Hibernate

v3.1

Plan

- Hibernate
- Rozpoczęcie pracy z Hibernate
- Model - podstawy stosowania
- Praca z encjami
- Relacje w Hibernate
- Zaawansowane tematy

Hibernate

Co to jest Hibernate?

Hibernate to biblioteka do realizacji warstwy dostępu do danych (ang. persistence layer). Zapewnia translację danych pomiędzy relacyjną bazą danych a światem obiektowym.

W aspekcie dostępu do danych często używamy określenia ORM - co oznacza mapowanie obiektowo-relacyjne (ang. Object-Relational Mapping – ORM).

Oficjalna strona projektu:

http://hibernate.org/orm/

Dlaczego ORM ?

Wykorzystanie **JDBC** w projekcie niesie ze sobą wiele niedogodności:

- Tworzymy dużo powtarzanego kodu.
- Łatwo o popełnienie błędów.
- Proces zarządzania zmianą jest czasochłonny.

Rozwiązaniem tych problemów jest stosowanie mapowania obiektowo-relacyjnego.

Oznacza to odwzorowanie obiektowej architektury naszej aplikacji relacyjną strukturę bazy danych.

Korzystając z **ORM** łatwo możemy zmienić wykorzystywaną bazę danych.

ORM stanowi warstwę abstrakcji nad różnymi źródłami danych.

Co to jest JPA?

JPA (Java Persistence API) - to specyfikacja Javy EE do mapowania obiektowo-relacyjnego.

JPA to specyfikacja, a hibernate jest jej konkretną implementacja.

Istnieją alternatywne implementacje, np.:

- EclipseLink
- iBATIS

Hibernate

Mimo dostępnych alternatyw Hibernate jest swego rodzaju standardem.

Specyfikacje JPA są opracowywane przy współpracy z jego twórcami.

Porównania szybkości działania wskazują na przewagę innych rozwiązań, ze względu jednak na nieustającą popularność Hibernate jest nadal rozwiązaniem dominującym.
http://www.jpab.org/Hibernate/MySQL/server/EclipseLink/MySQL/server.html

Rozpoczęcie pracy z Hibernate

Zależności

Pierwszym krokiem do skorzystania z możliwości Hibernate jest dołączenie odpowiedniej biblioteki do naszego projektu.

Zwróćmy uwagę, że dodajemy również sterownik do bazy danych - w naszym przypadku MySQL.

Zależności

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.9.Final</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>4.3.7.RELEASE</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.39</version>
</dependency>
```

Konfigurowanie połączenia

Pierwszym krokiem jest dodanie pliku o nazwie **persistence.xml** w lokalizacji:

src/main/resources/META-INF/persistence.xml

Mimo możliwości konfiguracji opartej o Javę, plik ten często stanowi cześć aplikacji.

W pliku tym określamy dane konfiguracyjne naszego połączenia.

persistence.xml

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
<persistence-unit name="bookstorePersistenceUnit">
    <properties>
    <property name="javax.persistence.jdbc.url"
              value="jdbc:mysql://localhost:3306/bookStore" />
    <property name="javax.persistence.jdbc.user" value="root" />
    <property name="javax.persistence.jdbc.password" value="coderslab" />
    <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
    <property name="javax.persistence.schema-generation.database.action"
              value="none"/>
      </properties></persistence-unit></persistence>
```

persistence.xml

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
<persistence-unit name="bookstorePersistenceUnit">
    <properties>
    <property name="javax.persistence.jdbc.url"
              value="jdbc:mysql://localhost:3306/bookStore" />
    <property name="javax.persistence.jdbc.user" value="root" />
    <property name="javax.persistence.jdbc.password" value="coderslab" />
    <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
    <property name="javax.persistence.schema-generation.database.action"
              value="none"/>
      </properties></persistence-unit></persistence>
```

Ten element powinien posiadać unikalną nazwę - będziemy sie do niej odwoływać w aplikacji podczas definicji fabryki **EntityManagera** - o tym elemencie dowiemy się na kolejnych slajdach.

persistence.xml

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
<persistence-unit name="bookstorePersistenceUnit">
    <properties>
    <property name="javax.persistence.jdbc.url"
              value="jdbc:mysql://localhost:3306/bookStore" />
    <property name="javax.persistence.jdbc.user" value="root" />
    <property name="javax.persistence.jdbc.password" value="coderslab" />
    <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
    <property name="javax.persistence.schema-generation.database.action"
              value="none"/>
      </properties></persistence-unit></persistence>
```

Dane dostępowe do bazy danych.

