nemo

ontology & conceptual modeling research group

Web Development in Java
Part III

Vítor E. Silva Souza

(vitorsouza@inf.ufes.br)

http://www.inf.ufes.br/~vitorsouza

Department of Informatics

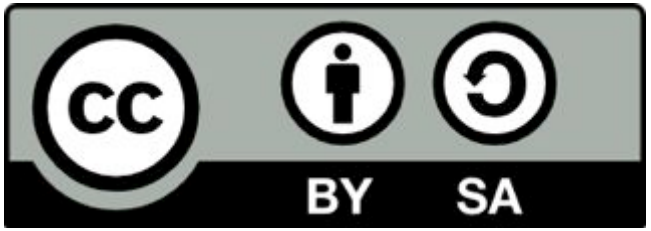
Federal University of Espírito Santo (Ufes),

Vitória, ES – Brazil

License for use and distribution

- This material is licensed under the Creative Commons license Attribution-ShareAlike 4.0 International;
- You are free to (for any purpose, even commercially):
  - Share: copy and redistribute the material in any medium or format;
  - Adapt: remix, transform, and build upon the material;
- Under the following terms:
  - Attribution: you must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use;
  - ShareAlike: if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

More information can be found at:
http://creativecommons.org/licenses/by-sa/4.0/



Outline of part III

- More on JPA: element collections, bean validation, JPQL, criteria API, etc.;
- More on EJBs: authorization, local/remote interfaces, singleton EJBs, asynchronous invocations, etc.

Web Development in Java – Part III

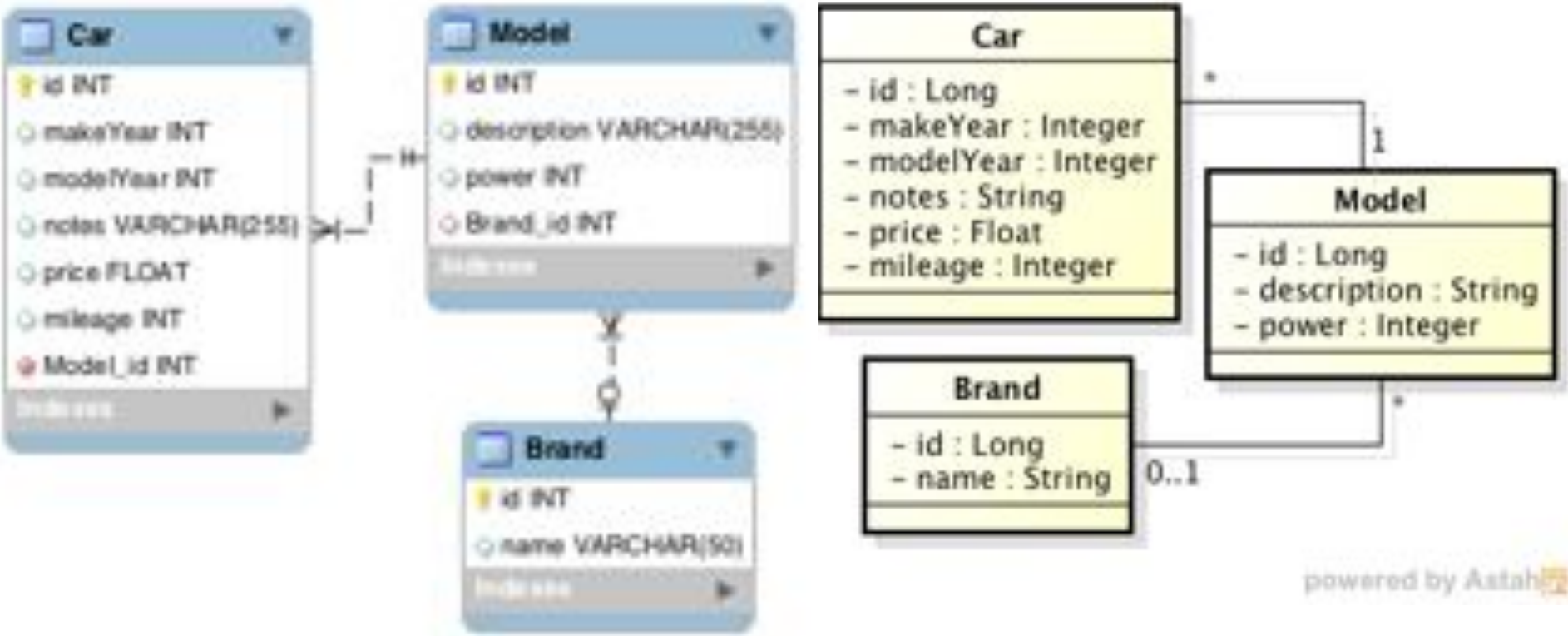
JAVA PERSISTENCE API

This section was inspired by...

- Aplicações Java para a web com JSF e JPA: http://www.casadocodigo.com.br/products/livro-jsf-jpa
  - Author: Gilliard Cordeiro;
  - Publisher: Casa do Código;
  - In Portuguese;
- Hibernate Tutorial (2007):
  - Author: me;
  - Both pt_BR and en_US;
  - Available at my website.

Motivation

- Most information systems (IS) need a database (DB);
- IS: Object Oriented / DB: Relational;
- (Object/Relational) Impedance Mismatch.



In the beginning: JDBC

- Java DataBase Connectivity;
- Standard API: packages `java.sql` and `javax.sql`;
- Vendors implemented drivers:
  - Type 1: JDBC-ODBC bridge;
  - Type 2: Native API (DB client side library);
  - Type 3: Network Protocol (DB middleware);
  - Type 4: Database Protocol (pure Java).

JDBC: example

```
// Domain class to be persisted.
public class Car {
   private Long id;
   private Integer makeYear;
   private Integer modelYear;
   private String notes;
   private Float price;
   private Integer mileage;

   // Association with class Model omitted for simplicity...

   // + getters and setters
}
```

```
// Interface for Data Access Objects that persist Car instances.
public interface CarDAO {
   void save(Car car);
   List<Car> list();
}
```

JDBC: example

```
// JDBC implementation for the Car DAO.
public class JDBCCarDAO implements CarDAO {
  @Override
  public void save(Car car) {
     String sql = "insert into Car (makeYear, modelYear, notes, " +
        "price, mileage) values (?, ?, ?, ?, ?)";
     Connection cn = openConnection();  // Implemented separately.
     try {
        PreparedStatement pst = cn.prepareStatement(sql);
        pst.setInt(1, car.getMakeYear());
        pst.setInt(2, car.getModelYear());
        pst.setString(3, car.getNotes());
        pst.setFloat(4, car.getPrice());
        pst.setInt(5, car.getMileage());
        pst.execute();
     } catch (SQLException e) { throw new RuntimeException(e); }
     finally {
        try { cn.close(); }
        catch (SQLException e) { throw new RuntimeException(e); }
     }
  }

  // Continues...
```

JDBC: example

```
public List<Car> list() {
   List<Car> cars = new ArrayList<>();
   String sql = "select * from Car";
   Connection cn = openConnection();
   try {
      PreparedStatement pst = cn.prepareStatement(sql);
      ResultSet rs = pst.executeQuery();
      while( rs.next() ) {
         Car car = new Car();
         car.setId(rs.getLong("id"));
         car.setMakeYear(rs.getInt("makeYear"));
         car.setModelYear(rs.getInt("modelYear"));
         car.setNotes(rs.getString("notes"));
         car.setPrice(rs.getFloat("price"));
         car.setMileage(rs.getInt("mileage"));
         cars.add(car);
      }
   } catch (SQLException e) { throw new RuntimeException(e); }
   finally {
      try { cn.close(); } catch (SQLException e) { /* ... */ }
   }
   return automoveis;
}
```

JDBC: example

```
// Somewhere else...
public Connection openConnection() {
   Connection conn = null;

   String url="jdbc:mysql://localhost:3306/cars";
   Class.forName("com.mysql.jdbc.Driver");
   conn = DriverManager.getConnection(url, ”user", ”password");

   return conn;
}
```

- Very complex code, even for the simplest operation;
  - Awareness of the relational model;
  - Lots of exception handling;
  - Etc.
- But also highly generalizable…

The paradigm mismatch

- Some problems:
  - Granularity: limited to table and column;
  - Inheritance: storage and polymorphism;
  - Identity: == vs. equals() vs. primary key;
  - Associations: transposition of primary keys;
  - Navigation on the object graph: the n+1 SELECTs problem.
- The cost:
  - 30% of the code used to manipulate data;
  - Domain model twisted to fit the data model;
  - Software becomes hard to maintain.

Alternatives

Solution	Problems
Hand-coded SQL/JDBC	• Wasted effort; • Low productivity and high maintenance; • Possibly lower performance compared to existing solutions.
Serialization	• Access to entire graph; • No searching; • Concurrency issues.
Entity EJBs	• Twists the object model; • No support for polymorphism and inheritance; • Not portable in practice; • Not serializable; • Intrusive model that makes unit testing very hard.
OO Databases	• Low market acceptance; • Immature standard.

Enter Object/Relational Mapping

In a nutshell, object/relational mapping is the automated (and transparent) persistence of objects in a Java application to the tables in a relational database, using metadata that describes the mapping between the objects and the database. ORM, in essence, works by (reversibly) transforming data from one representation to another.

Gavin King

Hibernate in Action

Components of an ORM solution

- API for the execution of CRUD operations;
- Language or API for the construction of queries that refer to the classes and their properties;
- Specification of the mapping metadata;
- RDBMS interaction techniques, including:
  - Dirty checking;
  - Lazy association fetching;
  - Optimization functions.

An ORM solution should specify

- How persistent classes and metadata should be written;
- How to map hierarchies of classes;
- How do object identity and table line identity relate;
- What is the lifecycle of a persistent object;
- How to retrieve data from associations in an efficient way;
- How to manage transactions, cache and concurrency.

Advantages of ORM

- Productivity: eliminates plumbing code.
- Maintainability:
  - Less lines of code, less maintenance;
  - Changes in data structure do not impact as much.
- Performance:
  - More time to implement optimizations;
  - More knowledge of each RDBMS detail.
- Vendor independence: use of SQL dialects.

Disadvantages of ORM frameworks

- They are not easy to learn;
- To use them well, you should also know well SQL and relational database technology;
- Problems that come from their use are complex and hard to solve.

ORM frameworks are not the "silver bullet" of persistence!

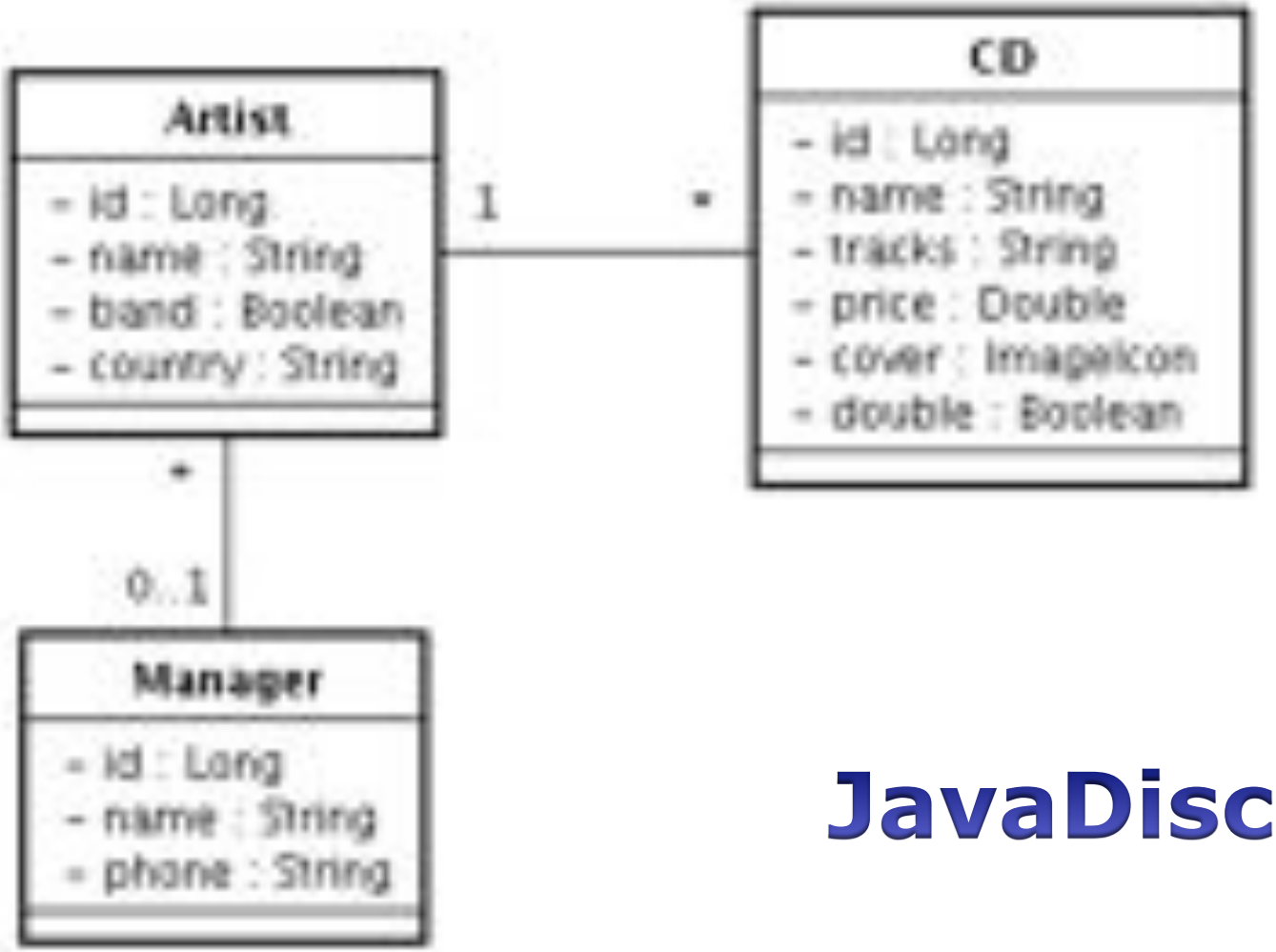
Hibernate

- First (started in 2001) and most popular ORM solution;
- Full ORM solution, basis for JPA 2.0;
- Greatly simplifies our DAOs:

```
// Hibernate implementation for the Car DAO.
public class HibernateCarDAO implements CarDAO {
  @Override
  public void save(Car car) {
     Session session = openSession(); // Implemented separately.
     Transaction tx = session.beginTransaction();
     session.save(car);
     tx.commit();
     session.close();
  }

  // list() implemented with:
  // session.createQuery("from Car");
```

