PA165 Persistence

Filip Nguyen
Lab Software Architectures and Information Systems

Lectures 02-03

September 23, September 30, 2014

Agenda 1

- Introduction to data persistence
- Persistence Layer
- Introduction to ORM
- JPA Overview
- JPA Mapping

Introduction

Introduction to data persistence

Overall goal is to build Information System that retrieves data from a datastore and allows user CRUD

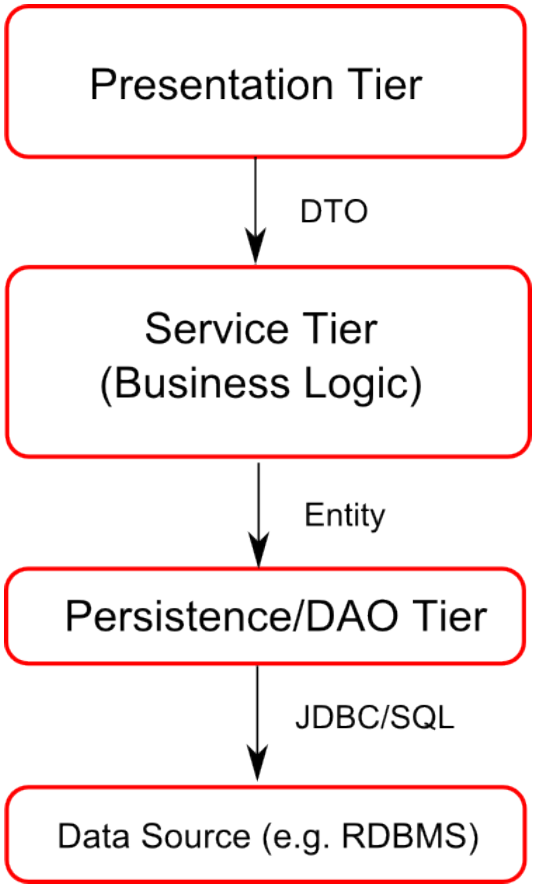
- To view the data
- Modify the data
- Delete the date
- Create the data

Persistence layer is part of the application that encapsulates logic related to accessing and manipulating sources of data.

Recap: Application Tiers/Layers

- Basic three tiers found in current systems
  - Presentation Tier
  - Business Service Tier
  - Persistence/Data Access Tier
- Access from top down. Layer sees and uses only the services from the layer directly underneath it

Diagram


Presentation Tier
DTO
Service Tier
(Business Logic)
Entity
Persistence/DAO Tier
JDBC/SQL
Data Source (e.g. RDBMS)

Where to store data

- RDBMS
- No-SQL database
- Object Database
- XML Database
- DMS (Document Management System)
- CMS (Content Management System)
- Post-relational database (Caché)
  - Temporary
  - Hierarchical
  - Spatial
- Another information system (CRM, ERP)

RDMBS

The most frequent data storage for enterprise applications

- Relational data model is simple but very powerful
- Suitable and sufficient for most of common applications
- Good theoretical model (Relational Algebra, Relational Calculus)
- Simplicity => High Performance (eg. due simple optimizations)
- Proven and well established technology (40 years of development,
- tools, standards, reliability, high penetration, lots of experts, etc.)
- Data are separated from application and can be easily shared between
- different applications
- Independent on concrete platform or programming language

Transactions

- JDBC Transactions
- ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Resource Local vs Global Transactions

Quiz 1

Which of the following are advantages of RDMBS?

1. RDMBS are transactional
2. RDBMS are based on good theoretical model
3. RDMBS are the best tools to store and retrieve enormous amounts of data

Answers 1

Which of the following are advantages of RDMBS?

1. **RDMBS are transactional**
2. **RDBMS are based on good theoretical model**
3. RDMBS are the best tools to store and retrieve enormous amounts of data

Quiz 2

Which of the following is true

1. Persistence tier can contain SQL fragments
2. Service tier contain methods that represent business function (registerUser)

Answers 2

Which of the following is true

1. **Persistence tier can contain SQL fragments**
2. **Service tier contain methods that represent business function (registerUser)**

