A Simple Guide to Connection Pooling in Java

1. Overview

Connection pooling is a well-known data access pattern, whose main purpose is to reduce the overhead involved in performing database connections and read/write database operations.

In a nutshell, a connection pool is, at the most basic level, a database connection cache implementation, which can be configured to suit specific requirements.

In this tutorial, we'll make a quick roundup of a few popular connection pooling frameworks, and we'll learn how to implement from scratch our own connection pool.

2. Why Connection Pooling?

The question is rhetorical, of course.

If we analyze the sequence of steps involved in a typical database connection life cycle, we'll understand why:

- Opening a connection to the database using the database driver
- Opening a TCP socket for reading/writing data
- Reading / writing data over the socket
- Closing the connection
- Closing the socket

It becomes evident that database connections are fairly expensive operations, and as such, should be reduced to a minimum in every possible use case (in edge cases, just avoided).

Here's where connection pooling implementations come into play.

By just simply implementing a database connection container, which allows us to reuse a number of existing connections, we can effectively save the cost of performing a huge number of expensive database trips, hence boosting the overall performance of our database-driven applications.

3. JDBC Connection Pooling Frameworks

From a pragmatic perspective, implementing a connection pool from the ground up is just pointless, considering the number of "enterprise-ready" connection pooling frameworks available out there.

From a didactic one, which is the goal of this article, it's not.

Even so, before we learn how to implement a basic connection pool, let's first showcase a few popular connection pooling frameworks.

3.1. Apache Commons DBCP

Let's start this quick roundup with Apache Commons DBCP Component, a full-featured connection pooling JDBC framework:

```
public class DBCPDataSource {

    private static BasicDataSource ds = new BasicDataSource();
    static {
        ds.setUrl("jdbc:h2:mem:test");
        ds.setUsername("user");
        ds.setPassword("password");
        ds.setMinIdle(5);
        ds.setMaxIdle(10);
        ds.setMaxOpenPreparedStatements(100);
    }

    public static Connection getConnection() throws SQLException {
        return ds.getConnection();
    }

    private DBCPDataSource(){ }
}
```

In this case, we’ve used a wrapper class with a static block to easily configure DBCP’s properties.

Here’s how to get a pooled connection with the DBCPDataSource class:

```
Connection con = DBCPDataSource.getConnection();
```

3.2. HikariCP
Moving on, let’s look at HikariCP, a lightning fast JDBC connection pooling framework created by Brett Wooldridge (for the full details on how to configure and get the most out of HikariCP, please check this article):

```
public class HikariCPDataSource {

    private static HikariConfig config = new HikariConfig();
    private static HikariDataSource ds;

    static {
        config.setJdbcUrl("jdbc:h2:mem:test");
        config.setUsername("user");
        config.setPassword("password");
        config.addDataSourceProperty("cachePrepStmts", "true");
        config.addDataSourceProperty("prepStmtCacheSize", "250");
        config.addDataSourceProperty("prepStmtCacheSqlLimit", "2048");
        ds = new HikariDataSource(config);
    }

    public static Connection getConnection() throws SQLException {
        return ds.getConnection();
    }

    private HikariCPDataSource(){}
}
```

Similarly, here's how to get a pooled connection with the HikariCPDataSource class:

```
Connection con = HikariCPDataSource.getConnection();
```

3.3. C3PO

Last in this review is C3PO, a powerful JDBC4 connection and statement pooling framework developed by Steve Waldman:

```
public class C3poDataSource {

    private static ComboPooledDataSource cpds = new ComboPooledDataSource();

    static {
        try {
            cpds.setDriverClass("org.h2.Driver");
            cpds.setJdbcUrl("jdbc:h2:mem:test");
            cpds.setUser("user");
            cpds.setPassword("password");
        } catch (PropertyVetoException e) {
            // handle the exception
        }
    }

    public static Connection getConnection() throws SQLException {
        return cpds.getConnection();
    }

    private C3poDataSource(){}
}
```

As expected, getting a pooled connection with the C3poDataSource class is similar to the previous examples:

```
Connection con = C3poDataSource.getConnection();
```

4. A Simple Implementation

To better understand the underlying logic of connection pooling, let's create a simple implementation.

