JPA

Java persistence API

JPA Entity classes

JPA Entity classes are user defined classes whose instances can be stored in a database.

JPA Persistable Types

The term persistable types refers to data types that can be used in storing data in the database. JPA persistable types are:

- User defined classes - Entity classes, Mapped superclasses, Embeddable classes.
- Simple Java data types: Primitive types, Wrappers, String, Date and Math types.
- Multi value types - Collections, Maps and Arrays.
- Miscellaneous types: Enum types and Serializable types (user or system defined).

Note: Only instances of entity classes can be stored in the database directly. Other persistable types can be embedded in entity classes as fields. In addition, only instances of entity classes preserve identity and are stored only once even if they are referenced multiple times. Referencing instances of other persistable types from multiple persistent fields would cause data duplication in the database.

JPA Persistable Types

Entity Classes

An entity class is an ordinary user defined Java class whose instances can be stored in the database. The easy way to declare a class as an entity is to mark it with the Entity annotation:

```
import javax.persistence.Entity;

@Entity
public class MyEntity {
}
```

Entity Class Requirements

A portable JPA entity class:

- should be a top-level class (i.e. not a nested / inner class).
- should have a `public` or `protected` no-arg constructor.
- cannot be `final` and cannot have `final` methods or `final` instance variables.

Aside from these constraints an entity class is like any other Java class. It can extend either another entity class or a non-entity user defined class (but not system classes, such as `ArrayList`) and implement any interface. It can contain constructors, methods, fields and nested types with any access modifiers (public, protected, package or private) and it can be either concrete or abstract.

JPA Persistable Types

Entity Class Names

Entity classes are represented in queries by *entity names*. By default, the entity name is the unqualified name of the entity class (i.e. the short class name excluding the package name). A different entity name can be set explicitly by using the name attribute of the Entity annotation:

```
@Entity(name="MyName")
public class MyEntity {
}
```

Entity names must be unique. When two entity classes in different packages share the same class name, explicit entity name setting is required to avoid collision.

Mapped Superclasses

In JPA, classes that are declared as mapped superclasses have some of the features of entity classes, but also some restrictions.

Mapped superclasses are really only useful in applications that use an ORM-based JPA provider (such as Hibernate, TopLink, EclipseLink, OpenJPA, JPOX, DataNucleus, etc.).

JPA Persistable Types

Embeddable Classes

Embeddable classes are user defined persistable classes that function as value types. As with other non entity types, instances of an embeddable class can only be stored in the database as embedded objects, i.e. as part of a containing entity object.
A class is declared as embeddable by marking it with the Embeddable annotation:

```
@Embeddable
public class Address {
    String street;
    String city;
    String state;
    String country;
    String zip;
}
```

Instances of embeddable classes are always embedded in other entity objects and do not require separate space allocation and separate store and retrieval operations. Therefore, using embeddable classes can save space in the database and improve efficiency.
Embeddable classes, however, do not have an identity (primary key) of their own which leads to some limitations (e.g. their instances cannot be shared by different entity objects and they cannot be queried directly), so a decision whether to declare a class as an entity or embeddable requires case by case consideration.

JPA Persistable Types

Simple Java Data Types

All the following simple Java data types are persistable:

- Primitive types: `boolean`, `byte`, `short`, `char`, `int`, `long`, `float` and `double`.
- Equivalent wrapper classes from package java.lang:
- `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float` and `Double`.
- `java.math.BigInteger`, `java.math.BigDecimal`.
- `java.lang.String`.
- `java.util.Date`, `java.util.Calendar`,
- `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`.

Date and time types are discussed in more detail in the next paragraph.

JPA Persistable Types

Date and Time (Temporal) Types

The `java.sql` date and time classes represent different parts of dates and times:

- `java.sql.Date` - represents date only (e.g. `2010-12-31`).
- `java.sql.Time` - represents time only (e.g. `23:59:59`).
- `java.sql.Timestamp` - represents date and time (e.g. `2010-12-31 23:59:59`).