Persistence Layer

Service Layer vs Persistence Layer

Service Layer uses Persistence Layer not vice versa

Service Layer

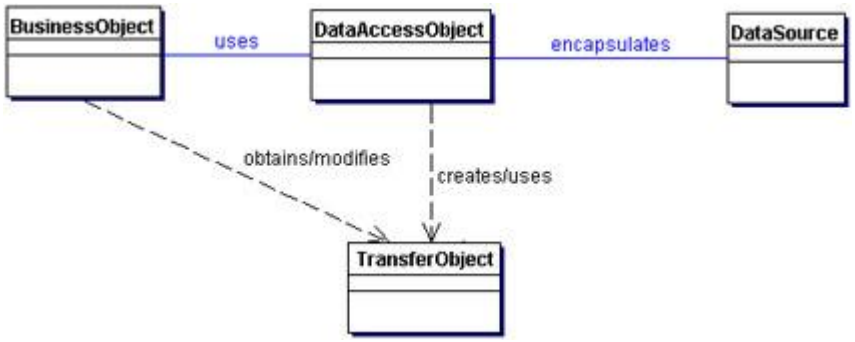
- Key element, contains business logic
- **Complete** - expose all operations client need
- **Simple -** coarse grained
- **Defined by interfaces**
- **No assumptions about DAO**
- **Handle Transactions**
- **Be easy to test**

Persistence Layer

- Encapsulates complexity of the data store
- Doesn’t contain any business logic
- Usually implemented with *Data Access Object* (DAO) design pattern

Data Access Object Pattern

- Generic interface for business objects without data store details
- **Transactions:** DAO should participate in transaction but should not usually drive them. This is because operations of the DAO are usually fine grained



Data Access Object Pattern

- **Test Driven Development:** Allows easy mocking of data access. Its easier to mock DAO interface then java.sql.Connection
- DAO is clear, strongly typed persistence API
- DAO contain finder methods with domain arguments

Data Access Object

- **Exception Handling** DAOs should throw generic data access exceptions. Throwing SQLException exposes underlying data store technology

Data Transfer Object

- Data container
- **No dependencies -** DTO should have no dependencies on anything but JDK
- **Serializable -** used to send data remotely
- We can use automatic tools to convert business objects to DTO such as Dozer:

```
Mapper mapper = new DozerBeanMapper();
DestinationObject destObject =
    mapper.map(sourceObject, DestinationObject.class);
```

Transaction Demarcation

- Its easy to demarcate transactions on DAO level. However, transactions should **NOT** be demarcated on DAO level
- It is responsibility of Service layer to demarcate transactions

Transactions on DAO level

Too limitting transaction demarcation

```
public void update(Pet pet) {
        Connection con = ds.getConnection();
        con.setAutoCommit(false);
        PreparedStatement st = con
        .prepareStatement("UPDATE PETS SET ID = ?, NAME=?, TYPENAME=?, CAGE_FK=? WHERE ID = ? ");
        st.setInt(1, pet.getId());
        st.setString(2, pet.getName());
        st.setString(3, pet.getTypename());
        st.setObject(4, pet.getCageFk());
        st.setInt(5, pet.getId());
        st.executeUpdate();
        con.commit();
}
```

Quiz 3

Which of these methods are probably suited for Service layer?

1. sellProduct
2. loadProductByID
3. deleteCustomer
4. sendPromotionToGoldenCustomers

Answers 3

Which of these methods are probably suited for Service layer?

1. **sellProduct**
2. loadProductByID
3. deleteCustomer
4. **sendPromotionToGoldenCustomers**

Quiz 4

Transactions should be demarcated on which layer?

