Java Persistence API

Imre Gábor

Q.B224

gabor@aut.bme.hu

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Outline

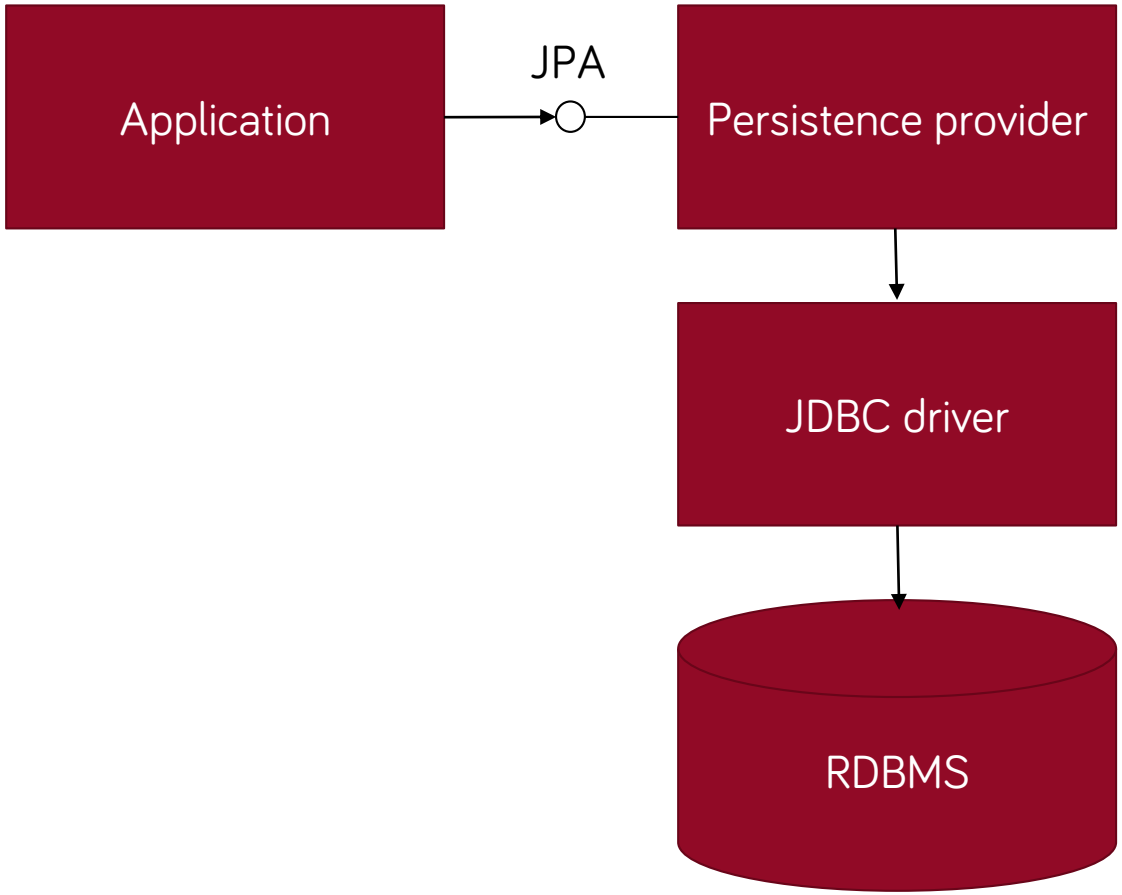
- Overview
- O-R mapping with annotations
- The persistence context
  - Lifecycle of entities
  - Database synchronization
  - Queries
- Criteria API
- Inheritance
- Relationships

