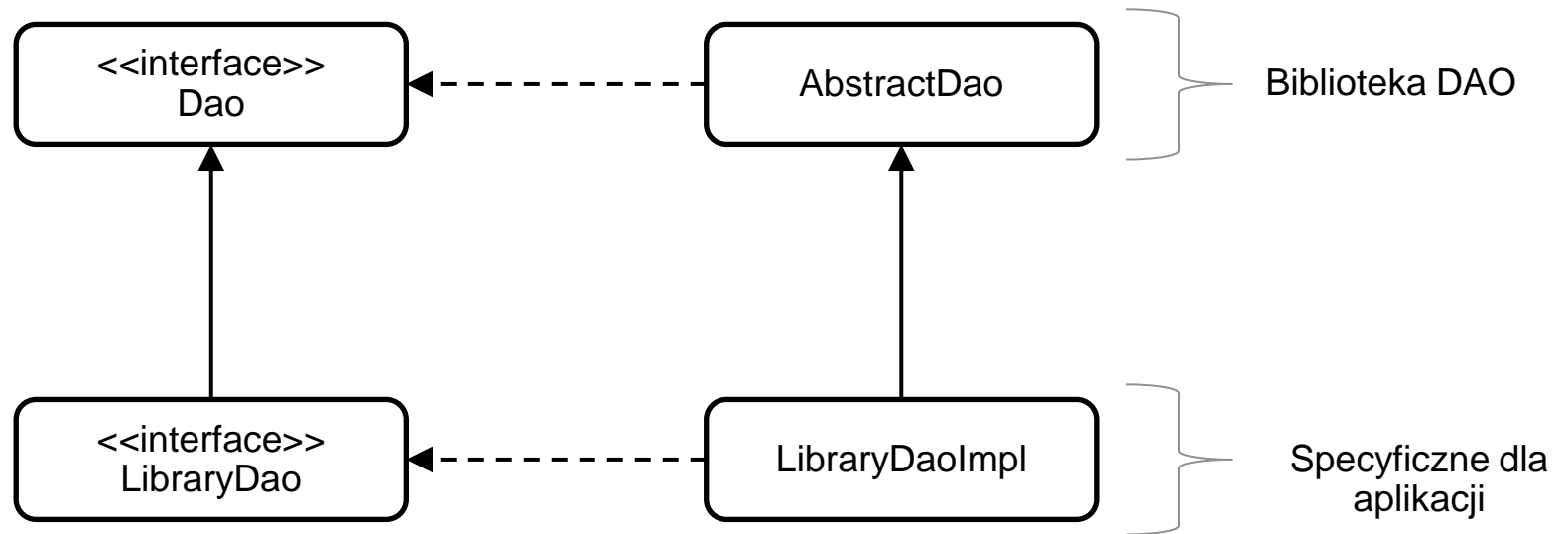
Dziękujemy za uwagę

www.capgemini.com



Oddzielenie logiki domeny od obsługi trwałości

■ Zastosowanie wzorca *Data Access Object* (DAO)



