**Hibernate’s Main Parts**

At broad level, there are three important pieces across all Hibernate applications:

* Persistent classes (our POJOs and AJOs [annotated Java objects!])
* Configuration and mapping definitions.
* Access and manipulation of the data using the API.

**Persistent Classes**

The fundamental requirement of a business application is that the data survives the application that created it. For instance, in our JustMovies! Application, a Movie object modeled to represent a movie can be accessed even if the application that created it ceases to exist.

We call fulfill this requirement of durable objects by creating domain objects (POJOs and AJOs) and using Hibernate to persist them in a durable storage space. Creating these objects for persistence is straightforward, as we’ll see…next.

**Example: Trading Application**

Let’s consider a Java application that persists and queries the Trades to and from a database. Each trade, which is defined as a Trade POJO, will be persisted in the database representing a unique row (assuming the id is generated uniquely by our application) in the TRADES table.

There is nothing special about the persistence class from Hibernate’s perspective. As the name suggests, they are plain old java objects (or annotated Java Objects) that Hibernate know exactly how to handle.

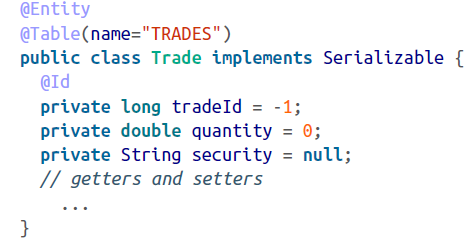
The only bit that we have discuss is the *identity* of the object. Hibernate mandates that every persistent object have a unique identifier. In the Trade class shown earlier, the traded variable forms a unique Trade object. It is the primary key of the table and hence persisted in the TRADE\_ID column of our TRADES table. We must tell Hibernate about identifiers via our mapping definitions.

**Using Annotations**

Note that, conventionally, the mapping of the persistent classes to the database table is done outside of the code, via mapping file (usually an XML file).

There is another way to do this. Java 5 introduced annotations to the language, which were quickly adopted by many frameworks, including Hibernate. Annotations include metadata about a class that is added to the class at the source code level. They do not alter or affect how the actual source code works though.

Here’s our modified Trade class, with mappings declared with annotations at the class and variable class.



Each persistent object is tagged (at a class level) with an @Entity annotation. The @Table annotation declares our database table where these entities will be stored. Ideally, we should not have to provide the @Table annotation if the name of the class and the table name are the same (in our example, the class is Trade, whereas the table name is TRADES.

The @Id annotation indicates that the variable is the unique identifier of the object instance (in other words, a primary key)

Hibernate uses the Java Persistence API (JPA) annotations. JPA is the standard specification dictating the persistence of Java objects. So the preceding annotations are imported from the javax.persistence package.

We will be working with both (annotations and XML file) in this book.

Once we define our persistent classes, the next step is to work on the configuration and mapping.

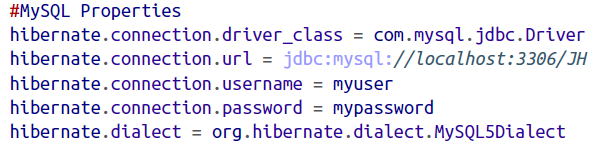
**Configuration**

In order to connect to the database, the Hibernate framework needs to know information about the database, such as its URL, credentials, and dialects. During the runtime of our application, this configuration is ready by the framework to establish a connection to the database.

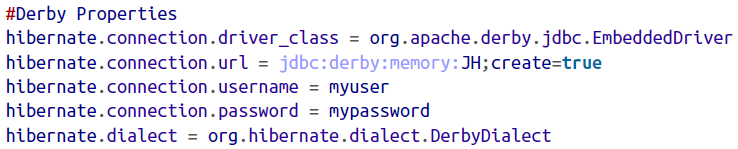
Hibernate database configuration is usually a one-off process. We saw in the previous chapter that the configuration is provided either via properties file ( hibernate.properties ) or an XML file ( hibernate.cfg.xml ). However, we can use different filenames according to our preference, In which case we need to tell Hibernate explicitly to load then straightaway, if no other filename is specified.

