**Introduction**

In software world there are two worlds:

1. One is the Java World, where everything is an object.
2. The other is the relational database world, where data is the king.

As a java developer we always work with objects that represent state and behavior modeling the real-world problem.

On the other hand, when it is time to store the data for a long time, we have to rely on relational databases, where the data is traditionally represented in a row-column format with relationships and associations.

Bringing java objects to the relational world is always a challenging and complex task for java developers. This process is often referred as to as **Object-Relational-Mapping (ORM).**

In real world if we want to store the state of our object for a long period of time, we need a durable storage spaces called databases. So, in spite of the so called “object-relational impedance mismatch”. Like:

* Inheritance is the fundamental object oriented Programming principle without which object association would be impossible to design. And Databases do not understand inheritance.
* When it comes to the rich set of object association like one-to-one, one-to-many, and many-to-many, databases fall flat, as they cannot support all types of relationships.
* Lastly, there is also an identity mismatch: objects carry both identity and equality, while database records are identified with their column valued.

And developer wants to minimize these differences by applying various frameworks and other technical solutions and strategies.

* Java has a standard tool set for accessing databases. It is called the Java Database Connectivity (JDBC) application programming interface (API). This API is very well suited for small projects, it becomes quite difficult to use (and sometimes out of hand) as the project start increasing in its complexity. And handling the object-relational model mapping is heavy-handed too. This was a pain for the programmers.

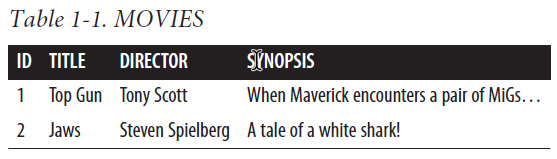
The Hibernate team found the gap in the ORM mapping space and took advantage by creating a simple framework that would make the developer’s life easy.

That’s when Hibernate Was Born. This is Open Source tool in the ORM tools domain.

**What kind of problem can hibernate solve?**

Let’s first understand the Uses of JDBC.

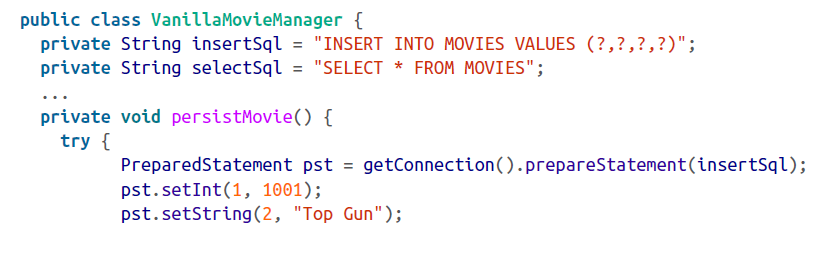
* The very first step in any database application is to establish a connection with the database. A connection is a gateway to the database for carrying out the operations on the data from a java application. JDBC provides a connection API to create connection based on the database properties that we provide. The database providers typically implement a class that holds the database connection mechanism- For Example, for MySQL database, “com.mysql.jdbc.Driver”

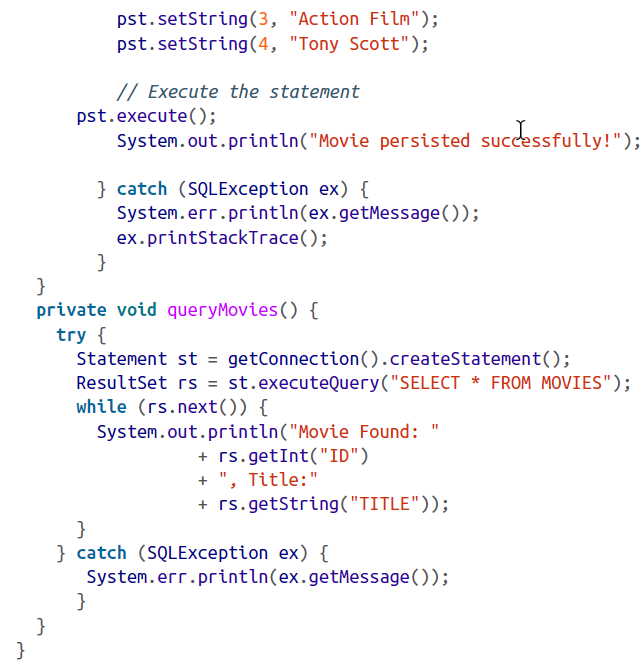


Example: 

We first instantiate the driver Class, and then get a connection using the DriverManager.