The `java.util.Date` and `java.util.Calendar` types, on the other hand, are generic and can represent any of the above, using the `@Temporal` JPA annotation:

```
@Entity
public class DatesAndTimes {
    // Date Only:
    java.sql.Date date1;
    @Temporal(TemporalType.DATE) java.util.Date date2
    @Temporal(TemporalType.DATE) java.util.Calendar date3;

    // Time Only:
    java.sql.Time time1;
    @Temporal(TemporalType.TIME) java.util.Date time2;
    @Temporal(TemporalType.TIME) java.util.Calendar time3;

    // Date and Time:
    java.sql.Timestamp dateAndTime1;
    @Temporal(TemporalType.TIMESTAMP) java.util.Date dateAndTime2;
    @Temporal(TemporalType.TIMESTAMP) java.util.Calendar dateAndTime3;
    java.util.Date dateAndTime4; // date and time but not JPA portable
    java.util.Calendar dateAndTime5; // date and time but not JPA portable
}
```

JPA Persistable Types

Persisting pure dates (without the time part), either by using the `java.sql.Date` type or by specifying the `@Temporal(TemporalType.DATE)` annotation has several benefits:

- It saves space in the database.
- It is more efficient (storage and retrieval is faster).
- It simplifies queries on dates and ranges of dates.

When an entity is stored, its date and time fields are automatically adjusted to the requested mode. For example, fields `date1`, `date2` and `date3` above may be initialized as `new Date()`, i.e. with both date and time. Their time part is discarded when they are stored in the database.

JPA Persistable Types

Multi Value Types

The following multi value types are persistable:

- Collection types from package `java.util`: `ArrayList`, `Vector`, `Stack`, `LinkedList`, `ArrayDeque`, `PriorityQueue`, `HashSet`, `LinkedHashSet`, `TreeSet`.
- Map types from package `java.util`: `HashMap`, `Hashtable`, `WeakHashMap`, `IdentityHashMap`, `LinkedHashMap`, `TreeMap` and `Properties`.
- Arrays (including multi dimensional arrays).

Both generic (e.g. `ArrayList<String>`) and non generic (e.g. `ArrayList`) collection and map types are supported, as long as their values (i.e. elements in collections and arrays and keys and values in maps) are either `null` values or instances of persistable types.

Proxy Classes

When entities are retrieved from the database, instances of mutable persistable system types (i.e. collections, maps and dates) are instantiated using proxy classes that extend the original classes and enable transparent activation and transparent persistence.
Most applications are not affected by this, because proxy classes extend the original Java classes and inherit their behavior. The difference can be noticed in the debugger view and when invoking the `getClass` method on an object of such proxy classes.

JPA Persistable Types

Enum Types

Every enum type (user defined or system defined) is persistable. But, if future portability to other platforms is important, only values of user defined enum types should be persisted.
By default, enum values are represented internally by their ordinal numbers. Caution is required when modifying an enum type that is already in use in an exiting database. New enum fields can be added safely only at the end (with new higher ordinal numbers).
Alternatively, enum values can be represented internally by their names. In that case, the names must be fixed, since changing a name can cause data loss in existing databases.
The `@Enumerated` annotation enables choosing the internal representation:

```
@Entity
public class Style {
    Color color1; // default is EnumType.ORDINAL
    @Enumerated(EnumType.ORDINAL) Color color2;
    @Enumerated(EnumType.STRING) Color color3;
}

enum Color { RED, GREEN, BLUE };
```

In the above example, values of the `color1` and `color2` fields are stored as ordinal numbers (i.e. 0, 1, 2) while values of the `color3` field are stored internally as strings (i.e. “RED”, “GREEN”, “BLUE”).

JPA Persistable Types

Serializable Types

Every serializable class (user defined or system defined) is also persistable, but relying on serialization in persisting data has a severe drawback in lack of portability. Therefore, it is recommended to only use explicitly specified persistable types.

JPA Entity Fields

Fields of persistable user defined classes (entity classes, embeddable classes and mapped superclasses) can be classified into the following five groups:

- Transient fields
- Persistent fields
- Inverse (Mapped By) fields
- Primary key (ID) fields
- Version field

The first three groups (transient, persistent and inverse fields) can be used in both entity classes and embeddable classes. However, the last two groups (primary key and version fields) can only be used in entity classes.

JPA Entity Fields

Transient Fields

Transient entity fields are fields that do not participate in persistence and their values are never stored in the database (similar to transient fields in Java that do not participate in serialization). Static and final entity fields are always considered to be transient. Other fields can be declared explicitly as transient using either the Java `transient` modifier (which also affects serialization) or the JPA `@Transient` annotation (which only affects persistence):

```
@Entity
public class EntityWithTransientFields {
    static int transient1; // not persistent because of static
    final int transient2 = 0;  // not persistent because of final
    transient int transient3; // not persistent because of transient
    @Transient int transient4; // not persistent because of @Transient
}
```

The above entity class contains only transient (non persistent) entity fields with no real content to be stored in the database.

