**An Introduction to Java Persistence for Client-Side Developers**

by [Joshua Marinacci](http://today.java.net/pub/au/79)  
05/25/2006

* **Contents**
* [What Are *Java EE Persistence*, *Hibernate*, and *Hypersonic*?](file:///C:\Documents%20and%20Settings\Marcos\Desktop\FSA\aulas%20des%20web\An%20Introduction%20to%20Java%20Persistence%20for%20Client-Side%20Developers.htm#what-are)
* [Building Your First Persistable Object](file:///C:\Documents%20and%20Settings\Marcos\Desktop\FSA\aulas%20des%20web\An%20Introduction%20to%20Java%20Persistence%20for%20Client-Side%20Developers.htm#building)
* [Setting Up Your Project](file:///C:\Documents%20and%20Settings\Marcos\Desktop\FSA\aulas%20des%20web\An%20Introduction%20to%20Java%20Persistence%20for%20Client-Side%20Developers.htm#setting-up)
* [Database Configuration](file:///C:\Documents%20and%20Settings\Marcos\Desktop\FSA\aulas%20des%20web\An%20Introduction%20to%20Java%20Persistence%20for%20Client-Side%20Developers.htm#db-config)
* [Executing a Query](file:///C:\Documents%20and%20Settings\Marcos\Desktop\FSA\aulas%20des%20web\An%20Introduction%20to%20Java%20Persistence%20for%20Client-Side%20Developers.htm#executing-query)
* [Starting the Database from Within Your Application](file:///C:\Documents%20and%20Settings\Marcos\Desktop\FSA\aulas%20des%20web\An%20Introduction%20to%20Java%20Persistence%20for%20Client-Side%20Developers.htm#starting-db)
* [Conclusion](file:///C:\Documents%20and%20Settings\Marcos\Desktop\FSA\aulas%20des%20web\An%20Introduction%20to%20Java%20Persistence%20for%20Client-Side%20Developers.htm#conclusion)
* [Resources](file:///C:\Documents%20and%20Settings\Marcos\Desktop\FSA\aulas%20des%20web\An%20Introduction%20to%20Java%20Persistence%20for%20Client-Side%20Developers.htm#resources)

Most applications involve creating, storing, and searching data in some form; in other words: use a database. This is often called *CRUD*, for *Create, Read, Update, and Delete.* Databases are integral parts of almost all computing systems. Well, I have a confession to make: I've been a professional software engineer for close to ten years now and I don't know anything about databases. Sure, I can write a simple SELECT call, but if you ask me to do an double outer join with foreign keys or convert my schema to 18th normal form, I'll simply get lost and give up. I don't know databases. In fact, I hate databases. I'm a client guy. To me, a database is simply a box to store stuff. I don't know how they work and I **don't want** to know. However, the sad fact of client software today is that as every application gets bigger, it eventually needs a database. From the largest billing system to something as simple as an address book, almost every application needs a database. But this doesn't change the fact that I still hate them. So what are we client developers to do? Here's a idea: why don't we steal some of the server guys' best technology to make our lives easier?

Building on the new features in Java SE 5 (a.k.a. Tiger), the Java Enterprise Edition Enterprise Java Beans 3 spec (hereafter known by the blessedly shorter "EJB3") introduced a new way of communicating with databases called the Java Persistence API (JPA). Though primarily designed for use in big, server-based applications, I will show you how to use JPA to easily load, save, and search for data objects in a simple address book application. In a follow-up article, I will cover advanced object mapping, inheritance, and fetching. Most importantly, you will be able to do all of the database tasks without needing to know SQL, JDBC, or any other traditional database technologies. Let's hear it for the client side!

**What Are Java EE Persistence, Hibernate, and Hypersonic?**

Before we get into our application, let's take a moment to go over our tools: the Java Persistence API, Hibernate, and Hypersonic.

EJB 3 introduced a new technology spec called the Java Persistence API, which defines a persistence framework. Persistence is a way of automatically mapping normal Java objects to an SQL database. In other words, it loads, searches, and saves your data model objects. Rather than writing cumbersome SQL insert, update, and select commands, the framework will take care of the boilerplate code through some kind of automation. The Java Persistence spec codifies the various persistence frameworks into a single API. This means you can write your application to the spec and know that it will work across several implementations, now and in the future.

[Hibernate](http://www.hibernate.org/) is an open source implementation of the Java Persistence spec. Though it preceded JPA, and in fact influenced the final design, it has been retrofitted to implement the full spec (along with some extensions that I won't cover here). Hibernate is one of several implementations available, but I have chosen this one because it is complete, mature, and freely available. Much like JDBC, if you stick to the spec and don't use any of Hibernate's extensions, you will be able to switch implementations as your needs change in the future.