* Once we got a connection to the database successfully, our next step is to write a method to persist and query.

Example: 



This is what we’ve done:

1. We’ve created a PreparedStatement from the connection with the insert Sql string.
2. We’ve set the statement with the values (column values against column numbers: 1 is ID, 2 is title, etc).
3. We’ve executed the statement that should insert the movie in the table.
4. We’ve queried the database for all Movies and printed then out to the console.

**However,** there are few things to note:

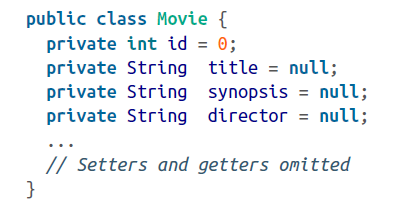
* We use predefined SQL statement to insert (or select) column values.
* We set the column values one by one using the position number (or column name).
* We catch the SQLException if the code misbehaves.

For simple programs, this way of creating the statement with the values and executing then if fine. However, in the real world the programs are much more complex. JDBC will work, if you are willing and able to write and manage a great deal of non-business code. Also, using JDBC might pose a challenge when you have a lot of tables or complex relationship between objects.

**Is there a better Solution?**

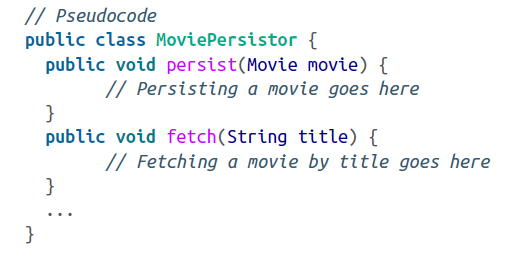
It would be ideal to call a method such as persist() on a utility class so that the Movie Object is persisted straightaway. After all, we are object-oriented programmers.

To Achieve this goal we’ll create a plain old java object (POJO) representing a movie: as shown below-



So, all we need now is for a facility to persist this POJO object into our database table MOVIES- in essence converting the object (Movie object) to a relational model.

Let’s create a MoviePersistor class that might do this job:



We haven’t written the persist or fetch functionality yet; that’s not the theme of the program. Now let’s see how easy to persist any Movie using the MoviePersistor utility class, as demonstrated in this sample test:

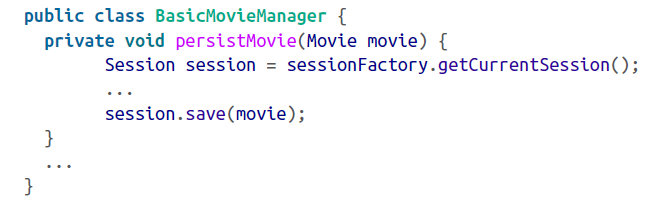


Wow…! A POJO representing a celluloid movies is persisted as a row of records into a database table- object model to relational model- via a utility class!

That’s all good , except for the actual *persist* and *fetch* method implementation. To implement this functionality, we not only need the connection facility to a database, we also need a mechanism to convert the object to a row (such as mapping our object properties to database columns). We could write our own framework of classes to hide the nitty-gritty of these conversations and persistence mechanisms (which may use good old JDBC statements behind the scenes). Although writing this framework isn’t rocket science, it would be a time-consuming and difficult effort to begin with.

Overtime , an organization’s persistence requirements may change or it may even migrate the database from Oracle to MySQL. This means the framework would have to be very generic and account for many of functional and technical requirements before hitting the ground. So such homegrown framework are unmanageable, inflexible, unscalable and sometime out of data too.

The Great News is: there is already a great framework that does the exactly this- Object persistence to a relational database- called Hibernate.

