

Sponsored by:



This story appeared on JavaWorld at
<http://www.javaworld.com/javaworld/jw-07-2008/jw-07-orm-comparison.html>

iBatis, Hibernate, and JPA: Which is right for you?

Object-relational mapping solutions compared

By K. L. Nitin, Ananya S., Mahalakshmi K., and S. Sangeetha, JavaWorld.com, 07/15/08

Object-relational mapping in Java is a tricky business, and solutions like JDBC and entity beans have met with less than overwhelming enthusiasm. But a new generation of ORM solutions has since emerged. These tools allow for easier programming and a closer adherence to the ideals of object-oriented programming and multi-tiered architectural development. Learn how Hibernate, iBatis, and the Java Persistence API compare based on factors such as query-language support, performance, and portability across different relational databases.

In this article we introduce and compare two of the most popular open source persistence frameworks, [iBatis](#) and [Hibernate](#). We also discuss the [Java Persistence API](#) (JPA). We introduce each solution and discuss its defining qualities, as well as its individual strengths and weaknesses in broad application scenarios. We then compare iBatis, Hibernate, and JPA based on factors such as performance, portability, complexity, and adaptability to data model changes.

If you are a beginning Java programmer new to persistence concepts, reading this article will serve as a primer to the topic and to the most popular open source persistence solutions. If you are familiar with all three solutions and simply want a straightforward comparison, you will find it in the section "[Comparing persistence technologies](#)."

Understanding persistence

Persistence is an attribute of data that ensures that it is available even beyond the lifetime of an application. For an object-oriented language like Java, persistence ensures that the state of an object is accessible even after the application that created it has stopped executing.

There are different ways to achieve persistence. The traditional approach to the problem is to use file systems that store the necessary information in flat files. It is difficult to manage large amounts of data in this way because the data is spread across different files. Maintaining data consistency is also an issue with flat-file systems, because the same information may be replicated in various files. Searching for data in flat files is time-consuming, especially if those files are unsorted. Also, file systems provide limited support for concurrent access, as they do not ensure data integrity. For all these reasons, file systems are not considered a good data-storage solution when persistence is desired.

The most common approach today is to use databases that serve as repositories for large amounts of data. There are many types of databases: relational, hierarchical, network, object-oriented, and so on. These databases, along with their database management systems (DBMSs), not only provide a persistence facility, but also manage the information that is persisted. Relational databases are the mostly widely used type. Data in a relational database is modeled as a set of interrelated tables.

The advent of enterprise applications popularized the *n-tier architecture*, which aims to improve maintainability by separating presentation, business, and database-related code into different tiers (or *layers*) of the application. The layer that separates the business logic and the database code is the *persistence layer*, which keeps the application independent of the underlying database technology. With this robust layer in place, the developer no longer needs to take care of data persistence. The persistence layer encapsulates the way in which the data is stored and retrieved from a relational database.

Java applications traditionally used the JDBC (Java Database Connectivity) API to persist data into relational databases. The JDBC API uses SQL statements to perform create, read, update, and delete (CRUD) operations. JDBC code is embedded in Java classes -- in other words, it's tightly coupled to the business logic. This code also relies heavily on SQL, which is not standardized across databases; that makes migrating from one database to another difficult.

Relational database technology emphasizes data and its relationships, whereas the object-oriented paradigm used in Java concentrates not on the data itself, but on the operations performed on that data. Hence, when these two technologies are required to work together, there is a conflict of interests. Also, the object-oriented programming concepts of inheritance, polymorphism, and association are not addressed by relational databases. Another problem resulting from this mismatch arises when user-defined data types defined in a Java application are mapped to relational databases, as the latter do not provide the required type support.

Object-relational mapping

Object-relational mapping (ORM) has emerged as a solution to what is sometimes called the [object-relational impedance mismatch](#). ORM is a technique that transparently persists application objects to the tables in a relational database. ORM behaves like a virtual database, hiding the underlying database architecture from the user. ORM provides functionality to perform complete CRUD operations and encourages object-oriented querying. ORM also supports metadata mapping and helps in the transaction management of the application.

An example will help illustrate how ORM works. Consider a simple `Car` object that needs to be persisted in the database. The `Car` object in the domain model is the representation of the `CAR` table in the data model. The attributes of the `Car` object are derived from the columns of the `CAR` table. There is a direct mapping between the `Car` class and the `CAR` table, as illustrated in Figure 1.