JPA Entity Fields

Every non-static non-final entity field is persistent by default unless explicitly specified otherwise (e.g. by using the `@Transient` annotation).

Storing an entity object in the database does not store methods or code. Only the persistent state of the entity object, as reflected by its persistent fields (including persistent fields that are inherited from ancestor classes), is stored.

When an entity object is stored in the database every persistent field must contain either `null` or a value of one of the supported persistable types.

Every persistent field can be marked with one of the following annotations:

- `OneToOne`, `ManyToOne` - for references of entity types.
- `OneToMany`, `ManyToMany` - for collections and maps of entity types.
- `Basic` - for any other persistable type.

JPA Entity Fields

In JPA only `Basic` is optional while the other annotations above are required when applicable. For example:

```
@Entity
public class EntityWithFieldSettings {
    @Basic(optional=false) Integer field1;
    @OneToOne(cascade=CascadeType.ALL) MyEntity field2;
    @OneToMany(fetch=FetchType.EAGER) List<MyEntity> field3;
}
```

The entity class declaration above demonstrates using field and relationship annotations to change the default behavior. `null` values are allowed by default. Specifying `optional=false` (as demonstrated for `field1`) causes an exception to be thrown on any attempt to store an entity with a `null` value in that field.

A persistent field whose type is embeddable may optionally be marked with the `@Embedded` annotation:

```
@Entity
public class Person {
    @Embedded Address address;
}
```

JPA Entity Fields

Inverse Fields

Inverse (or mapped by) fields contain data that is not stored as part of the entity in the database, but is still available after retrieval by a special automatic query.

Note: Navigation through inverse fields is **much less efficient** than navigation through ordinary persistent fields, since it requires running queries. Inverse fields are essential for collection fields when using ORM JPA implementations. Avoiding bidirectional relationships and inverse fields, and maintaining two unidirectional relationships is usually much more efficient (unless navigation in the inverse direction is rare).

The following entity classes demonstrate a bidirectional relationship:

```
@Entity
public class Employee {
    String name;
    @ManyToOne Department department;
}

@Entity
public class Department {
    @OneToMany(mappedBy="department") Set<Employee> employees;
}
```

JPA Entity Fields

The `mappedBy` element (above) specifies that the `employees` field is an inverse field rather than a persistent field. The content of the `employees` set is not stored as part of a `Department` entity. Instead, employees is automatically populated when a `Department` entity is retrieved from the database. JPA accomplishes this by effectively running the following query (where `:d` represents the `Department` entity):

```
SELECT e FROM Employee e WHERE e.department = :d
```

The `mappedBy` element defines a bidirectional relationship. In a bidirectional relationship, the side that stores the data (the `Employee` class in our example) is the owner. Only changes to the owner side affect the database, since the other side is not stored and calculated by a query.

An index on the owner field may accelerate the inverse query and the load of the inverse field. But even with an index, executing a query for loading a field is relatively slow. Therefore, if the `employees` field is used often, a persistent field rather than inverse field is expected to be more efficient. In this case, two unidirectional and unrelated relationships are managed by the `Employee` and the `Department` classes and the application is responsible to keep them synchronized.
Inverse fields may improve efficiency when managing very large collections that are changed often. This is because a change in the inverse field does not require storing the entire collection again. Only the owner side is stored in the database.

JPA Entity Fields

Special settings are available for inverse fields whose type is `List` or `Map`. For an inverse list field, the order of the retrieved owner entities can be set by the `OrderBy` annotation:

```
@Entity
public class Department {
    @OneToMany(mappedBy="department") @OrderBy("name")
    List<Employee> employees;
}
```

In that case the employees field is filled with the results of the following query:

```
SELECT e FROM Employee e WHERE e.department = :d ORDER BY e.name
```

The specified field ("name") must be a sortable field of the owner side.
For an inverse map field, the keys can be extracted from the inverse query results by specifying a selected key field using the `MapKey` annotation:

```
@Entity
public class Department {
    @OneToMany(mappedBy="department") @MapKey(name="name")
    Map<String,Employee> employees;
}
```

The employees map is filled with a mapping of employee names to the `Employee` objects to which they pertain.

