Configuring Hibernate

Configuring Hibernate

REVATURE

- To connect with a database in Hibernate application, you need to set various properties regarding driver class, user name, and password in the hibernate.cfg.xml file
- The structure of the hibernate.cfg.xml file is given in the following code snippet:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
    <session-factory>
    <property name="connection.username">SYSTEM</property>
              .......
              .......
              .......
        <mapping resource="Mapped_File.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Configuring Hibernate(Contd.)

REVATURE

- In the preceding code snippet, the hibernate.cfg.xml file contains the following tags:
  - `<?xml>`: Defines the version and encoding type used for the XML document.
  - `<DOCTYPE>`: Specifies the Document Type Definition (DTD) for the XML elements. DTD specifies the grammar rule for the XML document. For example, DTD for hibernate.cfg.xml file is specified as http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd.
  - `<hibernate-configuration>`: Specifies all the configuration details that the application uses to communicate with the underlying database.
  - `<session-factory>`: Contains the database and application specific properties that a Session object uses to establish a communication link between the application and the database.
  - `<property>`: Defines the various properties that are required to connect with the database.
  - `<mapping>`: Specifies the name of the Hibernate mapping file that defines the mapping of a database table to a class.

Configuring Hibernate(Contd.)

REVATURE

- The following properties are commonly set under the `<property>` tag:
  - **JDBC properties**: The JDBC properties are used to connect with a relational database. The following JDBC properties are commonly used in a Hibernate configuration file:
    - `hibernate.connection.driver_class`: Specifies the database specific driver name.
    - `hibernate.connection.url`: Specifies the complete path along with the port number and name of a database that needs to be connected with the application.
    - `hibernate.connection.username`: Specifies the user name that is used to connect with a particular database.
    - `hibernate.connection.password`: Specifies the password that is used to connect with a particular database.

Configuring Hibernate(Contd.)

REVATURE

- **Hibernate configuration properties**: The Hibernate configuration properties control the behavior of Hibernate at runtime. The following configurations are commonly used in the Hibernate configuration file:
  - `hibernate.dialect`: Specifies the database that is used to communicate with the application. It accepts a class name corresponding to the database used in the application. For example, to set the dialect property for the Oracle database, you need to use the following code snippet:

```
<property name="dialect">
org.hibernate.dialect.OracleDialect </property>
```

  - `hibernate.show_sql`: Specifies that the SQL statements are written on the console during the execution of the application. It helps in identifying errors during execution and improving the query performance. The following code snippet is used to set the hibernate.show_sql property:

```
<property name = "hibernate.show_sql">
true</property>
```

Configuring Hibernate(Contd.)

REVATURE

- The following code illustrates how to configure Hibernate with MySQL database:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
 <session-factory>
 <property name="hibernate.dialect"> org.hibernate.dialect.MySQLDialect </property>
 <property name="hibernate.connection.driver_class"> com.mysql.jdbc.Driver </property>
 <!-- Assume test is the database name -->
 <property name="hibernate.connection.url"> jdbc:mysql://localhost/test </property>
 <property name="hibernate.connection.username"> root </property>
 <property name="hibernate.connection.password"> root123 </property>
 <!-- List of XML mapping files -->
 <mapping resource="Employee.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Creating Hibernate Session

REVATURE

- In a Hibernate application, a session acts as a pipeline between the application and the database.
- To store the application data in the database, you need to create a `Session` object.
- A session object in an application is created by using a `SessionFactory` object.
- To create a `SessionFactory` object, the application needs various database specific configuration settings.
- Hibernate specifies the configuration settings and information about the mapping document in an XML file named hibernate.cfg.xml.
- Hibernate provides the `org.hibernate.cfg.Configuration` class that allows the application to specify the configuration properties and mapping documents to create a `SessionFactory` object.

Creating Hibernate Session(Contd.)

REVATURE

- The `Configuration` class provides the following methods to specify the configuration properties and mapping documents to create a `SessionFactory` object.
  - `configure()`: Loads the hibernate.cfg.xml file and initializes the object of the `Configuration` class with the mappings and configuration properties specified in this file.
  - `getProperties()`: Is used to fetch the properties for the configuration that are configured using the `configure()` method.
  - `applySettings()`: Uses the `ServiceRegistryBuilder` object to apply the settings fetched using the `getProperties()` method. It takes the output of the `getProperties()` method as its parameter.
  - `buildSessionFactory()`: Uses the `Configuration` object returned by the `configure()` method to instantiate a new `SessionFactory` object. It takes `ServiceRegistry` object as its parameter. This object carries the configuration settings that are to be applied.

Creating Hibernate Session(Contd.)

