Podstawowe informacje o technologii Java Persistence API

na podstawie
https://docs.oracle.com/javaee/7/JEETT.pdf

Programowanie komponentowe 4

1. Klasy typu Entity

Obiekt typu Entity (encja) jest lekkim obiektem należącym do **obiektowego modelu danych**.

Zazwyczaj **obiekt typu Entity reprezentuje tabelę w relacyjnej bazie** danych, a każde wystąpienie tego obiektu odpowiada wierszowi w tej tabeli.

Podstawowym artefaktem programowania jest klasa typu Entity, chociaż mogą one korzystać z klas pomocników.

Stan obiektu typu Entity jest reprezentowany przez atrybuty lub właściwości tego obiektu.

Te pola lub właściwości wykorzystują adnotacje mapowania obiektowo-relacyjnego do mapowania encji oraz powiązań między encjami, do danych relacyjnych w relacyjnej bazie danych.

Przykład - kod klasy **Produkt1 (konstruktor bezparametrowy – jako obowiązkowy, jest konstruktorem domyślnym)**

```
package  warstwa_biznesowa.entity;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Produkt1 implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
```

Obowiązkowe metody typu get i set

```
@Override
  public int hashCode() {
      int hash = 0;
      hash += (id != null ? id.hashCode() : 0);
      return hash;
  }

  @Override
  public boolean equals(Object object) {
      // TODO: Warning - this method won't work in the case the id fields are not set
      if (!(object instanceof Produkt1)) {
          return false;
      }
      Produkt1 other = (Produkt1) object;
      if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.id))) {
          return false;
      }
      return true;
  }

  @Override
  public String toString() {
      return „warstwa_biznesowa.entity.Produkt1[ id=" + id + " ]";
  }
}
```

1.1. Wymagania dla klas typu Entity

1) Klasa musi posiadać adnotację typu **javax.persistence.Entity**.
2) Klasa musi mieć bezparametrowy konstruktor typu **public** lub typu **protected**, oprócz innych konstruktorów.
3) Klasa nie może być zadeklarowana jako **final**. Żadne metody lub atrybuty utrwalane nie mogą być deklarowane jako **final**.
4) Jeśli instancja klasy typu Entity jest przekazywana jako parametr metod zadeklarowanych w zdalnym biznesowym interfejsie, musi **implementować interfejs typu Serializable**.
5) Klasy typu Entity mogą być następcami klas typu Entity oraz zwykłych klas, następcami klas typu Entity mogą być zwykłe klasy.
6) Atrybuty klas typu Entity, przeznaczone do utrwalania powinny być typu **private, protected, lub package-private** i mogą być dostępne tylko za pomocą **publicznych metod dostępu**. Inne obiekty mogą korzystać z wartości tych atrybutów za pomocą tych metod dostępu lub metod biznesowych.

1.2. Pola i właściwości klas typu Entity utrwalane w bazie danych

Typy utrwalanych atrybutów klas typu Entity

- proste typy języka Java
- java.lang.String
- Inne typy serializowane typy danych:
  - dane oparte na prostych typach danych
  - java.math.BigInteger
  - java.math.BigDecimal
  - java.util.Date
  - java.util.Calendar
  - java.sql.Date
  - java.sql.Time
  - java.sql.TimeStamp
  - seializowane typy danych zdefiniowane prze użytkownika
  - byte[]
  - Byte[]
  - char[]
  - Character[]
- typy wyliczeniowe
- inne encje lub kolekcje encji
- klasy wbudowame

1.2.1. Atrybuty klas typu Entity utrwalane w bazie danych

1. **Atrybuty klas typu Entity bez adnotacji javax.persistence.Transient** lub **Transient** są utrwalane w bazie danych
2. **Atrybuty jako instancje klas typu Entity** z adnotacjami mapowania obiektowo-relacyjnego są utrwalane w bazie danych za pomocą **kluczy obcych** odpowiadających kluczom głównym tych powiązanych instancji klas typu Entity, zmapowanych do odpowiednich krotek w tabelach.

1.2.2. Metody dostępu do utrwalanych atrybutów klas typu Entity wg 1.2.1

1) Wymagane są metody dostępu zdefiniowane wg ustalonego standardu:

   Atrybut utrwalany o nazwie **property** w klasie typu Entity jest typu **Type**.

   Type property;

   Wtedy metody dostepu w klasie typu Entity są następujące:

   Type getProperty()

   void setProperty(Type type)

2) Wyjątkiem są atrybuty typu Boolean:

   Boolean property;

   Type isProperty() zamiast **Type getProperty()**

1.2.3. Użycie kolekcji w definicji atrybutów i metod dostępu w klasach typu Entity

1. Do utrwalania atrybutów typu kolekcje należy używać kolekcji Javy:

■ java.util.Collection, ■ java.util.Set, ■ java.util.List ■ java.util.Map

2. Używa się również typy generyczne do zdefiniowania elementów tych kolekcji

```
@Entity
public class Person {
.....
        Set<PhoneNumber> phoneNumbers;
         public Set<PhoneNumber> getPhoneNumbers() { ... }
         public void setPhoneNumbers(Set<PhoneNumber>) { ... }
.....}
```

3. W przypadku określania, czy atrybut typu kolekcja ma być mapowany w trybie eager (zawsze), gdzie domyślnym trybem jest lazy (tylko, gdy nastąpiły zmiany), należy zdefiniować wykorzystując adnotację **@ElementCollection** oraz stałe LAZY lub EAGER typu **javax.persistence.FetchType**:

```
@Entity
public class Person {
...
        @ElementCollection(fetch=EAGER)
        protected Set<String> nickname = new HashSet();
.......}
```

1.2.4 Walidacja utrwalanych atrybutów i metod dostępu

- Api Javy do procesu walidacji typu **Bean Validation** wprowadza mechanizm do walidacji danych aplikacji. Proces Bean Validation jest zintegrowany z kontenerami Java EE, pozwalając na taką samą logikę walidacji w każdej z warstw aplikacji typu enterprise.

