CONNECTION POOLING

- 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.
- The most basic implementation is a cache for database connection to reuse existing connections instead of creating new ones

Why does connection pooling makes sense?

- The life cycle of a typical connection consist of:
  1. Opening connection to the database using the dabase driver
  2. Opening a TCP Socket for reading / writing data
  3. Reading / writing data over the socket
  4. Closing the connection
  5. Closing the socket
- Establishing the communication between two different computers is always a rather expensive operation, since the machines have to a certain degree wait for the response of the other
- Minimizing such operations is a common strategy for improving performance

Implementing a most basic connection pool

So what are the requirements for such an connection pool?

- Accepting password and username
- Connecting to a database
- Provide connections
- Take care for connections that are no longer used

The basic requirements for a connection pool could be describe like in the following interface:

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

[Filename: Servlet/ConnectionPool.java]

And the corresponding class providing a create factory and can manage up to 10 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);
    }

    @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();
    }
}
```

[Filename: Servlet/BasicConnectionPool.java]

Improving connection pool

- Connection pools can be used for more than caching connections
  - The connections pool could be expanded dynamically
  - Connection could be closed if too many are unoccupied
  - Unused connection could be set to sleep instead of closing
  - Make it usable in multi threading

```
@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;
}
```

[Filename: Servlet/ConnectionPool.java]

- But there are already many well-functioning and well-thought-out approaches for this

ConnectionPool Libraries

It's unlikely that a tool you write on the fly is actually better than a common tool written just for that purpose. There are a lot of 3rd party connection pooling frameworks. Here are some of the more well known:

- DBCP (Apache Commons DBCP)
- C3P0 (by Steve Waldman)
- HikariCP (by Brett Wooldridge)

When your program runs inside of an application server (like Tomcat) it is recommended to use the connection pooling framework that it provides!

- Tomcat JDBC Connection Pool (alternative to Apache Commons DBCP)

Tomcat DBCP Setup

- All necessary classes are contained in `$CATALINA_HOME/lib/tomcat-jdbc.jar`
  - DataSource, PoolProperties, ...
- Only dependency is `$CATALINA_HOME/bin/tomcat-juli.jar` containing Tomcats own logger framework
  - This file needs to be manually included in the build path (Eclipse)
- The used JDBC driver(s) (PostgreSQL, MySQL, Oracle,...) needs to be added to `$CATALINA_HOME/lib` so that they can be registered on Tomcat startup

Tomcat DBCP Configuration

Tomcat can be configured to use one global ConnectionPool for all applications in the container or each one maintains its own

- Global
  - Create and configure a database resource with JNDI (Java Naming and Directory Interface Resource) inside the `$CATALINA_HOME/conf/server.xml`
  - Create a ResourceLink in each web application inside the `PROJECT_DIR/META-INF/context.xml` to make the global resource available to the webapp
  - Create a Resource Reference in the `PROJECT_DIR/WEB_INF/web.xml` to make it available in the Context
  - In the Servlet: Obtain the environment name context und look up the DataSource with its global name
- Application
  - Programmatically configure the PoolProperties and add them to a DataSource
  - Add the DataSource to the ServletContext to make it available to all Servlets (a ServletContextListener would be an option to init and close the DataSource)

PoolProperties

Bundles the settings for the connection pool (DataSource)

- First set the necessary connection and credential information
  - `.setUrl(String)` sets the database URL
  - `.setDriverClassName(String)` sets the name of the JDBC driver
  - `.setUsername(String)` & `.setPassword(String)` default user/password
    - `DataSource.getConnection(username,password)` can be called to open a Connection with different credentials
    - `.setAlternativeUsernameAllowed(boolean)` to enable/disable this functionality
- Configurations of the pool size
  - `.setInitialSize(int)` The initial number of connections established on pool start
  - `.setMaxActive(int)` The maximum amount of active connections
    - `.setMaxWait(int)` Wait `int` (in miliseconds) before throwing an exception
  - `.setMaxIdle(int) / .setMinIdle(int)` The maximum and minimum amount of idle connections
    - `.setMinEvictableIdleTimeMillis(int)` Time a connection can be idle before being closed

- Testing and validation of connections (checks if a Connection still "works")
  - `.setTestOnBorrow / .setTestOnConnect / .setTestOnReturn / .setTestWhileIdle(boolean)` Enables/disables if a Connection should be tested before being borrowed / on creation / before being returned to the pool / by the idle object evictor
    * `.setTimeBetweenEvictionRunsMillis(int)` Sets the time between runs of the cleaner which flags idle and abandoned connections and validates idle connections
  - `.setValidationQuery(String)` Specifies the SQL query used to validate the Connection (typically "SELECT 1" or "SELECT 1 FROM DUAL"). Output is irrelevant, it will just be checked if a SQLException is thrown
  - `.setValidationInterval(long)` Sets the maximum frequency of validation runs to avoid excess validation
  - `.setValidationQueryTimeout(int)` The timeout in seconds before a connection validation queries fail

- Forgetting to close a Connection from the ConnectionPool results in it never returning to the pool creating a "leak" $\rightarrow$ may result in database connection failures when there are no more available connections in the pool
  - `.setRemoveAbandoned(true)` can be set to tell Tomcat to recover connections that are flagged abandoned
  - `.setRemoveAbamdonedTimeout(int)` flags a connection as abandoned after `int` seconds
  - `.setLogAbandoned(true)` can be set to create a stack trace log of the code that abandoned the connection
- Miscellaneous
  - `.setInitSQL(String)` Sets a SQL query which is run every time a Connection is created
  - `.setJdbcInterceptors(String)` A semicolon separated list of classnames extending JdbcInterceptor
    * `org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer` keeps track of opened statements, and closes them when the connection is returned to the pool

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import org.apache.tomcat.jdbc.pool.DataSource;
import org.apache.tomcat.jdbc.pool.PoolProperties;

public class TomcatConnectionPool {

    public static void main(String[] args) throws Exception {
        PoolProperties p = new PoolProperties();

        //URL and login credentials
        p.setUrl("jdbc:postgresql://localhost:5432/thedb?currentSchema=mondial_rel");
        p.setDriverClassName("org.postgresql.Driver");
        p.setUsername("scott");
        p.setPassword("tiger");

        //Pool size configurations
        p.setInitialSize(10);
        p.setMaxActive(100);
        p.setMaxWait(10000);
        p.setMaxIdle(80);
        p.setMinIdle(10);
        p.setMinEvictableIdleTimeMillis(30000);
```

```
        //Testing and validation of Connections
        p.setTestWhileIdle(false);
        p.setTestOnReturn(false);
        p.setTestOnBorrow(true);
        p.setValidationQuery("SELECT 1");
        p.setValidationInterval(30000);
        p.setTimeBetweenEvictionRunsMillis(30000);

        //Abandoned connection cleaner
        p.setRemoveAbandoned(true);
        p.setRemoveAbandonedTimeout(60);
        p.setLogAbandoned(true);

        //Miscellaneous
        p.setJdbcInterceptors(
          "org.apache.tomcat.jdbc.pool.interceptor.ConnectionState;"+
          "org.apache.tomcat.jdbc.pool.interceptor.StatementFinalizer");


        //Creating and configuring the ConnectionPool
        DataSource datasource = new DataSource();
        datasource.setPoolProperties(p);

        Connection con = null;
        try {
          con = datasource.getConnection();
            ...
        } finally {
          if (con!=null) try {con.close();}catch (Exception ignore) {}
        }
    }
}
```