REVATURE

- The following code snippet illustrates how to create object of the `SessionFactory` interface:

```
private static final SessionFactory sessionFactory;
Configuration configuration = new Configuration().configure();
ServiceRegistryBuilder registry = new ServiceRegistryBuilder();
registry.applySettings(configuration.getProperties());
ServiceRegistry serviceRegistry = registry.buildServiceRegistry();
sessionFactory=configuration.buildSessionFactory(serviceRegistry);
```

- You can use the `getSessionFactory()` method to access the Hibernate session in your application, as shown in the following code snippet:

```
Session session =
HibernateUtil.getSessionFactory().getCurrentSession();
```

Creating Hibernate Session(Contd.)

REVATURE

- The main function of the `Session` object is to offer create, read, and delete operations for instances of mapped entity classes. Instances may exist in one of the following three states at a given point in time:
  - **transient**: A new instance of a persistent class which is not associated with a `Session` and has no representation in the database and no identifier value is considered transient by Hibernate.
  - **persistent**: You can make a transient instance persistent by associating it with a `Session`. A persistent instance has a representation in the database, an identifier value and is associated with a `Session`.
  - **detached**: Once we close the Hibernate `Session`, the persistent instance will become a detached instance.

Creating Hibernate Session(Contd.)

REVATURE

- The following table contains some of the common methods available in the `Session` object.

Method	*Description*
`Transaction beginTransaction()`	Begin a unit of work and return the associated Transaction object.
`void cancelQuery()`	Cancel the execution of the current query.
`void clear()`	Completely clear the session.
`Connection close()`	End the session by releasing the JDBC connection and cleaning up.
`Serializable getIdentifier(Object object)`	Return the identifier value of the given entity as associated with this session.
`Query createFilter(Object collection, String queryString)`	Create a new instance of Query for the given collection and filter string.
`Query createQuery(String queryString)`	Create a new instance of Query for the given HQL query string.
`SQLQuery createSQLQuery(String queryString)`	Create a new instance of SQLQuery for the given SQL query string.
`void delete(Object object)`	Remove a persistent instance from the datastore.

Creating Hibernate Session(Contd.)

REVATURE

Method	*Description*
`void delete(String entityName, Object object)`	Remove a persistent instance from the datastore.
`Session 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.
`SessionFactory getSessionFactory()`	Get the session factory which created this session.
`Transaction getTransaction()`	Get the Transaction instance associated with this session.
`boolean isConnected()`	Check if the session is currently connected.
`boolean isOpen()`	Check if the session is still open.
`Serializable save(Object object)`	Persist the given transient instance, first assigning a generated identifier.
`void saveOrUpdate(Object object)`	Either save(Object) or update(Object) the given instance.
`void update(Object object)`	Update the persistent instance with the identifier of the given detached instance.
`void update(String entityName, Object object)`	Update the persistent instance with the identifier of the given detached instance.

Creating Hibernate Session(Contd.)

REVATURE

- A `Session` instance is serializable if its persistent classes are serializable. A typical transaction should use the following idiom:

```
Session session = factory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    // do some work
    ...
    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    e.printStackTrace();
}finally {
    session.close();
}
```

Configuring Mapping Properties

REVATURE

- Hibernate allows you to map the classes and their properties with the tables of the database in a configuration file.
- The name of this mapping file has the following syntax: `<persistent_class_name>.hbm.xml.`
- The Hibernate mapping file overcomes the problem of difference between the data types used in classes and columns of the tables.
- The Hibernate mapping file contains mapping information of the class data types with the database specific data types.
- The Hibernate mapping file acts as a medium of communication between the persistent classes and the database tables.
- The Hibernate mapping file enables the communication by mapping the data types used in the persistent classes with the database specific data types by using the Hibernate types.

Configuring Mapping Properties(Contd.)