[HSQLDB](http://www.hsqldb.org/), formerly "Hypersonic," is an open source SQL database that stores everything in a couple of files on disk. It has the advantages of being very small, easy to configure, and written entirely in Java. These features make it ideal to embed directly inside of your application: *no database server required*. Again, in this article we will stick to the database specs and not use any Hypersonic-specific features. This means you could upgrade to a full database server in the future if necessary.

**Building Your First Persistable Object**

In this article, we will build a simple address book application. The heart of any application is its data model. It is these objects that we want to create, save, and search. For an address book, I want to store people, so I have created the Person object below:

package addressbook;

import javax.persistence.\*;

@Entity

public class Person {

@Id @GeneratedValue

public Long id;

public String first;

public String middle;

public String last;

}

As you can see, this is a very simple class. It has three String fields for the first, middle, and last names, along with a Long for the ID. It looks like any normal Java object. In fact, the term most often used with persistence frameworks is *POJO*, or "Plain Old Java Object." The only things out of the ordinary for this object are the @Entity and @Id parts. Any symbol in Java code beginning with @ is known as an *annotation*. The use of annotations is a new language feature introduced in Java SE 5.0 that allows you to add metadata to your objects. Annotations mark parts of your objects so that other systems can do something special with them. In this article we will mark certain fields and classes with persistence annotations so that the Persistence framework will know how to save them.

In the code above, the Person class is marked with the @Entity annotation, which means that it is an EJB entity. Being marked as an entity has a lot of ramifications, but for our purposes it just means that this object can be persisted. Any object you want to stuff in the database must be marked with the @Entity annotation. The @Id annotation says that the id field will be used as the unique identifier for this object. Databases use a special table column called a *primary key* to distinguish between rows. The @Id annotation marks the id field as the primary key for this object. The @GeneratedValue annotation tells the Persistence framework to generate IDs for us, which is fine, because we don't care what they are as long as they are unique. The @Id annotation has many optional parameters that let you further specify how your primary key works, such as selecting a particular column name or using multiple fields. For our needs, the default behavior is fine.

You may notice that the other fields--first, middle, and last--don't have any annotations. That is because they are using the default annotations that say that the field is persistable as an SQL VARCHAR. As you explore the Java Persistence framework you will discover the JSR expert group did a very good job of defining defaults. This means that you only need to use non-default values when you want to do something out of the ordinary. This design choice, part of the overhaul in Java EE 5, makes persistence very easy to use for non-server developers.

**Setting Up Your Project**

Combining JPA, Hibernate, and Hypersonic requires a lot of .jars. If you want to build your own workspace, you must first go to the [Hibernate](http://www.hibernate.org/) website and download the Hibernate Core, Hibernate Annotations, and Hibernate Entity Manager .zip files. Be sure you download version 3.2cr2 or newer of the Hibernate Core file, because older versions won't work with annotations. Unpack them and put all of the .jars into your classpath (I typically put them in a *lib* subdirectory).

There are a lot of .jars and not all of them are needed, depending on what you are doing, so I recommend using them all during development and then removing the ones you don't need when you get to deployment. The only one you should need in your compile-time classpath is *ejb3-persistence.jar*. The rest are for runtime use. Hibernate contains a *lib/README.txt* file with more details on the library requirements. You will also need the HSQLDB database in the form of the *hsqldb.jar* file.

A quick note on library versions: the Hibernate Persistence implementation has been undergoing changes as the EJB 3 spec moves towards final ratification. Because of these changes, some versions of the Hibernate Core .jars don't work with some versions of the Annotations and Entity Manager .jars. When writing this series of articles, I found a few occasional NoSuchMethodErrors, indicating a version mismatch. I recommend downloading all versions of the modules that were released on the same day. For this article I used the versions released on March 27th, 2006. These are Hibernate Core v 3.2.0.cr1, Annotations v 3.1beta9, and Entity Manager v 3.1beta7. Until the changes settle down, I recommend staying with these versions. I have found them to work flawlessly.