**Using a Properties File**

The connection properties are provided as name/value pairs in the hibernate.properties file, as shown here:



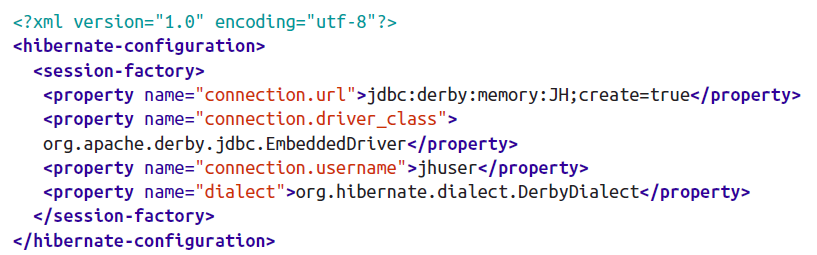
The preceding properties point to a MySQL database. We need to change the appropriate properties such as driver\_class, url, and others when working with different databases. For example, the following snippet shows the connection details for Derby (an in-memory java database, also called JavaDB):



Make sure the vendor’s driver classes are available in the classpath of the application.

**Using the XML File**

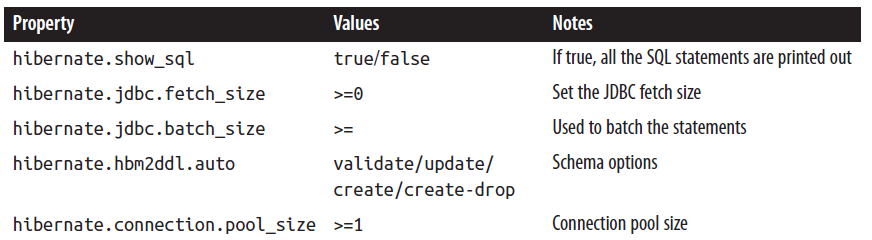
The alternative to using a properties file is declaring the metadata in an XML file as follows:



You may have noticed, the properties are used to create a SessionFactory object and hence declared under the session-factory tag. Also, the hibernate.\* prefix is dropped here as opposed to the properties defined in the hibernate.properties file.

**Configuration Properties**

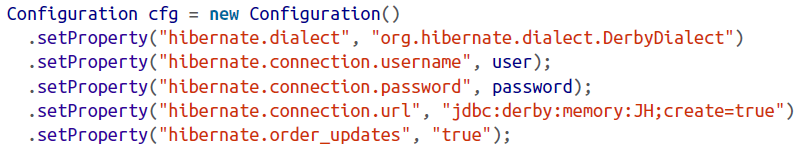
Now that you know how to write in the properties to a Hibernate runtime, let’s see what other properties we can use.



Now that we have given choice of configurations (both the properties file and an XML file), you might be wondering which one to use. There is no hard-and-fast rule; you can use both of them if you wish. In fact, Hibernate’s runtime does not complain if both of them are available to the application, but it simply ignores the properties file. So, any properties in the XML file take precedence over those in the properties file.

**Programmatic Configuration**

Both methods just discussed (properties and XML file) are declarative modes. Hibernate also supports programmatic configuration. In this case, you can use the appropriate classes to be instantiated with the Configuration class. This is shown in the following snippet:



Alternatively, you can pass the parameters as standard VM (virtual machine) arguments:

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

**Mapping**

The next thing we should think about is what kind of objects will be persisted by our application. We should also think about how object fields are mapped to the table columns and who controls what fields will be persisted.

By reading mapping XML files, Hibernate will transform object data to a row-column relational equivalent. Which we discuss in the following sections.

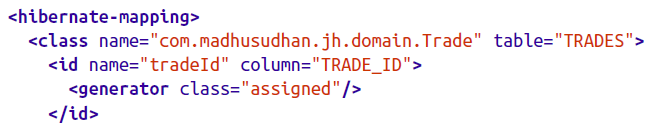
Mapping is metadata that Hibernate digests to produce an object-to-relational data transformational plan. This plan lets us use object-oriented persistence in our Java applications with relational database. As discussed earlier we can use simple XML files or annotations to declare our metadata mapping.

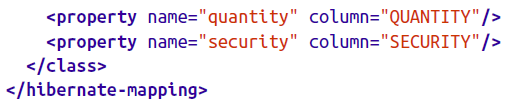
**XML Mapping Files**

We will declare each domain POJO using the metadata in an XML file with an extension of .hbm.xml. This file should be made available in the classpath so Hibernate can bootstrap(help itself) the mapping definitions.

For example, the Trade object mapping should be defined in the Trade.hbm.xml file, Movie in Movie.hbm.xml, and so on. It’s not mandatory to declare one mapping file or one object; in fact, we can define the entire model in one single mapping file. The better option is that you use one individual mapping for each persistent object, because domain model becomes more complex.