persistence.xml

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
<persistence-unit name="bookstorePersistenceUnit">
    <properties>
    <property name="javax.persistence.jdbc.url"
              value="jdbc:mysql://localhost:3306/bookStore" />
    <property name="javax.persistence.jdbc.user" value="root" />
    <property name="javax.persistence.jdbc.password" value="coderslab" />
    <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
    <property name="javax.persistence.schema-generation.database.action"
              value="none"/>
      </properties></persistence-unit></persistence>
```

Odpowiada za generowanie tabel - możliwe wartości są opisane na kolejnym slajdzie.

Sposób generowania

Właściwość **`javax.persistence.schema-generation.database.action`** może przyjmować następujące wartości:

- **create** - tworzy tabele na podstawie adnotacji encji, nie nadpisuje zmian
- **none** - nie wykonuje żadnych operacji
- **drop-and-create** - usunie a następnie utworzy
- **drop** - usunie elementy bazy danych zgodne z adnotacjami

Określona operacja wykona się przy każdorazowym uruchomieniu aplikacji na serwerze.

Dokumentacja

https://docs.oracle.com/javaee/7/tutorial/persistence-intro005.htm

Skrypty

Przydatną opcją jest możliwość definiowania skryptu SQL, który może służyć do ładowania danych.

Aby dodać skrypt wystarczy utworzyć odpowiedni wpis **`property`**.

```
<property name="javax.persistence.sql-load-script-source"
          value="META-INF/sql/data.sql" />
```

Dodatkowe ustawienia

W pliku **persistence.xml** - możemy również zdefiniować dodatkowe specyficzne dla konkretnego dostawcy JPA ustawienia.

Pełen zestaw opcji konfiguracyjnych znajdziemy w dokumentacji:
http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#configurations

```
<property name="hibernate.hbm2ddl.auto" value="update"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
```

Dodatkowe ustawienia

W pliku **persistence.xml** - możemy również zdefiniować dodatkowe specyficzne dla konkretnego dostawcy JPA ustawienia.

Pełen zestaw opcji konfiguracyjnych znajdziemy w dokumentacji:
http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#configurations

```
<property name="hibernate.hbm2ddl.auto" value="update"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
```

Aktualizacja struktury bazy danych.

Dodatkowe ustawienia

W pliku **persistence.xml** - możemy również zdefiniować dodatkowe specyficzne dla konkretnego dostawcy JPA ustawienia.

Pełen zestaw opcji konfiguracyjnych znajdziemy w dokumentacji:
http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#configurations

```
<property name="hibernate.hbm2ddl.auto" value="update"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
```

Aktualizacja struktury bazy danych.

Wyświetlanie generowanego sql.

Dodatkowe ustawienia

W pliku **persistence.xml** - możemy również zdefiniować dodatkowe specyficzne dla konkretnego dostawcy JPA ustawienia.

Pełen zestaw opcji konfiguracyjnych znajdziemy w dokumentacji:
http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#configurations

```
<property name="hibernate.hbm2ddl.auto" value="update"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
```

Aktualizacja struktury bazy danych.

Wyświetlanie generowanego sql.

Formatowanie wyświetlanego sql w sposób bardziej czytelny.

Dodatkowe ustawienia

Do pliku **persistence.xml** dodajemy ustawienia odpowiedzialne za wybór silnika **InnoDB** oraz za kodowanie **UTF-8**.

```
<property name="hibernate.dialect"
                    value="org.hibernate.dialect.MySQL57Dialect" />
<property name="hibernate.connection.useUnicode" value="true" />
<property name="hibernate.connection.characterEncoding" value="utf8" />
<property name="hibernate.connection.CharSet" value="utf8" />
```

Konfigurowanie

W celu skorzystania w aplikacji z możliwości **JPA** definiujemy ziarno typu **LocalEntityManagerFactoryBean**. Tworzy ono fabrykę menedżerów encji w wyniku wczytania pliku konfiguracyjnego - **persistence.xml**.

Jest to poznana już implementacja wzorca fabryki, którą udostępnia **Spring**.

LocalEntityManagerFactoryBean to podstawowa implementacja, którą udostępnia **Spring**.

Dzięki takiej konfiguracji będziemy mogli użyć wstrzykiwania zależności aby pozyskać instancję obiektu **EntityManager**.

Obiektu **EntityManager** będziemy używać w celu wykonywania operacji których wyniki będą zapisywane w bazie danych.

Konfigurowanie

Pierwsze ziarno definiuje fabrykę **EntityManagera** (zarządcy encji) - obiektu, którym będziemy się posługiwać w celu wykonywania operacji na naszych encjach.

Określamy również sposób zarządzania transakcjami - włączamy zarządzanie transakcjami przez **Springa**.