A full example with Hibernate

- From an old tutorial of mine…



A full example with Hibernate

```
// Domain class to be persisted.
package hibernatetutorial.domain;
public class Artist {
   private Long id;
   private String name;
   private Boolean band;
   private String country;

   /* Implicit construtor. */

   /* Properties getters and setters. */
}
```

```
// The table in the (HSQLDB) database.

CREATE TABLE Artist (
   id BIGINT NOT NULL IDENTITY,
   name VARCHAR(100) NOT NULL,
   band BIT NULL,
   country VARCHAR(50) NOT NULL,
   PRIMARY KEY(id)
);
```

A full example with Hibernate

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping ... >
<hibernate-mapping>
  <class name="hibernatetutorial.domain.Artist” table="Artist">
    <id name="id" column="id”><generator class="native" /></id>
    <property name="name" column="name” type="string" length="100" />
    <property name="band" column="band” type="boolean" />
    <property name="country" column="country” type="string"
        length="100" />
  </class>
</hibernate-mapping>
```


```
<class name="hibernatetutorial.domain.Artist">
  <id name="id">
    <generator class="native" />
  </id>
  <property name="name" length="100" />
  <property name="band" />
  <property name="country" length="100" />
</class>
```

A full example with Hibernate

```
/*
 * Alternatively, we can use Hibernate Annotations:
 */

@Entity
public class Artist {
  @Id @GeneratedValue(strategy = GenerationType.AUTO)
  public Long getId() { return id; }

  @Column(length = 100)
  public String getName() { return name; }

  /* ... */
}
```

A full example with Hibernate

```
/*
 * Saving objects with Hibernate.
 */

// Imports from org.hibernate.*

// Creates an object.
Artist artist = new Artist();
artist.setName("Dave Matthews Band");
artist.setBand(true);
artist.setCountry("USA");

// Obtains a session (implemented later).
Session session = HibernateUtil.openSession();

// Stores on database using Hibernate.
Transaction tx = session.beginTransaction();
session.save(artist);
tx.commit();
session.close();
```

A full example with Hibernate

```
/*
 * Retrieving objects with Hibernate.
 */

// Obtains a session (implemented later).
Session session = HibernateUtil.openSession();

// Retrieves all artists.
Transaction tx = session.beginTransaction();
Query query = session.createQuery("from Artist a order by a.name");
List result = query.list();

// Prints and closes the connection.
for (Object o : result) System.out.println(o);
tx.commit();
session.close();

// Artist must implement toString() for printing.
```

A full example with Hibernate

```
/*
 * The utility class for opening Hibernate sessions.
 */

public final class HibernateUtil {
  private static SessionFactory sessionFactory;

  private static SessionFactory getSessionFactory() {
    // Reads configuration form hibernate.cfg.xml
    if (sessionFactory == null) sessionFactory = new
Configuration().configure().buildSessionFactory();

    return sessionFactory;
  }

  public static Session openSession() {
    return getSessionFactory().openSession();
  }
}
```

A full example with Hibernate

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration ... >

<hibernate-configuration>
 <session-factory>
    <property name="connection.driver_class">
        org.hsqldb.jdbcDriver</property>
    <property name="connection.url">
        jdbc:hsqldb:hsql://localhost/javadiscs</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>

    <property name="connection.pool_size">1</property>
    <property name="dialect”>
        org.hibernate.dialect.HSQLDialect</property>
    <property name="current_session_context_class”>thread</property>
    <property name="cache.provider_class”>
        org.hibernate.cache.NoCacheProvider</property>
    <property name="show_sql">true</property>

    <mapping resource="hibernatetutorial/domain/Artist.hbm.xml” />
  </session-factory>
</hibernate-configuration>
```

Hibernate is awesome. Why JPA?

- Once ORM was recognized as a great idea, many frameworks were developed;
- Portability was a problem – vendor lock-in;
- In 2006, the Java Persistence API was proposed to solve this problem:
  - JPA 2.0 in 2009, 2.1 in 2013;
  - Very similar to Hibernate.

Hibernate vs. JPA

```
/* Hibernate. */

Session session = HibernateUtil.openSession();
Transaction tx = session.beginTransaction();
session.save(artist);
tx.commit();
session.close();
```

```
/* JPA. */

EntityManager em = JPAUtil.openEntityManager();
em.getTransaction().begin();
em.persist(artist);
em.getTransaction().commit();
em.close();
```

Hibernate Query Language (HQL) → Java Persistence Query Language (JPQL)

Let's learn JPA!

Configuration

O/R Mapping

Basic operations

JPQL

Criteria API

Bean Validation

JPA Configuration

- Done in a file called persistence.xml, in a folder called META-INF in the classpath of the project:

```
<?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="JavaDiscs">
    <properties>
        <property name="javax.persistence.jdbc.url"
            value="jdbc:hsqldb:hsql://localhost/javadiscs" />
        <property name="javax.persistence.jdbc.user" value="sa" />
        <property name="javax.persistence.jdbc.password" value="" />
        <property name="javax.persistence.jdbc.driver"
            value="org.hsqldb.jdbcDriver" />
    </properties>
  </persistence-unit>
</persistence>
```

JPA Configuration

- We can also use data sources configured in the application server (as in JavaHostel):

```
<?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="JavaHostel">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <jta-data-source>
        java:/jboss/datasources/JavaHostel
    </jta-data-source>

    <!-- ... -->

  </persistence-unit>
</persistence>
```

JPA Configuration

- `javax.persistence` properties are standard, but we can set specific properties of our chosen ORM framework;
- Some interesting options for Hibernate:
  - `hibernate.dialect`: the database dialect to use (optional, usually Hibernate can infer from driver);
  - `hibernate.hbm2ddl.auto`: creation of the tables: `create`, `create-drop`, `update` or `validate`;
  - `hibernate.show_sql`: print the SQL prepared statements before executing them;
  - `hibernate.format_sql`: format the SQL when printing.

Configuration

Getting the entity manager

- When using CDI (e.g., JavaHostel), we can request the injection of the entity manager:

```
@Stateless public class ArtistJPADAO implements ArtistDAO {
  @PersistenceContext private EntityManager em;
}
```

- Otherwise, we need to create it:

```
// import javax.persistence.Persistence;
EntityManagerFactory emf = Persistence.
    createEntityManagerFactory("JavaDiscs"); // persistence-unit name
EntityManager em = emf.createEntityManager();

Artist artist = new Artist(); // fill in some data in artist.

EntityTransaction tx = em.getTransaction();
tx.begin();
em.persist(artist);       // Note that without the transaction the
tx.commit();              // data would not have been persisted!
em.close();
emf.close();
```

Entity manager types

- Using CDI vs. creating your own generates different types of entity managers:
  - Container-managed: CDI injects an entity manager (@PersistenceContext) associated with a JTA transaction and manages the EM lifecycle;
  - Application-managed: when created manually, transactions and the lifecycle have to be managed by the application (as the previous example).
- Application-managed with CDI:
  - `@PersistenceUnit EntityManagerFactory emf;`
  - `@Resource UserTransaction utx;`

Configuration

JTA? What the hell is JTA?

- Java Transaction API: another part of Java EE;
- JTA allows you to create distributed transactions across multiple X/Open XA resources (data sources, message queues, etc. in different machines) in Java;
- JTA + EJB (Session and Message-driven):
  - Transaction context associated with method call;
  - `begin()` and `commit()` called automatically;
  - `rollback()` called if an application exception is thrown or `setRollbackOnly()` used in the (injected) context;
  - We can change the behavior with annotations.

Configuration

Factory and sessions/managers

- Hibernate session factory and JPA entity manager factory are heavy objects:
  - Take a long time to create;
  - There should be one per persistence unit;
  - Thread-safe.
- Hibernate sessions and JPA entity managers are lightweight objects (the opposite of the above).

```
public final class JPAUtil {
  private static final EntityManagerFactory emf = Persistence.
    createEntityManagerFactory("JavaDiscs");

  public static EntityManager getEntityManager() {
    return emf.createEntityManager();
  }
}
```

Configuration

Last piece of configuration

- We need to tell JPA which classes are entities, i.e., will be mapped to the database:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" ...>

  <persistence-unit name="JavaHostel">
    <!-- ... -->

    <class>br.ufes.inf.nemo.javahostel.domain.Bed</class>
    <class>br.ufes.inf.nemo.javahostel.domain.Booking</class>
    <class>br.ufes.inf.nemo.javahostel.domain.Guest</class>
    <class>br.ufes.inf.nemo.javahostel.domain.Room</class>

  </persistence-unit>
</persistence>
```

According to the JPA specification, however, specifying the mapped classes should not be necessary in a Java EE environment with a single persistence unit. In theory…

Configuration

Multiple persistence units

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" ...>

  <persistence-unit name="employees">
    <class>mysystem.entities.Employee</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
        ...
    </properties>
  </persistence>

  <persistence-unit name="providers">
    <class>mysystem.entities.ServiceProvider</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
        ...
    </properties>
  </persistence>

</persistence>
```

Configuration

Configuration

O/R mapping

- Key concept in ORM frameworks;
- Instructs the framework on:
  - How to create tables automatically;
  - How to store object data in tables;
  - How to convert data retrieved from tables into objects;
  - How to perform queries;
  - Etc.

In this section we assume you will use this feature of your ORM framework. Therefore, we will not show how to create the tables that correspond to the O/R mapping.

First of all: domain classes should…

- Be API-independent (no JDBC, Swing, Web or anything);
- Not worry about cross-cutting concerns (persistence, transaction management, logging, etc.);
- Have an artificial ID to serve as PK in the DB;
- Use interfaces when defining collection attributes;
- Follow the JPA specification:
  - Have a no-arg constructor, but it can be protected;
  - Be a top-level class (not an enum, not an interface);
  - Not be final or have final attributes/methods;
  - Implement Serializable.

O/R Mapping

Minimum mapping

- The only thing JPA **really** needs is that you:
  - Specify the class is an @Entity;
  - Indicate which attribute is the @Id.

```
@Entity
public class Artist {
  @Id @GeneratedValue
  private Long id;
  private String name;
  private Boolean band;
  private String country;    // getters and setters if needed.
}
```

```
@Entity
public class Book {
    @Id
    private String isbn;
    // ...
}
```

O/R Mapping

Identity generation

- There are four basic generation strategies:
  - AUTO: JPA will choose (default);
  - IDENTITY: use auto-increment columns*;
  - SEQUENCE: use one or more global sequence value (sequenceName can be specified)*;
  - TABLE: use an auxiliary table.

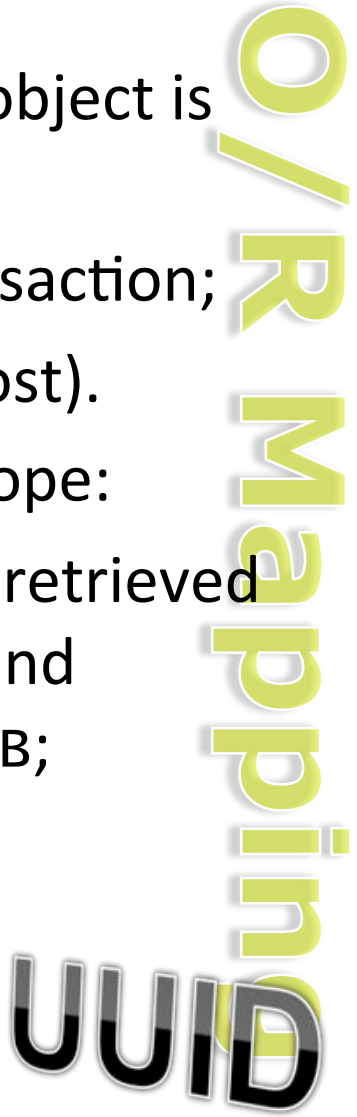
* Depends on database support.

