Hibernate

Understanding Object/Relational Persistence

Persistence:

In computer science, **persistence** refers to the characteristic of state that outlives the process that created it. This is achieved in practice by storing the state as data in computer data storage. Programs have to transfer data to and from storage devices and have to provide mappings from the native programming-language data structures to the storage device data structures.

In an object-oriented application, persistence allows an object to outlive the process that created it. The state of the object can be stored to disk, and an object with the same state can be re-created at some point in the future. This isn't limited to single objects—entire networks of interconnected objects can be made persistent and later re-created in a new process. Most objects aren't persistent; a **transient** object has a limited lifetime that is bounded by the life of the process that instantiated it. Almost all Java applications contain a mix of persistent and transient objects; hence, we need a subsystem that manages our persistent data.

Persistence layers and alternatives:

In a medium- or large-sized application, it usually makes sense to organize classes by concern. Persistence is one concern; others include presentation, workflow, and business logic. A typical object-oriented architecture includes layers of code that represent the concerns. It's normal and certainly best practice to group all classes and components responsible for persistence into a separate persistence layer in layered system architecture.

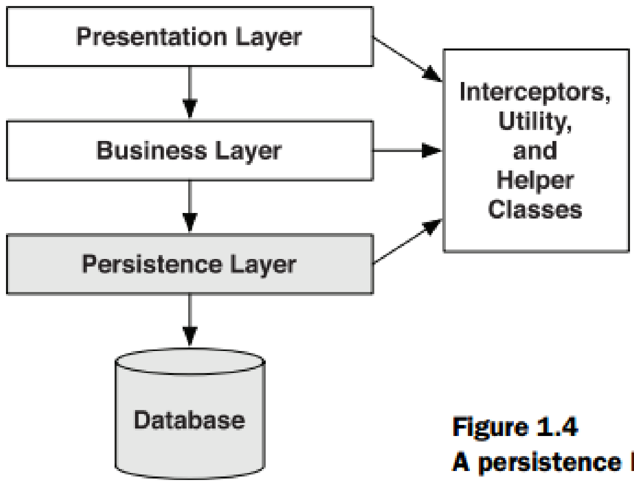

Figure 1.4
A persistence layer is the basis in a layered architecture

Different ways to implement Persistence Layer

1. Using Serialization
2. Object oriented database systems (OODBMS)
3. Using relational database systems (RDBMS – SQL/JDBC)

Because of the certain limitations of serialization and OODBMS and better flexibility of RDBMS, the most common approach to Java persistence is for application programmers to work directly with RDBMS using SQL and JDBC.

Persistence with RDBMS :

Java application developer when works with RDBMS (SQL database), the java code issues SQL statements to the database via the Java Database Connectivity (JDBC) API.

Though RDBMS are the only proven data management technology, and they're almost always a requirement in any Java project, data access tasks (e.g. bind arguments to prepare query parameters, execute the query, scroll through the query result table, retrieve values from the result set, and so on) using JDBC and SQL are often so tedious. Also there is a **paradigm mismatch** when object oriented program works with RDBMS using SQL.

The object/relational paradigm mismatch

Objects in object-oriented programming language (object model) and relations or tables (relational data model) in RDBMS are fundamentally different and hence there are certain problems because of this mismatch.

1. **Granularity:** Sometimes you will have an object model, which has more classes than the number of corresponding tables in the database.
2. **Inheritance:** RDBMSs do not define anything similar to Inheritance, which is a natural paradigm in object-oriented programming languages.
3. **Identity:** An RDBMS defines exactly one notion of 'sameness': the primary key. Java, however, defines both object identity (a==b) and object equality (a.equals(b)).
4. **Associations:** Object-oriented languages represent associations using object references whereas an RDBMS represents an association as a foreign key column.
5. **Navigation:** The ways you access objects in Java and in RDBMS are fundamentally different.

Solution to all these problems is **Object/Relational Mapping (ORM)**

Object/Relational Mapping (ORM)

In a nutshell, object/relational mapping is the automated (and transparent) persistence of objects in a Java application to the tables in a relational database, using metadata that describes the mapping between the objects and the database.

An ORM solution consists of the following four pieces:

1. An API for performing basic CRUD operations on objects of persistent classes
2. A language or API for specifying queries that refers to classes and properties of classes
3. A facility for specifying mapping metadata
4. A technique for the ORM implementation to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions

Advantages of ORM

1. Let's business code access objects rather than DB tables.
2. Hides details of SQL queries from OO logic.
3. No need to deal with the database implementation.
4. Entities based on business concepts rather than database structure.
5. Transaction management and automatic key generation.
6. Fast development of application.