Programming for the Java Platform Enterprise Edition

Java Persistence API (JPA)

Opracował:
mgr inż. Tomasz Jastrząb

Table of contents

1 Introduction

Java Persistence API (JPA) is a standard for data persistence, that first appeared as part of EJB 3.0 specification. Up to date three versions of JPA specification were released with the newest one having number 2.1.

1.1 Characteristic features of JPA

The basic feature of JPA is the fact that, unlike the solutions known from EJB 2.0 and 2.1, i.e. *Container Manager Persistence* (CMP) and *Bean Managed Persistence* (BMP), it is based on POJOs (*Plain Old Java Objects*). These are simple objects that neither implement any specific interfaces nor extend any particular classes – they are just plain classes with properties, accessors and mutators (i.e. getters and setters).

JPA is a standard for Object-to-Relational Mapping (ORM) solutions, allowing for automatic conversion of database table schema to Java objects and vice versa. Mappings configuration is performed basing on XML files (in older versions) or using annotations. Aforementioned annotations are defined in javax.persistence package. Thanks to the use of ORM mechanisms the programmer does not have to manually build data transfer objects or create SQL queries and other mechanisms provided by JDBC specification. The advantage of this solution is greater resistance against errors (it is not necessary to "concatenate" SQL queries, which can be error-prone) and no need for knowing the rules of SQL queries structure (or at least relatively large reduction of such a need). Additionally, implementations of this standard often allow for automatic generation (or validation) of database schema based on defined objects or sometimes the other way round as well.

JPA is only a specification, defining the interface of persistence layer, not giving however, any direct implementation. Consequently, to use JPA mechanisms it is necessary to select a product complying with this specification. Exemplary implementations include Toplink of Oracle Inc., EclipseLink and probably the most popular Hibernate, being one of the main sources of JPA specification (the author of Hibernate, Gavin King was one of the co-authors of initial version of JPA 1.0 specification). Using the annotations from javax.persistence package one obtains a universal solution which allows for easy interchange of implementations, without the need for any code modifications.

1.2 JPA configuration

To properly configure JPA environment it is necessary to perform two steps:

1. selection of one of the implementations and addition of necessary JAR archives to application's classpath (for details see section 2.1),
2. creation of *persistence.xml* configuration file placed in *META-INF* directory inside source folder.

The structure of *persistence.xml* file is usually quite similar regardless of chosen implementation, apart from options specific for given solution (these options are usually related to database schema generation based on entity classes, logging of sql queries sent to the database, etc.). Among the most important tags appearing in almost every *persistence.xml* file one may distinguish the following:

- persistence-unit – this tag encloses majority of other tags, defining the persistence unit. Inside single *persistence.xml* file there may be many persistence units, but each of them should have different name. The name of persistence unit is given in name attribute,
- provider – this tag depends on specific implementation of JPA, it shows the main class of persistence unit,
- class – appearing multiple times inside given persistence unit, specifies fully qualified names (including packages) of entity classes,
- property – tag that depending on the values of name and value attributes may for instance define database connection configuration (using JDBC or data source), as well as some additional properties such as automatic database structure generation, sql queries logging, etc..

Using Hibernate as JPA provider it is possible to replace separate database connection properties and some other additional ones with a single property named hibernate.ejb.cfgfile, which should give the path to Hibernate's configuration file (i.e. hibernate.cfg.xml).

1.3 Annotations

Apart from creating *persistence.xml* file, it is necessary to add appropriate annotations to the defined POJOs. Besides, usually, although it is not technically required, entity classes should implement *java.io.Serializable* interface and override *hashCode* and *equals* methods. Overriding of these methods is important from the point of view of proper identical objects/database records identification, which should be based on primary key value, which by its nature is unique.

@Entity, @Table *@Entity* annotation is placed on the level of a class and it is one of the obligatory annotations that have to appear for proper entity class definition. On the other hand *@Table* annotation is optional provided that the name of entity class and table name are the same. Otherwise *@Table* is mainly used for naming the table to which given entity class corresponds – the table name is given in *name* attribute.