Understanding object identity

- Two objects, A and B, can be:
  - Identical: (A == B) is true;
  - Equal: (A.equals(B)) is true;
  - Database-identical: they represent the same line, i.e., they are on the same table and have the same primary key value.
- Questions:
  - Should identical objects be equal and vice-versa? Should DB-identical objects be equal and vice-versa?
  - Should equals() and hashCode() be based on the ID?
  - Should we use natural or artificial IDs?

The scope of object identity

- An ORM solution can have as scope:
  - None: there are no guarantees that the same object is returned if the same query is made twice;
  - Transaction: the guarantee exists within a transaction;
  - Process: it exists within the entire JVM (high cost).
- Hibernate (JPA?) has session (EntityManager?) scope:
  - If A and B are objects from the same class, are retrieved by the same Session (EntityManager?) object and `A.getId().equals(B.getId())`, then `A == B`;
  - This is called “first level cache”;
  - The JPA `hashCode()` / `equals()` dilemma: http://stackoverflow.com/questions/5031614/the-jpa-hashcode-equals-dilemma



Composite IDs (if you must…)

- JPA supports IDs composed by multiple properties:
  - Using an embeddable ID (example below); or
  - Using an ID class.

```
@Embeddable
public class ClientProductPK implements Serializable {
  private Long clientId;
  private Long productId;
  // ...
}

@Entity
public class ClientProduct {
  @EmbeddedId
  private ClientProductPK id;
  //...
}
```

Versioning (optimistic locking)

- Three approaches: optimistic, pessimistic and *ostrich*;
- Optimistic locking assumes conflicts will not occur, but detects when they happen and prevent the overwrite;
- Conflicts are detected by checking that values of the object originally read remain the same;
- JPA does that with a version column:

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

  // So simple. Why ignore it?
  @Version
  private long version;
}
```

O/R Mapping

Pessimistic locking (from JPA 2.0)

- Lock modes:
  - None: no locking;
  - Optimistic: new name for “read”, which already existed, = optimistic lock;
  - Optimistic, with force increment: new name for “write”, which also existed;
  - Pessimistic read: locks for writing (repeatable read);
  - Pessimistic write: locks for everything (serialization);
  - Pessimistic, with force increment: same as before, but forcing the increment of the version column.

```
// cq is some CriteriaQuery that returns a single employee...
Employee emp = em.createQuery(cq).getSingleResult();
em.lock(emp, LockModeType.PESSIMISTIC_READ);
```

Simple attributes (not associations)

- Can be @Transient, @Basic, @Temporal or @Lob;
- By default, attributes are @Basic (persistent).

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

  @Basic
  private Double salary;

  @Temporal(TemporalType.DATE)  // Or: TIME, TIMESTAMP
  private Date birthDate;

  @Transient
  private int age;
  // Alternatively: private transient int age;

  @Lob
  private String curriculum;
}
```

O/R Mapping

Access type

- Annotations can be placed on attributes or getters;
- This specifies access type: FIELD or PROPERTY;
- You should use one consistently. If you mix, you have to specify @Access(AccessType.Field) (or .Property).

```
@Entity
public class Employee {
  /* ... */

  @Temporal(TemporalType.DATE)  // FIELD access.
  private Date birthDate;
}
```

```
@Entity
public class Employee {
  /* ... */

  @Temporal(TemporalType.DATE)  // PROPERTY access.
  public Date getBirthDate() { return birthDate; }
}
```

Column/table attributes

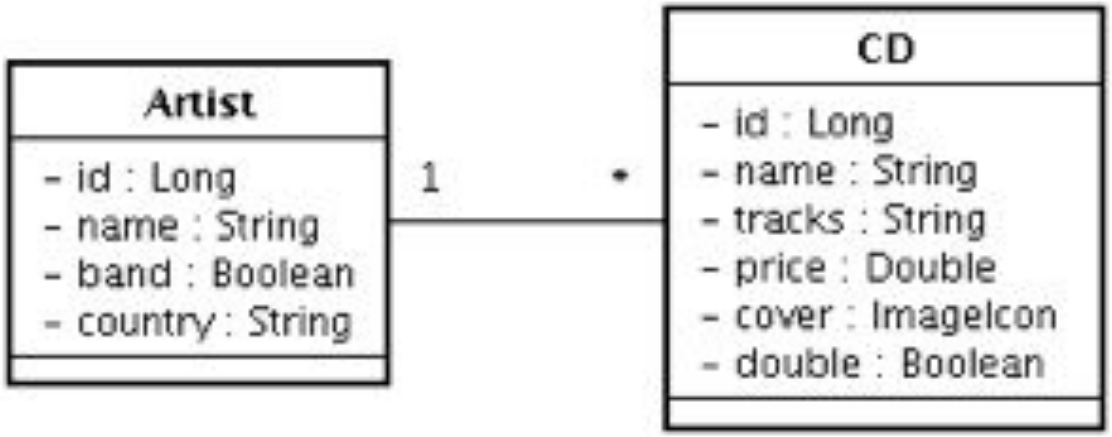
- We can specify characteristics of the column:

```
@Column(length = 50, nullable = false)
private String name;

// Using non-default values:
@Column(name = "ARTIST_COUNTRY")
private String country;

// updatable = false: removes the property from UPDATEs.
// E.g.: the creation date of an object.
```

- Also for the table:

```
@Entity
@Table(name = "ARTISTS", schema = "JAVADISCS")
public class Artist { }
```

Association mapping




```
public class Artist {
  /* ... */

  private Set<CD> cds;

  public Set<CD> getCds() { return cds; }

  private void setCds(Set<CD> cds) {
     this.cds = cds;
  }
}
```

O/R Mapping

Association mapping

- Can be very complex, we will focus on simple cases;
- JPA doesn't manage associations for you;
  - Create your own convenience/domain methods, also guaranteeing the cardinality of the association.

```
public class Artist {
  /* ... */

  public void addCd(CD cd) {
    if (cd == null) throw new
                IllegalArgumentException("null CD");
    if (cd.getArtist() != null)
                cd.getArtist().getCds().remove(cd);
    cd.setArtist(this);
    cds.add(cd);
  }
}
```

O/R Mapping

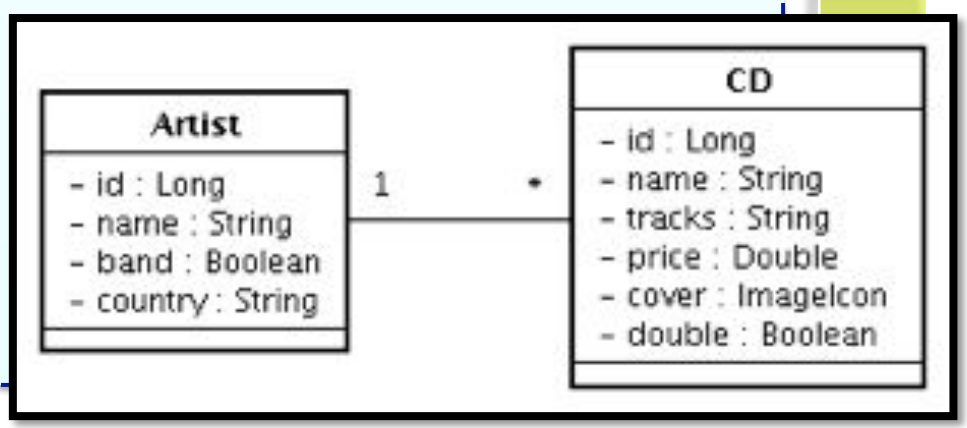
Association mapping

- There are four kinds:
  - @OneToOne;
  - @OneToMany;
  - @ManyToOne;
  - @ManyToMany.
- Most commonly used properties:
  - cascade = CascadeType.___;
  - mappedBy = "___";
  - fetch = FetchType.___;
  - @JoinColumn(nullable="true|false").

Association mapping – examples

- `cascade` configures transitive persistence;
- `mappedBy` characterizes the association as bi-directional (they are unidirectional by default).

```
public class CD {
  @ManyToOne
  private Artist artist;

  // ...
}

public class Artist {
  @OneToMany(
     cascade = CascadeType.ALL,
     mappedBy="artist"
  )
  public Set<CD> cds;

  // ...
}
```



O/R Mapping

What is transitive persistence?

- JPA applies persistence by transitivity:
  - If an object X is persistent and an object Y is associated to it, Y must become persistent also.
- Configurable by the parameter cascade="O", where O is one of: ALL, DETACH, MERGE, PERSIST, REFRESH, or REMOVE;
- If operation O happens to X, execute operation O in Y.

Association mapping – examples

- `fetch` indicates when to load the associated entity:
  - EAGER: must load it immediately;
  - LAZY: may load it later, when it is really needed;
  - Solves the N+1 SELECTs problem;
- `@JoinColumn` specifies the name of the foreign key column (if you don't want the default).

```
@Entity
public class Employee {
  @OneToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "ADDRESS_ID")
  private Address address;
  // ...
}

// Address uses mappedBy to indicate Employee as the "owner",
// i.e., the table which owns the foreign key.
```

The infamous lazy initialization exception

- Imagine this scenario:
  1. Create an entity manager (EM);
  2. Retrieve an object with a lazy association;
  3. Detach the object or close the EM;
  4. Try to access the lazy association.
- What happens?

Simple properties can also be fetched lazily (very useful for LOBs), specifying so in @Basic (not so useless after all!). But it only works with PROPERTY access type.

O/R Mapping

Association mapping – examples

- `@JoinTable` specifies the table that is created to associate two other tables in an N-to-N relation.

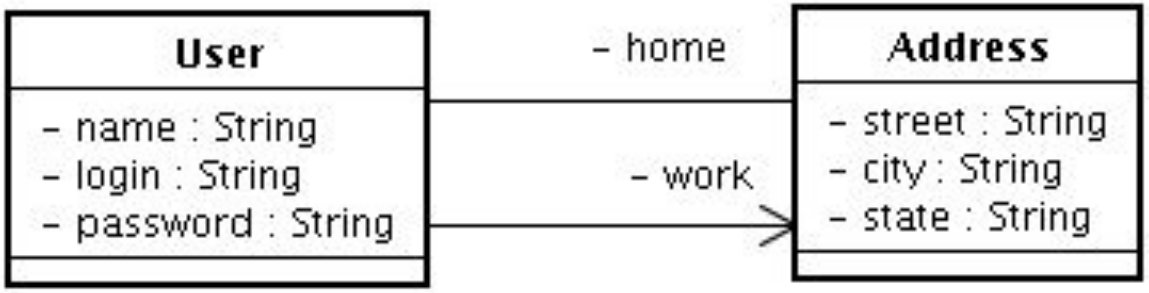
```
@Entity
public class Employee {
  @ManyToMany
  @JoinTable(
     name="EMP_PROJ”,
     joinColumns = {
          @JoinColumn(name="EMP_ID", referencedColumnName="ID")},
     inverseJoinColumns={
          @JoinColumn(name="PROJ_ID”,referencedColumnName="ID")})
   private List<Project> projects;

  // ...

}

// It’s starting to get really complicated. Better stop... :)
```

Employee examples taken from: http://en.wikibooks.org/wiki/Java_Persistence

O/R Mapping

Embeddable classes

- JPA allows us to have more classes than database tables (granularity specification);
- A class that doesn't have a table is embeddable;
  - It exists only associated to an entity (value types);
  - Its properties are stored in the associated entity's table, it doesn't have an id and follows the life cycle of its owner.