```
@Bean
public LocalEntityManagerFactoryBean entityManagerFactory() {
    LocalEntityManagerFactoryBean emfb = new LocalEntityManagerFactoryBean();
    emfb.setPersistenceUnitName("bookstorePersistenceUnit");
    return emfb; }
@Bean
public JpaTransactionManager transactionManager(EntityManagerFactory emf) {
    JpaTransactionManager tm = new JpaTransactionManager(emf);
    return tm; }
```

Konfigurowanie

Pierwsze ziarno definiuje fabrykę **EntityManagera** (zarządcy encji) - obiektu, którym będziemy się posługiwać w celu wykonywania operacji na naszych encjach.

Określamy również sposób zarządzania transakcjami - włączamy zarządzanie transakcjami przez **Springa**.

```
@Bean
public LocalEntityManagerFactoryBean entityManagerFactory() {
    LocalEntityManagerFactoryBean emfb = new LocalEntityManagerFactoryBean();
    emfb.setPersistenceUnitName("bookstorePersistenceUnit");
    return emfb; }
@Bean
public JpaTransactionManager transactionManager(EntityManagerFactory emf) {
    JpaTransactionManager tm = new JpaTransactionManager(emf);
    return tm; }
```

Nazwa jednostki utrwalania musi być taka sama jak ustalona wcześniej w pliku **persistence.xml**.

Konfigurowanie

Do pliku konfiguracji dodajemy adnotację **@EnableTransactionManagement**

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "pl.coderslab")
@EnableTransactionManagement
public class AppConfig extends WebMvcConfigurerAdapter {

 //definicje beanów
}
```

Konfigurowanie

Do pliku konfiguracji dodajemy adnotację **@EnableTransactionManagement**

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "pl.coderslab")
@EnableTransactionManagement
public class AppConfig extends WebMvcConfigurerAdapter {

 //definicje beanów
}
```

Adnotacje zdefiniowane we wcześniejszym module kursu.

Konfigurowanie

Do pliku konfiguracji dodajemy adnotację **@EnableTransactionManagement**

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "pl.coderslab")
@EnableTransactionManagement
public class AppConfig extends WebMvcConfigurerAdapter {

 //definicje beanów
}
```

Adnotacje zdefiniowane we wcześniejszym module kursu.

Włącza zarządzanie transakcjami.

Wykonaj zadania z działu

Tworzenie projektu

Klasy trwałe

Encja

Podstawowym pojęciem w Java Persistence jest **encja**.

Encja to lekki obiekt służący do reprezentacji trwałych danych.

Encje muszą spełniać poniższe warunki:

- Bezargumentowy konstruktor oznaczony jako public lub protected.
- Brak oznaczenia final dla klasy, jak i dla pól i metod.

Definicja Encji

Klasa musi być oznaczona adnotacją **@Entity**

Pola odpowiadające kolumnom tabeli.

Opcjonalna adnotacja **@Column**, określająca nazwę kolumny.

Adnotacje są odpowiedzialne za relacje.

Pola są mapowane na kolumny w bazie danych.

Tabela w bazie będzie miała nazwę taką samą jak nazwa klasy.

Pola nie odpowiadające kolumnom w bazie oznaczone adnotacją **@Transient**.

Przykład Encji

Encje nie dziedziczą po żadnej specjalnej klasie.

Wystarczy, że będą posiadać odpowiednie adnotacje.

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    String title;
    String author;
}
```

Klucze

Encja musi posiadać pole określające jej klucz główny.

W celu określenia, że atrybut jest kluczem stosujemy adnotację **@Id**.

Określamy również sposób w jaki klucz ma być generowany, służy do tego adnotacja **@GeneratedValue**.

Klucze

Mamy do dyspozycji kilka strategii generowania klucza:

- **IDENTITY** - generowanie klucza na podstawie kolumny zwiększającej swoją wartość (auto increment), MySQL posiada własność auto_increment.
- **TABLE** - utworzona zostanie dodatkowa tabela, na podstawie której będzie generowany klucz.
- **SEQUENCE** - generowanie klucza odbywa sie po stronie bazy danych z wykorzystaniem sekwencji - stosowane np. w bazach Oracle/PostgreSQL.
- **AUTO** - strategia jest dobierana automatycznie.

Własna nazwa tabeli

W celu połączenia encji z tabelą wystarczy dopisać adnotację:

- **@Table(name = "books")**

```
@Entity
@Table(name = "books")
public class Book {
    //...
}
```

Własna nazwa kolumny

Nazwy kolumn tworzą się na podstawie nazw atrybutów danej klasy.

Podobnie jak w przypadku adnotacji @Table dla kolumny możemy również określić nazwę.

Atrybut przypisujemy za pomocą adnotacji:

@Column(name = "title")

