Java Persistence API

Table of Contents

[Object Relational Mapping 2](#_Toc100630409)

[Entities 3](#_Toc100630410)

[Primary Keys 5](#_Toc100630411)

[Identifier Generation 5](#_Toc100630412)

[Customization 6](#_Toc100630413)

[Table Names 6](#_Toc100630414)

[Columns 7](#_Toc100630415)

[Relationships 7](#_Toc100630416)

[Managing Entities 10](#_Toc100630417)

[Application-Managed EntityManagers 10](#_Toc100630418)

[Container Managed EntityManager 11](#_Toc100630419)

[Entity Operations 12](#_Toc100630420)

[Persistence Units 12](#_Toc100630421)

The **Java Persistence API** is a specification for managing relational data (i.e. databases) in Java applications. It is available for both Java EE and Java SE.

Java persistence is composed of:

* The Java Persistence API
* The query language
* The Java Persistence Criteria API
* Object/relational mapping metadata

## Object Relational Mapping

**Object Relational Mapping** (ORM) refers to the process of mapping relations, i.e. tables, to objects. We can think of database tables as classes, with each column representing a separate property of the class, and records from the tables as objects. This is understandable since a table defines the properties of the objects that will fill it, but the actual values for the properties come from the records. This is exactly how classes and objects work.

## Entities

An **Entity** is a lightweight persistence domain object which represents a table in a relational database. Basically, it is just a class with some extra bells and whistles to make it obvious to our program that this refers to a database table. Each instance of this special entity class is thus a row of that table. The values we give these objects will be persisted in the database. If we want a particular attribute to not be persisted, we need to mark it with the @Transient annotation.

@Entity *// this identifies the class as a table*public class Customer{  
 @Id *// the next attribute is the primary key* private int customerId;  
 private String name;  
 public int getCustomerId() {  
 return customerId;  
 }  
 public void setCustomerId(int customerId) {  
 this.customerId = customerId;  
 }  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
}

JAVA

The attributes in entities can be of several types:

* **Primitives** like strings, integers, etc.
* **Serializable** types, i.e. things that can be converted to byte streams
* **Enumerated** types like lists, sets, etc.
* **Other entities** or collections of entities
* **Embeddable classes**, i.e. classes which are separate for our convenience when coding, but will be stored in the same table

The comprehensive list can be found [here](https://docs.oracle.com/javaee/7/tutorial/persistence-intro001.htm).

Some rules for entities:

* The entity must be annotated with javax.persistence.Entity.
* There has to be a public or protected no-argument constructor, but it’s okay to make other constructors.
* Entities can extend both other entities and non-entity classes and vice versa.
* Nothing can be declared final, not the class nor the methods and attributes.
* For detached entity instances (i.e. objects of the entity that are no longer under the entity’s control), if the instance is passed through a remote interface, it must implement the *Serializable* interface.
* The attributes must be private, protected or package-private.
* There are no required business or callback interfaces.

## Primary Keys

To identify the primary key, we use the @Id annotation above the required attribute. If we do this with one attribute, we get a **simple primary key**. With multiple attributes, we get a **compound primary key**. We can also use the @EmbeddedId annotation if we want to refer to an embeddable class that will work as the primary key.

## Identifier Generation

In several databases, it is possible to automatically generated IDs. In order to do this using JPA, we can use @GeneratedValue annotation above the relevant identifier.

@Id  
@GeneratedValue(strategy = GenerationType.*AUTO*)  
private int id;

JAVA

Four pre-defined **generation strategies** are available, *AUTO*, *IDENTITY*, *SEQUENCE* and *TABLE*. The generators may be **custom** or **pre-exist** in the database.

The default strategy is *AUTO*, which means the database should decide what generation strategy to use. This is usually *SEQUENCE*.

*IDENTITY* means that each insert operation will cause the value to be incremented. This is very efficient for the database, since it does not require any additional statements.

*SEQUENCE* uses a database sequence to generate unique values. This requires additional SELECT statements, but has no performance impact for most applications.

*TABLE* is rarely used. It simulates a sequence by storing and updating its value in a database table. This requires using pessimistic locks, which involves executing transactions one at a time. This is slow, which is why *SEQUENCE* is preferred.