@Id, @GeneratedValue *@Id* annotation is the second annotation, after *@Entity*, that is mandatory for proper entity class definition. It specifies primary key column provided that it is a simple primary key (i.e. composed of a single column). Usually it is accompanied by *@GeneratedValue* annotation denoting automatic primary key value generation. In the default version *@GeneratedValue* decides which mechanism is used for primary key value generation based on the availability in particular database server – it might be e.g. AUTO_INCREMENT or IDENTITY. This annotation allows also for naming own generator class if the primary key values are generated programmatically.

The location of *@Id* annotation (on field or accessor method level) determines the location of all other column-related annotations and defines the way respective entity properties are accessed. In case it is placed on the field level, then persistence unit reads and writes entity fields directly, regardless of their visibility level. In case of the second option accessors and mutators are used for fields manipulation. The difference between both solutions may be visible in case of lazy initialization of entity relationships and the use of proxy objects.

@Embeddable, @EmbeddedId, @IdClass In case of composite primary key, *@Id* annotation is not enough. In such a situation it is necessary to define a separate class representing the primary key, containing fields (columns) making up the primary key. This class may be marked with *@Embeddable* annotation and consequently an object of this class should be included as a field of an entity class and marked with *@EmbeddedId* annotation.

Otherwise, if *@Embeddable* annotation is not present, it is necessary to mark the entity class with *@IdClass* annotation, passing as an argument the class of primary key (in the form of a *Class* object). Additionally in the entity class it is necessary to repeat the fields constituting the primary key and denote each of them with *@Id* annotation.

@Basic, @Temporal, @Transient Annotations *@Basic*, *@Temporal* and *@Transient* are related to entity fields, but their meaning is quite different.

@Transient annotation means, that given field is not made persistent (i.e. is not stored in the database). *@Basic* annotation is a default annotation among others for the simple types, their wrappers, *String* class and enums[1] and usually it may be skipped unless field is to be specified as not optional. Being optional in this case means whether given column may take `NULL` values.

@Temporal annotation is used for *java.util.Date* and *java.util.Calendar* classes, defining the mapping of these types onto database types (as a date, time or timestamp). It is important from the viewpoint of data precision and compliance of stored data, since *Date* type has the precision on the level of miliseconds (date + time), while DATE type in database ignores the time element.

@Column *@Column* annotation is most often used for naming the column in the database corresponding to given entity field, in the case names of the field and column differ (similarly to the case of *@Table* annotation on the entity class level). To denote column name, *name* attribute is used.

Moreover, *@Column* annotation allows among others for specifying whether given column can take `NULL` values (*nullable* attribute) or defining the length of given *String* field (*length* attribute, defines the mapping onto VARCHAR type). Both aforementioned attributes are used at the stage of automatic schema generation based on entity classes.

This annotation allows also for defining *insertable* and *updatable* attributes, which if used, usually appear together having the value `false`. It means that given column is not included in INSERT and UPDATE operations performed on the database. It is important in the case of double column mapping through different entity fields, when it is necessary to denote which field is to be used as a source of data. Such a situation may appear for example if given column is a primary and a foreign key at the same time, but it is mapped as a primary key field and relationship field in the entity.

@OneToOne, @OneToMany, @ManyToOne, @ManyToMany Relationships between tables reflected usually by foreign keys (or join tables), are mapped as references to related classes (or collections of these) on the entity level. From the point of view of Java they may be uni- or bidirectional, meaning that references to related objects appear in one or both entities.

The multiplicity of relationships depends on the annotation used and is directly reflected in the type of related data type – in case of 1-1 or N-1 relationships entity contains only a single reference to related type, while

[1]For the full list of types see here.

in case of 1-N or M-N relationships a collection of references appears. The rule used for understanding the meaning of respective annotations can be summarized as follows: the first part of annotation name is related to the current object, while the second part is related to the object(s) marked with given annotaion. For instance, *@OneToMany* annotation means that one object of current class is related to many objects of the other class (so it is most probably mapped as a foreign key in the related table).