```
@Entity
@Table(name = "books")
public class Book {
    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private long id;
    @Column(name = "myTitle",
                    length=100,
                    nullable = false)
    private String title;
    @Column(scale=2, precision=4)
    private BigDecimal rating;
    @Column(columnDefinition="TEXT")
    private String description;
    private String author; }
```

@Column - atrybuty

Adnotacja **Column()** przyjmuje następujące parametry:

nullable

czy może być null - domyślnie **true**

unique

czy wartość musi być unikalna - domyślnie **false**

length

długość łańcucha znaków - domyślnie **255**

precision

precyzja liczb zmiennoprzecinkowych (łączna liczba cyfr). - domyślnie **0**

scale

precyzja liczb zmiennoprzecinkowych (liczba cyfr po przecinku) - domyślnie **0**

Praca z encjami

Entity Manager

EntityManager – zarządca encji, udostępnia nam możliwość operowania na naszych encjach.

Jest to element standardu **JPA**.

Dokumentacja jest dostępna pod adresem: http://docs.oracle.com/javaee/6/api/javax/persistence/EntityManager.html

Aby wykorzystać **EntityManager** w naszych klasach Dao lub Repository możemy wstrzyknąć ziarno korzystając z adnotacji **@PersistenceContext**

```
@PersistenceContext
private EntityManager entityManager;
```

Zapisywanie nowej encji do bazy danych

W ramach przykładu zapisu danych do bazy utworzymy klasę **BookDao**

```
@Component
@Transactional
public class BookDao {
}
```

Użycie adnotacji **@Transactional** powoduje, że każda metoda tej klasy będzie stanowić **transakcję**.

Oznacza to, że transakcja zacznie się przed wejściem do metody, a zakończy po jej wykonaniu.

Adnotację **@Transactional** - możemy umieścić nad całą klasą - wtedy będzie dotyczyć wszystkich jej metod lub nad pojedynczą metodą.

W celu przypomnienia - transakcje omawialiśmy podczas dnia drugiego zajęć z MySQL.

Zapisywanie nowej encji do bazy danych

Do zapisu encji do bazy danych służy metoda **persist** obiektu typu **EntityManager**, która jako parametr otrzymuje obiekt do zapisu.

Zauważmy, że nasza klasa przyjmuje obiekt typu **Book**, możliwe jest zdefiniowanie takiej klasy w sposób bardziej ogólny.

Za chwilę zapoznamy się z takim przykładem.

Przykład definicji klasy dostępu do danych.

```
@Component
@Transactional
public class BookDao {
    @PersistenceContext
    EntityManager entityManager;
    public void saveBook(Book entity) {
        entityManager.persist(entity);
    }
}
```

Zapisywanie nowej encji do bazy danych

Do zapisu encji do bazy danych służy metoda **persist** obiektu typu **EntityManager**, która jako parametr otrzymuje obiekt do zapisu.

Zauważmy, że nasza klasa przyjmuje obiekt typu **Book**, możliwe jest zdefiniowanie takiej klasy w sposób bardziej ogólny.

Za chwilę zapoznamy się z takim przykładem.

Przykład definicji klasy dostępu do danych.

```
@Component
@Transactional
public class BookDao {
    @PersistenceContext
    EntityManager entityManager;
    public void saveBook(Book entity) {
        entityManager.persist(entity);
    }
}
```

Określamy, że nasza klasa ma być komponentem zarządzanym przez Springa.

Zapisywanie nowej encji do bazy danych

Do zapisu encji do bazy danych służy metoda **persist** obiektu typu **EntityManager**, która jako parametr otrzymuje obiekt do zapisu.

Zauważmy, że nasza klasa przyjmuje obiekt typu **Book**, możliwe jest zdefiniowanie takiej klasy w sposób bardziej ogólny.

Za chwilę zapoznamy się z takim przykładem.

Przykład definicji klasy dostępu do danych.

```
@Component
@Transactional
public class BookDao {
    @PersistenceContext
    EntityManager entityManager;
    public void saveBook(Book entity) {
        entityManager.persist(entity);
    }
}
```

Określamy, że wszystkie metody w klasie stanowią są zamykane w transakcje.

Zapisywanie nowej encji do bazy danych

Do zapisu encji do bazy danych służy metoda **persist** obiektu typu **EntityManager**, która jako parametr otrzymuje obiekt do zapisu.

Zauważmy, że nasza klasa przyjmuje obiekt typu **Book**, możliwe jest zdefiniowanie takiej klasy w sposób bardziej ogólny.

Za chwilę zapoznamy się z takim przykładem.

Przykład definicji klasy dostępu do danych.

