Contents

Azure Key Vault sample version 6.0.0

Azure Key Vault sample version 6.2.2

Azure Key Vault sample version 7.0.0

Microsoft JDBC Driver for SQL Server

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

In our continued commitment to interoperability, Microsoft provides a Java Database Connectivity (JDBC) driver for use with SQL Server, and Azure SQL Database. The driver is available at no additional charge and provides Java database connectivity from any Java application, application server, or Java-enabled applet. This driver is a Type 4 JDBC driver that provides database connectivity through the standard JDBC application program interfaces (APIs).

The Microsoft JDBC Driver for SQL Server has been tested against major application servers such as IBM WebSphere, and SAP NetWeaver.

Getting Started

- Step 1: Configure development environment for Java development
- Step 2: Create a SQL database for Java development
- Step 3: Proof of concept connecting to SQL using Java

Documentation

- Getting Started
- Overview
- Programming Guide
- Security
- Performance and Reliability
- Troubleshooting
- Code Samples
- Compliance and Legal

Community

Finding Additional JDBC Driver Information

Download

Download Microsoft JDBC Driver for SQL Server - has additional information about Maven projects, and more.

Samples

- Sample JDBC Driver Applications
- Getting Started with Java on Windows
- Getting Started with Java on macOS
- Getting Started with Java on Ubuntu
- Getting Started with Java on Red Hat Enterprise Linux (RHEL)
- Getting Started with Java on SUSE Linux Enterprise Server (SLES)

Sample JDBC Driver Applications

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server sample applications demonstrate various features of the JDBC driver. Additionally, they demonstrate good programming practices that you can follow when using the JDBC driver with a SQL Server database.

All the sample applications are contained in *.java code files that can be compiled and run on your local computer, and they are located in various subfolders in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples
```

The topics in this section describe how to configure and run the sample applications, and include a discussion of what the sample applications demonstrate.

In This Section

TOPIC	DESCRIPTION
Connecting and Retrieving Data	These sample applications demonstrate how to connect to a SQL Server database. They also demonstrate different ways in which to retrieve data from a SQL Server database.
Working with Data Types (JDBC)	These sample applications demonstrate how to use the JDBC driver data type methods to work with data in a SQL Server database.
Working with Result Sets	These sample applications demonstrate how to use result sets to process data contained in a SQL Server database.
Working with Large Data	These sample applications demonstrate how to use adaptive buffering to retrieve large-value data from a SQL Server database without the overhead of server cursors.
SQL Data Discovery and Classification	This sample application demonstrates how to retreive Data Discovery and Classification information contained in a SQL Server database from a ResultSet object using JDBC Driver.

See Also

Overview of the JDBC Driver

JDBC Driver API Reference

8/13/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server provides an API that can be used within Java programming code to connect to and interact with a Microsoft SQL Server database.

JavaDoc.io Website is Primary

The Microsoft JDBC API Reference documentation is hosted for your viewing on the JavaDoc.io website. JavaDoc.io is now our primary website for JDBC reference documentation. Our JDBC reference documentation on JavaDoc.io is available at the following direct link:

- https://javadoc.io/doc/com.microsoft.sqlserver/mssql-jdbc/

JavaDoc.io has our JDBC reference documentation starting with version 6.0.

Only Legacy JDBC Documentation is Here on Docs

The JDBC API Reference documentation articles here on **https://docs.microsoft.com/sql/connect/jdbc/reference/** are no longer being updated when the JDBC classes are updated for new releases. However, the articles here do contain all the reference for JDBC versions 4.1 and 4.2.

Documentation for JDBC version 6.0, and some later versions, is also here. But for any version 6.0 or later, use the JavaDoc.io website.

Important Notes

NOTE

For conceptual information about using the JDBC driver, see Overview of the JDBC Driver.

IMPORTANT

For JDBC 4.1 and 4.2 compliance support, use Microsoft JDBC Driver 4.2 (or higher) for SQL Server. The previous Microsoft JDBC Drivers 4.1 and 4.0 releases do not support new methods introduced with JDBC 4.1 or 4.2.

API details for JDBC 4.1 compliance are not in this section. See JDBC 4.1 Compliance for the JDBC Driver.

API details for JDBC 4.2 compliance are not found in this section. See JDBC 4.2 Compliance for the JDBC Driver.

API details for Bulk Copy, available starting with Microsoft JDBC Driver 4.2 for SQL Server, are not found in this section. See Using Bulk Copy with the JDBC Driver.

API details for Always Encrypted, available starting with Microsoft JDBC Driver 6.0 for SQL Server, are not found in this section. See Always Encrypted API Reference for the JDBC Driver

API details for Using Table-Valued Parameters, available starting with Microsoft JDBC Driver 6.0 for SQL Server, are not found in this section. See Using Table-Valued Parameters

Microsoft JDBC Driver 6.4 supports compilation with JDK 7.0, 8.0, and 9.0.

Microsoft JDBC Driver 6.2 supports compilation with JDK 7.0, and 8.0.

Microsoft JDBC Drivers 6.0 and 4.2 support compilation with JDK 5.0, 6.0, 7.0, and 8.0.

Microsoft JDBC Driver 4.1 supports compilation with JDK 5.0, 6.0, and 7.0.

Interfaces

INTERFACE NAME	DESCRIPTION
ISQLServerCallableStatement Interface	Lets you specify the stored procedure name to call along with input and output parameters.
ISQLServerConnection Interface	Represents a JDBC connection to a SQL Server database.
SQLServerDataSource Class	Represents a list of properties specific to connecting to a SQL Server database by using a ISQLServerConnection object.
ISQLServerPreparedStatement	Represents the basic implementation of JDBC prepared statement functionality.
ISQLServerResultSet	Represents a JDBC result set.
ISQLServerStatement	Represents the basic implementation of JDBC statement functionality.

Classes

CLASS NAME	DESCRIPTION
DateTimeOffset	Represents an object of type microsoft.sql.DateTimeOffset.
SQLServerBlob	Represents a binary large object (BLOB).
SQLServerCallableStatement	Implements ISQLServerCallableStatement.
SQLServerClob	Represents a character large binary object (CLOB).
SQLServerConnection	Implements ISQLServerConnectopn.
SQLServerConnectionPoolDataSource	Represents physical database connections for connection pool managers.
SQLServerDatabaseMetaData	Represents the metadata for the database.
SQLServerDataSource	Represents a list of properties specific to connecting to a SQL Server database by using a SQLServerConnection object.
SQLServerDataSourceObjectFactory	Represents an object factory to materialize data sources from the Java Naming and Directory Interface (JNDI).
SQLServerDriver	Represents the JDBC driver. This class includes methods for connecting to a SQL Server database, and for obtaining information about the JDBC driver.
SQLServerException	Represents an unsuccessful or incomplete running of an SQL statement.

CLASS NAME	DESCRIPTION
SQLServerNClob Class	Represents a character large binary object using the National Character Set.
SQLServerParameterMetaData	Represents the metadata for prepared statement parameters.
SQLServerPooledConnection	Represents a physical database connection in a connection pool.
SQLServerPreparedStatement	Implements ISQLServerPreparedStatement.
SQLServerResource	Represents a localized error string resource. This class is intended for internal use only.
SQLServerResultSet	Implements ISQLServerResultSet.
SQLServerResultSetMetaData	Represents the metadata of the columns that are contained within a result set.
SQLServerSavepoint	Represents the checkpoint to which a transaction can be rolled back.
SQLServerStatement	Implements ISQLServerStatement.
SQLServerXAConnection	Represents JDBC connections that can participate in distributed (XA) transactions.
SQLServerXADataSource	Represents a factory for SQLServerXAConnection objects that is used internally.
SQLServerXAResource	Represents an XAResource for XA distributed transaction management.

See Also

Overview of the JDBC Driver

Getting Started with the JDBC Driver

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Getting Started

- Step 1: Configure development environment for Java development
- Step 2: Create a SQL database for Java development
- Step 3: Proof of concept connecting to SQL using Java

Step 1: Configure development environment for Java development

5/3/2018 • 2 minutes to read • Edit Online

Windows

- Identify which version of the JDBC driver you will use, based on your environment, as noted here: System Requirements for the JDBC Driver
- Download and install applicable JDBC Driver here: Download Microsoft JDBC Driver for SQL Server
- Set class path based on the driver version, as noted here: Using the JDBC Driver

Step 2: Create a SQL database for Java development

5/3/2018 • 2 minutes to read • Edit Online

The samples in this section only work with the AdventureWorks schema, on either Microsoft SQL Server or Azure SQL Database.

Azure SQL Database

Create a SQL database in minutes using the Azure portal

Microsoft SQL Server

Microsoft SQL Server Samples on GitHub

Step 3: Proof of concept connecting to SQL using Java

7/26/2018 • 2 minutes to read • Edit Online

This example should be considered a proof of concept only. The sample code is simplified for clarity, and doesn't necessarily represent best practices recommended by Microsoft.

Step 1: Connect

Use the connection class to connect to SQL Database.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class SQLDatabaseConnection {
    // Connect to your database.
    // Replace server name, username, and password with your credentials
    public static void main(String[] args) {
        String connectionUrl =
                "jdbc:sqlserver://yourserver.database.windows.net:1433;"
                        + "database=AdventureWorks;"
                        + "user=yourusername@yourserver;"
                        + "password=yourpassword;"
                        + "encrypt=true;"
                        + "trustServerCertificate=false;"
                        + "hostNameInCertificate=*.database.windows.net;"
                        + "loginTimeout=30;";

        try (Connection connection = DriverManager.getConnection(connectionUrl);) {
            // Code here.
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Step 2: Execute a query

In this sample, connect to Azure SQL Database, execute a SELECT statement, and return selected rows.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class SQLDatabaseConnection {

    // Connect to your database.
    // Replace server name, username, and password with your credentials
    public static void main(String[] args) {
        String connectionUrl =
                "jdbc:sqlserver://yourserver.database.windows.net:1433;"
                + "database=AdventureWorks;"
                + "user=yourusername@yourserver;"
                + "password=yourpassword;"
                + "encrypt=true;"
                + "trustServerCertificate=false;"
                + "hostNameInCertificate=*.database.windows.net;"
                + "loginTimeout=30;";

        ResultSet resultSet = null;

        try (Connection connection = DriverManager.getConnection(connectionUrl);
                Statement statement = connection.createStatement();) {

            // Create and execute a SELECT SQL statement.
            String selectSql = "SELECT TOP 10 Title, FirstName, LastName from SalesLT.Customer";
            resultSet = statement.executeQuery(selectSql);

            // Print results from select statement
            while (resultSet.next()) {
                System.out.println(resultSet.getString(2) + " " + resultSet.getString(3));
            }
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Step 3: Insert a row

In this example, execute an INSERT statement, pass parameters, and retrieve the auto-generated Primary Key value.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;

public class SQLDatabaseConnection {

    // Connect to your database.
    // Replace server name, username, and password with your credentials
    public static void main(String[] args) {
        String connectionUrl =
                "jdbc:sqlserver://yourserver.database.windows.net:1433;"
                        + "database=AdventureWorks;"
                        + "user=yourusername@yourserver;"
                        + "password=yourpassword;"
                        + "encrypt=true;"
                        + "trustServerCertificate=false;"
                        + "hostNameInCertificate=*.database.windows.net;"
                        + "loginTimeout=30;";

        String insertSql = "INSERT INTO SalesLT.Product (Name, ProductNumber, Color, StandardCost, ListPrice,
SellStartDate) VALUES "
                + "('NewBike', 'BikeNew', 'Blue', 50, 120, '2016-01-01');";

        ResultSet resultSet = null;

        try (Connection connection = DriverManager.getConnection(connectionUrl);
                PreparedStatement prepsInsertProduct = connection.prepareStatement(insertSql,
Statement.RETURN_GENERATED_KEYS);) {

            prepsInsertProduct.execute();
            // Retrieve the generated key from the insert.
            resultSet = prepsInsertProduct.getGeneratedKeys();

            // Print the ID of the inserted row.
            while (resultSet.next()) {
                System.out.println("Generated: " + resultSet.getString(1));
            }
        }
        // Handle any errors that may have occurred.
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Additional Samples

Sample JDBC Driver Applications

Overview of the JDBC Driver

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server is a Type 4 Java Database Connectivity (JDBC) 4.2 compliant driver that provides robust data access to SQL Server 2017, SQL Server 2016, SQL Server 2014, SQL Server 2012, SQL Server 2008 R2, SQL Server 2008, and Azure SQL Database.

The topics in this section provide a general overview of the JDBC driver, including the system requirements needed to use it, how it can be used, and where you can go for more information.

In This Section

TOPIC	DESCRIPTION
Download Microsoft JDBC Driver for SQL Server	Download links for Microsoft JDBC driver for SQL Server
Release Notes for the JDBC Driver	Describes the features that have been added to the current release of the Microsoft JDBC driver.
System Requirements for the JDBC Driver	Describes the system requirements needed to use the Microsoft JDBC driver.
Using the JDBC Driver	Describes how to configure your environment to use the Microsoft JDBC driver and how to make a simple connection to a SQL Server database.
Understanding Java EE Support	Describes how to use the Microsoft JDBC driver within a Java Platform, Enterprise Edition (Java EE) environment.
Deploying the JDBC Driver	Describes how to deploy the Microsoft JDBC driver on Windows and Unix operating systems.
Redistributing the Microsoft JDBC Driver	Describes how to register to redistribute the Microsoft JDBC driver.
Finding Additional JDBC Driver Information	Describes where to find additional resources about the Microsoft JDBC driver, including links to external resources.
Microsoft JDBC Driver for SQL Server Support Matrix	Support matrix and support lifecycle policy for the Microsoft JDBC driver for SQL Server.
Frequently Asked Questions (FAQ) for JDBC Driver	Frequently asked questions about the Microsoft JDBC driver.
Feature dependencies of Microsoft JDBC Driver for SQL Server	Feature dependencies of Microsoft JDBC Driver for SQL Server.

See Also

JDBC Driver GitHub Repository

Download Microsoft JDBC Driver for SQL Server

8/2/2018 • 2 minutes to read • Edit Online

Using the JDBC Driver with Maven Central

The JDBC Driver can be added to a Maven project by adding it as a dependency in the POM.xml file with the following code:

```
<dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
    <version>7.0.0.jre10</version>
</dependency>
```

Available downloads of JDBC Driver for SQL Server

- Microsoft JDBC Driver 7.0 for SQL Server
- Microsoft JDBC Driver 6.4 for SQL Server
- Microsoft JDBC Driver 6.2 for SQL Server
- Microsoft JDBC Driver 6.0 for SQL Server
- Microsoft JDBC Driver 4.2 for SQL Server
- Microsoft JDBC Driver 4.1 for SQL Server

Unsupported Drivers

Unsupported driver versions are not available for download here. We are continually improving the Java connectivity support. As such we highly recommend that you work with the latest version of Microsoft JDBC driver.

Release Notes for the JDBC Driver

8/8/2018 • 10 minutes to read • Edit Online

Download JDBC Driver

Updates in Microsoft JDBC Driver 7.0 for SQL Server

The Microsoft JDBC Driver 7.0 for SQL Server is fully compliant with JDBC API Specification 4.2. The jars in the 7.0 package are named according to Java version compatibility. For example, the mssql-jdbc-7.0.0.jre10.jar file from the 7.0 package should be used with Java 10.

Support for JDK 10

The Microsoft JDBC Driver 7.0 for SQL Server is now compatible with Java Development Kit (JDK) version 10.0 in addition to JDK 1.8. This update also exposes the driver's 'Automatic-Module-Name' as `com.microsoft.sqlserver.jdbc` through its MANIFEST file.

Support for Spatial Datatypes

The Microsoft JDBC Driver 7.0 for SQL Server now provides support for SQL Server Spatial Datatypes 'Geography' and 'Geometry'. For more information about Spatial datatypes APIs and how to use them, see here.

Implementation for JDBC 4.3 introduced java.sql.Connection APIs beginRequest() and endRequest()

The Microsoft JDBC Driver 7.0 for SQL Server now implements `beginRequest()` and `endRequest()` APIs from `java.sql.Connection` class. These APIs were introduced with JDBC 4.3 Specifications and JDK 9. For more information about the driver's implementation of these APIs, see here.

Support for 'SQL data discovery and classification'

The Microsoft JDBC Driver 7.0 for SQL Server provides support for 'SQL data discovery and classification' feature with any target database that supports this feature. The driver now exposes `SQLServerResultSet.getSensitivityClassification()` APIs to extract this information from the fetched ResultSet.

For more information about how to use this feature with JDBC Driver, refer sample here.

Added new connection property: useBulkCopyForBatchInsert

The Microsoft JDBC Driver 7.0 for SQL Server introduces a new connection property, 'useBulkCopyForBatchInsert', which is only supported for **Azure Data Warehouse**.

This property is **disabled** by default and can be enabled to increase performance of user applications when pushing large amounts data to Azure Data Warehouse. Enabling this property changes the behavior of Batch Insert operations to switch to Bulk Copy operations with user provided data. For more information about this property and its limitations, refer here.

Added new connection property: cancelQueryTimeout

The Microsoft JDBC Driver 7.0 for SQL Server introduces new connection property, `cancelQueryTimeout`, to cancel `queryTimeout` on `java.sql.Connection` and `java.sql.Statement` objects.

Added Azure Key Vault Provider Constructors

The Microsoft JDBC Driver 7.0 for SQL Server reintroduces a previously removed constructor, for `SQLServerColumnEncryptionAzureKeyVaultProvider`, which allowed authentication using a custom method implemented over `SQLServerKeyVaultAuthenticationCallback` to fetch an access token.

The new constructors have the below definition:

```
/* This constructor is added to provide backwards compatibility with 6.0
* version of the driver. It is marked deprecated for removal in next
* stable release.
*/
@Deprecated
public SQLServerColumnEncryptionAzureKeyVaultProvider(
        SQLServerKeyVaultAuthenticationCallback authenticationCallback,
        ExecutorService executorService) throws SQLServerException;

/*New Constructor to replace the above constructor*/
public SQLServerColumnEncryptionAzureKeyVaultProvider(
            SQLServerKeyVaultAuthenticationCallback authenticationCallback) throws SQLServerException;
```

Updated ADAL4J version to 1.6.0

The Microsoft JDBC Driver 7.0 for SQL Server has updated its maven dependency upon azure-activedirectory-library-for-java (ADAL4J) to version 1.6.0. For more information about dependencies, see here.

Updates in Microsoft JDBC Driver 6.4 for SQL Server

The Microsoft JDBC Driver 6.4 for SQL Server is fully compliant with JDBC specifications 4.1 and 4.2. The jars in the 6.4 package are named according to Java version compatibility. For example, the mssql-jdbc-6.4.0.jre8.jar file from the 6.4 package must be used with Java 8.

Support for JDK 9

Support for Java Development Kit (JDK) version 9.0 in addition to JDK 8.0 and 7.0.

JDBC 4.3 compliance

Support for Java Database Connectivity API 4.3 specification, in addition to 4.1 and 4.2. The JDBC 4.3 API methods are added but not implemented yet. For details see JDBC 4.3 Compliance for the JDBC Driver.

Added new connection property: sslProtocol

Added a new connection property that lets users specify the TLS protocol keyword. Possible values are: "TLS", "TLSv1", "TLSv1.1", "TLSv1.2". See SSLProtocol for details.

Deprecated connection property: fipsProvider

Connection property "fipsProvider" is removed from the list of accepted connection properties. See the details Here.

Added connection properties for specifying custom TrustManager

Driver now supports specifying custom TrustManager with added "trustManagerClass" and "trustManagerConstructorArg" connection properties. This allows for dynamic specification of a set of certificates that are trusted on a per connection basis without modifying the global settings for the JVM environment.

Added support for datetime/smallDatetime in Table-Valued Parameters (TVP)

Driver now supports datatypes DATETIME and SMALLDATETIME when using Table-Valued Parameters (TVP).

Added support for sql_variant datatype

The JDBC Driver now supports sql_variant datatypes to be used with SQL Server. Sql_variant is also supported with features such as Table-Valued Parameters (TVP) and BulkCopy with below limitations:

1. For Date values: When using TVP to populate a table that contains datetime/smalldatetime/date values stored in sql_variant column, calling getDateTime()/getSmallDateTime()/getDate() methods on resultset doesn't work and throws the following exception:
   `java java.lang.String cannot be cast to java.sql.Timestamp` Workaround: use "getString()" or "getObject()" methods instead.

2. Using TVP with SQL Variant for null values

If you're using TVP to populate a table and send NULL value to sql_variant column type, you'll encounter an exception as inserting NULL value with column type sql_variant in TVP is currently not supported.

Implemented Prepared Statement Metadata Caching

The JDBC Driver has implemented Prepared Statement Metadata Caching for performance improvement. Driver now supports caching Prepared Statement metadata in the driver with "disableStatementPooling" and "statementPoolingCacheSize" connection properties. This feature is disabled by default. More information can be found here

Added support for AAD Integrated Authentication on Linux/Mac

The JDBC Driver now also supports Azure Active Directory Integrated Authentication on all supported Operating Systems (Windows/Linux/Mac) with Kerberos. Alternatively, on Windows Operating Systems, users can authenticate with sqljdbc_auth.dll.

Updated ADAL4J version to 1.4.0

The JDBC Driver has updated its maven dependency upon azure-activedirectory-library-for-java (ADAL4J) to version 1.4.0. For more information about dependencies, see here

Updates in Microsoft JDBC Driver 6.2 for SQL Server

The Microsoft JDBC Driver 6.2 for SQL Server is fully compliant with JDBC specifications 4.1 and 4.2. The jars in the 6.2 package are named according to Java version compatibility. For example, the mssql-jdbc-6.2.2.jre8.jar file from the 6.2 package is recommended to be used with Java 8.

NOTE

An issue with the metadata caching improvement was found in the JDBC 6.2 RTW released on June 29, 2017. The improvement was rolled back and new jars (version 6.2.1) were released on July 17, 2017.

Another improvement to upgrade Azure Key Vault dependent library version to 1.0.0 was made, and new jars (version 6.2.2) were released on October 19, 2017.

Download the latest updates in JDBC Driver 6.2 on Microsoft Download Center, GitHub, and Maven Central. Please update your projects to use the 6.2.2 release jars. Please view release notes for v6.2.1 and v6.2.2 for more details.

Azure Active Directory (AAD) support for Linux

Connect your Linux applications to Azure SQL Database using AAD authentication via username/password and access token methods.

Federal Information Processing Standard (FIPS) enabled JVMs

The JDBC Driver can now be used on JVMs that run in FIPS 140 compliance mode to meet federal standards and compliance.

Kerberos Authentication Improvements

The JDBC Driver now has support for:

- Principal/Password method for applications where the Kerberos configuration can't be modified or unable to retrieve a new token or keytab. This method can be used for authenticating to a SQL Server that only allows Kerberos authentication.
- Cross-realm authentication using Kerberos Integrated authentication without explicitly setting the server SPN. The driver now automatically computes the REALM even when it hasn't been provided.
- Kerberos Constrained Delegation by accepting impersonated user credentials as a GSS Credential object via data source. This impersonated credential is then used to establish a Kerberos connection.

Added Timeouts

The JDBC Driver now supports the following configurable timeouts you can change based on your application's needs:

- Query Timeout to control the number of seconds to wait before a timeout occurs when running a query.
- Socket Timeout to specify the number of milliseconds to wait before a timeout occurs on a socket read or accept.

Updates in Microsoft JDBC Driver 6.1 for SQL Server

The Microsoft JDBC Driver 6.1 for SQL Server is fully compliant with JDBC specifications 4.1 and 4.2. This is the initial open-source release of the JDBC Driver and contains the mssql-jdbc-6.1.0.jre8.jar mssql-jdbc-6.1.0.jre7.jar files, which correspond to the Java version compatibility.

Updates in Microsoft JDBC Driver 6.0 for SQL Server

The Microsoft JDBC Driver 6.0 for SQL Server is fully compliant with JDBC specifications 4.1 and 4.2. The jars in the 6.0 package are named according to their compliance with the JDBC API version. For example, the sqljdbc42.jar file from the 6.0 package is JDBC API 4.2 compliant. Similarly, the sqljdbc41.jar file is compliant with JDBC API 4.1.

To ensure you have the right sqljdbc42.jar or sqljdbc41.jar, run the following lines of code. If the output is "Driver version: 6.0.7507.100", you have the JDBC Driver 6.0 package.

```
Connection conn = DriverManager.getConnection("jdbc:sqlserver://<server>;user=<user>;password=<password>;");
System.out.println("Driver version: " + conn.getMetaData().getDriverVersion());
```

Always Encrypted

Support for the recently released Always Encrypted feature in SQL Server 2016, a new security feature that ensures sensitive data is never seen in plaintext in a SQL Server instance. Always Encrypted works by transparently encrypting the data in the application, so that SQL Server will only handle the encrypted data and not plaintext values. Even if the SQL instance or the host machine is compromised, all an attacker can get is ciphertext of sensitive data. For details, see Using Always Encrypted with the JDBC Driver.

Internationalized Domain Name (IDN)

Support for Internationalized Domain Names (IDNs) for server names. For details see Using International Domain Names on the International Features of the JDBC Driver page.

Parameterized Query

Now supports retrieving parameter metadata with prepared statements for complex queries such as subqueries and/or joins. Note that this improvement is available only when using SQL Server 2012 and newer versions.

Azure Active Directory (AAD)

AAD authentication is a mechanism of connecting to Azure SQL Database v12 using identities in AAD. Use AAD authentication to centrally manage identities of database users and as an alternative to SQL Server authentication. The JDBC Driver 6.0 allows you to specify your AAD credentials in the JDBC connection string to connect to Azure SQL DB. For details see the authentication property on the Setting the Connection Properties page.

Table-Valued Parameters

Table-valued parameters provide an easy way to marshal multiple rows of data from a client application to SQL Server without requiring multiple round trips or special server-side logic for processing the data. You can use table-valued parameters to encapsulate rows of data in a client application and send the data to the server in a single parameterized command. The incoming data rows are stored in a table variable that can then be operated on by using Transact-SQL. For details, see Using Table-Valued Parameters.

AlwaysOn Availability Groups (AG)

The driver now supports transparent connections to AlwaysOn Availability Groups. The driver quickly discovers the current AlwaysOn topology of your server infrastructure and connects to the current active server transparently.

Updates in Microsoft JDBC Driver 4.2 for SQL Server and later

The Microsoft JDBC Driver 4.2 for SQL Server is fully compliant with JDBC specifications 4.1 and 4.2. The jars in the 4.2 package are named according to their compliance with the JDBC API version. For example, the sqljdbc42.jar file from the 4.2 package is JDBC API 4.2 compliant. Similarly, the sqljdbc41.jar file is compliant with JDBC API 4.1.

To ensure you have the right sqljdbc42.jar or sqljdbc41.jar, run the following lines of code. If the output is "Driver version: 4.2.6420.100", you have the JDBC Driver 4.2 package.