- Ograniczenia procesu **Bean Validation** mogą być zastosowane do utrwalanych klas typu Entity, wbudowanych klas i klas bazowych. Domyślnie, podczas tranzakcji, automatycznie wykonywana jest walidacja na utrwalanych atrybutach Encji lub metodach, z wykorzystaniem ograniczeń procesu Bean Validation - natychmiast po zdarzeniach cyklu życia PrePersist (utrwalanie), PreUpdate (modyfikacja), i PreRemove (usuwanie).

- Definiowanie ograniczeń dla procesu Bean Validation wykonuje się za pomocą **adnotacji** zastosowanych do atrybutów lub metod dostępu klasy typu Entity.

1.2.4 (cd) Walidacja utrwalanych atrybutów i metod dostępu – rola ograniczeń

- Proces Bean Validation wprowadza zbiór **ograniczeń domyślnych** oraz ograniczeń **zdefiniowanych przez programistę**.

- Te ograniczenia są specyficzną kombinacją domyślnych ograniczeń oraz nowych ograniczeń, których brakuje w domyślnych. Każde ograniczenie jest związane z jedną z klas walidacji, która dokonuje walidacji atrybutów lub metod dostępu. **Ograniczenia wprowadzone przez programistę muszą być realizowane przez odpowiednią klasę walidacji.**

- Ograniczenia proceu Bean Validation są zastosowane do utrwalanych atrybutów i metod dostępu do atrybutów utrwalanego obiektu typu Entity. Kiedy dodawane są ograniczenia walidacji, używa się się takiej samej strategii jak dla utrwalanych klas - proces walidacji jest dodawany do nich za pomocą **adnotacji wyrażających ograniczenia walidacji**. Jeśli klasa używa metod dostępu, ograniczenia są dostarczne do metod typu **get**.

1.2.4 (cd) Walidacja utrwalanych atrybutów i metod dostępu – rola ograniczeń (przykłady)

```
@Entity
public class Contact implements Serializable {
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;

@NotNull
protected String firstName;

@Pattern(regexp = "[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."+ "[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"+
"(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9]"+ "(?:[a-z0-9-]*[a-z0-9])?",message =
"{invalid.email}")
protected String email;

@Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",message = "{invalid.phonenumber}")
protected String mobilePhone;

@Pattern(regexp = "^\\(?(\\d{3})\\)?[- ]?(\\d{3})[- ]?(\\d{4})$",message = "{invalid.phonenumber}")
protected String homePhone;

@Temporal(javax.persistence.TemporalType.DATE)
@Past
protected Date birthday; }
```

1.2.4 (cd) Walidacja utrwalanych atrybutów i metod dostępu – rola ograniczeń (wyjaśnienie przykładów)

Komentarz do ograniczeń walidacji z poprzedniego slajdu:

- Adnotacja @**NotNull** na atrybucie *firstName* pecyfikuje, że te wartości tego atrybutu muszą być różne od wartości **null:**
  - — jeśli nowy obiekt typu *Contact* jest tworzony, gdzie *firstName* nie jest inicjowany, wtedy generowany jest błąd walidacji.
  - — jeśli obiekt typu *Contact* jest modyfikowany i wartość atrybutu *firstName* ma wartośc null, generowany jest błąd walidacji.

- Wartość atrybutu *email* jest walidowana za pomocą nadanego wzorca walidacji z wykorzystaniem adnotacji @**Pattern**. Do walidacji zdefiniowano wyrażenie wzorcowe za pomocą łańcucha regularnego – w przypadku różnic generowanny jest błąd walidacji. Podobnie są walidowane wartości atrybutów *homePhone* i *mobilePhone* z wykorzystaniem adnotacji @**Pattern**. Wzorzec walidacji, zdefiniowany za pomocą łańcuchów regularnych, ma 10 znaczących cyfr numeru telefonu w formacie USA i Kanada jako (*xxx*) *xxx-xxxx*

- Atrybut *birthday* musi zawierać datę z okresu przeszłego, co wynika z zastosowanej adnotacji @**Past**

1.3 Klucze główne w klasach typu Entity

Każdy obiekt typu Entity musi mieć unikatową wartość klucza głównego. Unikatowa wartość klucza głównego umożliwia identyfikację obiektu typu **Entity**. Ten obiekt może mieć prosty lub złożony typ klucza głównego:

1) **Prosty** typ klucza głównego jest definiowany za pomocą adnotacji **javax.persistence.Id** jako atrybut lub metody dostępu.
2) **Złożony** typ klucza głównego jest wtedy, gdy składa się z kilku atrybutów, które odpowiadają zbiorowi utrwalanych atrybutów i metod dostępu lub jako klasa. Złożone klucze główne są oznaczane za pomocą adnotacji **javax.persistence.EmbeddedId** i **javax.persistence.IdClass**.

Typy klucza głównego prostego lub złożonego lub metod dostępu klucza prostego lub złożonego:

- Java primitive types
- Java primitive wrapper types
- java.lang.String
- java.util.Date (typ temporal musi byc typu DATE)
- java.sql.Date
- java.math.BigDecimal
- java.math.BigInteger

1.3 Klucze główne w klasach typu Entity - wymagania (cd)

Typy zmiennoprzecinkowe nigdy nie mogą być użyte jako klucze główne.
Tylko typy całkowitoliczbowe są odpowiednie do definicji klucza głównego.

Wymagania stawiane typom kluczy głównych:

1) dostęp typu **public**.
2) metody dostępu w klasie użytej jako klucz główny muszą publicznymi metodami, jeśli muszą być wywołane od klucza głównego.
3) publiczny domyślny konstruktor.
4) klasa klucza głównego musi implementować metody **hashCode()** i **equals(Object other)**.