Overview

General features

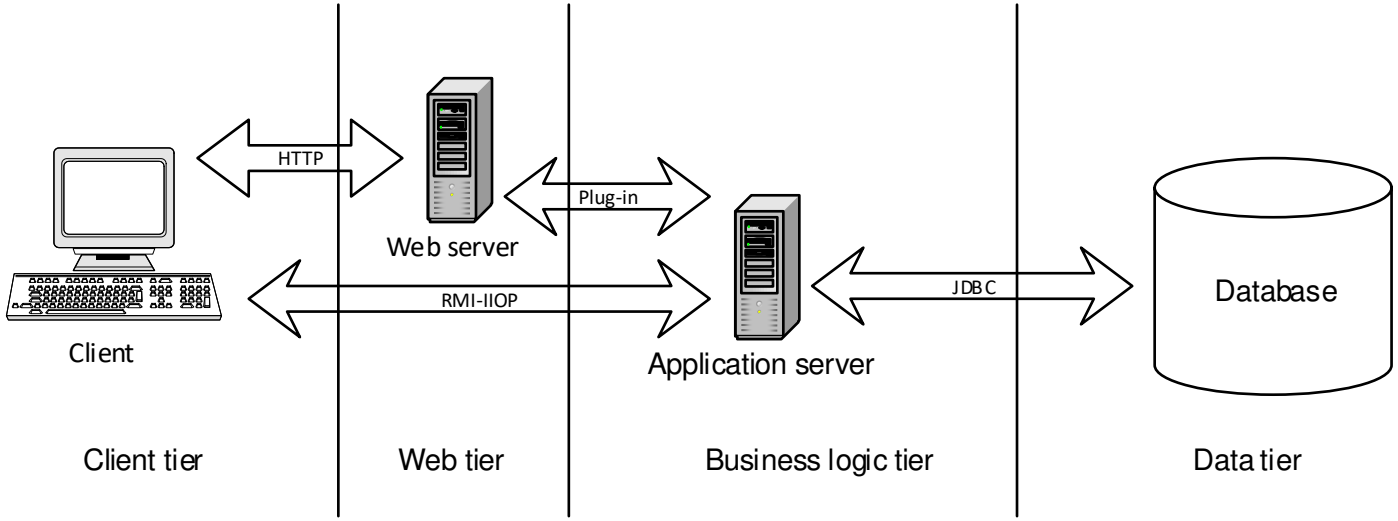
- Standard ORM API in the Java world
- Current version: **JPA 3.2**
- Specifies only interfaces→ several implementations (persistence provider – P.P.)
  - E.g. *Hibernate, EclipseLink, OpenJPA*
- JPA entity: a class whose instances are stored in a relational database by JPA
- Defined in package **javax.persistence**
  - Renamed to jakarta.persistence after 2.2

JPA architecture



Java Enterprise Edition

- JPA can be used in Java SE as well, by adding the jar files of the persistence provider to the application
- Since Java EE5 each Java EE application server contains a JPA persistence provider → no need to add JPA jar files to Java EE applications
  - Java EE: Java Enterprise Edition, adds support for server side development of enterprise-scale applications
  - N-tier architecture
  - Furhter Java EE technologies (EJB, JTA) ease the use of JPA



O-R mapping with annotations

O-R mapping with annotations

- Mandatory properties of JPA entities
  - No-arg constructor
  - **@Entity** annotation
  - Primary key attribute annotated with **@Id**
    - The primary key value can be generated via **@GeneratedValue**
    - Composed primary keys are supported via **@IdClass** or **@EmbeddedId**
- Persistent attributes are exposed as getter/setter methods

Entity example

```
@Entity
public class Employee {
  @Id
  private Integer id;
  private String name;
  private Date birthDate;

  // ... getters, setters
}
```

Customizing the O-R mapping

- Specifying the details of O-R mapping (table, column names) is optional: default values are the names of the class/attributes
- On the class: **@Table(name="MyTable")**
- On the attributes: **@Column(name="MyColumn")**
- XML configuration file can substitute annotations, but rarely used

Entity attribute types

- Primitive types and their wrappers
- String, char[], Character[]
- BigInteger, BigDecimal
- java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp
- byte[], Byte[]
  - **@Lob**
- Enum
- Other entities or collections of entities, or even non-entities
  - See Relationships
- Embedded classes
- Prevent an attribute from being stored in DB: **@Transient**

Embedded class

- A class that encapsulates some persistent attributes, but is not an entity on its own

```
@Embeddable
public class EmploymentPeriod {
  Date startDate;
  Date endDate;
  // ... getters, setters
}
```

- Using an embedded class inside an entity

```
@Entity public class Employee {
  //...
  @Embedded
  @AttributeOverrides({
    @AttributeOverride(name="startDate", column=@Column("EMP_START")),
    @AttributeOverride(name="endDate", column=@Column("EMP_END"))
  })
  private EmploymentPeriod empPeriod;
}
```

Persistence unit

- In, JPA, the persistence provider handles persistence units (P.U.)
- P.U.: a set of entities stored in the same database
- Persistence unit(s) are defined in the persistence.xml config file, packaged under the META-INF folder of the jar file containing the entity classes

```
<persistence>
   <persistence-unit name="MyPU">
      <jta-data-source>jdbc/MyDB</jta-data-source>
      <class>com.xy.Employee</class>
      <class>com.xy.Company</class>
   </persistence-unit>
</persistence>
```

Name of the P.U. Multiple persistence-unit tags are needed, when accessing multiple databases

JNDI name of the database

The entity classes. Not necessary to list them in case of Java EE.