Now that we’ve got a persistence framework, let’s see how the same method that persists our movies cab be refactored through Hibernate: 

Did you noticed that we saved the Movie instance to a database by executing a single line of code: session.save(movie)?

This is what we wish for… A class that would simply save the persistent objects in an object-oriented way? Hibernate’s API classes expose several methods to manipulate the java objects with ease and comfort. We neither have to write lines of codes using JDBC nor fold up our sleeves and write a framework while scratching our heads and gulping gallons o caffine!

**Using Hibernate:**

The standard steps to follow in creating a Hibernate application are:

1. Configure the database connection.
2. Create mapping definitions.
3. Persist the classes.

Here are the common steps involved in developing the java-Hibernate version of our MovieManager application:

1. Create a Movie domain object (domain model POJOs representing data tables).
2. Create configuration files such as Hibernate properties and mapping files.
3. Creates a test client that manages (insert/update/delete/find) the Movies.

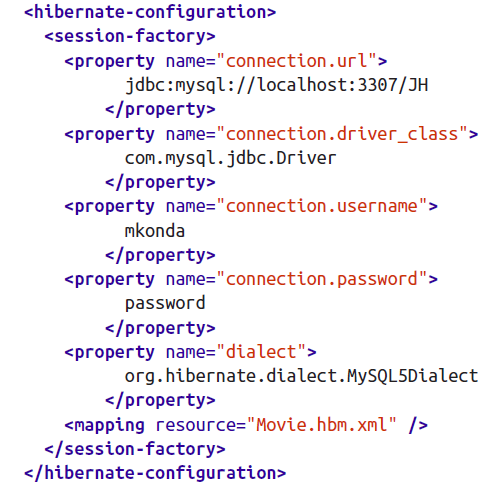
We have already prepared a Movie POJO.

**The heart of the hibernate application is its configuration.** There are two pieces of configuration required in any Hibernate application: one creates the database connections and the other creates the objects-to-table mapping. As with JDBC, we need to provide the database information to our application so it will open up a connection for manipulating the data. The mapping configuration defines which object properties are mapped to which columns of the table. (Details later)

**Configure the Database Connection.**

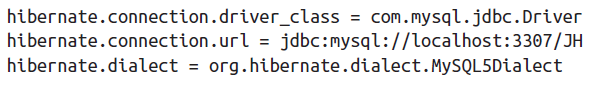
* To create a connection to the database, Hibernate must know the details of our database, tables, classes, and other mechanisms. This information is ideally provided as XML file (Usually named **hibernate.cfg.xml** or as a simple text file with name/value pairs (usually named hibernate.properties)
* For this exercise, we use XML style. We name this file hibernate.cfg.xml so the framework can load this file automatically.

The following snippet describes such a configuration file.



This file has enough information to get a live connection to a MySQL database.

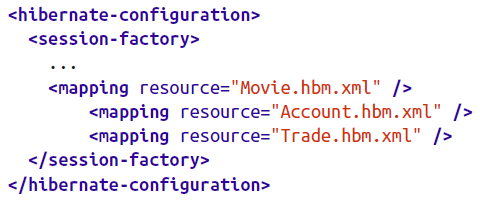
The preceding properties can also be expressed as name/value pairs. For Example, here’s the same information represented as name/value pairs in a text file titled hibernate.properties:



connection.url indicates the URL to which we should be connected,

driver\_class represents the relevant Driver class to make a connection, and the dialect indicates which database dialect we are using (MySQL, in this case).

Beyond providing the configuration properties we also have to provide mapping files and their locations. As mentioned earlier, a mapping file indicates the mapping of object properties to the row column values. This mapping is done in a separate file, usually suffixed with .hbm.xml. We must let Hibernate know our mapping definition files by including an element mapping property in the previous config file, as shown here:



The resource attribute indicates the name of the mapping resource that Hibernate should load.

**What does Hibernate do with this properties file?**

The Hibernate framework load this file to create a SessionFactory, which is a thread-safe global factory class for creating Sessions. We should ideally create a single SessionFactory and share it across the application. Note that a SessionFactory is defined for one, and only one , database. For instance, if you have another database alongside MySQL, you should define the relevant configuration in hibernate.hbm.xml to create separate *SessingFactory* for that Database too.

