31.10. Connection Pools and Data Sources

JDBC 2 introduced standard connection pooling features in an add-on API known as the JDBC 2.0 Optional Package (also known as the JDBC 2.0 Standard Extension). These features have since been included in the core JDBC 3 API. The PostgreSQL JDBC drivers support these features if it has been compiled with JDK 1.3.x in combination with the JDBC 2.0 Optional Package (JDBC 2), or with JDK 1.4 or higher (JDBC 3). Most application servers include the JDBC 2.0 Optional Package, but it is also available separately from the Sun JDBC download site.

31.10.1. Overview

The JDBC API provides a client and a server interface for connection pooling. The client interface is `javax.sql.DataSource`, which is what application code will typically use to acquire a pooled database connection. The server interface is `javax.sql.ConnectionPoolDataSource`, which is how most application servers will interface with the PostgreSQL JDBC driver.

In an application server environment, the application server configuration will typically refer to the PostgreSQL `ConnectionPoolDataSource` implementation, while the application component code will typically acquire a `DataSource` implementation provided by the application server (not by PostgreSQL).

For an environment without an application server, PostgreSQL provides two implementations of `DataSource` which an application can use directly. One implementation performs connection pooling, while the other simply provides access to database connections through the `DataSource` interface without any pooling. Again, these implementations should not be used in an application server environment unless the application server does not support the `ConnectionPoolDataSource` interface.

31.10.2. Application Servers: `ConnectionPoolDataSource`

PostgreSQL includes one implementation of `ConnectionPoolDataSource` for JDBC 2 and one for JDBC 3, as shown in Table 31-1.

Table 31-1. `ConnectionPoolDataSource` Implementations

JDBC	Implementation Class
2	`org.postgresql.jdbc2.optional.ConnectionPool`
3	`org.postgresql.jdbc3.Jdbc3ConnectionPool`

Both implementations use the same configuration scheme. JDBC requires that a `ConnectionPoolDataSource` be configured via JavaBean properties, shown in Table 31-2, so there are get and set methods for each of these properties.

Table 31-2. `ConnectionPoolDataSource` Configuration Properties

Property	Type	Description
`serverName`	`String`	PostgreSQL database server host name
`databaseName`	`String`	PostgreSQL database name
`portNumber`	`int`	TCP port which the PostgreSQL database server is listening on (or 0 to use the default port)
`user`	`String`	User used to make database connections
`password`	`String`	Password used to make database connections
`defaultAutoCommit`	`boolean`	Whether connections should have autocommit enabled or disabled when they are supplied to the caller. The default is `false`, to disable autocommit.

Many application servers use a properties-style syntax to configure these properties, so it would not be unusual to enter properties as a block of text. If the application server provides a single area to enter all the properties, they might be listed like this:

```
serverName=localhost
databaseName=test
user=testuser
password=testpassword
```

Or, if semicolons are used as separators instead of newlines, it could look like this:

```
serverName=localhost;databaseName=test;user=testuser;password=testpassword
```

31.10.3. Applications: `DataSource`

PostgreSQL includes two implementations of `DataSource` for JDBC 2 and two for JDBC 3, as shown in Table 31-3. The pooling implementations do not actually close connections when the client calls the `close` method, but instead return the connections to a pool of available connections for other clients to use. This avoids any overhead of repeatedly opening and closing connections, and allows a large number of clients to share a small number of database connections.

The pooling data-source implementation provided here is not the most feature-rich in the world. Among other things, connections are never closed until the pool itself is closed; there is no way to shrink the pool. As well, connections requested for users other than the default configured user are not pooled. Many application servers provide more advanced pooling features and use the `ConnectionPoolDataSource` implementation instead.

Table 31-3. `DataSource` Implementations

JDBC	Pooling	Implementation Class
2	No	`org.postgresql.jdbc2.optional.SimpleDataSource`
2	Yes	`org.postgresql.jdbc2.optional.PoolingDataSource`
3	No	`org.postgresql.jdbc3.Jdbc3SimpleDataSource`
3	Yes	`org.postgresql.jdbc3.Jdbc3PoolingDataSource`

All the implementations use the same configuration scheme. JDBC requires that a `DataSource` be configured via JavaBean properties, shown in Table 31-4, so there are get and set methods for each of these properties.

Table 31-4. `DataSource` Configuration Properties

Property	Type	Description
`serverName`	`String`	PostgreSQL database server host name
`databaseName`	`String`	PostgreSQL database name
`portNumber`	`int`	TCP port which the PostgreSQL database server is listening on (or 0 to use the default port)
`user`	`String`	User used to make database connections
`password`	`String`	Password used to make database connections

The pooling implementations require some additional configuration properties, which are shown in Table 31-5.

Table 31-5. Additional Pooling `DataSource` Configuration Properties

Property	Type	Description
`dataSourceName`	`String`	Every pooling `DataSource` must have a unique name.
`initialConnections`	`int`	The number of database connections to be created when the pool is initialized.
`maxConnections`	`int`	The maximum number of open database connections to allow. When more connections are requested, the caller will hang until a connection is returned to the pool.

Example 31-9 shows an example of typical application code using a pooling `DataSource`.

Example 31-9. `DataSource` Code Example

Code to initialize a pooling `DataSource` might look like this:

```
Jdbc3PoolingDataSource source = new Jdbc3PoolingDataSource();
source.setDataSourceName("A Data Source");
source.setServerName("localhost");
source.setDatabaseName("test");
source.setUser("testuser");
source.setPassword("testpassword");
source.setMaxConnections(10);
```

Then code to use a connection from the pool might look like this. Note that it is critical that the connections are eventually closed. Else the pool will "leak" connections and will eventually lock all the clients out.

```
Connection con = null;
try {
    con = source.getConnection();
    // use connection
} catch (SQLException e) {
    // log error
} finally {
    if (con != null) {
        try { con.close(); } catch (SQLException e) {}
    }
}
```

31.10.4. Data Sources and JNDI

All the `ConnectionPoolDataSource` and `DataSource` implementations can be stored in JNDI. In the case of the nonpooling implementations, a new instance will be created every time the object is retrieved from JNDI, with the same settings as the instance that was stored. For the pooling implementations, the same instance will be retrieved as long as it is available (e.g., not a different JVM retrieving the pool from JNDI), or a new instance with the same settings created otherwise.

In the application server environment, typically the application server's `DataSource` instance will be stored in JNDI, instead of the PostgreSQL `ConnectionPoolDataSource` implementation.

In an application environment, the application may store the `DataSource` in JNDI so that it doesn't have to make a reference to the `DataSource` available to all application components that may need to use it. An example of this is shown in Example 31-10.

Example 31-10. `DataSource` JNDI Code Example

Application code to initialize a pooling `DataSource` and add it to JNDI might look like this:

```
Jdbc3PoolingDataSource source = new Jdbc3PoolingDataSource();
source.setDataSourceName("A Data Source");
source.setServerName("localhost");
source.setDatabaseName("test");
source.setUser("testuser");
source.setPassword("testpassword");
source.setMaxConnections(10);
new InitialContext().rebind("DataSource", source);
```

Then code to use a connection from the pool might look like this:

```
Connection con = null;
try {
    DataSource source = (DataSource)new InitialContext().lookup("DataSource");
    con = source.getConnection();
    // use connection
} catch (SQLException e) {
    // log error
} catch (NamingException e) {
    // DataSource wasn't found in JNDI
} finally {
    if (con != null) {
        try { con.close(); } catch (SQLException e) {}
    }
}
```