```
Connection conn = DriverManager.getConnection("jdbc:sqlserver://<server>;user=<user>;password=<password>;");
System.out.println("Driver version: " + conn.getMetaData().getDriverVersion());
```

Support for JDK 8

Support for Java Development Kit (JDK) version 8.0 in addition to JDK 7.0, 6.0, and 5.0.

JDBC 4.1 and 4.2 compliance

Support for Java Database Connectivity API 4.1 and 4.2 specifications, in addition to 4.0. For details, see JDBC 4.1 Compliance for the JDBC Driver and JDBC 4.2 Compliance for the JDBC Driver.

Bulk copy

The bulk copy feature is used to quickly copy large amounts of data into tables or views in SQL Server databases. For details, see Using Bulk Copy with the JDBC Driver.

XA transaction rollback option

Added new timeout options for existing automatic rollback of unprepared transactions. For details, see Understanding XA Transactions.

New Kerberos Principal Connection Property

Added a new connection property to facilitate flexibility with Kerberos connections. For details, see Using Kerberos Integrated Authentication to Connect to SQL Server.

Updates in Microsoft JDBC Driver 4.1 for SQL Server and later

Support for JDK 7

Support for Java Development Kit (JDK) version 7.0 in addition to JDK 6.0 and 5.0.

Itanium Not Supported for JDBC Driver 6.4, 6.0, 4.2, and 4.1 Applications

Microsoft JDBC Drivers 6.4, 6.0, 4.2, and 4.1 for SQL Server applications aren't supported to run on an Itanium computer.

See Also

Overview of the JDBC Driver

System Requirements for the JDBC Driver

8/13/2018 • 7 minutes to read • Edit Online

Download JDBC Driver

To access data from a SQL Server or Azure SQL Database by using the Microsoft JDBC Driver for SQL Server, you must have the following components installed on your computer:

- Microsoft JDBC Driver for SQL Server (download)
- Java Runtime Environment

Java Runtime Environment Requirements

Starting with the Microsoft JDBC Driver 7.0 for SQL Server, Sun Java SE Development Kit (JDK) 10.0 and Java Runtime Environment (JRE) 10.0 are supported.

Starting with the Microsoft JDBC Driver 6.4 for SQL Server, Sun Java SE Development Kit (JDK) 9.0 and Java Runtime Environment (JRE) 9.0 are supported.

Starting with the Microsoft JDBC Driver 4.2 for SQL Server, Sun Java SE Development Kit (JDK) 8.0 and Java Runtime Environment (JRE) 8.0 are supported. Support for Java Database Connectivity (JDBC) Spec API has been extended to include the JDBC 4.1 and 4.2 API.

Starting with the Microsoft JDBC Driver 4.1 for SQL Server, Sun Java SE Development Kit (JDK) 7.0 and Java Runtime Environment (JRE) 7.0 are supported.

Starting with the Microsoft JDBC Driver 4.0 for SQL Server, the JDBC driver support for Java Database Connectivity (JDBC) Spec API has been extended to include the JDBC 4.0 API. The JDBC 4.0 API was introduced as part of the Sun Java SE Development Kit (JDK) 6.0 and Java Runtime Environment (JRE) 6.0. JDBC 4.0 is a superset of the JDBC 3.0 API.

When you deploy the Microsoft JDBC Driver for SQL Server on Windows and UNIX operating systems, you must use the installation packages, *sqljdbc_<version>_enu.exe*, and *sqljdbc_<version>_enu.tar.gz*, respectively. For more information about how to deploy the JDBC Driver, see Deploying the JDBC Driver topic.

Microsoft JDBC Driver 7.0 for SQL Server:

The JDBC Driver 7.0 includes two JAR class libraries in each installation package: **mssql-jdbc-7.0.0.jre8.jar**, and **mssql-jdbc-7.0.0.jre10.jar**.

The JDBC Driver 7.0 is designed to work with and be supported by all major Sun equivalent Java virtual machines, but is tested only on Sun JRE 8.0, and 10.0.

The following summarizes support provided by the two JAR files included with Microsoft JDBC Drivers 7.0 for SQL Server:

JAR	JDBC VERSION COMPLIANCE	RECOMMENDED JAVA VERSION	DESCRIPTION

JAR	JDBC VERSION COMPLIANCE	RECOMMENDED JAVA VERSION	DESCRIPTION
mssql-jdbc-7.0.0.jre8.jar	4.2	8	Requires a Java Runtime Environment (JRE) 8.0. Using JRE 7.0 or lower throws an exception. New Features in 7.0 include: JDK 10 support, updated default compliance level to JDBC 4.2 specifications, Spatial Datatypes support, cancelQueryTimeout connection property, Boundary Request methods, useBulkCopyForBatchInsert connection property, Data Discovery and Classification information, UTF-8 feature extension, and CityHash support.
mssql-jdbc-7.0.0.jre10.jar	4.3	10	Requires a Java Runtime Environment (JRE) 10.0. Using JRE 9.0 or lower throws an exception. New Features in 7.0 include: JDK 10 support, updated default compliance level to JDBC 4.2 specifications, Spatial Datatypes support, cancelQueryTimeout connection property, Boundary Request methods, useBulkCopyForBatchInsert connection property, Data Discovery and Classification information, UTF-8 feature extension, and CityHash support.

The JDBC Driver 7.0 is also available on the Maven Central Repository and can be added to a Maven project by adding the following code in the POM.XML:

```
<dependency>
   <groupId>com.microsoft.sqlserver</groupId>
   <artifactId>mssql-jdbc</artifactId>
   <version>7.0.0.jre10</version>
</dependency>
```

Microsoft JDBC Driver 6.4 for SQL Server:

The JDBC Driver 6.4 includes three JAR class libraries in each installation package: **mssql-jdbc-6.4.0.jre7.jar**, **mssql-jdbc-6.4.0.jre8.jar**, and **mssql-jdbc-6.4.0.jre9.jar**.

The JDBC Driver 6.4 is designed to work with and be supported by all major Sun equivalent Java virtual machines, but is tested only on Sun JRE 7.0, 8.0, and 9.0.

The following summarizes support provided by the three JAR files included with Microsoft JDBC Drivers 6.4 for

SQL Server:

JAR	JDBC VERSION COMPLIANCE	RECOMMENDED JAVA VERSION	DESCRIPTION
mssql-jdbc-6.4.0.jre7.jar	4.1	7	Requires a Java Runtime Environment (JRE) 7.0. Using JRE 6.0 or lower throws an exception. New Features in 6.4 include: Azure AD authentication for Linux, Principal/Password method for Kerberos, automatic detection of REALM in SPN for Cross-Domain authentication, Kerberos Constrained Delegation, Query Timeout, Socket Timeout, and prepared statement handle re-use.
mssql-jdbc-6.4.0.jre8.jar	4.2	8	Requires a Java Runtime Environment (JRE) 8.0. Using JRE 7.0 or lower throws an exception. New Features in 6.4 include: Azure AD authentication for Linux, Principal/Password method for Kerberos, automatic detection of REALM in SPN for Cross-Domain authentication, Kerberos Constrained Delegation, Query Timeout, Socket Timeout, and prepared statement handle re-use.
mssql-jdbc-6.4.0.jre9.jar	4.3	9	Requires a Java Runtime Environment (JRE) 9.0. Using JRE 8.0 or lower throws an exception. New Features in 6.4 include: Azure AD authentication for Linux, Principal/Password method for Kerberos, automatic detection of REALM in SPN for Cross-Domain authentication, Kerberos Constrained Delegation, Query Timeout, Socket Timeout, and prepared statement handle re-use.

The JDBC Driver 6.4 is also available on the Maven Central Repository and can be added to a Maven project by adding the following code in the POM.XML

```
<dependency>
   <groupId>com.microsoft.sqlserver</groupId>
   <artifactId>mssql-jdbc</artifactId>
   <version>6.4.0.jre9</version>
</dependency>
```

Microsoft JDBC Driver 6.2 for SQL Server:

The JDBC Driver 6.2 includes two JAR class libraries in each installation package: **mssql-jdbc-6.2.2.jre7.jar**, and **mssql-jdbc-6.2.2.jre8.jar**.

The JDBC Driver 6.2 is designed to work with and be supported by all major Sun equivalent Java virtual machines, but is tested only on Sun JRE 5.0, 6.0, 7.0, and 8.0.

The following summarizes support provided by the two JAR files included with Microsoft JDBC Drivers 6.0 and 4.2 for SQL Server:

JAR	JDBC VERSION COMPLIANCE	RECOMMENDED JAVA VERSION	DESCRIPTION
mssql-jdbc-6.2.2.jre7.jar	4.1	7	Requires a Java Runtime Environment (JRE) 7.0. Using JRE 6.0 or lower throws an exception. New Features in 6.2 include: Azure AD authentication for Linux, Principal/Password method for Kerberos, automatic detection of REALM in SPN for Cross-Domain authentication, Kerberos Constrained Delegation, Query Timeout, Socket Timeout, and prepared statement handle re-use.
mssql-jdbc-6.2.3.jre8.jar	4.2	8	Requires a Java Runtime Environment (JRE) 8.0. Using JRE 7.0 or lower throws an exception. New Features in 6.2 include: Azure AD authentication for Linux, Principal/Password method for Kerberos, automatic detection of REALM in SPN for Cross-Domain authentication, Kerberos Constrained Delegation, Query Timeout, Socket Timeout, and prepared statement handle re-use

The JDBC Driver 6.2 is also available on the Maven Central Repository and can be added to a Maven project by adding the following code in the POM.XML

```
<dependency>
   <groupId>com.microsoft.sqlserver</groupId>
   <artifactId>mssql-jdbc</artifactId>
   <version>6.2.2.jre8</version>
</dependency>
```

Microsoft JDBC Driver 6.0 and 4.2 for SQL Server:

The JDBC Drivers 6.0 and 4.2 include two JAR class libraries in each installation package: **sqljdbc41.jar**, and **sqljdbc42.jar**.

The JDBC Drivers 6.0 and 4.2 are designed to work with and be supported by all major Sun equivalent Java virtual machines, but is tested only on Sun JRE 5.0, 6.0, 7.0, and 8.0.

The following summarizes support provided by the two JAR files included with Microsoft JDBC Drivers 6.0 and 4.2 for SQL Server:

JAR	JDBC VERSION COMPLIANCE	RECOMMENDED JAVA VERSION	DESCRIPTION
sqljdbc41.jar	4.1	7	Requires a Java Runtime Environment (JRE) 7.0. Using JRE 6.0 or lower throws an exception. New Features in 6.0 & 4.2 packages include: JDBC 4.1 Compliance and Bulk Copy In Addition, new Features in only the 6.0 package include: Always Encrypted, Table-Valued Parameters, Azure Active Directory Authentication, transparent connections to Always On Availability Groups, improvement in parameter metadata retrieval for prepared queries and Internationalized Domain Name (IDN)

JAR	JDBC VERSION COMPLIANCE	RECOMMENDED JAVA VERSION	DESCRIPTION
sqljdbc42.jar	4.2	8	Requires a Java Runtime Environment (JRE) 8.0. Using JRE 7.0 or lower throws an exception. New Features in 6.0 & 4.2 packages include: JDBC 4.1 Compliance, JDBC 4.2 Compliance, and Bulk Copy In Addition, new Features in only the 6.0 package include: Always Encrypted, Table-Valued Parameters, Azure Active Directory Authentication, transparent connections to Always On Availability Groups, improvement in parameter metadata retrieval for prepared queries and Internationalized Domain Name (IDN)

Microsoft JDBC Driver 4.1 for SQL Server:

The JDBC Driver 4.1 includes one JAR class library in each installation package: **sqljdbc41.jar**.

JAR	DESCRIPTION
sqljdbc41.jar	**sqljdbc41.jar** class library provides support for JDBC 4.0 API. It includes all of the features of the JDBC 4.0 driver as well as the JDBC 4.0 API methods. JDBC 4.1 is not supported (throws an exception “SQLFeatureNotSupportedException”). **sqljdbc41.jar** class library requires a Java Runtime Environment (JRE) 7.0. Using **sqljdbc41.jar** on JRE 6.0 and 5.0 throws an exception.

The JDBC driver is designed to work with and be supported by all major Sun equivalent Java virtual machines, but is tested on Sun JRE 5.0, 6.0 and 7.0.

The following summarizes support provided by the JAR file included with Microsoft JDBC Driver 4.1 for SQL Server.

JAR	JDBC VERSION	JRE (CAN RUN)	JDK (CAN COMPILE)
sqljdbc41.jar	4	7	7 6 5

SQL Server Requirements

The JDBC driver supports connections to Azure SQL database and SQL Server. For Microsoft JDBC Driver 4.2 and 4.1 for SQL Server, support begins with SQL Server 2008.

Operating System Requirements

The JDBC driver is designed to work on any operating system that supports the use of a Java Virtual Machine (JVM). However, only Sun Solaris, SUSE Linux, and Windows operating systems have officially been tested.

Supported Languages

The JDBC driver supports all SQL Server column collations. For more information about the collations supported by the JDBC driver, see International Features of the JDBC Driver.

For more information about collations, see "Working with Collations" in SQL Server Books Online.

See Also

Overview of the JDBC Driver

Using the JDBC Driver

8/13/2018 • 6 minutes to read • Edit Online

Download JDBC Driver

This section provides quickstart instructions for making a simple connection to a SQL Server database by using the Microsoft JDBC Driver for SQL Server. Before you connect to a SQL Server database, SQL Server must first be installed on either your local computer or a server, and the JDBC driver must be installed on your local computer.

Choosing the Right JAR file

The Microsoft JDBC Driver provides different Jars to be used in correspondence with your preferred Java Runtime Environment (JRE) settings, as under:

The Microsoft JDBC Driver 7.0 for SQL Server provides **mssql-jdbc-7.0.0.jre8.jar**, and **mssql-jdbc-7.0.0.jre10.jar** class library files.

The Microsoft JDBC Driver 6.4 for SQL Server provides **mssql-jdbc-6.4.0.jre7.jar**, **mssql-jdbc-6.4.0.jre8.jar**, and **mssql-jdbc-6.4.0.jre9.jar** class library files.

The Microsoft JDBC Driver 6.2 for SQL Server provides **mssql-jdbc-6.2.2.jre7.jar**, and **mssql-jdbc-6.2.2.jre8.jar** class library files.

The Microsoft JDBC Drivers 6.0 and 4.2 for SQL Server provide **sqljdbc41.jar**, and **sqljdbc42.jar** class library files.

The Microsoft JDBC Driver 4.1 for SQL Server provides the **sqljdbc41.jar** class library file.

Your choice will also determine available features. For more information about which JAR file to choose, see System Requirements for the JDBC Driver.

Setting the Classpath

The Microsoft JDBC driver jars are not part of the Java SDK and must be included in Classpath of user application.

If using JDBC Driver 4.1 or 4.2, set the classpath to include **sqljdbc41.jar** or **sqljdbc42.jar** file from respective driver download.

If using JDBC Driver 6.2, set the classpath to include the **mssql-jdbc-6.2.2.jre7.jar** or **mssql-jdbc-6.2.2.jre8.jar**.

If using JDBC Driver 6.4, set the classpath to include the **mssql-jdbc-6.4.0.jre7.jar**, **mssql-jdbc-6.4.0.jre8.jar, or **mssql-jdbc-6.4.0.jre9.jar**.

If using JDBC Driver 7.0, set the classpath to include the **mssql-jdbc-7.0.0.jre8.jar** or **mssql-jdbc-7.0.0.jre10.jar**.

If the classpath is missing an entry for the right Jar file, an application will throw the common `Class not found` exception.

For Microsoft JDBC Driver 7.0

The **mssql-jdbc-7.0.0.jre8.jar** or **mssql-jdbc-7.0.0.jre10.jar** files are installed in the following locations:

```
\<installation directory>\sqljdbc_<version>\<language>\mssql-jdbc-7.0.0.jre8.jar

\<installation directory>\sqljdbc_<version>\<language>\mssql-jdbc-7.0.0.jre10.jar
```

The following snippet is an example of the CLASSPATH statement that is used for a Windows application:

```
CLASSPATH =.;C:\Program Files\Microsoft JDBC Driver 7.0 for SQL Server\sqljdbc_7.0\enu\mssql-jdbc-7.0.0.jre10.jar
```

The following snippet is an example of the CLASSPATH statement that is used for a Unix/Linux application:

```
CLASSPATH =.:/home/usr1/mssqlserverjdbc/Driver/sqljdbc_7.0/enu/mssql-jdbc-7.0.0.jre10.jar
```

Make sure that the CLASSPATH statement contains only one Microsoft JDBC Driver for SQL Server, such as either **mssql-jdbc-7.0.0.jre8.jar** or **mssql-jdbc-7.0.0.jre10.jar**.

For Microsoft JDBC Driver 6.4

The **mssql-jdbc-6.4.0.jre7.jar**, **mssql-jdbc-6.4.0.jre8.jar, or **mssql-jdbc-6.4.0.jre9.jar** files are installed in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\mssql-jdbc-6.4.0.jre7.jar

\<installation directory>\sqljdbc_<version>\<language>\mssql-jdbc-6.4.0.jre8.jar

\<installation directory>\sqljdbc_<version>\<language>\mssql-jdbc-6.4.0.jre9.jar
```

The following snippet is an example of the CLASSPATH statement that is used for a Windows application:

```
CLASSPATH =.;C:\Program Files\Microsoft JDBC Driver 6.4 for SQL Server\sqljdbc_6.4\enu\mssql-jdbc-6.4.0.jre9.jar
```

The following snippet is an example of the CLASSPATH statement that is used for a Unix/Linux application:

```
CLASSPATH =.:/home/usr1/mssqlserverjdbc/Driver/sqljdbc_6.4/enu/mssql-jdbc-6.4.0.jre9.jar
```

Make sure that the CLASSPATH statement contains only one Microsoft JDBC Driver for SQL Server, such as either **mssql-jdbc-6.4.0.jre7.jar**, **mssql-jdbc-6.4.0.jre8.jar, or **mssql-jdbc-6.4.0.jre9.jar**.

For Microsoft JDBC Driver 6.2

The **mssql-jdbc-6.2.2.jre7.jar** or **mssql-jdbc-6.2.2.jre8.jar** files are installed in the following locations:

```
\<installation directory>\sqljdbc_<version>\<language>\mssql-jdbc-6.2.2.jre7.jar

\<installation directory>\sqljdbc_<version>\<language>\mssql-jdbc-6.2.2.jre8.jar
```

The following snippet is an example of the CLASSPATH statement that is used for a Windows application:

```
CLASSPATH =.;C:\Program Files\Microsoft JDBC Driver 6.2 for SQL Server\sqljdbc_6.2\enu\mssql-jdbc-6.2.2.jre8.jar
```

The following snippet is an example of the CLASSPATH statement that is used for a Unix/Linux application:

```
CLASSPATH =.:/home/usr1/mssqlserverjdbc/Driver/sqljdbc_6.2/enu/mssql-jdbc-6.2.2.jre8.jar
```

Make sure that the CLASSPATH statement contains only one Microsoft JDBC Driver for SQL Server, such as either mssql-jdbc-6.2.2.jre7.jar or mssql-jdbc-6.2.2.jre8.jar.

For Microsoft JDBC Driver 4.1, 4.2, and 6.0

The sqljdbc.jar file, sqljdbc4.jar file, sqljdbc41.jar, or sqljdbc42.jar file are installed in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\sqljdbc.jar

\<installation directory>\sqljdbc_<version>\<language>\sqljdbc4.jar

\<installation directory>\sqljdbc_<version>\<language>\sqljdbc41.jar

\<installation directory>\sqljdbc_<version>\<language>\sqljdbc42.jar
```

The following snippet is an example of the CLASSPATH statement that is used for a Windows application:

```
CLASSPATH =.;C:\Program Files\Microsoft JDBC Driver 6.0 for SQL Server\sqljdbc_4.2\enu\sqljdbc42.jar
```

The following snippet is an example of the CLASSPATH statement that is used for a Unix/Linux application:

```
CLASSPATH =.:/home/usr1/mssqlserverjdbc/Driver/sqljdbc_4.2/enu/sqljdbc42.jar
```

Make sure that the CLASSPATH statement contains only one Microsoft JDBC Driver for SQL Server, such as either sqljdbc.jar, sqljdbc4.jar, sqljdbc41.jar, or sqljdbc42.jar.

NOTE

On Windows systems, directory names longer than the 8.3 filename convention or folder names with spaces may cause problems with classpaths. If you suspect these types of issues, you should temporarily move the sqljdbc.jar file, sqljdbc4.jar file, or the sqljdbc41.jar file into a simple directory name such as `C:\Temp`, change the classpath, and determine whether that addresses the problem.

Applications that are run directly at the command prompt

The classpath is configured in the operating system. Append sqljdbc.jar, sqljdbc4.jar, or sqljdbc41.jar to the classpath of the system. Alternatively, you can specify the classpath on the Java command line that runs the application by using the `java -classpath` option.

Applications that run in an IDE

Each IDE vendor provides a different method for setting the classpath in its IDE. Just setting the classpath in the operating system will not work. You must add sqljdbc.jar, sqljdbc4.jar, or sqljdbc41.jar to the IDE classpath.

Servlets and JSPs

Servlets and JSPs are run in a servlet/JSP engine such as Tomcat. The classpath must be set according to the servlet/JSP engine documentation. Just setting the classpath in the operating system will not work. Some servlet/JSP engines provide setup screens that you can use to set the classpath of the engine. In that situation, you must append the correct JDBC Driver JAR file to the existing engine classpath and restart the engine. In other situations, you can deploy the driver by copying sqljdbc.jar, sqljdbc4.jar, or sqljdbc41.jar to a specific directory, such as lib, during engine installation. The engine driver classpath can also be specified in an engine-specific configuration file.

Enterprise Java Beans

Enterprise Java Beans (EJB) are run in an EJB container. EJB containers are sourced from various vendors. Java applets run in a browser but are downloaded from a web server. Copy sqljdbc.jar, sqljdbc4.jar, or sqljdbc41.jar to the web server root and specify the name of the JAR file in the HTML archive tab of the applet, for example, `<applet ... archive=mssql-jdbc-***.jar>`.

Making a Simple Connection to a Database

Using the sqljdbc.jar class library, applications must first register the driver as follows:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

When the driver is loaded, you can establish a connection by using a connection URL and the getConnection method of the DriverManager class:

```
String connectionUrl = "jdbc:sqlserver://localhost:1433;" +
   "databaseName=AdventureWorks;user=MyUserName;password=*****;";
Connection con = DriverManager.getConnection(connectionUrl);
```

Starting from JDBC API 4.0, the `DriverManager.getConnection()` method is enhanced to load JDBC drivers automatically. Therefore, applications do not need to call the `Class.forName` method to register or load the driver when using driver jar libraries.

When the getConnection method of the DriverManager class is called, an appropriate driver is located from the set of registered JDBC drivers. sqljdbc4.jar, sqljdbc41.jar, or sqljdbc42.jar file includes "META-INF/services/java.sql.Driver" file, which contains the **com.microsoft.sqlserver.jdbc.SQLServerDriver** as a registered driver. The existing applications, which currently load the drivers by using the Class.forName method, will continue to work without modification.

> **NOTE**
> sqljdbc4.jar, sqljdbc41.jar, or sqljdbc42.jar class library cannot be used with older versions of the Java Runtime Environment (JRE). See System Requirements for the JDBC Driver for the list of JRE versions supported by the Microsoft JDBC Driver for SQL Server.

For more information about how to connect with data sources and use a connection URL, see Building the Connection URL and Setting the Connection Properties.

See Also

Overview of the JDBC Driver

Understanding Java EE Support

8/2/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The following sections document how the Microsoft JDBC Driver for SQL Server provides support for the Java Platform, Enterprise Edition (Java EE) and JDBC 3.0 optional API features. The source code examples provided in this Help system provide a good reference for getting started with these features.

First, make sure that your Java environment (JDK, JRE) includes the javax.sql package. This is a required package for any JDBC application that uses the optional API. JDK 1.5 and later versions already contain this package, so you don't have to install it separately.

Driver Name

The driver class name is **com.microsoft.sqlserver.jdbc.SQLServerDriver**. For JDBC Drivers 4.1, 4.2, and 6.0, the driver is contained in the **sqljdbc.jar**, **sqljdbc4.jar**, **sqljdbc41.jar**, or **sqljdbc42.jar** files.

For JDBC Driver 6.2, the driver is contained in **mssql-jdbc-6.2.2.jre7.jar** or **mssql-jdbc-6.2.2.jre8.jar**.

For JDBC Driver 6.4, the driver is contained in **mssql-jdbc-6.4.0.jre7.jar**, **mssql-jdbc-6.4.0.jre8.jar**, or **mssql-jdbc-6.4.0.jre9.jar**.

For JDBC Driver 7.0, the driver is contained in **mssql-jdbc-7.0.0.jre8.jar**, or **mssql-jdbc-7.0.0.jre10.jar**.

The class name is used whenever you load the driver with the JDBC DriverManager class. It's also used whenever you must specify the class name of the driver in any driver configuration. For example, configuring a data source within a Java EE application server might require you enter the driver class name.

Data Sources

The JDBC driver provides support for Java EE / JDBC 3.0 data sources. The JDBC driver SQLServerXADataSource class is implemented by `com.microsoft.sqlserver.jdbc.SQLServerXADataSource`.

Datasource Names

You can make database connections by using data sources. The data sources available with JDBC driver are described in the following table:

DATASOURCE TYPE	CLASS NAME AND DESCRIPTION
DataSource	`com.microsoft.sqlserver.jdbc.SQLServerDataSource` The non pooling data source.
ConnectionPoolDataSource	`com.microsoft.sqlserver.jdbc.SQLServerConnectionPoolDataSource` The data source to configure JAVA EE application server connection pools. Typically used when the application runs within a JAVA EE application server.

DATASOURCE TYPE	CLASS NAME AND DESCRIPTION
XADataSource	`com.microsoft.sqlserver.jdbc.SQLServerXADataSource` The data source to configure JAVA EE XA data sources. Typically used when the application runs within a JAVA EE application server and an XA transaction manager.

Data Source Properties

All data sources support the ability to set and get any property that is associated with the underlying driver's property set.

Examples:

`setServerName("localhost");`

`setDatabaseName("AdventureWorks");`

The following shows how an application connects by using a data source:

```
//initialize JNDI ..
Context ctx = new InitialContext(System.getProperties());
...
DataSource ds = (DataSource) ctx.lookup("MyDataSource");
Connection c = ds.getConnection("user", "pwd");
```

For more information about the data source properties, see Setting the Data Source Properties.

See Also

Overview of the JDBC Driver

Deploying the JDBC Driver

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When you deploy an application that depends on the Microsoft JDBC Driver for SQL Server, you must redistribute the JDBC driver together with your application. Unlike Windows Data Access Components (Windows DAC), which is a component of the Windows operating system, the JDBC driver is considered to be a component of SQL Server.

There are two approaches to deploying the JDBC driver with your application. One is to include the JDBC driver files as part of your own custom installation package. The second approach involves using the JDBC installation package provided by Microsoft, which you can download from the Microsoft JDBC Driver for SQL Server Developer Center.

The following sections discuss how to use the JDBC installation package on Windows and UNIX operating systems.

> **NOTE**
> For information about deploying Java applications in general, see the Java website.

Deploying the JDBC Driver on Windows Systems

When you deploy the JDBC driver on Windows operating systems, you must use the executable zip file version of the installation package, which is typically named `sqljdbc_<version>_<language>.exe`.

To run the executable zip file silently, you must use the `/auto` command-line option on the command line or in a batch file as in the following:

```
sqljdbc_<version>_<language>.exe /auto
```

> **NOTE**
> When you use the `/auto` option it is not a truly silent installation, as a WinZip dialog box still appears on the user's screen. However, you will not need to interact with it and it closes as soon as the unzip operation is complete.

Deploying the Driver on UNIX Systems

When you deploy the JDBC driver on UNIX operating systems, you must use the gzip file version of the installation package, which is typically named `sqljdbc_<version>_<language>.tar.gz`.

Before you install the JDBC driver, make sure that both the gzip and tar utilities are installed on the user's system, and that the folders that contain the executables for both utilities are added to the PATH environment variable.

To unpack the zipped tar file, navigate to the directory where you want the driver unpacked and type the following command:

```
gzip -d sqljdbc_<version>_<language>.tar.gz
```

To unpack the tar file, move it to the directory where you want the driver installed and type the following command:

```
tar -xf sqljdbc_<version>_<language>.tar
```

See Also

Overview of the JDBC Driver

Redistributing the Microsoft JDBC Driver

8/2/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Redistribute the Microsoft JDBC 4.1, 4.2, 6.0, 6.2, 6.4, and 7.0 Driver

The JDBC Drivers 4.1, 4.2, 6.0, 6.2, 6.4, and 7.0 can be redistributed. Please review the "Distributable Code" clause in the license agreements for the 4.1, 4.2, 6.0, 6.2, 6.4, and 7.0 versions of the driver.

Register to Redistribute the Microsoft JDBC 4.0 Driver

The Microsoft JDBC 4.0 Driver requires registration before you redistribute it. Click the following link to review the license agreement for the 4.0 version of the driver. You may print and retain a copy of the license agreement for your records if you wish.

If you choose to accept the license agreement, you will be directed to the registration page and then the download page.

Redistribution License for Microsoft JDBC Driver 4.0 for SQL Server

Note: Microsoft uses Microsoft account for secure authentication and registration. When you register for redistribution rights for SQL Server JDBC Driver Redistribution through Microsoft account, you will be prompted to provide information that becomes part of your secure profile.

If you do not have one already, you can get a Microsoft account

Finding Additional JDBC Driver Information

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

For more information about the Microsoft JDBC Driver for SQL Server and SQL Server development in general, see the following online resources:

Remarks

RESOURCE	DESCRIPTION
JDBC Driver for SQL Server GitHub Repository	This repository contains the source code for the JDBC Driver for SQL Server. Use this site to file issues and contribute directly to the driver.
Data Access and Storage Developer Center	This site provides documentation, technical articles, sample code, and other resources for all data access technologies at Microsoft.
SQL Server Data Access Forum	This site serves as a community forum for data access to SQL Server by using SQL Server Native Client, OLE DB, ODBC, ADO, MDAC, JDBC, or SOAP/HTTP.
JDBC Blog	This blog is used to provide information about the Microsoft JDBC Driver for SQL Server and lets you interact directly with members of the JDBC driver product team.
Microsoft Connect	This site allows you to search for driver issues submitted by customers, and it allows you to submit your own.

See Also

Overview of the JDBC Driver

Microsoft JDBC Driver for SQL Server Support Matrix

8/8/2018 • 4 minutes to read • Edit Online

Download JDBC Driver

This page contains the support matrix and support lifecycle policy for the Microsoft JDBC Driver for SQL Server.

Microsoft JDBC Driver Support Lifecycle Matrix and Policy

The Microsoft Support Lifecycle (MSL) policy provides transparent, predictable information regarding the support lifecycle of Microsoft products. JDBC driver versions 3.0, 4.x, 6.x, and 7.x have five year Mainstream support from the driver release date. Mainstream support is defined on the Microsoft support lifecycle website.

Extended and custom support options are not available for the Microsoft JDBC Driver.

The following Microsoft JDBC Drivers are supported, until the indicated End of Support date.

DRIVER NAME	DRIVER PACKAGE VERSION	APPLICABLE JAR(S)	END OF MAINSTREAM SUPPORT
Microsoft JDBC Driver 7.0 for SQL Server	7.0	mssql-jdbc-7.0.0.jre10.jar mssql-jdbc-7.0.0.jre8.jar	July 31, 2023
Microsoft JDBC Driver 6.4 for SQL Server	6.4	mssql-jdbc-6.4.0.jre9.jar mssql-jdbc-6.4.0.jre8.jar mssql-jdbc-6.4.0.jre7.jar	February 27, 2023
Microsoft JDBC Driver 6.2 for SQL Server	6.2	mssql-jdbc-6.2.2.jre8.jar mssql-jdbc-6.2.2.jre7.jar	June 30, 2022
Microsoft JDBC Driver 6.0 for SQL Server	6.0	sqljdbc42.jar sqljdbc41.jar	July 14, 2021
Microsoft JDBC Driver 4.2 for SQL Server	4.2	sqljdbc42.jar sqljdbc41.jar	August 24, 2020
Microsoft JDBC Driver 4.1 for SQL Server	4.1	sqljdbc41.jar	December 12, 2019

The following Microsoft JDBC Drivers are no longer supported.

DRIVER NAME	DRIVER PACKAGE VERSION	END OF MAINSTREAM SUPPORT
Microsoft JDBC Driver 4.0 for SQL Server	4.0	March 6, 2017
Microsoft SQL Server JDBC Driver 3.0	3.0	April 23, 2015
Microsoft SQL Server JDBC Driver 2.0	2.0	December 31, 2012

DRIVER NAME	DRIVER PACKAGE VERSION	END OF MAINSTREAM SUPPORT
Microsoft SQL Server 2005 JDBC Driver 1.2	1.2	June 25, 2011
Microsoft SQL Server 2005 JDBC Driver 1.1	1.1	June 25, 2011
Microsoft SQL Server 2005 JDBC Driver 1.0	1.0	June 25, 2011
Microsoft SQL Server 2000 JDBC Driver	2000	July 9, 2010

SQL Version Compatibility

DRIVER VERSION	SQL SERVER 2008	SQL SERVER 2008R2	SQL SERVER 2012	AZURE SQL DATABASE	PDW 2008R2 AU3[4]	SQL SERVER 2014	SQL SERVER 2016	SQL SERVER 2017	AZURE SQL MANAGED INSTANCE (EXTENDED PRIVATE PREVIEW)
7.0	N	Y	Y	Y	Y	Y	Y	Y	Y
6.4	N	Y	Y	Y	Y	Y	Y	Y	Y
6.2	Y	Y	Y	Y	Y	Y	Y	Y	N
6.1	Y	Y	Y	Y	Y	Y	Y	N	N
6.0	Y	Y	Y	Y	Y	Y	Y	N	N
4.2	Y	Y	Y	Y	Y	Y	Y	N	N
4.1	Y	Y	Y	Y	Y	Y	Y	N	N
4.0	Y	Y	Y	Y	Y	Y	Y	N	N
3.0	Y	Y	Y[1]	Y[2]	N	Y[5]	N	N	N
2.0	Y[3]	Y[3]	N	N	N	N	N	N	N
1.2	Y[3]	N	N	N	N	N	N	N	N
1.1	N	N	N	N	N	N	N	N	N
1.0	N	N	N	N	N	N	N	N	N
2000	N	N	N	N	N	N	N	N	N

[1]Microsoft SQL Server JDBC Driver version 3.0 can connect to SQL Server 2012 as a down-level client.

[2]Support for Azure SQL Database was introduced in the 3.0 driver as a hotfix. We recommend that Azure SQL Database customers use the latest driver version available.

[3]Microsoft SQL Server JDBC Driver version 2.0 and Microsoft SQL Server 2005 JDBC Driver version 1.2 can connect to SQL Server 2008 as a down-level client. When down-level conversions are allowed, applications can execute queries and perform updates on the new SQL Server 2008 data types, such as time, date, datetime2, datetimeoffset, and FILESTREAM. For more information about how to use these new data types with the JDBC driver, see Working with SQL Server 2008 Date/Time Data Types using JDBC Driver and Working with SQL Server 2008 FileStream using JDBC Driver. For more information about the down-level compatibility of these new data types, see Using Date and Time Dataand FILESTREAM Support topics in SQL Server Books Online.

[4]Support for connections between the Microsoft JDBC Driver and Parallel Data Warehouse was first introduced in the Microsoft JDBC Driver 4.0 for SQL Server and Microsoft SQL Server 2008 R2 Parallel Data Warehouse Appliance Update 3.

[5]Microsoft SQL Server JDBC Driver version 3.0 can connect to SQL Server 2014 as a down-level client.

Java and JDBC Specification Support

JDBC DRIVER VERSION	JRE VERSIONS	JDBC API VERSION
7.0	1.8, 10	4.2, 4.3 (partially)
6.4	1.7, 1.8, 9	4.1, 4.2, 4.3 (partially)
6.2	1.7, 1.8	4.1, 4.2
6.1	1.7, 1.8	4.1, 4.2
6.0	1.7, 1.8	4.1, 4.2
4.2	1.7, 1.8	4.1, 4.2
4.1	1.7	4.0
4.0	1.5, 1.6, 1.7	3.0, 4.0
3.0	1.5, 1.6,	3.0, 4.0
2.0	1.5, 1.6	3.0, 4.0
1.2	1.4, 1.5, 1.6	3.0
1.1	1.4	3.0
1.0	1.4	3.0
2000	1.4	3.0

Supported Operating Systems

The Microsoft JDBC driver is designed to work on any operating system that supports the use of a Java Virtual Machine (JVM). Some commonly used platforms include Windows 10, Windows 8.1, Windows 8, Windows 7, Windows Server 2008 R2, Windows Vista, Linux, Unix, AIX, MacOS, and others.

The JDBC product team tests our driver on Windows, Sun Solaris, SUSE Linux, and RedHat Linux. Customer Support is available to customers on all platforms, however we may ask you to reproduce the issue on a platform such as Windows.

Application Server Support

The Microsoft JDBC Driver for SQL Server is tested with various application servers. Consult your application server vendor for additional details on which driver version is compatible with their product.

Frequently Asked Questions (FAQ) for JDBC Driver

8/8/2018 • 6 minutes to read • Edit Online

Download JDBC Driver

This page provides answers to frequently asked questions about the Microsoft JDBC Driver for SQL Server.

Frequently Asked Questions

How can I help improve the JDBC Driver?
The JDBC Driver is open-source and the source code can be found on GitHub. You can help improve the driver by filing issues and contributing to the code base.

Which versions of SQL Server and Java do the driver support?
See the Microsoft JDBC Driver for SQL Server Support Matrix page for details.

What is the difference between the JDBC driver packages available on the Microsoft Download Center and the JDBC driver available on GitHub?
The JDBC driver files available on the GitHub repository for the Microsoft JDBC driver are the core of the JDBC driver and are under the open-source license listed in the repository. The driver packages on the Microsoft Download Center include additional libraries for Windows-integrated authentication and enabling XA transactions with the JDBC driver. Those additional libraries are under the license included with the downloadable package.

What should I know when upgrading my driver?
The Microsoft JDBC Driver 7.0 supports the JDBC 4.2, and 4.3 (partially) specifications and includes two JAR class libraries in the installation package as follows:

JAR	JDBC SPECIFICATION	JDK VERSION
mssql-jdbc-7.0.0.jre10.jar	JDBC 4.3 (partially), and 4.2	JDK 10.0
mssql-jdbc-7.0.0.jre8.jar	JDBC 4.2	JDK 8.0

The Microsoft JDBC Driver 6.4 supports the JDBC 4.1, 4.2, and 4.3 (partially) specifications and includes three JAR class libraries in the installation package as follows:

JAR	JDBC SPECIFICATION	JDK VERSION
mssql-jdbc-6.4.0.jre9.jar	JDBC 4.3 (partially), 4.2, and 4.1	JDK 9.0
mssql-jdbc-6.4.0.jre8.jar	JDBC 4.2, and 4.1	JDK 8.0
mssql-jdbc-6.4.0.jre7.jar	JDBC 4.1	JDK 7.0

The Microsoft JDBC Driver 6.2 supports the JDBC 4.0, 4.1, and 4.2 specifications and includes two JAR class libraries in the installation package as follows:

JAR	JDBC SPECIFICATION	JDK VERSION
mssql-jdbc-6.2.2.jre8.jar	JDBC 4.2, 4.1, and 4.0	JDK 8.0

JAR	JDBC SPECIFICATION	JDK VERSION
mssql-jdbc-6.2.2.jre7.jar	JDBC 4.1 and 4.0	JDK 7.0

The Microsoft JDBC Drivers 6.0 and 4.2 for SQL Server supports JDBC 4.0, 4.1, and 4.2 specifications and include two JAR class libraries in the installation package as follows:

JAR	JDBC SPECIFICATION	JDK VERSION
sqljdbc42.jar	JDBC 4.2, 4.1, and 4.0	JDK 8.0
sqljdbc41.jar	JDBC 4.1 and 4.0	JDK 7.0

The Microsoft JDBC Driver 4.1 for SQL Server supports the JDBC 4.0 specification and includes one JAR class library in the installation package as follows:

JAR	JDBC SPECIFICATION	JDK VERSION
sqljdbc41.jar	JDBC 4.0	JDK 7.0 and 6.0

Do I need to make any code changes in my application to use the latest driver with my existing SQL Server version?
In general, the driver is designed to be backward compatible so that you do not need to change your existing applications when upgrading the driver. In the event that a new driver version introduces a breaking change, the Release Notes for the JDBC Driver section provides clear details on the change and the impact to existing applications. In addition, you can review the release notes included with the driver for a list of bugs fixed in that release and known issues.

How much does the driver cost?
The Microsoft JDBC Driver for SQL Server is available at no additional charge.

Can I redistribute the driver? The JDBC Drivers 4.1, 4.2, 6.0, 6.2, 6.4, and 7.0 are redistributable. Review the “Distributable Code” clause in the license agreements.

Can I use the driver to access Microsoft SQL Server from a Linux computer? Yes! You can use the driver to access SQL Server from Linux, Unix, and other non-Windows platforms. For more information, see Microsoft JDBC Driver for SQL Server Support Matrix.

Does the driver support Secure Sockets Layer (SSL) encryption? Starting with version 1.2, the driver supports Secure Sockets Layer (SSL) encryption. For more information, see Using SSL Encryption.

Which authentication types are supported by the Microsoft JDBC Driver for SQL Server?
The table below lists available authentication options. A pure Java Kerberos authentication is available starting with the 4.0 release of the driver.

Platform	Authentication
Non-Windows	Pure Java Kerberos
Non-Windows	SQL Server
Non-Windows	Azure Active Directory Authentication

Windows	Pure Java Kerberos
Windows	SQL Server
Windows	Kerberos with NTLM backup
Windows	NTLM
Windows	Azure Active Directory Authentication

Does the driver support Internet Protocol version 6 (IPv6) addresses?
Yes. The driver supports the use of IPv6 addresses. Use the connection properties collection and the serverName connection string property. For more information, see Building the Connection URL.

What is adaptive buffering?
Adaptive buffering is introduced starting with Microsoft SQL Server 2005 JDBC Driver version 1.2. It is designed to retrieve any kind of large-value data without the overhead of server cursors. The adaptive buffering feature of the Microsoft SQL Server JDBC Driver provides a connection string property, responseBuffering, which can be set to "adaptive" or "full". In the version 1.2 release, the buffering mode is "full" by default and the application must set the adaptive buffering mode explicitly. Starting with the JDBC Driver version 2.0, the default behavior of the driver is "adaptive". Thus, your application does not have to request the adaptive behavior explicitly to get the adaptive buffering behavior. For more information, see Using Adaptive Buffering and the blog What is adaptiveresponse buffering and why should I use it?.

Does the driver support connection pooling?
The driver provides support for Java Platform, Enterprise Edition 5 (Java EE 5) connection pooling. The driver implements the JDBC 3.0 required interfaces to enable the driver to participate in any connection-pooling implementation that is provided by middleware application server vendors. The driver participates in pooled connections in these environments. For more information, see Using Connection Pooling. The driver does not provide its own pooling implementation, but rather it relies on third-party Java application servers.

Is support available for the driver?
Several support options are available. You may post your question or issue to our GitHub repository which is monitored by Microsoft. Forums are monitored by Microsoft, MVPs, and the community. You may also contact Microsoft Customer Support. The development team may ask you to reproduce the issue outside any third-party application servers. If the issue cannot be reproduced outside the hosting Java container environment, you will need to involve the related third-party so that the team can continue to assist you. The team may also ask you to reproduce your issue on an operating system such as Windows so the problem can be best supported.

Is the driver certified for use with any third-party application servers? The driver has been tested against various application servers including IBM WebSphere and SAP Netweaver.

How do I enable tracing?
The driver supports the use of tracing (or logging) to help resolve issues and problems with the JDBC Driver when it is used in your application. To enable the use of client-side JAR tracing, the JDBC Driver uses the logging APIs in java.util.logging. For more information, see Tracing Driver Operation. For server-side XA tracing, see Data Access Tracing in SQL Server.

Where can I download older versions of the driver such as the SQL Server 2000 JDBC driver, 2005 driver, 1.0, 1.1, or 1.2 driver?
These driver versions are not available for download as they are no longer supported. We are continually improving the Java connectivity support. As such, we highly recommend you work with the latest version of Microsoft JDBC driver.

I am using JRE 1.4. Which driver is compatible with JRE 1.4?
For customers who are using SAP products and require JRE 1.4 support, you may contact SAPService Marketplace to obtain the 1.2 Microsoft JDBC driver.

Can the driver communicate using FIPS validated algorithms?
The Microsoft JDBC Driver does not contain any cryptographic algorithms. If a customer leverages operating system, application, and JVM algorithms that are deemed acceptable by Federal Information Processing Standards (FIPS) and configures the driver to use those algorithms then the driver uses only the designated algorithms for communication.

See Also

Overview of the JDBC Driver

Feature Dependencies of Microsoft JDBC Driver for SQL Server

8/2/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

This page lists down libraries that the Microsoft JDBC Driver for SQL Server depends on. The project has the following dependencies.

Compile Time

- `azure-keyvault` : Azure Key Vault Provider for Always Encrypted Azure Key Vault feature (optional)
- `adal4j` : Azure ActiveDirectory Library for Java for Azure Active Directory Authentication feature and Azure Key Vault feature (optional)

Test Time

Specific projects that require either of the above two features need to explicitly declare the respective dependencies in their pom file:

For Example: When using *Azure Active Directory Authentication feature*, then you need to redeclare *adal4j* dependency in your project's pom file. See the following snippet:

```
<dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
    <version>7.0.0.jre10</version>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>adal4j</artifactId>
    <version>1.6.0</version>
</dependency>
```

For Example: When using *Azure Key Vault feature*, then you need to redeclare *azure-keyvault* dependency and *adal4j* dependency in your project's pom file. See the following snippet:

```xml
<dependency>
    <groupId>com.microsoft.sqlserver</groupId>
    <artifactId>mssql-jdbc</artifactId>
    <version>7.0.0.jre10</version>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>adal4j</artifactId>
    <version>1.6.0</version>
</dependency>

<dependency>
    <groupId>com.microsoft.azure</groupId>
    <artifactId>azure-keyvault</artifactId>
    <version>1.0.0</version>
</dependency>
```

Dependency Requirements for the JDBC Driver

Working with Azure Key Vault Provider:

- JDBC Driver version 7.0.0 - Dependency versions: Azure-Keyvault (version 1.0.0), Adal4j (version 1.6.0), and their dependencies (Sample Application)
- JDBC Driver version 6.4.0 - Dependency versions: Azure-Keyvault (version 1.0.0), Adal4j (version 1.4.0), and their dependencies (Sample Application)
- JDBC Driver version 6.2.2 - Dependency versions: Azure-Keyvault (version 1.0.0), Adal4j (version 1.4.0), and their dependencies (Sample Application)
- JDBC Driver version 6.0.0 - Dependency versions: Azure-Keyvault (version 0.9.7), Adal4j (version 1.3.0), and their dependencies (Sample application)

NOTE

With v6.2.2 and v6.4.0 driver versions, the azure-keyvault-java dependency had been updated to version 1.0.0. However, the new version was not compatible with the previous version (version 0.9.7) and therefore breaks the existing implementation in the driver. The new implementation in the driver required API changes, which in turn breaks client programs that use the Azure Key Vault Provider.

This problem has been resolved with latest driver version v7.0.0 as the removed constructor using authentication callback mechanism is added back to Azure Key Vault provider for backwards compatibility.

Working with Azure Active Directory Authentication:

- JDBC Driver version 7.0.0 - Dependency versions: Ada4j (version 1.6.0) and its dependencies
- JDBC Driver version 6.4.0 - Dependency versions: Adal4j (version 1.4.0) and its dependencies
- JDBC Driver version 6.2.2 - Dependency versions: Adal4j (version 1.4.0) and its dependencies
- JDBC Driver version 6.0.0 - Dependency versions: Adal4j (version 1.3.0), and its dependencies - In this version of the driver, you can connect using *ActiveDirectoryIntegrated* Authentication Mode only on a Windows operating system and using sqljdbc_auth.dll and Active Directory Authentication Library for SQL Server (ADALSQL.DLL).

From driver version 6.4.0 onwards, applications don't necessarily require using ADALSQL.DLL on Windows Operating Systems. For **Non-Windows operating systems**, the driver requires Kerberos ticket to work with ActiveDirectoryIntegrated Authentication. For more information about how to connect to Active Directory using Kerberos, see Set Kerberos ticket on Windows, Linux And Mac.

For **Windows operating systems**, driver looks for sqljdbc_auth.dll by default and doesn't require Kerberos ticket setup or Azure library dependencies. However, If sqljdbc_auth.dll isn't available, driver looks for Kerberos ticket for authenticating to Active Directory as on other Operating Systems.

A sample application using this feature can be found here.

See Also

JDBC Driver GitHub Repository
JDBC Driver API Reference

Securing JDBC Driver Applications

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Enhancing the security of a Microsoft JDBC Driver for SQL Server application involves more than avoiding common coding pitfalls. An application that accesses data has many potential points of failure that an attacker can exploit to retrieve, manipulate, or destroy sensitive data. It is important to understand all aspects of security, from the process of threat modeling during the design phase of your application to its eventual deployment, and continuing through its ongoing maintenance.

The topics in this section describe some common security concerns including connection strings, validating user input, and general application security.

In This Section

TOPIC	DESCRIPTION
Securing Connection Strings	Describes techniques to help protect information used to connect to a data source.
Validating User Input	Describes techniques to validate user input.
Application Security	Describes how to use Java policy permissions to help secure a JDBC driver application.
Using SSL Encryption	Describes how to establish a secure communication channel with a SQL Server database using Secure Sockets Layer (SSL).
FIPS Mode	Describes how to use JDBC driver in FIPS compliant mode.

See Also

Overview of the JDBC Driver

Securing Connection Strings

8/2/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Protecting access to your data source is one of the most important goals of helping to secure an application. To limit access to your data source, you must take precautions to help secure connection information such as a user ID, password, and data source name. Storing a user ID and password in plain text, such as in your source code, presents a serious security issue. Even if you supply a compiled version of code that contains user ID and password information in an external source, your compiled code can potentially be disassembled and the user ID and password exposed. As a result, it is imperative that critical information such as a user ID and password not exist in your code.

It is recommended that you ignore storing the password together with the connection URL in application source code. Instead, consider storing the password in a separate file that has restricted access. The access to that file can be granted to the context under which the application is running.

Another approach is to store the encrypted password in a file. Make sure that you use an encryption API that does not require the key to be stored somewhere and is not derived from the password of a user. For example, you might consider using certificate-based public/private key pairs, or use an approach where two parties use a key agreement protocol (Diffie-Hellman algorithm) to generate identical secret keys for encryption without ever having to transmit the secret key.

If you take connection string information from an external source, such as a user supplying a user ID and password, you must validate any input from the source to ensure that it follows the correct format and does not contain additional parameters that affect your connection.

See Also

Securing JDBC Driver Applications

Validating User Input

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When you construct an application that accesses data, you should assume all user input to be malicious until proven otherwise. Failure to do this can leave your application vulnerable to attack. One type of attack that can occur is called SQL injection, where malicious code is added to strings that are later passed to an instance of SQL Server to be parsed and run. To avoid this type of attack, you should use stored procedures with parameters where possible, and always validate user input.

Validating user input in client code is important so that you do not waste round trips to the server. It is equally important to validate parameters to stored procedures on the server to catch input that is not valid and that bypasses client-side validation.

For more information about SQL injection and how to avoid it, see "SQL Injection" in SQL Server Books Online. For more information about validating stored procedure parameters, see "Stored Procedures (Database Engine)" and subordinate topics in SQL Server Books Online.

See Also

Securing JDBC Driver Applications

Application Security

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When you use the Microsoft JDBC Driver for SQL Server, it is important to take precautions to ensure the security of your application. The following sections provide information regarding steps you can take to help secure your application.

Using Java Policy Permissions

When you use the Microsoft JDBC Driver for SQL Server, it is important to specify the required Java policy permissions that the JDBC driver requires. The Java Runtime Environment (JRE) provides an extensive security model that you can use at runtime to determine whether a thread has access to a resource. Security policy files can control this access. The policy files themselves are managed by the deployer and the sysadmin for the container, but the permissions listed in this topic are those that affect the functioning of the JDBC driver.

A typical permission in the policy file looks like the following.

```
// Example policy file entry.
grant [signedBy <signer>,] [codeBase <code source>] {
   permission  <class>  [<name> [, <action list>]];
};
```

The following codebase should be restricted to the JDBC driver codebase to ensure that you grant the least amount of privileges.

```
grant codeBase "file:/install_dir/lib/-" {

// Grant access to data source.
permission java.util.PropertyPermission "java.naming.*", "read,write";

// Specify which hosts can be connected to.
permission java.net.socketPermission "host:port", "connect";

// Logger permission to take advantage of logging.
permission java.util.logging.LoggingPermission;

// Grant listen/connect/accept permissions to the driver if
// connecting to a named instance as the client driver.
// This connects to a udp service and listens for a response.
permission java.net.SocketPermission "*", "listen, connect, accept";
};
```

NOTE

The code "file:/install_dir/lib/-" refers to the installation directory of the JDBC driver.

Protecting Server Communication

When you use the JDBC driver to communicate with a SQL Server database, you can secure the communication channel by using either Internet Protocol Security (IPSEC) or Secure Sockets Layer (SSL); or you can use both.

SSL support can be used to provide an additional level of protection besides IPSEC. For more information about using SSL, see Using SSL Encryption.

See Also

Securing JDBC Driver Applications

Using SSL Encryption

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Secure Sockets Layer (SSL) encryption enables transmitting encrypted data across the network between an instance of SQL Server and a client application.

Secure Sockets Layer (SSL) is a protocol for establishing a secure communication channel to prevent the interception of critical or sensitive information across the network and other Internet communications. SSL allows the client and the server to authenticate the identity of each other. After the participants are authenticated, SSL provides encrypted connections between them for secure message transmission.

The Microsoft JDBC Driver for SQL Server provides an infrastructure to enable and disable the encryption on a particular connection based on the user specified connection properties and the server and client settings. The user can specify the certificate store location and password, a host name to be used to validate the certificate, and when to encrypt the communication channel.

Enabling SSL encryption increases the security of data transmitted across networks between instances of SQL Server and applications. However, enabling encryption does slow performance.

The topics in this section describe how the Microsoft JDBC Driver for SQL Server version supports SSL encryption, including new connection properties, and how you can configure the trust store at the client-side.

NOTE

The **hostNameInCertificate** connection property is recommended to validate an SSL certificate.

In This Section

TOPIC	DESCRIPTION
Understanding SSL Support	Describes how the Microsoft JDBC Driver for SQL Server supports SSL encryption.
Connecting with SSL Encryption	Describes how to connect to a SQL Server database by using the new SSL specific connection properties.
Configuring the Client for SSL Encryption	Describes how to configure the default trust store at the client-side and how to import a private certificate to the client computer's trust store.

See Also

Securing JDBC Driver Applications

FIPS Mode

7/26/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server supports *FIPS 140 Compliant Mode*. For Oracle / Sun JVM, refer to the FIPS 140 Compliant Mode for SunJSSE section provided by Oracle to configure FIPS enabled JVM.

Prerequisites:

- FIPS configured JVM
- Appropriate SSL Certificate.
- Appropriate policy files.
- Appropriate Configuration Parameters.

FIPS Configured JVM

To see the approved modules for FIPS Configuration, refer to the Validated FIPS 140-1 and FIPS 140-2 Cryptographic Modules.

Vendors may have some additional steps to configure JVM with FIPS.

Ensure your JVM is in FIPS Mode

To ensure your JVM is FIPS enabled, execute the following snippet:

```
public boolean isFIPS() throws Exception {
    Provider jsse = Security.getProvider("SunJSSE");
    return jsse != null && jsse.getInfo().contains("FIPS");
}
```

Appropriate SSL Certificate

In order to connect SQL Server in FIPS mode, a valid SSL Certificate is required. Install or import it in the Java Key Store on the client machine (JVM) where FIPS is enabled.

Importing SSL Certificate in Java KeyStore

For FIPS, most likely you need to import the certificate (.cert) to either PKCS or in a provider-specific format. Use the following snippet to import the SSL certificate and store it in a working directory with the appropriate KeyStore format. *TRUST_STORE_PASSWORD* is your password for Java KeyStore.

```
    public void saveGenericKeyStore(String provider, String trustStoreType, String certName, String certPath)
throws KeyStoreException, CertificateException, NoSuchAlgorithmException, NoSuchProviderException, IOException
{
        KeyStore ks = KeyStore.getInstance(trustStoreType, provider);
        FileOutputStream os = new FileOutputStream("./MyTrustStore_" + trustStoreType);
        ks.load(null, null);
        ks.setCertificateEntry(certName, getCertificate(certPath));
        ks.store(os, TRUST_STORE_PASSWORD.toCharArray());
        os.flush();
        os.close();
    }

    private Certificate getCertificate(String pathName) throws FileNotFoundException, CertificateException {
        FileInputStream fis = new FileInputStream(pathName);
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        return cf.generateCertificate(fis);
    }
```

The following example is importing an Azure SSL Certificate in PKCS12 format with BouncyCastle Provider. The certificate is imported in the working directory named *MyTrustStore_PKCS12* by using the following snippet:

```
saveGenericKeyStore(BCFIPS, PKCS12, "SQLAzure SSL Certificate Name", "SQLAzure.cer");
```

Appropriate policy files

For some FIPS Providers, unrestricted Policy jars are needed. In such cases, for Sun / Oracle, download the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files for JRE 8 or JRE 7.

Appropriate Configuration Parameters

To run the JDBC Driver in FIPS-compliant mode, configure connection properties as shown in following table.

Properties:

PROPERTY	TYPE	DEFAULT	DESCRIPTION	NOTES
encrypt	boolean ["true / false"]	"false"	For FIPS enabled JVM encrypt property should be **true**	
TrustServerCertificate	boolean ["true / false"]	"false"	For FIPS, the user needs to validate certificate chain, so the user should use **"false"** value for this property.	
trustStore	String	null	Your Java Keystore file path where you imported your certificate. If you install certificate on your system, then no need to pass anything. Driver uses cacerts or jssecacerts files.	

PROPERTY	TYPE	DEFAULT	DESCRIPTION	NOTES
trustStorePassword	String	null	The password used to check the integrity of the trustStore data.	
fips	boolean ["true / false"]	"false"	For FIPS enabled JVM this property should be **true**	Added in 6.1.4 (Stable release 6.2.2)
fipsProvider	String	null	FIPS provider configured in JVM. For example, BCFIPS or SunPKCS11-NSS	Added in 6.1.2 (Stable release 6.2.2), deprecated in 6.4.0 - see the details Here.
trustStoreType	String	JKS	For FIPS mode set trust store type either PKCS12 or type defined by FIPS provider	Added in 6.1.2 (Stable release 6.2.2)

Understanding SSL Support

8/13/2018 • 7 minutes to read • Edit Online

Download JDBC Driver

When connecting to SQL Server, if the application requests encryption and the instance of SQL Server is configured to support SSL encryption, the Microsoft JDBC Driver for SQL Server initiates the SSL handshake. The handshake allows the server and client to negotiate the encryption and cryptographic algorithms to be used to protect data. After the SSL handshake is complete, the client and server can send the encrypted data securely. During SSL handshake, the server sends its public key certificate to the client. The issuer of a public key certificate is known as a Certificate Authority (CA). The client is responsible for validating that the certificate authority is one that the client trusts.

If the application does not request encryption, the Microsoft JDBC Driver for SQL Server will not force SQL Server to support SSL encryption. If the SQL Server instance is not configured to force the SSL encryption, a connection is established without encryption. If the SQL Server instance is configured to force the SSL encryption, the driver will automatically enable SSL encryption when running on properly configured Java Virtual Machine (JVM), or else the connection is terminated and the driver will raise an error.

NOTE

Make sure the value passed to **serverName** exactly matches the Common Name (CN) or DNS name in the Subject Alternate Name (SAN) in the server certificate for an SSL connection to succeed.

For more information about how to configure SSL for SQL Server, see the Encrypting Connections to SQL Server topic in SQL Server Books Online.

Remarks

In order to allow applications to use SSL encryption, the Microsoft JDBC Driver for SQL Server has introduced the following connection properties starting with the version 1.2 release: **encrypt**, **trustServerCertificate**, **trustStore**, **trustStorePassword**, and **hostNameInCertificate**. For more information, see Setting the Connection Properties.

The following table summarizes how the Microsoft JDBC Driver for SQL Server version behaves for possible SSL connection scenarios. Each scenario uses a different set of SSL connection properties. The table includes:

- **blank**: "The property does not exist in the connection string"
- **value**: "The property exists in the connection string and its value is valid"
- **any**: "It does not matter whether the property exists in the connection string or its value is valid"

NOTE

The same behavior applies for SQL Server user authentication and Windows integrated authentication.

ENCRYPT	TRUSTSERVERCERTIFICATE	HOSTNAMEINCERTIFICATE	TRUSTSTORE	TRUSTSTOREPASSWORD	BEHAVIOR

ENCRYPT	TRUSTSERVERCERTIFICATE	HOSTNAMEINCERTIFICATE	TRUSTSTORE	TRUSTSTOREPASSWORD	BEHAVIOR
false or blank	any	any	any	any	The Microsoft JDBC Driver for SQL Server will not force SQL Server to support SSL encryption. If the server has a self-signed certificate, the driver initiates the SSL certificate exchange. The SSL certificate will not be validated and only the credentials (in the login packet) are encrypted. If the server requires the client to support SSL encryption, the driver will initiate the SSL certificate exchange. The SSL certificate will not be validated, but the entire communication will be encrypted.

ENCRYPT	TRUSTSERVERCERTIFICATE	HOSTNAMEINCERTIFICATE	TRUSTSTORE	TRUSTSTOREPASSWORD	BEHAVIOR
true	true	any	any	any	The Microsoft JDBC Driver for SQL Server requests to use SSL encryption with the SQL Server. If the server requires the client to support SSL encryption or if the server supports encryption, the driver will initiate the SSL certificate exchange. Note that if the **trustServerCertificate** property is set to "true", the driver will not validate the SSL certificate. If the server is not configured to support encryption, the driver will raise an error and terminate the connection.

ENCRYPT	TRUSTSERVERCERTIFICATE	HOSTNAMEINCERTIFICATE	TRUSTSTORE	TRUSTSTOREPASSWORD	BEHAVIOR
true	false or blank	blank	blank	blank	The Microsoft JDBC Driver for SQL Server requests to use SSL encryption with the SQL Server. If the server requires the client to support SSL encryption or if the server supports encryption, the driver will initiate the SSL certificate exchange. The driver will use the **serverName** property specified on the connection URL to validate the server SSL certificate and rely on the trust manager factory's look-up rules to determine which certificate store to use. If the server is not configured to support encryption, the driver will raise an error and terminate the connection.

ENCRYPT	TRUSTSERVERCERTIFICATE	HOSTNAMEINCERTIFICATE	TRUSTSTORE	TRUSTSTOREPASSWORD	BEHAVIOR
true	false or blank	value	blank	blank	The Microsoft JDBC Driver for SQL Server requests to use SSL encryption with the SQL Server. If the server requires the client to support SSL encryption or if the server supports encryption, the driver will initiate the SSL certificate exchange. The driver will validate the SSL certificate's subject value by using the value specified for the **hostNameInCertificate** property. If the server is not configured to support encryption, the driver will raise an error and terminate the connection.

ENCRYPT	TRUSTSERVERCERTIFICATE	HOSTNAMEINCERTIFICATE	TRUSTSTORE	TRUSTSTOREPASSWORD	BEHAVIOR
true	false or blank	blank	value	value	The Microsoft JDBC Driver for SQL Server requests to use SSL encryption with the SQL Server. If the server requires the client to support SSL encryption or if the server supports encryption, the driver will initiate the SSL certificate exchange. The driver will use the **trustStore** property value to find the certificate trustStore file and **trustStorePassword** property value to check the integrity of the trustStore file. If the server is not configured to support encryption, the driver will raise an error and terminate the connection.

ENCRYPT	TRUSTSERVERCERTIFICATE	HOSTNAMEINCERTIFICATE	TRUSTSTORE	TRUSTSTOREPASSWORD	BEHAVIOR
true	false or blank	blank	blank	value	The Microsoft JDBC Driver for SQL Server requests to use SSL encryption with the SQL Server. If the server requires the client to support SSL encryption or if the server supports encryption, the driver will initiate the SSL certificate exchange. The driver will use the **trustStorePassword** property value to check the integrity of the default trustStore file. If the server is not configured to support encryption, the driver will raise an error and terminate the connection.

ENCRYPT	TRUSTSERVERCERTIFICATE	HOSTNAMEINCERTIFICATE	TRUSTSTORE	TRUSTSTOREPASSWORD	BEHAVIOR
true	false or blank	blank	value	blank	The Microsoft JDBC Driver for SQL Server requests to use SSL encryption with the SQL Server. If the server requires the client to support SSL encryption or if the server supports encryption, the driver will initiate the SSL certificate exchange. The driver will use the **trustStore** property value to look up the location of the trustStore file. If the server is not configured to support encryption, the driver will raise an error and terminate the connection.

ENCRYPT	TRUSTSERVERCERTIFICATE	HOSTNAMEINCERTIFICATE	TRUSTSTORE	TRUSTSTOREPASSWORD	BEHAVIOR
true	false or blank	value	blank	value	The Microsoft JDBC Driver for SQL Server requests to use SSL encryption with the SQL Server. If the server requires the client to support SSL encryption or if the server supports encryption, the driver will initiate the SSL certificate exchange. The driver will use the **trustStorePassword** property value to check the integrity of the default trustStore file. In addition, the driver will use the **hostNameInCertificate** property value to validate the SSL certificate. If the server is not configured to support encryption, the driver will raise an error and terminate the connection.

ENCRYPT	TRUSTSERVERCERTIFICATE	HOSTNAMEINCERTIFICATE	TRUSTSTORE	TRUSTSTOREPASSWORD	BEHAVIOR
true	false or blank	value	value	blank	The Microsoft JDBC Driver for SQL Server requests to use SSL encryption with the SQL Server. If the server requires the client to support SSL encryption or if the server supports encryption, the driver will initiate the SSL certificate exchange. The driver will use the **trustStore** property value to look up the location of the trustStore file. In addition, the driver will use the **hostNameInCertificate** property value to validate the SSL certificate. If the server is not configured to support encryption, the driver will raise an error and terminate the connection.

ENCRYPT	TRUSTSERVERCERTIFICATE	HOSTNAMEINCERTIFICATE	TRUSTSTORE	TRUSTSTOREPASSWORD	BEHAVIOR
true	false or blank	value	value	value	The Microsoft JDBC Driver for SQL Server requests to use SSL encryption with the SQL Server. If the server requires the client to support SSL encryption or if the server supports encryption, the driver will initiate the SSL certificate exchange. The driver will use the **trustStore** property value to find the certificate trustStore file and **trustStorePassword** property value to check the integrity of the trustStore file. In addition, the driver will use the **hostNameInCertificate** property value to validate the SSL certificate. If the server is not configured to support encryption, the driver will raise an error and terminate the connection.

If the encrypt property is set to **true**, the Microsoft JDBC Driver for SQL Server uses the JVM's default JSSE security provider to negotiate SSL encryption with SQL Server. The default security provider may not support all of the features required to negotiate SSL encryption successfully. For example, the default security provider may not support the size of the RSA public key used in the SQL Server SSL certificate. In this case, the default security provider might raise an error that will cause the JDBC driver to terminate the connection. In order to resolve this issue, do one of the following:

- Configure the SQL Server with a server certificate that has a smaller RSA public key
- Configure the JVM to use a different JSSE security provider in the "<java-home>/lib/security/java.security" security properties file

- Use a different JVM

Validating Server SSL Certificate

During SSL handshake, the server sends its public key certificate to the client. The JDBC driver or client has to validate that the server certificate is issued by a certificate authority that the client trusts. The driver requires that the server certificate must meet the following conditions:

- The certificate was issued by a trusted certificate authority.
- The certificate must be issued for server authentication.
- The certificate is not expired.
- The Common Name (CN) in the Subject or a DNS name in the Subject Alternate Name (SAN) of the certificate exactly matches the **serverName** value specified in the connection string or, if specified, the **hostNameInCertificate** property value.
- A DNS name can include wild card characters. But the Microsoft JDBC Driver for SQL Server does not support wild card matching. That is, abc.com will not match *.com but *.com will match *.com.

See Also

Using SSL Encryption

Securing JDBC Driver Applications

Connecting with SSL Encryption

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The examples in this article describe how to use connection string properties that allow applications to use Secure Sockets Layer (SSL) encryption in a Java application. For more information about these new connection string properties such as **encrypt**, **trustServerCertificate**, **trustStore**, **trustStorePassword**, and **hostNameInCertificate**, see Setting the Connection Properties.

When the **encrypt** property is set to **true** and the **trustServerCertificate** property is set to **true**, the Microsoft JDBC Driver for SQL Server won't validate the SQL Server SSL certificate. This is usually required for allowing connections in test environments, such as where the SQL Server instance has only a self signed certificate.

The following code example demonstrates how to set the **trustServerCertificate** property in a connection string:

```
String connectionUrl =
    "jdbc:sqlserver://localhost:1433;" +
     "databaseName=AdventureWorks;integratedSecurity=true;" +
     "encrypt=true;trustServerCertificate=true";
```

When the **encrypt** property is set to **true** and the **trustServerCertificate** property is set to **false**, the Microsoft JDBC Driver for SQL Server will validate the SQL Server SSL certificate. Validating the server certificate is a part of the SSL handshake and ensures that the server is the correct server to connect to. To validate the server certificate, the trust material must be supplied at connection time either by using **trustStore** and **trustStorePassword** connection properties explicitly, or by using the underlying Java Virtual Machine (JVM)'s default trust store implicitly.

The **trustStore** property specifies the path (including filename) to the certificate trustStore file, which contains the list of certificates that the client trusts. The **trustStorePassword** property specifies the password used to check the integrity of the trustStore data. For more information on using the JVM's default trust store, see the Configuring the Client for SSL Encryption.

The following code example demonstrates how to set the **trustStore** and **trustStorePassword** properties in a connection string:

```
String connectionUrl =
    "jdbc:sqlserver://localhost:1433;" +
     "databaseName=AdventureWorks;integratedSecurity=true;" +
     "encrypt=true; trustServerCertificate=false;" +
     "trustStore=storeName;trustStorePassword=storePassword";
```

The JDBC Driver provides an additional property, **hostNameInCertificate**, which specifies the host name of the server. The value of this property must match the subject property of the certificate.

The following code example demonstrates how to use the **hostNameInCertificate** property in a connection string:

```
String connectionUrl =
    "jdbc:sqlserver://localhost:1433;" +
     "databaseName=AdventureWorks;integratedSecurity=true;" +
     "encrypt=true; trustServerCertificate=false;" +
     "trustStore=storeName;trustStorePassword=storePassword" +
     "hostNameInCertificate=hostName";
```

NOTE

Alternatively, you can set the value of connection properties by using the appropriate **setter** methods provided by the SQLServerDataSource class.

If the **encrypt** property is set to **true** and the **trustServerCertificate** property is set to **false** and if the server name in the connection string doesn't match the server name in the SQL Server SSL certificate, the following error will be issued: The driver couldn't establish a secure connection to SQL Server by using Secure Sockets Layer (SSL) encryption. Error: "java.security.cert.CertificateException: Failed to validate the server name in a certificate during Secure Sockets Layer (SSL) initialization."

See Also

Using SSL Encryption
Securing JDBC Driver Applications

Configuring the Client for SSL Encryption

5/3/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server or client has to validate that the server is the correct server and its certificate is issued by a certificate authority that the client trusts. In order to validate the server certificate, the trust material must be supplied at connection time. In addition, the issuer of the server certificate must be a certificate authority that the client trusts.

This topic first describes how to supply the trust material in the client computer. Then, the topic describes how to import a server certificate to the client computer's trust store when the instance of SQL Server's Secure Sockets Layer (SSL) certificate is issued by a private certificate authority.

For more information about validating the server certificate, see the Validating Server SSL Certificate section in Understanding SSL Support.

Configuring the Client Trust Store

Validating the server certificate requires that the trust material must be supplied at connection time either by using **trustStore** and **trustStorePassword** connection properties explicitly, or by using the underlying Java Virtual Machine (JVM)'s default trust store implicitly. For more information about how to set the **trustStore** and **trustStorePassword** properties in a connection string, see Connecting with SSL Encryption.

If the **trustStore** property is unspecified or set to null, the Microsoft JDBC Driver for SQL Server will rely on the underlying JVM's security provider, the Java Secure Socket Extension (SunJSSE). The SunJSSE provider provides a default TrustManager, which is used to validate X.509 certificates returned by SQL Server against the trust material provided in a trust store.

The TrustManager tries to locate the default trustStore in the following search order:

- If the system property "javax.net.ssl.trustStore" is defined, the TrustManager tries to find the default trustStore file by using the filename specified by that system property.
- If the "javax.net.ssl.trustStore" system property was not specified, and if the file "<java-home>/lib/security/jssecacerts" exists, that file is used.
- If the file "<java-home>/lib/security/cacerts" exists, that file is used.

  For more information, see the SUNX509 TrustManager interface documentation on the Sun Microsystems Web site.

  The Java Runtime Environment allows you to set the trustStore and trustStorePassword system properties as follows:

```
java -Djavax.net.ssl.trustStore=C:\MyCertificates\storeName
java -Djavax.net.ssl.trustStorePassword=storePassword
```

In this case, any application running on this JVM will use these settings as default. In order to override the default settings in your application, you should set the **trustStore** and **trustStorePassword** connection properties either in the connection string or in the appropriate setter method of the SQLServerDataSource class.

In addition, you can configure and manage the default trust store files such as "<java-

home>/lib/security/jssecacerts" and "<java-home>/lib/security/cacerts". To do that, use the JAVA "keytool" utility that is installed with the JRE (Java Runtime Environment). For more information about the "keytool" utility, see keytool documentation on the Sun Microsystems Web site.

Importing the Server Certificate to Trust Store

During the SSL handshake, the server sends its public key certificate to the client. The issuer of a public key certificate is known as a Certificate Authority (CA). The client has to ensure that the certificate authority is one that the client trusts. This is achieved by knowing the public key of trusted CAs in advance. Normally, the JVM ships with a predefined set of trusted certificate authorities.

If the instance of SQL Server's SSL certificate is issued by a private certificate authority, you must add the certificate authority's certificate to the list of trusted certificates in the client computer's trust store.

In order to do that, use the JAVA "keytool" utility that is installed with the JRE (Java Runtime Environment). The following command prompt demonstrates how to use the "keytool" utility to import a certificate from a file:

```
keytool -import -v -trustcacerts -alias myServer -file caCert.cer -keystore truststore.ks
```

The example uses a file named "caCert.cer" as a certificate file. You must obtain this certificate file from the server. The following steps explain how to export the server certificate to a file:

1. Click Start and then Run, and type MMC. (MMC is an acronym for the Microsoft Management Console.)
2. In MMC, open the Certificates.
3. Expand Personal and then Certificates.
4. Right-click the server certificate, and then select All Tasks\Export.
5. Click Next to move past the welcome dialog box of the Certificate Export Wizard.
6. Confirm that "No, do not export the private key" is selected, and then click Next.
7. Make sure that either DER encoded binary X.509 (.CER) or Base-64 encoded X.509 (.CER) is selected, and then click Next.
8. Enter an export file name.
9. Click Next, and then click Finish to export the certificate.

See Also

Using SSL Encryption
Securing JDBC Driver Applications

Improving Performance and Reliability with the JDBC Driver

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

One aspect of application development that is common to all applications is the constant need to improve performance and reliability. There are a number of techniques for doing this with the Microsoft JDBC Driver for SQL Server.

The topics in this section describe various techniques for improving application performance and reliability when using the JDBC driver.

In This Section

TOPIC	DESCRIPTION
Closing Objects when Not In Use	Describes the importance of closing JDBC driver objects when they are no longer needed.
Managing Transaction Size	Describes techniques for improving transaction performance.
Working with Statements and Result Sets	Describes techniques for improving performance when using the Statement or ResultSet objects.
Using Adaptive Buffering	Describes an adaptive buffering feature, which is designed to retrieve any kind of large-value data without the overhead of server cursors.
Sparse Columns	Discusses the JDBC driver's support for SQL Server sparse columns.
Prepared Statement Metadata Caching for the JDBC Driver	Discusses the techniques for improving performance with prepared statement queries.
Using Bulk Copy API for Batch Insert Operation	Describes how to enable Bulk Copy API for batch insert operations and its benefits.

See also

Overview of the JDBC Driver

Closing Objects when Not In Use

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When you work with objects of Microsoft JDBC Driver for SQL Server, particularly the SQLServerResultSet or one of the Statement objects such as SQLServerStatement, SQLServerPreparedStatement or SQLServerCallableStatement, you should explicitly close them by using their close methods when they are no longer needed. This improves performance by freeing up driver and server resources as soon as possible, instead of waiting for the Java Virtual Machine garbage collector to do it for you.

Closing objects is particularly crucial to maintaining good concurrency on the server when you are using scroll locks. Scroll locks in the last accessed fetch buffer are held until the result set is closed. Similarly, statement prepared handles are held until the statement is closed. If you are reusing a connection for multiple statements, closing the statements before you let them go out of scope will allow the server to clean up the prepared handles earlier.

See Also

Improving Performance and Reliability with the JDBC Driver

Managing Transaction Size

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When you work with transactions, it is important to keep your transactions as brief as possible. The default mode of auto-commit, which you can enable or disable by using the setAutoCommit method, will commit every action for you. This is the easiest mode to work with for most developers.

When you use manual transactions, make sure that your code commits the transaction as quickly as possible. Holding open a transaction blocks other users from accessing the data. For example, a good programming practice might be to put a rollback call in your catch block and a commit call in the finally block. However, this depends on the design of your application.

Keeping the size of your transactions small creates better concurrency. For example, if you start a manual transaction and modify 10,000 rows in a 20,000-row table, you will have half the table completely blocked from all other users, even if they are only reading the data. Reducing your modifications to 2,000 rows leaves 90 percent of the table available.

Additionally, be sure to use the lock time out setting if your application expects some blocking issues and needs to time out from these. You can do this by using the setLockTimeout method. The default for the lock time out is -1, which means that it will block indefinitely while waiting for the lock. You can set the lock time out to 30 seconds, which will cause the blocked connection to time out in 30 seconds if blocked by another connection.

See Also

Improving Performance and Reliability with the JDBC Driver

Working with Statements and Result Sets

8/2/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When you work with the Microsoft JDBC Driver for SQL Server and the Statement and ResultSet objects that it provides, there are several techniques that you can use to improve the performance and reliability of your applications.

Use the Appropriate Statement Object

When you use one of the JDBC driver Statement objects, such as the SQLServerStatement, SQLServerPreparedStatement, or the SQLServerCallableStatement object, make sure that you are using the appropriate object for the job.

- If you do not have OUT parameters, you do not need to use the SQLServerCallableStatement object. Instead, use the SQLServerStatement or the SQLServerPreparedStatement object.
- If you do not intend to execute the statement more than once, or do not have IN or OUT parameters, you do not need to use the SQLServerCallableStatement or the SQLServerPreparedStatement object. Instead, use the SQLServerStatement object.

Use the Appropriate Concurrency for ResultSet Objects

Do not ask for updatable concurrency when you create statements that produce result sets unless you actually intend to update the results. The default forward-only, read-only cursor model is fastest for reading small result sets.

Limit the Size of Your Result Sets

Consider using the setMaxRows method (or SET ROWCOUNT or SELECT TOP N SQL syntax) to limit the number of rows returned from potentially large result sets. If you must deal with large result sets, consider using an adaptive response buffering by setting the connection string property responseBuffering=adaptive, which is the default mode. This approach allows the application to process large result sets without requiring the server-side cursors and minimizes the application memory usage. For more information, see Using Adaptive Buffering.

Use the Appropriate Fetch Size

For read-only server cursors, the tradeoff is round trips to the server versus the amount of memory used in the driver. For updatable server cursors, the fetch size also influences the sensitivity of the result set to changes and concurrency on the server. Updates to rows within the current fetch buffer are not visible until an explicit refreshRow method is issued or until the cursor leaves the fetch buffer. Large fetch buffers will have better performance (fewer server round trips) but are less sensitive to changes and reduce concurrency on the server if CONCUR_SS_SCROLL_LOCKS (1009) is used. For maximum sensitivity to changes, use a fetch size of 1. However, note that this will incur a round trip to the server for every row fetched.

Use Streams for Large IN Parameters

Use streams or BLOBs and CLOBs that are incrementally materialized to handle updating large column values or sending large IN parameters. The JDBC driver "chunks" these to the server in multiple round trips, allowing you to set and update values larger than what will fit in memory.

See Also

Improving Performance and Reliability with the JDBC Driver

Using Adaptive Buffering

8/13/2018 • 7 minutes to read • Edit Online

Download JDBC Driver

Adaptive buffering is designed to retrieve any kind of large-value data without the overhead of server cursors. Applications can use the adaptive buffering feature with all versions of SQL Server that are supported by the driver.

Normally, when the Microsoft JDBC Driver for SQL Server executes a query, the driver retrieves all of the results from the server into application memory. Although this approach minimizes resource consumption on the SQL Server, it can throw an OutOfMemoryError in the JDBC application for the queries that produce very large results.

In order to allow applications to handle very large results, the Microsoft JDBC Driver for SQL Server provides adaptive buffering. With adaptive buffering, the driver retrieves statement execution results from the SQL Server as the application needs them, rather than all at once. The driver also discards the results as soon as the application can no longer access them. The following are some examples where the adaptive buffering can be useful:

- **The query produces a very large result set:** The application can execute a SELECT statement that produces more rows than the application can store in memory. In previous releases, the application had to use a server cursor to avoid an OutOfMemoryError. Adaptive buffering provides the ability to do a forward-only read-only pass of an arbitrarily large result set without requiring a server cursor.
- **The query produces very large** SQLServerResultSet **columns or** SQLServerCallableStatement **OUT parameter values:** The application can retrieve a single value (column or OUT parameter) that is too large to fit entirely in application memory. Adaptive buffering allows the client application to retrieve such a value as a stream, by using the getAsciiStream, the getBinaryStream, or the getCharacterStream methods. The application retrieves the value from the SQL Server as it reads from the stream.

> **NOTE**
> With adaptive buffering, the JDBC driver buffers only the amount of data that it has to. The driver does not provide any public method to control or limit the size of the buffer.

Setting Adaptive Buffering

Starting with the JDBC driver version 2.0, the default behavior of the driver is "**adaptive**". In other words, in order to get the adaptive buffering behavior, your application does not have to request the adaptive behavior explicitly. In the version 1.2 release, however, the buffering mode was "**full**" by default and the application had to request the adaptive buffering mode explicitly.

There are three ways that an application can request that statement execution should use adaptive buffering:

- The application can set the connection property **responseBuffering** to "adaptive". For more information on setting the connection properties, see Setting the Connection Properties.
- The application can use the setResponseBuffering method of the SQLServerDataSource object to set the response buffering mode for all connections created through that SQLServerDataSource object.
- The application can use the setResponseBuffering method of the SQLServerStatement class to set the

response buffering mode for a particular statement object.

When using the JDBC Driver version 1.2, applications needed to cast the statement object to a SQLServerStatement class to use the setResponseBuffering method. The code examples in the Reading Large Data Sample and Reading Large Data with Stored Procedures Sample demonstrate this old usage.

However, with the JDBC driver version 2.0, applications can use the isWrapperFor method and the unwrap method to access the vendor-specific functionality without any assumption about the implementation class hierarchy. For example code, see the Updating Large Data Sample topic.

Retrieving Large Data with Adaptive Buffering

When large values are read once by using the get<Type>Stream methods, and the ResultSet columns and the CallableStatement OUT parameters are accessed in the order returned by the SQL Server, adaptive buffering minimizes the application memory usage when processing the results. When using adaptive buffering:

- The get<Type>Stream methods defined in the SQLServerResultSet and SQLServerCallableStatement classes return read-once streams by default, although the streams can be reset if marked by the application. If the application wants to `reset` the stream, it has to call the `mark` method on that stream first.
- The get<Type>Stream methods defined in the SQLServerClob and SQLServerBlob classes return streams that can always be repositioned to the start position of the stream without calling the `mark` method.

When the application uses adaptive buffering, the values retrieved by the get<Type>Stream methods can only be retrieved once. If you try to call any get<Type> method on the same column or parameter after calling the get<Type>Stream method of the same object, an exception is thrown with the message, "The data has been accessed and is not available for this column or parameter".

NOTE

A call to ResultSet.close() in the middle of processing a ResultSet would require the Microsoft JDBC Driver for SQL Server to read and discard all remaining packets. This may take substantial time if the query returned a large data set and especially if the network connection is slow.

Guidelines for Using Adaptive Buffering

Developers should follow these important guidelines to minimize memory usage by the application:

- Avoid using the connection string property **selectMethod=cursor** to allow the application to process a very large result set. The adaptive buffering feature allows applications to process very large forward-only, read-only result sets without using a server cursor. Note that when you set **selectMethod=cursor**, all forward-only, read-only result sets produced by that connection are impacted. In other words, if your application routinely processes short result sets with a few rows, creating, reading, and closing a server cursor for each result set will use more resources on both client-side and server-side than is the case where the **selectMethod** is not set to **cursor**.
- Read large text or binary values as streams by using the getAsciiStream, the getBinaryStream, or the getCharacterStream methods instead of the getBlob or the getClob methods. Starting with the version 1.2 release, the SQLServerCallableStatement class provides new get<Type>Stream methods for this purpose.
- Ensure that columns with potentially large values are placed last in the list of columns in a SELECT statement and that the get<Type>Stream methods of the SQLServerResultSet are used to access the columns in the order they are selected.
- Ensure that OUT parameters with potentially large values are declared last in the list of parameters in the SQL used to create the SQLServerCallableStatement. In addition, ensure that the get<Type>Stream

methods of the SQLServerCallableStatement are used to access the OUT parameters in the order they are declared.

- Avoid executing more than one statement on the same connection simultaneously. Executing another statement before processing the results of the previous statement may cause the unprocessed results to be buffered into the application memory.
- There are some cases where using **selectMethod=cursor** instead of **responseBuffering=adaptive** would be more beneficial, such as:
  - If your application processes a forward-only, read-only result set slowly, such as reading each row after some user input, using **selectMethod=cursor** instead of **responseBuffering=adaptive** might help reduce resource usage by SQL Server.
  - If your application processes two or more forward-only, read-only result sets at the same time on the same connection, using **selectMethod=cursor** instead of **responseBuffering=adaptive** might help reduce the memory required by the driver while processing these result sets.

  In both cases, you need to consider the overhead of creating, reading, and closing the server cursors.

In addition, the following list provides some recommendations for scrollable and forward-only updatable result sets:

- For scrollable result sets, when fetching a block of rows the driver always reads into memory the number of rows indicated by the getFetchSize method of the SQLServerResultSet object, even when the adaptive buffering is enabled. If scrolling causes an OutOfMemoryError, you can reduce the number of rows fetched by calling the setFetchSize method of the SQLServerResultSet object to set the fetch size to a smaller number of rows, even down to 1 row, if necessary. If this does not prevent an OutOfMemoryError, avoid including very large columns in scrollable result sets.
- For forward-only updatable result sets, when fetching a block of rows the driver normally reads into memory the number of rows indicated by the getFetchSize method of the SQLServerResultSet object, even when the adaptive buffering is enabled on the connection. If calling the next method of the SQLServerResultSet object results in an OutOfMemoryError, you can reduce the number of rows fetched by calling the setFetchSize method of the SQLServerResultSet object to set the fetch size to a smaller number of rows, even down to 1 row, if necessary. You can also force the driver not to buffer any rows by calling the setResponseBuffering method of the SQLServerStatement object with "**adaptive**" parameter before executing the statement. Because the result set is not scrollable, if the application accesses a large column value by using one of the get<Type>Stream methods, the driver discards the value as soon as the application reads it just as it does for the forward-only read-only result sets.

See Also

Improving Performance and Reliability with the JDBC Driver

Sparse Columns

8/13/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

Sparse columns are ordinary columns that have an optimized storage for null values. Sparse columns reduce the space requirements for null values at the cost of more overhead to retrieve non-null values. Consider using sparse columns when the space saved is at least 20 percent to 40 percent.

The SQL Server JDBC Driver 3.0 supports sparse columns when you connect to a SQL Server 2008 (or later) server. You can use SQLServerDatabaseMetaData.getColumns, SQLServerDatabaseMetaData.getFunctionColumns, or SQLServerDatabaseMetaData.getProcedureColumns to determine which column is sparse and which column is the column set column.

The code file for this sample is named SparseColumns.java, and it can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\sparse
```

Column sets are computed columns that return all sparse columns in untyped XML form. You should consider using column sets when the number of columns in a table is large or greater than 1024 or operating on individual sparse columns is cumbersome. A column set can contain up to 30,000 columns.

Example

Description

This sample demonstrates how to detect column sets. It also shows how to parse a column set's XML output to get the data from the sparse columns.

The code listing is the Java source code. Before you compile the application, change the connection string.

