MSE Hibernate

Nathaniel Zimmerman

Jon Colabella

**Contents**

1. **Project Introduction………………………………………………………………...1**
2. **Project Goals…………………………………………………………………………1**
3. **Rejected Methods……………………………………………………………………..…1**
   1. **JDBC………………………………………………………………………………..2**
   2. **MyBatis……………………………………………………………………………..5**

**4. Hibernate…………………………………………………………………………………...7**

**5. Demo………………………………………………………………………………………..11**

**6. Additional Resources……………………………………………………………………12  
 7. Final Thoughts….………………………………………………………………………...21**

**8.** **Useful Links and Sources………………………………………………………………22**

**I.** **Project Introduction**

Third party products create a dependency on your developed and deployed programs. If the provider decides to change their interface or implementation, it may affect how your program behaves and drive other unwanted impacts to your product. These risks can be mitigated by following good object oriented practices such as encapsulation. The task is to research and explore methods of database storage and retrieval that remove the dependence from FOSS, COTS, and GOTS. Examples of this could include, object relational mapping (ORM) frameworks that auto-generate class objects based on schema definitions and data models. The class objects are used to store, retrieve, and convert data to and from the database into a specific programming language. Another approach could be to create database functions (stored procedures) that implement the SQL calls.

**II. Project Introduction**  
After researching different approaches and determining their impacts to existing legacy code, we have compiled this paper list of pros and cons to choose three methods to focus on. Once these three methods had been determined, we then chose the one that best fit the project goals in mind and created a prototype to display its strengths. The additional methods will not be prototyped, instead they will be surveyed as to why they are not as effective. Topics to evaluate will include ease of implementation, continuous maintenance, performance concerns, existing database support, ability to swap database engines, etc.

**III. Rejected Methods**

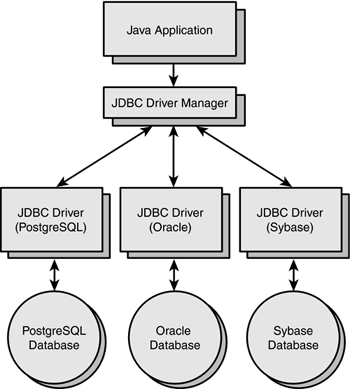
Our complete list of researched methods included: JDBC, ORMs as a whole, JPA, Hibernate, Spring, Avaje, and MyBatis. We created a master document and categorized each method based on their learning curve, performance, limitations, ease of implementation, and ease of database swapping.

**A. JDBC**

The three methods we chose to research were JDBC, MyBatis, and Hibernate. An API, a framework, and an ORM were chosen to give us a diverse look at each type of method to better choose our final implementation. Initially we looked Java Database Connectivity (JDBC) which is an application program interface (API) specification for connecting programs written in Java to the data in popular databases. We thought at first this was a lucrative option to pursue, it was free and open source, had a large community following for troubleshooting, and research into our five categories indicated it had high remarks in almost all of them.

With JDBC the application program interface lets you encode access request statements in SQL that are then passed to the program that manages the database. It returns the results through a similar interface. JDBC is very similar to the SQL Access Group's Open Database Connectivity (ODBC) and, with a small "bridge" program, you can use the JDBC interface to access databases through the ODBC interface. For example, you could write a program designed to access many popular database products on a number of operating system platforms. When accessing a database on a PC running Microsoft's Windows 2000 and, for example, a Microsoft Access database, your program with JDBC statements would be able to

access the Microsoft Access database.



Source: http://etutorials.org/shared/images/tutorials/tutorial\_47/13fig01.gif

We chose not to implement JDBC for a number or reasons. For a start, look at the common factor in the following statements: With JDBC, developer has to write code to map an object model's data representation to a relational data model and its corresponding database schema. With JDBC, the automatic mapping of Java objects with database tables and vice versa conversion is to be taken care of by the developer manually with lines of code. JDBC supports only native Structured Query Language (SQL). In each case the developer has to find out the efficient way to access database, i.e. to select effective query from a number of queries to perform same task. An application using JDBC to handle persistent data (database tables) having database specific code in large amount. The code written to map table data to application objects and vice versa is actually to map table fields to object properties. As table changed or database changed then it’s essential to change object structure as well as to change code written to map table-to-object/object-to-table. With JDBC, it is developer’s responsibility to handle JDBC result set and convert it to Java objects through code to use this persistent data in application. So with JDBC, mapping between Java objects and database tables is done manually. With JDBC, caching is maintained by hand-coding. In JDBC there is no check that always every user has updated data. This check has to be added by the developer. In each case the capabilities of JDBC are limited by the developer. The needs of our project require auto generation of code, as the sheer volume of the data being used makes it unrealistic to have developers code everything by hand.