Now that you have a persistable object, the big question is: how do you persist it? This is where the EntityManager comes in. Each implementation of the Persistence API will have its own ways of connecting to a database and managing the objects, but as long as you don't use any implementation-specific extensions, you can stick to the official interfaces. The EntityManager keeps track of your objects and allows you to save, load, and search them at will. Like many Java APIs, Persistence uses a factory pattern, so that's where we will start. Here is a typical test application:

package addressbook;

import javax.persistence.\*;

public class Main {

EntityManagerFactory factory;

EntityManager manager;

public void init() {

factory = Persistence.createEntityManagerFactory("sample");

manager = factory.createEntityManager();

}

To use persistence, you must first create an EntityManagerFactory using the static Persistence.createEntityManagerFactory() method. Notice the string "sample" passed to the method. This is the *persistence unit* you will be using, which must match the configuration in the *persistence.xml* file that we will get to later. Once you have a factory, you can get an EntityManager with the createEntityManager() method. In general, you should have one factory per application and one manager per user, usually handed out and managed by your application container. This is because the factory is thread-safe and the manager is not. However, these constraints were designed with web server applications in mind. For a desktop app where there is only person using it you can simply save the references and use them as you need. Just remember that the EntityManager isn't thread-safe, so be sure to do all of your persistence on the same thread or use multiple EntityManagers.

Once you have finished your persistence operations you can shut down the manager and factory with their respective close methods.

private void shutdown() {

manager.close();

factory.close();

}

public static void main(String[] args) {

// TODO code application logic here

Main main = new Main();

main.init();

try {

// do some persistence stuff

} catch (RuntimeException ex) {

ex.printStackTrace();

} finally {

main.shutdown();

}

}

}

Above, you can see the main() method which creates a Main object, initializes the persistence system, does some work, and then calls shutdown() in a finally block. This simple Main class encapsulates the typical lifecycle of an application that uses persistence.

Now that the persistence system is set up, we can finally persist some objects! You just need to create a transaction, create your objects, and then save them.

private void create() {

System.out.println("creating two people");

EntityTransaction tx = manager.getTransaction();

tx.begin();

try {

Person person = new Person();

person.first = "Joshua";

person.middle = "Michael";

person.last = "Marinacci";

manager.persist(person);

Person sister = new Person();

sister.first = "Rachel";

sister.middle = "Suzanne";

sister.last = "Hill";

manager.persist(sister);

tx.commit();

} catch (Exception ex) {

tx.rollback();

}

System.out.println("created two people");

}

Transactions are pretty much required with persistence because you are talking to a database underneath. You will usually create a transaction with manager.getTransaction(), call tx.begin(), and then do all of your work. The code above calls manager.persist() to save the new objects, but this could easily be calls to load, search, or update objects as well. Once finished you must call tx.commit() to make the changes permanent. Notice that the calls are wrapped in try/catch blocks. This is so you can rollback the transaction if anything goes wrong, leaving your database in a consistent state.

**Database Configuration**

There is one final piece of the puzzle before you can start saving objects. The entire persistence stack is configured with a *persistence.xml* file. This file lists the classes you want to persist, the database connection, and any properties that are specific to the Persistence implementation (Hibernate, in this case). Once you are set up, *this XML file will be the only thing you need to modify if you change or move your database*. This feature underlies the power of the Java Persistence API and makes the transition from developer database to big Oracle server quite easy. Here's what the *persistence.xml* file looks like:

<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation=

"http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence\_1\_0.xsd"

version="1.0">

<persistence-unit name="sample" transaction-type="RESOURCE\_LOCAL">

<class>addressbook.Person</class>

<properties>

<property name="hibernate.connection.driver\_class"

value="org.hsqldb.jdbcDriver"/>

<property name="hibernate.connection.username"

value="sa"/>

<property name="hibernate.connection.password"

value=""/>

<property name="hibernate.connection.url"

value="jdbc:hsqldb:hsql://localhost"/>

<property name="hibernate.dialect"

value="org.hibernate.dialect.HSQLDialect"/>

<property name="hibernate.hbm2ddl.auto"

value="update"/>

</properties>

</persistence-unit>

</persistence>

In the file above, the first thing you should notice is the name attribute of the persistence-unit element. It doesn't matter what this name is, as long as it matches the string passed into the Persistence.createEntityManagerFactory() method. Again, this API was designed to support advanced uses like multiple persistence stores and databases. Since this is uncommon for desktop apps, I recommend just using one name and never changing it. The *persistence.xml* file must be in the *META-INF* directory of your application .jar. If you are using NetBeans, you can put it in a *META-INF* directory of your *src* directory and let NetBeans copy it to the right place during compilation.

Finally we are ready to crank up the database and start saving objects. Hypersonic is contained in a single .jar, *hsqldb.jar*, and you can start it with all of the default settings right from the command line.

java -classpath lib/hsqldb.jar org.hsqldb.Server

This will start an instance of Hypersonic on its default port with the database stored in the *test.\** files placed in the current working directory. Using the defaults is convenient for testing because you can remove the data by just deleting the created *test.\** files when you are done.

