**Board**

**커넥션 풀(DBCP)** 데이터베이스와 연결된 커넥션을 미리 만들어서 풀(pool) 속에 저장해 두고 있다가 필요할 때 커넥션을 풀에서 쓰고 다시 풀에 반환하는 기법을 말한다.웹 프로그램에서는 데이터베이스의 환경설정과 연결 관리 등을 따로 XML파일이나 속성 파일을 사용해서 관리하고, 이렇게 설정된 정보를 이름을 사용하여 획득하는 방법을 사용한다.

**ConnectionProvider Class**

package util;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class JdbcUtil {

public static void close(Connection con){

if(con!=null){

try {

con.close();

} catch (SQLException e) {

}

}

}

public static void close(PreparedStatement pstmt){

if(pstmt!=null){

try {

pstmt.close();

} catch (SQLException e) {

}

}

}

public static void close(ResultSet rs){

if(rs!=null){

try {

rs.close();

} catch (SQLException e) {

}

}

}

public static void close(Statement stmt){

if(stmt!=null){

try {

stmt.close();

} catch (SQLException e) {

}

}

}

public static void rollback(Connection con){

try {

con.rollback();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**DBMgrLoader Class**

package util;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.util.StringTokenizer;

import javax.servlet.ServletConfig;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.sql.PooledConnection;

import assets.DBConnectionMgr;

public class DBMgrLoader extends HttpServlet{

static DBConnectionMgr pool;

@Override

public void init(ServletConfig config) throws ServletException {

pool = DBConnectionMgr.getInstance();

}

public static Connection getConnection(){

Connection con=null;

try {

con = pool.getConnection();

} catch (SQLException e) {

e.printStackTrace();

} catch (Exception e) {

e.printStackTrace();

}

return con;

}

}

**JDBCLoader Class**

package util;

import java.util.StringTokenizer;

import javax.servlet.ServletConfig;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

public class JDBCLoader extends HttpServlet{

@Override

public void init(ServletConfig config) throws ServletException {

try {

String drivers = config.getInitParameter("jdbcdriver");

StringTokenizer st = new StringTokenizer(drivers, ",");

while (st.hasMoreTokens()) {

String jdbcDriver = st.nextToken();

Class.forName(jdbcDriver);

Class.forName("org.apache.commons.dbcp.PoolingDriver");

}

} catch (Exception e) {

throw new ServletException(e);

}

}

}

**JdbcUtil Class**

package util;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class JdbcUtil {

public static void close(Connection con){

if(con!=null){

try {

con.close();

} catch (SQLException e) {

}

}

}

public static void close(PreparedStatement pstmt){

if(pstmt!=null){

try {

pstmt.close();

} catch (SQLException e) {

}

}

}

public static void close(ResultSet rs){

if(rs!=null){

try {

rs.close();

} catch (SQLException e) {

}

}

}

public static void close(Statement stmt){

if(stmt!=null){

try {

stmt.close();

} catch (SQLException e) {

}

}

}

public static void rollback(Connection con){

try {

con.rollback();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**class DBConnectionMgr**

import java.sql.\*;

import java.util.Properties;

import java.util.Vector;

/\*\*

\* Manages a java.sql.Connection pool.

\*

\* @author Anil Hemrajani

\*/

public class DBConnectionMgr {

private Vector<ConnectionObject> connections = new Vector<ConnectionObject>(10);

private String \_driver = "org.gjt.mm.mysql.Driver",

\_url = "jdbc:mysql://localhost:3306/music?useUnicode=true&amp;characterEncoding=utf8&connectTimeout=3000&zeroDateTimeBehavior=convertToNull",

\_user = "root",

\_password = "1234";

private boolean \_traceOn = false;

private boolean initialized = false;

private int \_openConnections = 50;

private static DBConnectionMgr instance = null;

public DBConnectionMgr() {

}

/\*\* Use this method to set the maximum number of open connections before

unused connections are closed.

\*/

public static DBConnectionMgr getInstance() {

if (instance == null) {

synchronized (DBConnectionMgr.class) {

if (instance == null) {

instance = new DBConnectionMgr();

}

}

}

return instance;

}

public void setOpenConnectionCount(int count) {

\_openConnections = count;

}

public void setEnableTrace(boolean enable) {

\_traceOn = enable;

}

/\*\* Returns a Vector of java.sql.Connection objects \*/

public Vector<ConnectionObject> getConnectionList() {

return connections;

}

/\*\* Opens specified "count" of connections and adds them to the existing pool \*/

public synchronized void setInitOpenConnections(int count)

throws SQLException {

Connection c = null;

ConnectionObject co = null;

for (int i = 0; i < count; i++) {

c = createConnection();

co = new ConnectionObject(c, false);

connections.addElement(co);

trace("ConnectionPoolManager: Adding new DB connection to pool (" + connections.size() + ")");

}

}

/\*\* Returns a count of open connections \*/

public int getConnectionCount() {

return connections.size();

}

/\*\* Returns an unused existing or new connection. \*/

public synchronized Connection getConnection()

throws Exception {

if (!initialized) {

Class<?> c = Class.forName(\_driver);

