Getting Started with Java Persistence API and SAP JPA 1.0

Applies to:

EHP1 for SAP NetWeaver Composition Environment 7.1 – Preview Version

Summary

A key feature of SAP NetWeaver Application Server, Java EE 5 Edition is the new Java Persistence API (JPA). This article is the first one of a series of articles on JPA, which will appear here in the following months. The article gives an introduction to the fundamental concepts of JPA and illustrates the concepts by means of a small sample application.

Authors: Sabine Heider, Adrian Görler and Julia Reisbich

Company: SAP AG

Created on: 15 April 2009 (Updated version)

10 November 2006 (Original version)

Author Bio

Sabine Heider, SAP AG

Sabine is a member of the SAP NetWeaver Application Server Java development team, focusing on Java persistence and the different persistence technologies. Together with her colleagues, she has developed SAP JPA 1.0, which is part of SAP NetWeaver Application Server, Java EE 5 Edition. She has joined SAP in 1997.

Adrian Görler, SAP AG

Adrian is a member of the SAP NetWeaver Application Server Java development team, focusing on Java persistence and the different persistence technologies. Together with his colleagues, he has developed SAP JPA 1.0, which is part of SAP NetWeaver Application Server, Java EE 5 Edition. He has been with SAP since 1999.

Julia Reisbich

Julia worked as an intern for the SAP NetWeaver Application Server Java development team. In the meantime, she has finished her studies of Digital Media at the University of Applied Sciences Kaiserslautern, location Zweibrücken. Since 2007 she is a member of the SAP NetWeaver Business Process Event Management development team.

Table of Contents

The Java Persistence API

Business applications often have to deal with persistent data stored in a relational database. For the Java platform, there is a variety of standards and proprietary solutions that address the issue, so that it's sometimes not easy to decide. The most fundamental, low-level programming interface is the Java Database Connectivity, better known as JDBC. Although wide-spread, it is actually not convenient to use. JDBC works with tabular row and column data rather than with Java objects, so that a considerable amount of work has to be spent to convert data back and forth. Object-relational persistence frameworks, on the other hand, allow you to work with the Java object model only. The framework undertakes the task to map the Java objects to the relational database, it translates searches for or manipulations on the objects to SQL and it handles the entire communication with the database.

From the beginning, the Java Platform, Enterprise Edition (Java EE, previously called J2EE) came with a built-in object-relational persistence framework based on the Enterprise JavaBeans (EJB) technology: EJB container-managed persistence (CMP). Although standardized and an integral part of any J2EE application server, EJB CMP has never been popular or widely adopted – mainly because it is rather complex and has the reputation to be difficult to use. Many people opted instead for a lightweight, modern object-relational persistence framework like Java Data Objects (JDO), TopLink or Hibernate. However, neither solution is part of the Java EE standard, so there is a certain additional effort to integrate it into a Java EE application server.

With Java EE 5, a new object-relational persistence API, the *Java Persistence API (JPA)*, has been added to the Java EE standard. JPA draws upon the best ideas from the existing persistence frameworks, and finally provides a lightweight and easy-to-use persistence API integrated into any Java EE 5 application server.

SAP NetWeaver Application Server, Java EE 5 Edition contains a JPA implementation that is provided by SAP and called SAP JPA 1.0.

Entities

JPA is an API for storing lightweight Java objects called *entities* in a relational database. Technically, it's just as easy as it could be: An entity is nothing but a regular Java object, often referred to as a *"Plain Old Java Object (POJO)"*. Neither does it have to implement any particular interface or extend a special class – all that technical overhead for which EJB CMP entity beans have gained notoriety is gone with JPA. You are free to define the entity according to your data model, including inheritance and polymorphism, without major restrictions being imposed by the persistence framework.

Although an entity looks like a simple Java object, it is actually more than that: It is a Java object with a mapping to a relational database. Therefore you have to declare an entity explicitly in order to distinguish it from other regular Java objects that you might use within your application, and you need a way to define how exactly to map the entity onto your existing database table(s). Since such kind of information can be considered metadata for the entity class, it is not surprising that JPA relies on the Java SE 5 standard technique for providing metadata: annotations.

Note that JPA allows also providing the object-relational mapping information in XML format. However, annotations are easier to understand for a human reader, so we decided to use annotations within this (and any further) article. If you are not yet familiar with Java SE 5 annotations, refer to the resources collected in section *Additional Resources*.