In addition JDBC makes use of drivers. The drivers implement the defined interfaces in the JDBC API for interacting with your database server. For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java. The *Java.sql* package that ships with JDK contains various classes with their behaviours defined and their actual implementations are done in third-party drivers. Third party vendors implements the *java.sql.Driver* interface in their database driv. The Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database. When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.The JDBC-ODBC bridge that comes with JDK 1.2 is a good example of this kind of driver.

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database. These drivers typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge, the vendor-specific driver must be installed on each client machine. If we change the Database we have to change the native API as it is specific to a database and they are mostly obsolete now but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead. The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

In a Type 3 driver, a three-tier approach is used to accessing databases. The JDBC clients use standard network sockets to communicate with an middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server. This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases. You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type. Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

In a Type 4 driver, a pure Java-based driver that communicates directly with vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself. This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically. MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers. If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4. If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver. Type 2 drivers are useful in situations where a type 3 or type 4 driver is not available yet for your database. The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

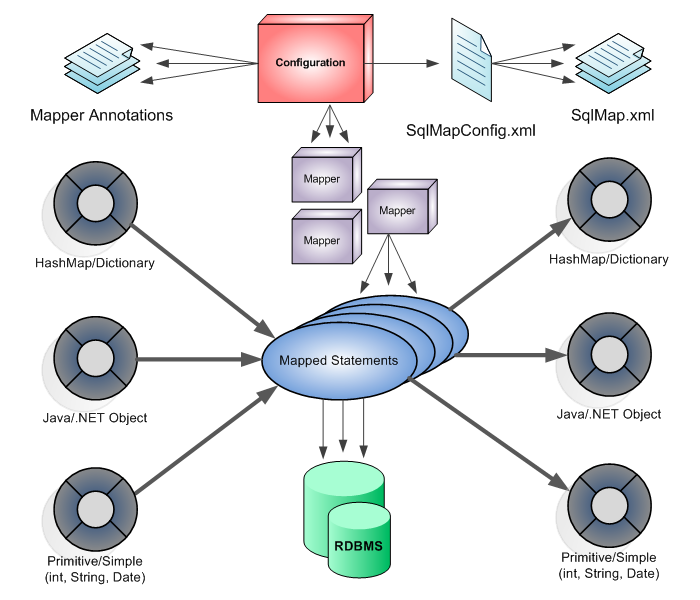
The use of drivers and the need to have multiple types on each individual machine, solidified our rejection of JDBC. Not counting the security hassle, installing and updating drivers on multiple workstations in multiple locations is a waste of time and resources that our client is not interested in investing.

B. MyBatis

MyBatis is a Java persistence framework that couples objects with stored procedures or SQL statements using an XML descriptor or annotations. MyBatis is free software that is distributed under the Apache License 2.0. We chose to take an in depth look at MyBatis to cover frameworks, as well as look at a smaller scale project to see if it met all of our requirements.

The true power of MyBatis is in the Mapped Statements. For all of their power, the Mapper XML files are relatively simple. Certainly if you were to compare them to the equivalent JDBC code, you would immediately see a savings of 95% of the code. The Mapper XML files have only a few first class elements:

* cache – Configuration of the cache for a given namespace.
* cache-ref – Reference to a cache configuration from another namespace.
* resultMap – The most complicated and powerful element that describes how to load your objects from the database result sets.
* sql – A reusable chunk of SQL that can be referenced by other statements.
* insert – A mapped INSERT statement.
* update – A mapped UPDATE statement.
* delete – A mapped DELETE statement.
* select – A mapped SELECT statement.

Source: https://javamagic.files.wordpress.com/2012/02/flow\_mybatis.png

MyBatis intrigued us initially because of its claim to reduce code. If we couldn’t have complete auto generation, then having significantly less code to make was the next best thing. However the issues with how Mybatis does this became apparent quickly. While it does indeed reduce the code needed SQL statements are deeply imbedded into the framework. This of course was an immediate deal breaker on using it in our implementation as we needed to move away from SQL statements entirely for our encapsulation needs.

In addition fundamentally the use of MyBatis is not what we want. In most cases when using a database you want to do one of two thing either to Create/Update/Delete some complex domain entities or to run analytic fetch queries (i.e. summation/aggregation queries). MyBatis is great for fetch queries (case 2) where you just want an answer. If you have a simple domain and just fetch information, use myBatis. If you have a complex domain and persist entities use something else. Again given the sheer volume of data being used by our client, it was safe to assume we would need to use every aspect of the databases. In addition as is apparent from how short this section is, MyBatis does not have a wealth of information available for use. Unlike JDBC there is little to no community available to help troubleshoot or even find basic information on how the whole method works.