Embeddable classes

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

  // ...
}

@Entity
public class User {
  @Embedded
  private Address home;

  @Embedded
  private Address work;

  // ...
}
```

O/R Mapping

Embeddable classes

- Are like entities, but have no ID;
- Can (actually, must!) be used in JPQL queries;
- Many entities can use the same embeddable class;
- They can be associated to other embeddable classes as well as other entities;
- Limitations:
  - Two entity instances cannot share the same embeddable instance;
  - JPA cannot tell between a null value type and an empty value type (not null, but all properties null).

Element collections

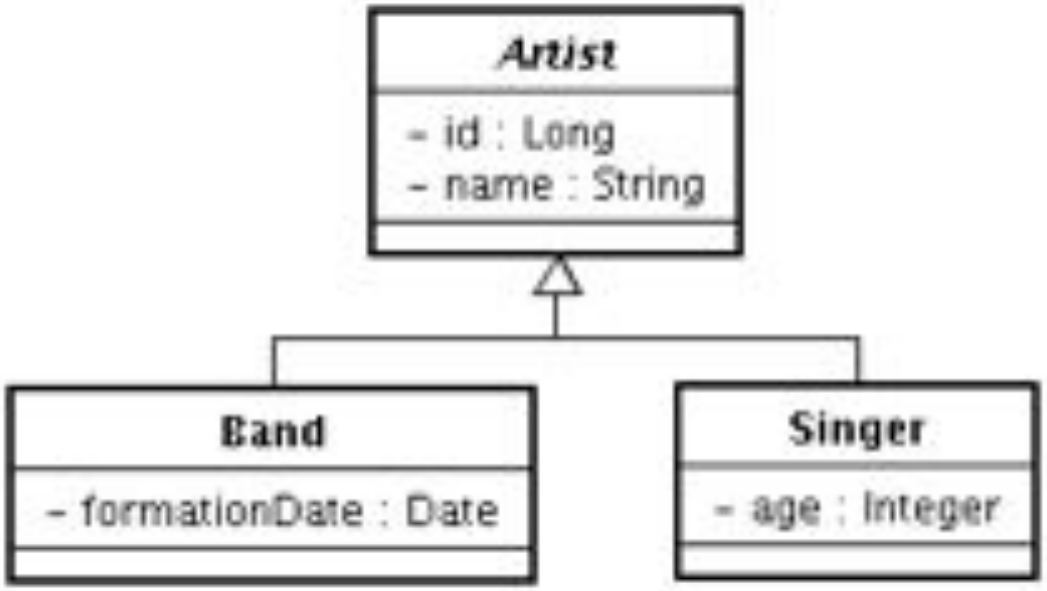
- Collection of entities are mapped via associations;
- Collection of value objects (e.g. String) have to be mapped as element collections:

```
public class Artist {
  @ElementCollection @Column(length = 20)
  public Set<String> tags;

  // ...
}
```

O/R Mapping

Inheritance mapping

- Three possibilities (Scott Ambler):
  - A table for each class of the hierarchy;
  - A table for each concrete class of the hierarchy;
  - A single table for all the hierarchy.
- Cannot be combined in the same hierarchy.



A table for each concrete class

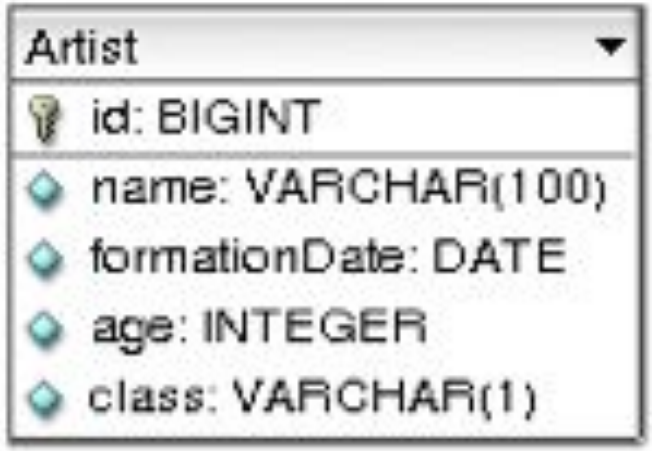
- Bad support for polymorphism (e.g.: associations in the superclass);
- Inefficient polymorphic queries (several SELECTs);
- Efficient concrete class queries;
- Column duplication is bad for maintenance;
- Use when polymorphism is not a requirement.

A table for each concrete class

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Artist { }

@Entity
public class Band extends Artist { }

@Entity
public class Singer extends Artist { }
```

Single table

- Efficient polymorphic queries;
- No redundant columns;
- Columns that belong only to subclasses must be nullable (integrity constraint problem);
- Waste of space;
- Seems to be the best cost x benefit.



O/R Mapping

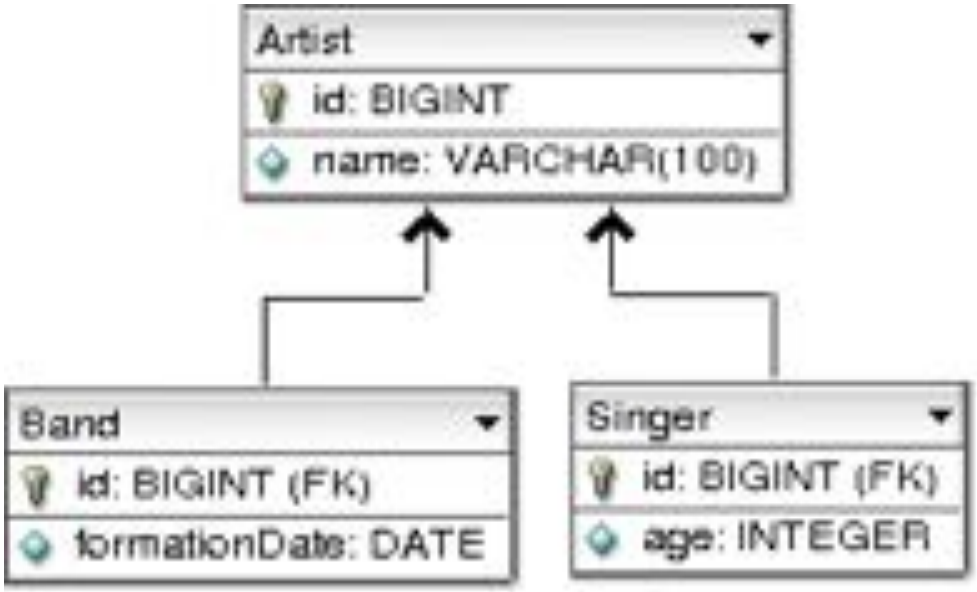
Single table

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "class",
    discriminatorType = DiscriminatorType.CHAR)
@DiscriminatorValue("A")
public class Artist { }

@Entity
@DiscriminatorValue("B")
public class Band extends Artist { }

@Entity
@DiscriminatorValue("S")
public class Singer extends Artist { }

// Discriminator type can be CHAR, INTEGER, STRING.
```

O/R Mapping

A table for each class

- No problems with integrity constrains or ambiguity;
- Bad performance because of JOIN operations;
- Recommended when integrity is a strong requirement.



A table for each class

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Artist { }

@Entity
public class Band extends Artist { }

@Entity
public class Singer extends Artist { }
```

Mapped superclasses

- Another kind of inheritance mapping, introduced by ORM frameworks;
- Mapped superclasses are not entities;
- Their purpose is to declare persistent properties that can be inherited by entities:

```
@MappedSuperclass
public class Artist {
  @Id @GeneratedValue
  private Long id;
  private String name;
}

// Band inherits the id and the name as persistent properties.
@Entity
public class Band extends Artist { }
```

Mapped superclasses

- A glimpse of nemo-utils…

```
@MappedSuperclass
public abstract class DomainObjectSupport implements
DomainObject {
  @Basic @Column(nullable = false, length = 40)
  protected String uuid;

  // equals() and hashCode() based on UUID.
}

@MappedSuperclass
public abstract class PersistentObjectSupport extends
DomainObjectSupport implements PersistentObject {
  @Id @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;

  @Version @Column(nullable = false)
  private Long version;

  // ...
}
```

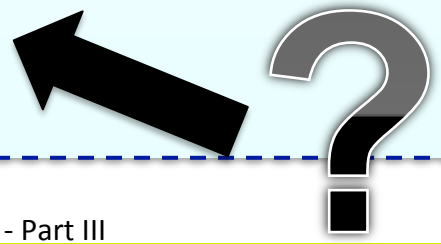
O/R Mapping

O/R Mapping

ORM operations

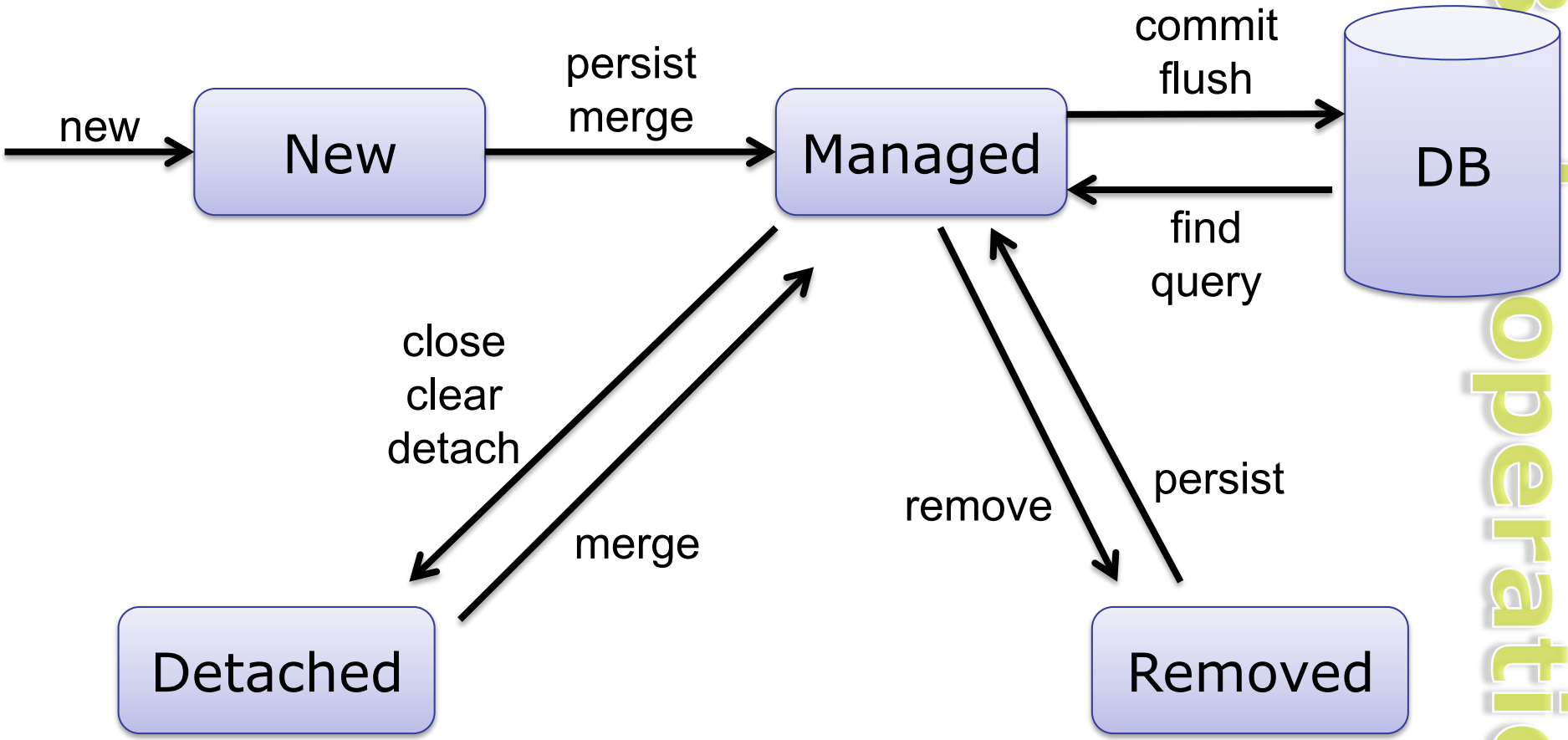
- From SQL: insert, update, select and delete;
- In ORM: persist, merge, find/query and remove;
- Persisting, merging, finding and removing are basic operations:

```
// Transaction omitted in code below. Assume it is done.

// Equivalent to SQL INSERT:
Car carA = new Car(); // Set some data.
em.persist(carA);

// Equivalent to SELECT with PK, then SQL DELETE:
Car carB = em.find(Car.class, 1);
em.remove(carB);

// Equivalent to SQL UPDATE:
Car carC = getCarFromAnotherManager();
em.merge(carC);
```



Basic operations

JPA Entity life-cycle



Managed objects are always kept in sync with the DB (upon commits/flushes) through a feature called "dirty checking".

Basic operations

Detach and merge

```
EntityManager em = JPAUtil.getEntityManager();
em.getTransaction().begin();

Model detachedPorsche = em.find(Model.class, 1);
em.detach(detachedPorsche);
detachedPorsche.setDescription("Porsche 911 Turbo");
detachedPorsche.setPower(500);

Model managedPorsche = em.find(Model.class, 1);
managedPorsche.setDescription("Porsche 911 T.");

Model modifiedPorsche = em.merge(detachedPorsche);

// True and true.
System.out.println(detachedPorsche != modifiedPorsche);
System.out.println(managedPorsche == modifiedPorsche);

// Flush to the database.
em.getTransaction().commit();
```

There can be only one managed object from the same entity with the same id (in the same EM).