Code

```
import java.io.StringReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;


public class SparseColumns {

    public static void main(String args[]) {

        // Create a variable for the connection string.
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";
```

```
        try (Connection con = DriverManager.getConnection(connectionUrl); Statement stmt =
con.createStatement()) {

            createColdCallingTable(stmt);

            // Determine the column set column
            String columnSetColName = null;
            String strCmd = "SELECT name FROM sys.columns WHERE object_id=(SELECT OBJECT_ID('ColdCalling'))
AND is_column_set = 1";

            try (ResultSet rs = stmt.executeQuery(strCmd)) {
                if (rs.next()) {
                    columnSetColName = rs.getString(1);
                    System.out.println(columnSetColName + " is the column set column!");
                }
            }

            strCmd = "SELECT * FROM ColdCalling";
            try (ResultSet rs = stmt.executeQuery(strCmd)) {

                // Iterate through the result set
                ResultSetMetaData rsmd = rs.getMetaData();

                DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
                DocumentBuilder db = dbf.newDocumentBuilder();
                InputSource is = new InputSource();
                while (rs.next()) {
                    // Iterate through the columns
                    for (int i = 1; i <= rsmd.getColumnCount(); ++i) {
                        String name = rsmd.getColumnName(i);
                        String value = rs.getString(i);

                        // If this is the column set column
                        if (name.equalsIgnoreCase(columnSetColName)) {
                            System.out.println(name);

                            // Instead of printing the raw XML, parse it
                            if (value != null) {
                                // Add artificial root node "sparse" to ensure XML is well formed
                                String xml = "<sparse>" + value + "</sparse>";

                                is.setCharacterStream(new StringReader(xml));
                                Document doc = db.parse(is);

                                // Extract the NodeList from the artificial root node that was added
                                NodeList list = doc.getChildNodes();
                                Node root = list.item(0); // This is the <sparse> node
                                NodeList sparseColumnList = root.getChildNodes(); // These are the xml column
nodes

                                // Iterate through the XML document
                                for (int n = 0; n < sparseColumnList.getLength(); ++n) {
                                    Node sparseColumnNode = sparseColumnList.item(n);
                                    String columnName = sparseColumnNode.getNodeName();
                                    // The column value is not in the sparseColumNode, it is the value of the
                                    // first child of it
                                    Node sparseColumnValueNode = sparseColumnNode.getFirstChild();
                                    String columnValue = sparseColumnValueNode.getNodeValue();

                                    System.out.println("\t" + columnName + "\t: " + columnValue);
                                }
                            }
                        } else { // Just print the name + value of non-sparse columns
                            System.out.println(name + "\t: " + value);
                        }
                    }
                    System.out.println();// New line between rows
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void createColdCallingTable(Statement stmt) throws SQLException {
        stmt.execute("if exists (select * from sys.objects where name = 'ColdCalling')" + "drop table 
ColdCalling");

        String sql = "CREATE TABLE ColdCalling  (  ID int IDENTITY(1,1) PRIMARY KEY,  [Date] date,  [Time] 
time,  PositiveFirstName nvarchar(50) SPARSE,  PositiveLastName nvarchar(50) SPARSE,  SpecialPurposeColumns 
XML COLUMN_SET FOR ALL_SPARSE_COLUMNS  );";
        stmt.execute(sql);

        sql = "INSERT ColdCalling ([Date], [Time])  VALUES ('10-13-09','07:05:24')  ";
        stmt.execute(sql);

        sql = "INSERT ColdCalling ([Date], [Time], PositiveFirstName, PositiveLastName)  VALUES ('07-20-
09','05:00:24', 'AA', 'B')  ";
        stmt.execute(sql);

        sql = "INSERT ColdCalling ([Date], [Time], PositiveFirstName, PositiveLastName)  VALUES ('07-20-
09','05:15:00', 'CC', 'DD')  ";
        stmt.execute(sql);
    }
}
```

See Also

Improving Performance and Reliability with the JDBC Driver

Prepared Statement Metadata Caching for the JDBC Driver

5/3/2018 • 5 minutes to read • Edit Online

Download JDBC Driver

This article provides information on the two changes that are implemented to enhance the performance of the driver.

Batching of Unprepare for Prepared Statements

Since version 6.1.6-preview, an improvement in performance was implemented through minimizing server round trips to SQL Server. Previously, for every prepareStatement query, a call to unprepare was also sent. Now, the driver is batching unprepare queries up to the threshold "ServerPreparedStatementDiscardThreshold", which has a default value of 10.

NOTE

Users can change the default value with the following method: setServerPreparedStatementDiscardThreshold(int value)

One more change introduced from 6.1.6-preview is that prior to this, driver would always call sp_prepexec. Now, for the first execution of a prepared statement, driver calls sp_executesql and for the rest it executes sp_prepexec and assigns a handle to it. More details can be found here.

NOTE

Users can change the default behavior to the previous versions of always calling sp_prepexec by setting enablePrepareOnFirstPreparedStatementCall to **true** using the following method: setEnablePrepareOnFirstPreparedStatementCall(boolean value)

List of the New APIs Introduced With This Change, for Batching of Unprepare for Prepared Statements

SQLServerConnection

NEW METHOD	DESCRIPTION
int getDiscardedServerPreparedStatementCount()	Returns the number of currently outstanding prepared statement unprepare actions.
void closeUnreferencedPreparedStatementHandles()	Forces the unprepare requests for any outstanding discarded prepared statements to be executed.

NEW METHOD	DESCRIPTION
boolean getEnablePrepareOnFirstPreparedStatementCall()	Returns the behavior for a specific connection instance. If false the first execution calls sp_executesql and not prepare a statement, once the second execution happens it calls sp_prepexec and actually setup a prepared statement handle. Following executions calls sp_execute. This relieves the need for sp_unprepare on prepared statement close if the statement is only executed once. The default for this option can be changed by calling setDefaultEnablePrepareOnFirstPreparedStatementCall().
void setEnablePrepareOnFirstPreparedStatementCall(boolean value)	Specifies the behavior for a specific connection instance. If value is false the first execution calls sp_executesql and not prepare a statement, once the second execution happens it calls sp_prepexec and actually setup a prepared statement handle. Following executions calls sp_execute. This relieves the need for sp_unprepare on prepared statement close if the statement is only executed once.
int getServerPreparedStatementDiscardThreshold()	Returns the behavior for a specific connection instance. This setting controls how many outstanding prepared statement discard actions (sp_unprepare) can be outstanding per connection before a call to clean up the outstanding handles on the server is executed. If the setting is <= 1, unprepare actions are executed immediately on prepared statement close. If it is set to {@literal >} 1, these calls are batched together to avoid overhead of calling sp_unprepare too often. The default for this option can be changed by calling getDefaultServerPreparedStatementDiscardThreshold().
void setServerPreparedStatementDiscardThreshold(int value)	Specifies the behavior for a specific connection instance. This setting controls how many outstanding prepared statement discard actions (sp_unprepare) can be outstanding per connection before a call to clean up the outstanding handles on the server is executed. If the setting is <= 1 unprepare actions are executed immediately on prepared statement close. If it is set to > 1 these calls are batched together to avoid overhead of calling sp_unprepare too often.

SQLServerDataSource

NEW METHOD	DESCRIPTION
void setEnablePrepareOnFirstPreparedStatementCall(boolean enablePrepareOnFirstPreparedStatementCall)	If this configuration is false the first execution of a prepared statement calls sp_executesql and not prepare a statement, once the second execution happens it calls sp_prepexec and actually setup a prepared statement handle. Following executions calls sp_execute. This relieves the need for sp_unprepare on prepared statement close if the statement is only executed once.
boolean getEnablePrepareOnFirstPreparedStatementCall()	If this configuration returns false the first execution of a prepared statement calls sp_executesql and not prepare a statement, once the second execution happens, it calls sp_prepexec and actually setup a prepared statement handle. Following executions calls sp_execute. This relieves the need for sp_unprepare on prepared statement close if the statement is only executed once.

NEW METHOD	DESCRIPTION
void setServerPreparedStatementDiscardThreshold(int serverPreparedStatementDiscardThreshold)	This setting controls how many outstanding prepared statement discard actions (sp_unprepare) can be outstanding per connection before a call to clean up the outstanding handles on the server is executed. If the setting is <= 1 unprepare actions are executed immediately on prepared statement close. If it is set to {@literal >} 1 these calls are batched together to avoid overhead of calling sp_unprepare too often
int getServerPreparedStatementDiscardThreshold()	This setting controls how many outstanding prepared statement discard actions (sp_unprepare) can be outstanding per connection before a call to clean up the outstanding handles on the server is executed. If the setting is <= 1 unprepare actions are executed immediately on prepared statement close. If it is set to {@literal >} 1 these calls are batched together to avoid overhead of calling sp_unprepare too often.

Prepared Statement Metatada caching

As of 6.3.0-preview version, Microsoft JDBC driver for SQL Server supports prepared statement caching. Prior to v6.3.0-preview, if one executes a query that has been already prepared and stored in the cache, calling the same query again will not result in preparing it. Now, the driver looks up the query in cache and find the handle and execute it with sp_execute. Prepared Statement Metadata caching is **disabled** by default. In order to enable it, you need to call the following method on the connection object:

```
setStatementPoolingCacheSize(int value) //value is the desired cache size (any value bigger than 0)
setDisableStatementPooling(boolean value) //false allows the caching to take place
```

For example: `connection.setStatementPoolingCacheSize(10)` `connection.setDisableStatementPooling(false)`

List of the New APIs Introduced With This Change, for Prepared Statement Metadata Caching

SQLServerConnection

NEW METHOD	DESCRIPTION
void setDisableStatementPooling(boolean value)	Sets statement pooling to true or false.
boolean getDisableStatementPooling()	Returns true if statement pooling is disabled.
void setStatementPoolingCacheSize(int value)	Specifies the size of the prepared statement cache for this connection. A value less than 1 means no cache.
int getStatementPoolingCacheSize()	Returns the size of the prepared statement cache for this connection. A value less than 1 means no cache.
int getStatementHandleCacheEntryCount()	Returns the current number of pooled prepared statement handles.
boolean isPreparedStatementCachingEnabled()	Whether statement pooling is enabled or not for this connection.

SQLServerDataSource

NEW METHOD	DESCRIPTION
void setDisableStatementPooling(boolean disableStatementPooling)	Sets the statement pooling to true or false
boolean getDisableStatementPooling()	Returns true if statement pooling is disabled.
void setStatementPoolingCacheSize(int statementPoolingCacheSize)	Specifies the size of the prepared statement cache for this connection. A value less than 1 means no cache.
int getStatementPoolingCacheSize()	Returns the size of the prepared statement cache for this connection. A value less than 1 means no cache.

See Also

Improving Performance and Reliability with the JDBC Driver

Diagnosing Problems with the JDBC Driver

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

No matter how well an application is designed and developed, problems will inevitably occur. When they do, it is important to have some techniques for diagnosing those problems. When using the Microsoft JDBC Driver for SQL Server, some common problems that can occur are not having the right driver version or being unable to connect to a database.

The topics in this section discuss a number of techniques for diagnosing these and other problems including error handling, checking the driver version, tracing, and troubleshooting connectivity issues.

In This Section

TOPIC	DESCRIPTION
Handling Errors	Describes how to handle errors that are returned from SQL Server.
Getting the Driver Version	Describes how to determine which version of the JDBC driver is installed.
Tracing Driver Operation	Describes how to enable tracing when using the JDBC driver.
Troubleshooting Connectivity	Describes how to troubleshoot database connectivity.
Accessing Diagnostic Information in the Extended Events Log	Describes how to use information in the server's extended events log to understand connection failures.

See Also

Overview of the JDBC Driver

Handling Errors

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When using the Microsoft JDBC Driver for SQL Server, all database error conditions are returned to your Java application as exceptions using the SQLServerException class. The following methods of the SQLServerException class are inherited from java.sql.SQLException and java.lang.Throwable; and they can be used to return specific information about the SQL Server error that has occurred:

- getSQLState returns the standard X/Open or SQL99 state code of the exception.
- getErrorCode returns the specific database error number.
- getMessage returns the full text of the exception. The error message text describes the problem, and frequently includes placeholders for information, such as object names, that are inserted in the error message when it is displayed.
- getNextException returns the next SQLServerException object or null if there are no more exception objects to return.

  In the following example, an open connection to the SQL Server AdventureWorks sample database is passed in to the function and a malformed SQL statement is constructed that does not have a FROM clause. Then, the statement is run and an SQL exception is processed.

```
public static void executeSQLException(Connection con) {
    try (Statement stmt = con.createStatement();) {
        String SQL = "SELECT TOP 10 * Person.Contact";
        ResultSet rs = stmt.executeQuery(SQL);

        while (rs.next()) {
            System.out.println(rs.getString("FirstName") + " " + rs.getString("LastName"));
        }
    }
    catch (SQLException se) {
        do {
            System.out.println("SQL STATE: " + se.getSQLState());
            System.out.println("ERROR CODE: " + se.getErrorCode());
            System.out.println("MESSAGE: " + se.getMessage());
            System.out.println();
            se = se.getNextException();
        }
        while (se != null);
    }
}
```

See Also

Diagnosing Problems with the JDBC Driver

Getting the Driver Version

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The version of the installed Microsoft JDBC Driver for SQL Server can be found in the following ways:

- Call the SQLServerDatabaseMetaData methods getDriverMajorVersion, getDriverMinorVersion, or getDriverVersion.
- The version is displayed in the readme.txt file of the product distribution.

  In addition, the JDBC driver name can be returned from the getDriverName method call on the SQLServerDatabaseMetaData class. It will return, for example, "Microsoft JDBC Driver 6.4 for SQL Server".

  The following is an example of the output from calls to the methods of the SQLServerDatabaseMetaData class:

  `getDriverName = Microsoft JDBC Driver 6.4 for SQL Server`

  `getDriverMajorVersion = 6`

  `getDriverMinorVersion = 4`

  `getDriverVersion = 6.4.` *XXX.X*

  Where "xxx.x" is the final version number.

See Also

Diagnosing Problems with the JDBC Driver

Tracing Driver Operation

8/13/2018 • 8 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server supports the use of tracing (or logging) to help resolve issues and problems with the JDBC driver when it's used in your application. To enable the use of tracing, the JDBC driver uses the logging APIs in java.util.logging, which provides a set of classes for creating Logger and LogRecord objects.

NOTE

For the native component (sqljdbc_xa.dll) that is included with the JDBC driver, tracing is enabled by the Built-In Diagnostics (BID) framework. For information about BID, see Data Access Tracing in SQL Server.

When you develop your application, you can make calls to Logger objects, which in turn create LogRecord objects, which are then passed to Handler objects for processing. Logger and Handler objects both use logging levels, and optionally logging filters, to regulate which LogRecords are processed. When the logging operations are complete, the Handler objects can optionally use Formatter objects to publish the log information.

By default, the java.util.logging framework writes its output to a file. This output log file must have write permissions for the context under which the JDBC driver is running.

NOTE

For more information about using the various logging objects for program tracing, see the Java Logging APIs documentation on the Sun Microsystems Web site.

The following sections describe the logging levels and the categories that can be logged, and provide information about how to enable tracing in your application.

Logging Levels

Every log message that is created has an associated logging level. The logging level determines the importance of the log message, which is defined by the **Level** class in java.util.logging. Enabling logging at one level also enables logging at all higher levels. This section describes the logging levels for both public logging categories and internal logging categories. For more information about the logging categories, see the Logging Categories section in this article.

The following table describes each of the available logging levels for public logging categories.

NAME	DESCRIPTION
SEVERE	Indicates a serious failure and is the highest level of logging. In the JDBC driver, this level is used for reporting errors and exceptions.
WARNING	Indicates a potential problem.
INFO	Provides informational messages.

NAME	DESCRIPTION
CONFIG	Provides configuration messages. Note that the JDBC driver doesn't currently provide any configuration messages.
FINE	Provides basic tracing information including all exceptions thrown by the public methods.
FINER	Provides detailed tracing information including all public method entry and exit points with the associated parameter data types, and all public properties for public classes. In addition, input parameters, output parameters, and method return values except CLOB, BLOB, NCLOB, Reader, <stream> return value types.
FINEST	Provides highly detailed tracing information. This is the lowest level of logging.
OFF	Turns off logging.
ALL	Enables logging of all messages.

The following table describes each of the available logging levels for the internal logging categories.

NAME	DESCRIPTION
SEVERE	Indicates a serious failure and is the highest level of logging. In the JDBC driver, this level is used for reporting errors and exceptions.
WARNING	Indicates a potential problem.
INFO	Provides informational messages.
FINE	Provides tracing information including basic object creation and destruction. In addition, all exceptions thrown by the public methods.
FINER	Provides detailed tracing information including all public method entry and exit points with the associated parameter data types, and all public properties for public classes. In addition, input parameters, output parameters, and method return values except CLOB, BLOB, NCLOB, Reader, <stream> return value types. The following logging categories existed in version 1.2 of the JDBC driver and had the FINE logging level: SQLServerConnection, SQLServerStatement, XA, and SQLServerDataSource. Beginning in the version 2.0 release, these are upgraded to the FINER level.
FINEST	Provides highly detailed tracing information. This is the lowest level of logging. The following logging categories existed in version 1.2 of the JDBC driver and had the FINEST logging level: TDS.DATA and TDS.TOKEN. Beginning in the version 2.0 release, they retain the FINEST logging level.

NAME	DESCRIPTION
OFF	Turns off logging.
ALL	Enables logging of all messages.

Logging Categories

When you create a Logger object, you must tell the object which named entity or category that you're interested in getting log information from. The JDBC driver supports the following public logging categories, which are all defined in the com.microsoft.sqlserver.jdbc driver package.

NAME	DESCRIPTION
Connection	Logs messages in the SQLServerConnection class. The applications can set the logging level as FINER.
Statement	Logs messages in the SQLServerStatement class. The applications can set the logging level as FINER.
DataSource	Logs messages in the SQLServerDataSource class. The applications can set the logging level as FINE.
ResultSet	Logs messages in the SQLServerResultSet class. The applications can set the logging level as FINER.
Driver	Logs messages in the SQLServerDriver class. The applications can set the logging level as FINER.

Starting with the Microsoft JDBC Driver version 2.0, the driver also provides the com.microsoft.sqlserver.jdbc.internals package, which includes the logging support for the following internal logging categories.

NAME	DESCRIPTION
AuthenticationJNI	Logs messages regarding the Windows integrated authentication issues (when the **authenticationScheme** connection property is implicitly or explicitly set to **NativeAuthentication**). The applications can set the logging level as FINEST and FINE.
SQLServerConnection	Logs messages in the SQLServerConnection class. The applications can set the logging level as FINE and FINER.
SQLServerDataSource	Logs messages in the SQLServerDataSource, SQLServerConnectionPoolDataSource, and SQLServerPooledConnection classes. The applications can set the logging level as FINER.

NAME	DESCRIPTION
InputStream	Logs messages regarding the following data types: java.io.InputStream, java.io.Reader and the data types, which have a max specifier such as varchar, nvarchar, and varbinary data types. The applications can set the logging level as FINER.
SQLServerException	Logs messages in the SQLServerException class. The applications can set the logging level as FINE.
SQLServerResultSet	Logs messages in the SQLServerResultSet class. The applications can set the logging level as FINE, FINER, and FINEST.
SQLServerStatement	Logs messages in the SQLServerStatement class. The applications can set the logging level as FINE, FINER, and FINEST.
XA	Logs messages for all XA transactions in the SQLServerXADataSource class. The applications can set the logging level as FINE and FINER.
KerbAuthentication	Logs messages regarding type 4 Kerberos authentication (when the **authenticationScheme** connection property is set to **JavaKerberos**). The application can set the logging level as FINE or FINER.
TDS.DATA	Logs messages containing the TDS protocol-level conversation between the driver and SQL Server. The detailed contents of each TDS packet sent and received are logged in ASCII and hexadecimal. The login credentials (user names and passwords) aren't logged. All other data is logged. This category creates very verbose and detailed messages, and can only be enabled by setting the logging level to FINEST.
TDS.Channel	This category traces actions of the TCP communications channel with SQL Server. The logged messages include socket opening and closing as well as reads and writes. It also traces messages related to establishing a Secure Sockets Layer (SSL) connection with SQL Server. This category can only be enabled by setting the logging level to FINE, FINER, or FINEST.
TDS.Writer	This category traces writes to the TDS channel. Note that only the length of the writes is traced, not the contents. This category also traces issues when an attention signal is sent to the server to cancel a statement's execution. This category can only be enabled by setting the logging level to FINEST.

NAME	DESCRIPTION
TDS.Reader	This category traces certain read operations from the TDS channel at the FINEST level. At the FINEST level, tracing can be verbose. At WARNING and SEVERE levels, this category traces when the driver receives an invalid TDS protocol from SQL Server before the driver closes the connection. This category can only be enabled by setting the logging level to FINER and FINEST.
TDS.Command	This category traces low-level state transitions and other information associated with executing TDS commands, such as Transact-SQL statement executions, ResultSet cursor fetches, commits, and so on. This category can only be enabled by setting the logging level to FINEST.
TDS.TOKEN	This category logs only the tokens within the TDS packets, and is less verbose than the TDS.DATA category. It can only be enabled by setting the logging level to FINEST. At the FINEST level, this category traces TDS tokens as they're processed in the response. At the SEVERE level, this category traces when an invalid TDS token is encountered.
SQLServerDatabaseMetaData	Logs messages in the SQLServerDatabaseMetaData class. The applications can set the logging level as FINE.
SQLServerResultSetMetaData	Logs messages in the SQLServerResultSetMetaData class. The applications can set the logging level as FINE.
SQLServerParameterMetaData	Logs messages in the SQLServerParameterMetaData class. The applications can set the logging level as FINE.
SQLServerBlob	Logs messages in the SQLServerBlob class. The applications can set the logging level as FINE.
SQLServerClob	Logs messages in the SQLServerClob class. The applications can set the logging level as FINE.
SQLServerSQLXML	Logs messages in the internal SQLServerSQLXML class. The applications can set the logging level as FINE.
SQLServerDriver	Logs messages in the SQLServerDriver class. The applications can set the logging level as FINE.
SQLServerNClob	Logs messages in the SQLServerNClob class. The applications can set the logging level as FINE.

Enabling Tracing Programmatically

Tracing can be enabled programmatically by creating a Logger object and indicating the category to be logged. For example, the following code shows how to enable logging for SQL statements:

```
Logger logger = Logger.getLogger("com.microsoft.sqlserver.jdbc.Statement");
logger.setLevel(Level.FINER);
```

To turn off logging in your code, use the following:

```
logger.setLevel(Level.OFF);
```

To log all available categories, use the following:

```
Logger logger = Logger.getLogger("com.microsoft.sqlserver.jdbc");
logger.setLevel(Level.FINE);
```

To disable a specific category from being logged, use the following:

```
Logger logger = Logger.getLogger("com.microsoft.sqlserver.jdbc.Statement");
logger.setLevel(Level.OFF);
```

Enabling Tracing by Using the Logging.Properties File

You can also enable tracing by using the `logging.properties` file, which can be found in the `lib` directory of your Java Runtime Environment (JRE) installation. This file can be used to set the default values for the loggers and handlers that will be used when tracing is enabled.

The following is an example of the settings that you can make in the `logging.properties` files:

```
# Specify the handler, the handlers will be installed during VM startup.
handlers= java.util.logging.FileHandler

# Default global logging level.
.level= OFF

# default file output is in user's home directory.
java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 5000000
java.util.logging.FileHandler.count = 20
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.level = FINEST

# Facility specific properties.
com.microsoft.sqlserver.jdbc.level=FINEST
```

NOTE

You can set the properties in the `logging.properties` file by using the LogManager object that is part of java.util.logging.

See Also

Diagnosing Problems with the JDBC Driver

Troubleshooting Connectivity

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server requires that TCP/IP be installed and running to communicate with your SQL Server database. You can use the SQL Server Configuration Manager to verify which network library protocols are installed.

A database connection attempt might fail for many reasons. These can include the following:

- TCP/IP is not enabled for SQL Server, or the server or port number specified is incorrect. Verify that SQL Server is listening with TCP/IP on the specified server and port. This might be reported with an exception similar to: "The login has failed. The TCP/IP connection to the host has failed." This indicates one of the following:
  - SQL Server is installed but TCP/IP has not been installed as a network protocol for SQL Server by using the SQL Server Network Utility for SQL Server 2000 (8.x), or the SQL Server Configuration Manager for SQL Server 2005 (9.x) and later.
  - TCP/IP is installed as a SQL Server protocol, but it is not listening on the port specified in the JDBC connection URL. The default port is 1433, but SQL Server can be configured at product installation to listen on any port. Make sure that SQL Server is listening on port 1433. Or, if the port has been changed, make sure that the port specified in the JDBC connection URL matches the changed port. For more information about JDBC connection URLs, see Building the Connection URL.
  - The address of the computer that is specified in the JDBC connection URL does not refer to a server where SQL Server is installed and started.
  - The networking operation of TCP/IP between the client and server running SQL Server is not operable. You can check TCP/IP connectivity to SQL Server by using telnet. For example, at the command prompt, type `telnet 192.168.0.0 1433` where 192.168.0.0 is the address of the computer that is running SQL Server and 1433 is the port it is listening on. If you receive a message that states "Telnet cannot connect," TCP/IP is not listening on that port for SQL Server connections. Use the SQL Server Network Utility for SQL Server 2000 (8.x), or the SQL Server Configuration Manager for SQL Server 2005 (9.x) and later to make sure that SQL Server is configured to use TCP/IP on port 1433.
  - The port that is used by the server has not been opened in the firewall. This includes the port that is used by the server or optionally, the port associated with a named instance of the server.
- The specified database name is incorrect. Make sure that you are logging on to an existing SQL Server database.
- The user name or password is incorrect. Make sure that you have the correct values.
- When you use SQL Server Authentication, the JDBC driver requires that SQL Server is installed with SQL Server Authentication, which is not the default. Make sure that this option is included when you install or configure your instance of SQL Server.

See Also

Diagnosing Problems with the JDBC Driver

Accessing Diagnostic Information in the Extended Events Log

8/13/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

In the Microsoft JDBC Driver 4.0 for SQL Server, tracing (Tracing Driver Operation) has been updated to make it easier to correlate client events with diagnostic information, such as connection failures, from the server's connectivity ring buffer and application performance information in the extended events log. For information about reading the extended events log, see View Event Session Data.

Details

For connection operations, the Microsoft JDBC Driver for SQL Server will send a client connection ID. If the connection fails, you can access the connectivity ring buffer (Connectivity troubleshooting in SQL Server 2008 with the Connectivity Ring Buffer) and find the **ClientConnectionID** field and get diagnostic information about the connection failure. Client connection IDs are logged in the ring buffer only if an error occurs. (If a connection fails before sending the prelogin packet, a client connection ID won't be generated.) The client connection ID is a 16-byte GUID. You can also find the client connection ID in the extended events target output, if the **client_connection_id** action is added to events in an extended events session. If you need further client driver diagnostic assistance, you can enable tracing and rerun the connection command to observe the **ClientConnectionID** field in the trace.

You can get the client connection ID programmatically by using ISQLServerConnection Interface. The connection ID will also be present in any connection-related exceptions.

When there's a connection error, the client connection ID in the server's Built In Diagnostics (BID) trace information and in the connectivity ring buffer can help correlate the client connections to connections on the server. For more information about BID traces on the server, see Data Access Tracing. Note, the data access tracing article also contains information about data access trace, which doesn't apply to the Microsoft JDBC Driver for SQL Server; see Tracing Driver Operation for information on doing a data access trace using the Microsoft JDBC Driver for SQL Server.

The JDBC Driver also sends a thread-specific activity ID. The activity ID is captured in the extended events sessions if the sessions are started with the TRACK_CAUSAILITY option enabled. For performance issues with an active connection, you can get the activity ID from the client's trace (ActivityID field) and then locate the activity ID in the extended events output. The activity ID in extended events is a 16-byte GUID (not the same as the GUID for the client connection ID) appended with a 4-byte sequence number. The sequence number represents the order of a request within a thread. The ActivityId is sent for SQL batch statements and RPC requests. To enable sending ActivityId to the server, you first need to specify the following key-value pair in the Logging.Properties file:

```
com.microsoft.sqlserver.jdbc.traceactivity = on
```

Any value other than `on` (case sensitive) will disable sending the ActivityId.

For more information, see Tracing Driver Operation. This trace flag is used with corresponding JDBC object loggers to decide whether to trace and send the ActivityId in the JDBC driver. In addition to updating the Logging.Properties file, the logger com.microsoft.sqlserver.jdbc needs to be enabled at FINER or higher. If you want to send ActivityId to the server for requests that are made by a particular class, the corresponding class logger needs to be enabled at FINER or FINEST. For example, if the class is, SQLServerStatement, enable the logger com.microsoft.sqlserver.jdbc.SQLServerStatement.

The following sample uses Transact-SQL to start an extended events session that will be stored in a ring buffer and will record the activity ID sent from a client on RPC and batch operations:

```
create event session MySession on server
add event connectivity_ring_buffer_recorded,
add event sql_statement_starting (action (client_connection_id)),
add event sql_statement_completed (action (client_connection_id)),
add event rpc_starting (action (client_connection_id)),
add event rpc_completed (action (client_connection_id))
add target ring_buffer with (track_causality=on)
```

See Also

Diagnosing Problems with the JDBC Driver

Sample JDBC Driver Applications

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server sample applications demonstrate various features of the JDBC driver. Additionally, they demonstrate good programming practices that you can follow when using the JDBC driver with a SQL Server database.