## Customization

### Table Names

By default, the Entity name refers directly to the table name in the database. However, it is possible to have a different table name. In that case, we must specify the table name.

@Entity(name = "Table Name")  
public class entityName {}  
  
*// alternative format*@Entity  
@Table(name = "Table Name")  
public class entityName {}

JAVA

### Columns

Columns can be customized using the @Column annotation.

@Column(name = "Column Name in Database", nullable = false, length = 100)  
private String columnNameInEntity;

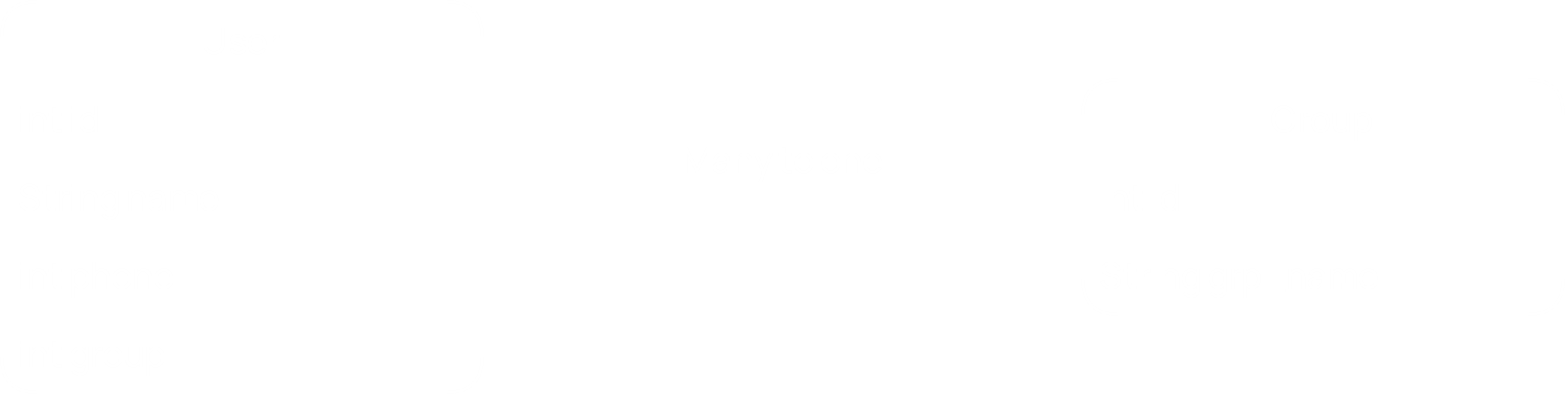
JAVA

## Relationships

There are four annotations we can use to create relationships between entities, @OneToOne, @OneToMany, @ManyToOne and @ManyToMany. The direction of these relationships can either be **unidirectional**, with the **owning side** having a relationship field that refers to the other entity, or **bidirectional**, with both the owning side and the **inverse side** having fields that refer to each other. This difference will only become apparent when using the @OneToMany and the @ManyToOneannotations, with the owning side having the first annotation and the inverse side having the other. For @ManyToMany, either side can be the owning side. For @OneToOne, the side with the foreign key is the owning side.

For **bidirectional relationships**, the inverse side must refer to the field in the owning side to which it is mapped using the mappedBy element.

To understand all of this better, consider the many-to-one relationship below:



If we want to create a **unidirectional** relationship in this case, we can do so like this:

public class User {  
 private int id;  
 private String name;  
 private int phone;  
 @ManyToOne  
 @JoinColumn(name = "group\_id")  
 private Group group;  
}

public class Group {  
 private int id;  
 private String grp\_name;  
}

JAVA

Thus, the User class is saying that many of its members may have Group values that are the same.