JNDI

- Java Naming and Directory Interface
- Java API for accessing naming and directory services
- A name service can store an object registered with a name, by which the object can be looked up later
- Java EE application servers always contain a JNDI provider, for these typical use cases:
  - Components can look up each other by name
  - External resources (e.g. DB, message queue, SMTP server) can be looked up by name

Accessing databases in Java EE environment

- Drawbacks of using **DriverManager.getConnection()**:
  - Developers have to know the details of the connection (DB vendor, server, port, name of DB)
  - The open connection cannot be reused efficiently
- Solution: **DataSource** interface
  - Operation (e.g. server administrator) registers the details of the database with a JNDI name in the application server's name service
  - Developer looks up the datasource via JNDI name
  - Implements connection pooling, i.e. closing a connection requested from a datasource does not closes it physically, just marks it as free (reusable at other connection requests)

Typical JDBC code in Java EE environment

```
@Stateless
public class MyBean {
 @Resource(lookup="mydb")
 DataSource ds;

 public void myMethod() {
  try(Connection conn = ds.getConnection()) {
    ...
  } catch(Exception e){...}
 }
}
```

- Due to **@Resource** the application server performs a JNDI lookup (dependency injection)
- If we would need to do it programmatically:

```
DataSource ds = (DataSource) new InitialContext().lookup("mydb");
```

- JDBC driver class, user/pass, JDBC URL defined in the configuration file of the application server, under **mydb** name

The persistence context

Persistence context (P.C.)

- A set of in-memory entities handled by the persistence provider
- It is the connection between the in-memory entities and the database, we manipulate our entities through the P.C.
- A P.C. is accessed through the EntityManager interface, e.g.

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("MyPU");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
em.persist(new Employee(12345, "Gabor"));
em.getTransaction().commit();
em.close();
emf.close();
```

Persistence context

- An EntityManager references one P.C., each entity with a given primary key is contained at most once in a P.C.
- Managing the lifecycle of an EntityManager is challenging:
  - When should we open it?
  - When should we close it?
    - When calling another method, it would make sense, if that method could use the same EntityManager reference, so it is needless to close it → it could be a member variable
    - But if it is a member variable, and we call a method in another class, it would make sense if the same EntityManager could be seen there → should it be a method argument? In each method that wants to manipulate entities?
- Solution: dependency injection with Java EE (EJB) or Spring

Managed persistence context

```
@Stateless
class PersonService {
  @PersistenceContext
  EntityManager em;