```
@Component
@Transactional
public class BookDao {
    @PersistenceContext
    EntityManager entityManager;
    public void saveBook(Book entity) {
        entityManager.persist(entity);
    }
}
```

Do pola oznaczonego adnotacją **@PersistenceContext** Spring wstrzyknie managera encji.

Zapisywanie nowej encji do bazy danych

Do zapisu encji do bazy danych służy metoda **persist** obiektu typu **EntityManager**, która jako parametr otrzymuje obiekt do zapisu.

Zauważmy, że nasza klasa przyjmuje obiekt typu **Book**, możliwe jest zdefiniowanie takiej klasy w sposób bardziej ogólny.

Za chwilę zapoznamy się z takim przykładem.

Przykład definicji klasy dostępu do danych.

```
@Component
@Transactional
public class BookDao {
    @PersistenceContext
    EntityManager entityManager;
    public void saveBook(Book entity) {
        entityManager.persist(entity);
    }
}
```

Wywołujemy metodę **persist**.

Zapisywanie nowej encji do bazy danych

Przykład wywołania w kontrolerze Springa

```
@Controller
public class HomeController {
    @Autowired
    private BookDao bookDao;
    @RequestMapping("/")
    @ResponseBody
    public String hello(){
        Book book  = new Book();
        book.setTitle("Thinking in
                       Java");
        book.setAuthor("Bruce Eckel");
        bookDao.saveBook(book);
        return "Id dodanej książki to:"
               +  book.getId(); } }
```

Zapisywanie nowej encji do bazy danych

Przykład wywołania w kontrolerze Springa

```
@Controller
public class HomeController {
    @Autowired
    private BookDao bookDao;
    @RequestMapping("/")
    @ResponseBody
    public String hello(){
        Book book  = new Book();
        book.setTitle("Thinking in
                       Java");
        book.setAuthor("Bruce Eckel");
        bookDao.saveBook(book);
        return "Id dodanej książki to:"
               +  book.getId(); } }
```

Za pomocą adnotacji **@Controller** oznaczamy naszą klasę jako kontroler.

Zapisywanie nowej encji do bazy danych

Przykład wywołania w kontrolerze Springa

```
@Controller
public class HomeController {
    @Autowired
    private BookDao bookDao;
    @RequestMapping("/")
    @ResponseBody
    public String hello(){
        Book book  = new Book();
        book.setTitle("Thinking in
                       Java");
        book.setAuthor("Bruce Eckel");
        bookDao.saveBook(book);
        return "Id dodanej książki to:"
               +  book.getId(); } }
```

Za pomocą adnotacji **@Controller** oznaczamy naszą klasę jako kontroler.

Wstrzykujemy Dao.

Zapisywanie nowej encji do bazy danych

Przykład wywołania w kontrolerze Springa

```
@Controller
public class HomeController {
    @Autowired
    private BookDao bookDao;
    @RequestMapping("/")
    @ResponseBody
    public String hello(){
        Book book  = new Book();
        book.setTitle("Thinking in
                       Java");
        book.setAuthor("Bruce Eckel");
        bookDao.saveBook(book);
        return "Id dodanej książki to:"
               +  book.getId(); } }
```

Za pomocą adnotacji **@Controller** oznaczamy naszą klasę jako kontroler.

Wstrzykujemy Dao.

Tworzymy obiekt i wypełniamy jego właściwości.

Zapisywanie nowej encji do bazy danych

Przykład wywołania w kontrolerze Springa

```
@Controller
public class HomeController {
    @Autowired
    private BookDao bookDao;
    @RequestMapping("/")
    @ResponseBody
    public String hello(){
        Book book  = new Book();
        book.setTitle("Thinking in
                       Java");
        book.setAuthor("Bruce Eckel");
        bookDao.saveBook(book);
        return "Id dodanej książki to:"
               +  book.getId(); } }
```

Za pomocą adnotacji **@Controller** oznaczamy naszą klasę jako kontroler.

Wstrzykujemy Dao.

Tworzymy obiekt i wypełniamy jego właściwości.

Wywołujemy zdefiniowaną przez nas metodę **saveBook** .

Zapisywanie nowej encji do bazy danych

Na zajęciach, w celu ułatwienia pracy z encjami, możemy wywoływać poszczególne metody bezpośrednio w akcji kontrolera.

Pamiętajmy jednak, że w aplikacjach produkcyjnych możemy spotkać się z podziałem:

- **klasy Dao/Repository** - realizujące dostęp do danych
- **serwisy** - realizujące logikę biznesową
- **kontrolery** - wywołujące metody serwisów

Pod pojęciem **logiki biznesowej** mieści się kompletny proces związany np. z zapisem książki.

Taki proces będzie zawierał więcej operacji np.:

- dodanie dokumentów magazynowych,
- umieszczenie jej w pozycjach polecanych,
- przesłanie informacji do księgowości.

Pobieranie obiektu po identyfikatorze

Do pobierania pojedynczego obiektu na podstawie klucza służy metoda **find**.

Metoda ta dostaje 2 parametry - pierwszy to klasa jakiej obiektu poszukujemy - w naszym przypadku klasa **Book**, drugi to identyfikator do wyszukiwania z bazy danych.

W naszej encji jest to właściwość oznaczona adnotacją **@Id**.

Rozszerzamy naszą klasę **Dao** o nową metodę:

```
public Book findById(long id) {
    return entityManager.find(Book.class, id);
}
```

Aktualizacja obiektu

Do zapisania zmian na istniejącym już obiekcie użyjemy metody **merge**.

Rozszerzamy naszą klasę **Dao** o nową metodę:

```
public void update(Book entity) {
      entityManager.merge(entity);
}
```

Usuwanie obiektu

Do usuwania obiektu użyjemy metody **remove**.

EntityManager posiada metodę **`contains`**, która zwraca wartość **true** jeżeli encja jest przez niego zarządzana.

W praktyce dodatkowo sprawdzamy stan obiektu, a w razie potrzeby dodatkowo go aktualizujemy.

Rozszerzamy naszą klasę Dao o nową metodę:

```
public void delete(Book entity) {
  entityManager.remove(entityManager.contains(entity) ?
  entity : entityManager.merge(entity)); }
```

Jest to zabezpieczenie przed próbą usunięcia obiektu, który został odłączony, np. w skutek niepowodzenia transakcji.

Informacje na temat cyklu życia encji znajdziemy pod adresem:
https://docs.oracle.com/cd/E16439_01/doc.1013/e13981/undejbs003.htm#BABIAAGE

Wykonaj zadania z działu

Encje

Relacje w Hibernate

Relacje/Asocjacje

W **Hibernate** mamy możliwość stworzenia relacji między encjami.

Podział relacji:

- Jeden do jednego – **@OneToOne**,
- Jeden do wielu – **@OneToMany**,
- Wiele do jednego – **@ManyToOne**,
- Wiele do wielu – **@ManyToMany**.

Relacje w Hibernate możemy również podzielić na:

- jednokierunkowe
- dwukierunkowe

Szczegółowy opis znajdziemy w dokumentacji:
http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#associations

Jak czytać relacje

Jako przykład przyjmiemy dwie klasy **Person** oraz **Address**.

Relacja **OneToMany** oznacza:

One **Person** to Many **Addresses**

Relacja **OneToOne** oznacza:

One **Person** to One **Address**

Relacja **ManyToOne** oznacza:

Many **Persons** to One **Address**

Relacja **ManyToMany** oznacza:

Many **Persons** to Many **Addresses**

Mimo niepoprawnej gramatycznie formy.

Kierunkowość relacji

Kierunkowość relacji - oznacza możliwość nawigacji między powiązanymi obiektami.

Relacja **dwukierunkowa** jest definiowana jako para relacji **jednokierunkowych**, ze wskazaniem jednej z nich jako głównej.

Klasa **Address** będzie w relacji z klasą **Person** jeżeli będzie posiadać atrybut lub listę atrybutów jej typu.

W przypadku powiązań **dwukierunkowych** należy pamiętać, że odpowiednie definicje należy dodać w obu klasach, które w danej relacji uczestniczą.

Przykłady relacji

W przykładach dotyczących encji w celu uproszczenia pominięte zostały getery i setery do atrybutów.

Encje są to znane nam obiekty Javy.

Operacje kaskadowe

W przykładach dotyczących relacji określamy również atrybut **cascade**.

Oznacza to, że operacje na encji mają powodować wykonanie operacji dla powiązanych encji.

Zestaw wszystkich możliwości znajdziemy w dokumentacji: https://docs.oracle.com/cd/E19798-01/821-1841/bnbqm/index.html .

Najważniejsze z dostępnych opcji to

- **CascadeType.ALL** - oznacza wykonanie wszystkich operacji
- **CascadeType.REMOVE** - oznacza usunięcie powiązanych encji
- **CascadeType.PERSIST** - oznacza zapis powiązanych encji

Leniwe i zachłanne ładowanie

W przykładach dotyczących relacji określamy atrybut **fetch**.

Może on przyjmować następujące wartości

- **FetchType.EAGER** - pobieranie natychmiastowe
- **FetchType.LAZY** - pobieranie opóźnione

Oznacza to sposób ładowania powiązanych encji.

Relacje **@OneToOne** i **@ManyToOne** - domyślnie otrzymują wartość: **FetchType.EAGER**

Relacje **@OneToMany** i **@ManyToMany** - domyślnie otrzymują wartość: **FetchType.LAZY**

Leniwe i zachłanne ładowanie

FetchType.LAZY - oznacza, że dane z bazy zawierające elementy powiązane zostaną odczytane z bazy dopiero w momencie gdy się do nich odwołamy.

Odwołanie to następuje przez wywołanie metody pobierającej kolekcję powiązanych obiektów, czyli tzw. getter.

FetchType.EAGER - oznacza, że dane z bazy zawierające elementy powiązane zostaną odczytane wraz z pobieraniem encji.

Np. wczytując obiekt klasy **Person** wczytamy automatycznie listę wszystkich obiektów klasy **Address**.

Ustawianie wszystkich relacji w taki sposób nie jest zalecane ze względu na kwestie wydajnościowe. Może się okazać, że zbędnie pobieramy duże ilości danych.

@ManyToOne

Jedna z encji nie posiada bezpośrednio żadnych informacji o powiązaniu.

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Long id;
    // ....
}
```