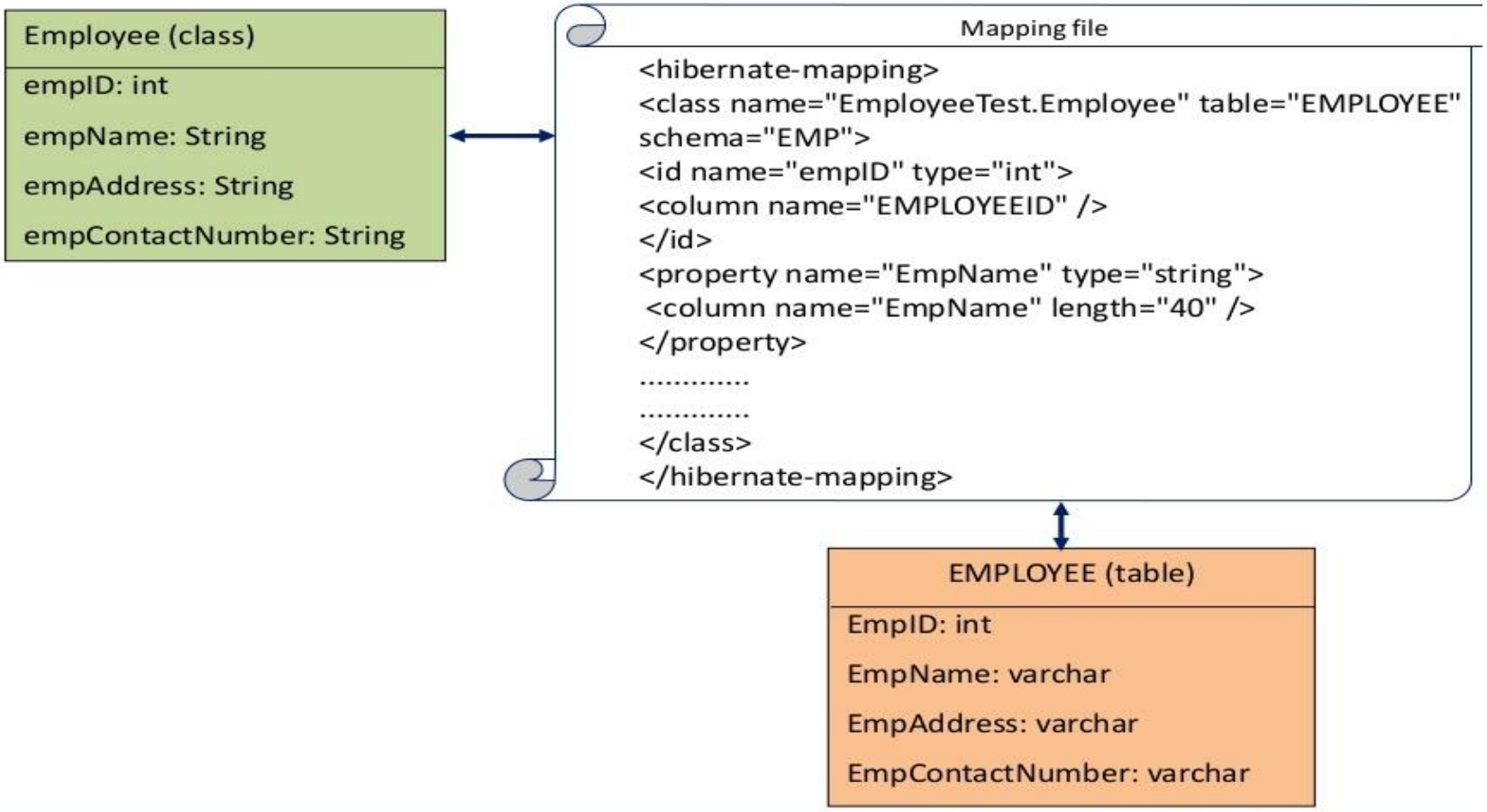
REVATURE

- Hibernate Types:
  - Provide an abstract representation of the underlying database types.
  - Allow you to develop the application without worrying about the target database and the data types supported by it.
  - Provide you the flexibility to change the database without changing the application code.
  - The following table lists some of the Hibernate built-in types along with the corresponding Java and SQL data types.

Hibernate Type	Java Type	SQL Type
integer, long, short	Integer, int, long short	NUMERIC, NUMBER, INT, or other vendor specific type.
character	char	CHAR
big_decimal	java.math.BigDecimal	NUMERIC, NUMBER
float, double	float, double	FLOAT, DOUBLE
boolean, yes_no, true_false	java.lang.Boolean, boolean	BOOLEAN, INT
string	java.lang.String	VARCHAR, VARCHAR2
date, time, timestamp	java.util.Date	DATE, TIME, and TIMESTAMP

Configuring Mapping Properties(Contd.)

- The following figure shows the communication of a persistent class with a database table through the Hibernate mapping file.



Configuring Mapping Properties(Contd.)

REVATURE

- While mapping the persistent class objects with the tables in the database, Hibernate classifies objects in the following types:
  - **Entity type**: Is an independent entity and has its own primary key in a database table.
  - **Value type**: A value type object does not have an identifier. Therefore, a value type object depends on an entity type object for its existence.

