Hibernate – II

Hibernate Crud

Table of Contents

Hibernate Core API's

1. org.hibernate.cfg.Configuration
2. org.hibernate.SessionFactory
3. org.hibernate.cfg.Session
4. org.hibernate.cfg.Transaction
5. org.hibernate.cfg.Query
6. org.hibernate.cfg.Criteria

Frequently Used Methods of Configuration

Methods	Description
Configuration configure( )	Use the mappings and properties specified in an application resource named **hibernate.cfg.xml.**
Configuration configure(String resource)	Use the mappings and properties specified in the given application resource.
SessionFactory buildSessionFactory( )	Instantiate a new **SessionFactory**, using the properties and mappings in this configuration.
Configuration addAnnotatedClass(Class clazz)	It adds the annotated class to the configuration.
Configuration addFile(File xmlFile)	Read mappings from a particular XML file.
Configuration addFile(String xmlFile)	Read mappings from a particular XML file.
Configuration addDirectory(File dir)	Read all mapping documents from a directory tree.
Configuration addResource(String resource)	Read mappings as a application resourceName
Configuration addProperties(Properties prop)	Set the given properties.

Frequently Used Methods of SessionFactory

Methods	Description
void close( )	Destroy this SessionFactory and release all resources(caches, connection pools, etc).
Session openSession()	Open a Session.
Session getCurrentSession()	Obtains the current session.

Frequently Used Methods of Session

Methods	Description
Serializable **save**(Object object)	Persist the given transient instance, first assigning a generated identifier.
void **persist**(Object object)	Make a transient instance persistent.
Object **get**(String entityName, Serializable id)	Return the persistent instance of the given named entity with the given identifier, or null if there is no such persistent instance.
void **load**(Object object, Serializable id)	Read the persistent state associated with the given identifier into the given transient instance.
void **update**(String entityName, Object object)	Update the persistent instance with the identifier of the given detached instance.
Object **merge**(Object object)	Copy the state of the given object onto the persistent object with the same identifier.
void **saveOrUpdate**(Object object)	Either save(Object) or update(Object) the given instance, depending upon resolution of the unsaved-value checks.
void **delete**(Object object)	Remove a persistent from the datastore.
void **close**()	End the session by releasing the JDBC connection and

	cleaning up.
void **flush**()	Force this session to flush.
boolean **isOpen**()	Check if the session is still open.
void **clear**()	Completely clear the session.
boolean **contains**(Object object)	Check if this instance is associated with this Session.
Connection **disconnect**()	Disconnect the session from its underlying JDBC connection.
String **getEntityName**(Object object)	Return the entity name for a persistent entity.

Mostly used Hibernate-mapping attributes

```
<hibernate-mapping
        Schema =”schemaName”
        default-lazy=”true|false”
        auto-import=”true|false”
        package =”package.name” >
</hibernate-mapping>
```

Schema (optional)	The name of a database schema.
default-lazy (optional – defaults to true)	The default value for unspecified lazy attributes of class and collection mappings.
auto-import(optional – defaults to true)	Specifies whether we can use unqualified class names of classes in this mapping in the query language.
package (optional)	Specifies a package prefix to use for unqualified class names in the mapping document.

Mostly frequently class element are used

```
<class
      name =”ClassName”
      table =”tableName”
      discriminator-value =”discriminator_value”
      proxy =”ProxyInterface”
      where =”arbitrary sql where condition”
      batch-size =”N”
      lazy =”true|false”
      entity-name =”EntityName” >
</class>
```

name (optional)	The fully qualified Java class name of the persistent class or interface. If this attribute is missing, it is assumed that the mapping is for a non-POJO entity.
table (optional – defaults to the unqualified class name)	The name of its database table.
discriminator-value (optional – defaults to the class name)	A value that distinguishes individual subclasses that is used for polymorphic behavior. Acceptable values include null and not null.
proxy (optional)	Specifies an interface to use for lazy initializing proxies. You can specify the name of the class itself.
dynamic-update (optional – defaults to false)	Specifies that UPDATE SQL should be generated at runtime and can contain only those columns whose values have changed.

dynamic-insert (optional – defaults to false)	Specifies that INSERTSQL should be generated at runtime and can contain only those columns whose values are not null.
where (optional)	Specifies an arbitrary SQL WHERE condition to be used when retrieving objects of this class.
batch-size (optional – defaults to 1);	Specifies a "batch size" for fetching instances of this class by identifier.
lazy (optional)	Lazy fetching can be disabled by setting lazy="false".
entity-name	optional – defaults to the class name

Frequently used element with id

```
<id
      name="propertyName"
      type="typename"
      column="column_name"
      unsaved-value="null | any | none | undefined | id_value"
      access="field | property | ClassName">
      <generator class="generatorClass"/>
</id>
```

name (optional)	The name of the identifier property.
type (optional)	A name that indicates the Hibernate type
column (optional – defaults to the property name)	The name of the primary key column
name (optional – defaults to property)	The strategy Hibernate should use for accessing the property value.