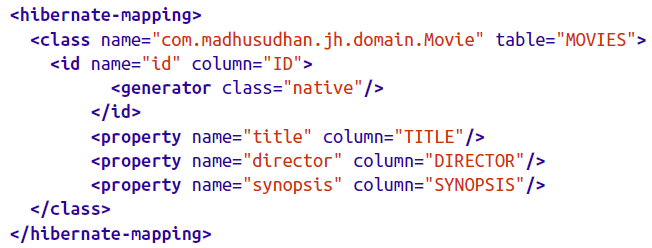
The goal of the *SessionFactory* is to create *Session* objects. SessionFactory is gateway to our database. It is the Session’s job to take care of all database operations such as saving, loading, and retrieving records from relevant tables.

The framework also maintain a transactional medium around our application. The operations involving the database access are wrapped up in a single unit of work called a ***transaction.*** So, all the operations in that transaction are either successful or rolled back.

Keep in mind that the configuration is used to create a *Session*  via *a SessionFactory*  instance. Before we move on, note that Session objects are not thread-safe and therefore should not be shared across different classes. We will see the details of how they should be used as we progress.

**Creating Mapping Definitions**

Once we have a connection configuration ready, the next step is to prepare the Movie.hbm.xml file consisting of object-table mapping definitions. The following XML snippet defines mapping for our Movie object against the MOVIES table.



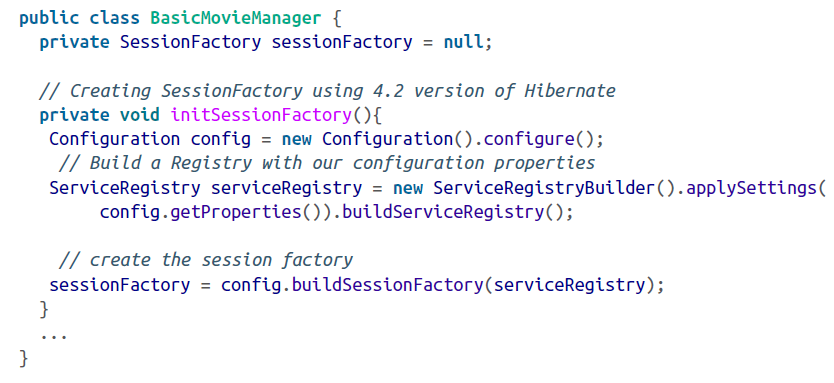
There is a lot going on in this mapping file. The hibernate-mapping element holds all the class-to-table mapping definitions. Individual mapping per object are declared under the class element. The *name* attribute of the *class* tag refers to our POJO domain class *com.madhusudhan.jh.domain.Movie,* while the table attribute refers to the table *movies*  to which the objects are persisted. The remaining properties indicate the mapping from the object’s variable to the table’s column (e.g the *id* mapped to *id,* the *title* to *title*, *director* to *director* , etc);

Each object must have a unique identifier—similar to a primary key on the table. We set this identifier by implementing the *id*  tag using a native strategy. Don’t pay too much attention to this *id* and the generation strategy yet. It will be discussed later.

**Persist the Objects**

Now that the configuration is out of our way, let’s create the persist objects with the help of Hibernate.

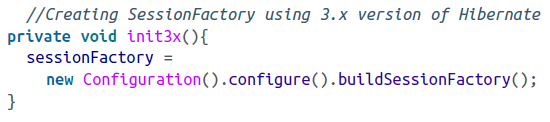
We need the SessionFactory instance from which we create a Session object. The following snippet shows the initial setup for creating the *SessionFactory class:*



Note that we don’t have to explicitly mention the mapping or configuration or properties files, because the Hibernate runtime looks for the default filename, such as hibernat.cfg.xml or hibernate.properties, in the classpath and loads them. If we have a non-default name, make sure you pass that as an argument—like configure(“my-hib-cfg.xml”), for example.

