Hibernate: Storing and Retrieving

by Perron Jones

Once Hibernate has been set up in your Java environment, you may have major difficulty figuring what you need to do get make it do what you want. I will give an overview of important information that will help you get started so you can store, retrieve, and remove objects using your database.

Initial actions

Hibernate makes use of an entity manager. Whenever you want to start interacting with database to store or get something, you have to establish a connection with the database through the entity manager.

First, you will want to import the EntityManagerFactory, which makes the EntityManager. Also, you must import the EntityManager class itself. (The Persistence class must also be imported)

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
```

Next, you should set up those variables. Here is an example.

```
@javax.persistence.PersistenceContext private EntityManagerFactory emf = null;
private EntityManager em = null;
private String persistence_unit = "domain";
/*The persistence_unit string was set up for use by the EntityManagerFactory as you
will soon see (we had a package for our domain objects named domain)*/
```

Now, with the EntityManager set up. You can establish the connection to the database. Here is how it was done in a method.

```
public void connectToDatabase(){
	try{
		this.emf_
=Persistence.createEntityManagerFactory(this.persistence_unit);
		this.em = this.emf.createEntityManager(); // Retrieve an application
managed entity manager
	}
	catch(Throwable t){
		t.printStackTrace();
		return;
	}
}

You can also check to see if the connection is establish. Here is an example method
of how it is done.

public boolean isConnected(){
	return (this.em != null && this.emf != null && this.em.isOpen());
}
```

Also, if you would like to close the connection to the database you can do it as shown here:

```
public void closeConnectionToDatabase()
{
```

```
        this.em.close();
        this.emf.close();}
```

Those are the basics you will need to get the database to where you can start storing and retrieving.

Storing

Using Hibernate allows you to store objects. This is extremely helpful since we are doing object-oriented programming. Before you start making changes to be able to store an object, you must make some changes within the class(object) that you want to store.

I will show examples from our code(disregard the red lines, it comes from my computer just being stupid, but it all does work) that will give an good display of what is happening. First, we had a User class that served as a super class for other specific types of users. Here is how we set it up.

```
import javax.persistence.Embedded;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

import org.hibernate.annotations.GenericGenerator;

/**
 *
 * @author James Rundquist
 */

@Entity
@Table(name="users")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class User{

    @Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy = "increment")
    private long id;                // Unique id of the user

    private String email;       // Email for each user
    private String password;    // Hashed password of the user

    @Embedded
    private Name name;          // Name of the user

    private UserType type;      // Type the user is [ Enum type ]
    private boolean suspended;  // Whether or not they are suspended from the system
    private boolean loggedIn;   // Whether or not they are logged into the system currently
```

Keep note of the usage of the id. Notice how it is configured to automatically increment. This avoids a lot of problems in storage and retrieval from the database.

Now, I will show an example of a subclass of user called Patient.

```
4
5  import javax.persistence.CascadeType;
6  import javax.persistence.Entity;
7  import javax.persistence.Inheritance;
8  import javax.persistence.InheritanceType;
9  import javax.persistence.OneToMany;
10 import javax.persistence.OneToOne;
11 import javax.persistence.Query;
12 import javax.persistence.Table;
13 import javax.persistence.Transient;
14 import javax.persistence.CascadeType;
15
16 import edu.gatech.volt2.drdoommgt.system.DataStore;
17
18
19 @Entity
20 @Table(name="patients")
21 public class Patient extends User {
22     @OneToOne (cascade= {CascadeType.ALL})
23     private PatientInfo info;
24     @OneToMany(cascade = {CascadeType.ALL})
25     private List<Appointment> appointments;
26
```

*OneToOne means a single Patient object would map to one of whatever (a single object)
*OneToMany means a single Patient object would map to many of whatever(eg. a collection)
*Transient is used as a tagged when you DO NOT want to store part of an object into the database
*Cascade deals with what happens to that tagged instance when the Patient object is updated or deleted.

Here are details to the classes that are instances to the Patient class shown above.

*PatientInfo

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Transient;
import javax.persistence.OneToMany;
import javax.persistence.CascadeType;


import javax.persistence.GeneratedValue;
import org.hibernate.annotations.GenericGenerator;


/**
 * This class represents the information of a patient
 *
 * @author Perron Jones
 */

@Entity
@Table(name="patients_info")
public class PatientInfo
{
    @Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy = "increment")
    private long id;
    private String gender;
    private Calendar birthday;
    private Address address;
    private String phone;
    @Transient
    private MedicalInfo medInfo;
    @OneToMany(cascade = {CascadeType.ALL})
    private List<TreatmentRecord> medHistory;
```

*Appointment

```
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

import org.hibernate.annotations.GenericGenerator;

/**
 * This class represents an appointment of the hospital
 *
 * @author Perron Jones
 */

@Entity
@Table(name="appointments")
public class Appointment
{
    @Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy = "increment")
    private long id;                // Unique id of the appointment

    private String patientName;
    private String reason;
    @Temporal(TemporalType.TIMESTAMP)
    private Calendar requestedDate;

    @ManyToOne(cascade = CascadeType.PERSIST)
    private Doctor requestedDoc;

    private boolean confirmed;
```

Now that we have the class that we want to store set up, I can show you an example of how it is properly saved into the database. Remember the EntityManager em that we created earlier. The saving is very simple. Here is an example of how the Patient object shown above is stored into our database using Hibernate.

```
public void saveUser(Patient user)
{
      this.em.getTransaction().begin();
      this.em.persist(user);
      this.em.getTransaction().commit();
}
```

That pretty much displays the basics that you will need to start up and save an object into a database using Hibernate.