While these methods were the most substantial in moving towards our final goal, we also researched a number of other methods, that while aren’t worth going into extensive detail about, are worth mentioning while they did not fit our project needs. Spring was one of these, the idea behind the framework was to keep code simple and without any cross cutting concerns, as well as multiple modules that could be attached to improve performance. However when put into practice most of the information we found about it seemed to say the opposite, one hobbyist claimed he had three thousand classes for a small hobby project. Other pitfalls included a specific framework that needed to be learned and poorly documented parts of the core and documentation that vary in quality over the major versions. Avaje and JPA were two other implementations we looked at.

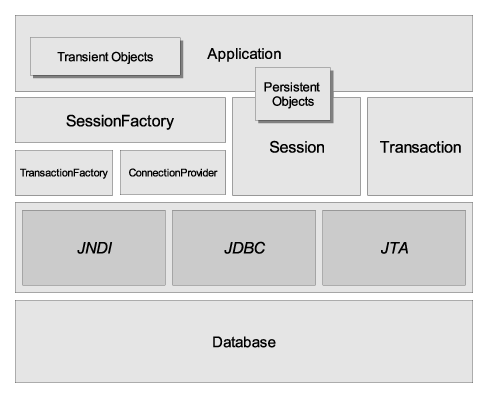
Avaje is an Object Relational Mapping Persistence Layer for Java, or JPA without a Container. Ebean as its also called , just like JPA has a Persistence Context but has some architectural differences to allow it to have a different approach to the entity bean lifecycle and removing the need to manage EntityManagers. Performance was the main reason we didn’t go with Avaje is performance. For example Clobs and Blobs (BLOBs (Binary Large Objects) and CLOBs (Character Large Objects)) are especially expensive and the ability to include/exclude clob/blob properties per use case/query can be significant in terms of performance.

Our next approach was looking at JPA which didn’t become our method of choice, but rather a stepping stone to reaching our final chosen methods. JPA represents the lowest common denominator between the various ORMs that support it. It hasn't really unified anything and it hasn't really fixed anything. In fact, a good case can be made that standardization wasn't even necessary. JPA is just an API (hence Java Persistence API) that requires an implementation to use. An analogy would be using JDBC. JDBC is an API for accessing databases, but you need an implementation (a driver jar file) to be able to connect to a database. On its own, without a driver, you cannot do anything with a database.With JPA you need an implementation, a set of classes that lie "below" JPA, and such an implementation will do what want.Your application uses the JPA API to communicates with the underlying implementation. Popular implementations we learned include Hibernate, EclipseLink, and OpenJPA. In addition to being a gateway to testing other methods of encapsulation, JPA also has numerous procedures that our end product would use such as entity classes and POJOs.

An entity is a lightweight persistence domain object. Typically an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes. The persistent state of an entity is represented either through persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store. An entity class must follow these requirements, the class must be annotated with the javax.persistence.Entity annotations, the class must have a public or protected, no-argument constructor. The class may have other constructors. The class must not be declared final. No methods or persistent instance variables must be declaredfinal. If an entity instance be passed by value as a detached object, such as through a session bean’s remote business interface, the class must implement the Serializable interface. Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes. Persistent instance variables must be declared private, protected, or package-private, and can only be accessed directly by the entity class’s methods. Clients must access the entity’s state through accessor or business methods. Entity classes use comes not only from its persistence but its ability to be auto generated which will be gone into specifically further on in the paper.

**IV. Hibernate**

The method we chose to implement is Hibernate ORM (Hibernate in short) which is an object-relational mapping framework for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions. Hibernate is a free software that is distributed under the GNU Lesser General Public License. Hibernate's primary feature is mapping from Java classes to database tables (and from Java data types to SQL data types). Hibernate also provides data query and retrieval facilities. It generates SQL calls and relieves the developer from manual result set handling and object conversion. Applications using Hibernate are portable to supported SQL databases with little performance overhead.



Source :https://docs.jboss.org/hibernate/orm/3.5/reference/en/html/images/full\_cream.png

The main reasons we chose to implement is its ability to increase productivity, Hibernate gets rid of the need to write complex and tedious SQL statements, and removes the need for JDBC API to handle data. It’s much more maintainable as the amount of code is reduced from raw SQL calls to Java Criteria API statements. In addition to being free and open source the popularity of Hibernate makes troubleshooting easy as there are a huge number of resources available on the internet.