Let's start out with a loosely-coupled design, based on just one single interface:

```
public interface ConnectionPool {
    Connection getConnection();
    boolean releaseConnection(Connection connection);
    String getUrl();
    String getUser();
    String getPassword();
}
```

The ConnectionPool interface defines the public API of a basic connection pool.

Now, let’s create an implementation, which provides some basic functionality, including getting and re-leasing a pooled connection:

```
public class BasicConnectionPool
  implements ConnectionPool {

    private String url;
    private String user;
    private String password;
    private List<Connection> connectionPool;
    private List<Connection> usedConnections = new ArrayList<>();
    private static int INITIAL_POOL_SIZE = 10;

    public static BasicConnectionPool create(
      String url, String user,
      String password) throws SQLException {

        List<Connection> pool = new ArrayList<>(INITIAL_POOL_SIZE);
        for (int i = 0; i < INITIAL_POOL_SIZE; i++) {
            pool.add(createConnection(url, user, password));
        }
        return new BasicConnectionPool(url, user, password, pool);
    }

    // standard constructors

    @Override
    public Connection getConnection() {
        Connection connection = connectionPool
          .remove(connectionPool.size() - 1);
        usedConnections.add(connection);
        return connection;
    }

    @Override
    public boolean releaseConnection(Connection connection) {
        connectionPool.add(connection);
        return usedConnections.remove(connection);
    }

    private static Connection createConnection(
      String url, String user, String password)
      throws SQLException {
        return DriverManager.getConnection(url, user, password);
    }

    public int getSize() {
        return connectionPool.size() + usedConnections.size();
    }

    // standard getters
}
```

While pretty naive, the BasicConnectionPool class provides the minimal functionality that we'd expect from a typical connection pooling implementation.

In a nutshell, the class initializes a connection pool based on an ArrayList that stores 10 connections, which can be easily reused.

It's possible to create JDBC connections with the DriverManager class and with Datasource implementations.

As it's much better to keep the creation of connections database agnostic, we've used the former, within the create() static factory method.

In this case, we've placed the method within the BasicConnectionPool, because this is the only implementation of the interface.

In a more complex design, with multiple ConnectionPool implementations, it'd be preferable to place it in the interface, therefore getting a more flexible design and a greater level of cohesion.

The most relevant point to stress here is that once the pool is created, connections are fetched from the pool, so there's no need to create new ones.

Furthermore, when a connection is released, it's actually returned back to the pool, so other clients can reuse it.

There's no any further interaction with the underlying database, such as an explicit call to the Connection's close() method.

5. Using the BasicConnectionPool Class

As expected, using our BasicConnectionPool class is straightforward.

Let's create a simple unit test and get a pooled in-memory H2 connection:

```
@Test
public whenCalledgetConnection_thenCorrect() {
    ConnectionPool connectionPool = BasicConnectionPool
      .create("jdbc:h2:mem:test", "user", "password");

    assertTrue(connectionPool.getConnection().isValid(1));
}
```

6. Further Improvements and Refactoring

Of course, there's plenty of room to tweak/extend the current functionality of our connection pooling implementation.

For instance, we could refactor the getConnection() method, and add support for maximum pool size. If all available connections are taken, and the current pool size is less than the configured maximum, the method will create a new connection:

```
@Override
public Connection getConnection() throws SQLException {
    if (connectionPool.isEmpty()) {
        if (usedConnections.size() < MAX_POOL_SIZE) {
            connectionPool.add(createConnection(url, user, password));
        } else {
            throw new RuntimeException(
              "Maximum pool size reached, no available connections!");
        }
    }

    Connection connection = connectionPool
      .remove(connectionPool.size() - 1);
    usedConnections.add(connection);
    return connection;
}
```

Note that the method now throws SQLException, meaning we'll have to update the interface signature as well.

Or, we could add a method to gracefully shut down our connection pool instance:

```
public void shutdown() throws SQLException {
    usedConnections.forEach(this::releaseConnection);
    for (Connection c : connectionPool) {
        c.close();
    }
    connectionPool.clear();
}
```

In production-ready implementations, a connection pool should provide a bunch of extra features, such as the ability for tracking the connections that are currently in use, support for prepared statement pooling, and so forth.