1) Klasa musi byc serializowana.
2) Kompozytowy klucz główny musi być reprezentowany i mapowany przez wiele atrybutów lub metod dostępu klasy typu Entity lub musi być reprezentowany i mapowany jako wbudowana klasa.
3) Jeśli klasa klucza głównego jest mapowana do wielu atrybutów lub metod dostępu klasy typu Entity, nazwy i typy atrybutów lub metod dostępu klucza głównego muszą wskazywać rodzaj klasy typu Entity

1.3 (cd) Klucze główne w klasach typu Entity - przykład

```
public final class LineItemKey implements Serializable {
private Integer customerOrder;
private int itemId;
public LineItemKey() {}
public LineItemKey(Integer order, int itemId) {
      this.setCustomerOrder(order);
      this.setItemId(itemId);
}
@Override
public int hashCode() {
   return ((this.getCustomerOrder() == null ? 0 : this.getCustomerOrder().hashCode()) ^ ((int) this.getItemId()));
}
@Override
public boolean equals(Object otherOb) {
    if (this == otherOb)
       return true;
     if (!(otherOb instanceof LineItemKey))
         return false;
      LineItemKey other = (LineItemKey) otherOb;
      return ((this.getCustomerOrder() == null
                ? other.getCustomerOrder() == null : this.getCustomerOrder().equals(other.getCustomerOrder()))
                  && (this.getItemId() == other.getItemId()));
      }
@Override
public String toString() {
       return "" + getCustomerOrder() + "-" + getItemId();
}
/* Getters and setters */ }
```

1.4 Liczność relacji dla klas typu Entity

Typy liczności relacji:

1) **One-to-one**: Każdy obiekt typu Entity jest powiązany tylko z jednym obiektem typu Entity.
2) Relacja One-to-one używa adnotacji the **javax.persistence.OneToOne** dla atrybutu lub metody dostępu reprezentujących relację.
3) **One-to-many**: jeden obiekt typu Entity jest powiązany z wieloma obiektami typu Entity. Np rachunek ma wiele pozycji. Relacja One-to-many używa adnotacji j**avax.persistence.OneToMany** dla atrybutu lub metody dostępu reprezentujących relację.
4) **Many-to-one**: Wiele obiektów typu Entity jest powiązanych tylko z jednym obiektem typu Entity. Ta liczność jest przeciwna do relacji one-to-many. W przykładzie jedna pozycja rachunku jest powiązana tylko z jednym rachunkiem. Relacja **Many-to-one** jest definiowana za pomocą adnotacji **javax.persistence.ManyToOn**e dla atrybutu lub metody dostępu reprezentujących relację.
5) **Many-to-many**: Wiele obiektów typu Entity jest powiązanych z wieloma obiektami typu Entity. Np każdy kurs ma wielu studentów, a każdy student ma wiele kursów. Relacja **Many-to-many** używa adnotację **javax.persistence.ManyToMany** dla atrybutu lub metody dostępu reprezentujących relację.

1.5. Kierunek relacji między obiektami typu Entity

1.5.1 Dwukierunkowa relacja

W dwukierunkowej relacji (**bidirectional)**, każda encja ma atrybut lub metody dostępu reprezentujące powiązanie z inną encją (implementowane za pomoca referencji do tej encji) – dzięki temu instancja klasy typu Entity ma dostęp do metod obiektu powiązanego za pomocą referencji. Np jeśli rachunek (**Order**) zna swoje pozycje (**LineItem**), to każda pozycja rachunku (**LineItem**) zna swój rachunek (**Order**).

Właściwości dwukierunkowej relacji

1) Przeciwna strona relacji dwukierunkowej jest powiązana ze stroną właściciela relacji za pomocą atrybutu **mappedBy** relacji oznaczonej za pomocą adnotacji @OneToOne, @OneToMany, lub @ManyToMany. Element **mappedBy** posiada atrybut lub metodę dostępu do obiektu, który jest właścicielem relacji.
2) Strona many relacji **many-to-one** dwukierunkowej nie musi definiować elementu oznaczonego przez **mappedBy**. Strona many jest stroną właściciela relacji.
3) W dwukierunkowej relacji one-to-one, strona właściciela i strona przeciwna mają imlementację relacji w postaci klucza głównego strony przeciwnej
4) W dwukierunkowej relacji **many-to-many,** każda strona jest właścicielem relacji.

1.5. (cd) Kierunek relacji między obiektami typu Entity

1.5.2 Jednokierunkowe relacje

Tylko jedna encja, jako strona relacji, ma atrybut lub metodę dostępu umożliwiający dostęp do strony przeciwnej. Np. pozycja (**LineItem**) rachunku (**Order**) ma atrybut relacji, które identyfikuje produkt (**Product)**, ale **Product** nie ma atrybutu lub metody dostępu do encji **LineItem** (**LineItem** „zna” **Product**, ale **Product** „nie zna” encji **LineItem**).

1.5.3 Zapytania i kierunek relacji

Java Persistence query language i **API** zapytań obiektowych często korzysta z powiązań między encjami. Kierunek relacji określa, kiedy pytanie może nawigować zjednej encji do drugiej. Np zapytanie może nawigować z encji **LineItem** do encji **Product**, jednak przeciwnej nawigacji nie można zrealizować. Natomiast encja **Order** i encje **LineItem** mogą nawigować w obu kierunkach.

1.5. (cd) Kierunek relacji między obiektami typu Entity

1.5.4 Operacje kaskadowe i powiązania między encjami

Encje, które używają relacji, są często uzależnione od istnienia encji, należących do relacji.

Np. **LineItem** jest częścią encji **Order** – jeśli **Order** zostanie usunięty, wszystkie encje **LineItem** też zostaną usunięte. Oznacza to kaskadową relację usuwania encji powiązanych.