One of our primary requirements was the need for portability. This is one of Hibernate’s strengths and a large contender for why it was our primary choice of method. Hibernate makes object-relational mapping simple by putting 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. This way 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 also supports many databases, including MySQL, Oracle, Sybase, Derby, and PostgreSQL, and works well with plain old Java object (POJO)-based models, too.

Hibernate provides transparent persistence for Plain Old Java Objects (POJOs). The only strict requirement for a persistent class is a no-argument constructor, not necessarily *public*. Proper behavior in some applications also requires special attention to the *equals()* and *hashCode()* methods. Collections of data objects are typically stored in Java collection objects such as Set and List. Java generics, introduced in Java 5, are supported. Hibernate can be configured to lazy load associated collections. Lazy loading is the default as of Hibernate 3. Related objects can be configured to *cascade* operations from one to the other. For example, a parent Album object can be configured to cascade its save and/or delete operation to its child Track objects. This can reduce development time and ensure referential integrity. A *dirty checking* feature avoids unnecessary database write actions by performing SQL updates only on the modified fields of persistent objects.

Hibernate's architecture consists mainly of two interfaces called *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 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.

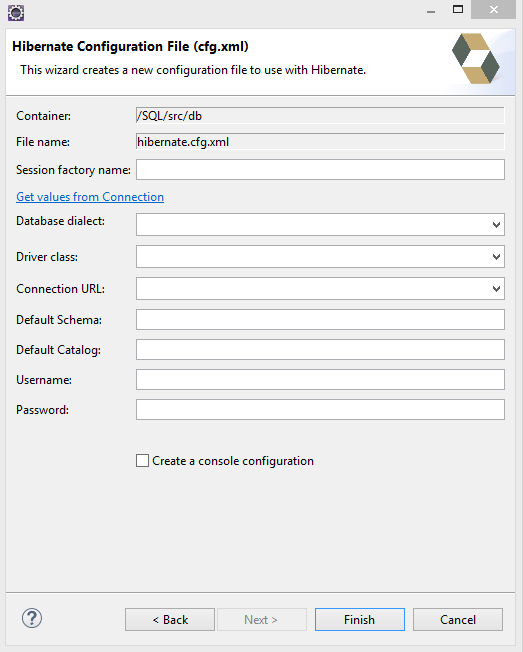
You configure Hibernate through an XML file named hibernate.cfg.xml. The configuration file aids in establishing a connection to a particular relational database. The configuration file should know which mapping file it needs to refer to. At runtime, Hibernate reads the mapping file and then uses it 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

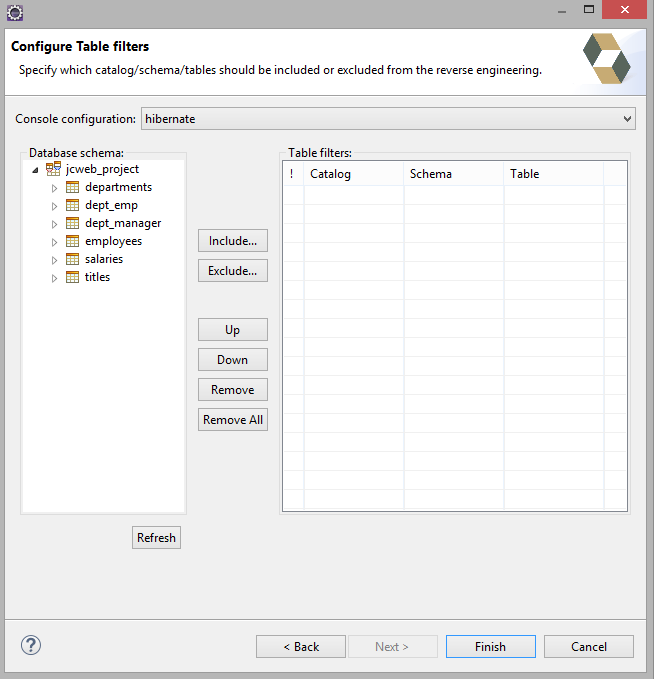
<hibernate-configuration>  
 <session-factory>  
 *<!-- local connection properties -->*  
 <property name="hibernate.connection.url">  
 jdbc:mysql://localhost/hibernateDemo  
 </property>  
 <property name="hibernate.connection.driver\_class">  
 com.mysql.jdbc.Driver  
 </property>  
 <property name="hibernate.connection.username">  
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>