So, the Trade.hbm.xml mapping definition file is as follows:





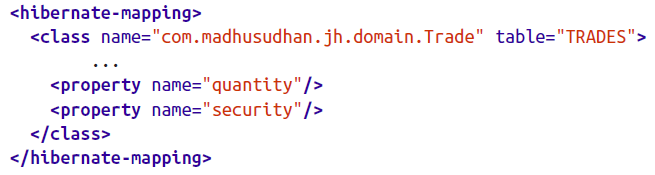
There are few things to take away from this file before we move on.

The class versus table mapping is done via a class tag, as highlighted in the preceding snippet. The class declaration simply says that our Trade object will be persisted to the TRADES table. You can see that a table attribute is defined too; this would indicate where objects are saved.

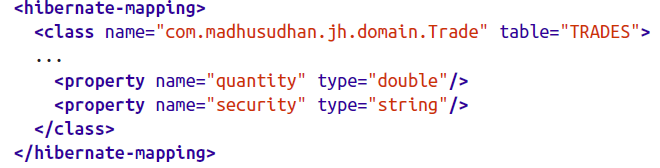
In addition to declaring the mapping from the class to a table, we need to tell Hibernate which object properties should be persisted.

The first (and most important) of the list of persistent properties is the identifier (or primary key) of the object. The tradeId is mapped to the primary key of the table TRADE\_ID via the id tag. The name of the id tag correspond to the variable tradeId we defined on the Trade class. Hibernate calls the getTradeId nad setTradeId accordingly to set or get the TRADE\_ID variable.

The field to which our Trade object will be persisted is defined by the property tag. In the preceding example, the quantity field on the Trade objet will be stored in the QUANTITY column as defined by the column tag, and the security value is mapped to the SEQURITY column.

There is a simpler way to create a mapping. If the name of the column matches the variable name of the object, we can omit the declaration of the column property. Thus, we can modify the previous example and omit the column tags for quantity and security (as the variable name match the column names.)

In the preceding snippet, we omitted mentioning the data types of our attributes. How does Hibernate know that *quantity* is a double type or *security* is a String type? Well, ideally we should declare the types too, using the type tag, as shown here:



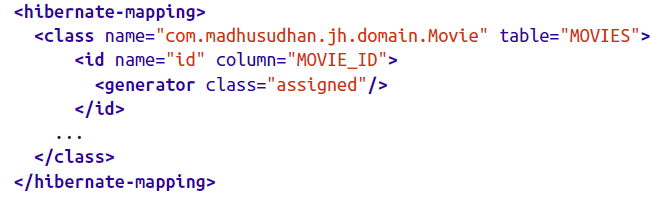
We can omit these types and let Hibernate work out the variable types (by using Java reflection) or we can explicitly declare them.

I will prefer to declare them explicitly to reduce the startup, as Hibernate will not waste time finding out the types by using Java Reflection.

**Identifier Generation Strategies**

Each object must be persisted to the database with a unique identifier. We can use different strategies for generating these identifiers automatically. Hibernate provides a rich set of such Strategies, which we will see in this section:

We define a generator class for setting up an id element as follows:

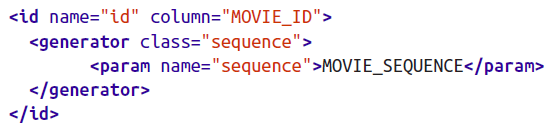


The generator is the key element that lets Hibernate know our choice of identifier generation strategy. The class attribute on the generator defines the actual strategy we are going to use. For example, in the preceding snippet, we are using assigned strategy, meaning our code is expected to set a unique identifier on each of the persistent objects.

The “assigned” is a shortcut name for the org.hibernate.id.Assigned class. Instead of requiring us to provide the fully qualified name of the generator class, Hibernate lets us use short version for our convenience. For example, the “sequence” name corresponds to org.hibernate.id.SequenceGenerator, “identity” corresponds to org.hibernate.id.IdentityGenerator, and so on. All these classes implement a common interface, org.hibernate.id.IdentifierGenerator.

Hibernate offers quite a few generators out of the box- for example, identity sequence, native, and assigned. Should you have a specific requirement of you own primary key generation strategy, you can create a custom strategy by implementing the Identifier Generator interface and providing the custom logic.

We can specify required arguments to these generators using *param* elements. For example, when using sequence strategy, we need to let Hibernate know what our sequencer table is:



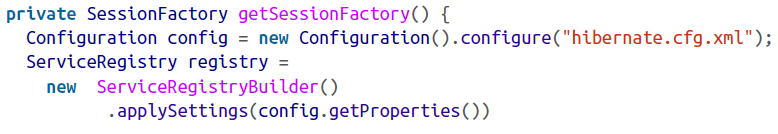
Refer Hibernate’s documentation to run through the various strategies Hibernate Provides.