Basic operations

Java Persistence Query Language

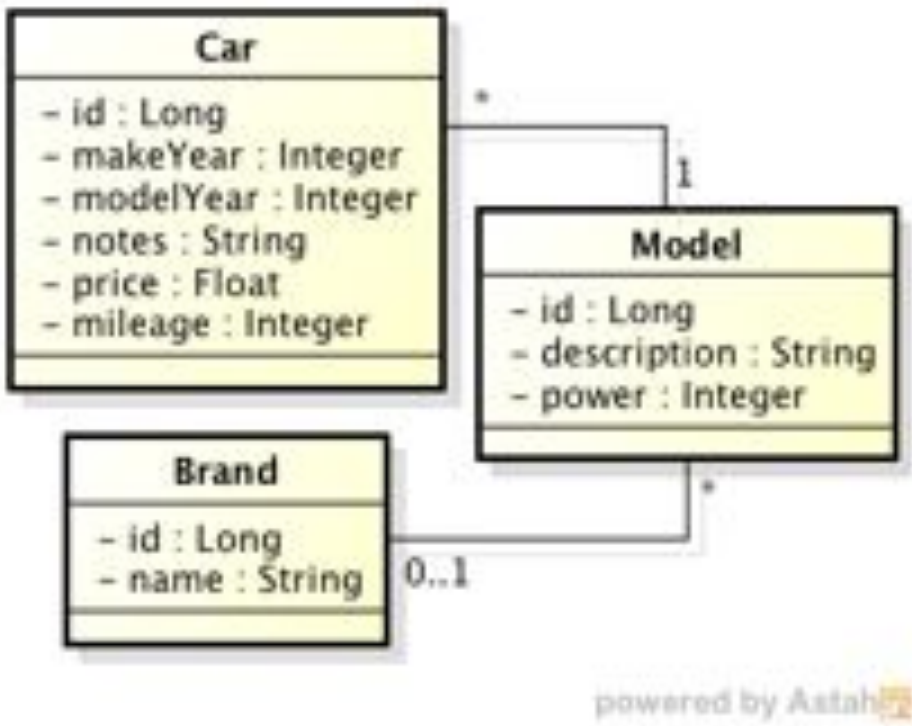
- ORM frameworks support SQL;
- For full ORM, though, we need an OO query language;
- JPA offers JPQL, very similar to Hibernate's HQL;
- A simple select query with JPQL:

```
EntityManager em = JPAUtil.getEntityManager();

TypedQuery<Car> q = em.createQuery("select c from Car c",
    Car.class);

List<Car> cars = q.getResultList();
for(Car c : cars) {
    System.out.println(c.getNotes());
}
```

JPQL

An object-oriented query language

- Meaning you should refer to your entities, not your tables, in the queries;
- Otherwise it's very similar to SQL:



JPQL

```
// What in SQL could be:
select * from TABLE_CARS where MODEL_YEAR >= 2010

// In JPQL becomes:
select c from Car c where c.modelyear >= 2010
```

Use your SQL knowledge

- Lots of SQL operators are available in JPQL:
  - =, >, >=, <, <=, <>;
  - NOT, BETWEEN, LIKE, IN, IS NULL, IS EMPTY;
  - MEMBER [OF], EXISTS.
- Aggregation functions:
  - Average: AVG(property);
  - Max/min: MAX(property), MIN(property);
  - Sum: SUM(property);
  - Count: COUNT(property);

```
String jpql = "select AVG(c.modelYear) from Car c";
Query query = em.createQuery(jpql, Double.class);
Double average = query.getSingleResult();
```

JPQL functions

- String functions:
  - Concatenation: CONCAT(String, String, …);
  - Substring: SUBSTRING(String, int start [, int length]);
  - Trimming: TRIM() (with many variations);
  - Case manipulation: LOWER(String), UPPER(String);
  - String length: LENGTH(String);
  - IndexOf: LOCATE(String str, String substr [, int start]).
- Date constants:
  - CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP.

JPQL functions

- Numeric functions:
  - Absolute value: ABS(int);
  - Square root: SQRT(int);
  - Remainder: MOD(int, int);
- Collection functions:
  - Size: SIZE(collection);
  - Element index: INDEX(object):

JPQL

```
// Assuming Competition.racerRanking is a list (indexed collection),
// retrieve top 5 racers in a competition:
select racer from Competition c
  join c.racerRanking racer
  where INDEX(racer) < 5
```

New functions in JPA 2.0

- Case expressions:

```
update Employee e set e.salary = case e.position
  when 'Director' then e.salary * 1.15
  when 'Manager' then e.salary * 1.10
  else e.salary * 1.05
end
```

- NULLIF:

```
select nullif(e.salary, -1) from Employee e
```

- COALESCE:

```
select coalesce(e.name, e.username) from Employee e
```

New functions in JPA 2.0

- INDEX:

```
// Assuming a.drivers is a list instead of a set.
select d from Ambulance a join a.drivers d
where a.id = :id and index(d) between 0 and 4
```

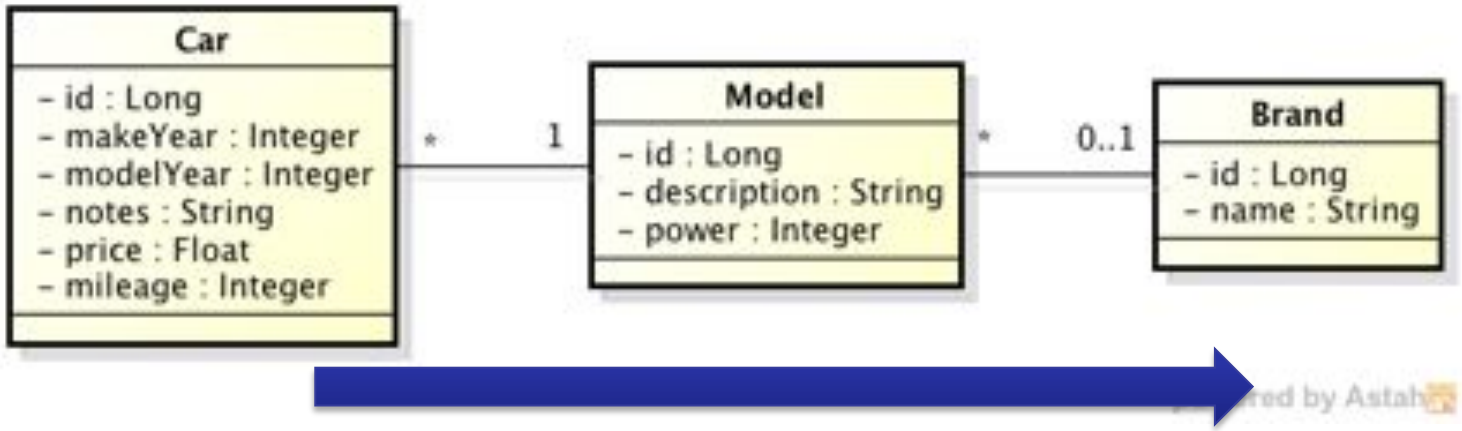
- TYPE:

```
// Assuming hierarchy of Employee instead of enum.
select e from Employee e
where type(e) in (Operator, Dispatcher)
```

- KEY, VALUE, ENTRY:

```
// Assuming a.drivers is a map instead of a set.
select key(d), value(d) from Ambulance a join a.drivers d
where a.id = :id
```

Simplifying the joins in many-to-one

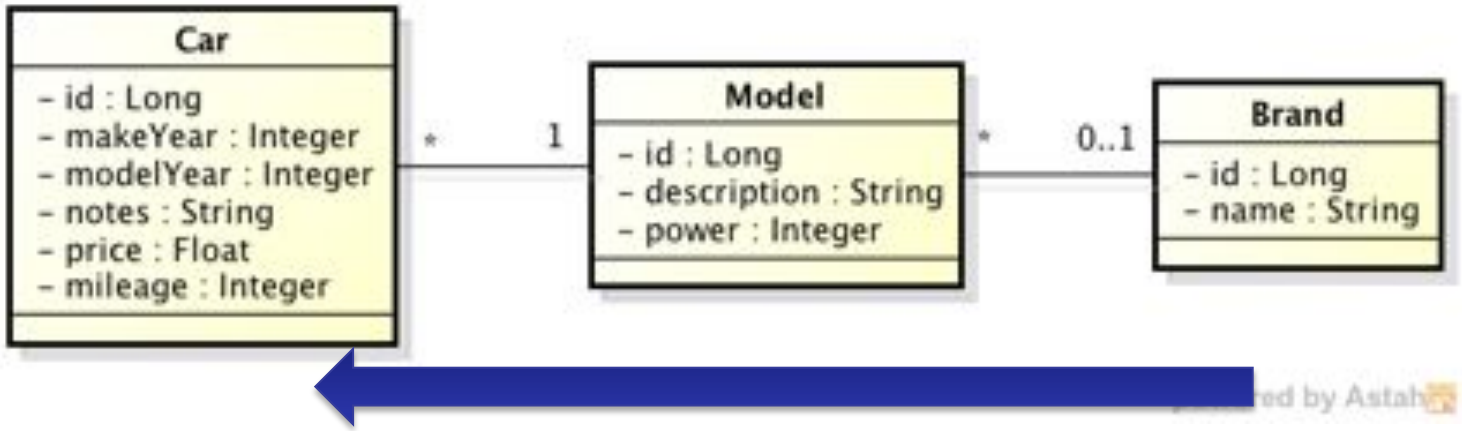



```
// In SQL:
select * from Car c
    left outer join Model m on c.model_id = m.id
    left outer join Brand b on model.brand_id = brand.id
where brand.name = 'Ferrari'

// In JPQL:
select c from Car c where c.model.brand.name = 'Ferrari'
```

No joins needed in JPQL when the path is many-to-one and navigable.

JPQL

Navigating one-to-many




```
// In SQL:
select * from Brand b
  left outer join Model m on b.id = m.brand_id
  where m.description like '%911%'

// In JPQL:
select b from Brand b
    join brand.models m
    where m.description like '%911%'
```

JPQL

Specifying parameters

- Back to code, imagine a DAO method that returns all cars from a given brand;
- How to specify the parameter?

```
@Stateless
public class CarJPADAO implements CarDAO {
   @PersistenceContext
   private EntityManager em;

   @Override
   public List<Car> retrieveByBrandName(String name) {
      String jpql = "select c from Car c where
                           c.model.brand.name = :brandName";
      TypedQuery<Car> query = em.createQuery(jpql, Car.class);
      query.setParameter("brandName", name);
      return query.getResultList();
   }

   // ...
}
```

JPQL

Specifying entities as parameters

- Why retrieve the cars by brand name, when we can use the entity (the brand) directly in the quer?

```
@Stateless
public class CarJPADAO implements CarDAO {
   @PersistenceContext
   private EntityManager em;

   @Override
   public List<Car> retrieveByBrand(Brand brand) {
      String jpql = "select c from Car c where
                           c.model.brand = :brand";
      TypedQuery<Car> query = em.createQuery(jpql, Car.class);
      query.setParameter("brand", brand);
      return query.getResultList();
   }

   // ...
}
```

JPQL

Sub-queries

- Also possible in JPQL. Note the use of different variables for Car in the query and the sub-query:

```
// All cars that are older than the average make year.
select c from Car c
  where c.makeYear >
  (select AVG(car.makeYear) from Car car)

// All brands that have at least one car costing more than
// a million.
select b from Brand b where EXISTS (
  select c from Car c where
     c.model.brand = b and c.price >= 1000000
)
```

JPQL

Group-by and having

- Yet another SQL feature brought to the OO world:

```
// Brands and number of cars, filtering those which have more
// than 10 cars.
select c.brand, COUNT(c) from Car c
   group by c.brand
   having COUNT(c) > 10
```

- But this is a complex result! No problem:

```
String jpql = "...";   // Use string from above.

Query query = em.createQuery(jpql); // Un-typed.
List result = query.getResultList();

for(Object obj : result){
   Object[] row = (Object[])obj;
   Brand brand = (Brand) row[0];    // Casting needed.
   int count = (int) row[1];
   // ...
}
```

JPQL

Select new

- No problem? Casting from object is a problem! But we can solve it with “select new”:

```
package pkg;

public class BrandCount {
   private Brand brand;
   private int count;

   public BrandCount(Brand brand, int count) { /* ... */ }
   // ...

}
```

```
String jpql = "select new pkg.BrandCount(c.brand, COUNT(c))
     from Car c group by c.brand having COUNT(c) > 10”

TypedQuery<BrandCount> query = em.createQuery(jpql,
     BrandCount.class);
List<BrandCount> result = query.getResultList();
// ...
```

JPQL

Named queries

- Let's improve our DAO even further:

```
@NamedQuery(name = "Car.retrieveByBrand", query = "select c
   from Car c where c.model.brand = :brand")
@Entity
public class Car { /* ... */ }
```