Creating an Entity

The sample application that we are going to create in this article is based on a simple employee data model and uses only one entity. In order to create the entity, let's start with a simple Java class `Employee`:

```
public class Employee {

    private int id;
    private String name;

    public Employee() {
    }

    public Employee(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

As you see, it is a regular Java class without special characteristics. The class defines two fields, `id` and `name`, and provides the corresponding getter and setter methods according to the JavaBeans naming conventions. The only point worth mentioning is that JPA requires you to provide a public constructor without arguments. Other constructors may be added, though, as you can see in the example.

To convert this class into a valid entity, we simply have to add two annotations from the `javax.persistence` package:

1. `@Entity` – to declare it as an entity.
2. `@Id` – to declare the unique identifier, corresponding to the primary key in the database.

The `@Entity` annotation is placed at the class definition. The `@Id` annotation can be placed either at the definition of the field that serves as the unique identifier for the entity or, alternatively, at the corresponding getter method. By deciding for a location for the `@Id` annotation, you also choose the mechanism by which the JPA implementation accesses the entity. If you place the `@Id` annotation at a field, the JPA implementation will directly read data from and write data to the fields of an entity instance (*field based access*). If you place the `@Id` annotation at a getter method, however, the JPA implementation will rather work with the getter and setter methods, ignoring any instance variables that might be available (*property based access*).

In our example, the field `id` serves as the unique identifier and we decide for field based access, so that the entity finally looks as follows:

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Employee {

    @Id
    private int id;
    private String name;

    [...]
}
```

Defining the Object-Relational Mapping

When developing a JPA application, you will often consider entities just regular Java objects that you instantiate, fill with data, send to other application components, and otherwise manipulate by Java means. The relational aspect of the entity, the fact that it also resides as a row in a database table and the problem of accessing it there, is most of the time hidden by the JPA framework. When designing the entity class itself, however, you indeed have to think about its representation in the database. You have to tell the JPA framework how exactly to map the entity to the relational database, that is, you have to define the *object-relational mapping*. In contrast to the situation with CMP 2.1 or JDO 1.0, the object-relational mapping is an integral part of the JPA specification and therefore standardized.

As an attentive reader, you might wonder about our sample entity `Employee` – we have just stated that it's finished, without caring for object-relational mapping so far. The point is that JPA makes the application developer's job as easy as possible by defining a set of very reasonable default mappings. If nothing else is stated, JPA makes assumptions on the names of tables and columns, for example. Only if they don't match your situation you have to add an appropriate declaration. The following table summarizes the most important default rules:

Area	Default Rule	Override by
Table name	An entity is mapped onto a table having the same name as the entity, i.e. usually the unqualified name of the entity class (see also comment below).	@Table
Persistent fields	All fields (with field-based access) or properties (with property-based access) are persistent.	@Transient
Column name	A persistent field or property is mapped onto a column of the same name (see also comment below).	@Column

According to these rules, the sample entity `Employee` is mapped onto a table named `EMPLOYEE`. Both fields `id` and `name` are persistent fields and they are mapped onto columns called `ID` and `NAME`, respectively.

Case Sensitivity of Table and Column Names

Unfortunately, the JPA specification does not define whether table and column names are case-sensitive. With SAP JPA 1.0, the following applies:

If a table name or column name is specified explicitly, its case is respected. If the table name or column name is generated according to a default rule, upper case is assumed.

Defining the Persistence Unit

The data model of a JPA application typically consists of several related entity classes, which have to be mapped together to the same database. The entity classes form a logical unit that is called a *persistence unit*. Within a JPA application, you have to define the persistence unit by a configuration file called `persistence.xml`. It is possible but not required to list the entities explicitly that form the persistence unit. If you don't do so, the JPA implementation scans the application for entities and detects them automatically. For the sample application, we configure the persistence unit in the simplest possible way, so the `persistence.xml` file looks as follows:

```
<persistence version="1.0"
        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">

    <persistence-unit name="JPAModel">
        <jta-data-source>JPA_EXAMPLE_01</jta-data-source>
    </persistence-unit>
</persistence>
```

The name of the persistence unit specified in the `persistence-unit` tag can be chosen arbitrarily. Just remember it, as we will refer to the persistence unit name later when working with the entity manager.