Kaskada operacji	***Kaskadowe operacje na encjach:* javax.persistence.CascadeType**
ALL	cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}
DETACH	Jeśli encja główna jest wyłączona z kontekstu trwałości, również powiązane z nią encje są wyłączone
MERGE	Encja główna i z nią powiązane są połączone w kontekście trwałości
PERSIST	Encja główna i z nią powiązane są utrwalane w kontekście trwałości
REFRESH	Encja główna i z nią powiązane są odświeżane w kontekście trwałości
REMOVE	Encja główna i z nią powiązane są usuwane w kontekście trwałości

Kaskadowe relacje usuwania są zdefiniowane za pomocą atrybutu **cascade=REMOVE** adnotacji **@OneToOne i @OneToMany.**

Np: **@OneToMany(cascade=REMOVE, mappedBy="order")**
public Set<LineItem> getItems() { return items; }

1.5. (cd) Kierunek relacji między obiektami typu Entity

1.5.5. Usuwanie tzw „sierot w relacjach”

- Kiedy docelowa encja jest w relacjach @**OneToMany** i @**OneToOne**, często nie jest możliwe kaskadowe usuwanie docelowej encji. Takie encje są określane jako "orphans" (sierota).
- Wtedy dodaje się specjalny atrybut **orphanRemoval** do adnotacji relacji @**OnetoMany** lub @**OnetoOne**, aby uniknąć takich problemów.
- Atrybut **orphanRemoval** w adnotacjach @**OneToMany** i @**OneToOne** ma wartość logiczną i domyślnie jest ustawiona na **false**.

Np. jeśli **Order** ma wiele pozycji (**items**), i jedna z pozycji ma być usunięta, wtedy ta pozycja staje się „sierotą” przy usuwaniu. Jeśli atrybut **orphanRemoval** ma wartość true, wtedy encja typu **LineItem** jest usunięta z ze zbioru **items** należącym do encji typu **Order**.

Przykład kaskadowego usuwania encji typu **LineItem**, gdy usuwana jest encja **order**:

```
@OneToMany(mappedBy="order", orphanRemoval="true")
public List<LineItem> getItems() { return items; }
```

1.6. Klasy „do osadzania” w klasach typu Entity

Klasy Embeddable są używane do reprezentowania stanu encji, ale nie mają własnej identyfikacji trwałości – inaczej niż w przypadku klas typu Entity.

Obiekty tych zagnieżdżonych klas (np. **ZipCode**) dzielą się identyfikacją z klasą typu Entity, która ma atrybut typu klasa zagnieżdżona (np. **zipCode**).

Przykład:

```
@Embeddable
public class ZipCode {
String zip;
String plusFour;
...
}
```

Klasa „do osadzania” jest użyta przez klasę **Address** typu Entity:

```
@Entity
public class Address {
@Id
protected long id
String street1;
String province;
@Embedded
ZipCode zipCode;
String country;
...}
```

1.6 Klasy „do osadzania” w klasach typu Entity

- Klasy typu Entity mogą posiadać atrybut oznaczony adnotacją **@Embedded** – jest on wtedy częścią stanu encji.
  Ten atrybut może być oznaczony adnotacją **@Embedded** (nazwa pakietowa: **javax.persistence.Embedded**), ale nie jest ona wymagana.

- Klasy „do osadzania” mogą używać innych klas „do osadzania” lub kolekcji takich klas lub zwykłych klas w celu reprezentowania swojego stanu.

- Klasy „do osadzania” mogą zawierać relacje do innych encji lub kolekcji encji. Jeżeli mają one takie relacje, to główną relacją jest relacja między tymi encjami lub kolekcjami tych encji i encją, która jest „właścicielem” klasy „do osadzania”.

2. Dziedziczenie encji

Klasy typu Entity wspierają:

- dziedziczenie,
- relacje oparte na polimorfizmie i polimorficznych zapytaniach.

Klasy typu Entity mogą dziedziczyć:

- od klas, które nie są encjami

Klasy, które nie są encjami, mogą dziedziczyć:

- od klas typu Entity.

Klasy typu Entity mogą byc klasami abstrakcyjnymi oraz zwykłymi klasami

2. Dziedziczenie encji

2.1 Abstrakcyjne klasy typu Entity

Zapytania definiowane na abstrakcyjnej encji zwsze są realizowane na encji, która implementuje encję abstrakcyjną:

```
@Entity
public abstract class Employee {
@Id
protected Integer employeeId;
...
}
@Entity
public class FullTimeEmployee extends Employee {
protected Integer salary;
...
}
@Entity
public class PartTimeEmployee extends Employee {
protected Float hourlyWage;
}
```

2. (cd) Dziedziczenie encji

2.2 Mapowanie klas bazowych

Klasa bazowa nie jest typu Entity, ale jest zdefiniowana za pomocą adnotacji **javax.persistence.MappedSuperclass:**

```
@MappedSuperclass
public class Employee {
@Id
protected Integer employeeId;
...
}
@Entity
public class FullTimeEmployee extends Employee {
protected Integer salary;
...
}
@Entity
public class PartTimeEmployee extends Employee {
protected Float hourlyWage;
...
}
```

2. (cd) Dziedziczenie encji

2.2. (cd) Mapowanie klas bazowych

- Mapowane klasy bazowe nie mogą używane w operacjach obiektu typu EntityManager lub w zapytaniach.
- Podczas mapowania nie są tworzone tabele jako odwzorownaie klas bazowych. Tylko klasy typu Entity mogą być mapowane na tabele – wtedy również mapowane są atrybuty dziedziczone od klas bazowych oznaczonych jako @**MappedSuperclass**.

2.3 Klasy bazowe, które nie są typu Entity

- Klasy typu Entity mogą mieć klasy bazowe bez adnotacji @**MappedSuperclass** i te klasy mogą klasami abstrakcyjnymi lub konktretnymi.

- Stan tych klas nie jest utrwalany, i stan dziedziczony od takich klas bazowych przez klasę typu Entity nie jest utrwalany.

2. (cd) Dziedziczenie encji

2.4 Strategia dziedziczenia dla klas typu Entity

Dziedziczone elementy po klasach typu Entity przez dodanie do klasy, która stanowi wierzchołek drzewa dziedziczenia adnotacji **javax.persistence.Inheritance.**

Strategie dziedziczenia używane podczas mapowania:

1) Jedna tabela dla wszystkich klas typu Entity z drzewa dziedziczenia
2) Jedna tabela na każdą klasę typu Entity
3) Strategia typu "join", gdzie atrybuty lub metody dostępu specyficzne dla klasy dziedziczącej są mapowane do innej tabeli niż atrybuty lub metody dostępu klasy bazowej.