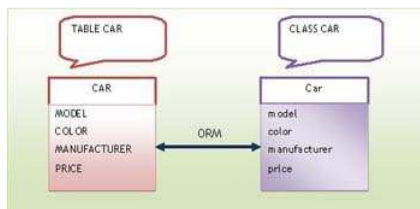


Figure 1. Mapping an object to a table

There are many open source ORM tools, including Hibernate, iBatis SQL Maps, and Java Ultra-Lite Persistence. Most of these tools are *persistence frameworks* that provide a layer of abstraction between the Java application and the database. A persistence framework maps the objects in the application domain to data that needs to be persisted in a database. The mappings can be defined using either XML files or metadata annotations (the latter introduced to the language as part of Java 1.5). The persistence framework aims to separate the database-related code and the application code (that is, the business logic), thereby increasing application flexibility. A persistence framework simplifies the development process by providing a wrapper around the persistence logic.

With this basic introduction to persistence out of the way, we're ready to move on to discussing two of the most popular open source persistence frameworks, iBatis and Hibernate. We'll also introduce the Java Persistence API and discuss the strengths and weaknesses of all three solutions in various application scenarios.

iBatis: Using SQL directly

Object-relational mapping (ORM) uses direct mapping to generate JDBC or SQL code under the hood. For some application scenarios, however, you will need more direct control over SQL queries. When writing an application that involves a series of update queries, it's more effective to write your own SQL queries than to rely on ORM-generated SQL. Also, ORM cannot be used when there is a mismatch between the object model and the data model. As we've mentioned, JDBC code was once the common solution to such problems, but it introduced a lot of database code within application code, making applications harder to maintain. A persistence layer is needed to decouple the application and the database.

The iBATIS Data Mapper framework helps solve these problems. iBATIS is a persistence framework that provides the benefits of SQL but avoids the complexity of JDBC. Unlike most other persistence frameworks, iBATIS encourages the direct use of SQL and ensures that all the benefits of SQL are not overridden by the framework itself.

Simplicity is iBATIS's greatest advantage, as it provides a simple mapping and API layer that can be used to build data-access code. In this framework the data model and the object model need not map to one another precisely. This is because iBATIS uses a *data mapper*, which maps objects to stored procedures, SQL statements, or `ResultSet`s via an XML descriptor, rather than a *metadata mapper*, which maps objects in the domain to tables in the database. Thus, iBATIS enables the data model and the object model to be independent of each other.

How iBATIS works

iBATIS allows loose coupling of the database and application by mapping the input to and output from the database to the domain objects, thus introducing an abstraction layer. The mapping is done using XML files that contain SQL queries. This loose coupling allows the mapping to work for systems where the application and the database design are mismatched. It also helps in dealing with legacy databases and with databases that change over time.

The iBATIS framework mainly uses the following two XML files as descriptors:

- SQLMapConfig.xml
- SQLMap.xml

We'll look at each file in detail.

SQLMapConfig.xml

SQLMapConfig.xml is a central XML file that contains all the configuration details, like the details for the data sources; it also optionally includes information about transaction management. This file identifies all the SQLMap.xml files -- of which there may be more than one -- and loads them.

Consider an `Employee` class that maps to an `EMPLOYEE` table in the database. The properties of the class -- `emp_id`, `emp_firstname`, and `emp_lastname` -- correspond to similarly named columns in the table. The class diagram for the `Employee` class is shown in Figure 2. (This class will be used to demonstrate the different persistence techniques that are discussed in this article.)



Figure 2. Class diagram for the `Employee` class

The SQLMapConfig.xml file for the `Employee` class can be written as shown in Listing 1.

Listing 1. SQLMapConfig.xml file for Employee

```

<sqlMapConfig>
  <transactionManager type="JDBC" commitRequired="false">
    <dataSource type="EMPLOYEE">
      <property name="JDBC.Driver" value="com.mysql.jdbc.Driver"/>
      <property name="JDBC.ConnectionURL" value="jdbc:mysql://localhost:3306/ibatis"/>
      <property name="JDBC.Username" value="root"/>
      <property name="JDBC.Password" value=""/>
    </dataSource>
  </transactionManager>
  <!-- List the SQL Map XML files. They can be loaded from the classpath, as they are here (com.mydomain.data...) -->
  <sqlMap resource="com/mydomain/data/Employee.xml"/>
</sqlMapConfig>
  
```

SQLMapConfig.xml uses a `transactionManager` tag to configure a data source to use with this particular SQL map. It specifies the type of the data source, along with some details, including information about the driver, the database URL, and the username and password. The `sqlMap` tag specifies the location of the SQLMap.xml file so as to load it.

SQLMap.xml

The other XML file is SQLMap.xml, which is in practice named after the table to which it relates. There can be any number of such files in a single application. This file is the place where domain objects are mapped to SQL statements. This descriptor uses parameter maps to map the inputs to the statements and the result maps for mapping SQL `ResultSet`s. This file also contains the queries. Therefore, to change the queries, you need to change the XML, not your application's Java code. The mapping is done by using the actual SQL statements that will interact with the database. Thus, using SQL provides greater flexibility to the developer and makes iBATIS easy to understand to anyone with SQL programming experience.