Obiekty DAO

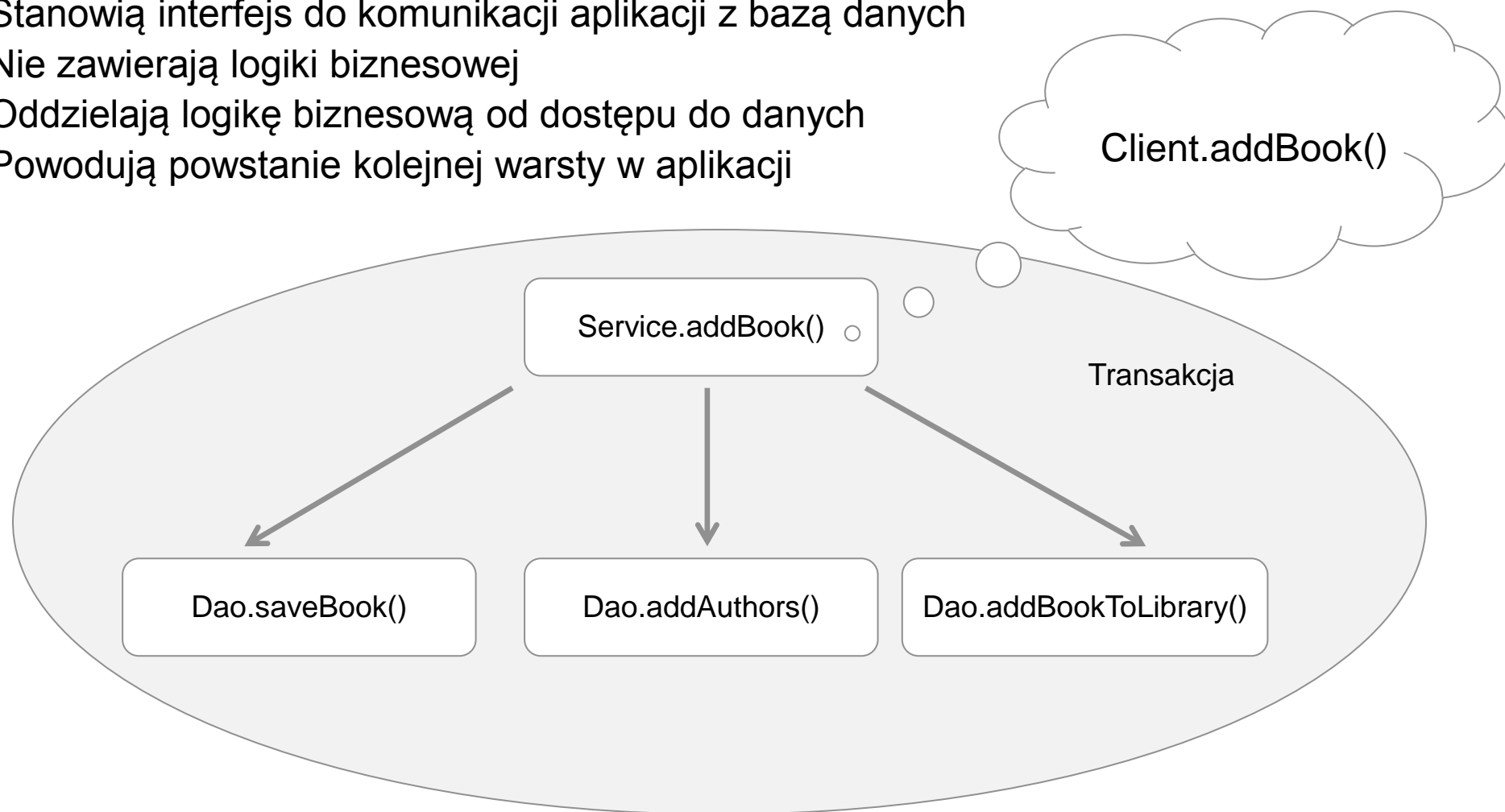
```
public interface Dao<T> {  
  
    void create(T entity);  
    T get(Serializable id);  
    T load(Serializable id);  
    List<T> getAll();  
    void update(T entity);  
    void saveOrUpdate(T entity);  
    void delete(T entity);  
    void delete(Serializable id);  
    void deleteAll();  
    long count();  
    boolean exists(Serializable id);  
}
```

```
@Transactional(Transactional.TxType.SUPPORTS)  
public abstract class AbstractDao<T> implements Dao<T> {  
    @Autowired  
    private SessionFactory sessionFactory;  
    private Class<T> domainClass;  
  
    protected Session getSession() {  
        return sessionFactory.getCurrentSession();  
    }  
  
    @Override  
    public void create(T entity) {  
        getSession().save(entity);  
    }  
  
    @Override  
    @SuppressWarnings("unchecked")  
    public T get(Serializable id) {  
        return (T) getSession().get(getDomainClass(), id);  
    }  
  
    @Override  
    @SuppressWarnings("unchecked")  
    public T load(Serializable id) {  
        return (T) getSession().load(getDomainClass(), id);  
    }  
  
    @Override  
    @SuppressWarnings("unchecked")  
    public List<T> getAll() {  
        return getSession().createQuery("from " + getDomainClassName()).list();  
    }  
}
```

```
@Repository  
public class LibraryDaoImpl extends AbstractDao<LibraryEntity> {  
  
    @SuppressWarnings("unchecked")  
    public List<LibraryEntity> findByName(String name) {  
        return getSession().createQuery("from LibraryEntity l where l.name like :name")  
            .setString("name", name + "%").list();  
    }  
}
```

Obiekty DAO

- ❑ Stanowią interfejs do komunikacji aplikacji z bazą danych
- ❑ Nie zawierają logiki biznesowej
- ❑ Oddzielają logikę biznesową od dostępu do danych
- ❑ Powodują powstanie kolejnej warsty w aplikacji



Spring-data-jpa repozytoria

```
package pl.spring.demo.repository;

import ...

public interface LibraryRepository extends JpaRepository<LibraryEntity, Long>, LibraryLambdaRepository {

    List<LibraryEntity> findByNameLike(String name);

    @Query("from LibraryEntity l where l.name like :name%")
    List<LibraryEntity> findByName(@Param("name") String name);

    @Query("from LibraryEntity l where l.address.city = :cityName")
    List<LibraryEntity> findByCity(@Param("cityName") String cityName);

    @Query("from LibraryEntity l where exists (select 1 from BookEntity b where b.library.id = l.id and b.title = :bookTitle)")
    List<LibraryEntity> findLibrariesThatHaveBookByTitle(@Param("bookTitle") String bookTitle);
}
```

- ☐ Skanowanie pakietu *pl.spring.demo.repository*
- ☐ Walidacja wszystkich zapytań *@Query*
- ☐ Automatyczna implementacja interfejsu
- ☐ Możliwość generowania zapytań przez konwencję

Spring-data-jpa repozytoria

- ❑ Możliwość tworzenia niestandardowych repozytoriów
- ❑ Bezpośredni dostęp do obiektu *EntityManager*

```
public interface LibraryRepository extends JpaRepository<LibraryEntity, Long>, MyCustomLibraryRepository {  
  
    @Query("from LibraryEntity l where l.name like :name%")  
    List<LibraryEntity> findByName(@Param("name") String name);  
  
}
```

```
public class LibraryRepositoryImpl implements MyCustomLibraryRepository {  
  
    @PersistenceContext(name = "hsqldb")  
    private EntityManager entityManager;  
  
    @Override  
    public LibraryEntity findLibraryById(long id) {  
        return entityManager.find(LibraryEntity.class, id);  
    }  
  
}
```

```
public interface MyCustomLibraryRepository {  
  
    LibraryEntity findLibraryById(long id);  
  
}
```