1. Presentation Layer
2. Service Layer
3. Persistence Layer
4. Depends on situation

Answers 4

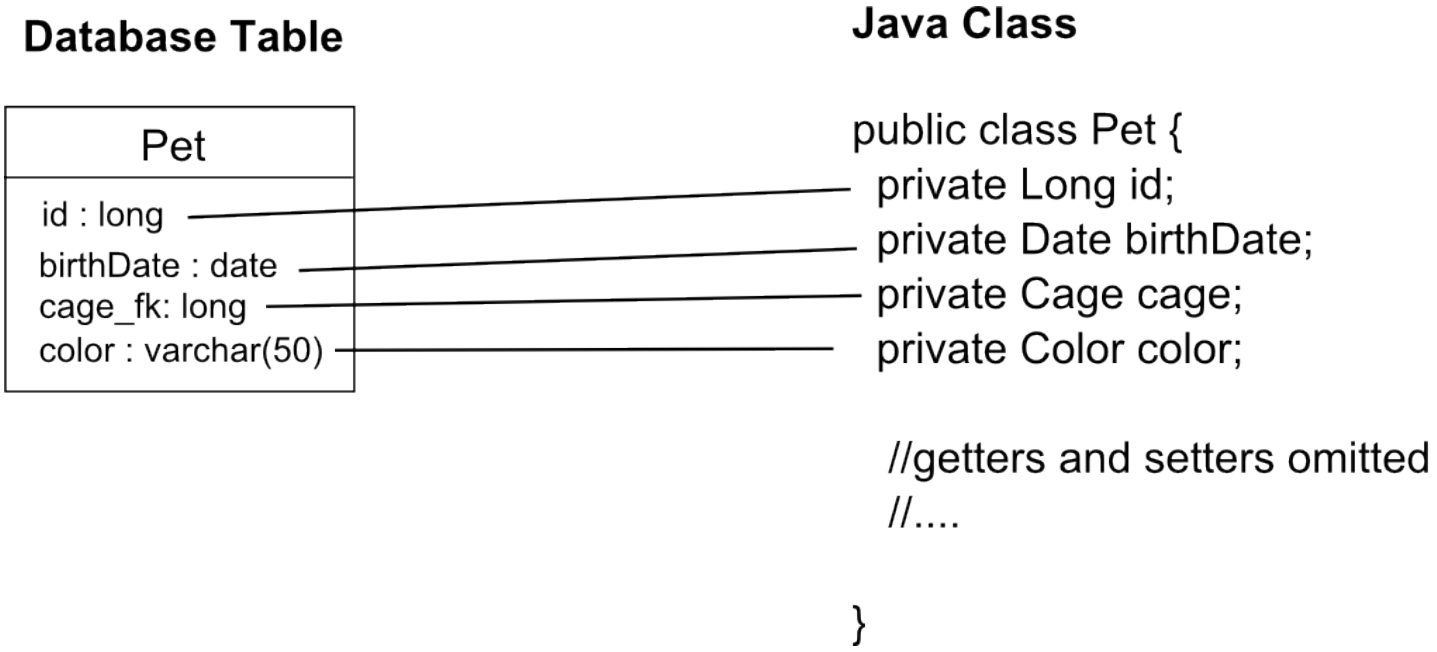
Transactions should be demarcated on which layer?

1. Presentation Layer
2. **Service Layer**
3. Persistence Layer
4. Depends on situation

Introduction to ORM

What is ORM

- Technique to map database table columns to fields of a class in OOP



Advantages and Disadvantages

- Advantages
  - Less code to write
  - SQL dialect agnostic
  - Caching in Web application development
- Disadvantages
  - learning curve
  - ORM is a big abstraction. Practical example is N+1 problem
  - lower application performance when used by inexperienced programmer

Example of ORM disadvantage N+1

- N+1 problem is a known problem with lazy loading a database

```
// Issues c.size() number of SQL statements
for (Cage c : allCages) {
  System.out.println("Is cage empty?");
  System.out.println(c.getPets().isEmpty());
}
```

Isn’t SQL/JDBC + ORM too limiting?

- Not necessarily
  - JDBC standard is relatively easy to implement
  - Any JDBC+SQL compliant driver can be easily used with Hibernate
  - Hibernate implements JPA
- Conclusion: As soon as JDBC driver is available, it’s possible to use it with existing JPA compliant code in Java

Quiz 5

Which of the following statements are true?

1. ORM acronym states for Object Real Managing
2. N+1 problem is a real performance threat
3. When we use ORM we can easily change underlying RDMBS
4. Existence of JDBC driver automatically implies possibility to use ORM such as Hibernate

Answers 5

Which of the following statements are true?

1. ORM acronym states for Object Real Managing
2. **N+1 problem is a real performance threat**
3. **When we use ORM we can easily change underlying RDMBS**
4. Existence of JDBC driver automatically implies possibility to use ORM such as Hibernate

Java Persistence API Overview

History

- Java Persistence API
  - POJO Entities, inspired by ORM tool Hibernate
  - API implemented by various ORM tools from different vendors.
  - Just basic functionality, implementations could provide other features
  - and functions through its proprietary API
- Versions and specifications
  - JPA 1.0 – part of Java EE 5; created as part of EJB 3.0 (JSR 220), but independent.
  - JPA 2.0 – part of Java EE 6; JSR 317
  - **JPA 2.1** – part of Java EE 7; JSR 338
- ORM tools implementing JPA
  - Hibernate, Open JPA
  - TopLink, TopLink Essentials, Eclipse Link

Entity

- Entity
  - Represent domain object
  - Simple POJO class
  - Attributes represents domain object properties
  - Attributes accessible with set/get methods
  - Mandatory parameterless constructor
  - It is useful (but not mandatory) to implement Serializable
- Mapping definition
  - With annotations or xml file
  - Convention-over-configuration principle

JPA Entity Example

- Pet entity

```
@Entity
public class Pet {
        @Id
        @GeneratedValue
        private long id = 0;
        @Temporal(TemporalType.DATE)
        private Date birthDate;
        @Column(nullable=false)
        private String name;
        @ManyToOne()
        private Cage cage = null;
        @Enumerated(EnumType.STRING)
        private PetColor color;
```

Quiz 6

What is POJO class?