Druga encja posiada pole z dodatkową adnotacją **@ManyToOne**

```
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private Long id;
    @ManyToOne
    @JoinColumn(name = "person_id")
    private Person person;
}
```

@ManyToOne

Jedna z encji nie posiada bezpośrednio żadnych informacji o powiązaniu.

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Long id;
    // ....
}
```

Druga encja posiada pole z dodatkową adnotacją **@ManyToOne**

```
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private Long id;
    @ManyToOne
    @JoinColumn(name = "person_id")
    private Person person;
}
```

Jest to adnotacja opcjonalna - za jej pomocą określamy nazwę klucza obcego. W przypadku braku otrzyma składającą się z nazwy pola znaku podkreślenia oraz nazwy klucza drugiej encji.

@OneToMany

W tym przypadku to pierwsza klasa posiada informacje o drugiej.

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Long id;
    @OneToMany(cascade =
        CascadeType.ALL)
    @JoinColumn(name="id_phone")
    private List<Phone> phones =
        new ArrayList<>();
}
```

Druga encja nie posiada dodatkowych adnotacji.

```
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private Long id;
}
```

@OneToMany

W tym przypadku to pierwsza klasa posiada informacje o drugiej.

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Long id;
    @OneToMany(cascade =
        CascadeType.ALL)
    @JoinColumn(name="id_phone")
    private List<Phone> phones =
        new ArrayList<>();
}
```

Druga encja nie posiada dodatkowych adnotacji.

```
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private Long id;
}
```

Elementów powiązanych może być wiele, więc występuje tutaj kolekcja danych. Możemy również skorzystać z kolekcji typu **Set**.

Asocjacja dwukierunkowa

W tym przypadku to pierwsza klasa posiada informacje o drugiej.

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Long id;
    @OneToMany(mappedBy = "person",
        cascade = CascadeType.ALL)
    private List<Phone> phones =
        new ArrayList<>();
}
```

Druga encja również posiada informacje o pierwszej.

```
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private Long id;
    @ManyToOne
    private Person person;
}
```

Asocjacja dwukierunkowa

W tym przypadku to pierwsza klasa posiada informacje o drugiej.

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Long id;
    @OneToMany(mappedBy = "person",
        cascade = CascadeType.ALL)
    private List<Phone> phones =
        new ArrayList<>();
}
```

Druga encja również posiada informacje o pierwszej.

```
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private Long id;
    @ManyToOne
    private Person person;
}
```

Za pomocą atrybutu adnotacji **OneToMany** o nazwie **mappedBy** - wskazujemy nazwę pola, które odpowiada drugiej stronie relacji.

@OneToOne - jednokierunkowa

Pierwsza klasa posiada informacje o drugiej.

```
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne
    @JoinColumn(name = "details_id"
                , unique=true)
    private PhoneDetails details;
}
```

Druga encja nic nie wie o pierwszej.

```
@Entity
public class PhoneDetails {
    @Id
    @GeneratedValue
    private Long id;
}
```

@OneToOne

Dodając do adnotacji **@OneToOne** atrybut **optional=false**:

```
@OneToOne(optional=false)
```

- kolumna w bazie danych będzie unikalna oraz nie będzie mogła być **null**.

Jako alternatywę powyższego ustawienia możemy zastosować atrybut **unique=true** dla adnotacji **@JoinColumn**:

```
@JoinColumn(name = "details_id", unique=true)
```

zapewnimy w ten sposób unikalność, ale możliwa będzie wartość **null**.

@OneToOne - dwukierunkowa

Pierwsza klasa posiada informacje o drugiej.