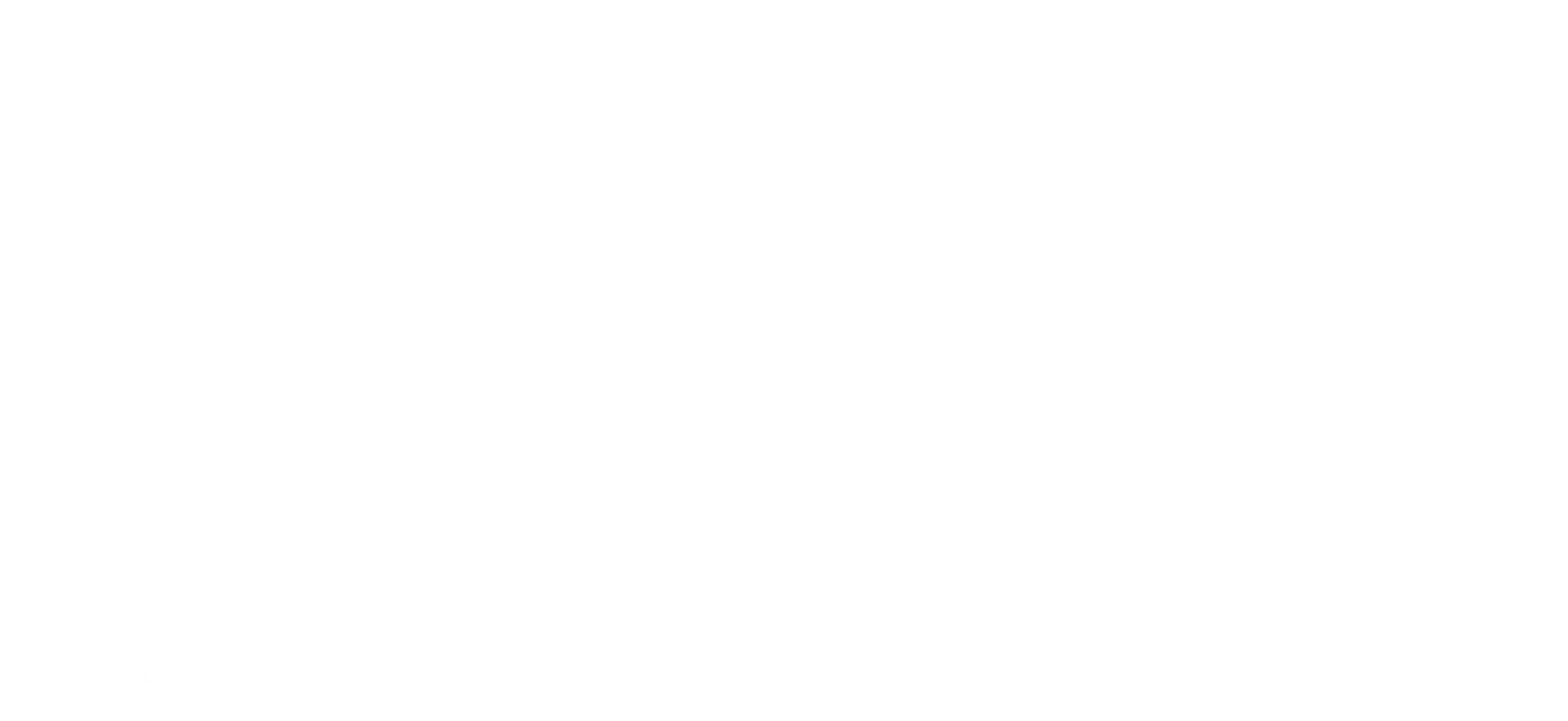
Instead, if we want a **bidirectional relationship**, we can do so like this:

public class User {  
 private int id;  
 private String name;  
 private int phone;  
 @ManyToOne  
 @JoinColumn(name = "group\_id")  
 private Group group;  
}  
  
public class Group {  
 private int id;  
 private String grp\_name;  
 @OneToMany(mappedBy = "group")  
 private *List*<User> users;  
}

JAVA

Essentially, the Group class has become aware that there is a list of User objects that refer to it.

For a more complicated example, consider the following many-to-many relationship:



In this case, we must create a **junction table**. This will be created inside whichever entity we decide is the owning side.

public class Employee {  
 *// other properties* @JoinTable(  
 *// junction table name* name = "employee\_projects",  
 *// columns from owning side* joinColumns = { @JoinColumn(name = "employee\_id") },  
 *// columns from inverse side* inverseJoinColumns = { @JoinColumn(name = "project\_id") }  
 )  
 *Set*<Project> projects = new HashSet<Project>();  
}  
  
public class Project {  
 *// other properties* @ManyToMany(mappedBy = "projects", cascade = {CascadeType.*ALL*})  
 private *Set*<Employee> employees = new HashSet<Employee>();  
}

JAVA

Notice that the @ManyToMany annotation is only used on the inverse side, not the owning side.