1. Class that doesn’t extend another class
2. Class that doesn’t implement equals method
3. Class that doesn’t have long name
4. Class that contains complex logic

Answers 6

What is POJO class?

1. **Class that doesn’t extend another class**
2. Class that doesn’t implement equals method
3. Class that doesn’t have long name
4. Class that contains complex logic

Persistence Unit

- JPA Specification chapter 8.1
- Intuitively, the Persistence unit is a collection of configuration for our application. Our application needs to use JPA.
- Defined by persistence.xml file in META-INF directory
- *A persistence unit is a logical grouping that includes:*
  - *An entity manager factory and its entity managers, together with their configuration information.*
  - *The set of managed classes included in the persistence unit and managed by the entity managers of the entity manager factory.*
  - *Mapping metadata (in the form of metadata annotations and/or XML metadata) that specifies the mapping of the classes to the database.*

Persistence Unit - Example

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"      version="1.0">
        <persistence-unit name="myUnit" transaction-type="RESOURCE_LOCAL">
                <properties>
                        <property name="hibernate.dialect" value="org.hibernate.dialect.DerbyDialect" />
                        <property name="hibernate.hbm2ddl.auto" value="create-drop" />
                        <property name="hibernate.show_sql" value="true" />
                        <property name="hibernate.format_sql" value="true" />
                </properties>
        </persistence-unit>
</persistence>
```

Persistent Unit - Example

- file **META-INF/persistence.xml** must exist
- The following code constructs entry point for JPA which is EntityManagerFactory

```
EntityManagerFactory emf =
                    Persistence.createEntityManagerFactory("myUnit");
EntityManager em = emf.createEntityManager();
```

Quiz 7

Persistence unit can be used to

1. Implement a web interface
2. Provide information about SQL dialect of the data source
3. Provide list of entities mapped to the data source
4. Provide configuration for Data Access Objects

Answers 7

Persistence unit can be used to

1. Implement a web interface
2. **Provide information about SQL dialect of the data source**
3. **Provide list of entities mapped to the data source**
4. Provide configuration for Data Access Objects

Persistent Context

- JPA Specification, section 7.1
- Intuitively, persistence context is an abstract set of in memory loaded entities and a persistence context is typically associated with one EntityManager instance
- *A persistence context is a set of managed entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed by the entity manager.*
- In Java EE environment, the entity manager is typically retrieved by dependency injection:

```
@PersistenceContext
EntityManager em;
```

EntityManager

- Container-managed Entity Manager
  - Obtained via dependency injection or JNDI lookup
  - Such Entity Manager is propagated through a JTA transaction. Multiple DAOs may access the same Persistence Context
- Application-managed Entity Manager
  - Obtained from EntityManagerFactory
  - There is a new PersistenceContext created for such Entity Manager
  - Such manager must be closed by close() method
  - In Java SE, this is the only supported Entity Manager type

EntityManager - Example

```
EntityManagerFactory emf =
   Persistence.createEntityManagerFactory("myUnit");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Pet pet1 = new Pet();
pet1.setName("Sisi");
pet1.setColor(PetColor.BLACK);
em.persist(pet1);
em.getTransaction().commit();
em.close();
```

Entity - Lifecycle

- JPA Specification, section 3.2
- Entity states
  - NEW - A new entity instance has no persistent identity, and is not yet associated with a persistence context.
  - MANAGED - A managed entity instance is an instance with a persistent identity that is currently associated with a persistence context.
  - DETACHED - A detached entity instance is an instance with a persistent identity that is not (or no longer) associated with a persistence context.
  - REMOVED - A removed entity instance is an instance with a persistent identity, associated with a persistence context, that will be removed from the database upon transaction commit.

Lifecycle Operations

- JPA Specification, section 3.2
- *Entity instances are created by means of the new operation. An entity instance, when first created by new is not yet persistent. An instance becomes persistent by means of the EntityManager API.*
- *A new entity instance becomes both managed and persistent by invoking the persist method on it or by cascading the persist operation.*

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Pet pet = new Pet();
pet.setName("Tomas");
pet.setColor(PetColor.BLACK);
pet.setBirthDate(new Date());
em.persist(pet);
.....
```

Lifecycle Operations

- JPA Specification, section 3.2
- *A managed entity instance becomes removed by invoking the **remove** method on it or by cascading the remove operation.*
- *The state of a managed entity instance is refreshed from the database by invoking the **refresh** method on it or by cascading the refresh operation.*
- *An entity instance is removed from the persistence context by invoking the **detach** method on it or cascading the detach operation. Changes made to the entity, if any (including removal of the entity), will not be synchronized to the database after such eviction has taken place.*
- *The **merge** operation allows for the propagation of state from detached entities onto persistent entities managed by the entity manager.*

Quiz 8

After object is persisted and EntityManager method is closed, what is the state of the entity?