If you compile and run the Main class, and have the database started in another window, then you should get a lot of output from Hibernate followed by the two lines:

creating two people

created two people

Congratulations! You've just persisted your first objects. After you get the initial setup done, you'll discover how easy it is to store and retrieve your objects. You'll never need to write SQL again.

**Executing a Query**

Now that we can save our objects it would be nice to load them back up. You can do this with a simplified SQL-ish syntax called *EJB3-QL*. It lets you perform simple queries into the persistence store, much like SQL, except that instead of returning ResultSets, it will return Lists of your data objects. Below is a simple search function.

private void search() {

EntityTransaction tx = manager.getTransaction();

tx.begin();

System.out.println("searching for people");

Query query = manager.createQuery("select p from Person p");

List<Person> results = (List<Person>)query.getResultList();

for(Person p : results) {

System.out.println("got a person: " + p.first + " " + p.last);

}

System.out.println("done searching for people");

tx.commit();

}

The query itself is generated by the manager.createQuery("select p from Person p") call. This is the simplest possible query. It will return all instances of the Person class. You could fully qualify the Person class as addressbook.Person, but this isn't necessary because the Persistence engine will use the unqualified classname by default. Once you have created a query, you can execute it with query.getResultList(). The code above places the query inside of a transaction, just like the create() method did earlier in this article.

The EJB3-QL language is case-insensitive except for the Java class and method names, so select is equivalent to SeLeCt. It supports most of the usual SQL select features, such as where clauses. For example, if I wanted to search for just Persons with the first name of Joshua I could use the query: select p from Person p where p.first='Joshua'. The EJB3-QL language is powerful enough to perform most of the complicated things you could do with real SQL, like inner and outer joins, but I find that to be unnecessary for most client applications. I prefer to do simple where searches and then do any extra refinement in Java code.

**Starting the Database from Within Your Application**

Now that the program can load and save objects, it would be nice to have the database bundled with the application. We can't expect our users to manually start Hypersonic in another window. Fortunately Hypersonic was designed to run in-process, so this is quite easy.

Before you create the EntityManager you must start Hypersonic by obtaining a database Connection object like this:

public static void main(String[] args) throws Exception {

// start hypersonic

**Class.forName("org.hsqldb.jdbcDriver").newInstance();**

**Connection c = DriverManager.getConnection(**

**"jdbc:hsqldb:file:test", "sa", "");**

// start persistence

Main main = new Main();

main.init();

// rest of the program....

Then, just before your program ends, you must shut down Hypersonic with an SQL SHUTDOWN command like this:

// shutdown hypersonic

**Statement stmt = c.createStatement();**

**stmt.execute("SHUTDOWN");**

c.close();

} // end main() method

Also, since you have now changed the database configuration, you'll need to update your persistence.xml file, specifically changing the connection URL property from this:

<property name="hibernate.connection.url" value="jdbc:hsqldb:hsql:**//localhost**"/>

to this:

<property name="hibernate.connection.url" value="jdbc:hsqldb:**test**"/>

With those few changes, the database will start up inside of your application, save all data to the test.\* files, and shut down when your application ends. Hypersonic is a simple and compact way to store all of your application in a real database without your users ever knowing.

**Conclusion**

The Java Persistence API was designed as a simple way to load and save your data objects without using any SQL at all. Since it was also designed to be implementation independent, you can change your database or persistence provider as your needs change and not worry about anything breaking. Though Java Persistence was originally designed for server tasks, it performs beautifully in client applications, letting client-side developers spend less time on storage and more time on what they are good at: building killer GUIs.

Join me in a few weeks a follow-up article in which we explore advanced persistence features like constraints, customized properties, subclasses, and managing entire graphs of objects at once.

**Resources**

* [Source code](http://today.java.net/today/2006/05/23/AddressBook.zip) for this article
* [The Java EE 5 Tutorial](http://java.sun.com/javaee/5/docs/tutorial/doc/index.html): see Chapter 25: "Introduction to Java Persistence"
* [The Java (EJB/JDO) Persistence FAQ](http://java.sun.com/j2ee/persistence/faq.html)
* [Hibernate: Relational Persistence for Java and .NET](http://www.hibernate.org/)
* [HSQLDB (Hypersonic)](http://www.hsqldb.org/)

[*Joshua Marinacci*](http://today.java.net/pub/au/79) *first tried Java in 1995 at the request of his favorite TA and has never looked back.*