The Hibernate 4.x version introduced service registries, which we will see later.

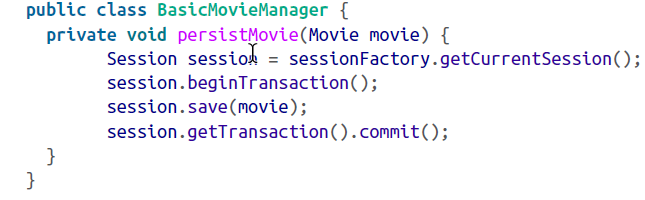
In 3.x versions, the configure method Search haphazardly through the classpath looking for a file name hibernate.cfg.xml (or hibernate.properties) to create a Configuration object. This configuration object is then used to create a SessionFactory instance. If you are using a pre 4.x version of Hibernate, use the following code to initialize the SessionFactory:



In 4.x versions, this is slightly modified by the introduction ServiceRegistry, which takes a Map of properties that can be fed from a Configuration object, as just shown.

**Creating the Persist Method**

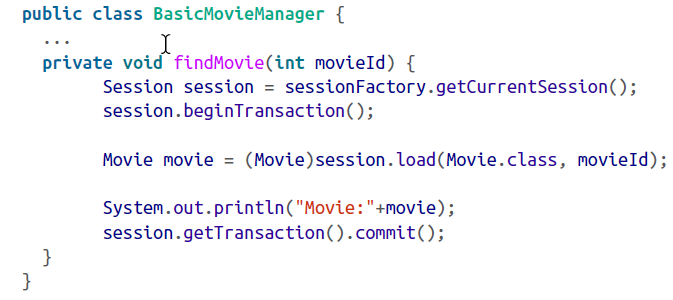
The persist method is defined on the class to persist a movie using Session’s save method. This is shown in the following snippet:



It looks simple, doesn’t it? We did not write unnecessary or repetitious code at all, but concentrated on the business of saving the object.

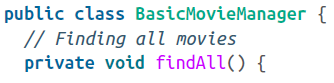
In the above snippet, we first grab a session from the factory. We then create a transaction object (we’ll learn more about transactions later) and persist the incoming Movie object using the session.save method. Finally, we commit the transaction, and the Movie is stored permanently in our database.

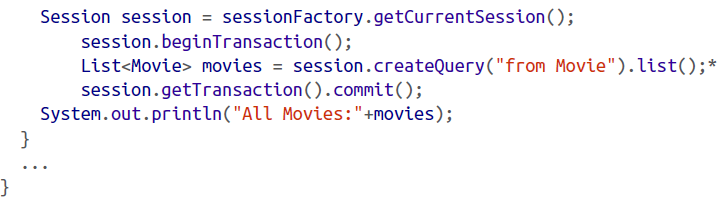
Alternatively, we can create another method in our test client, say findMovie. This method will use the Session’s load method to fetch the record from the database. We invoke the findMovie method, passing the movie ID as the argument, to fetch the movie:



The load method on the *Session*  API fetches the appropriate Movie object for a given identifier. If you are thinking that Hibernate may use a SELECT statement behind the scenes, you are correct!

You might wish to fetch all the movies from the table, you create a Query object with the simple string “from Movie” and execute it. The list method on the query (created via session.createQuery) returns a List of movies, as shown here:





**Autogeneration:**

AcroRd32_2017-01-08_19-00-22.png

When we set this property, the tables are automatically either created if they do not exist or updated if there’s a change in the table schema.

\*\*Warning\*\*

**Never use the hbm2ddl.auto property in production**! You must create a schema with all the table definitions and deploy to production via a proper release process.

That’s all: We wished for a mechanism that hides away the nitty-gritty of clumsy JDBC statements and connections. We dreamed of creating facility methods that would store a POJO object directly to the database without the hassle of setting/getting the columns. Thanks a lot to hibernate.

The problem with JDBC is that, it requires a lot of manual and unnecessarily repetitive code. We took a small step and introduced Hibernate to solve the problem of object-to-relational data persistence. From a high level, we took a look at the Hibernate concepts of SessionFactory and Sessions.