The SQLMap.xml file that defines the SQL statements to perform CRUD operations on the `EMPLOYEE` table is shown in Listing 2.

Listing 2. SQLMap.xml for operations on EMPLOYEE

```

<sqlMap namespace="Employee">
  <typeAlias alias="Employee" type="com.sample.Employee"/>
  <resultMap id="EmpResult" class="Employee">
    <result property="id" column="emp_id"/>
    <result property="firstName" column="emp_firstname"/>
    <result property="lastName" column="emp_lastname"/>
  </resultMap>
  <!-- Select all data from the table using the result map for Employee class.-->
  <select id="selectAllEmps" resultMap="EmpResult">
    select * from EMPLOYEE
  </select>
  <!-- Select the data from the table based on the id. -->
  <select id="selectEmpById" parameterClass="int" resultClass="Employee">
    <select emp_id as id,emp_firstname as firstName, emp_lastname as lastName from EMPLOYEE where emp_id= #id#>
  </select>
  <!-- insert the data into the table -->
  <insert id="insertEmp" parameterClass="Employee">
    insert into EMPLOYEE (
      emp_id,
      emp_firstname,
      emp_lastname)
    values (
      #id#, #firstName# , #lastName# )
  </insert>
  <!-- update the Employee record based on the id -->
  <update id="updateEmp" parameterClass="Employee">
    update EMPLOYEE set
      emp_firstname = #firstName#,
      emp_lastname = #lastName#
    where
      emp_id = #id#
  </update>
  <!-- delete the Employee record based on the given id -->
  <delete id="deleteEmp" parameterClass="int">
    delete from EMPLOYEE where emp_id = #id#
  </delete>
</sqlMap>
  
```

In Listing 2, the `typeAlias` tag is used to represent type aliases, so you can avoid typing the full class name every time it would otherwise appear. It contains the `resultMap` tag, which describes the mapping between the columns returned from a query and the properties of the class represented by the `Employee` class. The `resultMap` is optional and it isn't required if the columns in the table (or aliases) match the properties of the class exactly. This `resultMap` tag is followed by a series of queries. `SQLMap.xml` can contain any number of queries. All the select, insert, update, and delete statements are written within their respective tags. Every statement is named using the `id` attribute.

The output from a select query can be mapped to a `resultMap` or to a result class that is a `JavaBean`. The aliases in the queries should match the properties of the target result class (that is, the `JavaBean`). The `parameterClass` attribute is used to specify the `JavaBean` whose properties are the inputs. Any parameters in the hash symbol are the properties of the `JavaBean`.

Loading the descriptor files to your Java application

After you have completed the entire configuration and mapped in both the XML files, `SQLMapConfig.xml` needs to be loaded by the Java application. The first step is to load the `SQLMap.xml` configuration file that was created earlier. To do this, you would use the `com.ibatis.common.resources.Resources` class, which is included with the `iBATIS` framework, as shown in Listing 3.

Listing 3. Loading SQLMap.xml

```
private static SqlMapClient sqlMapper;
...
try {
    Reader reader = Resources.getResourceAsReader("com/mydomain/data/SqlMapConfig.xml");
    sqlMapper = SqlMapClientBuilder.buildSqlMapClient(reader);
    reader.close();
} catch (IOException e) {
    // Fail fast.
    throw new RuntimeException("Something bad happened while building the SqlMapClient instance." + e, e);
}
```

The `SqlMapClient` class is used for working with `SQLMaps`. It allows you to run mapped statements like select, insert, update, and so on. The `SqlMapClient` object is thread safe; hence, one object is enough. This makes it a good candidate to be a static member. This object is created by reading a single `SQLMapConfig.xml` file. The `iBATIS` framework provides the `Resources.getResourceAsReader()` utility, with which you can read the `SQLMapConfig.xml` file. Thus, by using this instance of the `SQLMap`, you can access an object from the database -- in this case, an `Employee` object.

To invoke the operations on the `EMPLOYEE` table, different methods are provided on the `SQLMap`, such as `queryForList()`, `queryForObject()`, `insert()`, `update()`, and `queryForMap()`, among others. The `queryForList()` method, shown in Listing 4, returns a list of `Employee` objects.

Listing 4. queryForList()

```
sqlMapper.queryForList("selectAllEmps");
```

Similarly, you would use the `queryForObject()` method when only one row was returned as a result of the query. Both methods take the statement name as the parameter.

Corresponding methods are available for performing insert, update, and delete operations, as shown in Listing 5. These methods take both the statement name declared in the `SQLMap.xml` file and the `Employee` object as the input.

Listing 5. Insert, update, and delete operations

```
sqlMapper.insert("insertEmp", emp);
sqlMapper.update("updateEmp", emp);
sqlMapper.delete("deleteEmp", id);
```

In this way, Java objects are persisted using straight SQL statements in `iBATIS`.