**V. Hibernate Demo**

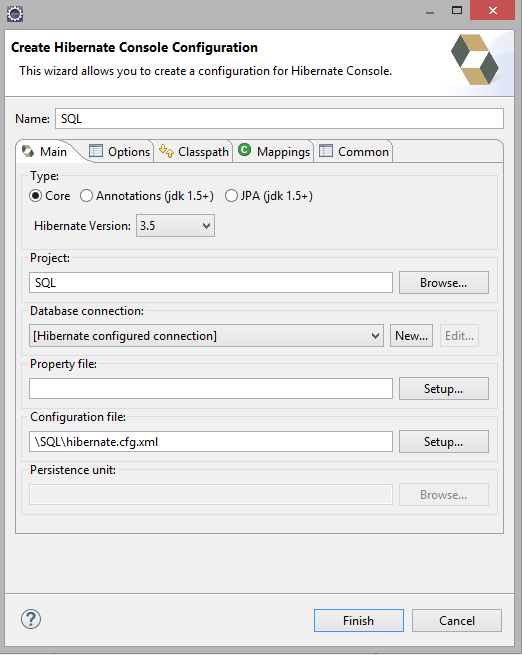
The following is a step by step process that was demoed on our visit to MSE. It included everything we learned through our process of building our first demo.

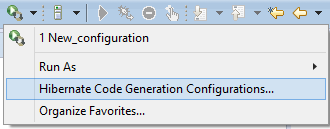


First you go to create and make a new configuration file needed to connect to the database. Using the hibernate cfg.xml create has many already built in options that given the correct info will make the config file for you.

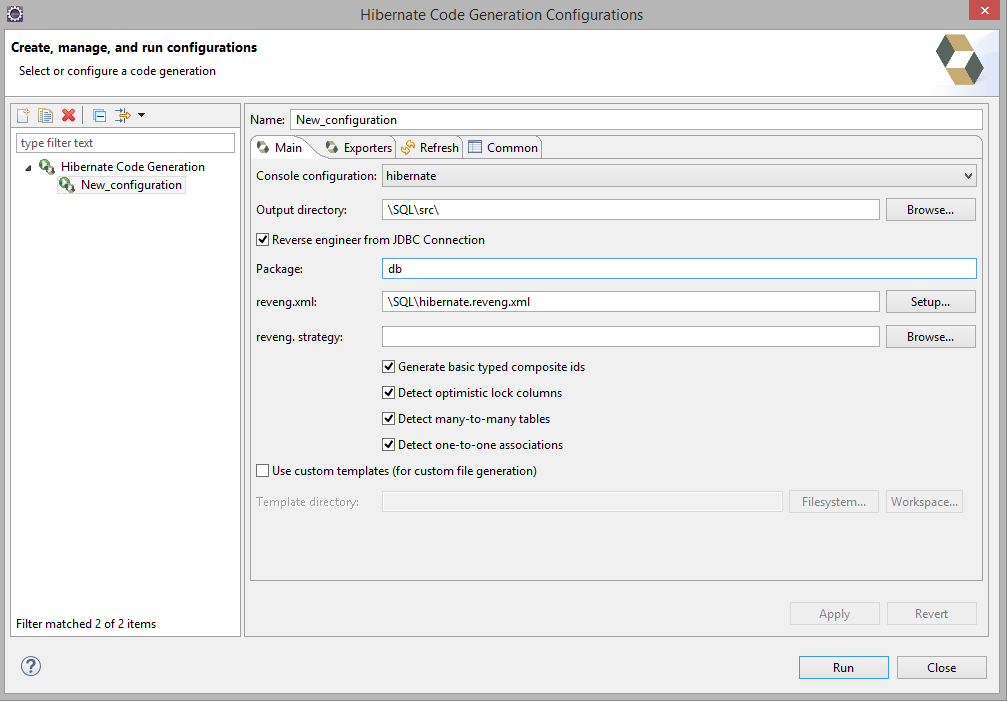
Next you need to create a Hibernate Reverse Engineer File

First you need to set the console to the configuration, which is the file you created earlier. After selecting, you must hit refresh otherwise you won’t be able to select, see, and include the tables or databases you want.