Strategie te opierają się na adnotacji @**Inheritance** zdefiniowanej dla typów wyliczeniowych: **javax.persistence.InheritanceType**

```
public enum InheritanceType {
   SINGLE_TABLE,
   JOINED,
   TABLE_PER_CLASS
};
```

Domyślna strategia InheritanceType.SINGLE_TABLE jest użyta, jeśli adnotacja @Inheritance nie jest wyspecyfikowana w drzewie dziedziczenia.

2. (cd) Dziedziczenie encji

2.4.1. Strategia mapowania - jedna tabela na klasę typu Entity

W tej strategii domyślny typ mapowania **InheritanceType.SINGLE_TABLE**, - wszystkie klasy są mapowane do jednej tabeli. Taka tabela ma kolumnę typu **discriminator column** zawierającą wartość identyfikującą typ klasy.

Taka kolumna jest znaczona adnotacją **javax.persistence.DiscriminatorColumn** w korzeniu drzewa dziedziczenia.

Typ wyliczeniowy **javax.persistence.DiscriminatorType** jest używany jako typ kolumny **discriminator** w bazie danych przez ustawienie atrybutu **discriminatorType** adnotacji @**DiscriminatorColumn**, jako jednego ze zdefiniowanych typów (poniżej).
DiscriminatorType jest zdefiniowany jako:

```
public enum DiscriminatorType {
  STRING,
  CHAR,
  INTEGER
};
```

2. (cd) Dziedziczenie encji

Jeśli nie zastosowano adnotacji **@DiscriminatorColumn do specyfikacji** korzenia drzewa dziedziczenia, a kolumna **discriminator jest wymagana**, dodawane jest narzędzie do utrwalania domyślnej kolumny o nazwie **DTYPE** i typem tej kolumny jest DiscriminatorType.STRING.

Adnotacja **javax.persistence.DiscriminatorValue** może być użyta do ustawienia wartości wprowadzonej do kolumny **discriminator** dla każdej encji z drzewa dziedziczenia. Można oznaczyć konkretne klasy typu Entity adnotacją @**DiscriminatorValue**.

Jeśli @**DiscriminatorValue** nie jest wyspecyfikowana w klasach należących do drzewa dziedziczenia, zostanie nadana tym klasom domyślna wartość. Jeśli atrybut **discriminatorType** z kolumny oznaczonej jako @DiscriminatorColumn jest DiscriminatorType.STRING, domyślną nazwą **jest nazwa encji.**

Ta strategia dostarcza dobre wsparcie dla polimorficznych relacji pomiędzy encjami i zapytaniami, pokrywającymi drzewo dziedziczenia klasy. Wymagana jest dodatkowa kolumna, która zawiera stan podklas ustawioną na nullable.

2. (cd) Dziedziczenie encji

Atrybuty adnotacji @DiscriminatorColumn

Typ	Nazwa	Opis
String	name	Nazwa kolumny używana jako kolumna typu discriminator. Domyślna nazwa to DTYPE. Ten element jest opcjonalny.
DiscriminatorType	discriminatorType	Typ kolumny użytej jako kolumna typu discriminator. Domyślny jest DiscriminatorType.STRING. Ten element jest opjonalny.
String	columnDefinition	Fragment SQL używany do tworzenia kolumny discriminator. Domyślna nazwa jest generowana przez narzędzie do utrwalania. Ten element jest opcjonalny.
String	length	Rozmiar kolumny dla typu String-based discriminator. Ten element jest ignorowany przez kolumny, które są zdefiniowane jako non-String Ten element jest ignorowany przez koumnę discriminator dla non-String typów danych. Domyślna ilość to 31. Ten element jest opcjonalny.

2. (cd) Dziedziczenie encji

2.4.2 Strategia - jedna tabela na jedną konkretną klasę typu Entity

W tej strategii, która koresponduje ze strategią **InheritanceType.TABLE_PER_CLASS**, każda konkretna klasa jest mapowana jako oddzielna tabela w bazie danych.

Wszystkie atrybuty, włączając dziedziczone atrybuty i metody dostępu są mapowane na kolumny każdej z tabel w bazie danych.

- Ta strategia wprowadza ubogie wsparcie dla polimorficznych relacji i zazwyczaj używa albo do zapytań typu SQL UNION lub oddzielnych dla każdej podklasy zapytań, które pokrywają wejściowe drzewo dziedziczenia
- Wsparcie dla tej strategii jest opcjonalne i może nie być wspierane przez Java Persistence API.
- Domyślne narzędzia Java Persistence API w serwerze GlassFish nie wspierają tej strategii.

2. (cd) Dziedziczenie encji

2.4.3 Strategia – łączenie klas dziedziczących

W tej strategii, która koresponduje ze strategią InheritanceType.JOINED, wierzchołek drzewa dziedziczenia jest reprezentowany przez pojedynczą tabelę i każda klasa dziedzicząca ma oddzielną tabelę, która zawiera tylko te atrybuty, które odpowiadają danej podklasie.Opowiadająca jej tabela nie zawiera kolumn dla dziedziczonych atrybutów i metod dostępu. Ta tabela zawiera także kolumnę lub kolumny reprezentujące własny klucz główny, który jest kluczem obcym dla tabeli superklasy.

Ta strategia:

- wprowadza dobre wsparcie dla polimorficznych relacji,
- ale wymaga jednej lub więcej operacji złączenia podczas tworzenia instancji odpowiadające podklasy. To powoduje niską wydajność dla dużych drzew dziedziczenia.
- Podobnie, pytania, które pokrywają drzewo dziedziczenia wymagają operacji złączenia tabel jako odwzorowanie wszystkich klas z drzewa dziedziczenia.