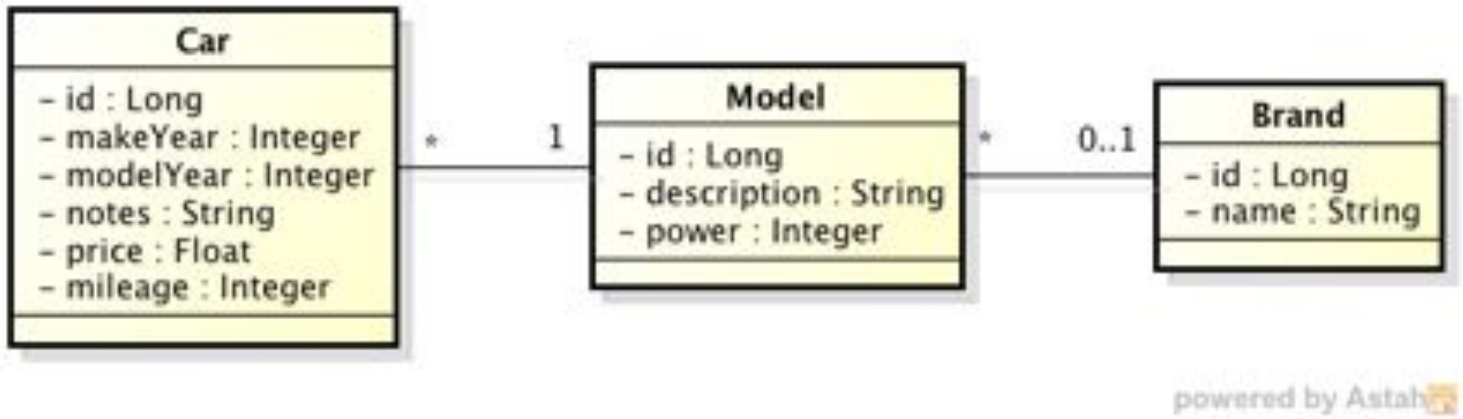
```
@Stateless
public class CarJPADAO implements CarDAO {
   @PersistenceContext
   private EntityManager em;

   @Override
   public List<Car> retrieveByBrand(Brand brand) {
      // We could replace the string below with a constant.
      TypedQuery<Car> query = em.createNamedQuery(
         "Car.retrieveByBrand", Car.class);
      query.setParameter("brand", brand);
      return query.getResultList();
   }
   // ...
}
```

JPQL

Join fetch

- Specifying associations as lazy/eager doesn't solve the N+1 SELECTs problem alone;
- Sometimes we want a given association to be lazy, sometimes we would like it to be eager;
- We then set it as lazy and use join fetch when needed:

```
select b from Brand b join fetch b.models
```



JPQL

Criteria API

- Introduced in JPA 2.0;
- Before there was JPQL only;
- Similar to Hibernate Criteria API (like JPQL is similar to HQL);
- Allows programmatic construction of queries:
  - Uses objects instead of Strings;
  - Thus, can be verified at compile time.
- Two modes: static and dynamic.

Criteria API

Dynamic mode

```
public Employee retrieveByUsername(String username) {
   CriteriaBuilder cb = em.getCriteriaBuilder();
   CriteriaQuery<Employee> cq = cb.createQuery(Employee.class);
   Root<Employee> root = cq.from(Employee.class);

   EntityType<Employee> model = root.getModel();
   cq.where(cb.equal(root.get(model.getSingularAttribute(
      "username", String.class)), username));

   Employee employee = null;
   try {
      employee = em.createQuery(cq).getSingleResult();
   } catch (RuntimeException e) {
      /* Do something... */
      return null;
   }

   return employee;
}
```

Criteria API

Static mode

```
public Employee retrieveByUsername(String username) {
   /* Same stuff before... */

   cq.where(cb.equal(root.get(EmployeeJPAMetamodel.username),
      username));

   /* Same stuff after... */
}
```

```
@StaticMetamodel(Employee.class)
public class EmployeeJPAMetamodel {
 public static volatile
      SingularAttribute<Employee, String> name;
 public static volatile
      SingularAttribute<Employee, String> username;
 public static volatile
      SingularAttribute<Employee, String> password;
 public static volatile
      CollectionAttribute<Employee, EmployeeType> type;
}
```

Criteria API

Dynamic vs. static

- Dynamic model uses String (prone to error);
- Static model requires an extra class (meta-model);
  - Code generators could could help here…

```
EntityType<Employee> model = root.getModel();
cq.where(cb.equal(root.get(model.getSingularAttribute(
  "username", String.class)), username));
```

```
cq.where(cb.equal(root.get(EmployeeJPAMetamodel.username),
  username));
```

Check the source code of https://github.com/feees/Sigme for more examples of Criteria API use.

Criteria API

Criteria API

Bean Validation

- Transversal validation: from the form in the Web page to the persistence database;
- Centered in the domain layer, but without loosing focus of its purpose – annotations;
- Based on Hibernate Validator.

```
public class Ambulance extends PersistentObjectImpl {
  @NotNull
  private int number;

  @NotNull
  @Size(min = 8, max = 8)
  private String licensePlate;

  /* ... */
}
```

Bean Validation

Some of the available annotations

- @AssertFalse, @AssertTrue (for boolean);
- @DecimalMax, @DecimalMin (for real numbers, but works only with BigDecimal);
- @Max, @Min (for integer numbers);
- @Digits (only digits, string OK, can specify min/max digits of integer and decimal parts);
- @Future, @Past (for dates);
- @Pattern (regular expressions);
- @Valid: added to association properties, traverses the object graph, validating all attributes.

Custom validator

- When the existing annotations don't cut it:

```
public class PlateValidator implements
                        ConstraintValidator<Plate, String> {

  public void initialize(Plate constraintAnnotation) { }

  // License plate valid only if numbers in ascending order:
  public boolean isValid(String value,
                        ConstraintValidatorContext context) {
    if (value.length() != 8) return false;
    boolean ascending = true;
    int previous = Character.digit(value.charAt(4), 10);
    for (int i = 5; ascending && i < 8; i++) {
      int current = Character.digit(value.charAt(i), 10);
      ascending = current > previous;
      previous = current;
    }
    return ascending;
  }
}
```

Bean Validation

Custom validator

```
@NotNull
@Pattern(regexp = "^[A-Z]{3} [0-9]{4}$")
@Constraint(validatedBy = PlateValidator.class)
@Documented
@Target({ANNOTATION_TYPE, METHOD, FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Plate {
  String message() default "Invalid license plate";
  String[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}
```

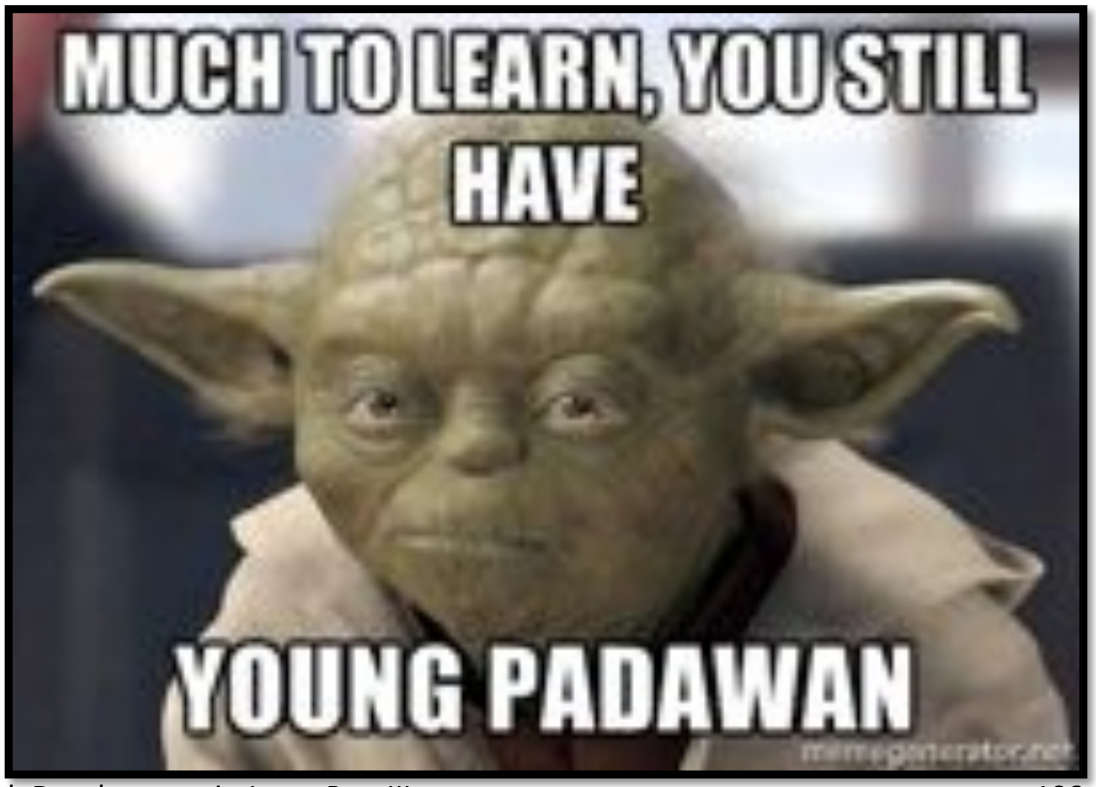
```
public class Ambulance extends PersistentObjectImpl {
  @Plate
  private String licensePlate;

  // ...
}
```

Bean Validation

Bean Validation

There's always more

- Stuff for you to learn on your own time:
  - Go deeper into many of these topics (data sources, connection pools, O/R mapping, bean validation, …);
  - Second-level cache;
  - Pagination of results;
  - Etc…



Web Development in Java – Part III

ENTERPRISE JAVA BEANS

Last, but definitely not least

- EJB is at the core of Java EE;
- Benefits:
  - Simplified development of large scale applications;
  - System level services provided (transactions, logging, load balancing, persistence, exception handling);
  - Life-cycle managed by container;
- Types:
  - Session bean (stateful or stateless);
  - Message-driven bean (JMS);
  - Entity bean (JPA).

Stateless (session) beans

- Does not hold client state;
- Thus, can be provided as a pool of instances;
- To create one:
  - Create an interface exposing the business methods;
  - Tag it as @Local if the client is also deployed in the application server, @Remote otherwise;
  - Create a class implementing the interface, annotate it with @Stateless;
  - Provide the interface to the client, if remote.

We have already seen @Local stateless session beans in Java Hostel and CDI Travel.

Example: remote session bean

```
@Remote
public interface LibrarySessionBeanRemote {
   void addBook(String bookName);
   List getBooks();
}
```

```
@Stateless
public class LibrarySessionBean implements LibrarySessionBeanRemote {
   /* Data connection... */

   public void addBook(String bookName) {
      /* ... */
   }

   public List<String> getBooks() {
      /* ... */
   }
}
```

Example taken from: http://www.tutorialspoint.com/ejb/ejb_pdf_version.htm

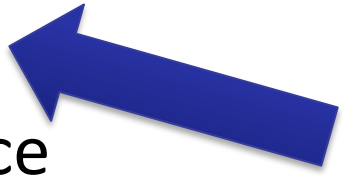
Example: remote session bean

```
# jndi.properties
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=localhost
```

```
// Main code (exceptions omitted).
Properties props = new Properties();
Props.load(new FileInputStream("jndi.properties");
InitialContext ctx = new InitialContext(props);

LibrarySessionBeanRemote bean =
(LibrarySessionBeanRemote)ctx.lookup("LibrarySessionBean/remote");
bean.addBook("A Game of Thrones");
bean.addBook("A Clash of Kings");
bean.addBook("A Storm of Swords");

// ...

List<String> books = bean.getBooks();

// ...
```

Stateful (session) beans

- Preserves conversational state with clients (attributes);
- A new bean is created for each client, destroyed when scope is over;
- To create one:
  - Create an interface exposing the business methods;
  - Tag it as @Local if the client is also deployed in the application server, @Remote otherwise;
  - Create a class implementing the interface, annotate it with @Stateful;
  - Provide the interface to the client, if remote.



The only difference w.r.t. stateless beans

Message-driven beans

- Similar to stateless EJBs, but:
  - Are not accessed directly by clients;
  - Monitor JMS queues for messages and responds.
- JMS (Java Message Service) is a message-oriented middleware for sending messages between peers;
- Need to be configured in the application server, then @Resource annotations allow us to access queues, etc.

An interesting example combining CDI events with JMS queues for asynchronous processing can be seen here:

https://weblogs.java.net/blog/jjviana/archive/2010/04/14/decoupling-event-producers-and-event-consumers-java-ee-6-using-cdi-a

New in Java EE 6

- No-interface EJBs (is it a good idea?):

```
@Stateless @LocalBean @Named
public class SomeStatelessBean {
   public void aMethod() { /* ... */ }
   public String anotherMethod() { /* ... */ }

   @PostConstruct
   public void init() {
      /* Initialization code... */
   }
}
```

Callback method. There are more…

- Singleton EJBs:

```
@Stateless
@Singleton
@Named
public class HighlanderBean {
   /* There can be only one... */
}
```

Singleton EJBs are thread-safe, serializing method calls.