JPA Entity Fields

Single value inverse fields are also supported:

```
@Entity
public class Employee {
    @OneToOne MedicalInsurance medicalInsurance;
}

@Entity
public class MedicalInsurance {
    @OneToOne(mappedBy="medicalInsurance") Employee employee;
}
```

A single value inverse field is less efficient than an inverse collection or map field because no proxy class is used and the inverse query is executed eagerly when the entity object is first accessed.

JPA Entity Fields

Version Field

The initial version of a new entity object (when stored in the database for the first time) is 1. For every transaction in which an entity object is modified its version number is automatically increased by one.

You can expose entity object versions and make their values accessible to your application by marking the version fields in your entity classes with the `Version` annotation. In addition, a version field should have a numeric type:

```
@Entity public class EntityWithVersionField {
     @Version long version;
}
```

JPA Entity Fields

Version fields should be treated as read only by the application and no mutator methods should be written against them. Only one version field per entity class is allowed. In fact, a single version field per entity class hierarchy is sufficient because a version field is inherited by subclasses.

Nevertheless, defining a version field has some advantages:

- The application becomes more portable (to ORM-based JPA implementations).
- Even when entity object versions are not in use directly by the application, exposing their values might be useful occasionally for debugging and logging.
- Version values cannot be preserved for detached entity objects unless either the entity class is enhanced or a version field is explicitly defined.

JPA Entity Fields

Property Access

When an entity is being stored to the database data is extracted from the persistent fields of that entity. Likewise, when an entity is retrieved the persistent fields are initialized with data from the database.
To use property access mode, every non-transient field must have get and set methods based on the Java bean property convention.

Property access is enabled by moving all the JPA annotations from the fields to their respective get methods and
specifying the Access annotation on the class itself:

```
@Entity @Access(AccessType.PROPERTY)
public static class PropertyAccess {
    private int _id;
    @Id int getId() { return _id; }
    void setId(int id) { _id = id; }

    private String str;
    String getStr() { return str; }
    void setStr(String str) { this.str = str; }
}
```

In some JPA implementations, such as Hibernate, using property access may have some performance benefits. Therefore, considering the extra complexity that is involved in setting up and maintaining property access, the default field access mode is usually preferred.

JPA Primary Key

Every entity object that is stored in the database has a primary key. Once assigned, the primary key cannot be modified. It represents the entity object as long as it exists in the database.

Entity Identification

Every entity object in the database is uniquely identified (and can be retrieved from the database) by the combination of its type and its primary key. Primary key values are unique per entity class. Instances of different entity classes, however, may share the same primary key value.

Only entity objects have primary keys. Instances of other persistable types are always stored as part of their containing entity objects and do not have their own separate identity.

JPA Primary Key

Automatic Primary Key

By default the primary key is a sequential 64 bit number (`long`). The primary key of the first entity object in the database is 1, the primary key of the second entity object is 2, etc. Primary key values are **not** recycled when entity objects are deleted from the database.

The primary key value of an entity can be accessed by declaring a primary key field:

```
@Entity
public class Project {
    @Id @GeneratedValue long id; // still set automatically
     :
}
```

The `@Id` annotation marks a field as a primary key field.
The `@GeneratedValue` annotation specifies that the primary key is automatically allocated.

JPA Primary Key

Application Set Primary Key

If an entity has a primary key field that is not marked with `@GeneratedValue`, automatic primary key value is not generated and the application is responsible to set a primary key by initializing the primary key field. That must be done before any attempt to persist the entity object:

```
@Entity
public class Project {
    @Id long id; // must be initialized by the application
     :
}
```

A primary key field that is set by the application can have one of the following types:

- Primitive types: `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`.
- Equivalent wrapper classes from package java.lang:
- `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`.
- `java.math.BigInteger`, `java.math.BigDecimal`.
- `java.lang.String`.
- `java.util.Date`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`.
- Any enum type.
- Reference to an entity object.

JPA Primary Key

Composite Primary Key

A composite primary key consist of multiple primary key fields. Each primary key field must be one of the supported types listed above.