Create, Retrieve, Update and Delete (CRUD) Operations

Let's see the below application we will do all the CRUD operation

pojo.class

```
package com.Biditvats.domain;

public class Customer {
 private Long id;
 private String firstName;
 private String lastName;
 private String email;
 private Long mobile;

 public Customer() {
          //Do nothing
 }

 //corresponding setter and getter
}
```

Customer.hbm.xml

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

```
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

   <hibernate-mapping>
        <class name="com.Biditvats.domain.Customer" table="CUSTOMER_MASTER2">
        <!-- primary key -->
        <id name="id" column="CUSTOMER_ID">
                <generator class="identity" />
        </id>

        <property name="firstName" column="FIRST_NAME" />
        <property name="lastName" column="LAST_NAME" />
        <property name="email" column="EMAIL" />
        <property name="mobile" column="MOBILE"></property>
        </class>
   </hibernate-mapping>
```

hibernate.cfg.xml

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

<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
        <session-factory>
          <!-- Data Source Details -->
                <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
                <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernatedb</property>
                <property name="hibernate.connection.username">root</property>
                <property name="hibernate.connection.password">password</property>

                <!-- Hibernate Properties -->
                <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
                <property name="hibernate.hbm2ddl.auto">update</property>
                <property name="hibernate.show_sql">true</property>

                <!-- Resource mapping -->
                <mapping resource="Customer.hbm.xml"/>

        </session-factory>
</hibernate-configuration>
```

Difference between save () and persist ()

- Both methods are used for make a transient instance to persistent
- It depends on the generator class

persist( )

- It does not guarantee that the identifier value will be assigned to the persistent instance immediately; the assignment might happen at flush time.
- Does not save the changes to the db outside of the transaction.
- This is useful in long-running conversations with an extended Session/persistence context.

save ( )

- it does guarantee to return an identifier.
- If an INSERT has to be executed to get the identifier (e.g. “identity” generator, not “sequence”), this INSERT happens immediately, no matter if you are inside or outside of a transaction.
- This is a problematic in a long-running conversation with an extended Session/persistence context.

SaveAndPersist.java

```
package com.Biditvats.test;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import com.Biditvats.domain.Customer;

public class SaveAndPersist {

	public static void main(String[] args) {
		Configuration configuration = new Configuration().configure();

		SessionFactory factory = configuration.buildSessionFactory();

		Customer customer1 = new Customer("Bidit","vats","Biditvats@gmail.com",9916712669L);
		Customer customer2 = new Customer("Aditya","vats","Adityavats@gmail.com",8892550034L);

		//Save Customer-1 using save() method, it will return primary-key
		Session session1 = factory.openSession();
		Transaction transaction1 = session1.beginTransaction();
		Long id = (Long) session1.save(customer1); //it will return Primary-key
		transaction1.commit();
		session1.close();

		// Save Customer-1 using persist() method
		Session session2 = factory.openSession();
		Transaction transaction2 = session2.beginTransaction();
		session2.persist(customer2);
		transaction2.commit();
		//session.persist(customer2); //it will give exception
		session2.close();

		System.out.println("Customer-1 ID:" +id);
		System.out.println("Record has been saved Successfully");


	}

}
```

Difference between load () and get ()

The load ( ) and get ( ) methods of Session provide a way of retrieving a persistent instance if you know its identifier.

load ( )

- If the class is mapped with a proxy, load() just returns an uninitialized proxy and does not Actually hit the database until you invoke a method proxy.
- Hibernate will prepare some fake object with given identifier value in the memory without hitting the database remaining properties of Product class will not even be initialized.
- It will hit the database only when we try to retrieve the other properties.
- load () will throw an **unrecoverable exception** if there is **no** matching database row.
- This is useful if you wish to create an association to an object without actually loading it from the database.

get ( )

- If you are not certain that a matching row exists, you should use the get () method which hits the database immediately and retuns null if there is no matching row.

GetAndLoad.java

```
package com.Biditvats.test;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

import com.Biditvats.domain.Customer;


public class GetAndLoad {

	public static void main(String[] args) {
		Configuration configuration = new Configuration().configure();