The `persistence.xml` file is also the place where you define any global settings for the persistence unit as a whole, such as, for example, the name of the data source that is used to connect to the database.

Always Specify a Data Source in persistence.xml

Technically, specifying the name of the data source related to the persistence unit is optional in the persistence.xml file. If the name of the data source is omitted, some providers automatically use a built-in default data source, others assume a default name but don't provide the data-source itself, while others again might even reject the persistence unit as being incomplete. For this reason – and for the sake of clarity – it is highly recommended to always specify the name of the data source to be used.

Since we develop a standard Java EE application, we use the Java Transaction API (JTA) for controlling transactions. Correspondingly, we declare a data source named `JPA_EXAMPLE_01` in the `jta-data-source` tag. You can choose any name here, but make sure that the specified data source does exist in the system on deployment of the application.

It is common practice not to refer to the physical name of a data source directly, but to work with a so-called data source alias, i.e. a logical name for a data source that is declared once within an application and then used (inside the application) wherever needed. For this purpose, our application contains a file data-source-aliases.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE data-source-aliases SYSTEM 'data-source-aliases.dtd'>
<data-source-aliases>
    <aliases>
        <data-source-name>JPA_SDN_EXAMPLE_DS</data-source-name>
        <alias>JPA_EXAMPLE_01</alias>
    </aliases>
</data-source-aliases>
```

As you see, the physical name of the data source we refer to is `JPA_SDN_EXAMPLE_DS`. Note that you have to create this data source before deploying the sample application. But don't worry: All you have to do is to deploy yet another small application that we provide for you.

Working with Entities – The Entity Manager

Since entities are regular Java objects, you can work with them in the same way you usually do. Instantiate an entity, read and modify its data, do whatever you like – but don't expect any of those operations to be reflected in the database automatically. If you want the JPA framework to manage a particular entity instance, you have to put it explicitly under the control of a JPA component called *entity manager*. As long as the entity manager controls the entity, you can expect that changes to the entity will be synchronized with the database. Once this control ends, however, the entity is again nothing but a regular Java object.

You interact with the entity manager via the Java interface `javax.persistence.EntityManager`. We will discuss the `EntityManager` interface in detail in a following article, so let's focus on the basic concepts of working with the entity manager. We will illustrate the mechanisms to create and persist a new entity, to find an existing entity by its unique identifier, and to remove an entity.

Persisting a New Entity

To put a new entity under the management of the entity manager and to schedule it to be inserted into the database, use the `persist` method defined by the `EntityManager` interface:

```
public void persist(Object entity);
```

The following code example illustrates how to use the persist method within an application:

```
EntityManager em;

// set up a new entity instance
Employee emp = new Employee(10);
emp.setName("Miller");

// put it under the management of the entity manager
em.persist(emp);
```

For the moment, just assume that the variable `em` has already been initialized with an instance of type `EntityManager`. The first step is to create a new Java object of type `Employee` in the usual way: Call a constructor to obtain a new instance and fill it with data using its setter methods. Up to that time, the JPA runtime, namely the entity manager, is unaware of the existence of the new entity. This changes only when you pass the entity to the entity manager using the `persist` method. After calling `persist`, the entity is managed, that is, controlled by the entity manager, and you can be sure that it will eventually be inserted into the database.

Finding an Entity by Its Unique Identifier

To find an existing entity with a given identifier or primary key in the database and to put it under the management of the entity manager, use the `find` method defined by the `EntityManager` interface:

```
public <T> T find(Class<T> entityClass, Object primaryKey);
```

As you see, the `find` method requires two arguments. The first one is the class object of the entity you want to find, that is, the entity class itself. The second argument is the object representation of the entity's unique identifier (corresponding to the primary key in the database). If you are not yet familiar with the Java language elements introduced with Java SE 5, you might be irritated by the return type declaration, but in fact it's easy: The method is parameterized to return the same class as you pass as its first argument.

Have a look at the code example to see how the `find` method is used in practice:

```
EntityManager em;

Employee emp = em.find(Employee.class, Integer.valueOf(10));
```

The code sample shows how to find the entity of type `Employee` that has the primary key 10. Assume that `em` has already been initialized. We have to call the `find` method with two arguments: the entity class, here

`Employee.class`, and the object representation of the entity's unique identifier. The `find` method returns the found entity instance (or `null` if there is no such entity in the database). We don't have to cast the result when assigning it to the variable `emp`. The `find` method is parameterized to return whatever we pass as a first argument, so here it will automatically return an instance of type `Employee`.

Removing an Entity

To schedule an entity to be deleted from the database, use the `remove` method defined by the `EntityManager` interface:

```
public void remove(Object entity);
```

The entity that you pass as an argument to the `remove` method must be a managed entity instance, i.e. the entity must already be under the control of the entity manager. Note that the entity returned by the `find` method is automatically managed by the entity manger. See the box *The Role of Transactions* for more information.

Again we demonstrate the usage of the `remove` method by means of a code example:

```
EntityManager em;

// retrieve a managed entity instance
Employee emp = em.find(Employee.class, Integer.valueOf(10));

if (emp != null) {
    // schedule the entity for removal
    em.remove(emp);
}
```

Assume that `em` has already been initialized. To remove an entity, we have to pass it as an argument to the `remove` method, so the first task is to retrieve the entity. Its identifier is given, so we use the `find` method for that purpose. With the entity at hand, which is automatically managed, we can call the `remove` method. The entity manager then takes care that the entity will be removed from the database.

The Role of Transactions

A fundamental idea of JPA is a separation of tasks between the application and the JPA implementation. The application on the one hand deals with entities as Java objects, implements the business logic and is fully responsible for consistency and integrity of the object representation of entities. The JPA implementation on the other hand takes care for the persistent representation of an entity in the database and assures that the information contained in the entity objects is consistently reflected in the database. Whenever the JPA framework creates, updates or removes an entity, it has to carry out these changes within a transaction. However, since a transaction defines a logical unit of work that is executed atomically, the scope of a transaction is closely related with the business logic. That's why it's the application that defines the scope of the transaction, that is, starts and ends the transaction, while the JPA framework just uses an existing transaction. JPA allows different transaction types to be used for transaction control, and you will learn more about these options in a subsequent article. Within a Java EE application, typically the Java Transaction API (JTA) is used. This is also the standard behavior if nothing else is specified in the `persistence.xml` *file.*

In its default configuration, the entity manager is closely related with the transaction. The entity manager keeps track of all entities it comes in contact with while the transaction is active. Whether you find an entity using the `find` *method, call* `persist` *with a new entity or access entities via other mechanisms like queries or navigation that we will introduce in later articles – the entity manager will add all those entities to its internal bookkeeping. As long as the transaction remains active, the entity manager controls the entities and tracks changes on them. When appropriate, but at the latest just before commit, the entity manager writes the changes to those entities to the database. With the commit of the transaction, the updates become irreversibly manifested in the database. At the same time, or rather immediately after, the entity manager "forgets" its entire state. All entities that have been managed before are then nothing but regular Java objects – we say they are detached.*

Once you have understood that the entity manager's internal management of entities is always synchronized with the transactions, it's clear how the API methods behave. The methods `persist` *and* `remove` *trigger database changes that require a transaction – consequently, the entity manager throws an exception if these methods are executed outside of a transaction. The* `find` *method simply reads an entity from the database and does not necessarily need a transaction. Nevertheless, the presence of a transaction influences the result: If a transaction is active, the entity returned by the find method is automatically managed by the entity manager. If there is no transaction, the entity manager simply returns the entity without further caring for it – the entity instance is detached.*

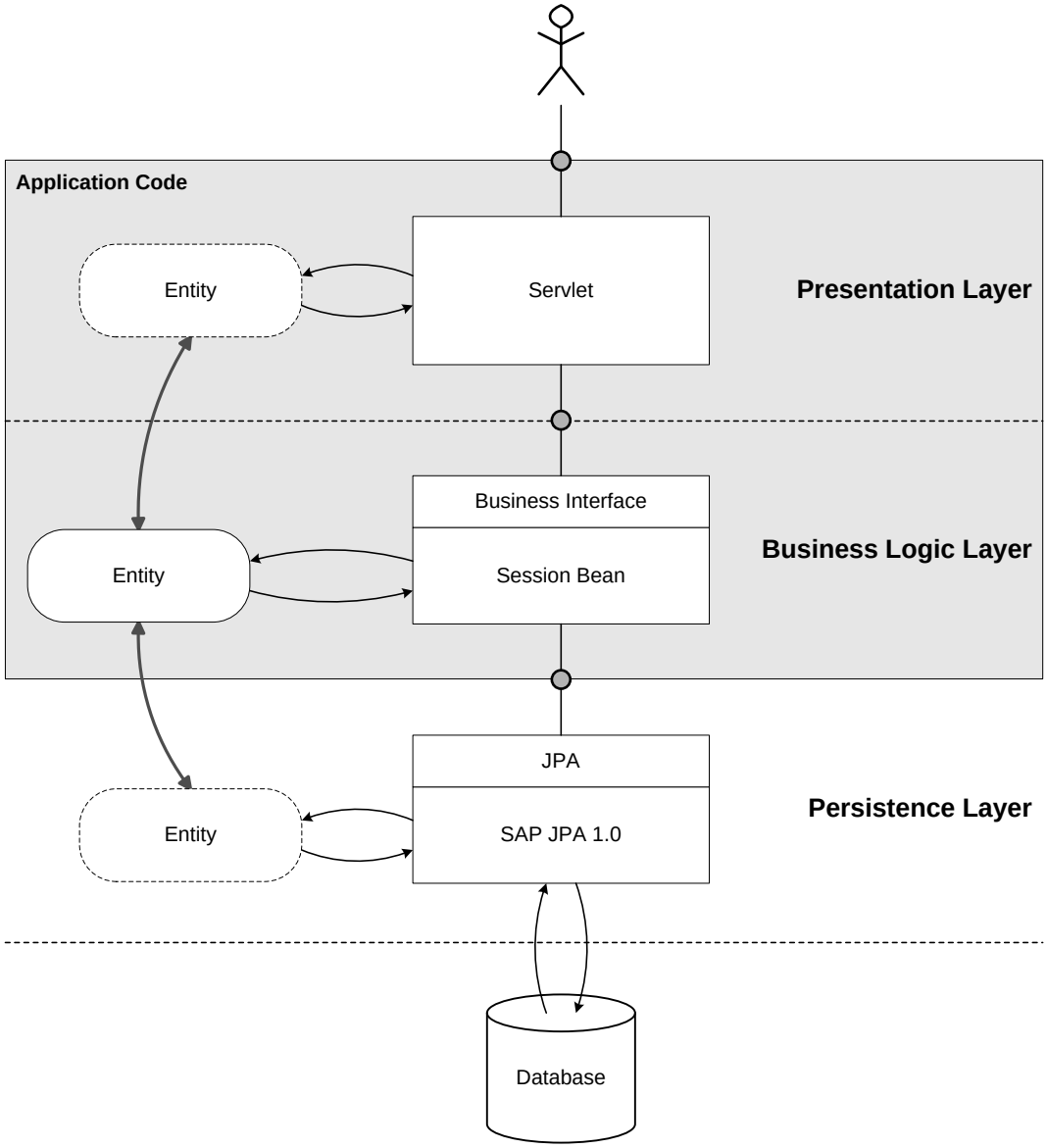

Figure 1: Architecture of the Sample Application