## Managing Entities

All the entity instances that exist within a single data store are said to be part of the same **persistence context**. This defines the scope under which the entities are created, persisted and removed. To interact with a particular persistence context, we use an *EntityManager*, which is an interface which defines the methods that can be used with the persistence context.

The EntityManager API:

* Manages the state and lifecycle of entities
* Creates and removes persistent entity instances
* Finds entities by their primary key
* Allows queries to be run on the entities

There are two types of EntityManagers, Application-Managed EntityManagers and Container-Managed EntityManagers.

### Application-Managed EntityManagers

In this method, the Persistence class is used to get a reference to an *EntityManagerFactor*. This is done using a named **persistence class**, which we will be looking into soon. The *EntityManagerFactory* in turn creates the *EntityManager*.

*EntityManagerFactory* emf =  
 Persistence.*createEntityManagerFactory*(  
 "Some Persistence Unit");  
*EntityManager* em = emf.createEntityManager();  
  
em.getTransaction().begin();  
*// this is where all our queries go*em.getTransaction().commit();  
em.close();  
emf.close();

JAVA

### Container Managed EntityManager

In this case, the lifecycle of the *EntityManager* is managed by the container itself. The persistence context in this case is injected in when needed.

public class BookmarkService {  
 @PersistenceContext  
 private *EntityManager* em;  
  
 public void save(Bookmark bookmark) {  
 *// if id not in database, add* if (bookmark.getId() == null) em.persist(bookmark);  
 else em.merge(bookmark); *// else override* }  
}

JAVA

Notice that in this case we did not need to specify which persistence unit to use, or use the begin() or commit() methods.

### Entity Operations

There are several operations we can perform using an EntityManager on Entity objects.

* persist() – Insert entity instance (i.e. insert a record)
* remove() – Delete entity instance
* refresh() – Reload the entity instance
* merge() – Synchronize database with entity instance
* find() – Execute a primary key query; takes primary key as parameter
* createQuery() – Creates a query instance using dynamic JP QL
* createNamedQuery() – Create a query instance for a predefined query
* createNativeQuery() – Create a query instance for an SQL query
* contains() – Determine if the entity is managed by the PC
* flush() – Replace entity instance in database

### Persistence Units

A **persistence unit** defines all the entity classes that should be associated with a single EntityManager. These are placed in a persistence.xml file.

<persistence xmlns="http://java.sun.com/xml/ns/persistence"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence  
 http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd"  
 version="2.0">  
 <persistence-unit name="default" transaction-type="JTA">  
 *<!-- provider only necessary when multiple available -->* <provider>org.hibernate.ejb.HibernatePersistence</provider>

*<!-- JNDI name of datasource ; not needed for local database -->* <jta-data-source>java:/DefaultDS</jta-data-source>  
 *<!-- only need to add classes that are not annotated with @Entity -->* <class>com.example.demo.Employee</class>  
 <class>com.example.demo.Person</class>  
 <class>com.example.demo.Address</class>  
 <properties>  
 <property name="hibernate.dialect"  
 value="org.hibernate.dialect.HSQLDialect"/>  
 <property name="hibernate.hbm2ddl.auto"  
 value="create-drop"/>  
  
 *<!-- following lines needed for local database -->* <property name="javax.persistence.jdbc.driver"  
 value="oracle.jdbc.OracleDriver"/>  
 <property name="javax.persistence.jdbc.url"  
 value="jdbc:oracle:thin:@localhost:5500/em"/>  
 <property name="javax.persistence.jdbc.user"  
 value="username"/>  
 <property name="javax.persistence.jdbc.password"  
 value="password"/>  
 </properties>  
 </persistence-unit>  
</persistence>

XML

More information about persistence units can be found [here](https://stackoverflow.com/questions/43243485/whats-the-meaning-of-properties-of-persistence-xml-file).