```
@Entity
public class Phone {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne(mappedBy = "phone",
              cascade = CascadeType.ALL,
              fetch = FetchType.LAZY)
    private PhoneDetails details;
}
```

Druga encja również posiada informacje o pierwszej.

```
@Entity
public class PhoneDetails {
    @Id
    @GeneratedValue
    private Long id;
    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "phone_id")
    private Phone phone;
}
```

@ManyToMany - jednokierunkowa

Pierwsza klasa posiada informacje o drugiej.

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Long id;
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private List<Address> addresses = new ArrayList<>();
}
```

Druga encja nic nie wie o pierwszej.

```
@Entity
public class Address {
@Id
@GeneratedValue
private Long id;
}
```

@ManyToMany - dwukierunkowa

Pierwsza klasa posiada informacje o drugiej.

```
@Entity
public class Person {
    @Id
    @GeneratedValue
    private Long id;
    private String registrationNumber;
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private List<Address> addresses = new ArrayList<>();
}
```

@ManyToMany - dwukierunkowa

Druga encja również posiada informacje o pierwszej.

```
@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;
    @ManyToMany(mappedBy = "addresses")
    private List<Person> owners = new ArrayList<>();
}
```

Java Persistence Query Language (JPQL)

Jeśli chcemy znaleźć naszą encję na podstawie bardziej rozbudowanego zapytania, to możemy skorzystać z **JPQL**.

JPQL jest językiem zapytań podobnym do **SQL**.

JPQL - operuje na modelu obiektowym - wykorzystujemy nazwy klas i pól a nie jak w SQL nazwy tabel oraz kolumn.

Dla osób znających **SQL** język **JPQL** powinien być od razu zrozumiały.

JPQL

Zamiast kolumn wybieramy cały obiekt,

a zamiast tabeli - przeszukujemy naszą klasę.

```
SQL:
SELECT * FROM books;
JPQL:
SELECT b FROM Book b
```

```
SQL:
SELECT * FROM books WHERE rating > 4;
JPQL:
SELECT b FROM Book b where rating > 4
```

Przygotowywanie zapytań JPQL

Zapytania przygotowujemy używając obiektu **EntityManagera** i jego metody **createQuery()**.

```
Query query = entityManager.createQuery("SELECT b FROM Book b");
```

Za pomocą metody **getResultList()** wykonujemy zapytanie przypisując jego wynik do listy.

```
List<Book> books = query.getResultList();
```

W przypadku gdy zwracany jest tylko jeden element możemy wykorzystać metodę: **getSingleResult()**

Przygotowywanie zapytań JPQL

Jeżeli chcemy dynamicznie nastawiać wartość, według której będziemy wyszukiwać, możemy przekazać do zapytania zmienną.

```
Query queryp = entityManager.
    createQuery("SELECT b FROM Book b where rating >:rating");
queryp.setParameter("rating", 4);
List<Book> booksp = queryp.getResultList();
```

Przygotowywanie zapytań JPQL

Jeżeli chcemy dynamicznie nastawiać wartość, według której będziemy wyszukiwać, możemy przekazać do zapytania zmienną.

```
Query queryp = entityManager.
    createQuery("SELECT b FROM Book b where rating >:rating");
queryp.setParameter("rating", 4);
List<Book> booksp = queryp.getResultList();
```

Określamy nazwę zmiennej, wpisując jej nazwę ze znakiem dwukropka - **:rating**.

Przygotowywanie zapytań JPQL

Jeżeli chcemy dynamicznie nastawiać wartość, według której będziemy wyszukiwać, możemy przekazać do zapytania zmienną.

```
Query queryp = entityManager.
    createQuery("SELECT b FROM Book b where rating >:rating");
queryp.setParameter("rating", 4);
List<Book> booksp = queryp.getResultList();
```

Określamy nazwę zmiennej, wpisując jej nazwę ze znakiem dwukropka - **:rating**.

Ustawiamy wartość zmiennej określonej w zapytaniu.

Określanie limitu zwracanych danych

Jeżeli chcemy nastawić limit na liczbę zwracanych danych to możemy użyć metody **setMaxResults(n)** na naszym zapytaniu:

```
Query query = entityManager.createQuery("SELECT b FROM Book b");
query.setMaxResults(1);
```

Wykonaj zadania z działu

Relacje