2.(cd) Dziedziczenie encji

2.4.3 (cd) Strategia – łączenie klas dziedziczących

Część narzędzii Java Persistence API, włączając domyślny mechanizm w serwerze GlassFish, wymaga kolumny **discriminator,** która odpowiada wierzchołkowi drzewa dziedziczenia, kiedy używa strategii łączenia klas dziedziczących.

Jeśli nie korzysta się z automatycznego tworzenia tabel w aplikacji, należy upewnić się, że tabela w bazie danych jest ustawiona poprawnie dla domyślnej kolumny **discriminator** lub należy użyć adnotacji **@DiscriminatorColumn** do wskazania schematu bazy danych.

3. Zarządzanie obiektami typu Entity

Klasami typu Entity zarządza obiekt **typu javax.persistence.EntityManager**. Każdy obiekt typu **EntityManager** jest powiązany z kontekstem utrwalania: zbiór zarządzanych obiektów typu Entity, które są mapowane do tabel w określonej bazie danych. Kontekst utrwalania definiuje zakres tworzenia, utrwalania i usuwania obiektów typu Entity. Interfejs EntityManager definiuje metody, które są używane do w kontekście utrwalania

3.1 Interfejs EntityManager

EntityManager API tworzy i usuwa obiekty typu Entity, wyszukuje za pomocą klucza głównego obiekty typu Entity, i umożliwia wykonanie zapytań na zbiorach obiektów typu Entity.

3.1.1 Entity Manager zarządzany za pomocą kontenera

Obiekt typu EntityManager zarządzany za pomocą kontenera ma kontekst utrwalania automatycznie przesyłany przez kontener do wszystkich komponentów aplikacji, które są używane przez obiekt typu EntityManager - przez pojedynczą tranzakcję typu Java Transaction API (JTA).

3. (cd) Zarządzanie obiektami typu Entity

3.1.1 (cd) Entity Manager zarządzany za pomocą kontenera

Tranzakcja JTA zazwyczaj uruchamia zapytania komponentów aplikacji. Do zakończenia tranzakcji JTA, te komponenty zazwyczaj potrzebują dostępu do pojedynczego kontekstu utrwalania.

To jest, kiedy obiekt typu Entity Manager jest wstrzyknięty do komponentów aplikacji za pomocą adnotacji **javax.persistence.PersistenceContext.**

Kontekst utrwalania jest przekazany przez tranzakcję JTA, i obiekt typu EntityManager jest mapowany do pliku typu persistence unit dostarczający ten sam kontekst utrwalania tranzakcji. Dzięki automatycznemu przekazywaniu kontekstu utrwalania, komponenty aplikacji nie potrzebują przekazywać między sobą referencji do obiektu typu EntityManager w porządku wykonania zmian za pomocą pojedynczej tranzakcji. Kontener Java EE zarządza cyklem życia obiektów typu Entity Manager.

Aby otrzymać obiekt typu EntityManager, należy wstrzyknąć obiekt typu EntityManager nastepująco:

```
@PersistenceContext
EntityManager em;
```

3. (cd) Zarządzanie obiektami typu Entity

3.1.2 Obiekty typu Entity Manager zarządzane aplikacją

- nie ma kontekstu utrwalania przekazywanego do komponentów aplikacji,
- cykl życia obiektu typu EntityManager jest zarządzany przez aplikację.

Obiekt typu Entity Manager zarządzany aplikacją jest używany, kiedy aplikacje potrzebują dostępu do kontekstu utrwalania, który nie jest przekazywany przez tranzakcję JTA z wykorzystaniem obiektu typu EntityManager w pliku persistence unit.

- W tym przypadku każdy obiekt typu EntityManager tworzy nowy, izolowany kontekst utrwalania. EntityManager i powiązany z nim kontekst utrwalania jest tworzony i usuwany przez aplikację.
- Są one używane również wtedy, gdy wstrzyknięty obiekt typu EntityManager nie może byż utworzony, ponieważ obiekt typu EntityManager może wywołać konflikty w operacjach wielowątkowych.

3. (cd) Zarządzanie obiektami typu Entity

3.1.2 (cd) Obiekty typu Entity Manager zarządzane aplikacją

- Aby otrzymać obiekt typu EntityManager, należy najpierw wykonać obiekt typu EntityManagerFactory za pomocą wstrzyknięcia do komponentu aplikacji za pomocą adnotacji javax.persistence.PersistenceUnit:

```
@PersistenceUnit
EntityManagerFactory emf;
```

- Potem można otrzymać obiekt typu EntityManager za pomocą obiektu typu EntityManagerFactory:

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

- Takie aplikacje muszą ręcznie propagować kontekst utrwalania tranzakcji JTA. Interfejs javax.transaction.UserTransaction definiuje metody rozpoczynające, realizujace i wycofujace tranzakcje (begin, commi, rollback). Wstrzyknięcie obiektu typu UserTransaction wykonuje się za pomocą adnotacji @Resource:

```
@Resource
UserTransaction utx;
```

3 (cd) Zarządzanie obiektami typu Entity

3.1.2 (cd) Obiekty typu Entity Manager zarządzane aplikacją

- Aby rozpocząć tranzakcję, należy wywołać metodę **UserTransaction.begin**.
- W celu zatwiedzenia tranzakcji na obiektach typu Entity, należy wywołać metodę **UserTransaction.commit** w celu zatwierdzenia tranzakcji.
- Metoda **UserTransaction.rollback** jest użyta do cofnięcia aktualnej tranzakcji.

```
@PersistenceUnit
EntityManagerFactory emf;
EntityManager em;
@Resource
UserTransaction utx;
...
em = emf.createEntityManager();
try {
        utx.begin();
        em.persist(SomeEntity);
        em.merge(AnotherEntity);
        em.remove(ThirdEntity);
        utx.commit();
} catch (Exception e) {
    utx.rollback();
}
```

3. (cd) Zarządzanie obiektami typu Entity

3.1.3 Szukanie obiektów typu Entity za pomocą obiektu typu EntityManager

Metoda **EntityManager.find** jest używana do wyszukania encji w bazie danych na podstawie klucza głównego:

```
@PersistenceContext
EntityManager em;

public void enterOrder(int custID, CustomerOrder newOrder) {
    Customer cust = em.find(Customer.class, custID);
    cust.getOrders().add(newOrder);
    newOrder.setCustomer(cust);
}
```

3.(cd) Zarządzanie obiektami typu Entity

3.1.4 Cykl życia zarzadzania obiektem typu Entity

Zarządzanie obiektami typu Entity za pomocą wywoływanych metod obiektu typu EntityManager.