[Filename: Servlet/TomcatConnectionPool.java]

Using the pool for multiple Web Applications - Server.xml

- The server.xml could look like this, specifying the configuration of the connection pool:

```
...
  <GlobalNamingResources>
    ...
    <Resource name="jdbc/myoracle"
              global="jdbc/myoracle"
              factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
              auth="Container"
              type="javax.sql.DataSource"
              username="scott"
              password="tiger"
              driverClassName="oracle.jdbc.OracleDriver"
              description="Great Database. Some people say its the best."
              url="jdbc:oracle:thin:@127.0.0.1:1521:mysid"
              maxTotal="10"
              maxIdle="10"
              maxWaitMillis="10000"
              removeAbandonedTimeout="300"
              defaultAutoCommit="true" />
     ...
  </GlobalNamingResources>
```

[Filename: Servlet/server.xml]

Using the pool for multiple Web Applications - Context.xml

- The name attribute is the name of the link to be created. For consistency it is better to give the same name to the link as the name of the global resource.
- The global attribute is the name of the global resource defined in the global JNDI context in the server.xml configuration file.
- type attribute is the fully qualified Java class name expected to be returned on lookup of this resource performed in the web application.

```
<Context>
    <ResourceLink name="jdbc/myoracle"
    global="jdbc/myoracle"
    type="javax.sql.DataSource"/>
</Context>
```

[Filename: Servlet/context.xml]

Using the pool for multiple Web Applications - Web.xml

```
<web-app>
...
    <resource-ref>
        <description>
            This is a reference to the global Resource for a database connetion.
        </description>
        <res-ref-name>
            jdbc/myoracle
        </res-ref-name>
        <res-type>
            javax.sql.DataSource
        </res-type>
        <res-auth>
            Container
        </res-auth>
    </resource-ref>
...
</web-app>
```

[Filename: Servlet/web.xml]

Using the pool for multiple Web Applications - Web.xml

- **Resource Reference** is needed to enable a web application to look up a Resource using “Context” element prepared for that web application on its deployment, and to keep track of “Resources” that application depends on
- **res-ref-name** element is used to provide the name of the “Resource” referenced. Note that there must exist a “Resource” entry with the same name as in this element.
- **res-type element** is used to define the type of the object factory generated by “Resource”.
- **res-auth element** is used to specify who will authenticate into the “Resource”.

Using the pool for multiple Web Applications - Getting Connections in the actual web app

- Through a context object JNDI is able to find the resource and return a propper class of the type datasource, which can be used just like in the application specific connection pool method.
- The initial context access all context resources
  - The "java:comp/env" is selecting the "local" structure and is always necessary
- Second context then can selected a specific resource of them
  - Therefore the specific named defined in the xml files has to be used
- Connection, Statement, and ResultSet needs to be closed/returned to the pool, otherwise the connection neither used, nor usable for others

```
Context initialContext = new InitialContext();
Context environmentContext = (Context) initialContext.lookup("java:comp/env");
String dataResourceName = "jdbc/JCGExampleDB";
DataSource dataSource = (DataSource) environmentContext.lookup(dataResourceName);
Connection con = dataSource.getConnection()
```

[Filename: Servlet/web.xml]

Connection leaks

- A web application has to explicitly close ResultSet's, Statement's, and Connection's.
- Failure of a web application to close these resources can result in them never being available again for reuse, a database connection pool "leak".
- This can eventually result in database connections failing if there are no more available connections.
- DBCP can be configured to track and recover these abandoned database connections

References

- https://examples.javacodegeeks.com/enterprise-java/tomcat/tomcat-connection-pool-configuration-example/