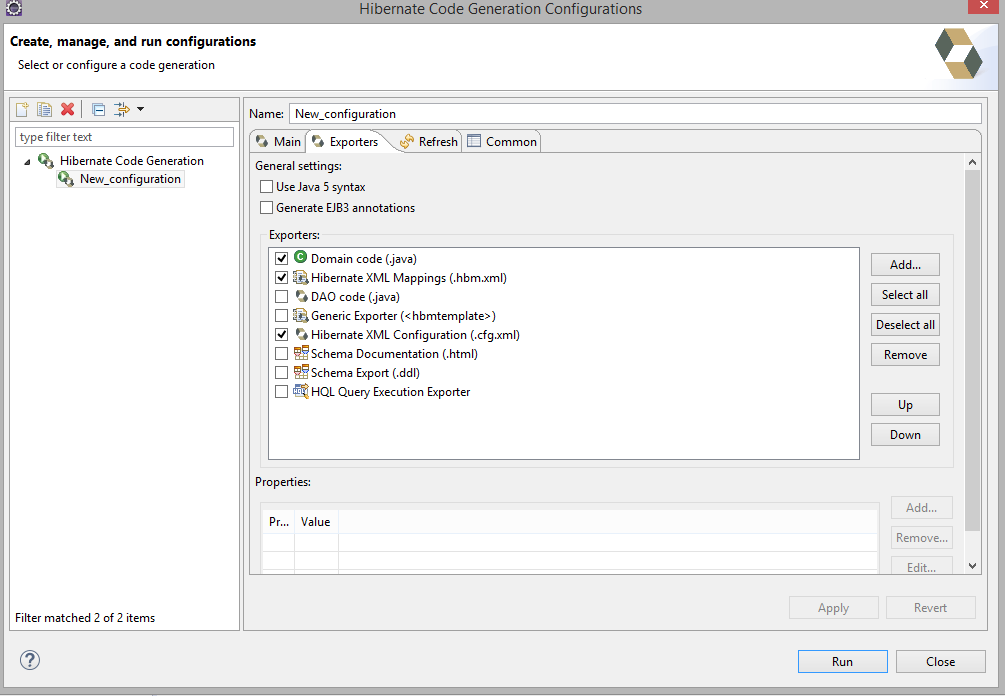


Next you’ll want to click the down arrow next to the hibernate run button and select the Hibernate Code Generation Configuration. 

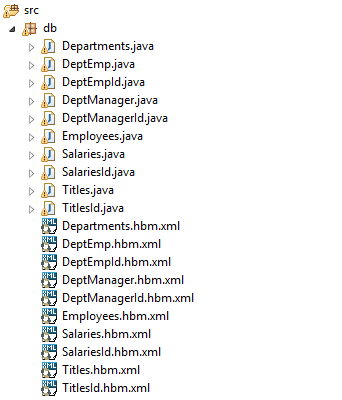
This brings up our next window where we specify where the files are going to be placed after being reverse engineered. Most of the options can be left as default, just specify where the reveng.xml you created is. If you want them to be in their own package, for example db, you must put it in the Package otherwise it will generate in the output but won’t function correctly.



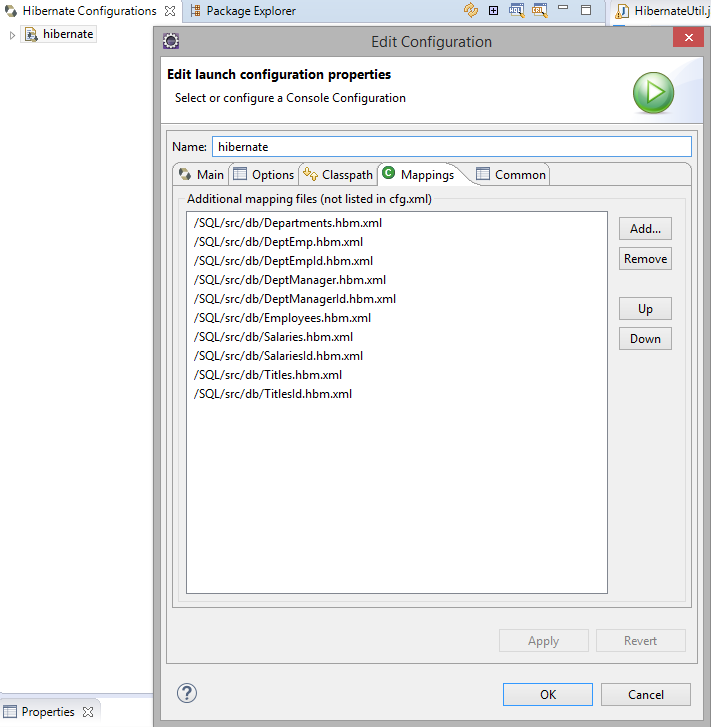
Next proceed to the Exporters tab of the window. This is where you will specify exactly you want to have outputted during the reverse engineering process. At the very least you will want Domain code which generates the Java POJO’s and XML Mapping files which connect your POJO’s to the respective database.