When to use iBATIS

`iBATIS` is best used when you need complete control of the SQL. It is also useful when the SQL queries need to be fine-tuned. `iBATIS` should not be used when you have full control over both the application and the database design, because in such cases the application could be modified to suit the database, or vice versa. In such situations, you could build a fully object-relational application, and other ORM tools are preferable. As `iBATIS` is more SQL-centric, it is generally referred to as *inverted* -- fully ORM tools generate SQL, whereas `iBATIS` uses SQL directly. `iBATIS` is also inappropriate for non-relational databases, because such databases do not support transactions and other key features that `iBATIS` uses.

Hibernate:

Hibernate is an open source, lightweight object-relational mapping solution. The main feature of Hibernate is its support for object-based modeling, which allows it to provide a transparent mechanism for persistence. It uses XML to map a database to an application and supports fine-grained objects. The current version of Hibernate is 3.x, and it supports Java annotations and hence satisfies the EJB specification.

Hibernate includes a very powerful query language called *Hibernate Query Language*, or HQL. HQL is very similar to SQL, and also defines some additional conventions. HQL is completely object-oriented, enabling you to leverage the complete strength of the object-oriented pillars of inheritance, polymorphism, and association. HQL queries are case insensitive, except for the names of the Java classes and properties being used. HQL returns query results as objects that can be directly accessed and manipulated by the programmer. HQL also supports many advanced features of pagination and dynamic profiling that SQL has never supported. HQL does not require any explicit joins when working with multiple tables.

Hibernate in brief

<p>Hibernate was developed by a team headed by Gavin King. The development of Hibernate began in 2001 and the team was later acquired by JBoss, which now manages it. Hibernate was developed initially for Java; in 2005 a .Net version named NHibernate was introduced.</p>

Why do we need Hibernate?

Entity beans, which have traditionally been used for object-relational mapping, are very difficult to understand and hard to maintain. Hibernate makes object-relational mapping simple by mapping the metadata in an XML file that defines the table in the database that needs to be mapped to a particular class. In other persistence frameworks, you need to modify the application class to achieve object-relational mapping; this is not necessary in Hibernate.

With Hibernate, you needn't worry about database changes, as manual changes in the SQL script files are avoided. If you ever need to change the database your application uses, that can be easily accommodated by altering the `dialect` property in the configuration file. Hibernate gives you the complete power of SQL, something that was never offered by earlier commercial ORM frameworks. Hibernate also supports many databases, including MySQL, Oracle, Sybase, Derby, and PostgreSQL, and works well with plain old Java object (POJO)-based models, too.

Hibernate generates JDBC code based on the underlying database chosen, and so saves you the trouble of writing JDBC code. It also supports connection pooling. The APIs that are used by Hibernate are very simple and easy to learn. Developers with very little knowledge of SQL can make use of Hibernate, as it lessens the burden of writing SQL queries.

Hibernate architecture

Internally, Hibernate uses JDBC, which provides a layer of abstraction to the database, while it employs the Java Transaction API (JTA) and JNDI to integrate with other applications. The connection information that the Hibernate needs to interact with the database is provided by the JDBC connection pool, which has to be configured.

Hibernate's architecture consists mainly of two interfaces -- `Session` and `Transaction` -- along with the `Query` interface, which is in the persistence layer of the application. The classes that are defined in the business layer of the application interact through independent metadata with the Hibernate persistence layer, which in turn talks to the database layer using certain JDBC APIs. In addition, Hibernate uses other interfaces for configuration, mainly the aptly named `Configuration` class. Hibernate also makes use of callback interfaces and some optional interfaces for extending the mapping functionality. The overall Hibernate architecture is illustrated in Figure 3.

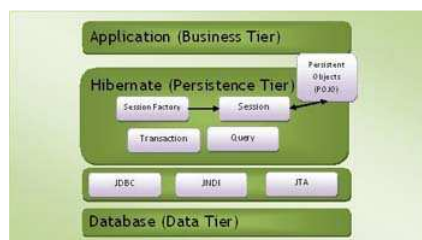


Figure 3. Hibernate architecture: The big picture

The major programming interfaces that are part of Hibernate are:

- `org.hibernate.SessionFactory` is basically used to obtain a session instance, and can be seen as an analogue to the connection pooling mechanism. This is thread safe, as all the application threads can use a single `SessionFactory` (as long as Hibernate uses a single database). This interface is configured through the configuration file, which determines the mapping file to be loaded.
- `org.hibernate.Session` provides a single thread that determines the conversation between the application and the database. This is analogous to a specific (single) connection. It is very lightweight and not thread safe.
- `org.hibernate.Transaction` provides a single-thread object that spans through the application and determines an atomic unit of work. It basically abstracts JDBC, JTA, and CORBA transactions.
- `org.hibernate.Query` is used to perform a query, either in HQL or in the SQL dialect of the underlying database. A `Query` instance is lightweight, and it is important to note that it cannot be used outside the session through which it was created.