DriverManager.registerDriver((Driver) c.newInstance());

initialized = true;

}

Connection c = null;

ConnectionObject co = null;

boolean badConnection = false;

for (int i = 0; i < connections.size(); i++) {

co = (ConnectionObject) connections.elementAt(i);

// If connection is not in use, test to ensure it's still valid!

if (!co.inUse) {

try {

badConnection = co.connection.isClosed();

if (!badConnection)

badConnection = (co.connection.getWarnings() != null);

} catch (Exception e) {

badConnection = true;

e.printStackTrace();

}

// Connection is bad, remove from pool

if (badConnection) {

connections.removeElementAt(i);

trace("ConnectionPoolManager: Remove disconnected DB connection #" + i);

continue;

}

c = co.connection;

co.inUse = true;

trace("ConnectionPoolManager: Using existing DB connection #" + (i + 1));

break;

}

}

if (c == null) {

c = createConnection();

co = new ConnectionObject(c, true);

connections.addElement(co);

trace("ConnectionPoolManager: Creating new DB connection #" + connections.size());

}

return c;

}

/\*\* Marks a flag in the ConnectionObject to indicate this connection is no longer in use \*/

public synchronized void freeConnection(Connection c) {

if (c == null)

return;

ConnectionObject co = null;

for (int i = 0; i < connections.size(); i++) {

co = (ConnectionObject) connections.elementAt(i);

if (c == co.connection) {

co.inUse = false;

break;

}

}

for (int i = 0; i < connections.size(); i++) {

co = (ConnectionObject) connections.elementAt(i);

if ((i + 1) > \_openConnections && !co.inUse)

removeConnection(co.connection);

}

}

public void freeConnection(Connection c, PreparedStatement p, ResultSet r) {

try {

if (r != null) r.close();

if (p != null) p.close();

freeConnection(c);

} catch (SQLException e) {

e.printStackTrace();

}

}

public void freeConnection(Connection c, Statement s, ResultSet r) {

try {

if (r != null) r.close();

if (s != null) s.close();

freeConnection(c);

} catch (SQLException e) {

e.printStackTrace();

}

}

public void freeConnection(Connection c, PreparedStatement p) {

try {

if (p != null) p.close();

freeConnection(c);

} catch (SQLException e) {

e.printStackTrace();

}

}

public void freeConnection(Connection c, Statement s) {

try {

if (s != null) s.close();

freeConnection(c);

} catch (SQLException e) {

e.printStackTrace();

}

}

/\*\* Marks a flag in the ConnectionObject to indicate this connection is no longer in use \*/

public synchronized void removeConnection(Connection c) {

if (c == null)

return;

ConnectionObject co = null;

for (int i = 0; i < connections.size(); i++) {

co = (ConnectionObject) connections.elementAt(i);

if (c == co.connection) {

try {

c.close();

connections.removeElementAt(i);

trace("Removed " + c.toString());

} catch (Exception e) {

e.printStackTrace();

}

break;

}

}

}

private Connection createConnection()

throws SQLException {

Connection con = null;

try {

if (\_user == null)

\_user = "";

if (\_password == null)

\_password = "";

Properties props = new Properties();

props.put("user", \_user);

props.put("password", \_password);

con = DriverManager.getConnection(\_url, props);

} catch (Throwable t) {

throw new SQLException(t.getMessage());

}

return con;

}

/\*\* Closes all connections and clears out the connection pool \*/

public void releaseFreeConnections() {

trace("ConnectionPoolManager.releaseFreeConnections()");

@SuppressWarnings("unused")

Connection c = null;

ConnectionObject co = null;

for (int i = 0; i < connections.size(); i++) {

co = (ConnectionObject) connections.elementAt(i);

if (!co.inUse)

removeConnection(co.connection);

}

}

/\*\* Closes all connections and clears out the connection pool \*/

public void finalize() {

trace("ConnectionPoolManager.finalize()");

@SuppressWarnings("unused")

Connection c = null;

ConnectionObject co = null;

for (int i = 0; i < connections.size(); i++) {

co = (ConnectionObject) connections.elementAt(i);

try {

co.connection.close();

} catch (Exception e) {

e.printStackTrace();

}

co = null;

}

connections.removeAllElements();

}

private void trace(String s) {

if (\_traceOn)

System.err.println(s);

}

}

class ConnectionObject {

public java.sql.Connection connection = null;

public boolean inUse = false;

public ConnectionObject(Connection c, boolean useFlag) {

connection = c;

inUse = useFlag;

}

}

**Bcryp(라이브러리)**는 패스워드 **암호화**에 특화된 Password hashing function이다. 애초부터 패스워드 저장을 목적으로 설계되어 매우 강력한 패스워드 다이제스트를 생성하는 시스템을 쉽게 구현할 수 있다. salt 값을 따로 DB에 저장하지 않는다

import java.io.UnsupportedEncodingException;

import java.security.SecureRandom;

/\*\*

\* BCrypt implements OpenBSD-style Blowfish password hashing using

\* the scheme described in "A Future-Adaptable Password Scheme" by

\* Niels Provos and David Mazieres.