All the sample applications are contained in *.java code files that can be compiled and run on your local computer, and they are located in various subfolders in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples
```

The topics in this section describe how to configure and run the sample applications, and include a discussion of what the sample applications demonstrate.

In This Section

TOPIC	DESCRIPTION
Connecting and Retrieving Data	These sample applications demonstrate how to connect to a SQL Server database. They also demonstrate different ways in which to retrieve data from a SQL Server database.
Working with Data Types (JDBC)	These sample applications demonstrate how to use the JDBC driver data type methods to work with data in a SQL Server database.
Working with Result Sets	These sample applications demonstrate how to use result sets to process data contained in a SQL Server database.
Working with Large Data	These sample applications demonstrate how to use adaptive buffering to retrieve large-value data from a SQL Server database without the overhead of server cursors.
SQL data discovery and classification	This sample application demonstrates how to retreive Data Discovery and Classification information contained in a SQL Server database from a ResultSet object using JDBC Driver.

See Also

Overview of the JDBC Driver

Connecting and Retrieving Data

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When you are working with the Microsoft JDBC Driver for SQL Server, there are two primary methods for establishing a connection to a SQL Server database. One is to set connection properties in the connection URL, and then call the getConnection method of the DriverManager class to return a SQLServerConnection object.

NOTE
For a list of the connection properties supported by the JDBC driver, see Setting the Connection Properties.

The second method involves setting the connection properties by using setter methods of the SQLServerDataSource class, and then calling the getConnection method to return a SQLServerConnection object.

The topics in this section describe the different ways in which you can connect to a SQL Server database, and they also demonstrate different techniques for retrieving data.

In This Section

TOPIC	DESCRIPTION
Connection URL Sample	Describes how to use a connection URL to connect to SQL Server and then use an SQL statement to retrieve data.
Data Source Sample	Describes how to use a data source to connect to SQL Server and then use a stored procedure to retrieve data.

See Also

Sample JDBC Driver Applications

Connection URL Sample

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to connect to a SQL Server database by using a connection URL. It also demonstrates how to retrieve data from a SQL Server database by using an SQL statement.

The code file for this sample is named ConnectURL.java, and it can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\connections
```

Requirements

To run this sample application, you must set the classpath to include the mssql-jdbc jar file. You'll also need access to the AdventureWorks sample database. For more information about how to set the classpath, see Using the JDBC Driver.

NOTE

The Microsoft JDBC Driver for SQL Server provides mssql-jdbc class library files to be used depending on your preferred Java Runtime Environment (JRE) settings. For more information about which JAR file to choose, see System Requirements for the JDBC Driver.

Example

In the following example, the sample code sets various connection properties in the connection URL, and then calls the getConnection method of the DriverManager class to return a SQLServerConnection object.

Next, the sample code uses the createStatement method of the SQLServerConnection object to create a SQLServerStatement object, and then the executeQuery method is called to execute the SQL statement.

Finally, the sample uses the SQLServerResultSet object returned from the executeQuery method to iterate through the results returned by the SQL statement.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ConnectURL {
    public static void main(String[] args) {

        // Create a variable for the connection string.
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";

        try (Connection con = DriverManager.getConnection(connectionUrl); Statement stmt =
con.createStatement();) {
            String SQL = "SELECT TOP 10 * FROM Person.Contact";
            ResultSet rs = stmt.executeQuery(SQL);

            // Iterate through the data in the result set and display it.
            while (rs.next()) {
                System.out.println(rs.getString("FirstName") + " " + rs.getString("LastName"));
            }
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

See Also

Connecting and Retrieving Data

Data Source Sample

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to connect to a SQL Server database by using a data source object. It also demonstrates how to retrieve data from a SQL Server database by using a stored procedure.

The code file for this sample is named ConnectDataSource.java, and it can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\connections
```

Requirements

To run this sample application, you must set the classpath to include the mssql-jdbc jar file. You'll also need access to the AdventureWorks sample database. For more information about how to set the classpath, see Using the JDBC Driver.

NOTE

The Microsoft JDBC Driver for SQL Server provides mssql-jdbc class library files to be used depending on your preferred Java Runtime Environment (JRE) settings. For more information about which JAR file to choose, see System Requirements for the JDBC Driver.

Example

In the following example, the sample code sets various connection properties by using setter methods of the SQLServerDataSource object, and then calls the getConnection method of the SQLServerDataSource object to return a SQLServerConnection object.

Next, the sample code uses the prepareCall method of the SQLServerConnection object to create a SQLServerCallableStatement object, and then the executeQuery method is called to execute the stored procedure.

Finally, the sample uses the SQLServerResultSet object returned from the executeQuery method to iterate through the results returned by the stored procedure.

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;

import com.microsoft.sqlserver.jdbc.SQLServerDataSource;

public class ConnectDataSource {

    public static void main(String[] args) {

        // Create datasource.
        SQLServerDataSource ds = new SQLServerDataSource();
        ds.setUser("<user>");
        ds.setPassword("<password>");
        ds.setServerName("<server>");
        ds.setPortNumber(<port>);
        ds.setDatabaseName("AdventureWorks");

        try (Connection con = ds.getConnection();
                CallableStatement cstmt = con.prepareCall("{call dbo.uspGetEmployeeManagers(?)}");) {
            // Execute a stored procedure that returns some data.
            cstmt.setInt(1, 50);
            ResultSet rs = cstmt.executeQuery();

            // Iterate through the data in the result set and display it.
            while (rs.next()) {
                System.out.println("EMPLOYEE: " + rs.getString("LastName") + ", " +
rs.getString("FirstName"));
                System.out.println("MANAGER: " + rs.getString("ManagerLastName") + ", " +
rs.getString("ManagerFirstName"));
                System.out.println();
            }
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

See Also

Connecting and Retrieving Data

Working with Data Types (JDBC)

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The primary function of the Microsoft JDBC Driver for SQL Server is to allow Java developers to access data contained in SQL Server databases. To accomplish this, the JDBC driver mediates the conversion between SQL Server data types and Java language types and objects.

NOTE
For a detailed discussion of the SQL Server and JDBC driver data types, including their differences and how they are converted to Java language data types, see Understanding the JDBC Driver Data Types.

In order to work with SQL Server data types, the JDBC driver provides get<Type> and set<Type> methods for the SQLServerPreparedStatement and SQLServerCallableStatement classes, and it provides get<Type> and update<Type> methods for the SQLServerResultSet class. Which method you use depends on the type of data that you are working with, and whether you are using result sets or queries.

The topics in this section describe how to use the JDBC driver data types to access SQL Server data in your Java applications.

In This Section

TOPIC	DESCRIPTION
Basic Data Types Sample	Describes how to use result set getter methods to retrieve basic SQL Server data type values, and how to use result set update methods to update those values.
SQLXML Data Type Sample	Describes how to store an XML data in a relational database, how to retrieve an XML data from a database, and how to parse an XML data with the **SQLXML** Java data type.
Spatial Data Types Sample	Describes how to store and retreive data with Spatial Datatypes 'Geometry' and 'Geography' of SQL Server database with **Geometry** and **Geography** Java types defined by Microsoft JDBC Driver.

See Also

Sample JDBC Driver Applications

Basic Data Types Sample

8/13/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to use result set getter methods to retrieve basic SQL Server data type values, and how to use result set update methods to update those values.

The code file for this sample is named BasicDT.java, and it can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\datatypes
```

Requirements

To run this sample application, you must set the classpath to include the mssql-jdbc jar file. You'll also need access to the AdventureWorks sample database. For more information about how to set the classpath, see Using the JDBC Driver.

The sample will create the required table and insert sample data in the AdventureWorks sample database:

NOTE

The Microsoft JDBC Driver for SQL Server provides mssql-jdbc class library files to be used depending on your preferred Java Runtime Environment (JRE) settings. For more information about which JAR file to choose, see System Requirements for the JDBC Driver.

Example

In the following example, the sample code makes a connection to the AdventureWorks database, and then retrieves a single row of data from the DataTypesTable test table. The custom displayRow method is then called to display all the data in the result set using various get<Type> methods of the SQLServerResultSet class.

Next, the sample uses various update<Type> methods of the SQLServerResultSet class to update the data in the result set, and then calls the updateRow method to persist that data back to the database.

Finally, the sample refreshes the data in the result set and then calls the custom displayRow method again to display the data in the result set.

```
import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Time;
import java.sql.Timestamp;

import com.microsoft.sqlserver.jdbc.SQLServerResultSet;

import microsoft.sql.DateTimeOffset;

public class BasicDataTypes {
```

```java
    private static final String tableName = "DataTypesTable";

    public static void main(String[] args) {

        // Create a variable for the connection string.
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=<database>;user=<user>;password=
<password>";

        try (Connection con = DriverManager.getConnection(connectionUrl);
                Statement stmt = con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_UPDATABLE);) {

            dropAndCreateTable(stmt);
            insertOriginalData(con);

            String SQL = "SELECT * FROM " + tableName;
            ResultSet rs = stmt.executeQuery(SQL);
            rs.next();
            displayRow("ORIGINAL DATA", rs);

            // Update the data in the result set.
            rs.updateString(2, "B");
            rs.updateString(3, "Some updated text.");
            rs.updateBoolean(4, true);
            rs.updateDouble(5, 77.89);
            rs.updateDouble(6, 1000.01);
            long timeInMillis = System.currentTimeMillis();
            Timestamp ts = new Timestamp(timeInMillis);
            rs.updateTimestamp(7, ts);
            rs.updateDate(8, new Date(timeInMillis));
            rs.updateTime(9, new Time(timeInMillis));
            rs.updateTimestamp(10, ts);

            // -480 indicates GMT - 8:00 hrs
            ((SQLServerResultSet) rs).updateDateTimeOffset(11, DateTimeOffset.valueOf(ts, -480));

            rs.updateRow();

            // Get the updated data from the database and display it.
            rs = stmt.executeQuery(SQL);
            rs.next();
            displayRow("UPDATED DATA", rs);
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void displayRow(String title,
            ResultSet rs) throws SQLException {
        System.out.println(title);
        System.out.println(rs.getInt(1) + " , " +                 // SQL integer type.
                rs.getString(2) + " , " +                         // SQL char type.
                rs.getString(3) + " , " +                         // SQL varchar type.
                rs.getBoolean(4) + " , " +                        // SQL bit type.
                rs.getDouble(5) + " , " +                         // SQL decimal type.
                rs.getDouble(6) + " , " +                         // SQL money type.
                rs.getTimestamp(7) + " , " +                      // SQL datetime type.
                rs.getDate(8) + " , " +                           // SQL date type.
                rs.getTime(9) + " , " +                           // SQL time type.
                rs.getTimestamp(10) + " , " +                     // SQL datetime2 type.
                ((SQLServerResultSet) rs).getDateTimeOffset(11)); // SQL datetimeoffset type.
        System.out.println();
    }

    private static void dropAndCreateTable(Statement stmt) throws SQLException {
        stmt.executeUpdate("if object_id('" + tableName + "','U') is not null" + " drop table " + tableName);
        String sql = "create table " + tableName + " (" + "c1 int, " + "c2 char(20), " + "c3 varchar(20), " +
"c4 bit, "
                + "c5 decimal(10,5), " + "c6 money, " + "c7 datetime, " + "c8 date, " + "c9 time(7), "
                + "c10 datetime2(7), " + "c11 datetimeoffset(7), " + ");";

        stmt.execute(sql);
    }

    private static void insertOriginalData(Connection con) throws SQLException {
        String sql = "insert into " + tableName + " values( " + "?,?,?,?,?,?,?,?,?,?,?" + ")";
        try (PreparedStatement pstmt = con.prepareStatement(sql)) {
            pstmt.setObject(1, 100);
            pstmt.setObject(2, "original text");
            pstmt.setObject(3, "original text");
            pstmt.setObject(4, false);
            pstmt.setObject(5, 12.34);
            pstmt.setObject(6, 56.78);
            pstmt.setObject(7, new java.util.Date(1453500034839L));
            pstmt.setObject(8, new java.util.Date(1453500034839L));
            pstmt.setObject(9, new java.util.Date(1453500034839L));
            pstmt.setObject(10, new java.util.Date(1453500034839L));
            pstmt.setObject(11, new java.util.Date(1453500034839L));
            pstmt.execute();
        }
    }
}
```

See Also

Working with Data Types (JDBC)

SQLXML Data Type Sample

8/2/2018 • 7 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to store XML data in a relational database, how to retrieve XML data from a database, and how to parse XML data with the **SQLXML** Java data type.

The code examples in this section use a Simple API for XML (SAX) parser. The SAX is a publicly developed standard for the events-based parsing of XML documents. It also provides an application programming interface for working with XML data. Note that the applications can use any other XML parser as well, such as the Document Object Model (DOM) or the Streaming API for XML (StAX), or so on.

The Document Object Model (DOM) provides a programmatic representation of XML documents, fragments, nodes, or node-sets. It also provides an application programming interface for working with XML data. Similarly, the Streaming API for XML (StAX) is a Java-based API for pull-parsing XML.

IMPORTANT

In order to use the SAX parser API, you must import the standard SAX implementation from the javax.xml package.

The code file for this sample is named SqlXmlDataType.java, and it can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\datatypes
```

Requirements

To run this sample application, you must set the classpath to include the sqljdbc4.jar file. If the classpath is missing an entry for sqljdbc4.jar, the sample application throws the "Class not found" exception. For more information about how to set the classpath, see Using the JDBC Driver.

In addition, you need access to the AdventureWorks sample database to run this sample application.

Example

In the following example, the sample code makes a connection to the AdventureWorks database and then calls the createSampleTables method.

The createSampleTables method drops the test tables, TestTable1, and TestTable2, if they exist. Then, it inserts two rows into TestTable1.

In addition, the code sample includes the following three methods and one additional class, which is named ExampleContentHandler.

The ExampleContentHandler class implements a custom content handler, which defines methods for parser events.

The showGetters method demonstrates how to parse the data in the SQLXML object by using the SAX, ContentHandler, and XMLReader. First, the code sample creates an instance of a custom content handler, which is ExampleContentHandler. Next, it creates and executes an SQL statement that returns a set of data from TestTable1. Then, the code example gets a SAX parser and parses the XML data.

The showSetters method demonstrates how to set the **xml** column by using the SAX, ContentHandler, and ResultSet. First, it creates an empty SQLXML object by using the createSQLXML method of the Connection class. Then, it gets an instance of a content handler to write the data into the SQLXML object. Next, the code example writes the data to TestTable1. Finally, the sample code iterates through the rows of data that are in the result set, and uses the getSQLXML method to read the XML data.

The showTransformer method demonstrates how to get an XML data from one table and insert that XML data into another table by using the SAX and the Transformer. First, it retrieves the source SQLXML object from the TestTable1. Then, it creates an empty destination SQLXML object by using the createSQLXML method of the Connection class. Next, it updates the destination SQLXML object and writes the XML data to TestTable2. Finally, the sample code iterates through the rows of data that are in the result set, and uses the getSQLXML method to read the XML data in TestTable2.

```
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.SQLXML;
import java.sql.Statement;

import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.sax.SAXResult;
import javax.xml.transform.sax.SAXSource;
import javax.xml.transform.sax.SAXTransformerFactory;

import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;


public class SqlXmlDataType {

    public static void main(String[] args) {

  // Create a variable for the connection string.
  String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=<database>;username=<user>;password=
<password>;";

  // Establish the connection.
  try (Connection con = DriverManager.getConnection(connectionUrl);
    Statement stmt = con.createStatement(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE)) {

   // Create initial sample data.
   createSampleTables(stmt);

   // The showGetters method demonstrates how to parse the data in the
   // SQLXML object by using the SAX, ContentHandler and XMLReader.
   showGetters(stmt);

   // The showSetters method demonstrates how to set the xml column
   // by using the SAX, ContentHandler, and ResultSet.
   showSetters(con, stmt);

   // The showTransformer method demonstrates how to get an XML data
   // from one table and insert that XML data to another table
   // by using the SAX and the Transformer.
   showTransformer(con, stmt);
  }
        // Handle any errors that may have occurred.
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void showGetters(Statement stmt) throws IOException, SAXException, SQLException {

        // Create an instance of the custom content handler.
        ExampleContentHandler myHandler = new ExampleContentHandler();

        // Create and execute an SQL statement that returns a
        // set of data.
        String SQL = "SELECT * FROM TestTable1";

        try (ResultSet rs = stmt.executeQuery(SQL)) {

            rs.next();

            SQLXML xmlSource = rs.getSQLXML("Col3");

            // Send SAX events to the custom content handler.
            SAXSource sxSource = xmlSource.getSource(SAXSource.class);
            XMLReader xmlReader = sxSource.getXMLReader();
            xmlReader.setContentHandler(myHandler);

            System.out.println("showGetters method: Parse an XML data in TestTable1 => ");
            xmlReader.parse(sxSource.getInputSource());
        }
    }

    private static void showSetters(Connection con, Statement stmt) {

        // Create and execute an SQL statement, retrieving an updatable result set.
        String SQL = "SELECT * FROM TestTable1;";
        try (ResultSet rs = stmt.executeQuery(SQL)) {

            // Create an empty SQLXML object.
            SQLXML sqlxml = con.createSQLXML();

            // Set the result value from SAX events.
            SAXResult sxResult = sqlxml.setResult(SAXResult.class);
            ContentHandler myHandler = sxResult.getHandler();

            // Set the XML elements and attributes into the result.
            myHandler.startDocument();
            myHandler.startElement(null, "contact", "contact", null);
            myHandler.startElement(null, "name", "name", null);
            myHandler.endElement(null, "name", "name");
            myHandler.startElement(null, "phone", "phone", null);
            myHandler.endElement(null, "phone", "phone");
            myHandler.endElement(null, "contact", "contact");
            myHandler.endDocument();

            // Update the data in the result set.
            rs.moveToInsertRow();
            rs.updateString("Col2", "C");
            rs.updateSQLXML("Col3", sqlxml);
            rs.insertRow();

            // Display the data.
            System.out.println("showSetters method: Display data in TestTable1 => ");
            while (rs.next()) {
                System.out.println(rs.getString("Col1") + " : " + rs.getString("Col2"));
                SQLXML xml = rs.getSQLXML("Col3");
                System.out.println("XML column : " + xml.getString());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    }

    private static void showTransformer(Connection con, Statement stmt) throws Exception {

        // Create and execute an SQL statement that returns a
        // set of data.
        String SQL = "SELECT * FROM TestTable1";
        try (ResultSet rs = stmt.executeQuery(SQL)) {

            rs.next();

            // Get the value of the source SQLXML object from the database.
            SQLXML xmlSource = rs.getSQLXML("Col3");

            // Get a Source to read the XML data.
            SAXSource sxSource = xmlSource.getSource(SAXSource.class);

            // Create a destination SQLXML object without any data.
            SQLXML xmlDest = con.createSQLXML();

            // Get a Result to write the XML data.
            SAXResult sxResult = xmlDest.setResult(SAXResult.class);

            // Transform the Source to a Result by using the identity transform.
            SAXTransformerFactory stf = (SAXTransformerFactory) TransformerFactory.newInstance();
            Transformer identity = stf.newTransformer();
            identity.transform(sxSource, sxResult);
            // Insert the destination SQLXML object into the database.
            try (PreparedStatement psmt = con
                    .prepareStatement("INSERT INTO TestTable2" + " (Col2, Col3, Col4, Col5) VALUES (?, ?, ?,
?)")) {
                psmt.setString(1, "A");
                psmt.setString(2, "Test data");
                psmt.setInt(3, 123);
                psmt.setSQLXML(4, xmlDest);
                psmt.execute();
            }
        }
        // Execute the query and display the data.
        SQL = "SELECT * FROM TestTable2";
        try (ResultSet rs = stmt.executeQuery(SQL)) {

            System.out.println("showTransformer method : Display data in TestTable2 => ");
            while (rs.next()) {
                System.out.println(rs.getString("Col1") + " : " + rs.getString("Col2"));
                System.out.println(rs.getString("Col3") + " : " + rs.getInt("Col4"));

                SQLXML xml = rs.getSQLXML("Col5");
                System.out.println("XML column : " + xml.getString());
            }
        }
    }

    private static void createSampleTables(Statement stmt) throws SQLException {
        // Drop the tables.
        stmt.executeUpdate("if exists (select * from sys.objects where name = 'TestTable1')" + "drop table
TestTable1");

        stmt.executeUpdate("if exists (select * from sys.objects where name = 'TestTable2')" + "drop table
TestTable2");

        // Create empty tables.
        stmt.execute("CREATE TABLE TestTable1 (Col1 int IDENTITY, Col2 char, Col3 xml)");
        stmt.execute("CREATE TABLE TestTable2 (Col1 int IDENTITY, Col2 char, Col3 varchar(50), Col4 int, Col5
xml)");

        // Insert two rows to the TestTable1.
        String row1 = "<contact><name>Contact Name 1</name><phone>XXX-XXX-XXXX</phone></contact>";
        String row2 = "<contact><name>Contact Name 2</name><phone>YYY-YYY-YYYY</phone></contact>";
        stmt.executeUpdate("insert into TestTable1" + " (Col2, Col3) values('A', '" + row1 + "')");
        stmt.executeUpdate("insert into TestTable1" + " (Col2, Col3) values('B', '" + row2 + "')");
    }
}

/**
 * Handles output for XML elements for the test.
 */
class ExampleContentHandler implements ContentHandler {

    public void startElement(String namespaceURI, String localName, String qName, Attributes atts) throws
SAXException {
        System.out.println("startElement method: localName => " + localName);
    }

    public void characters(char[] text, int start, int length) throws SAXException {
        System.out.println("characters method");
    }

    public void endElement(String namespaceURI, String localName, String qName) throws SAXException {
        System.out.println("endElement method: localName => " + localName);
    }

    public void setDocumentLocator(Locator locator) {
        System.out.println("setDocumentLocator method");
    }

    public void startDocument() throws SAXException {
        System.out.println("startDocument method");
    }

    public void endDocument() throws SAXException {
        System.out.println("endDocument method");
    }

    public void startPrefixMapping(String prefix, String uri) throws SAXException {
        System.out.println("startPrefixMapping method: prefix => " + prefix);
    }

    public void endPrefixMapping(String prefix) throws SAXException {
        System.out.println("endPrefixMapping method: prefix => " + prefix);
    }

    public void skippedEntity(String name) throws SAXException {
        System.out.println("skippedEntity method: name => " + name);
    }

    public void ignorableWhitespace(char[] text, int start, int length) throws SAXException {
        System.out.println("ignorableWhiteSpace method");
    }

    public void processingInstruction(String target, String data) throws SAXException {
        System.out.println("processingInstruction method: target => " + target);
    }
}
```

See Also

Working with Data Types (JDBC)

Spatial Data Types Sample

8/2/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to create, insert and retrieve Spatial Data types (Geometry and Geography).

The code file for this sample is named SpatialDataTypes.java, and it can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\datatypes
```

Requirements

To run this sample application, you must set the classpath to include the mssql-jdbc jar file. For more information about how to set the classpath, see Using the JDBC Driver.

NOTE

The Microsoft JDBC Driver for SQL Server provides mssql-jdbc class library files to be used depending on your preferred Java Runtime Environment (JRE) settings. For more information about which JAR file to choose, see System Requirements for the JDBC Driver.

Example

In the following example, the sample code creates a table called SpatialDataTypesTable_JDBC_Sample that contains 'Geometry' and 'Geography' columns.

The sample first creates 'Geometry' and 'Geography' objects from a Well-Known-Text (WKT) representing a POINT. It uses a SQLServerPreparedStatement with a parameterized query to map the data to each column accordingly.

Finally, the sample inserts the data into the table, and retrieves it. The data is displayed in the form of WKT.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import com.microsoft.sqlserver.jdbc.Geography;
import com.microsoft.sqlserver.jdbc.Geometry;
import com.microsoft.sqlserver.jdbc.SQLServerPreparedStatement;
import com.microsoft.sqlserver.jdbc.SQLServerResultSet;

public class SpatialDataTypes {

    private static String tableName = "SpatialDataTypesTable_JDBC_Sample";

    public static void main(String[] args) {

        // Create a variable for the connection string.
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=<database>;user=<user>;password=
<password>";
        // Establish the connection.
        try (Connection con = DriverManager.getConnection(connectionUrl); Statement stmt =
con.createStatement();) {
            dropAndCreateTable(stmt);

            // TODO: Implement Sample code
            String geoWKT = "POINT(3 40 5 6)";
            Geometry geomWKT = Geometry.STGeomFromText(geoWKT, 0);
            Geography geogWKT = Geography.STGeomFromText(geoWKT, 4326);

            try (SQLServerPreparedStatement pstmt = (SQLServerPreparedStatement) con
                    .prepareStatement("insert into " + tableName + " values (?, ?)");) {
                pstmt.setGeometry(1, geomWKT);
                pstmt.setGeography(2, geogWKT);
                pstmt.execute();

                SQLServerResultSet rs = (SQLServerResultSet) stmt.executeQuery("select * from " + tableName);
                rs.next();

                System.out.println("Geometry data: " + rs.getGeometry(1));
                System.out.println("Geography data: " + rs.getGeography(2));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void dropAndCreateTable(Statement stmt) throws SQLException {
        stmt.executeUpdate("if object_id('" + tableName + "','U') is not null" + " drop table " + tableName);

        stmt.executeUpdate("Create table " + tableName + " (c1 geometry, c2 geography)");
    }
}
```

See Also

Working with JDBC Data Types

Working with Result Sets

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When you work with the data contained in a SQL Server database, one method of manipulating the data is to use a result set. The Microsoft JDBC Driver for SQL Server supports the use of result sets through the SQLServerResultSet object. By using the SQLServerResultSet object, you can retrieve the data returned from an SQL statement or stored procedure, update the data as needed, and then persist that data back to the database.

In addition, the SQLServerResultSet object provides methods for navigating through its rows of data, getting or setting the data that it contains, and for establishing various levels of sensitivity to changes in the underlying database.

NOTE

For more information about managing result sets, including their sensitivity to changes, see Managing Result Sets with the JDBC Driver.

The topics in this section describe different ways that you can use a result set to manipulate the data contained in a SQL Server database.

In This Section

TOPIC	DESCRIPTION
Retrieving Result Set Data Sample	Describes how to use a result set to retrieve data from a SQL Server database and display it.
Modifying Result Set Data Sample	Describes how to use a result set to insert, retrieve, and modify data in a SQL Server database.
Caching Result Set Data Sample	Describes how to use a result set to retrieve large amounts of data from a SQL Server database, and to control how that data is cached on the client.

See Also

Sample JDBC Driver Applications

Retrieving Result Set Data Sample

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to retrieve a set of data from a SQL Server database, and then display that data.

The code file for this sample is named RetrieveResultSet.java, and it can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\resultsets
```

Requirements

To run this sample application, you must set the classpath to include the mssql-jdbc jar file. You'll also need access to the AdventureWorks sample database. For more information about how to set the classpath, see Using the JDBC Driver.

NOTE

The Microsoft JDBC Driver for SQL Server provides mssql-jdbc class library files to be used depending on your preferred Java Runtime Environment (JRE) settings. For more information about which JAR file to choose, see System Requirements for the JDBC Driver.

Example

In the following example, the sample code makes a connection to the AdventureWorks sample database. Then, using an SQL statement with the SQLServerStatement object, it runs the SQL statement and places the data that it returns into a SQLServerResultSet object.

Next, the sample code calls the custom displayRow method to iterate through the rows of data that are in the result set, and uses the getString method to display some of the data.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class RetrieveResultSet {

    public static void main(String[] args) {

        // Create a variable for the connection string.
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";

        try (Connection con = DriverManager.getConnection(connectionUrl); Statement stmt =
con.createStatement();) {
            createTable(stmt);
            String SQL = "SELECT * FROM Production.Product;";
            ResultSet rs = stmt.executeQuery(SQL);
            displayRow("PRODUCTS", rs);
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void displayRow(String title,
            ResultSet rs) throws SQLException {
        System.out.println(title);
        while (rs.next()) {
            System.out.println(rs.getString("ProductNumber") + " : " + rs.getString("Name"));
        }
    }

    private static void createTable(Statement stmt) throws SQLException {
        stmt.execute("if exists (select * from sys.objects where name = 'Product_JDBC_Sample')"
                + "drop table Product_JDBC_Sample");

        String sql = "CREATE TABLE [Product_JDBC_Sample](" + "[ProductID] [int] IDENTITY(1,1) NOT NULL,"
                + "[Name] [varchar](30) NOT NULL," + "[ProductNumber] [nvarchar](25) NOT NULL,"
                + "[MakeFlag] [bit] NOT NULL," + "[FinishedGoodsFlag] [bit] NOT NULL," + "[Color] [nvarchar]
(15) NULL,"
                + "[SafetyStockLevel] [smallint] NOT NULL," + "[ReorderPoint] [smallint] NOT NULL,"
                + "[StandardCost] [money] NOT NULL," + "[ListPrice] [money] NOT NULL," + "[Size] [nvarchar](5)
NULL,"
                + "[SizeUnitMeasureCode] [nchar](3) NULL," + "[WeightUnitMeasureCode] [nchar](3) NULL,"
                + "[Weight] [decimal](8, 2) NULL," + "[DaysToManufacture] [int] NOT NULL,"
                + "[ProductLine] [nchar](2) NULL," + "[Class] [nchar](2) NULL," + "[Style] [nchar](2) NULL,"
                + "[ProductSubcategoryID] [int] NULL," + "[ProductModelID] [int] NULL,"
                + "[SellStartDate] [datetime] NOT NULL," + "[SellEndDate] [datetime] NULL,"
                + "[DiscontinuedDate] [datetime] NULL," + "[rowguid] [uniqueidentifier] ROWGUIDCOL  NOT NULL,"
                + "[ModifiedDate] [datetime] NOT NULL,)";

        stmt.execute(sql);

        sql = "INSERT Product_JDBC_Sample VALUES ('Adjustable Time','AR-
5381','0','0',NULL,'1000','750','0.00','0.00',NULL,NULL,NULL,NULL,'0',NULL,NULL,NULL,NULL,NULL,'2008-04-30
00:00:00.000',NULL,NULL,'694215B7-08F7-4C0D-ACB1-D734BA44C0C8','2014-02-08 10:01:36.827') ";
        stmt.execute(sql);

        sql = "INSERT Product_JDBC_Sample VALUES ('ML Bottom Bracket','BB-
8107','0','0',NULL,'1000','750','0.00','0.00',NULL,NULL,NULL,NULL,'0',NULL,NULL,NULL,NULL,NULL,'2008-04-30
00:00:00.000',NULL,NULL,'694215B7-08F7-4C0D-ACB1-D734BA44C0C8','2014-02-08 10:01:36.827') ";
        stmt.execute(sql);

        sql = "INSERT Product_JDBC_Sample VALUES ('Mountain-500 Black, 44','BK-M18B-
44','0','0',NULL,'1000','750','0.00','0.00',NULL,NULL,NULL,NULL,'0',NULL,NULL,NULL,NULL,NULL,'2008-04-30
00:00:00.000',NULL,NULL,'694215B7-08F7-4C0D-ACB1-D734BA44C0C8','2014-02-08 10:01:36.827') ";
        stmt.execute(sql);
    }
}
```

See Also

Working with Result Sets

Modifying Result Set Data Sample

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to retrieve an updatable set of data from a SQL Server database. Then, using methods of the SQLServerResultSet object, it inserts, modifies, and then finally deletes a row of data from the set of data.

The code file for this sample is named UpdateResultSet.java, and it can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\resultsets
```

Requirements

To run this sample application, you must set the classpath to include the mssql-jdbc jar file. You'll also need access to the AdventureWorks sample database. For more information about how to set the classpath, see Using the JDBC Driver.