Configuring Hibernate

You configure Hibernate through an XML file named `hibernate.cfg.xml`. The configuration file aids in establishing a connection to a particular relational database. The configuration file should know which mapping file it needs to refer to. At runtime, Hibernate reads the mapping file and then uses it to build a dynamic Java class corresponding to that table of the database. A sample configuration file is shown in Listing 6.

Listing 6. hibernate.cfg.xml

```
<hibernate-configuration>
  <session-factory>
    <!-- local connection properties -->
    <property name="hibernate.connection.url">
      jdbc:mysql://localhost/hibernateDemo
    </property>
    <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
    <property name="hibernate.connection.username">
      root
    </property>
    <property name="hibernate.connection.password">
      infosys
    </property>
    <!-- dialect for MySQL -->
    <property name="dialect">
      org.hibernate.dialect.MySQLDialect
    </property>
    <property name="hibernate.show_sql">false</property>
    <property name="hibernate.transaction.factory_class">
      org.hibernate.transaction.JDBCTransactionFactory
    </property>
    <mapping resource="Employee.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

Working with Hibernate

When a `SessionFactory` instance is created in the application, Hibernate reads the configuration file and identifies the respective mapping file. The session object that is created from the `SessionFactory` gets a particular connection to the database, and this session object is the persistence context for the instance of a persistence class. The instance can be in one of the three states: *transient*, *persistent*, or *detached*. In the transient state, the object is yet to be associated with a table; in the persistent state, the object is associated with the table; and in the detached state, there is no guarantee that the object is in sync with the table. The Hibernate code that is used to persist an `Employee` object is shown in Listing 7.

Listing 7. Persisting an object with Hibernate

```
Session session = null;
Transaction tx = null;

// At this point the Configuration file is read
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();

// A specific session object is obtained
session = sessionFactory.openSession();

// A new database transaction is started
tx = session.beginTransaction();

// Employee Object is created & populated
Employee emp = new Employee();
emp.setId(1);
emp.setEmpFirstname("K L");
emp.setEmpLastname("Nitin");

// Using the session, emp object is persisted in the database
session.save(emp);
```

The mapping file that the configuration file identifies maps a particular persistent class to the database table. It maps specific columns to specific fields, and has associations, collections, primary key mapping, and ID key generation mechanisms. The mapping files are generally given names based on the tables to which they map; in the example application, you'd use `Employee.hbm.xml` for the file that corresponds to the `EMPLOYEE` table. As you can see in Listing 8, the mapping file specifies that the `Employee` class has to be mapped to the `EMPLOYEE` table in the database, which has columns named `id`, `emp_firstname`, and `emp_lastname`. `id` is the primary key, and should have the value assigned.

Listing 8. Employee.hbm.xml

```
<hibernate-mapping package="demo">
  <class name="Employee" table="employee" >
    <meta attribute="sync-DAO">false</meta>
    <id name="Id" type="integer" column="emp_id">
      <generator class="assigned"/>
    </id>
  </class>
</hibernate-mapping>
```

```

</id>
<property name="EmpFirstname" column="emp_firstname"
          type="string" not-null="true" length="30" />
<property name="EmpLastname" column="emp_lastname"
          type="string" not-null="true" length="30" />
</class>
</hibernate-mapping>

```

When to use Hibernate

Hibernate is best used to leverage end-to-end OR mapping. It provides a complete ORM solution, but leaves you control over queries. Hibernate is an ideal solution for situations where you have complete control over both the application and the database design. In such cases you may modify the application to suit the database, or vice versa. In these cases you could use Hibernate to build a fully object-relational application. Hibernate is the best option for object-oriented programmers who are less familiar with SQL.

The Java Persistence API

The Java Persistence API is the standard object-relational mapping and persistence management interface for the Java EE 5 platform. As part of the EJB 3 specification effort, it is supported by all major Java vendors. The Java Persistence API draws on ideas from leading persistence frameworks and APIs, such as Hibernate, Oracle TopLink, Java Data Objects (JDO), and EJB container-managed persistence. JPA provides a platform on which specific implementations of persistence providers can be used. One of the main features of the Java Persistence API is that any persistence provider can be plugged in to it.

JPA is a POJO-based standard persistence model for ORM. It is part of the EJB 3 specification and replaces entity beans. The entity beans defined as part of the EJB 2.1 specification had failed to impress the industry as a complete persistence solution for several reasons:

- Entity beans are heavyweight components and are tightly coupled to a Java EE server. This makes them less suitable than lightweight POJOs, which are more desirable for their reusability.
- Entity beans are difficult to develop and deploy.
- BMP entity beans force you to use JDBC, while CMP entity beans are highly dependent on the Java EE server for their configuration and ORM declaration. These restrictions will affect the performance of the application.

To address these issues, the EJB 3 software expert group developed JPA as part of JSR 220. JPA borrows the best ideas from other persistence technologies. It defines a standard persistence model for all Java applications. JPA can be used as the persistence solution for both Java SE and Java EE applications.

JPA uses metadata annotations and/or XML descriptor files to configure the mapping between Java objects in the application domain and tables in the relational database. JPA is a complete ORM solution and supports inheritance and polymorphism. It also defines an SQL-like query language, JPQL (Java Persistence Query Language), which is different from EJB-QL (EJB Query Language), the language used by entity beans.

With JPA, you can plug in any persistence provider that implements the JPA specification instead of using whatever default persistence provider comes with your Java EE container. For example, the GlassFish server uses TopLink Essentials, provided by Oracle, as its default persistence provider. But you could choose to use Hibernate as the persistence provider instead by including all the necessary JAR files in your application.

Working with JPA

JPA uses many interfaces and annotation types defined in the `javax.persistence` package available with version 5 of Java EE. JPA uses entity classes that are mapped to tables in the database. These entity classes are defined using JPA annotations. Listing 9 shows the entity class named `Employee` that corresponds to the `EMPLOYEE` table in the sample application's database.

Listing 9. Employee entity class