Retrieving

Now I would like to discuss getting the object from the database using Hibernate. This process is much less tedious I will refer back to the Patient example used when I was showing how to store an object.

Recall the id tag and instance shown in the earlier. This id was eventually set by the GeneratedValue tag that you may have noticed earlier. If you know this tag, you can simply retrieve an object like this.

```
public void retrieveUser(Long id)
{
	User userx= null;
	try
	{
		this.em.getTransaction().begin();
		userx = em.find(User.class, id);
		this.em.getTransaction().commit();
	}
	finally
	{
		//do nothing
	}
	return userx;
}
```

Most of the time is not that simple, sometimes you may have to create a query.
This is when we should import Query like this:

```
import javax.persistence.Query;
```

Also, import the following in the case that you want to use them in try-catch statements:

```
import javax.persistence.NoResultException;
import javax.persistence.NonUniqueResultException;
```

Now for the actual searching part. The following example will show you how the query is used to get a list of patients from the database using Hibernate.

```
public ArrayList<Patient> getAllPatients()
{

    Query findAllQuery=null;

    findAllQuery = this.em.createQuery("from Patient where id > :id and type = :type");
    long limit= -1;
    UserType type= UserType.PATIENT;
    findAllQuery.setParameter("id", limit);
    findAllQuery.setParameter("type", type);

    ArrayList<Patient> users= null;


    try
    {
        users = (ArrayList<Patient>) findAllQuery.getResultList();
    }
    catch(NoResultException e)
    {
        System.out.println("There are no patients.");
    }
    catch(NonUniqueResultException g)
    {
        //System.out.println("Multiple emails.");
        //List<>foundUser = (User) findByEmailQuery.getSingleResult();
    }


    return users;
}
```

Notice:
findAllQuery = this.em.createQuery("from Patient where id > :id and type = :type");

*"from Patient" indicates the class/table.
*"where id > :id" compares the id in the database table to the id(:id) indicated in a variable following. (the same goes for type)
*the compare id's are finally set by findAllQuery.setParameter("id", limit) and so on..
*the findAllQuery.getResultList() returns a list of the objects that meet the comparison/query search criteria.

Here is another example, but it retrieves a single Patient using a string of the Patient inherited from its super class, User.

```
public Patient findPatientByEmail(String email)
{
    Patient foundUser= null;
    Query findByEmailQuery=null;
    try
    {
        findByEmailQuery = this.em.createQuery("from Patient where email LIKE :email");
        findByEmailQuery.setParameter("email", email);

        foundUser = (Patient) findByEmailQuery.getSingleResult();
        long userId = foundUser.getId();
        foundUser = this.em.find(Patient.class, userId);

    }
    catch(NoResultException e)
    {
        System.out.println("Email address is not found.");
    }
    catch(NonUniqueResultException g)
    {
        System.out.println("Multiple emails.");
        //List<>foundUser = (User) findByEmailQuery.getSingleResult();
    }
    return foundUser;
}
```

Notice here how findAllQuery.getSingleResult is used here to obtain only a single object that meets the comparison/query search criteria.
*if you notice how the word "LIKE" is used in the em.createQuery method call. This is called Wildcard searching. This means that if what you are searching for is similar to something found in the database, it will get that object from the database. This deletes relying on everything to match exactly.

Those are the main points of retrieving objects from the database using Hibernate.

Removal
This part is actually quite simple. The object contains an id that is automatically generated, so this id is used to find the object and remove it from the database. Here is an example of removing the patient.

```
public void removeUser(Patient user)
{
	try
	{
		this.em.getTransaction().begin();
		User userx = em.find(Patient.class, user.getId());
		this.em.remove(userx);
		this.em.getTransaction().commit();
	}
	finally
	{
		//do nothing
	}
}
```

This covers pretty much what you need to know to atleast give you a great start at using Hibernate to deal with objects and databases after it is set up for use.

Useful links

Java2s
http://www.java2s.com/Code/Java/Hibernate/Query.htm

JBoss
http://docs.jboss.org/hibernate/search/3.4/reference/en-US/html_single/#d0e364

Dr. Winston Prakash
http://www.winstonprakash.com/articles/netbeans/JPA_Add_Update_Delete.html