  public void createEmployee(){
    em.persist(new Employee(12345, "Gabor"));
  }
}
```

- Condition: PersonService cannot be directly instantiated with **new**, we should ask a container to give us an instance of PersonService, and that container performs the injection of the proper EntityManager instance
- Both EJB (@Stateless) and Spring (@Service) are capable of handling such *managed persistence contexts*

Managed persistence context

- The EntityManagerFactory
  - Only one instance is created, when starting the application
  - We can inject it via **@PersistenceUnit**, but it is seldom needed
- The EntityManager member variable, annotated with **@PersistenceContext**
  - is created at the beginning of the transaction
  - is closed at the end of the transaction
  - if methods in several classes are called in the same transaction, the different EntityManager member variables will access the same P.C.

EntityManager

- The interface for manipulating entities
- 3 types of methods:
  - Handle the lifecycle of entities
  - Synchronize entities with DB
  - Query entities

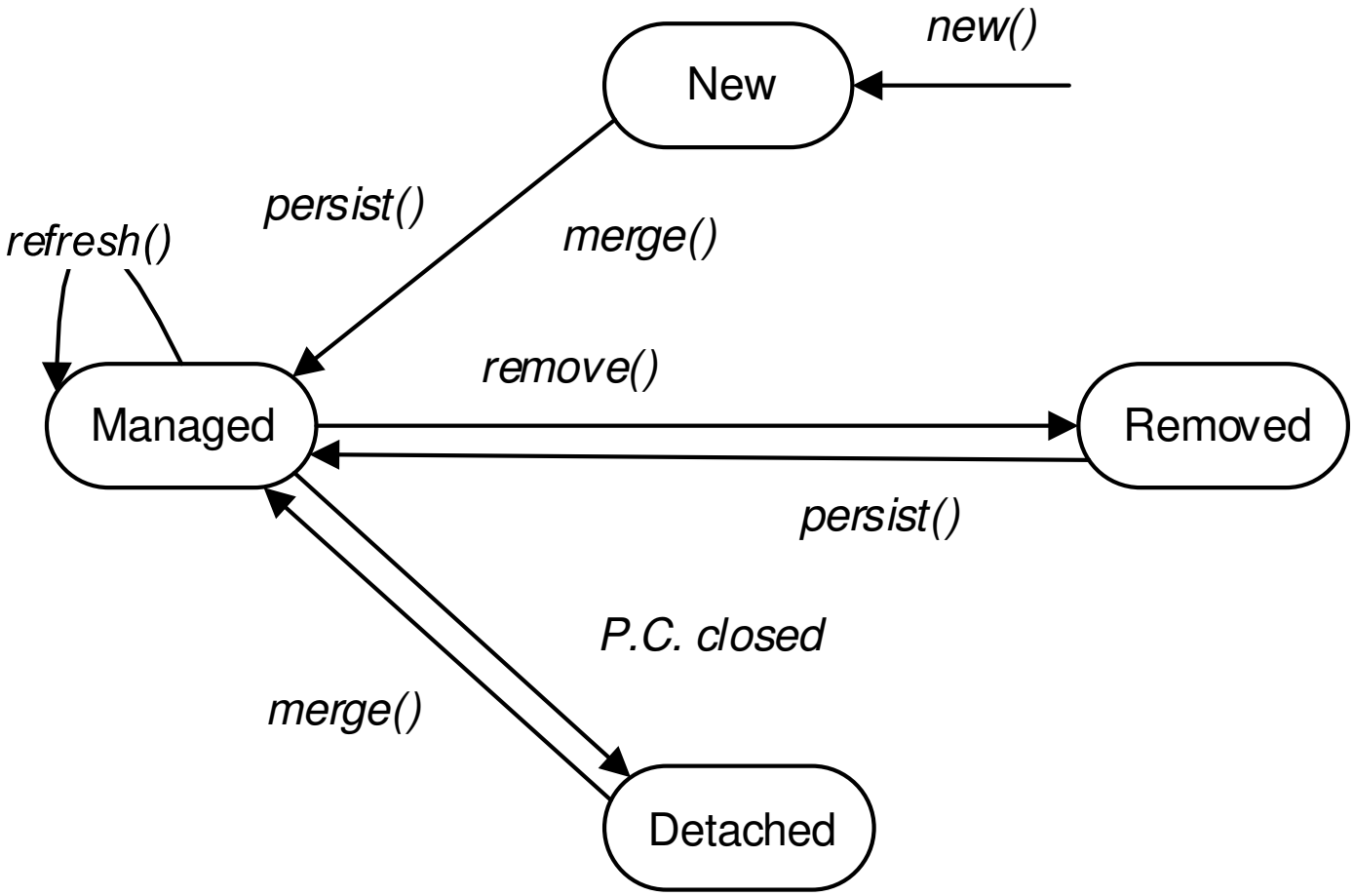
The persistence context

The lifecycle of entities

Entity states

- new: created via new, exists only in memory, is not saved to DB
- managed: exists in the DB, and is contained in a P.C. The in-memory modifications of the entity are saved into DB when the flush() method is called on the EntityManager. (This method is called automatically at transaction commit)
- detached: exists in the DB, but is not contained in the P.C.
- removed: contained in the P.C., but it is marked for removal, will be deleted from DB at the end of the transaction

The lifecycle of entities



The lifecycle of entities

- Making a new entity managed:
  - **persist()**: tries to INSERT, throws an Exception for existing primary key
  - **merge()**: performs an SQL UPDATE for existing primary key (and INSERTS for non-existing PK)
- The *return value* of **merge()** is the managed instance
- An entity becomes detached
  - After clearing the P.C. via **em.clear();**
  - After closing the P.C. via **em.close();**
  - After detaching it individually via **em.detach(entity);**

The persistence context

Database synchronization

Database synchronization

- The persistence provider uses 2 methods of the **EntityManager** to synchronize entity data with the database:
  - **flush()**: writes all the modifications of the managed entities of the P.C. into the DB
    - Rarely called explicitly, because transaction commmit calls it automaticaly
    - It is also called by default before running a query
  - **refresh(entity)**: reads the most up to date state of the entity from the DB

The persistence context

Queries

Queries

- Multiple possibilities, all through EntityManager
- Query for an entity by its primary key:
  - **<T> T find(Class<T> entityClass, Object primaryKey)**
- Custom query, dynamically created:
  - Use JPQL : **public Query createQuery(String jpqlString)**
    - JPQL (**J**ava **P**ersistence **Q**uery **L**anguage) is a language similar to SQL, but returns complete entity instances
    - E.g. **"SELECT e from Employee e WHERE e.name = :name"**
  - Use native SQL: **public Query createNativeQuery(String sqlString)**
- Custom query, defined statically, identified by name
  - **public Query createNamedQuery(String nameOfQuery)**
  - The JPQL text of the query is defined in a **@NamedQueries** annotation on an entity