Building Your First JPA Application

By now we have gathered all the knowledge we need to create a first JPA application. The application is based on the simple employee data model that we used above and basically combines all the parts you have already seen: It allows you to create a new employee, to find an existing one by its unique identifier, and finally to remove an employee.

We designed the sample application to separate clearly between presentation layer and business logic layer. That makes it slightly more complex than absolutely necessary, but it also reflects the typical architecture of enterprise applications. Still we tried to keep the application simple – especially in the frontend part – in order not to distract from the main point: how to use JPA within a Java EE application. So we decided to build the web layer as a servlet that combines both presentation and navigation. In a real-life application you would rather choose a more powerful technique, such as WebDynpro or Java Server Faces, for example. The business logic is implemented based on Enterprise JavaBeans and consists of a single stateless session bean. The business methods of the session bean are exposed using a business interface, so the servlet accesses the bean only via its interface. As the session bean provides the business logic, this is where JPA comes into play. The business data, finally, is modeled as a JPA entity. Note that we use the same entity instance in the business logic layer as well as in the presentation layer. Since an entity instance is nothing but a regular Java object unless managed by an entity manager, we can use the entity itself to transport the data to the servlet. A dedicated data transfer object is not required. Figure 1 summarizes the architecture of the sample application.

Note that our sample application makes use of Java EE 5 features such as resource injection and uses programming models such as EJB 3.0. If you want to learn more about these concepts, refer to the resources listed in section *Additional Resources*.

The Business Data – Creating the Entity

Our application is based on the simple employee data model, consisting of the single entity `Employee` only, that we have introduced in section *Creating an Entity*. See there for the code example.

JPA does not automatically create the database table onto which the entity `Employee` is mapped. Although most design time tools for JPA and several JPA vendors allow generating database tables or table definition scripts automatically based the entity definition, this feature is not intrinsic to a JPA implementation, and it is rather intended for rapid prototyping. Generally, JPA assumes that the database tables corresponding to your entity's object-relational mapping definitions are present in the database. Most applications will therefore use a mechanism to deliver predefined database tables together with the application itself. Our example uses an SQL script for that purpose, which can be executed using SAP NetWeaver Developer Studio. Section *Related Content* points to a document that explains the steps in detail that are necessary to deploy and run the sample application.

The Business Logic Layer – Creating the Session Bean

It is a typical design pattern for Java EE applications to encapsulate their business logic within session beans and to expose the functionality to the presentation layer only via a well-defined business interface. Since the session bean thus acts as a façade towards the upper layers, the pattern is often referred to as *session façade* pattern.

In our case, the session bean encapsulates the access to the persistence API. We use a stateless session bean based on the EJB 3.0 programming model, which technically consists of a business interface and the session bean implementation. We are not going to give a thorough introduction into EJB 3.0 here, so if you want to learn more, refer to the resources listed in section *Additional Resources*.

Defining the Business Interface

In EJB 3.0, a business interface is a regular Java interface that exposes the functionality of the session bean as business methods to the upper layers. Our application needs to create, find and remove an employee, so the business interface provides appropriate methods for these tasks:

```
public interface EmployeeServiceLocal {
    public Employee createEmployee(int id, String name);
    public Employee findEmployee(int id);
    public void deleteEmployee(int id);
}
```

Note that the methods `createEmployee` and `findEmployee` return an object of type `Employee` – so the entity itself is used as a data container here. This is possible because the entity, unless managed by the entity manager, is nothing but a regular Java object. There is simply no need to copy its data into a dedicated data transfer object.

Implementing the Stateless Session Bean

A session bean in EJB 3.0 is much easier to implement than its counterpart in the older EJB versions. Basically, it's just a regular Java class implementing the business interface. A single annotation – `@Stateless` for a stateless session bean or `@Stateful` for a stateful session bean – turns the class into the implementation of a session bean.

The code sample below shows the implementation of the session bean in our sample application.

```
@Stateless
public class EmployeeServiceBean implements EmployeeServiceLocal {

    @PersistenceContext(unitName="JPAModel")
    private EntityManager em;

    public Employee createEmployee(int id, String name) {
        Employee emp = new Employee(id);
        emp.setName(name);
        em.persist(emp);
        return emp;
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Employee findEmployee(int id) {
        return em.find(Employee.class, id);
    }

    public void deleteEmployee(int id) {
        Employee emp = findEmployee(id);
        if (emp != null) {
            em.remove(emp);
        }
    }
}
```

The implementation of the business methods is straightforward: Aside from slight modifications in order to use parameters for id and name of the employee, you know the coding already from section *Working with Entities – The Entity Manager* where we demonstrated the basic usage of the entity manager interface. At that time, however, we ignored the problem of obtaining an entity manager. Let's have a closer look at that issue now.

As in the code snippets before, the session bean simply declares a variable of type `EntityManager` without assigning a value to it. Two aspects are different, though: First, the code is part of a stateless session bean, so it will be executed inside an application server. Second, the variable is annotated with `@PersistenceContext(unitName="JPAModel")`. In fact, the session bean makes use of a technique called *resource injection*. The annotation `@PersistenceContext` instructs the application server to fill ("inject") an entity manager instance into the annotated variable (here: `em`). You are guaranteed that the variable is properly initialized when a business method of the session bean is called for the first time. The element `unitName` of the annotation `@PersistenceContext` specifies the name of the persistence unit on which the entity manager operates. That means the value provided there (`JPAModel`) must correspond to the name that we chose in the `persistence.xml` file.

Another annotation might catch your eye: The method `findEmployee` has been annotated with `@TransactionAttribute(TransactionAttributeType.SUPPORTS)`. It is a mechanism to declaratively define the scope of transactions. Although this annotation is specific for EJB 3.0 rather than JPA, we will shortly discuss it here. An EJB application usually delegates the task of controlling transactions completely to the application server. The application declares the desired behavior, but it's the application server that actually starts and commits (or rolls back) the transaction. The annotation `@TransactionAttribute` allows you to specify the transactional context individually for each business method. See the table below for a summary of the possible values.

The methods `createEmployee` and `removeEmployee` of the sample session bean are not annotated at all. Therefore, as you can see from the table below, they behave according to the transaction attribute `REQUIRED`: In case there is no transaction active when the business method is called, the application server will automatically start a new transaction, execute the business method, and commit the transaction right afterwards. If on the other hand a transaction is already available, that transaction will be used (and of course not committed, as it has not been started here). As both scenarios, creating and removing an employee, do require a transaction, but not necessarily an exclusive one, this is indeed the adequate transaction attribute. The business method `findEmployee`, however, does only read an employee without changing any data. It does not require a transaction, but it is tolerant if an active transaction is already present. Consequently, we overwrote the default settings with the transaction attribute `SUPPORTS`.