1. NEW
2. MANAGED
3. DETACHED
4. REMOVED

Answers 8

After object is persisted and EntityManager method is closed, what is the state of the entity?

1. NEW
2. MANAGED
3. **DETACHED**
4. REMOVED

Quiz 9

When we persist an object X and during after calling persist method we set a property of X by X.setName("NewValue") what will happen in database?

1. Nothing until we call EntityManager.merge()
2. The row for X will be updated with "NewValue"
3. Exception will be thrown by JPA

Quiz 10

When we persist an object X and during after calling persist method we set a property of X by X.setName("NewValue") what will happen in database?

1. Nothing until we call EntityManager.merge()
2. **The row for X will be updated with "NewValue"**
3. Exception will be thrown by JPA

JPA Configuration Wrapup

- Configuration
  - Stored in persistence.xml
  - Contains one or more Persistence Units.
- Persistence Unit
  - List of classes managed by given Persistence Unit
  - Database connection configuration
    - JNDI name of DataSource
    - JDBC url, name, password
  - Transaction control configuration (RESOURCE_LOCAL or JTA)
  - Table creation strategy
- Configuration parameter names
  - Vendor specific for JPA 1.0, standardized for JPA 2.0
  - Parameters could be also set when creating EntityManagerFactory or EntityManager

ORM vs JPA vs Hibernate

- Object Relational Mapping
  - Abstract concept of mapping RDBMS to Objects in OOP
- Java Persistence API
  - Concrete Java standard for ORM
  - Set of interfaces in javax.persistence
  - Written (PDF) specification of the requirements of the behavior of the implementation of the interfaces
- Hibernate
  - An implementation of Java Persistence API
  - Set of JAR files that implement interfaces from JPA

Java Persistence API Mapping

JPA Specification - Entities

- Chapter 2 Entities
  - 2.4 Primary Keys and Entity Identity

Entity Mapping

The entity class must be annotated with the Entity annotation or denoted in the XML descriptor as an entity.

The entity class must have a no-arg constructor.

Every entity must have a primary key. The Id annotation or id XML element must be used to denote a simple primary key.

The primary key class must define equals and hashCode methods. The semantics of value equality for these methods must be consistent with the database equality for the database types to which the key is mapped.

Relationships

- Direction
  - Uniderctional
  - Bidirectional
- Cardinality
  - OneToOne
  - OneToMany
  - ManyToOne
  - ManyToMany

Quiz 10

Which of the following are true?

1. Entity must be annotated with @Entity
2. Entity must have an equals method
3. Entity doesn’t need to have hashCode method

Answers 10

Which of the following are true?

1. Entity must be annotated with @Entity
2. **Entity must have an equals method**
3. Entity doesn’t need to have hashCode method

Bidirectional - Owning Side

- *A bidirectional relationship has both an owning side and an inverse (non-owning) side. A unidirectional relationship has only an owning side.*
- *The inverse side of a bidirectional relationship must refer to its owning side by use of the mappedBy element of the OneToOne, OneToMany, or ManyToMany annotation. The mappedBy element designates the property or field in the entity that is the owner of the relationship.*

Bidirectional - Mapping

```
@Entity
public class Employee {
        private Department department;
        @ManyToOne
        public Department getDepartment() {
                return department;
        }
}
@Entity
public class Department {
        private Collection<Employee> employees = new HashSet();
        @OneToMany(mappedBy="department")
        public Collection<Employee> getEmployees() {
                return employees;
        }
}
```

Bidirectional - saving

....

```
Department dep = new Department();
Employee e = new Employee();
e.setDepartment(dep);
em.persist(e);
em.getTransaction().commit(); //throws exception unless Department is
```

persisted in this transaction! (or persist *cascading* is set - this is more advanced concept)

Bidirectional - saving

This piece of code doesn't throw exception anymore

....

```
Department dep = new Department();
Employee e = new Employee();
e.setDepartment(dep);
em.persist(dep);
em.persist(e);
em.getTransaction().commit();
```