Encje są w jednym z czterech stanów:**new, managed, detached, lub removed:**

1) **new** - Nowy obiekt typu Entity nie jest połączony z kontekstem utrwalania i nie ma tożsamości utrwalania.
2) **managed -** Zarządzana encja ma tożsamość utrwalania i jest powiązana z kontekstem urtwalania.
3) **detached -** Luźne encje mają tożsamość utrwalania i nie są w danym momencie powiązane z kontekstem urtwalania. .
4) **removed -** Usuwane encje mają tożsamość utrwalania i są w danym momencie powiązane z kontekstem urtwalania, i są planowane do usunięcia za bazy danych.

3. (cd) Zarządzanie obiektami typu Entity

3.1.5 Utrwalanie obiektów typu Entity

Nowy obiekt typu Entity jest zarządzany i utrwalany:

1) albo za pomocą wywołanej metody **persist**
2) lub przez kaskadowo wywoływaną **operację persist od danej encji, która ma ustawione atrybuty cascade=PERSIST lub cascade=ALL** w adnotacji relacji.

To oznacza, że dane encji są wstawiane do bazy danych, kiedy tranzakcja związana z operacją utrwalania jest kompletna.

Brak utrwalania:

1) Jeśli encja jest już zarządzana, operacja utrwalania jest ignorowana, chociaż operacje utrwalania kaskadowo sa odniesione do powiązanych encji, które mają element kaskadowy ustawiony na PERSIST lub ALL w adnotacjach relacji. Jeśli utrwalanie jest wywołane na usuwanej encji, encja staje się zarządzana.
2) Jeśli encja jest lużna, lub utrwalanie jest przerwane przez IllegalArgumentException lub tranzakcja nie jest zatwierdzona poprawnie.

3. (cd) Zarządzanie obiektami typu Entity

3.1.5 (cd) Utrwalanie obiektów typu Entity

```
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(CustomerOrder order,
                                Product product, int quantity)  {
    LineItem li = new LineItem(order, product, quantity);
    order.getLineItems().add(li);
    em.persist(li);
    return li;
}
```

Operacja utrwalania jest przekazana do wszystkich encji powiązanych z wywołaną encją, która ma atrybut cascade ustawiony na ALL lub PERSIST w adnotacji relacji:

```
@OneToMany(cascade=ALL, mappedBy="order")
public Collection<LineItem> getLineItems() {
    return lineItems;
}
```

3. (cd) Zarządzanie obiektami typu Entity

3.1.6 Removing Entity Instances

Zarzadzane encje są usuwane przez wywołanie:

1) metody **remove**
2) przez **kaskadową operacę remove wywołana od powiązanych encji, które mają ustawione atrtybuty cascade=REMOVE lub cascade=ALL w adnotacji relacji.**

Brak usuwania:

1) usuwanie jest wywołana na **nowej encji,** operacja usuwania jest ignorowana, chociaż usuwanie jest kaskadowo wywoływane u powiązanych encji, które mają ustawiony atrybut cascade na REMOVE lub ALL w adnotacji relacji.
2) usuwanie jest wywołane na luźnej encji,
3) usuwanie wywołuje przerwanie IllegalArgumentException
4) tranzakcja nie jest zatwierdzona
5) encja już jest usuwana.

3 Zarządzanie obiektami typu Entity (cd)

Poprawne usuwanie

1. dane encji są usuwane z bazy danych, kiedy tranzakcja jest zakończona
2. rezultat operacji czyszczenia jest poprawny (flush).

Przykład

Zostaną usunięte wszystkie encje typu **LineItems** powiązane z encją **order**, gdy Order.getLineItems ma atrybut cascade=ALL, ustawiony w adnotacji relacji (str. 43):

```
public void removeOrder(Integer orderId) {
    try {
            Order order = em.find(Order.class, orderId);
            em.remove(order);
}...
```

3. (cd) Zarządzanie obiektami typu Entity

3.1.7 Synchronizacja danych encji w bazie danych

Stan utrwalanych encji jest synchronizowany z bazą danych, kiedy tranzakcja na danej encji zatwierdzona.

Jeśli zarządzana encja jest w dwukierunkowej relacji z zarządzaną encją, wtedy dane są utrwalone w bazie danych, znajdujące się po stronie właściciela relacji.

W celu przeforsowania synchronizacji zarządzanej encji z bazą danych, wywołuje się metodę **flush obiektu typu EntityManager**.

Jeśli encja jest powiązana z inną encją i adnotacja relacji posiada atrybut **cascade ustawiony na PERSIST lub ALL**, powiązane encje będą synchronizowane z bazą danych, kiedy metoda flush będzie wywołana.

Jeśli encja jest usunięta, wywołanie operacji **flush** usunie dane encji z bazy danych.

3 Zarządzanie obiektami typu Entity (cd)

3.2 Persistence Unit

Plik Persistence unit definiuje zbiór klas typu Entity, które są zarządzane przez obiekt typu **EntityManager** w aplikacji.