> **NOTE**
> The Microsoft JDBC Driver for SQL Server provides mssql-jdbc class library files to be used depending on your preferred Java Runtime Environment (JRE) settings. For more information about which JAR file to choose, see System Requirements for the JDBC Driver.

Example

In the following example, the sample code makes a connection to the AdventureWorks sample database. Then, using an SQL statement with the SQLServerStatement object, it runs the SQL statement and places the data that it returns into an updatable SQLServerResultSet object.

Next, the sample code uses the moveToInsertRow method to move the result set cursor to the insert row, uses a series of updateString methods to insert data into the new row, and then calls the insertRow method to persist the new row of data back to the database.

After inserting the new row of data, the sample code uses an SQL statement to retrieve the previously inserted row, and then uses the combination of updateString and updateRow methods to update the row of data and again persist it back to the database.

Finally, the sample code retrieves the previously updated row of data and then deletes it from the database using the deleteRow method.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class UpdateResultSet {

    public static void main(String[] args) {

        // Create a variable for the connection string.
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";

        try (Connection con = DriverManager.getConnection(connectionUrl);
                Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);) {

            // Create and execute an SQL statement, retrieving an updateable result set.
            String SQL = "SELECT * FROM HumanResources.Department;";
            ResultSet rs = stmt.executeQuery(SQL);

            // Insert a row of data.
            rs.moveToInsertRow();
            rs.updateString("Name", "Accounting");
            rs.updateString("GroupName", "Executive General and Administration");
            rs.updateString("ModifiedDate", "08/01/2006");
            rs.insertRow();

            // Retrieve the inserted row of data and display it.
            SQL = "SELECT * FROM HumanResources.Department WHERE Name = 'Accounting';";
            rs = stmt.executeQuery(SQL);
            displayRow("ADDED ROW", rs);

            // Update the row of data.
            rs.first();
            rs.updateString("GroupName", "Finance");
            rs.updateRow();

            // Retrieve the updated row of data and display it.
            rs = stmt.executeQuery(SQL);
            displayRow("UPDATED ROW", rs);

            // Delete the row of data.
            rs.first();
            rs.deleteRow();
            System.out.println("ROW DELETED");
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void displayRow(String title,
            ResultSet rs) throws SQLException {
        System.out.println(title);
        while (rs.next()) {
            System.out.println(rs.getString("Name") + " : " + rs.getString("GroupName"));
            System.out.println();
        }
    }
}
```

See Also

Caching Result Set Data Sample

8/13/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to retrieve a large set of data from a database, and then control the number of rows of data that are cached on the client by using the setFetchSize method of the SQLServerResultSet object.

NOTE

Limiting the number of rows cached on the client is different from limiting the total number of rows that a result set can contain. To control the total number of rows that are contained in a result set, use the setMaxRows method of the SQLServerStatement object, which is inherited by both the SQLServerPreparedStatement and SQLServerCallableStatement objects.

To set a limit on the number of rows cached on the client, you must first use a server-side cursor when you create one of the Statement objects by specifically stating the cursor type to use when creating the Statement object. For example, the JDBC driver provides the TYPE_SS_SERVER_CURSOR_FORWARD_ONLY cursor type, which is a fast forward-only, read-only server-side cursor for use with SQL Server databases.

NOTE

An alternative to using the SQL Server specific cursor type is to use the selectMethod connection string property, setting its value to "cursor". For more information about the cursor types supported by the JDBC driver, see Understanding Cursor Types.

After you have run the query contained in the Statement object and the data is returned to the client as a result set, you can call the setFetchSize method to control how much data is retrieved from the database at one time. For example, if you have a table that contains 100 rows of data, and you set the fetch size to 10, only 10 rows of data will be cached on the client at any point in time. Although this will slow down the speed at which the data is processed, it has the advantage of using less memory on the client, which can be especially useful when you need to process large amounts of data.

The code file for this sample is named CacheResultSet.java, and it can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\resultsets
```

Requirements

To run this sample application, you must set the classpath to include the mssql-jdbc jar file. You will also need access to the AdventureWorks sample database. For more information about how to set the classpath, see Using the JDBC Driver.

NOTE

The Microsoft JDBC Driver for SQL Server rovides mssql-jdbc class library files to be used depending on your preferred Java Runtime Environment (JRE) settings. For more information about which JAR file to choose, see System Requirements for the JDBC Driver.

Example

In the following example, the sample code makes a connection to the AdventureWorks sample database. Then it uses an SQL statement with the SQLServerStatement object, specifies the server-side cursor type, and then runs the SQL statement and places the data that it returns into a SQLServerResultSet object.

Next, the sample code calls the custom timerTest method, passing as arguments the fetch size to use and the result set. The timerTest method then sets the fetch size of the result set by using the setFetchSize method, sets the start time of the test, and then iterates through the result set with a `While` loop. As soon as the `While` loop is exited, the code sets the stop time of the test, and then displays the result of the test including the fetch size, the number of rows processed, and the time it took to execute the test.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;

import com.microsoft.sqlserver.jdbc.SQLServerResultSet;

public class CacheResultSet {

    @SuppressWarnings("serial")
    public static void main(String[] args) {

        // Create a variable for the connection string.
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";

        try (Connection con = DriverManager.getConnection(connectionUrl);
                Statement stmt = con.createStatement(SQLServerResultSet.TYPE_SS_SERVER_CURSOR_FORWARD_ONLY,
SQLServerResultSet.CONCUR_READ_ONLY);) {

            String SQL = "SELECT * FROM Sales.SalesOrderDetail;";

            for (int n : new ArrayList<Integer>() {
                {
                    add(1);
                    add(10);
                    add(100);
                    add(1000);
                    add(0);
                }
            }) {
                // Perform a fetch for every nth row in the result set.
                try (ResultSet rs = stmt.executeQuery(SQL)) {
                    timerTest(n, rs);
                }
            }
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }
    private static void timerTest(int fetchSize,
            ResultSet rs) throws SQLException {

        // Declare the variables for tracking the row count and elapsed time.
        int rowCount = 0;
        long startTime = 0;
        long stopTime = 0;
        long runTime = 0;

        // Set the fetch size then iterate through the result set to
        // cache the data locally.
        rs.setFetchSize(fetchSize);
        startTime = System.currentTimeMillis();
        while (rs.next()) {
            rowCount++;
        }
        stopTime = System.currentTimeMillis();
        runTime = stopTime - startTime;

        // Display the results of the timer test.
        System.out.println("FETCH SIZE: " + rs.getFetchSize());
        System.out.println("ROWS PROCESSED: " + rowCount);
        System.out.println("TIME TO EXECUTE: " + runTime);
        System.out.println();
    }
}
```

See Also

Working with Result Sets

Working with Large Data

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The JDBC driver provides support for adaptive buffering, which allows you to retrieve any kind of large-value data without the overhead of server cursors. With adaptive buffering, the Microsoft JDBC Driver for SQL Server retrieves statement execution results from the SQL Server as the application needs them, rather than all at once. The driver also discards the results as soon as the application can no longer access them.

In the Microsoft SQL Server 2005 (9.x) JDBC Driver version 1.2, the buffering mode was "**full**" by default. If your application did not set the "responseBuffering" connection property to "**adaptive**" either in the connection properties or by using the setResponseBuffering method of the SQLServerStatement object, the driver supported reading the entire result from the server at once. In order to get the adaptive buffering behavior, your application had to set the "responseBuffering" connection property to "**adaptive**" explicitly.

The **adaptive** value is the default buffering mode and the JDBC driver buffers the minimum possible data when necessary. For more information about using adaptive buffering, see Using Adaptive Buffering.

The topics in this section describe different ways that you can use to retrieve large-value data from a SQL Server database.

In This Section

TOPIC	DESCRIPTION
Reading Large Data Sample	Describes how to use a SQL statement to retrieve large-value data.
Reading Large Data with Stored Procedures Sample	Describes how to retrieve a large CallableStatement OUT parameter value.
Updating Large Data Sample	Describes how to update a large-value data in a database.

See Also

Sample JDBC Driver Applications

Reading Large Data Sample

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to retrieve a large single-column value from a SQL Server database by using the getCharacterStream method.

The code file for this sample is named ReadLargeData.java, and it can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\adaptive
```

Requirements

To run this sample application, you'll need access to the AdventureWorks sample database. You must also set the classpath to include the mssql-jdbc jar file. For more information about how to set the classpath, see Using the JDBC Driver.

NOTE

The Microsoft JDBC Driver for SQL Server provides mssql-jdbc class library files to be used depending on your preferred Java Runtime Environment (JRE) settings. For more information about which JAR file to choose, see System Requirements for the JDBC Driver.

Example

In the following example, the sample code makes a connection to the AdventureWorks database. Next, the sample code creates sample data and updates the Production.Document table by using a parameterized query.

In addition, the sample code demonstrates how to get the adaptive buffering mode by using the getResponseBuffering method of the SQLServerStatement class. Note that starting with the JDBC driver version 2.0 release, the responseBuffering connection property is set to "adaptive" by default.

Then, using an SQL statement with the SQLServerStatement object, the sample code runs the SQL statement and places the data that it returns into a SQLServerResultSet object.

Finally, the sample code iterates through the rows of data that are in the result set, and uses the getCharacterStream method to access some of the data.

```
import java.io.IOException;
import java.io.Reader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerStatement;

public class ReadLargeData {

    public static void main(String[] args) {
        // Create a variable for the connection string.
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";

        // Create test data as an example.
        StringBuffer buffer = new StringBuffer(4000);
        for (int i = 0; i < 4000; i++)
            buffer.append((char) ('A'));

        try (Connection con = DriverManager.getConnection(connectionUrl);
                Statement stmt = con.createStatement();
                PreparedStatement pstmt = con.prepareStatement("UPDATE Production.Document SET DocumentSummary
= ? WHERE (DocumentID = 1)");) {

            pstmt.setString(1, buffer.toString());
            pstmt.executeUpdate();

            // In adaptive mode, the application does not have to use a server cursor
            // to avoid OutOfMemoryError when the SELECT statement produces very large
            // results.

            // Create and execute an SQL statement that returns some data.
            String SQL = "SELECT Title, DocumentSummary FROM Production.Document";

            // Display the response buffering mode.
            SQLServerStatement SQLstmt = (SQLServerStatement) stmt;
            System.out.println("Response buffering mode is: " + SQLstmt.getResponseBuffering());
            SQLstmt.close();

            // Get the updated data from the database and display it.
            ResultSet rs = stmt.executeQuery(SQL);

            while (rs.next()) {
                Reader reader = rs.getCharacterStream(2);
                if (reader != null) {
                    char output[] = new char[40];
                    while (reader.read(output) != -1) {
                        // Do something with the chunk of the data that was
                        // read.
                    }

                    System.out.println(rs.getString(1) + " has been accessed for the summary column.");
                    // Close the stream.
                    reader.close();
                }
            }
        }
        // Handle any errors that may have occurred.
        catch (SQLException | IOException e) {
            e.printStackTrace();
        }
    }
}
```

See Also

Working with Large Data

Reading Large Data with Stored Procedures Sample

8/2/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to retrieve a large OUT parameter from a stored procedure.

The code file for this sample is named ExecuteStoredProcedure.java, and can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\adaptive
```

Requirements

To run this sample application, you'll need access to the AdventureWorks sample database. You must also set the classpath to include the mssql-jdbc jar file. For more information about how to set the classpath, see Using the JDBC Driver.

NOTE

The Microsoft JDBC Driver for SQL Server provides mssql-jdbc class library files to be used depending on your preferred Java Runtime Environment (JRE) settings. For more information about which JAR file to choose, see System Requirements for the JDBC Driver.

The sample would create the required stored procedure in the AdventureWorks sample database:

Example

In the following example, the sample code makes a connection to the AdventureWorks database. Next, the sample code creates sample data and updates the Production.Document table by using a parameterized query. Then, the sample code gets the adaptive buffering mode by using the getResponseBuffering method of the SQLServerStatement class and executes the GetLargeDataValue stored procedure. Starting with the JDBC driver version 2.0 release, the responseBuffering connection property is set to "adaptive" by default.

Finally, the sample code displays the data returned with the OUT parameters and also demonstrates how to use the `mark` and `reset` methods on the stream to re-read any portion of the data.

```
import java.io.Reader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerCallableStatement;


public class ExecuteStoredProcedures {

    public static void main(String[] args) {

        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";
```

```
    try (Connection con = DriverManager.getConnection(connectionUrl); Statement stmt = con.createStatement()) {

     createTable(stmt);
     createStoredProcedure(stmt);

     // Create test data as an example.
     StringBuffer buffer = new StringBuffer(4000);
     for (int i = 0; i < 4000; i++)
      buffer.append((char) ('A'));

     try (PreparedStatement pstmt = con.prepareStatement(
       "UPDATE Document_JDBC_Sample " + "SET DocumentSummary = ? WHERE (DocumentID = 1)")) {

      pstmt.setString(1, buffer.toString());
      pstmt.executeUpdate();
     }

     // Query test data by using a stored procedure.
     try (SQLServerCallableStatement cstmt = (SQLServerCallableStatement) con
       .prepareCall("{call GetLargeDataValue(?, ?, ?, ?)}")) {

      cstmt.setInt(1, 1);
      cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
      cstmt.registerOutParameter(3, java.sql.Types.CHAR);
      cstmt.registerOutParameter(4, java.sql.Types.LONGVARCHAR);

      // Display the response buffering mode.
      System.out.println("Response buffering mode is: " + cstmt.getResponseBuffering());

      cstmt.execute();
      System.out.println("DocumentID: " + cstmt.getInt(2));
      System.out.println("Document_Title: " + cstmt.getString(3));

      try (Reader reader = cstmt.getCharacterStream(4)) {

       // If your application needs to re-read any portion of the value,
       // it must call the mark method on the InputStream or Reader to
       // start buffering data that is to be re-read after a subsequent
       // call to the reset method.
       reader.mark(4000);

       // Read the first half of data.
       char output1[] = new char[2000];
       reader.read(output1);
       String stringOutput1 = new String(output1);

       // Reset the stream.
       reader.reset();

       // Read all the data.
       char output2[] = new char[4000];
       reader.read(output2);
       String stringOutput2 = new String(output2);

       System.out.println("Document_Summary in half: " + stringOutput1);
       System.out.println("Document_Summary: " + stringOutput2);
      }
     }
    }
          // Handle any errors that may have occurred.
          catch (Exception e) {
              e.printStackTrace();
          }
      }

      private static void createStoredProcedure(Statement stmt) throws SQLException {
          String outputProcedure = "GetLargeDataValue";
```

```
        String sql = " IF EXISTS (select * from sysobjects where id = object_id(N'" + outputProcedure
                + "') and OBJECTPROPERTY(id, N'IsProcedure') = 1)" + " DROP PROCEDURE " + outputProcedure;
        stmt.execute(sql);

        sql = "CREATE PROCEDURE " + outputProcedure + " @p0 int, @p1 int OUTPUT, @p2 char(50) OUTPUT, "
                + "@p3 varchar(max) OUTPUT " + " AS" + " SELECT top 1 @p1=DocumentID, @p2=Title,"
                + " @p3=DocumentSummary FROM Document_JDBC_Sample where DocumentID = @p0";

        stmt.execute(sql);
    }

    private static void createTable(Statement stmt) throws SQLException {
        stmt.execute("if exists (select * from sys.objects where name = 'Document_JDBC_Sample')"
                + "drop table Document_JDBC_Sample");

        String sql = "CREATE TABLE Document_JDBC_Sample(" + "[DocumentID] [int] NOT NULL identity,"
                + "[Title] [char](50) NOT NULL," + "[DocumentSummary] [varchar](max) NULL)";

        stmt.execute(sql);

        sql = "INSERT Document_JDBC_Sample VALUES ('title1','summary1') ";
        stmt.execute(sql);

        sql = "INSERT Document_JDBC_Sample VALUES ('title2','summary2') ";
        stmt.execute(sql);

        sql = "INSERT Document_JDBC_Sample VALUES ('title3','summary3') ";
        stmt.execute(sql);
    }
}
```

See Also

Working with Large Data

Updating Large Data Sample

8/2/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to update a large column in a database.

The code file for this sample is named UpdateLargeData.java, and can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\adaptive
```

Requirements

To run this sample application, you'll need access to the AdventureWorks sample database. You must also set the classpath to include the sqljdbc4.jar file. If the classpath is missing an entry for sqljdbc4.jar, the sample application will throw the common "Class not found" exception. For more information about how to set the classpath, see Using the JDBC Driver.

NOTE

The Microsoft JDBC Driver for SQL Server provides sqljdbc.jar, sqljdbc4.jar, sqljdbc41.jar, or sqljdbc42.jar class library files to be used depending on your preferred Java Runtime Environment (JRE) settings. This sample uses the isWrapperFor and unwrap methods, which are introduced in the JDBC 4.0 API, to access the driver-specific response buffering methods. In order to compile and run this sample, you will need sqljdbc4.jar class library, which provides support for JDBC 4.0. For more information about which JAR file to choose, see System Requirements for the JDBC Driver.

Example

In the following example, the sample code makes a connection to the AdventureWorks database. Then, the sample code creates a Statement object and uses the isWrapperFor method to check whether the Statement object is a wrapper for the specified SQLServerStatement class. The unwrap method is used to access the driver-specific response buffering methods.

Next, the sample code sets the response buffering mode as "**adaptive**" by using the setResponseBuffering method of the SQLServerStatement class and also demonstrates how to get the adaptive buffering mode.

Then, it runs the SQL statement, and places the data that it returns into an updateable SQLServerResultSet object.

Finally, the sample code iterates through the rows of data that are in the result set. If it finds an empty document summary, it uses the combination of updateString and updateRow methods to update the row of data and again persist it to the database. If there's already data, it uses the getString method to display some of the data.

The default behavior of the driver is "**adaptive.**" However, for the forward-only updatable result sets and when the data in the result set is larger than the application memory, the application has to set the adaptive buffering mode explicitly by using the setResponseBuffering method of the SQLServerStatement class.

```
import java.io.Reader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerStatement;


public class UpdateLargeData {

    public static void main(String[] args) {

     // Create a variable for the connection string.
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";

        // Establish the connection.
        try (Connection con = DriverManager.getConnection(connectionUrl); Statement stmt =
con.createStatement();
                Statement stmt1 = con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_UPDATABLE);) {

            createTable(stmt);

            // Since the summaries could be large, we should make sure that
            // the driver reads them incrementally from a database,
            // even though a server cursor is used for the updatable result sets.

            // The recommended way to access the Microsoft JDBC Driver for SQL Server
            // specific methods is to use the JDBC 4.0 Wrapper functionality.
            // The following code statement demonstrates how to use the
            // Statement.isWrapperFor and Statement.unwrap methods
            // to access the driver specific response buffering methods.

            if (stmt.isWrapperFor(com.microsoft.sqlserver.jdbc.SQLServerStatement.class)) {
                SQLServerStatement SQLstmt =
stmt.unwrap(com.microsoft.sqlserver.jdbc.SQLServerStatement.class);

                SQLstmt.setResponseBuffering("adaptive");
                System.out.println("Response buffering mode has been set to " +
SQLstmt.getResponseBuffering());
            }

            // Select all of the document summaries.
            try (ResultSet rs = stmt1.executeQuery("SELECT Title, DocumentSummary FROM Document_JDBC_Sample"))
{

                // Update each document summary.
                while (rs.next()) {

                    // Retrieve the original document summary.
                    try (Reader reader = rs.getCharacterStream("DocumentSummary")) {

                        if (reader == null) {
                            // Update the document summary.
                            System.out.println("Updating " + rs.getString("Title"));
                            rs.updateString("DocumentSummary", "Work in progress");
                            rs.updateRow();
                        }
                    }
                }
            }
        }
        // Handle any errors that may have occurred.
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void createTable(Statement stmt) throws SQLException {
        stmt.execute("if exists (select * from sys.objects where name = 'Document_JDBC_Sample')"
                + "drop table Document_JDBC_Sample");

        String sql = "CREATE TABLE Document_JDBC_Sample (" + "[DocumentID] [int] NOT NULL identity,"
                + "[Title] [char](50) NOT NULL," + "[DocumentSummary] [varchar](max) NULL)";

        stmt.execute(sql);

        sql = "INSERT Document_JDBC_Sample VALUES ('title1','summary1') ";
        stmt.execute(sql);

        sql = "INSERT Document_JDBC_Sample (title) VALUES ('title2') ";
        stmt.execute(sql);

        sql = "INSERT Document_JDBC_Sample (title) VALUES ('title3') ";
        stmt.execute(sql);

        sql = "INSERT Document_JDBC_Sample VALUES ('title4','summary3') ";
        stmt.execute(sql);
    }
}
```

See Also

Working with Large Data

SQL data discovery and classification

8/13/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

This Microsoft JDBC Driver for SQL Server sample application demonstrates how to use result set getter methods to retrieve SQL Server 'SQL data discovery and classification information' from the tables holding such information.

The code file for this sample is named DataDiscoveryAndClassification.java, and it can be found in the following location:

```
\<installation directory>\sqljdbc_<version>\<language>\samples\dataclassification
```

Requirements

To run this sample application, you must set the classpath to include the mssql-jdbc jar file. You'll also need access to the AdventureWorks sample database. For more information about how to set the classpath, see Using the JDBC Driver.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerResultSet;
import com.microsoft.sqlserver.jdbc.dataclassification.SensitivityProperty;


public class DataDiscoveryAndClassification {

    private static boolean featureSupported = false;

    public static void main(String[] args) {

        // Provides table name to be used for running test.
        String tableName = "JDBC_SQL_DATA_DISCOVERY_CLASSIFICATION";

        // Create a variable for the connection string.
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=<database>;username=
<user>;password=<password>;";

        // Establish the connection.
        try (Connection con = DriverManager.getConnection(connectionUrl); Statement stmt =
con.createStatement()) {
            verifySupportability(stmt);
            if (featureSupported) {
                createTable(stmt, tableName);
                runTests(stmt, tableName);
                drop_table(stmt, tableName);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
```

```
     * Verifies if SQL Discovery and Classification feature is applicable on target server.
     *
     * @param stmt
     *        Statement object to work with
     */
    private static void verifySupportability(Statement stmt) {
        try {
            stmt.execute("SELECT * FROM SYS.SENSITIVITY_CLASSIFICATIONS");
            featureSupported = true;
        } catch (SQLException e) {
            // Error Code 208 : Object Not Found
            if (e.getErrorCode() == 208) {
                featureSupported = false;
                System.err.println("This feature is not supported on the target SQL Server.");
            }
        }
    }

    /**
     * Creates table for the test and sets tags for Sensitivity Classification
     *
     * @param stmt
     *        Statement to work with
     * @param tableName
     *        Table to be created
     * @throws SQLException
     *         If an exception occurs
     */
    private static void createTable(Statement stmt, String tableName) throws SQLException {
        // Creates table for storing Supplier data
        stmt.execute("CREATE TABLE " + tableName + " (" + "[Id] [int] IDENTITY(1,1) NOT NULL,"
                + "[CompanyName] [nvarchar](40) NOT NULL," + "[ContactName] [nvarchar](50) NULL,"
                + "[ContactTitle] [nvarchar](40) NULL," + "[City] [nvarchar](40) NULL,"
                + "[Country] [nvarchar](40) NULL," + "[Phone] [nvarchar](30) MASKED WITH (FUNCTION =
'default()') NULL,"
                + "[Fax] [nvarchar](30) MASKED WITH (FUNCTION = 'default()') NULL," + "CONSTRAINT [PK_" +
tableName
                + "] PRIMARY KEY CLUSTERED" + "([Id] ASC "
                + ")WITH (STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]" + ") ON
[PRIMARY]");

        // Set Sensitivity Classification tags to table columns
        stmt.execute("ADD SENSITIVITY CLASSIFICATION TO " + tableName
                + ".CompanyName WITH (LABEL='PII', LABEL_ID='L1', INFORMATION_TYPE='Company name',
INFORMATION_TYPE_ID='COMPANY')");
        stmt.execute("ADD SENSITIVITY CLASSIFICATION TO " + tableName
                + ".ContactName WITH (LABEL='PII', LABEL_ID='L1', INFORMATION_TYPE='Person name',
INFORMATION_TYPE_ID='NAME')");
        stmt.execute("ADD SENSITIVITY CLASSIFICATION TO " + tableName
                + ".Phone WITH (LABEL='PII', LABEL_ID='L1', INFORMATION_TYPE='Contact Information',
INFORMATION_TYPE_ID='CONTACT')");
        stmt.execute("ADD SENSITIVITY CLASSIFICATION TO " + tableName
                + ".Fax WITH (LABEL='PII', LABEL_ID='L1', INFORMATION_TYPE='Contact Information',
INFORMATION_TYPE_ID='CONTACT')");
    }

    /**
     * Runs query to fetch ResultSet from target table
     *
     * @param stmt
     *        Statement to work with
     * @param tableName
     *        Name of table to fetch results from
     * @throws SQLException
     *         If an exception occurs
     */
    private static void runTests(Statement stmt, String tableName) throws SQLException {
        String query = "SELECT * FROM " + tableName;
        try (SQLServerResultSet rs = (SQLServerResultSet) stmt.executeQuery(query)) {
            printSensitivityClassification(rs);
        }
    }

    /**
     * Prints Sensitivity Classification data as received in ResultSet
     *
     * @param rs
     *        Active ResultSet to read data from
     * @throws SQLException
     *         If an exception occurs
     */
    private static void printSensitivityClassification(SQLServerResultSet rs) throws SQLException {
        if (null != rs.getSensitivityClassification()) {
            for (int columnPos = 0; columnPos <
rs.getSensitivityClassification().getColumnSensitivities().size();
                    columnPos++) {
                for (SensitivityProperty sp :
rs.getSensitivityClassification().getColumnSensitivities().get(columnPos)
                        .getSensitivityProperties()) {
                    if (sp.getLabel() != null) {
                        System.out.println("Labels received for Column : " + columnPos);
                        System.out.println("Label ID: " + sp.getLabel().getId());
                        System.out.println("Label Name: " + sp.getLabel().getName());
                        System.out.println();
                    }

                    if (sp.getInformationType() != null) {
                        System.out.println("Information Types received for Column : " + columnPos);
                        System.out.println("Information Type ID: " + sp.getInformationType().getId());
                        System.out.println("Information Type Name: " + sp.getInformationType().getName());
                        System.out.println();
                    }
                }
            }
        }
    }