```
@NamedQueries({
  @NamedQuery(name="Employee.findAll",
    query="SELECT e FROM Employee e")
})
@Entity
public class Employee { …}
```

Queries

- What can we do with a **Query**?
  - **setParameter**: by name or by index
  - Support for paging: **setMaxResult, setFirstResult**
  - Actually running the query:
    - **getSingleResult** (Exception for 0 or more than 1 result)
    - **getResultList** (empty list for 0 results)
    - **executeUpdate** (in case of UPDATE, DELETE)
- Some important JPQL features
  - Bulk delete, update
  - JOIN, GROUP BY, HAVING, subquery
  - Parameters: numbered (?1, ?2, ..) or named (:mypar)
  - Projection
    - returning only some attributes as List<Object[]>
    - or return custom new objects

JPQL examples

- **SELECT o FROM Order o WHERE o.shippingAddress.state = 'CA'**
  - Selects the Order entites that have to be shipped to California
- **SELECT DISTINCT o FROM Order o JOIN o.lineItems l WHERE l.shipped = FALSE**
  - Selects the Order entities that contain not yet shipped orderItems

Criteria API

Criteria API

- A typesafe, object-oriented alternative of the string based JPQL for creating queries (since JPA 2.0), e.g.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Employee> cq =
        cb.createQuery(Employee.class);
Root<Employee> emp = cq.from(Employee.class);
cq.select(emp);
cq.where(cb.equal(emp.get("lastName"), "Smith"));
TypedQuery<Employee> query = em.createQuery(cq);
List <Employee> rows = query.getResultList();
```

Criteria API

- More code than in case of JPQL, but the compiler protects us against type errors and typos (this version not yet completely)
- This last solution contains a string based attribute ("lastName"), this can be eliminated by using the Metamodel API
- The metamodel of the persistence unit contains metadata about the entities (and embedded classes)
- Can be generated via tools of the persistence provider, but could be written by hand as well

Criteria API

```
@Entity
public class Employee {
  @Id Long id;
  String firstName;
  String lastName;
  Department dept;
}

@StaticMetamodel(Employee.class)
public class Employee_ {
  public static volatile SingularAttribute<Employee, Long> id;
  public static volatile SingularAttribute<Employee, String> firstName;
  public static volatile SingularAttribute<Employee, String> lastName;
  public static volatile SingularAttribute<Employee, Department> dept;
}
```

Criteria API

- Our criteria queries can be completely type-safe with the use of the metamodel class, e.g.:

```
CriteriaQuery<Employee> cq =
        cb.createQuery(Employee.class);
Root<Employee> emp = cq.from(Employee.class);
cq.select(emp);
cq.where(cb.equal(
        emp.get(Employee_.lastName), "Smith"));
TypedQuery<Employee> query = em.createQuery(cq);
List<Employee> rows = query.getResultList();
```

Inheritance

Mapping inheritance in JPA

- Implementing the inhertance hierarchy of classes
  - **extends** keyword
  - **@Entity** on each entity
  - **@Id** attribute only on the topmost superclass
- The mapping of the inheritance is defined at the topmost superclass, via annotation:
  - **@Inheritance(strategy = SINGLE_TABLE)** → Mapping into a single table
  - **@Inheritance(strategy = JOINED)** → Mapping all classes
  - **@Inheritance(strategy = TABLE_PER_CLASS)** → Mapping real classes
    - Not mandatory for a P.P. To support it, because supporting polymorphism is cumbersome