For example, the primary key of the following `Project` entity class consists of two fields:

```
@Entity @IdClass(ProjectId.class)
public class Project {
    @Id int departmentId;
    @Id long projectId;
     :
}
```

When an entity has multiple primary key fields, JPA requires defining a special ID class that is attached to the entity class using the `@IdClass` annotation. The ID class reflects the primary key fields and its objects can represent primary key values:

```
Class ProjectId {
    int departmentId;
    long projectId;
}
```

JPA Primary Key

Embedded Primary Key

An alternate way to represent a composite primary key is to use an embeddable class:

```
@Entity
public class Project {
    @EmbeddedId ProjectId id;
     :
}

@Embeddable
Class ProjectId {
    int departmentId;
    long projectId;
}
```

The primary key fields are defined in an embeddable class. The entity contains a single primary key field that is annotated with `@EmbeddedId` and contains an instance of that embeddable class. When using this form a separate ID class is not defined because the embeddable class itself can represent complete primary key values.

JPA Primary Key

Obtaining the Primary Key

JPA 2 provides a generic method for getting the object ID (primary key) of a specified managed entity object. For example:

```
PersistenceUnitUtil util = emf.getPersistenceUnitUtil();
Object projectId = util.getIdentifier(project);
```

A `PersistenceUnitUtil` instance is obtained from the `EntityManagerFactory`.
The `getIdentifier` method takes one argument, a managed entity object, and returns the primary key. In case of a composite primary key - an instance of the ID class or the embeddable class is returned.

JPA Primary Key

Using Primary Keys for Object Clustering

Entity objects are physically stored in the database ordered by their primary key. Sometimes it is useful to choose a primary key that helps clustering entity objects in the database in an efficient way. This is especially useful when using queries that return large result sets.
As an example, consider a real time system that detects events from various sensors and stores the details in a database. Each event is represented by an Event entity object that holds time, sensor ID and additional details. Suppose that queries that retrieve all the events of a specified sensor in a specified period are common and return thousands of Event objects. In that case the following primary key can significantly improve query run performance:

```
@Entity
public class Event {
    @EmbeddedId EventId id;
     :
}

@Embeddable
Class EventId {
    int sensorId;
    Date time;
}
```

Because entity objects are ordered in the database by their primary key, events of the same sensor during a period of time are stored continuously and can be collected by accessing a minimum number of database pages.
On the other end, such a primary key requires more storage space (especially if there are many references to Event objects in the database because references to entities hold primary key values) and is less efficient in store operations. Therefore, all factors have to be considered and a benchmark might be needed to evaluate the different alternatives in order to select the best solution.
Marking a field with the `@GeneratedValue` annotation specifies that a value will be automatically generated for that field.

Auto Generated Values

The Auto Strategy

JPA maintains a special global number generator for every database. This number generator is used to generate automatic object IDs for entity objects with no primary key fields defined (as explained in the previous section).
The same number generator is also used to generate numeric values for primary key fields annotated by `@GeneratedValue` with the `AUTO` strategy:

```
@Entity
public class EntityWithAutoId1 {
    @Id @GeneratedValue(strategy=GenerationType.AUTO) long id;
     :
}
```

`AUTO` is the default strategy, so the following definition is equivalent:

```
@Entity
public class EntityWithAutoId2 {
    @Id @GeneratedValue long id;
     :
}
```

During a commit the `AUTO` strategy uses the global number generator to generate a primary key for every new entity object. These generated values are unique at the database level and are never recycled, as explained in the previous section.

Auto Generated Values

The Identity Strategy

The `IDENTITY` strategy is very similar to the `AUTO` strategy:

```
@Entity
public class EntityWithIdentityId {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY) long id;
     :
}
```

The `IDENTITY` strategy also generates an automatic value during commit for every new entity object. The difference is that a separate identity generator is managed per type hierarchy, so generated values are unique only per type hierarchy.

Auto Generated Values

The Sequence Strategy

The sequence strategy consists of two parts - defining a named sequence and using the named sequence in one or more fields in one or more classes. The `@SequenceGenerator` annotation is used to define a sequence and accepts a name, an initial value (the default is 1) and an allocation size (the default is 50). A sequence is global to the application and can be used by one or more fields in one or more classes. The `SEQUENCE` strategy is used in the `@GeneratedValue` annotation to attach the given field to the previously defined named sequence:

```
@Entity
// Define a sequence - might also be in another class:
@SequenceGenerator(name="seq", initialValue=1, allocationSize=100)
public class EntityWithSequenceId {
    // Use the sequence that is defined above:
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="seq")
    @Id long id;
}
```

Unlike `AUTO` and `IDENTITY`, the `SEQUENCE` strategy generates an automatic value as soon as a new entity object is persisted (i.e. before commit). This may be useful when the primary key value is needed earlier. To minimize round trips to the database server, IDs are allocated in groups. The number of IDs in each allocation is specified by the `allocationSize` attribute. It is possible that some of the IDs in a given allocation will not be used. Therefore, this strategy does not guarantee there will be no gaps in sequence values.