```

@Entity
@Table(name = "employee")
@NamedQuery(name = "Employee.findByEmpId", query = "SELECT e FROM Employee e WHERE e.empId = :empId"), @NamedQuery(name = "Employee.findByEmp", query = "SELECT e FROM Employee e WHERE e.empId = :empId")
public class Employee implements Serializable {
    @Id
    @Column(name = "emp_id", nullable = false)
    private Integer empId;
    @Column(name = "emp_firstname", nullable = false)
    private String empFirstname;
    @Column(name = "emp_lastname", nullable = false)
    private String empLastname;

    public Employee() { }
    public Employee(Integer empId) {
        this.empId = empId;
    }
    public Employee(Integer empId, String empFirstname, String empLastname) {
        this.empId = empId;
        this.empFirstname = empFirstname;
        this.empLastname = empLastname;
    }
    public Integer getEmpId() {
        return empId;
    }
    public void setEmpId(Integer empId) {
        this.empId = empId;
    }
    public String getEmpFirstname() {
        return empFirstname;
    }
    public void setEmpFirstname(String empFirstname) {
        this.empFirstname = empFirstname;
    }
    public String getEmpLastname() {
        return empLastname;
    }
    public void setEmpLastname(String empLastname) {
        this.empLastname = empLastname;
    }
}
/****
*override equals, hashCode and toString methods
*using @Override annotation
*****/

```

The features of an entity class are as follows:

- The entity class is annotated using the `javax.persistence.Entity` annotation (`@Entity`).
- It must have a public or protected no-argument constructor, and may also contain other constructors.
- It cannot be declared `final`.
- Entity classes can extend from other entities and non-entity classes as well; the converse is also possible.
- They cannot have public instance variables. The class members should be exposed using only public getter and setter methods, following JavaBean style.
- Entity classes, being POJOs, generally need not implement any special interfaces. However, if they are to be passed as arguments over the network, then they must implement the `Serializable` interface.

Hibernate and JPA
Having just finished learning about how Hibernate can serve as a standalone persistence solution, you may be surprised to discover that it can also work with JPA. Strictly speaking, if you're going to use Hibernate by itself, you'll be using the <i>Hibernate Core</i> module, which generates SQL using HQL without the need for handling JDBC objects; the application is still independent of databases. Hibernate Core can be used with any application server, and for any generic Java application that needs to perform object-relational mapping. This mapping will be achieved by using native Hibernate APIs, the Hibernate Query Language, and XML mapping.
The Hibernate team was deeply involved in the development of the EJB 3 specification. After the introduction of EJB 3, a standalone implementation of EJB 3 persistence was made available as part of Hibernate -- Hibernate Annotations and Hibernate EntityManager. These two are built on top of Hibernate Core. For applications developed using Java EE 5 in which there is a need to use EJB 3, Hibernate EntityManager can be considered as an option for the persistence provider. Applications developed using Java EE 5 will utilize Hibernate and JPA working together.

The `javax.persistence.Table` annotation specifies the name of the table to which this entity instance is mapped. The class members can be Java primitive types, wrappers of Java primitives, enumerated types, or even other embeddable classes. The mapping to each column of the table is specified using the `javax.persistence.Column` annotation. This mapping can be used with persistent fields, in which case the entity uses persistent fields as well, or with getter/setter methods, in which case the entity uses persistent properties. However, the same convention must be followed for a particular entity class. Also, fields that are annotated using the `javax.persistence.Transient` annotation or marked `transient` will not be persisted into the database.

Each entity has a unique object identifier. This identifier is used to differentiate among different entity instances in the application domain; it corresponds to a primary key that is defined in the corresponding table. A primary key can be simple or composite. A simple primary key is denoted using the `javax.persistence.Id` annotation. Composite primary keys can be a single persistent property/field or a set of such fields/properties; they must be defined in a primary key class. Composite primary keys are denoted using the `javax.persistence.EmbeddedId` and `javax.persistence.IdClass` annotations. Any primary key class should implement the `hashCode()` and `equals()` methods.

The life cycle of JPA entities is managed by the entity manager, which is an instance of `javax.persistence.EntityManager`. Each such entity manager is associated with a persistence context. This context can be either propagated across all application components or managed by the application. The `EntityManager` can be created in the application using an `EntityManagerFactory`, as shown in Listing 10.

Listing 10. Creating an EntityManager