Discriminator column

- The table itself determines the Java type only in case of **TABLE_PER_CLASS**
- In case of **SINGLE_TABLE** and **JOINED** an additional column is needed to store the type → discriminator column
- The name of this column is by default **DTYPE**
  - But can be ovverriden in the topmost superclass: **@DiscriminatorColumn(name="mycolumn")**
- The value stored in this column is by default the name of the entity
  - But can be ovverriden at each entity: **@DiscriminatorValue("mytype")**

Other means of inheritance

- An entity can be a subclass of a non-entity class
  - Question: should the attributes in the non-entity parent class be persistent in the entity subclasses?
    - By default not
    - But marking the non-entity superclass with **@MappedSuperClass** → yes
- Non-entity classes can inherit from entity classes
  - It won't have a corresponding table, cannot be used in queries or passed to EntityManager methods
- An entity class can be abstract
  - Cannot be instantiated, but can be mapped to a table (in case of JOINED inheritance), can be used in queries

Relationships

Mapping relationships

- The type of the member variable representing the relationship
  - Either an entity
  - Or **Collection**, **Set**, **List** or **Map** of entities
- The member variable must be annotated based on the multiplicity:
  - **@OneToOne** (Implemented with a foreign key in DB)
  - **@OneToMany** (Implemented with a foreign key in DB)
  - **@ManyToOne** (Implemented with a foreign key in DB)
  - **@ManyToMany** (Implemented with a join table in DB)
- Further possible annotations:
  - **@JoinColumn** (when omitting, default column name is used)
  - **@JoinTable** (in case of many-to-many, when omitting, default table name is used)
  - **@OrderBy** (in case of a List relationship)
  - **@MapKey** (in case of a Map relationship)

Mapping relationships

- Based on direction:
  - Uni-directional
  - Bi-directional
    - The two sides are kept consistent by the developer!!!
    - The two sides of the relationship have to be mapped to each other, e.g.
- On the owner side (Employee):

```
@ManyToOne
@JoinColumn(name="company_id")
private Company company;
```

- On the other side (Company):

```
@OneToMany(mappedBy="company")
private Collection<Employee> employees;
```

- The relationship has always one owner side. The other side references to it via mappedBy

Collection typed attributes with non-entity elements

- Available since JPA 2.0, for base types or embedded classes
- Separate table for the collection elements, with foreign key pointing on the parent entity
- But the table of the collection has neither primary key, nor corresponding entity
- Can be customized via **@ElementCollection** és **@CollectionTable**

@ElementCollection example

```
public enum FeatureType { AC, CRUISE, PWR, BLUETOOTH, TV, ... }

@Embeddable
public class ServiceVisit {
  @Temporal(DATE)
  Date serviceDate;
  String workDesc;
  int cost;
}

@Entity
public class Vehicle {
...
```

@ElementCollection example

```
@Entity
public class Vehicle {
        @Id int vin;

        @ElementCollection
        @CollectionTable(name="VEH_OPTNS")
        Set<FeatureType> optionalFeatures;

        @ElementCollection
        @CollectionTable(name="VEH_SVC")
        @OrderBy("serviceDate")
        List<ServiceVisit> serviceHistory;
        ...
}
```

Relationships

Fine-tuning

Cascade

- All 4 relationship annotations accept a cascade parameter, e.g.

```
@OneToMany(cascade={
  CascadeType.PERSIST, CascadeType.MERGE
})
```

- Possible values: PERSIST, MERGE, REMOVE, REFRESH, ALL
- It defines the EntityManager operations that should be automatically called on the related entity instances
- Default: no cascade

Fetch

- All 4 relationship annotations accept a fetch parameter, e.g.

@OneToMany(fetch=FetchType.LAZY)

- It defines, whether the related entities should be automatically loaded when loading the entity itself
- LAZY: loaded only when actually accessing the relationship → no needless memory usage, but +1 query
  - P.P. can ignore it!
- EAGER (default, except for OneToMany és ManyToMany): loaded immediately, together with the referring entity → faster, but may consume memory in vain
  - Mandatory for the P.P., not ignorable
- Possible fine-tuning:
  - Eager fetch defined in the query:

    SELECT c FROM Customer c LEFT JOIN FETCH c.orders
- Problem related to fetch
  - Entity becoming detached without the lazy-fetched relationships loaded in memory → trying to access them in detached state throws Exception