\* Usage is really simple. To hash a password for the first time,

\* call the hashpw method with a random salt, like this:

\* <p>

\* <code>

\* String pw\_hash = BCrypt.hashpw(plain\_password, BCrypt.gensalt()); <br />

\* </code>

\* <p>

\* To check whether a plaintext password matches one that has been

\* hashed previously, use the checkpw method:

\* <p>

\* <code>

\* if (BCrypt.checkpw(candidate\_password, stored\_hash))<br />

\* &nbsp;&nbsp;&nbsp;&nbsp;System.out.println("It matches");<br />

\* else<br />

\* &nbsp;&nbsp;&nbsp;&nbsp;System.out.println("It does not match");<br />

\* </code>

\* <p>

\* The gensalt() method takes an optional parameter (log\_rounds)

\* that determines the computational complexity of the hashing:

\* <p>

\* <code>

\* String strong\_salt = BCrypt.gensalt(10)<br />

\* String stronger\_salt = BCrypt.gensalt(12)<br />

\* </code>

\* <p>

\* The amount of work increases exponentially (2\*\*log\_rounds), so

\* each increment is twice as much work. The default log\_rounds is

\* 10, and the valid range is 4 to 31.

\*

\* @author Damien Miller

\* @version 0.2

\*/

public class BCrypt {

// BCrypt parameters

private static final int GENSALT\_DEFAULT\_LOG2\_ROUNDS = 10;

private static final int BCRYPT\_SALT\_LEN = 16;

// Blowfish parameters

private static final int BLOWFISH\_NUM\_ROUNDS = 16;

// Initial contents of key schedule

private static final int P\_orig[] = {

0x243f6a88, 0x85a308d3, 0x13198a2e, 0x03707344,

0xa4093822, 0x299f31d0, 0x082efa98, 0xec4e6c89,

0x452821e6, 0x38d01377, 0xbe5466cf, 0x34e90c6c,

0xc0ac29b7, 0xc97c50dd, 0x3f84d5b5, 0xb5470917,

0x9216d5d9, 0x8979fb1b

};

private static final int S\_orig[] = {

0xd1310ba6, 0x98dfb5ac, 0x2ffd72db, 0xd01adfb7,

0xb8e1afed, 0x6a267e96, 0xba7c9045, 0xf12c7f99,

0x24a19947, 0xb3916cf7, 0x0801f2e2, 0x858efc16,

**…… 이하 중략**

**public** **static** String gensalt(**int** log\_rounds, SecureRandom random) {

StringBuffer rs = **new** StringBuffer();

**byte** rnd[] = **new** **byte**[***BCRYPT\_SALT\_LEN***];

random.nextBytes(rnd);

rs.append("$2a$");

**if** (log\_rounds < 10)

rs.append("0");

rs.append(Integer.*toString*(log\_rounds));

rs.append("$");

rs.append(*encode\_base64*(rnd, rnd.length));

**return** rs.toString();

}

/\*\*

\* Generate a salt for use with the BCrypt.hashpw() method

\* **@param** log\_rounds the log2 of the number of rounds of

\* hashing to apply - the work factor therefore increases as

\* 2\*\*log\_rounds.

\* **@return** an encoded salt value

\*/

**public** **static** String gensalt(**int** log\_rounds) {

**return** *gensalt*(log\_rounds, **new** SecureRandom());

}

/\*\*

\* Generate a salt for use with the BCrypt.hashpw() method,

\* selecting a reasonable default for the number of hashing

\* rounds to apply

\* **@return** an encoded salt value

\*/

**public** **static** String gensalt() {

**return** *gensalt*(***GENSALT\_DEFAULT\_LOG2\_ROUNDS***);

}

/\*\*

\* Check that a plaintext password matches a previously hashed

\* one

\* **@param** plaintext the plaintext password to verify

\* **@param** hashed the previously-hashed password

\* **@return** true if the passwords match, false otherwise

\*/

**public** **static** **boolean** checkpw(String plaintext, String hashed) {

**return** (hashed.compareTo(*hashpw*(plaintext, hashed)) == 0);

}

}

Table

|  |
| --- |
| article |
| `sequence\_no` int(11) NOT NULL |
| PRIMARY KEY (`sequence\_no`) |

|  |  |
| --- | --- |
| board | |
| artboardarticle\_idtitleicle\_id | int(11) NOT NULL AUTO\_INCREMENT |
| title | varchar(100) DEFAULT NULL |
| grp | int(11) DEFAULT NULL |
| seq | int(11) DEFAULT NULL |
| lvl | int(11) DEFAULT NULL |
| posting\_date | timestamp NULL DEFAULT NULL |
| read\_count | bigint(1) DEFAULT '0' |
| writer\_name | varchar(45) DEFAULT NULL |
| password | varchar(45) DEFAULT NULL |
| content | varchar(2048) DEFAULT NULL |
| deleted | bigint(1) DEFAULT '0' |
| PRIMARY KEY (`article\_id`) | |