```
public class Main {
    private EntityManagerFactory emf;
    private EntityManager em;
    private String PERSISTENCE_UNIT_NAME = "EmployeePU";
    public static void main(String[] args) {
        try {
            Main main = new Main();
            main.initEntityManager();
            main.create();
            System.out.println("Employee successfully added");
            main.closeEntityManager();
        }
        catch(Exception ex) {
            System.out.println("Error in adding employee");
            ex.printStackTrace();
        }
    }
    private void initEntityManager() {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
        em = emf.createEntityManager();
    }
    private void closeEntityManager() {
        em.close(); emf.close();
    }
    private void create() {
        em.getTransaction().begin();
        Employee employee=new Employee(100);
        employee.setEmpFirstname("bob");
        employee.setEmpLastname("smith");
        em.persist(employee);
        em.getTransaction().commit();
    }
}
```

The `PERSISTENCE_UNIT_NAME` represents the name of the persistence unit that is used to create the `EntityManagerFactory`. The `EntityManagerFactory` can also be propagated across the application components using the `javax.persistence.PersistenceUnit` annotation.

In the `create()` method in Listing 10, a new employee record is being inserted into the `EMPLOYEE` table. The data represented by the entity instance is persisted into the database once the `EntityManagerTransaction` associated with `persist()` is completed. JPA also defines static and dynamic queries to retrieve the data from the database. Static queries are written using the `javax.persistence.NamedQuery` annotation, as shown in the `Employee` entity class. Dynamic queries are defined directly in the application using the `createQuery()` method of the `EntityManager`.

JPA uses a combination of annotation- and XML-based configuration. The XML file used for this purpose is `persistence.xml`, which is located in the application's `META-INF` directory. This file defines all the persistence units that are used by this application. Each persistence unit defines all the entity classes that are mapped to a single database. The `persistence.xml` file for the `Employee` application is shown in Listing 11.

Listing 11. persistence.xml

