20. Hibernate Configuration with Annotations

Hibernate Development Process:

1. Add Hibernate Configuration file
2. Annotate Java Class
3. Develop Java Code to perform database operations

1) Hibernate Configuration file:

The configuration file basically tells Hibernate how to connect to the database. Hibernate use JDBC in the background for communication with the database.

First create a project “20-hibernate-configuration(Annotations)” and paste the following “hibernate.cfg.xml” file in the “src” directory.

File: hibernate.cfg.xml:

<!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>

<!-- JDBC Database connection settings -->

<property name=*"connection.driver\_class"*>com.mysql.cj.jdbc.Driver

</property>

<property name=*"connection.url"*>

jdbc:mysql://localhost:3306/hb\_student\_tracker?useSSL=false&amp;serverTimezone=UTC

</property>

<property name=*"connection.username"*>hbstudent</property>

<property name=*"connection.password"*>hbstudent</property>

<!-- JDBC connection pool settings ... using built-in test pool -->

<property name=*"connection.pool\_size"*>1</property>

<!-- Select our SQL dialect -->

<property name=*"dialect"*>org.hibernate.dialect.MySQLDialect</property>

<!-- Echo the SQL to stdout -->

<property name=*"show\_sql"*>true</property>

<!-- Set the current session context -->

<property name=*"current\_session\_context\_class"*>thread</property>

</session-factory>

</hibernate-configuration>

Session-Factory (<session-factory> </session-factory>):

SessionFactory is an interface. SessionFactory can be created by providing Configuration object, which will contain all DB related property details pulled from either hibernate.cfg.xml file or hibernate.properties file. SessionFactory is a factory for Session objects.

We can create one SessionFactory implementation per database in any application. If your application

is referring to multiple databases, then we need to create one SessionFactory per database.

The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use. The SessionFactory is a thread safe object and used by all the threads of an application.

Connection Pooling:

Connection pooling is a technique to open/prepare/close connections. A connection pooling mechanism is a piece of software (component), to which we delegate the function of managing connections. Our application would just ask for a connection, use it, and deliver it back to the pool. The component is responsible for opening N connections and leave them ready for when our application asks. If a connection is stale, the pooling mechanism would then close it and reopen a new one. This represents a better usage of connections, as we don't need to wait for the connection to be established during the actual execution of our code and we don't have to worry about stale connections.

Hibernate doesn't really ship any real connection pooling mechanism. It provides an internal connection manager, which is very rudimentary. The reason is simple: almost (if not all) Application Servers (like JBoss AS) and Servlet Containers (like Tomcat) provides a connection pooling mechanism by default. Thus, our application doesn’t have to worry about the details about it. It just asks the AS for a connection.

There are only two cases where you need to worry about connection pooling.

1. When we are dealing with a standalone application (which doesn't run inside a container)
2. If we are really expert in connection pooling and none of the existing suits your needs.

most people that uses an "external" connection pooling do so for lack of knowledge about connection pooling and lack of knowledge about their container.

connection.pool\_size:

Limits the number of connections waiting in the Hibernate database connection pool.

connection.pool\_size indicates the maximum number of pooled connections. So, it is better to keep it at a logical count. It depends on your application and DB how much it can handle. 10 is a reasonable count that will typically use as it is sufficient for most cases.

SQL Dialects in Hibernate:

The dialect specifies the type of database used in hibernate so that hibernate generate appropriate type of SQL statements. Every database has its own dialect property. For connecting any hibernate application with the database, it is required to provide the configuration of SQL dialect.

Syntax of SQL Dialect:

<property name=*"dialect"*>org.hibernate.dialect.MySQLDialect</property>

List of SQL Dialects:

|  |  |
| --- | --- |
| REBMS | Dialect |
| Oracle (any version) | org.hibernate.dialect.OracleDialect |
| Oracle10g | org.hibernate.dialect.Oracle10gDialect |
| MySQL | org.hibernate.dialect.MySQLDialect |
| MySQL with InnoDB | org.hibernate.dialect.MySQLInnoDBDialect |
| MySQL with MyISAM | org.hibernate.dialect.MySQLMyISAMDialect |
| DB2 | org.hibernate.dialect.DB2Dialect |
| DB2 AS/400 | org.hibernate.dialect.DB2400Dialect |
| DB2 OS390 | org.hibernate.dialect.DB2390Dialect |
| Microsoft SQL Server | org.hibernate.dialect.SQLServerDialect |
| Sybase | org.hibernate.dialect.SybaseDialect |
| Sybase Anywhere | org.hibernate.dialect.SybaseAnywhereDialect |
| PostgreSQL | org.hibernate.dialect.PostgreSQLDialect |
| SAP DB | org.hibernate.dialect.SAPDBDialect |
| Informix | org.hibernate.dialect.InformixDialect |
| HypersonicSQL | org.hibernate.dialect.HSQLDialect |
| Firebird | org.hibernate.dialect.FirebirdDialect |

“show\_sql = true”:

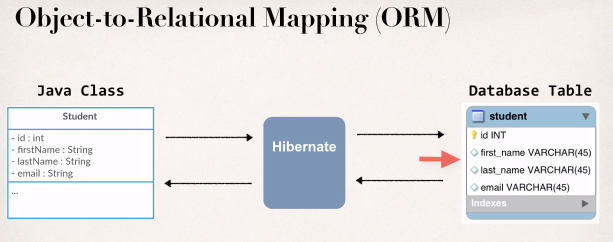
Hibernate has a built-in function to enable the logging of all the generated SQL statements to the console. We can enable it by add a “show\_sql” property in the Hibernate configuration file “hibernate.cfg.xml“. This function is good for basic troubleshooting, and to see what’s Hibernate is doing in the background.

current\_session\_context\_class:

The “current\_session\_context\_class” describe how the hibernate session should be bound. Although hibernate supports "thread", "jta", and "managed" as legal values for this parameter, in Viewpoint this should be set to "thread". "thread" causes sessions to be bound to threads. "jta" binds sessions to JTA transactions and "managed" causes the responsibility for managing session scope, start, and end to the application.

2) Annotate Java Class:

First, we have to create an Entity class that map with a Database table. That’s actually a POJO class (have private field and public getter and setter method).



File: Customer.java:

**package** com.ruhul.entity;

**import** javax.persistence.\*;

@Entity

@Table(name = "student")

**public** **class** Student {

@Id

@Column(name = "id")

**private** **int** id;

@Column(name = "first\_name")

**private** String firstName;

@Column(name = "last\_name")

**private** String lastName;

@Column(name = "email")

**private** String email;

**public** Student() {

}

**public** Student(**int** id, String firstName, String lastName , String email) {

**this**.id = id;

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.email = email;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getFirstName() {

**return** firstName;

}

**public** **void** setFirstName(String firstName) {

**this**.firstName = firstName;

}

**public** String getLastName() {

**return** lastName;

}

**public** **void** setLastName(String lastName) {

**this**.lastName = lastName;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

@Override

**public** String toString() {

**return** "Student [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName + ", email=" + email + "]";

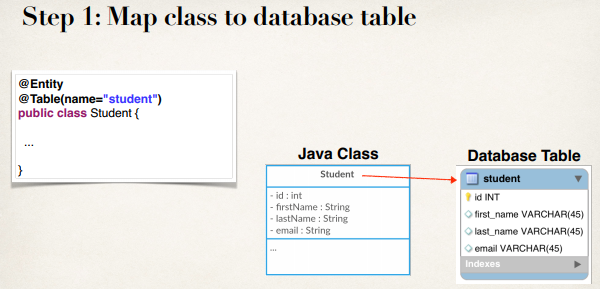
}

}

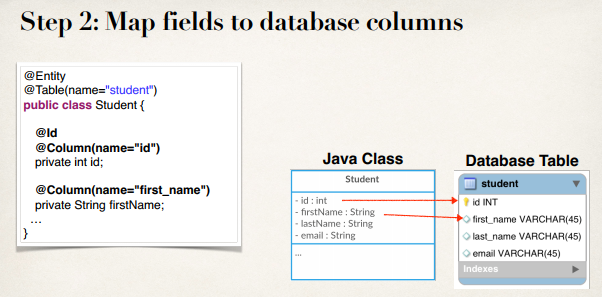
To annotate a Java class with Hibernate we have to follow the flowing steps

1. Map class to database table
2. Map fields to database column

1) Map class to database table:



2) Map fields to database column:



1) Develop Java Code to perform database operations:

First, we will talk about two key players that we need to be aware of for Hibernate.

|  |  |
| --- | --- |
| Class | Description |
| SessionFactory | * Reads the hibernate configuration file * Create Session Object * Heavy-weight object * Only create once in our app |
| Session | * Wraps a JDBC connection * Main object used to save/retrieve objects * Short-live object * Retrieved from SessionFactory |

File: CreateStudentDemo.java:

**package** com.ruhul.hibernateDemo;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.cfg.Configuration;

**import** com.ruhul.entity.Student;

**public** **class** CreateStudentDemo {

**public** **static** **void** main(String[] args) {

//create session factory

SessionFactory factory = **new** Configuration()

.configure("hibernate.cfg.xml")

.addAnnotatedClass(Student.**class**)

.buildSessionFactory();

//create session

Session session = factory.getCurrentSession();

**try** {

// create a student object

System.***out***.println("Creating new student object...");

Student tempStudent = **new** Student("Md. Rezaul Islam", "Reza", "reza@gmail.com");

// start transaction

session.beginTransaction();

// save the student

System.***out***.println("Save the student...");

session.save(tempStudent);

// commit the transaction

session.getTransaction().commit();

System.***out***.println("Done!!!");

} **finally** {

factory.close();

}

}

}

Now if we run our project the student object saves into our database.

Note:

Here the name of Hibernate config file "hibernate.cfg.xml" is optional. If we don’t pass this name then hibernate provide a default name called "hibernate.cfg.xml".

Hence if the name of the Hibernate config file "hibernate.cfg.xml" then we no need to provide the name inside configure() method.

Hibernate Development Process