Callback methods

- All beans:
  - @PostConstruct: after the bean is created;
  - @PreDestroy: before it is destroyed;
- Stateful beans:
  - @PrePassivate: before the bean is passivated;
  - @PostActivate: after the bean is activated;
- Entity beans (JPA):
  - @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate, @PostLoad.

Transaction demarcation

- @Transactional (CDI) or @TransactionAttribute (EJB):
  - MANDATORY: if no transaction context, exception;
  - NEVER: if there is a transaction context, exception;
  - NOT_SUPPORTED: transaction context unspecified;
  - REQUIRED: if there is a transaction context, use it. Otherwise, create it;
  - REQUIRES_NEW: create a new transaction context;
  - SUPPORTS: if there is a transaction context, use it.
- Annotating the class applies it to all methods;
- Annotating a method overrides the class annotation.

Asynchronous calls

- Execute long methods in background:

```
public class RegisterCallServiceBean extends RegisterCallService {
  @Asynchronous
  public Future<List<Call>> searchForSimilar(Call call) {
     List<Call> xList = callDAO.searchByX(call.getX());
     List<Call> yList = callDAO.searchByY(call.getY());
     // ...

     List<Call> similars = new ArrayList<Call>();
     similars.addAll(xList);
     similars.addAll(yList);
     // ...

    similars.remove(call);

    return new AsyncResult<List<Call>>(similars);
  }
}
```

Asynchronous calls

- Check if done:

```
public class RegisterCallAction ... {
   private Future<List<Call>> result;

   public List<Call> getSimilars() {
      if ((result != null) && (result.isDone())) return result.get();
      return null;
   }

   public boolean isDone() {
      return ((result != null) && (result.isDone()));
   }

   public void searchForSimilar() {
      result = registerCallService.searchForSimilar(call);
   }
}
```

getSimilars() and isDone() should be called using AJAX.

Auth^2 with JAAS

- Authentication = guaranteeing the user is who she says she is;
- Authorization = guaranteeing the user can access resources she is authorized to;
- For Java applications, we can use JAAS: Java Authentication and Authorization Services;
  - Data integrity;
  - Confidentiality;
  - Non-repudiation;
  - Auditing.

Basic JAAS concepts

- **Security realm**: set of security configurations registered under a name;
- **User**: an individual or software identified by an username and a password (credentials);
- **Group**: group of users;
- **Role**: a name associated with a set of access rights. Can be associated to users or groups.

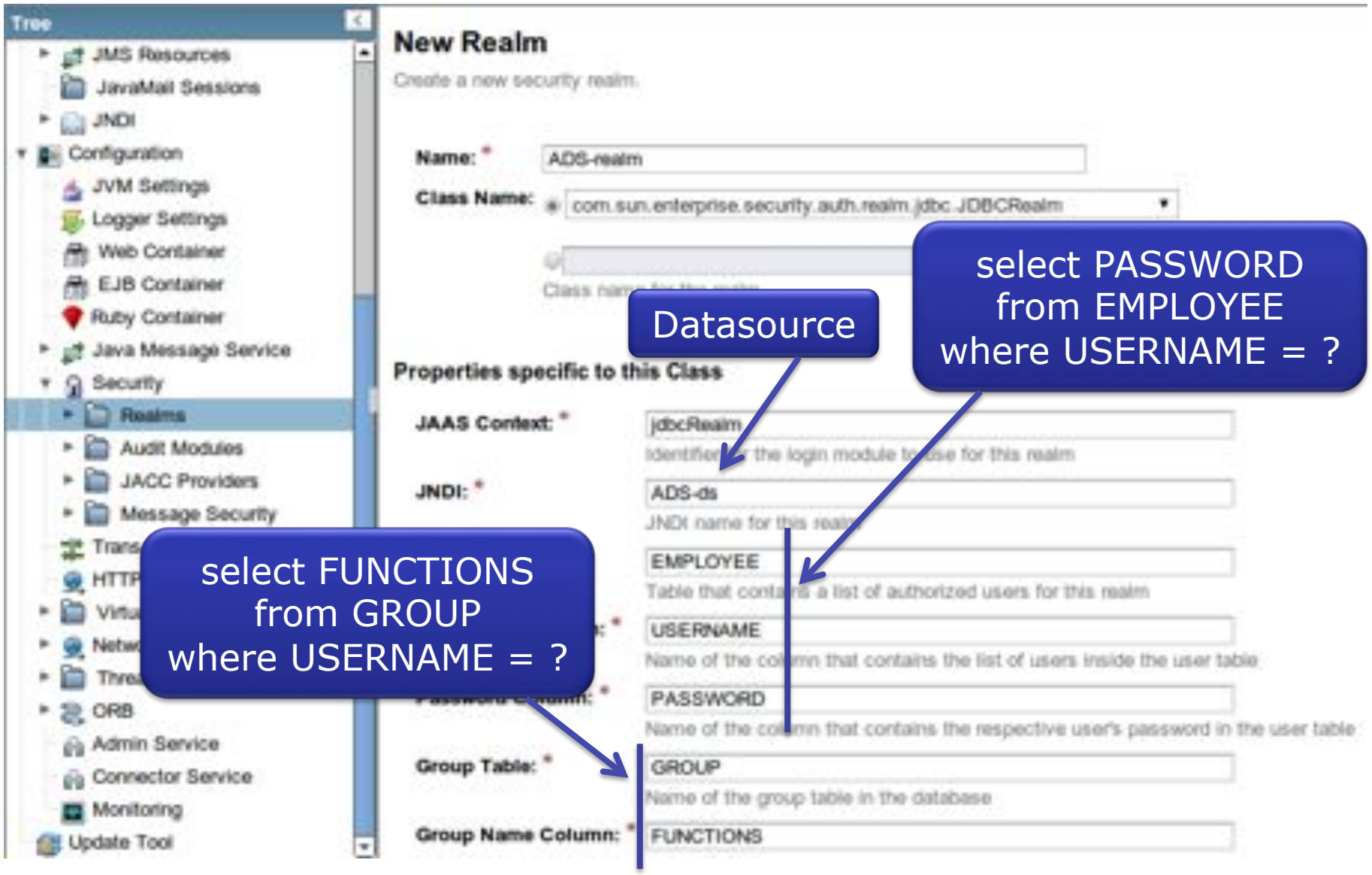
Authentication = what users exist and what are their passwords?

Authorization = which roles can access what?

Realms are container-dependent

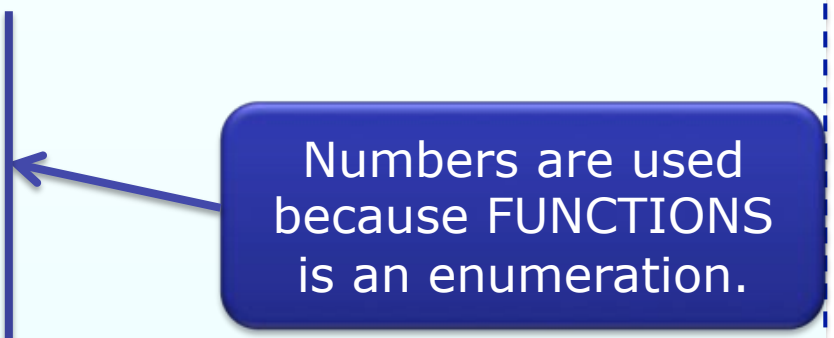
- In GlassFish 3:
  - Flat files;
  - JDBC;
  - Certificate;
  - Solaris;
  - LDAP / Microsoft Active Directory;
  - Any class implementing the Realm interface (proprietary).

Realm setup in GlassFish 3

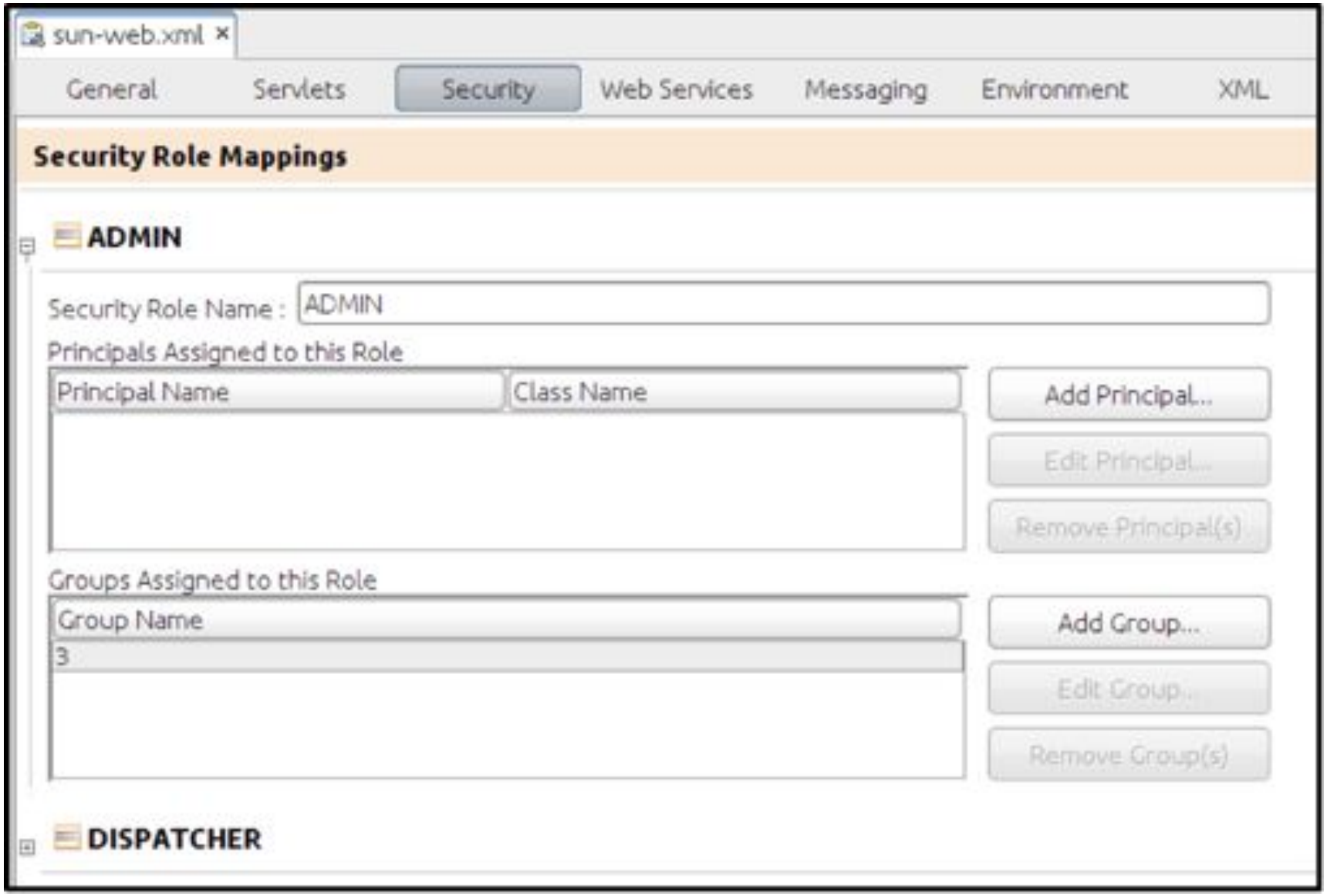


GlassFish configuration (sun-web.xml)

```
<sun-web-app error-url="">
  <context-root>/ADS-war</context-root>
  <security-role-mapping>
    <role-name>OPERATOR</role-name>
    <group-name>0</group-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>DISPATCHER</role-name>
    <group-name>1</group-name>
  </security-role-mapping>

  ...

  <class-loader delegate="true"/>
  <jsp-config>
    <property name="keepgenerated" value="true" />
  </jsp-config>
</sun-web-app>
```

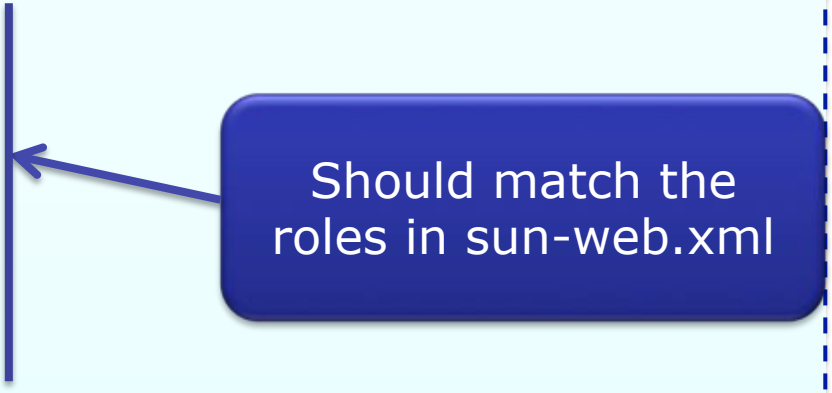


GlassFish configuration (sun-web.xml)