Bidirectional - saving

This is also possible

....

```
Department dep = new Department();
Employee e = new Employee();
e.setDepartment(dep);
em.persist(e);
em.persist(dep);
em.getTransaction().commit();
```

Bidirectional - maintain runtime consistency

```
....
Department dep = new Department();
Employee e = new Employee();
e.setDepartment(dep);
em.persist(e);
em.persist(dep);
System.out.println(dep.getEmployees().getSize()); // This will print “0”!
em.getTransaction().commit();
```

Bidirectional - maintain runtime consistency

```
....
Department dep = new Department();
Employee e = new Employee();
e.setDepartment(dep);
dep.addEmployee(e);
em.persist(e);
em.persist(dep);
System.out.println(dep.getEmployees().getSize()); // This will print “1”!
em.getTransaction().commit();
```

Maintain runtime consistency

JPA spec section 2.9:

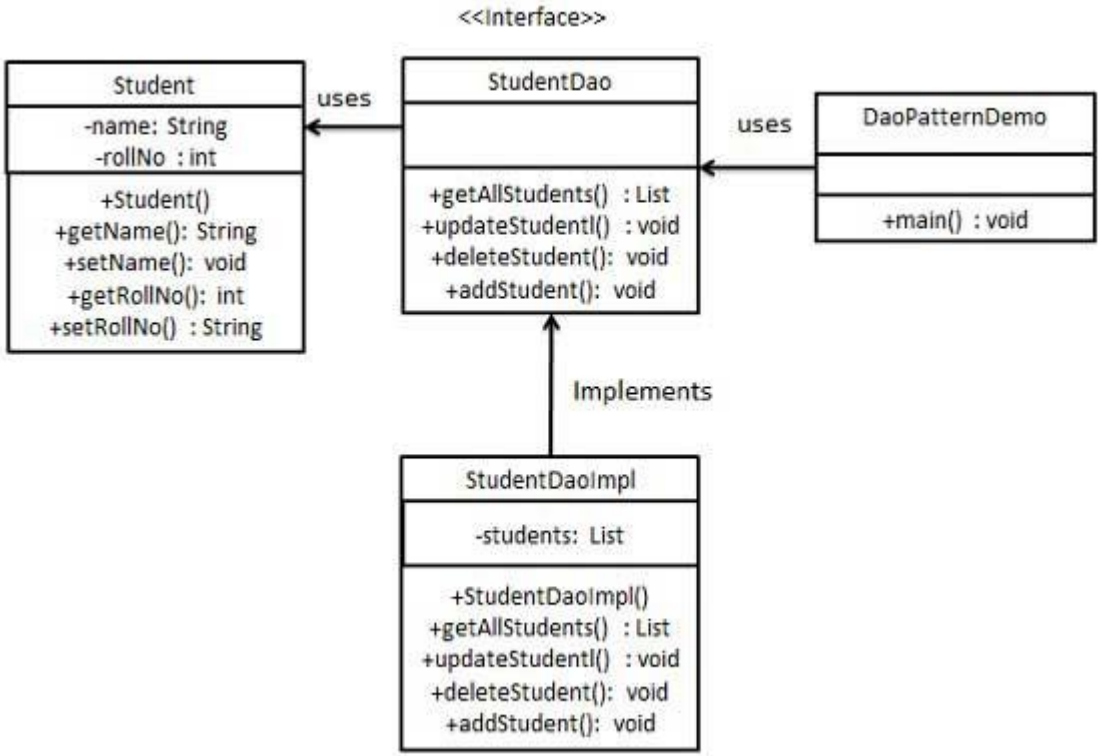
Note that it is the application that bears responsibility for maintaining the consistency of runtime relationships - for example, for insuring that the “one” and the “many” sides of a bidirectional relationship are consistent with one another when the application updates the relationship at runtime

Revising Data Access Object Pattern

- Every entity has a corresponding DAO object
  - Student entity has corresponding StudentDAO
- The DAO object contains basic data operations
  - CRUD - findById, delete, update
  - Also more complicated methods can be there “findByName”

Data Access Object Pattern

- Every DAO should have an interface and an implementation



Persistence Overview

History - EJB 2.x

- Incompatible with DAO Design Pattern
- Actually, DAO Pattern was designed as replacement for EJB 2.x Entities
- Requires Java EE Application server with EJB Container
- Entity is heavyweight component, instances are located in EJB Container and accessed remotely
- Problem with latencies (reason for introducing DAO and DTO design patterns)
- CMP versus BMP
- JPA is preferred from EJB 3.0