Persistence unit jest zdefiniowany w pliku konfiguracyjcnym **persistence.xml np:**

```
<persistence>
  <persistence-unit name="OrderManagement">
    <description>This unit manages orders and customers.
        It does not rely on any vendor-specific features and  can
        therefore be deployed to any persistence provider.
    </description>
<jta-data-source>jdbc/MyOrderDB </jta-data-source>
<jar-file>MyOrderApp.jar</jar-file>
<class>com.widgets.CustomerOrder</class>
<class>com.widgets.Customer</class>
</persistence-unit>
</persistence>.
```

3. (cd) Zarządzanie obiektami typu Entity

3.2 (cd) Persistence Unit

Persistent unit może być spakowany jako część plików w formacie WAR, EJB JAR lub jako plik w formacie JAR, który może być zawarty w plikach WAR lub EAR:

- persistent unit jest spakowny w pliku EJB JAR, wtedy plik persistence.xml jest umieszczony w katalogu EJB JAR's META-INF.
- persistent unit jest spakowany w pliku WAR, wtedy plik persistence.xml jest umieszczony w katalogu WEB-INF/classes/META-INF.
- persistent unit jest spakowany w pliku JAR, który jest zawarty w plikach WAR lub EAR, wtedy plik JAR jest umieszczony w katalogach:
  - WEB-INF/lib
  - lub w katalogu library pliku EAR

4. Zapytania dotyczące klas typu Entity

Język Java Persistence API umożliwia tworzenie zapytań w następujący sposób:

- za pomocą języka zapytań Java Persistence (JPQL) - język podobny do SQL, oparty na manipulowaniu danymi
- używanie metod obiektów tworzących zapytania używając języka programowania Java API

Zarówno JPQL i metody Java API mają wady i zalety.

JPQL:

Podobne do SQL, zwięzłe, zdefiniowane przy użyciu języka programwania Java lub adnotacji w deskryptorze aplikacji

Błędy nie są wychwycone podczas kompilacji

Metody Java API:

- Zapytania zdefiniowane w warstwie biznesowej aplikacji
- Są bardziej wydajne
- Nie wymagają znajomości języka JPQL
- Wymagają często kilku operacji pomocniczych

5. Tworzenie schematu bazy danych

JPA tworzy automatycznie tabele bazy danych, ładuje dane do tabel oraz usuwa tabele podczas procesu wdrażania z wykorzystaniem standardowych właściwości w deskryptorze procesu wdrażania, nie tylko dotyczących bazy danych.

Oto zawartość deskryptora persistence.xml dotyczącego procesu wdrażania związanego z mapowaniem obiektowo/relacyjnym (następny slajd)

5. (cd) Tworzenie schematu bazy danych

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" 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">
<persistence-unit name="examplePU" transaction-type="JTA">
<jta-data-source>java:global/ExampleDataSource</jta-data-source>
<properties>
    <property name="javax.persistence.schema-generation.database.action"
              value="drop-and-create"/>
    <property name="javax.persistence.schema-generation.create-source"
              value="script"/>
    <property name="javax.persistence.schema-generation.create-script-source"
              value="META-INF/sql/create.sql" />
    <property name="javax.persistence.sql-load-script-source"
              value="META-INF/sql/data.sql" />
    <property name="javax.persistence.schema-generation.drop-source"
              value="script" />
    <property name="javax.persistence.schema-generation.drop-script-source"
              value="META-INF/sql/drop.sql" />
</properties>
</persistence-unit>
</persistence>
```

5. (cd) Tworzenie schematu bazy danych

5.1. Konfiguracja bazy danych w celu tworzenia i usuwania tabel

Właściwość:
javax.persistence.schema-generation.database.action
jest użyta **do specyfikacji akcji w bazie danych podczas wdrażania aplikacji**. Jeżeli właściwość nie jest ustawiona, artefakty bazy danych nie są tworzone i usuwane.

Przykład 1: podczas zamykania aplikacji wszystkie artefakty bazy danych są usuwane i tworzone na nowo podczas wdrażania aplikacji:

```
<property
name="javax.persistence.schema-generation.database.action"
value="drop-and-create"/>
```

5. (cd) Tworzenie schematu bazy danych

Akcje podczas tworzenia i usuwania schematu bazy danych

Ustawienia	Opis
none	Brak tworzenia lub usuwania schematu bazy danych .
create	Podczas procesu wdrażania tworzone są artefakty bazy danych, które pozostaną niezmienione podczas procesu ponownego wdrażania aplikacji.
drop-and-create	Artefakty w bazie danych są usuwane i ponownie tworzone podczas procesu.
drop	Artefakty w bazie danych są usuwane podczas procesu wdrażania aplikacji.

5. (cd) Tworzenie schematu bazy danych

W pliku **persistence.xml** są zapisane polecenia do realizacji takiego scenariusza **tworzenia schematu bazy danych**.

Właściwości zdefiniowane jako:

- **javax.persistence.schema-generation.create-source**
- **javax.persistence.schema-generation.drop-source properties**

kontrolują przebieg tych operacji.

Przykład 2:

1) Definicja skrypu **spakowanego razem z aplikacją:**

```
<property name="javax.persistence.schema-generation.create-source"
          value="script"/>
```

2) Skrypt jest zadeklarowany w pliku **persistence unit**:

```
<property name="javax.persistence.schema-generation.create-script-source"
          value="META-INF/sql/create.sql" />
```

5. (cd) Tworzenie schematu bazy danych

Ustawienia do tworzenia i usuwania właściwości bazy danych

Ustawienia	Opis
metadata	Użycie obiektowo/relacyjnych metadanych w aplikacji do tworzenia i usuwania artefaktów w bazie danych.
script	Skrypt z poleceniami do tworzenia i usuwania artefaktów bazy danych.
metadata-then-script	Użycie kombinacji obiektowo/relacyjnych metadanych oraz skryptu do tworzenia i usuwania artefaktów bazy danych.
script-then-metadata	Użycie kombinacji skryptu i obiektowo/relacyjnych metadanych do tworzenia i usuwania artefaktów bazy danych.

5. (cd)Tworzenie schematu bazy danych

5.2 Ładowanie danych za pomocą skryptów SQL

Jeśli należy **utworzyć tabele z danymi przed załadowaniem aplikacji**, należy wyspecyfikować ścieżkę dostępu do skryptu we właściwości **javax.persistence.sql-load-script-source**, w podkatalogu należacym do katalogu, w którym znajduje się plik persitence unit.

W przykładzie jest to plik **data.sql**, znajdujący się w katalogu META-INF/sql, zadefiniowany jako:

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