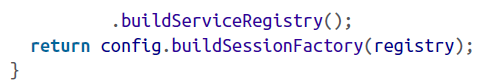
**Session APIs**

Hibernate comes with a rich set of APIs for object persistence and retrieval. While looking at the entire API set is outside the scope of this book, we’ll discuss the most important and fundamental Session APIs, which are necessary for understanding Hibernate concepts.

We had a primer on Hibernate’s most basic APIs earlier in the chapter. The SessionFactory, represented by the org.hibernate.SessionFactory class, is a factory class for giving out our Session instances. **It is a thread-safe object** and hence can be shared across various classes without our having to worry about the data being corrupted. We supply the mapping information to this class upon creation, so it contains all the mapping data in a compiled form.

If you remember:

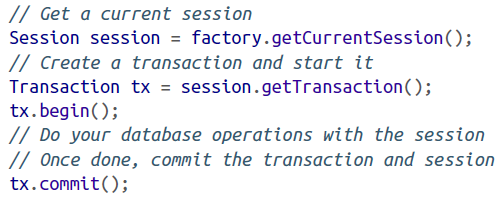




Note that we are using Hibernate’s newly introduced (in 4.x) service registry classes to create the configuration.

The SessionFactory also maintains a second level of cache, which is available globally across all other components in the applications. Global cache is used if multiple application requires the same data that’s been loaded from the database. This will speed the application’s request time.

While SessionFactory holds the access keys to the doors of a database, the Session itself is the key to interact with and access the database. Session is a single-threaded object and therefore should not be shared across various components. It represents a single unit of work. We use factory.getCurrentSession() to fetch a session from the factory. Once we have the session object, we can perform our database operations in a transaction. The Session and Transaction go hand in hand. The usual life cycle of the session and a transaction is demonstrated in the following snippet:



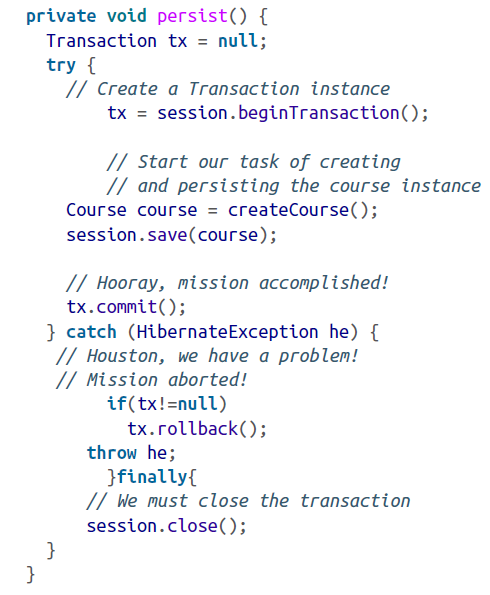
Note that the first level of cache is maintained in the session; (i.e, all the objects that were fetched or accessed will be held in the session until the session is close). We should not use the session per individual operations.

Ideally, all the related database operations should be grouped under one transaction.

**Transactions**

One of the fundamental pieces in performing actions on business data is to work with transactions. In simple terms, transactions keep our work segregated from others, and synchronize with the durable storage to avoid incorrect data being read or written. There are four fundamental properties that database transactions revolve around: **atomicity, consistency, isolation, and durability**, which are collectively known as ACID properties. Understanding transactions will help you design great software.

Take the example of booking a train ticket from London to Paris on EuroStart. When you pay for your ticket, you want the ticket to be confirmed and your seat allocated all in one go. Should there be a hiccup in the booking process, you wouldn’t want to have spent you money and have no booking! So either everything should go in an orderly fashion, succeeding at each step, or the entire process should abort if any step fails in processing this order. This sequence of steps is wrapped in a *transaction.*

In a standalone JVM, we need to work with the transaction exclusively. The following snippet persist the Course object to the durable storage space:

We initiate a transaction by invoking the *session.beginTransaction* method, which creates a new *Transaction* object and returns the reference to us. It gets associated with the session and is open until that transaction is committed or rolled back.

We perform the required work in a transaction, then issue a commit on this transaction. At this stage, the entities are persisted to the database. While persisting, If for whatever reason there are any errors, the Hibernate runtime will catch and throw a HibernateException (which is an unchecked RuntimeException). We then have to catch the exception and roll back the transaction. We may throw additional exceptions back to the client for information or debugging purposes.