JDO

- JSR-243
- Java Data Objects
- Independent of underlying data technology
- Not used very much

Embedded SQL

- Code is processed with special preprocessor before compiling
- Preprocessor process SQL expressions, checks their validity,
- performs type checking a translates them into expression of used
- programming language.
- Preprocessor requires database connection

```
public String getPersonName(long personId) {
    String name;
    #sql {
        SELECT name INTO :name
        FROM people WHERE id = :personId
    };
    return name;
}
```

Spring JDBC

- Spring library implementing Template Method design pattern
- Cleaner code, faster development, easier maintenance
- Unlike ORM or Embedded SQL does not solve the problem
- with errors in SQL expressions, that become apparent until
- runtime

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
public String getPersonName(long personId) {
 return jdbcTemplate.queryForObject(
 "SELECT name FROM people WHERE id = ?",
 String.class,personId);
}
```

Java Persistence API

- We view tables in RDMBs as collection of objects

```
@Entity
public class Pet {
    @Temporal(TemporalType.DATE)
    private Date birthDate;
    @Column(nullable=false)
    private String name;
    @ManyToOne()
    private Cage cage = null;
    @Enumerated(EnumType.STRING)
    private PetColor color;
```

Java Persistence API

- Store objects

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

Pet pet = new Pet();
pet.setName("FILIP");
pet.setBirthDate(cal.getTime());
pet.setColor(PetColor.WHITE);
em.persist(pet);

em.getTransaction().commit();
em.close();
```

iBatis SQL Map

- Originally Apache project, retired, currently developed as MyBatis
- SQL Queries are separated from code
  - In XML file
  - In annotations (in new versions)
- More powerful that simple libraries like Spring JDBC, more lightweight than ORM

JPA Advanced Mapping

Load State

- JPA Specification 3.2.9
- Each attribute have default *FetchType* (important especially with collections)
- *Attributes with FetchType.LAZY may or may not have been loaded*
- *A collection-valued attribute is considered to be loaded if the collection was loaded from the database or the value of the attribute was assigned by the application, and, if the attribute references a collection instance (i.e., is not null), each element of the collection (e.g. entity or embeddable) is considered to be loaded.*

Load State

- JPA Specification 3.2.9
- Intuitively, collections have default FetchType.LAZY
  - They are loaded only after the collection is touched (typically by traversing the collection with loop)
  - Setting FetchType.EAGER may result in serious performance problems since large number of objects might be loaded from the database
  - Leaving FetchType.LAZY may also result in serious performance problems since accessing the LAZY collections with loops might result in large number of queries sent to the database

Fetching strategy

- OneToOne - Default is EAGER
- OneToMany - Default is LAZY
- ManyToOne - Default is EAGER
- ManyToMany - Default is LAZY

Relationships - Operation Cascading

- JPA Specification, section 3
- *Use of the cascade annotation element may be used to propagate the effect of an operation to associated entities. The cascade functionality is most typically used in parent-child relationships.*
- *If X is a preexisting managed entity, it is ignored by the persist operation. However, the persist operation is cascaded to entities referenced by X, if the relationships from X to these other entities are annotated with the cascade=PERSIST or cascade=ALL annotation element value or specified with the equivalent XML descriptor element.*

Operation Cascading - example

```
@ManyToOne(cascade=CascadeType.PERSIST)
 private Cage cage = null;
```

Operation Cascading

- ALL
- DETACH
- MERGE
- PERSIST
- REFRESH
- REMOVE

Embeddable

- Classes that are stored as part of owning entity, but don’t have persistent identity

```
@Embeddable
public class Address {
  private String street;
  private String city;
  ...
}
```

Embeddable

- Classes that are stored as part of owning entity, but don’t have persistent identity

```
@Embeddable
public class Address {
  private String street;
  private String city;
  ...
}
```

Embedded

- Embeddes the Embeddable class into an Entity
- In database, the Embedded instance is stored inside the table dedicated to the Entity

```
@Embedded
private Address address;
```

ElementCollection

- When an Entity needs to store list of basic types or Embeddable types. Its more consise then to create a new entity to represent the list
- On database level there must be a new table created for the elements of the collection. This new table has foreign key that is directed to the owning Entity

ElementCollection - Database

```
@ElementCollection(fetch=FetchType.EAGER)
private Set<Address> addresses=
        new HashSet<Address>();
```

create table PetStore (id bigint generated by default as identity, city...

create table (PetStore_id bigint not null, street varchar (255)...