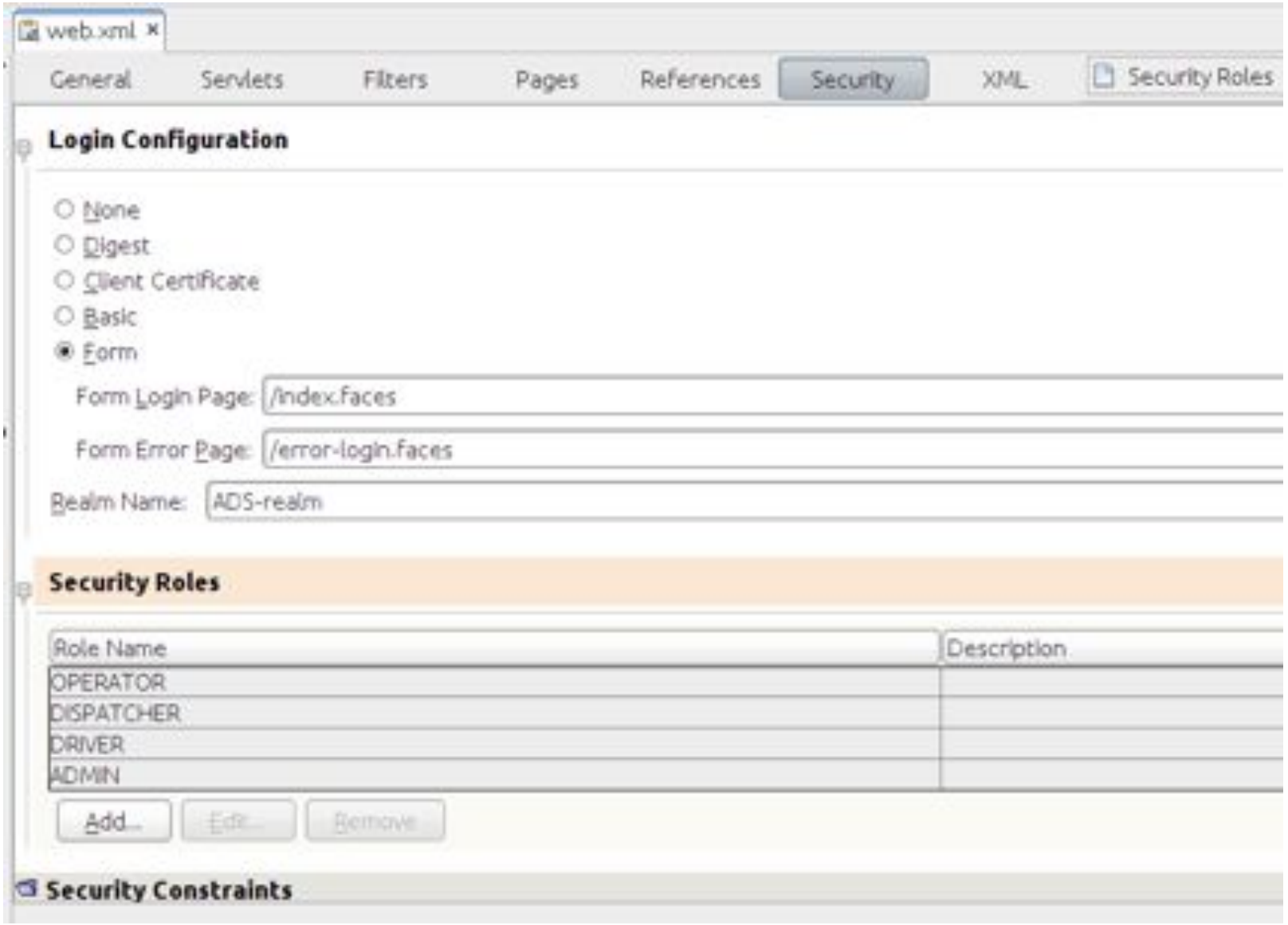


WebApp configuration (web.xml)

```
<web-app ...>
  <login-config>
    <auth-method>FORM</auth-method>
    <realm-name>ADS-realm</realm-name>
    <form-login-config>
      <form-login-page>/index.faces</form-login-page>
      <form-error-page>/error-login.faces</form-error-page>
    </form-login-config>
  </login-config>
  <security-role>
    <description />
    <role-name>OPERATOR</role-name>
  </security-role>
  <security-role>
    <description />
    <role-name>DISPATCHER</role-name>
  </security-role>

  ...

</web-app>
```



WebApp configuration (web.xml)



The login form

```
<ui:decorate template="/templates/form.xhtml">
  <form id="form" method="POST" action="j_security_check">
    <ui:decorate template="/templates/field.xhtml">
      <ui:param name="id" value="form:username" />
      <ui:define name="nome">Username</ui:define>
      <input type="text" id="username" name="j_username" />
    </ui:decorate>
    <ui:decorate template="/templates/field.xhtml">
      <ui:param name="id" value="form:pwd" />
      <ui:define name="nome">Password</ui:define>
      <input type="password" id="pwd" name="j_password" />
    </ui:decorate>
    <ui:decorate template="/templates/buttons.xhtml">
      <input type="submit" value="Log in" />
    </ui:decorate>
  </form>
</ui:decorate>
```

Note HTML field tags, not JSF…

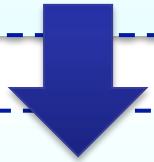
Checking authentication at the server side

```
public class LoginManagerBean implements LoginManager {
  @Resource
  private SessionContext sessionCtx;

  @EJB
  private EmployeeDAO employeeDAO;

  public Employee checkJaasLogin() {
    Employee emp = null;
    Principal principal = sessionCtx.getCallerPrincipal();
    if (principal != null) {
      String username = principal.getName();
      if (! "ANONYMOUS".equals(username)) {
        emp = employeeDAO.retrieveByUsername(username);
      }
    }
    return emp;
  }
}
```

Form vs. programmatic authentication

- Form login:
  - Container is called directly;
  - Our application constantly checks for the principal.
- Programmatic login:
  - Our application is called;
  - The container is programmatically called from our application's code.

```
<form id="form" method="POST" action="j_security_check">
<input type="text" id="username" name="j_username" />
```



```
<h:form id="form">
<h:inputText id="username" value="#{loginBean.username}" />
```

The login() method

```
public class LoginManagerBean implements LoginManager {
  public void login(String username, String password) {
     Employee emp = employeeDAO.retrieveByUsername(username);
     String md5pwd = TextUtils.produceMd5Hash(password);
     String pwd = emp.getPassword();

     if ((pwd != null) && (pwd.equals(md5pwd))) {
     HttpServletRequest request = (HttpServletRequest)FacesContext.
              getCurrentInstance().getExternalContext().getRequest();
     request.login(username, password);

     // request.logout() also available if you want.

     currentUser = emp;
     pwd = password = null;
  }
  else {
     throw new LoginFailedException();
     }
  }
}
```

Once JAAS is done, back to EJBs:

- Use of annotation @RolesAllowed:

```
@RolesAllowed("ADMIN")
public class AmbulanceCrudServiceBean implements AmbulanceCrudService
{
   /* ... */
}
```

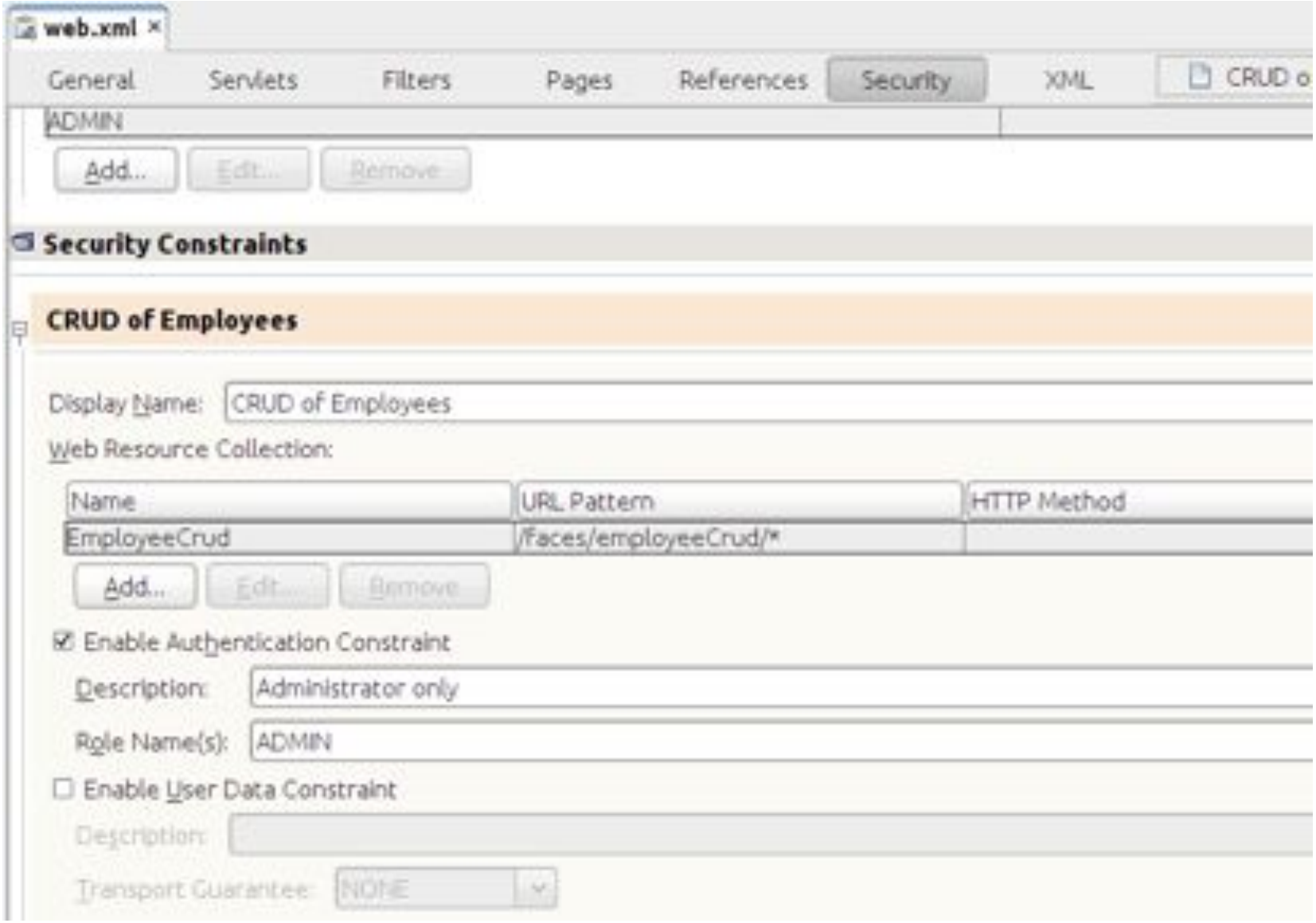
- Applies to the whole class or single methods;
- Limitation: does not extend to inherited methods;
- If a method is called and the user doesn't have the role, javax.ejb.EJBAccessException is thrown;
- Less useful: @PermitAll and @DenyAll.

You can also authorize page access

```
<!-- web.xml -->
<web-app ...>
  <security-constraint>
    <display-name>CRUD of Employees</display-name>
    <web-resource-collection>
      <web-resource-name>EmployeeCrud</web-resource-name>
      <description />
      <url-pattern>/faces/employeeCrud/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <description>Administrator only</description>
      <role-name>ADMIN</role-name>
    </auth-constraint>
  </security-constraint>

  ...

</web-app>
```

HTTP error code 403 in case of violation.

You can also authorize page access

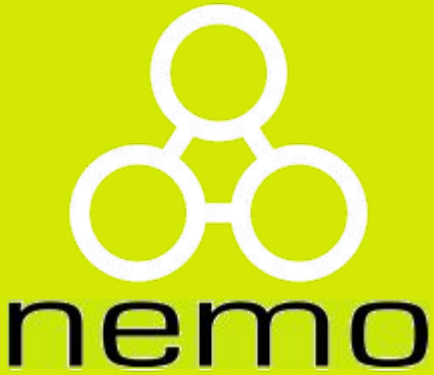


Just an overview…

Why should I use all this?

1. It's thread-safe;
2. You can implement remote access;
3. Transactions are started for you;
4. It's monitored (EJBs visible via JMX in application server);
5. It's pooled – you can control the concurrency, prevent DoS attacks;
6. DI just works, relieving you of a lot of code;
7. It's (mostly) portable and vendor-neutral;
8. There is very little XML;
9. It's easily acessible (via DI) from Restful services, etc.;
10. EJB comes with clustering and security features.

Source: http://www.adam-bien.com/roller/abien/entry/ejb_3_persistence_jpa_for



http://nemo.inf.ufes.br/