```
<persistence>
  <persistence-unit name="EmployeePU" transaction-type="RESOURCE_LOCAL">
    <provider>oracle.toplink.essentials.PersistenceProvider</provider>
    <class>com.trial.Employee</class>
    <properties>
      <property name="toplink.jdbc.url" value="jdbc:mysql://localhost:3306/projects"/>
      <property name="toplink.jdbc.user" value="root"/>
      <property name="toplink.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="toplink.jdbc.password" value="infosys"/>
    </properties>
  </persistence-unit>
</persistence>
```

The `persistence.xml` file defines a persistence unit named `EmployeePU`. The configuration for the corresponding database is also included in the persistence unit. An application can have multiple persistence units that relate to different databases.

To summarize, JPA provides a standard POJO-based ORM solution for both Java SE and Java EE applications. It uses entity classes, entity managers, and persistence units to map and persist the domain objects and the tables in the database.

When to use JPA

JPA should be used when you need a standard Java-based persistence solution. JPA supports inheritance and polymorphism, both features of object-oriented programming. The downside of JPA is that it requires a provider that implements it. These vendor-specific tools also provide certain other features that are not defined as part of the JPA specification. One such feature is support for caching, which is not clearly defined in JPA but is well supported by Hibernate, one of the most popular frameworks that implements JPA. Also, JPA is defined to work with relational databases only. If your persistence solution needs to be extended to other types of data stores, like XML databases, then JPA is not the answer to your persistence problem.

Comparing persistence technologies

You've now examined three different persistence mechanisms and their operations. Each of these frameworks has its own pros and cons. Let's consider several parameters that will help you decide the best possible option among them for your requirements.

Simplicity

In the development of many applications, time is a major constraint, especially when team members need to be trained to use a particular framework. In such a scenario, iBATIS is the best option. It is the simplest of the three frameworks, because it only requires knowledge of SQL.

Complete ORM solution

Traditional ORM solutions like Hibernate and JPA should be used to leverage complete object-relational mapping. Hibernate and JPA map Java objects directly to database tables, whereas

iBatis maps Java objects to the results of SQL queries. In some applications, the objects in the domain model are designed according to the business logic and might not completely map to the data model. In such a scenario, iBatis is the right choice.

Dependence on SQL

There has always been a demarcation between the people who are well versed in Java and those who are comfortable with SQL. For a proficient Java programmer who wants to use a persistence framework without much interaction with SQL, Hibernate is the best option, as it generates efficient SQL queries at runtime. However, if you want complete control over database querying using stored procedures, then iBatis is the recommended solution. JPA also supports SQL through the `createNativeQuery()` method of the `EntityManager`.

Support for query languages

iBatis strongly supports SQL, while Hibernate and JPA use their own query languages (HQL and JPQL, respectively), which are similar to SQL.

Performance

An application must perform well in order to succeed. Hibernate improves performance by providing caching facilities that help with faster retrieval of data from the database. iBatis uses SQL queries that can be fine-tuned for better performance. The performance of JPA depends on that of the vendor implementation. The choice is particular to each application.

Portability across different relational databases

Sometimes, you will need to change the relational database that your application uses. If you use Hibernate as your persistence solution, then this issue is easily resolved, as it uses a database dialect property in the configuration file. Porting from one database to another is simply a matter of changing the dialect property to the appropriate value. Hibernate uses this property as a guide to generate SQL code that is specific to the given database.

As previously mentioned, iBatis requires you to write your own SQL code; thus, an iBatis application's portability is dependent on that SQL. If the queries are written using portable SQL, then iBatis is also portable across different relational databases. On the other hand, the portability of JPA depends on the vendor implementation that is being used. JPA is portable across different implementations, like Hibernate and TopLink Essentials. So, if no vendor-specific features are used by the application, portability becomes a trivial issue.

Community support and documentation

Hibernate is a clear winner in this aspect. There are many Hibernate-focused forums where members actively respond to queries. iBatis and JPA are catching up slowly in this regard.

Portability across non-Java platforms

iBatis supports .Net and Ruby on Rails. Hibernate provides a persistence solution for .Net in the form of NHibernate. JPA, being a Java-specific API, obviously does not support any non-Java platform.

This comparison is summarized in Table 1.

Table 1. Persistence solutions compared

Features	iBatis	Hibernate	JPA
Simplicity	Best	Good	Good
Complete ORM solution	Average	Best	Best
Adaptability to data model changes	Good	Average	Average
Complexity	Best	Average	Average
Dependence on SQL	Good	Average	Average
Performance	Best	Best	N/A *
Portability across different relational databases	Average	Best	N/A *
Portability to non-Java platforms	Best	Good	Not Supported
Community support and documentation	Average	Good	Good

* The features supported by JPA are dependent on the persistence provider and the end result may vary accordingly.

Conclusion

iBatis, Hibernate, and JPA are three different mechanisms for persisting data in a relational database. Each has its own advantages and limitations. iBatis does not provide a complete ORM solution, and does not provide any direct mapping of objects and relational models. However, iBatis provides you with complete control over queries. Hibernate provides a complete ORM solution, but offers you no control over the queries. Hibernate is very popular and a large and active community provides support for new users. JPA also provides a complete ORM solution, and provides support for object-oriented programming features like inheritance and polymorphism, but its performance depends on the persistence provider.

The choice of a particular persistence mechanism is a matter of weighing all of the features discussed in the comparison section of this article. For most developers the decision will be made based on whether you require complete control over SQL for your application, need to auto-generate SQL, or just want an easy-to-program complete ORM solution.

Acknowledgements

The authors would like to sincerely acknowledge S. V. Subrahmanya (SVS) for his valuable guidance and support.

About the author

[S. Sangeetha](#) works as a technical architect at the E-Commerce Research Labs at Infosys Technologies. She has close to 10 years of experience in design and development of Java and Java EE applications. She has co-authored a book on Java EE architecture and also has written articles for JavaWorld and Java.net.

K. L. Nitin works at the E-Commerce Research Labs at Infosys Technologies. He is involved in the design and development of Java EE applications using Hibernate and JPA, and has expertise on agile frameworks like Ruby on Rails.

Ananya S. works at the E-Commerce Research Labs at Infosys Technologies. She has been involved in the design, development, and deployment of Java EE applications using JPA and iBatis. She also has experience in programming with Ruby and Ruby on Rails.

Mahalakshmi K. works at the E-Commerce Research Labs at Infosys Technologies. She has experience in Java EE technologies and database programming. She is involved in the design and development of Java EE applications using Hibernate and JPA. She has also worked on application development using the Ruby on Rails framework.

All contents copyright 1995-2009 Java World, Inc. <http://www.javaworld.com>