Temporal

The Temporal annotation must be specified for persistent fields or properties of type java.util.Date and java.util. Calendar unless a converter is being applied. It may only be specified for fields or properties of these types.

```
@Temporal(TemporalType.DATE)
private Date dateOfOpening;
```

Basic

- Mapping of a field to a database column
- fetch type can be specified
- optional can be specified (nonnull)

Column

- More powerful mapping of a column
- uniqueness
- nullable
- length
- precision and scale for numbers

PersistenceUtil

- interface in javax.persistence
- in Hibernate it is implemented by PersistenceUtilHelper
- You can use the PersistenceUtilHelper to find out the load state of your collections on an Entity
  - LoadState.LOADED
  - LoadState.NOT_LOADED
  - LoadState.UNKNOWN

Java Persistence API
Querying

Introduction

- JPQL
  - Query language similar to SQL
  - Supports scalar values, tuples, entities or constructed objects
- Criteria API
  - From JPA 2.0
  - Allows to build query programatically
- Native queries
  - Queries in SQL, non portable, not integrated with JPA infrastructure
  - Use only in exceptional cases

Simple Example

```
List<Pet> pets =
  em.createQuery("SELECT p FROM Pet p",Pet.class)
  .getResultList();
```

- Developer is responsible to understand what type of result a query generates
  - Usually the result is a list of Entities or single value
  - When result is more complicated it may even become List<Object[]> which is the most general result

Creating the query using EntityManager

- **createQuery(String)**
- createNativeQuery(String)
- createNamedQuery(String)

Using the Query Object

- getResultList() - actually runs the query
- getSingleResult()
  - aggregation function COUNT, MAX, etc.
  - when query returns exactly 1 Entity
- setParameter(..) - used to supply parameters

Path Expression

- JPA Specification 4.4.4
- *An identification variable followed by the navigation operator (.) and a state field or association field is a path expression.*

```
SELECT i.name, VALUE(p)
FROM Item i JOIN i.photos p
WHERE KEY(p) LIKE ‘%egret’
```

GROUP BY

```
SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p
GROUP BY s.name
```

- The result of this query is a list of Object[]

Fetch Join

- A FETCH JOIN enables the fetching of an association or element collection as a side effect of the execution of a query.
- Intuitively, it is a **side effect** of the query

```
SELECT d
FROM Department d LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

Empty Collection Comparison Expressions

- Expression tests whether or not the collection designated by the collection-valued path expression is empty (i.e, has no elements).

```
SELECT o
FROM Order o
WHERE o.lineItems IS EMPTY
```

Named Query

- Named queries are static queries expressed in metadata or queries registered by means of the Entity-ManagerFactory addNamedQuery method.

```
@NamedQuery(name="findAll",query="SELECT p FROM Pet p")
@Entity
public class Pet {


List<Pet> pets = em.createNamedQuery("findAll",Pet.class).getResultList();
```

Constructor Expressions in SELECT

- So called SELECT NEW

```
SELECT NEW com.acme.example.CustomerDetails(c.id, c.status, o.count)
FROM Customer c JOIN c.orders o WHERE o.count > 100
```

EMPTY operator

- Can be used to check emptiness of a collection navigated by a path expression

```
SELECT c FROM Cage c WHERE c.pets IS NOT EMPTY
```

LEFT (OUTER) JOIN

- LEFT JOIN and LEFT OUTER JOIN are synonymous. They enable the retrieval of a set of entities where matching values in the join condition may be absent.

LEFT [OUTER] JOIN

join_association_path_expression [AS] identification_variable

[ON condition]

Simple Left Outer Join

SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p

SQL:

```
SELECT s.name, COUNT(p.id)
FROM Suppliers s LEFT JOIN Products p
ON s.id = p.supplierId
```

Adding ON to Left Join

- This returns all the suppliers and when a supplier has a product in status ‘inStock’ it will pair it

SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p
ON p.status = 'inStock'

SQL:

```
SELECT s.name, COUNT(p.id)
FROM Suppliers s LEFT JOIN Products p
ON s.id = p.supplierId AND p.status = 'inStock'
```

Left Join with WHERE

```
SELECT s.name, COUNT(p)
FROM Suppliers s LEFT JOIN s.products p
WHERE p.status = 'inStock'
```

- Will exclude the suppliers without products!

Parametrized Queries

```
FROM Pet p WHERE p.birthDate = :date
```

```
em.createQuery(
        "SELECT p FROM
          Pet p WHERE p.birthDate = :date",Pet.class)
          .setParameter("date", new Date())
```