As we'll keep things simple, we'll omit how to implement these additional features and keep the implementation non-thread-safe for the sake of clarity.

7. Conclusion

In this article, we took an in-depth look at what connection pooling is and learned how to roll our own connection pooling implementation.

Of course, we don't have to start from scratch every time that we want to add a full-featured connection pooling layer to our applications.

That’s why we made first a simple roundup showing some of the most popular connection pool frameworks, so we can have a clear idea on how to work with them, and pick up the one that best suits our requirements.

Apache Commons DBCP

Beans.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd ">

<bean id="dataSource" class="org.apache.commons.dbcp2.BasicData-
Source">
<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
<property name="url" value="jdbc:oracle:thin:@localhost:1521:XE" />
<property name="username" value="system" />
<property name="password" value="Admin@123" />
<property name="initialSize" value="1" />
<property name="maxTotal" value="5" />
<property name="maxIdle" value="2" />
<property name="minIdle" value="0" />
</bean>

<!-- Definition for studentJDBCTemplate bean -->
<bean id="studentimpl" class="com.java.db.cp.repo.StudentDAOImpl">
<property name="dataSource" ref="dataSource"/>
</bean>
</beans>
```

```
package com.java.db.cp.bean;

public class Student {
private int id;
private int age;
private String name;

public void setId(int id) {
this.id = id;
}

public int getId() {
return id;
}

public void setAge(int age) {
this.age = age;
}

public void setName(String name) {
this.name = name;
}

public int getAge() {
return age;
}
```

```
public String getName() {
return name;
}
}

package com.java.db.cp.repo;

import javax.sql.DataSource;

public interface StudentDAO {

/** This is the method to be used to initialize database resources ie. connec-
tion.*/
public void setDataSource(DataSource ds);

/** This is the method to be used to create a record in the Student and Marks ta-
bles.*/
public void create(String name, Integer age);
}


package com.java.db.cp.repo;

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class StudentDAOImpl implements StudentDAO {

private DataSource dataSource;
private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {
this.dataSource=dataSource;
jdbcTemplate = new JdbcTemplate(dataSource);
}

public void create(String name, Integer age) {
String sql ="insert into Student(name, age) values(?, ?)";
jdbcTemplate.update(sql, name, age);
System.out.println("Updated Student data successfully");
}

}

package com.java.db.cp;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.java.db.cp.repo.StudentDAOImpl;

public class MainApp {
public static void main(String[] args) {
ApplicationContext context = new ClassPathXmlApplicationContext("re-
sources/Beans.xml");
StudentDAOImpl studJdbcTemplate = (StudentDAOImpl)context.getBean("studentimpl");
StudentDAOImpl studJdbcTemplate2 = (StudentDAOImpl)context.getBean("studentimpl");
StudentDAOImpl studJdbcTemplate3 = (StudentDAOImpl)context.getBean("studentimpl");
StudentDAOImpl studJdbcTemplate4 = (StudentDAOImpl)context.getBean("studentimpl");
StudentDAOImpl studJdbcTemplate5 = (StudentDAOImpl)context.getBean("studentimpl");
StudentDAOImpl studJdbcTemplate6 = (StudentDAOImpl)context.getBean("studentimpl");
StudentDAOImpl studJdbcTemplate7 = (StudentDAOImpl)context.getBean("studentimpl");
StudentDAOImpl studJdbcTemplate8 = (StudentDAOImpl)context.getBean("studentimpl");
StudentDAOImpl studJdbcTemplate9 = (StudentDAOImpl)context.getBean("studentimpl");

System.out.println("------Records creation--------");
studJdbcTemplate.create("Samanvi1", 11);
studJdbcTemplate2.create("Avishka1", 20);
studJdbcTemplate3.create("Litchi1", 25);
studJdbcTemplate4.create("Samanvi2", 11);
studJdbcTemplate5.create("Avishka2", 20);
studJdbcTemplate6.create("Litchi2", 25);
studJdbcTemplate7.create("Samanvi3", 11);
studJdbcTemplate8.create("Avishka3", 20);
studJdbcTemplate9.create("Litchi3", 25);
System.out.println("------Records created successfully--------");
}
}
```