Transaction Attributes

EJB 3.0 defines the following transaction attributes:

Transaction Attribute	*Description*
*REQUIRED**	*Start a new transaction if necessary.*
MANDATORY	*Require the caller to have started a transaction.*
REQUIRES_NEW	*Suspend any active transaction and run in a newly started transaction.*
SUPPORTS	*Don't start a new transaction, but use one if it exists.*
NOT_SUPPORTED	*Suspend any active transaction and run without transaction.*
NEVER	*Forbid the caller to have started a transaction.*

** This is also the default if no transaction attribute is specified.*

The Presentation Layer

The presentation layer is almost not worth discussing here. In order not to distract from JPA, we keep the user interface as simple as possible and realize it as a plain servlet. Since we have encapsulated any JPA access within the session bean, the servlet does not contain any JPA specific coding. Refer to the application sources if you are interested in details.

There is just one issue that we should at least mention, because you might not yet be familiar with EJB 3.0: how to access the session bean, or rather its business interface, from the servlet. The mechanism is as easy as it could be – again we use resource injection for that purpose. We define a variable of type `EmployeeServiceLocal`, the business interface, within the servlet and annotate it with `@EJB`. The application server will automatically initialize the annotated variable appropriately. See the code sample below:

```
public class EmployeeServlet extends HttpServlet {
    [...]

    @EJB
    EmployeeServiceLocal service;

    [...]
}
```

Conclusion

This article has provided you with a practical understanding of the fundamentals of the Java Persistence API. We introduced you to the key concepts of JPA, and showed you how to make use of them to build an enterprise application.

The sample application hopefully demonstrated one thing: Using the Java Persistence API is really easy! Entities are just regular Java objects that you manipulate as usual. To make them persistent and to have them synchronized with the database, simply hand them over to the JPA entity manager.

The next step is up to you – try it yourself! Download the application and play around with it, or write your first JPA application from scratch. Get familiar with the programming model – and wait for our next articles that highlight more advanced aspects of the Java Persistence API. See you then!

Related Content

- The Sample Application JPAExample01App

  The archive contains the sample application we refer to in this article as projects that can be imported into SAP NetWeaver Developer Studio.

- How to Create, Deploy and Run the Sample Application JPAExample01App

  The document describes how to import the sample application into SAP NetWeaver Developer Studio, alternatively how to create the projects from scratch, and how to build, deploy and run the application.

- EHP1 for SAP NetWeaver Composition Environment 7.1 – Preview Version

  The download package includes: Java EE 5-compliant SAP NetWeaver Application Server, MaxDB 7.7 database, and SAP NetWeaver Developer Studio.

- Java Platform, Enterprise Edition 5 at SAP

  The site collects information on the Java EE 5 platform at SAP, such as, for example, documentation, articles, sample applications and the Java EE 5 forum.

Additional Resources

- Java Platform, Enterprise Edition (Java EE)

  http://java.sun.com/javaee

  The site is a good entry point to information on the Java EE platform as a whole as well as its technologies such as, for example, Enterprise JavaBeans.

- JDK 5.0 Documentation

  http://java.sun.com/j2se/1.5.0/docs/index.html

  The site provides the documentation on JDK 5.0. It covers besides other topics the new features such as annotations and generics.

Copyright