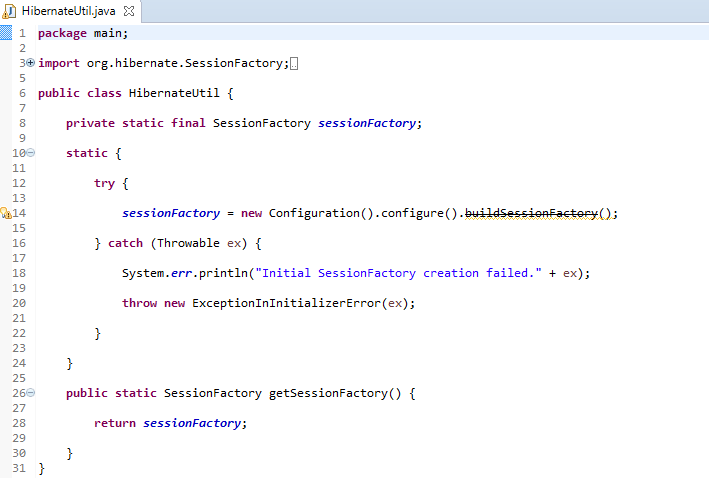
After all is configured, you will run it and if everything is set up correctly you will now have something similar to the screenshot below only in your specified database tables. The Java files are the POJOs and the xmls connect them to their respective databases.



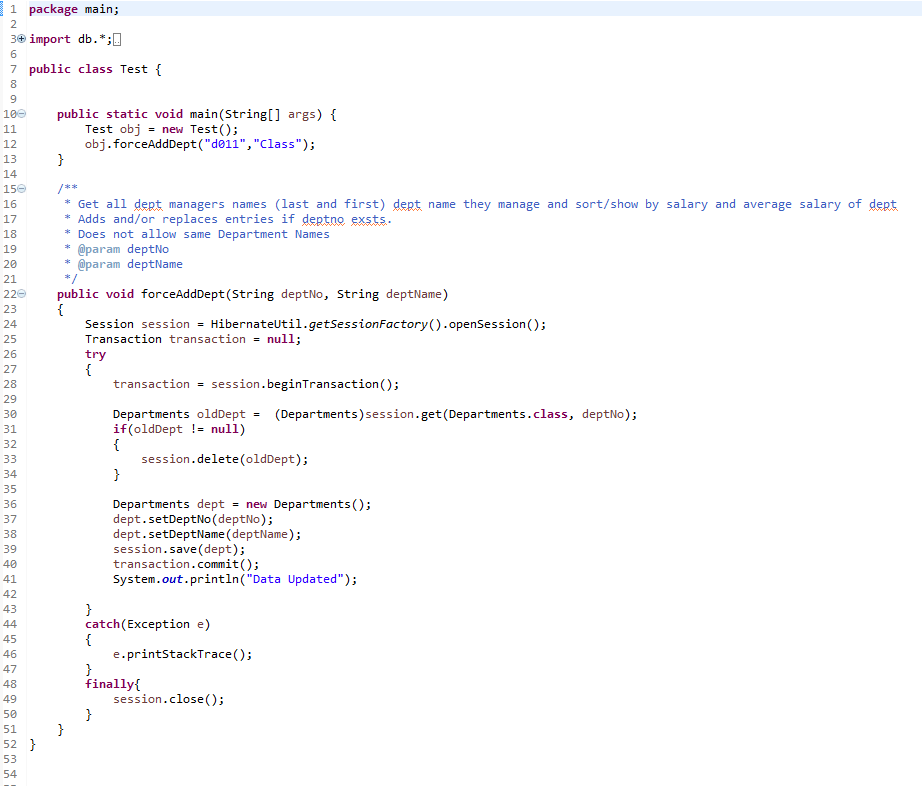
Finally you will want to go back to your configuration and double click it to bring up the below window. Go to mapping and make sure all the xml files are shown for you database, if none are there add them.



Next you'll want to create a class called HibernateUtil which manages the sessionfactory that you’ll be using with all your classes that use hibernate.