		SessionFactory factory = configuration.buildSessionFactory();

		/*Get Object using get() method, it performs Early Loading,
		 * if Object does not exist return null*/
		Session session1 = factory.openSession();
		Customer customer1 = session1.get(Customer.class, 1L);
		System.out.println(customer1);
		session1.close();

		/*Get Object using load() method, it performs Lazy Loading,
		 * if Object does not exist throw ObjetNotFoundException,
		 * performs Lazy loading within session only, otherwise
		 * throws LazyInitializationException */
		Session session2 = factory.openSession();
		Customer customer2 = session2.load(Customer.class, 2L);
		System.out.println(customer2);
		session2.close();


	}

}
```

Difference between update () and saveOrUpdate ()

- Both **update()** and **merge()** methods in hibernate are used to convert the object which is in **detached state** into **persistence** state.
- We can update only non-primary key using these methods.

update ( )

- We can update the object in the session only.
- It will check first object is present in current session or not, if present then it will throw exception **"NonUniqueObjectException"** otherwise it will update.

merge ( )

- If we load the object and close the session then, using merge method we can update object within another session.

saveOrUpdate( )

- 1st it will fire **select query,** if id match then it will update the object, otherwise it will fire **insert Query**.

UpdateAndMerge.java

```
package com.Biditvats.test;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import com.Biditvats.domain.Customer;

public class UpdateAndMerge {

	public static void main(String[] args) {
		Configuration configuration = new Configuration().configure();

		SessionFactory factory = configuration.buildSessionFactory();


		/*Session session1 = factory.openSession();
		Transaction transaction1 = session1.beginTransaction();
		Customer customer1 = session1.get(Customer.class, 2L);
		customer1.setEmail("abc@biditvats.com");
		transaction1.commit();
		session1.close();

		Session session2 = factory.openSession();
		Transaction transaction2 = session2.beginTransaction();
		Customer customer2 = session2.get(Customer.class, 2L);
		customer2.setLastName("MNP");
		session2.update(customer1);
		transaction2.commit();
		session2.close();*/
```

```
		Session session1 = factory.openSession();
		Transaction transaction1 = session1.beginTransaction();
		Customer customer1 = session1.get(Customer.class, 2L);
		customer1.setEmail("Adityajha@gmail.com");
		transaction1.commit();
		session1.close();


		Session session2 = factory.openSession();
		Transaction transaction2 = session2.beginTransaction();
		Customer customer2 = session2.get(Customer.class, 3L);
		customer2.setLastName("JHA");
		session2.merge(customer1);
		transaction2.commit();
		session2.close();


	}

}
```

UpdateandSaveOrUpdate.java

```
package com.Biditvats.test;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import com.Biditvats.domain.Customer;


public class UpdateandSaveOrUpdate {

	public static void main(String[] args) {
		Configuration configuration = new Configuration().configure();

		SessionFactory factory = configuration.buildSessionFactory();

		//It will update ID-1 Record- execute UPDATE Query
		/*Session session1 = factory.openSession();
		Transaction transaction1 = session1.beginTransaction();
		Customer customer1 = session1.get(Customer.class, 1L);
		customer1.setEmail("support@biditvats.com");
		session1.update(customer1);
		transaction1.commit();
		session1.close();*/

		//It will create new Record- Execute INSERT Query
		/*Session session2 = factory.openSession();
		Transaction transaction2 = session2.beginTransaction();
		Customer customer2 = new Customer("Xyz", "Kumar", "xyz@biditvats.com", 8892550034L);
		session2.saveOrUpdate(customer2);
		transaction2.commit();
		session2.close();*/
```

```
		//It will update id-2 Record- Execute UPDATE Query
		Session session3 = factory.openSession();
		Transaction transaction3 = session3.beginTransaction();
		Customer customer3 = new Customer("Vicky","Jha","Vickyjha@gmail.com",7728902579L);
		customer3.setId(2L);
		session3.saveOrUpdate(customer3);
		transaction3.commit();
		session3.close();

	}

}
```

delete ( )

- Delete a row (Object) from the database just like using delete query in the jdbc program.

Delete.java

```
package com.Biditvats.test;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import com.Biditvats.domain.Customer;

public class Delete {

	public static void main(String[] args) {
		Configuration configuration = new Configuration().configure();

		SessionFactory factory = configuration.buildSessionFactory();

		Session session = factory.openSession();
		Transaction transaction = session.beginTransaction();
		Customer customer = session.get(Customer.class, 4L);
		session.delete(customer);
		transaction.commit();
		session.close();

		System.out.println("Record has been deleted successfully");


	}

}
```