Mapping Beans with the Database

REVATURE

- Hibernate provides you with the following mapping techniques to create and map the persistent classes of the application with the database tables to retrieve and store data:
  - Using the Hibernate mapping file
  - Using annotations

Using the Hibernate Mapping File

REVATURE

- In this technique, you need to set values for the various mapping elements in the Hibernate mapping file.
- The Hibernate mapping file contains the following commonly used elements:
  - `<DOCTYPE>`: Specifies the type of the current XML document and the name of the Document Type Definition (DTD).
  - `<hibernate-mapping>`: Refers to the root mapping element that contains other mapping elements, such as <class>, <id>, and <generator>.
  - `<class>`: Refers to the Java class that is mapped with a database table and it contains the following commonly used attributes:
    - `name`: Specifies the fully qualified class name of the persistent class.
    - `table`: Specifies the name of the database table to which the Java class is mapped.
    - `Mutable`: Specifies whether the Java class is mutable. A class whose objects are declared as read-only objects is referred to as the mutable class.
    - `schema`: Specifies the schema of the database being mapped with the Java class.

Using the Hibernate Mapping File(Contd.)

REVATURE

- <id>: Refers to the primary key column of a database table. The commonly used attributes of the <id> element are:
  - name: Specifies the property name of the Java class that is mapped with the primary key column of the database table.
  - type: Specifies the Hibernate type that is used to map a property of the Java class with the primary key column of a database table.
  - column: Specifies the name of the primary key column of the database table.
- <generator>: Refers to a Java class used to generate unique identifiers for the instances of the persistent class in a database table. This element defines the class name with the help of the class attribute. The commonly used aliases as the value of the class attribute are:
  - assigned:
  - Increment
  - Identity
  - Sequence
  - native

Using the Hibernate Mapping File(Contd.)

REVATURE

- `<property>`: Is used to map the properties of the Java class with the specific columns of a database table. This element has the following commonly used attributes:
  - `name`: Specifies the name of a property of the Java class that is being mapped with a column of the database table.
  - `type`: Specifies the Hibernate type that is used to map a property of the Java class with a column of the database table.
  - `column`: Specifies the name of a column of the database table that is mapped to a property of the Java class.
  - `length`: Specifies the maximum number of characters that a column, which is being mapped with a property of the Java class, can store.
- `<column>`: Is used to specify a column that is to be mapped with a property of the Java class. It is used within the `<property>` tag. However, this tag can be replaced by using the column attribute of the `<property>` tag. The `<column>` tag has the following commonly used attributes:
  - `name`: Specifies the name of the column in the database table.
  - `length`: Specifies the maximum number of characters that a column can store.

Using the Hibernate Mapping File(Contd.)

REVATURE

- The following code snippet map the `Employee` class with the `EMPLOYEE` table in a database.

```
<?xml version="1.0"?>
<!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="Employee" table="EMPLOYEE" schema="EMP">
      <id name="empid" type="int">
         <column name="EMPID" />
         <generator class="assigned" />
      </id>
      <property name="empname" type="string">
         <column name="EMPNAME" length="20" />
      </property>
      <property name="empaddress" type="string">
         <column name="EMPADDRESS" length="40" />
      </property>
   </class>
</hibernate-mapping>
```

Using Annotations

REVATURE

- In this technique, you map the elements of the table in the database with the bean using annotations. The following annotations are commonly used while mapping:
  - `@Entity`: Is used to specify a class as an entity bean. It is contained inside the `javax.persistence` package.
  - `@Table`: Is used to specify the name of the table.
  - `@Column`: Is used to specify the column name. It contains the following attributes:
    - `name`: Specifies the name of the column.
    - `length`: Specifies the length of the column.
    - `nullable`: Specifies the column to be NOT NULL.
    - `unique`: Specifies the column to contain unique values.
  - `@Id`: Is used to define the column as an identifier or a primary key.
  - `@GeneratedValue`: This annotation is used to specify the identifier generation strategy. The identifier generation strategies are of the following types:
    - `AUTO`
    - `IDENTITY`
    - `SEQUENCE`
    - `TABLE`

Using Annotations(Contd.)

REVATURE

- Consider a scenario where you are developing a Web page that accepts the employee id of an employee from the user and displays the details about that employee. The employee details need to be fetched from the EMPLOYEE table stored in the database.
- The Employee table displays the details in the database, as shown in the following figure.

EmpID	EmpName	EmpAddress	EmpContact
238	JAMES JONES	California	7728374859
239	JONAH DOMES	Boston	7528384559
240	RICHARD PARKER	New York	7238495867

Using Annotations(Contd.)

REVATURE

- The following code snippet displays the EMPLOYEE table mapped with the managed bean.

```
import java.io.Serializable;
import javax.inject.Named;
import javax.enterprise.context.RequestScoped;
import javax.persistence.*;
@Named("employee")
@RequestScoped
@Entity
@Table(name="Employee")
public class Employee implements Serializable
{
@Id @GeneratedValue
@Column(name="EmpID")
private int empID;
@Column(name="EmpName")
private String empName;
@Column(name="EmpAddress")
private String empAddress;
@Column(name="EmpContact")
private long empContact;
.....
.....
public Employee() {
    }
}
```