Among the most important attributes appearing in relationships-related annotations (depending on the multiplicity) one may distinguish:

- *mappedBy* attribute – it is used in case of bidirectional relationship to avoid the need for additional join table creation. It names the owner of the relationship, i.e. the entity responsible for introducing modifications in the dependent table,
- *cascade* attribute – it denotes cascading of selected (or all) operations to dependent table, e.g. removing a record in the superior table, should automatically remove all related records in the dependent table,
- *orhpanRemoval* attribute – it is used for "orphans" management, which are entities that were removed from the relationship on the Java level by removing them from the respective collection. In case this attribute is set to false, removed entity is out of control, which in case of superior record removal leaves an unexisting connection in the database.

@JoinColumn, @JoinTable *@JoinColumn* annotation may appear in the case of unidirectional relationship to mark the foreign key column. The most often used attribute of this annotation is *name*, denoting the name of foreign key column, which might be accompanied (but usually it is not necessary) by *referencedColumnName* attribute denoting column name which given relationship is based on. Since usually tables are related based on primary keys, *referencedColumnName* attributes by default takes the value of the primary key column name. Sometimes *insertable* and *updatable* attributes may appear as well, which in this case work the same way as described in the paragraph dedicated to *@Column* annotation.

In case of relationship with multiplicity many-to-many there appears the need for additional join table creation, but this table is not directly mapped to an entity class. Instead it is defined inside *@JoinTable* annotation, which denotes its name (*name* attribute), column name on the owner side of relationship (*joinColumns* attribute) and column name on the other side (*inverseJoinColumns* attribute). Relationship owner in this case is the entity that is responsible for introducing changes in the join table.

1.4 Entity manager

After the entities are created it is necessary to create some code that will be able to manipulate them and cooperate with the database. To achieve this an entity manager (*EntityManager*) has to be created, using *createEntityManager()* method, defined in *EntityManagerFactory*. The aforementioned factory can be in turn obtained using the static method *Persistence.createEntityManagerFactory(...)*, which as an argument takes the name of persistence unit specified in *persistence.xml* file, inside persistence-unit tag.

Using the entity manager, representing database connection, it is possible to add, remove, modify or retrieve database records, directly in the form of entity objects. Operations modifying the database shall be performed inside transactions (i.e. *EntityTransaction* objects). The details related to respective methods of the entity manager can be found here.

2 Laboratory

The description enclosed in this section is based on Netbeans IDE 7.1. It mainly considers the automatic generation of entity classes and automatic data visualization in the form of a table.

2.1 Configuration

Since JPA does not require any enterpise functionality so far, i.e. elements related to Java Platform Enterprise Edition, it is enough to create a standard project working on Java SE platform. To do this one should select ***File*** → ***New Project...***, and then *Java* → *Java Application*.

Depending on the selected JPA implementation, the next step is to add appropriate JAR files related to this implementation. Using the Netbeans IDE (regardless of version number), assuming that necessary archives were downloaded with Netbeans release, to add them to classpath it is necessary to use right mouse button (rmb) on the level of *Libraries* directory and select ***Add Library. . .*** option. After a dialog appears one shall choose selected library, e.g. **EclipseLink (JPA 2.0)** or **Hibernate JPA**. After confirming the selection with **Add Library** button, inside *Libraries* folder, necessary archives shoud appear.

Apart from the JAR files related to persistence layer it is required to add database driver archive, used for establishing the connection with the database. For the Apache Derby database using network driver (i.e. a driver that uses network connection to connect to the database), the required archive is named *derbyclient.jar*. To find its location one may select menu ***Window*** → ***Services***, and then proceed with ***Databases*** → ***Drivers*** → ***Java DB (Network)*** → ***rmb*** → ***Customize***. After locating it, driver can be added once again to the *Libraries* folder, this time by clicking with rmb and choosing ***Add JAR/Folder...***.

2.2 Entity classes generation

To automatically generate entity classes from existing database schema it is necessary to use rmb on the project level, and then select ***New*** → ***Entity Classes from Database***. In the first dialog window that appears, in the field named *Database Connection* one should select existing connection with *sample* database – jdbc:derby://localhost:1527/sample. Next, after the list marked as *Available Tables* fills up with table names one should click **Add All** button, a finally **Next**.

In the next window a package for entities should be given (*Package* field) and at least the checkbox for persistence unit generation shall be checked (*Create Persistence Unit*). Optionally the remaining two checkboxes may be checked as well. Checkbox saying *Generate Named Query Annotations for Persistent Fields* allows for generation of exemplary queries selecting database data based on respective columns (entity fields). Option *Generate JAXB*[2] *Annotations* allows for generation of annotations converting objects to XML representation and vice versa.