Auto Generated Values

The Table Strategy

The `TABLE` strategy is very similar to the `SEQUENCE` strategy:

```
@Entity
@TableGenerator(name="tab", initialValue=0, allocationSize=50)
public class EntityWithTableId {
    @GeneratedValue(strategy=GenerationType.TABLE, generator="tab")
    @Id long id;
}
```

ORM-based JPA providers (such as Hibernate, TopLink, EclipseLink, OpenJPA, JPOX, etc.) simulate a sequence using a table to support this strategy.
A tiny difference is related to the initial value attribute. Whereas the `SEQUENCE` strategy maintains the next sequence number to be used the `TABLE` strategy maintains the last value that was used. The implication for the `initialValue` attribute is that if you want sequence numbers to start with 1 in the `TABLE` strategy `initialValue=0` has to be specified in the `@SequenceGenerator` annotation.

JPA Persistence Unit

A JPA Persistence Unit is a logical grouping of user defined persistable classes (entity classes, embeddable classes and mapped superclasses) with related settings. Defining a persistence unit is required by JPA.

persistence.xml

Persistence units are defined in a `persistence.xml` file, which has to be located in the `META-INF` directory in the classpath. One `persistence.xml` file can include definitions for one or more persistence units. The portable way to instantiate an `EntityManagerFactory` in JPA (as explained in the JPA Overview section) requires a persistence unit.

JPA Persistence Unit

The following `persistence.xml` file defines one persistence unit:

```
<?xml version="1.0" encoding="UTF-8" ?>
<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_1_0.xsd" version="1.0">

   <persistence-unit name="my-pu">
     <description>My Persistence Unit</description>
     <provider>com.objectdb.jpa.Provider</provider>
     <mapping-file>META-INF/mappingFile.xml</mapping-file>
     <jar-file>packedEntity.jar</jar-file>
     <class>sample.MyEntity1</class>
     <class>sample.MyEntity2</class>
     <properties>
       <property name="javax.persistence.jdbc.url"
                 value="objectdb://localhost/my.odb"/>
       <property name="javax.persistence.jdbc.user" value="admin"/>
       <property name="javax.persistence.jdbc.password" value="admin"/>
     </properties>
   </persistence-unit>

</persistence>
```

JPA Persistence Unit

A persistence unit is defined by a `persistence-unit` XML element. The required *name* attribute (“my-pu” in the example) identifies the persistence unit when instantiating an `EntityManagerFactory`. It may also have optional sub elements:

- The provider element indicates which JPA implementation should be used. If not specified, the first JPA implementation that is found is used.
- The mapping-file elements specify XML mapping files that are added to the default `META-INF/orm.xml` mapping file. Every annotation that is described in this manual can be replaced by equivalent XML in the mapping files (as explained below).
- The `jar-file` elements specify JAR files that should be searched for managed persistable classes.
- The `class` elements specify names of managed persistable classes (see below).
- The `property` elements specify general properties. JPA 2 defines standard properties for specifying database url, username and password, as demonstrated above.

JPA Persistence Unit

XML Mapping Metadata

Both JPA mapping files and JDO *package.jdo* files are supported. This manual focuses on using annotations which are more common and usually more convenient. Details on using XML metadata can be found in the JPA and JDO specifications and books.

JPA Persistence Unit

Managed Persistable Classes

JPA requires registration of all the user defined persistable classes (entity classes, embeddable classes and mapped superclasses), which are referred to by JPA as managed classes, as part of a persistence unit definition.

Classes that are mentioned in mapping files as well as annotated classes in the `JAR` that contains the `persistence.xml` file (if it is packed) are registered automatically. Other classes have to be registered explicitly by using `class` elements (for single class registration) or `jar-file` elements (for registration of all the classes in the jar file).