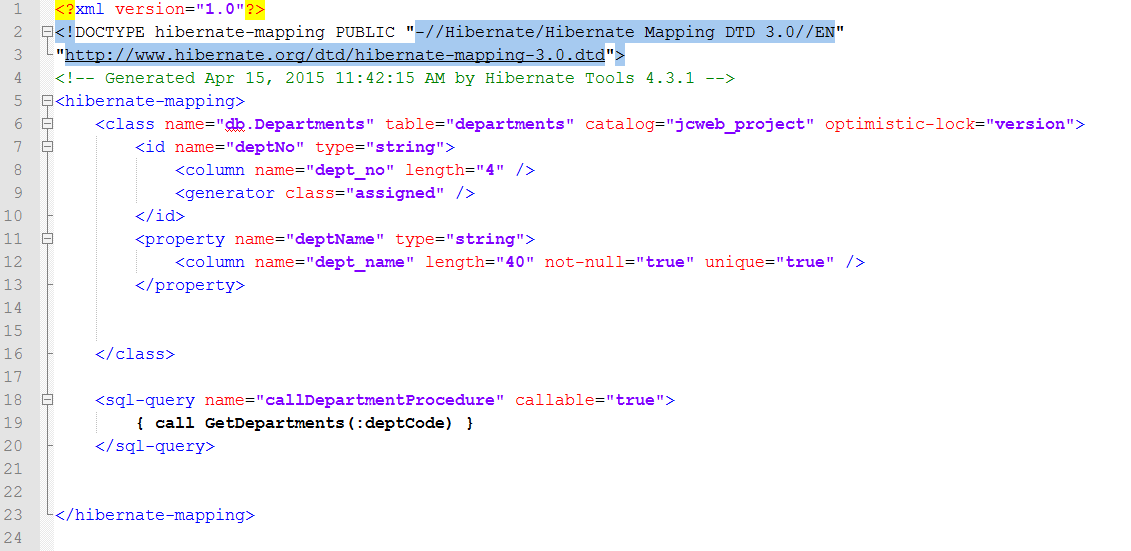


Heres an example on how to use the generated code. All the program does is check to see if a department exists, if it does it deletes it, and then adds the one you want to have added.

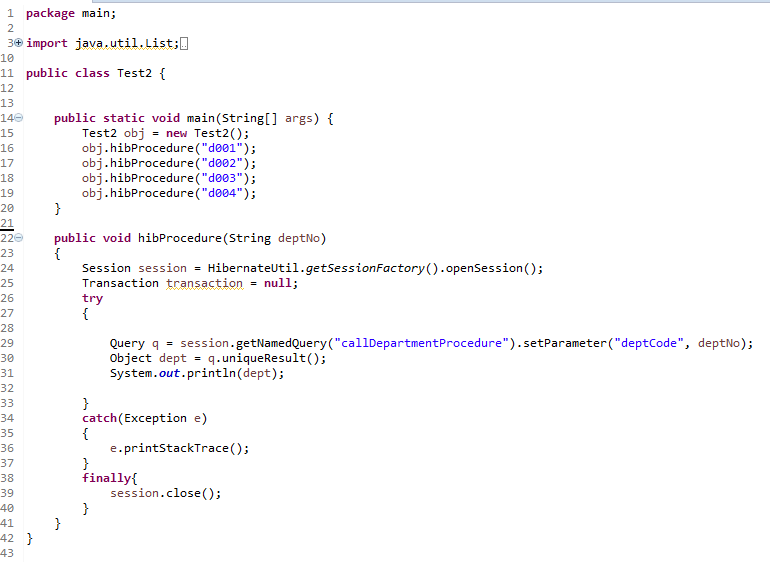


The following link is to a similar tutorial that may provide more information if you are attempting to complete a similar project on your own.

**Tutorial Source: http://examples.javacodegeeks.com/enterprise-java/hibernate/eclipse-hibernate-tools-plugin-tutorial/**

**VI. Additional Resources**Hibernate allows you to use stored procedures, as shown below, you are able to add them into the xml after the end of the class mapping. 

Finally a simple example on how to use the procedure. The below classes uses the stored procedure to get a requested department by number.



**VII. Final Thoughts**

Hibernate requires a lot of time to set up and use. For a smaller scale project I think it’s a great way to allow other team members to use a database without extensive knowledge of how a database works. Unfortunately Hibernate seems to struggle when working with larger scale tables and using multiple tables to get something done. I do believe that it is something worth looking into but it has its limitations and should not be used in some cases. I personally would consider using a hybrid of both hibernate and hard coded calls.

**VIII. Useful Links and Sources**

<http://www.javaworld.com/article/2077875/open-source-tools/ibatis--hibernate--and-jpa--which-is-right-for-you-.html>

<http://community.actian.com/w/files/4/44/JDBCvsORM.pdf>

<http://www.ibm.com/developerworks/java/library/j-genericdao/index.html>

<http://www.ibm.com/developerworks/java/library/j-genericdao/index.html>

<http://www.java-samples.com/showtutorial.php?tutorialid=813>

<http://www.tutorialspoint.com/jdbc/jdbc-driver-types.htm>

<http://www.oracle.com/technetwork/java/overview-141217.html>

<http://stackoverflow.com/questions/13433326/java-how-to-change-current-database-to-another>

<http://javarevisited.blogspot.com/2012/01/improve-performance-java-database.html>

<http://stackoverflow.com/questions/4477082/what-is-a-jpa-implementation>

<http://www.javaworld.com/article/2072999/data-storage/get-started-with-hibernate.html>

http://docs.oracle.com/javaee/5/tutorial/doc/bnbqa.html