After clicking **Next** in the last window option *Use Column Names in Relationships* should be marked and **Finish** button clicked to complete the process.

2.3 Data visualization

Data included in database tables can be visualized using the *JTable* graphical component, representing simple table. Netbeans IDE allows for creation of an application with GUI using drag-and-drop method. To do so it is necessary to use the rmb on the *Source packages* folder or selected package, and choose ***New*** → ***JFrame form***. Next, from the ***Window*** menu one should select ***Palette***, and from the *Swing Controls* tab *Table* should be picked and dropped on the form.

To bind the data to table model, rmb should be used again, but on the *JTable* component level and ***Bind*** → ***elements*** should be selected. In the window that appears one shall click **Import Data to Form...** button, then select once again the existing connection to the *sample* database and pick the table whose data is to be visualized. In the bottom part of the window a list of available columns should appear, out of which the ones to be displayed in the table should be selected.

In the case of columns connected with relationships, it is usually hard to show the important information directly, so one may add another table and follow the steps described previously. But this time instead of importing data from a database table it is better to specify the name of existing *JTable* as the *Binding Source* and in the field called *Binding Expression* the following contents should be inserted ${selectedElement.<relationship field name>}. Thanks to that when a record is selected in the first table, related records from the other table will be shown. To modify displayed columns further on may use ***Table Contents*** option, available after using rmb on the table component, moving then to the *Columns* tab.

[2]JAXB – Java Architecture for XML Binding: information on Oracle Inc. website

3 Tasks

The tasks mentioned in this section are supposed to be performed during the laboratory classes, except for task no. 4 which defines the requirements for the report.

3.1 Task no. 1

Create a class performing CRUD operations (Create, Retrieve, Update, Delete) using one of the entities generated automatically for the *sample* database:

- create single entity manager factory (*EntityManagerFactory*) object,
- create single entity manager (*EntityManager*) object using the factory,
- use transactions (*EntityTransaction*) for DML operations (insert, update, delete),
- entity retrieval shall be performed in two ways: basing on the primary key value and as a query selecting all entities of given type.

3.2 Task no. 2

Create a form allowing for insertion of data stored later in the database. Please use the class created in task no. 1 performing CRUD operations on the database. To create the form you may use either the Designer or your own code.

3.3 Task no. 3

Create manually entity classes for the database schema composed of the following tables:

- Address (address_id PK, city, street, zip_code),
- Subject (subject_id PK, subject_name),
- Student (student_id PK, first_name, last_name, address_id FK),
- Student_Detail (detail_id PK, pesel_no, birth_date, student_id FK),
- Grade (student_id PK/FK, subject_id PK/FK, exam_date PK, mark),
- Student_Subject (student_id PK/FK, subject_id PK/FK).

PK denotes the primary key column, whereas FK denotes the foreign key column. Table Student_Subject is a join table resulting from relationship of multiplicity M-N between tables Student and Subject.

Multiplicities of respective relationships are as follows:

- Address-Student: 1-N,
- Student-Student_Detail: 1-1,
- Student-Grade: 1-N,
- Subject-Grade: 1-N,
- Student-Subject: M-N.

3.4 Task no. 4

Create a console application performing CRUD operations on a database schema composed of at least two tables connected with a relationship. The schema should be designed on your own:

- primary key values should be generated automatically,
- respective CRUD operations should be done in separate methods,
- data constituting the contents of entity classes should be provided by the user,
- erroneous data should be detected and corrected before performing database operations,
- classes responsible for database cooperation and user interface should be independent of each other,
- class fields should be private,
- report should contain Javadoc documentation for all created elements (classes, fields, methods),

The report should consist of three folders: *bin*, *doc*, *src*. *bin* folder should contain a *.jar file and a *.bat file running the JAR file. *doc* folder should contain generated Javadoc documentation including *@author* and *@version* tags on the class level as well as private and package private members. *src* folder should contain source codes with packages.

Identifiers of created classes, fields and methods should be written in English and comply with Java naming convention.