    /**
     * Drops the table created for test
     *
     * @param stmt
     *        Statement to work with
     * @param tableName
     *        Table Name to be used
     * @throws SQLException
     *         If an exception occurs
     */
    private static void drop_table(Statement stmt, String tableName) throws SQLException {
        stmt.execute("DROP TABLE " + tableName);
    }
}
```

See Also

Sample JDBC Driver Applications

JDBC Specification Compliance

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

This page contains the list of JDBC compliance information for the JDBC Driver.

- JDBC 4.1 Compliance for the JDBC Driver
- JDBC 4.2 Compliance for the JDBC Driver
- JDBC 4.3 Compliance for the JDBC Driver

JDBC 4.1 Compliance for the JDBC Driver

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

NOTE

Versions prior to Microsoft JDBC Driver 4.2 for SQL Server are compliant for Java Database Connectivity API 4.0 specifications. This section does not apply for versions prior to the 4.2 release.

The Java Database Connectivity API 4.1 specification is supported by the Microsoft JDBC Driver 4.2 for SQL Server, with the following API methods.

SQLServerConnection class

NEW METHOD	DESCRIPTION	JDBC DRIVER IMPLEMENTATION
void abort(Executor executor)	Terminates an open connection to SQL Server.	Implemented as described in the java.sql.Connection interface. For more details see java.sql.Connection.
void setSchema(String schema)	Sets schema for the current connection.	SQL Server does not support setting schema for the current session. The driver silently logs a warning message if this method is called. For more details see java.sql.Connection.
String getSchema()	Returns the schema name for the current connection.	Since SQL Server does not support setting schema for the current connection, the driver instead returns the default schema of the user. For more details see java.sql.Connection.

SQLServerDatabaseMetaData class

NEW METHOD	DESCRIPTION	JDBC DRIVER IMPLEMENTATION
boolean generatedKeyAlwaysReturned()	Returns true as the driver supports retrieving generated keys	Implemented as described in the java.sql. DatabaseMetaData interface. For more details, see java.sql.DatabaseMetaData.
ResultSet getPseudoColumns(String catalog, String schemaPattern,String tableNamePattern,String columnNamePattern)	Retrieves a description of the pseudo/hidden columns	Return an empty result set as SQL Server does not have a formal notion of pseudo-columns. For more details, see java.sql.DatabaseMetaData.

SQLServerStatement class

NEW METHOD	DESCRIPTION	JDBC DRIVER IMPLEMENTATION

NEW METHOD	DESCRIPTION	JDBC DRIVER IMPLEMENTATION
void closeOnCompletion()	Specifies that this Statement will be closed when all its dependent result sets are closed.	Implemented as described in the java.sql.Statement interface. For more details, see java.sql.Statement.
boolean isCloseOnCompletion()	Returns a value indicating whether this Statement will be closed when all its dependent result sets are closed.	Implemented as described in the java.sql.Statement interface. For more details, see java.sql.Statement.

The Java Database Connectivity API 4.1 specification is supported by the Microsoft JDBC Driver 4.2 for SQL Server, with the following features.

NEW FEATURE	DESCRIPTION
New Escape Function Limited Return Rows Escape	Partially supported Escape syntax: LIMIT <rows> OFFSET <row_offset>.

The Java Database Connectivity API 4.1 specification is supported by the Microsoft JDBC Driver 4.2 for SQL Server, with the following Data Type Mappings.

DATA TYPE MAPPINGS	DESCRIPTION
New data type mappings are now supported in PreparedStatement.setObject() and PreparedStatement.setNull() methods.	1. New Java to JDBC type mapping (a) java.math.BigInteger to JDBC BIGINT (b) java.util.Date and java.util.Calendar to JDBC TIMESTAMP 2. New data type conversions: (a) java.math.BigInteger to CHAR, VARCHAR, LONGVARCHAR and BIGINT (b) java.util.Date and java.util.Calendar to CHAR, VARCHAR, LONGVARCHAR, DATE, TIME and TIMESTAMP For more details, see JDBC 4.1 specification.

JDBC 4.2 Compliance for the JDBC Driver

8/2/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

NOTE

Versions prior to Microsoft JDBC Driver 4.2 for SQL Server are compliant for Java Database Connectivity API 4.0 specifications. This section does not apply for versions prior to the 4.2 release.

The Java Database Connectivity API 4.2 specification is supported by the Microsoft JDBC Driver 4.2 for SQL Server, with the following API methods.

SQLServerStatement class

NEW METHODS	DESCRIPTION	NOTEWORTHY IMPLEMENTATION
long[] executeLargeBatch()	Executes batch where returned update counts can be long.	Implemented as described in the java.sql.Statement interface. For more details see java.sql.Statement.
long executeLargeUpdate(String sql) long executeLargeUpdate(String sql, int autoGeneratedKeys) long executeLargeUpdate(String sql, int[] columnIndexes) executeLargeUpdate(String sql, String[] columnNames)	Executes a DML/DDL statement where returned update counts can be long. There are 4 new (overloaded) methods to support long update count.	Implemented as described in the java.sql.Statement interface. For more details see java.sql.Statement.
long getLargeMaxRows()	Retrieves the maximum number of rows as a long value that the ResultSet can contain.	SQL Server only supports integer limits for max rows. For more details see java.sql.Statement.
long getLargeUpdateCount()	Retrieves the current result as a long update count.	SQL Server only supports integer limits for max rows. For more details see java.sql.Statement.
void setLargeMaxRows(long max)	Sets the maximum number of rows as a long value that the ResultSet can contain.	SQL Server only supports integer limits for max rows. This method throws a not supported exception if greater than max integer size is passed as the parameter. For more details see java.sql.Statement.

SQLServerCallableStatement class

NEW METHODS	DESCRIPTION	NOTEWORTHY IMPLEMENTATION
void registerOutParameter(int parameterIndex, SQLType sqlType) void registerOutParameter(int parameterIndex, SQLType sqlType, int scale) void registerOutParameter(int parameterIndex, SQLType sqlType, String typeName) void registerOutParameter(String parameterName, SQLType sqlType) void registerOutParameter(String parameterName, SQLType sqlType, int scale) registerOutParameter(String parameterName, SQLType sqlType, String typeName)	Registers the OUT parameter. There are 6 new (overloaded) methods to support the new SQLType interface.	Implemented as described in the java.sql.CallableStatement interface. For more details see java.sql.CallableStatement.
void setObject(String parameterName, Object x, SQLType targetSqlType) void setObject(String parameterName, Object x, SQLType targetSqlType, int scaleOrLength)	Sets the value of the parameter with the given object. There are 2 new (overloaded) methods to support the new SQLType interface	Implemented as described in the java.sql.CallableStatement interface. For more details see java.sql.CallableStatement.

SQLServerPreparedStatement class

NEW METHODS	DESCRIPTION	NOTEWORTHY IMPLEMENTATION
long executeLargeUpdate()	Execute DML/DDL statement and return long update count	Implemented as described in the java.sql.PreparedStatement interface. For more details see java.sql.PreparedStatement.
void setObject(int parameterIndex, Object x, SQLType targetSqlType) void setObject(int parameterIndex, Object x, SQLType targetSqlType, int scaleOrLength)	Sets the value of the parameter with the given object. There are 2 new (overloaded) methods to support the new SQLType interface.	Implemented as described in the java.sql.PreparedStatement interface. For more details see java.sql.PreparedStatement.

SQLServerDatabaseMetaData class

NEW METHODS	DESCRIPTION	NOTEWORTHY IMPLEMENTATION
long getMaxLogicalLobSize()	Retrieves the maximum number of bytes this database allows for the logical size for a LOB.	For SQL Server this value is 2^31-1. For more details see java.sql.DatabaseMetaData.
boolean supportsRefCursors()	Retrieves whether this database supports REF CURSOR.	Returns false as SQL Server does not support REF CURSOR. For more details see java.sql.DatabaseMetaData.

SQLServerResultSet class

New Methods	Description	Noteworthy Implementation
	Updates the designated column with an Object value. There are 4 new (overloaded) methods to support the new SQLType interface.	Implemented as described in the java.sql.ResultSet interface. For more details see java.sql.ResultSet.

The Java Database Connectivity API 4.2 specification is supported by the Microsoft JDBC Driver 4.2 for SQL Server, with the following Data Type Mappings.

New Data Type Mappings	Description
New Java classes in Java 8: LocalDate/LocalTime/LocalDateTime OffsetTime/OffsetDateTime **New JDBC types:** TIME_WITH_TIMEZONE TIMESTAMP_WITH_TIMEZONE REF_CURSOR	REF_CURSOR is not supported in SQL Server. Driver throws a SQLFeatureNotSupportedException exception if this type is used. The driver supports all other new Java and JDBC type mappings as specified in the JDBC 4.2 specification.

JDBC 4.3 Compliance for the JDBC Driver

8/2/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

NOTE

Versions prior to Microsoft JDBC Driver 6.4 for SQL Server are only compliant for Java Database Connectivity (JDBC) API 4.2 specifications. This section does not apply for versions including and prior to the 6.4 release.

As of version 6.4, Microsoft JDBC Driver for SQL Server is JAVA 9 compatible and throws `SQLFeatureNotSupportedException` for new JDBC 4.3 APIs that have unimplemented methods.

With Microsoft JDBC Driver 7.0 for SQL Server release, the driver is now JAVA 10 compatible, and supports below mentioned APIs. The driver throws `SQLFeatureNotSupportedException` for other unimplemented methods from JDBC 4.3 Specifications.

NEW API	DESCRIPTION	NOTEWORTHY IMPLEMENTATION
void java.sql.connection.beginRequest()	Hints to the driver that a request, an independent unit of work, is beginning on this connection. For more details, see java.sql.Connection.	Saves the values of the connection fields that are modifiable through public API methods: `databaseAutoCommitMode`, `transactionIsolationLevel`, `networkTimeout`, `holdability`, `sendTimeAsDatetime`, `statementPoolingCacheSize`, `disableStatementPooling`, `serverPreparedStatementDiscardThreshold`, `enablePrepareOnFirstPreparedStatementCall`, `catalogName`, `sqlWarnings`, `useBulkCopyForBatchInsert`.
void java.sql.connection.endRequest()	Hints to the driver that a request, an independent unit of work, has completed. For more details, see java.sql.Connection.	Closes the statements that are created during the work unit and rolls back any open transactions. The method also reverts the changes to the connection fields that are listed above.

Programming Guide for JDBC SQL Driver

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

- Connecting to the SQL Server with the JDBC Driver
- Understanding the JDBC Driver Data Types
- Using Statements with the JDBC Driver
- Managing Result Sets with the JDBC Driver
- Performing Transactions with the JDBC Driver
- Handling Metadata with the JDBC Driver
- Using Bulk Copy with the JDBC Driver
- Using Always Encrypted with the JDBC Driver
- Using Table-Valued Parameters
- International Features of the JDBC Driver
- JDBC Driver API Reference

Connecting to SQL Server with the JDBC Driver

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

One of the most fundamental things that you'll do with the Microsoft JDBC Driver for SQL Server is to make a connection to a SQL Server database. All interaction with the database occurs through the SQLServerConnection object, and because the JDBC driver has such a flat architecture, almost all interesting behavior touches the SQLServerConnection object.

If a SQL Server is only listening on an IPv6 port, set the java.net.preferIPv6Addresses system property to make sure that IPv6 is used instead of IPv4 to connect to the SQL Server:

```
System.setProperty("java.net.preferIPv6Addresses", "true");
```

The topics in this section describe how to make and work with a connection to a SQL Server database.

In This Section

TOPIC	DESCRIPTION
Building the Connection URL	Describes how to form a connection URL for connecting to a SQL Server database. Also describes connecting to named instances of a SQL Server database.
Setting the Connection Properties	Describes the various connection properties and how they can be used when you connect to a SQL Server database.
Setting the Data Source Properties	Describes how to use data sources in a Java Platform, Enterprise Edition (Java EE) environment.
Working with a Connection	Describes the various ways in which to create an instance of a connection to a SQL Server database.
Using Connection Pooling	Describes how the JDBC driver supports the use of connection pooling.
Using Database Mirroring (JDBC)	Describes how the JDBC driver supports the use of database mirroring.
JDBC Driver Support for High Availability, Disaster Recovery	Describes how to develop an application that will connect to an AlwaysOn availability group.
Using Kerberos Integrated Authentication to Connect to SQL Server	Discusses a Java implementation for applications to connect to a SQL Server database using Kerberos integrated authentication.
Connecting to an Azure SQL database	Discusses connectivity issues for databases on SQL Azure.

See Also

Overview of the JDBC Driver

Understanding the JDBC Driver Data Types

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Microsoft JDBC Driver for SQL Server supports the use of JDBC basic and advanced data types within a Java application that uses SQL Server as its database.

The JDBC type system mediates the conversion between SQL Server data types and Java language types and objects. The JDBC types are modeled on the SQL-92 and SQL-99 types. The JDBC driver adheres to the JDBC specification and is designed to provide the right balance between predictability and flexibility.

The topics in this section describe how to use the basic and advanced data types, and how data types can be converted into other data types.

In This Section

TOPIC	DESCRIPTION
Using Basic Data Types	Describes the JDBC basic data types. Includes examples of how to work with the data types by using result sets, parameterized queries, and stored procedures.
Configuring How java.sql.Time Values are Sent to the Server	Describes how the JDBC Driver generates dates.
Using Advanced Data Types	Describes the JDBC advanced data types.
Understanding Data Type Differences	Describes differences between the various JDBC driver data types.
Understanding Data Type Conversions	Describes how data type conversion is handled when using getter and setter methods.
National Character Set Support	Describes the support for the national character set types.
Supporting XML Data	Describes the SQLXML interface. Also describes how to read and write an XML data from and to the relational database with the **SQLXML** Java data type.
Wrappers and Interfaces	Discusses the interfaces that have the Microsoft JDBC Driver for SQL Server specific methods and constants that allow an application server to create a proxy of the class, Also discusses supports for the the `java.sql.Wrapper` interface.

See Also

Overview of the JDBC Driver

Using Statements with the JDBC Driver

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server can be used to work with data in a SQL Server database in a variety of ways. The JDBC driver can be used to run SQL statements against the database, or it can be used to call stored procedures in the database, using both input and output parameters. The JDBC driver also supports using SQL escape sequences, update counts, automatically generated keys, and performing updates within a batch operation.

The JDBC driver provides three classes for retrieving data from a SQL Server database:

1. SQLServerStatement - used for running SQL statements without parameters.
2. SQLServerPreparedStatement - (inherited from SQLServerStatement), used for running compiled SQL statements that might contain IN parameters.
3. SQLServerCallableStatement - (inherited from SQLServerPreparedStatement), used for running stored procedures that might contain IN parameters, OUT parameters, or both.

   The topics in this section discuss how you can use each of the three statement classes to work with data in a SQL Server database.

In This Section

TOPIC	DESCRIPTION
Using Statements with SQL	Describes how to use SQL statements with the JDBC driver to work with data in a SQL Server database.
Using Statements with Stored Procedures	Describes how to use stored procedures with the JDBC driver to work with data in a SQL Server database.
Using Multiple Result Sets	Describes how to use the JDBC driver to retrieve data from multiple result sets.
Using SQL Escape Sequences	Describes how to use SQL escape sequences, such as date and time literals and functions.
Using Auto Generated Keys	Describes how to use automatically generated keys.
Performing Batch Operations	Describes how to use the JDBC driver to perform batch operations.
Handling Complex Statements	Describes how to use the JDBC driver to run complex statements that perform a variety of tasks and might return different types of data.

See Also

Overview of the JDBC Driver

Managing Result Sets with the JDBC Driver

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The result set is an object that represents a set of data returned from a data source, usually as the result of a query. The result set contains rows and columns to hold the requested data elements, and it is navigated with a cursor. A result set can be updatable, meaning that it can be modified and have those modifications pushed to the original data source. A result set can also have various levels of sensitivity to changes in the underlying data source.

The type of result set is determined when a statement is created, which is when a call to the createStatement method of the SQLServerConnection class is made. The fundamental role of a result set is to provide Java applications with a usable representation of the database data. This is typically done with the typed getter and setter methods on the result set data elements.

In the following example, which is based on the AdventureWorks sample database, a result set is created by calling the executeQuery method of the SQLServerStatement class. Data from the result set is then displayed by using the getString method of the SQLServerResultSet class.

```
public static void executeStatement(Connection con){
    try(Statement stmt = con.createStatement();) {
        String SQL = "SELECT TOP 10 * FROM Person.Contact";
        ResultSet rs = stmt.executeQuery(SQL);

        while (rs.next()) {
            System.out.println(rs.getString("FirstName") + " " + rs.getString("LastName"));
        }
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

The topics in this section describe various aspects of result set usage, including cursor types, concurrency, and row locking.

In This Section

TOPIC	DESCRIPTION
Understanding Cursor Types	Describes the different cursor types that the Microsoft JDBC Driver for SQL Server supports.
Understanding Concurrency Control	Describes how the JDBC driver supports concurrency control.
Understanding Row Locking	Describes how the JDBC driver supports row locking.

See Also

Overview of the JDBC Driver

Performing Transactions with the JDBC Driver

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Transaction processing is a mandatory requirement of all applications that must guarantee consistency of their persistent data. With the Microsoft JDBC Driver for SQL Server, transaction processing can either be performed locally or distributed. Transactions are atomic, consistent, isolated, and durable (ACID) modules of execution.

The topics in this section describe how the JDBC driver supports transactions including isolation levels, transaction savepoints, and result set holdability.

In This Section

TOPIC	DESCRIPTION
Understanding Transactions	Provides an overview of how transactions are used with the JDBC driver.
Understanding XA Transactions	Provides an overview of how XA transactions are used with the JDBC driver.
Understanding Isolation Levels	Describes the various isolation levels that are supported by the JDBC driver.
Using Savepoints	Describes how to use the JDBC driver with transaction savepoints.
Using Holdability	Describes how to use the JDBC driver with result set holdability.

See Also

Overview of the JDBC Driver

Handling Metadata with the JDBC Driver

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server can be used to work with metadata in a SQL Server database in a variety of ways. The JDBC driver can be used to get metadata about the database, a result set, or parameters.

The JDBC driver provides three classes for retrieving metadata from a SQL Server database:

- SQLServerDatabaseMetaData, which is used to return information about the database that is currently connected.
- SQLServerResultSetMetaData, which is used to return information about the result set.
- SQLServerParameterMetaData, which is used to return information about the parameters of prepared and callable statements.

  The topics in this section describe how you can use each of the three metadata classes to work with metadata in a SQL Server database.

NOTE

The metadata methods discussed in this section are generally expensive in terms of application performance, so care should be taken with their usage.

In This Section

TOPIC	DESCRIPTION
Using Database Metadata	Describes how to retrieve metadata information about the currently connected database.
Using Result Set Metadata	Describes how to retrieve metadata information about the current result set.
Using Parameter Metadata	Describes how to retrieve metadata information about the parameters of prepared and callable statements.

See Also

Overview of the JDBC Driver

Using Always Encrypted with the JDBC driver

8/8/2018 • 33 minutes to read • Edit Online

Download JDBC Driver

This page provides information on how to develop Java applications using Always Encrypted and the Microsoft JDBC Driver 6.0 (or higher) for SQL Server.

Always Encrypted allows clients to encrypt sensitive data and never reveal the data or the encryption keys to SQL Server or Azure SQL Database. An Always Encrypted enabled driver, such as the Microsoft JDBC Driver 6.0 (or higher) for SQL Server, achieves this behavior by transparently encrypting and decrypting sensitive data in the client application. The driver automatically determines which query parameters correspond to Always Encrypted database columns, and encrypts the values of those parameters before it sends them to SQL Server or Azure SQL Database. Similarly, the driver transparently decrypts data retrieved from encrypted database columns in query results. For more information, see Always Encrypted (Database Engine) and Always Encrypted API Reference for the JDBC Driver.

Prerequisites

- Make sure Microsoft JDBC Driver 6.0 (or higher) for SQL Server is installed on your development machine.
- Download and install the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files. Be sure to read the Readme included in the zip file for installation instructions and relevant details on possible export/import issues.
  - If using the mssql-jdbc-X.X.X.jre7.jar or sqljdbc41.jar, the policy files can be downloaded from Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 7 Download
  - If using the mssql-jdbc-X.X.X.jre8.jar or sqljdbc42.jar, the policy files can be downloaded from Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 8 Download
  - If using the mssql-jdbc-X.X.X.jre9.jar, no policy file needs to be downloaded. The jurisdiction policy in Java 9 defaults to unlimited strength encryption.

Working with column master key stores

To encrypt or decrypt data for encrypted columns, SQL Server maintains column encryption keys. Column encryption keys are stored in encrypted form in the database metadata. Each column encryption key has a corresponding column master key that is used to encrypt the column encryption key. The database metadata doesn't contain the column master keys. Those keys are only held by the client. However the database metadata does contain information about where the column master keys are stored relative to the client. For example, the database metadata may say that the keystore holding a column master key is the Windows Certificate Store and the specific certificate used to encrypt and decrypt is located at a specific path within the Windows Certificate Store. If the client has access to that certificate in the Windows Certificate Store, it can obtain the certificate. The certificate can then be used to decrypt the column encryption key. Then that encryption key can be used to decrypt or encrypt data for encrypted columns that use that column encryption key.

The Microsoft JDBC Driver for SQL Server communicates with a keystore using a column master key store provider, which is an instance of a class derived from **SQLServerColumnEncryptionKeyStoreProvider**.

Using built-in column master key store providers

The Microsoft JDBC Driver for SQL Server comes with the following built-in column master key store providers.

Some of these providers are pre-registered with the specific provider names (used to look up the provider) and some require either additional credentials or explicit registration.

CLASS	DESCRIPTION	PROVIDER (LOOKUP) NAME	IS PRE-REGISTERED?
SQLServerColumnEncryptionAzureKeyVaultProvider	A provider for a keystore for the Azure Key Vault.	AZURE_KEY_VAULT	No
SQLServerColumnEncryptionCertificateStoreProvider	A provider for the Windows Certificate Store.	MSSQL_CERTIFICATE_STORE	Yes
SQLServerColumnEncryptionJavaKeyStoreProvider	A provider for the Java keystore	MSSQL_JAVA_KEYSTORE	Yes

For the pre-registered keystore providers, you don't need to make any application code changes to use these providers but note the following items:

- You (or your DBA) need to make sure the provider name configured in the column master key metadata is correct and the column master key path complies with the key path format that is valid for a given provider. It's recommended that you configure the keys using tools, such as SQL Server Management Studio, which automatically generates the valid provider names and key paths when issuing the CREATE COLUMN MASTER KEY (Transact-SQL) statement.
- Ensure your application can access the key in the keystore. This task may involve granting your application access to the key and/or the keystore, depending on the keystore, or performing other keystore-specific configuration steps. For example, for using the SQLServerColumnEncryptionJavaKeyStoreProvider, you need to provide the location and the password of the keystore in the connection properties.

All of these keystore providers are described in more detail in the sections that follow. You only need to implement one keystore provider to use Always Encrypted.

Using Azure Key Vault provider

Azure Key Vault is a convenient option to store and manage column master keys for Always Encrypted (especially if your application is hosted in Azure). The Microsoft JDBC Driver for SQL Server includes a built-in provider, SQLServerColumnEncryptionAzureKeyVaultProvider, for applications that have keys stored in Azure Key Vault. The name of this provider is AZURE_KEY_VAULT. In order to use the Azure Key Vault store provider, an application developer needs to create the vault and the keys in Azure Key Vault and create an App registration in Azure Active Directory. The registered application must be granted Get, Decrypt, Encrypt, Unwrap Key, Wrap Key, and Verify permissions in the Access policies defined for the key vault created for use with Always Encrypted. For more information on how to set up the key vault and create a column master key, see Azure Key Vault – Step by Step and Creating Column Master Keys in Azure Key Vault.

For the examples on this page, if you've created an Azure Key Vault based column master key and column encryption key by using SQL Server Management Studio, the T-SQL script to re-create them might look similar to this example with its own specific **KEY_PATH** and **ENCRYPTED_VALUE**:

```
CREATE COLUMN MASTER KEY [MyCMK]
WITH
(
    KEY_STORE_PROVIDER_NAME = N'AZURE_KEY_VAULT',
    KEY_PATH = N'https://<MyKeyValutName>.vault.azure.net:443/keys/Always-Encrypted-
Auto1/c61f01860f37302457fa512bb7e7f4e8'
)

CREATE COLUMN ENCRYPTION KEY [MyCEK]
WITH VALUES
(
    COLUMN_MASTER_KEY = [MyCMK],
    ALGORITHM = 'RSA_OAEP',
    ENCRYPTED_VALUE = 0x01BA000001680074507400700073003A002F002F006400610076006...
)
```

To use the Azure Key Vault, client applications need to instantiate the SQLServerColumnEncryptionAzureKeyVaultProvider and register it with the driver.

Here is an example of initializing SQLServerColumnEncryptionAzureKeyVaultProvider:

```
SQLServerColumnEncryptionAzureKeyVaultProvider akvProvider = new
SQLServerColumnEncryptionAzureKeyVaultProvider(clientID, clientKey);
```

clientID is the Application ID of an App registration in an Azure Active Directory instance. **clientKey** is a Key Password registered under that Application, which provides API access to the Azure Key Vault.

After the application creates an instance of SQLServerColumnEncryptionAzureKeyVaultProvider, the application must register the instance with the driver using the SQLServerConnection.registerColumnEncryptionKeyStoreProviders() method. It's highly recommended that the instance is registered using the default lookup name, AZURE_KEY_VAULT, which can be obtained by calling the SQLServerColumnEncryptionAzureKeyVaultProvider.getName() API. Using the default name will allow you to use tools such as SQL Server Management Studio or PowerShell to provision and manage Always Encrypted keys (the tools use the default name to generate the metadata object to column master key). The following example shows registering the Azure Key Vault provider. For more information on the SQLServerConnection.registerColumnEncryptionKeyStoreProviders() method, see Always Encrypted API Reference for the JDBC Driver.

```
Map<String, SQLServerColumnEncryptionKeyStoreProvider> keyStoreMap = new HashMap<String,
SQLServerColumnEncryptionKeyStoreProvider>();
keyStoreMap.put(akvProvider.getName(), akvProvider);
SQLServerConnection.registerColumnEncryptionKeyStoreProviders(keyStoreMap);
```

IMPORTANT

If you use the Azure Key Vault keystore provider, the Azure Key Vault implementation of the JDBC driver has dependencies on these libraries (from GitHub) which must be included with your application:

azure-sdk-for-java

azure-activedirectory-library-for-java libraries

For an example of how to include these dependencies in a Maven project, see Download ADAL4J And AKV Dependencies with Apache Maven

Using Windows Certificate Store provider

The SQLServerColumnEncryptionCertificateStoreProvider can be used to store column master keys in the

Windows Certificate Store. Use the SQL Server Management Studio (SSMS) Always Encrypted wizard or other supported tools to create the column master key and column encryption key definitions in the database. The same wizard can be used to generate a self signed certificate in the Windows Certificate Store that can be used as a column master key for the always encrypted data. For more information on column master key and column encryption key T-SQL syntax, see CREATE COLUMN MASTER KEY and CREATE COLUMN ENCRPTION KEY respectively.

The name of the SQLServerColumnEncryptionCertificateStoreProvider is MSSQL_CERTIFICATE_STORE and can be queried by the getName() API of the provider object. It's automatically registered by the driver and can be used seamlessly without any application change.

For the examples on this page, if you've created a Windows Certificate Store based column master key and column encryption key by using SQL Server Management Studio, the T-SQL script to re-create them might look similar to this example with its own specific **KEY_PATH** and **ENCRYPTED_VALUE**:

```
CREATE COLUMN MASTER KEY [MyCMK]
WITH
(
    KEY_STORE_PROVIDER_NAME = N'MSSQL_CERTIFICATE_STORE',
    KEY_PATH = N'CurrentUser/My/A2A91F59C461B559E4D962DA9D2BC6131B32CB91'
)

CREATE COLUMN ENCRYPTION KEY [MyCEK]
WITH VALUES
(
    COLUMN_MASTER_KEY = [MyCMK],
    ALGORITHM = 'RSA_OAEP',
    ENCRYPTED_VALUE = 0x016E00000163007500720072006500740075007300650072007200...
)
```

IMPORTANT

While the other keystore providers in this article are available on all platforms supported by the driver, the SQLServerColumnEncryptionCertificateStoreProvider implementation of the JDBC driver is available on Windows operating systems only. It has a dependency on the sqljdbc_auth.dll that is available in the driver package. To use this provider, copy the sqljdbc_auth.dll file to a directory on the Windows system path on the computer where the JDBC driver is installed. Alternatively you can set the java.libary.path system property to specify the directory of the sqljdbc_auth.dll. If you are running a 32-bit Java Virtual Machine (JVM), use the sqljdbc_auth.dll file in the x86 folder, even if the operating system is the x64 version. If you are running a 64-bit JVM on a x64 processor, use the sqljdbc_auth.dll file in the x64 folder. For example, if you are using the 32-bit JVM and the JDBC driver is installed in the default directory, you can specify the location of the DLL by using the following virtual machine (VM) argument when the Java application is started: `-Djava.library.path=C:\Microsoft JDBC Driver <version> for SQL Server\sqljdbc_<version>\enu\auth\x86`

Using Java Key Store provider

The JDBC driver comes with a built-in keystore provider implementation for the Java Key Store. If the **keyStoreAuthentication** connection string property is present in the connection string and it's set to "JavaKeyStorePassword", the driver automatically instantiates and registers the provider for Java Key Store. The name of the Java Key Store provider is MSSQL_JAVA_KEYSTORE. This name can also be queried by using the SQLServerColumnEncryptionJavaKeyStoreProvider.getName() API.

There are three connection string properties that allow a client application to specify the credentials the driver needs to authenticate to the Java Key Store. The driver initializes the provider based on the values of these three properties in the connection string.

keyStoreAuthentication: Identifies the Java Key Store to use. With Microsoft JDBC Driver 6.0 and higher for SQL Server, you can authenticate to the Java Key Store only through this property. For the Java Key Store, the value for this property must be `JavaKeyStorePassword`.

keyStoreLocation: The path to the Java Key Store file that stores the column master key. The path includes the keystore filename.

keyStoreSecret: The secret/password to use for the keystore as well as for the key. For using the Java Key Store, the keystore and the key password must be the same.

Here is an example of providing these credentials in the connection string:

```
String connectionUrl = "jdbc:sqlserver://<server>:<port>;user=<user>;password=
<password>;columnEncryptionSetting=Enabled;keyStoreAuthentication=JavaKeyStorePassword;keyStoreLocation=
<path_to_the_keystore_file>;keyStoreSecret=<keystore_key_password>";
```

You can also get or set these settings using the SQLServerDataSource object. For more information, see Always Encrypted API Reference for the JDBC Driver.

The JDBC driver automatically instantiates the SQLServerColumnEncryptionJavaKeyStoreProvider when these credentials are present in connection properties.

Creating a column master key for the Java Key Store

The SQLServerColumnEncryptionJavaKeyStoreProvider can be used with JKS or PKCS12 keystore types. To create or import a key to use with this provider use the Java keytool utility. The key must have the same password as the keystore itself. Here is an example of how to create a public key and its associated private key using the keytool utility:

```
keytool -genkeypair -keyalg RSA -alias AlwaysEncryptedKey -keystore keystore.jks -storepass mypassword -
validity 360 -keysize 2048 -storetype jks
```

This command creates a public key and wraps it in an X.509 self signed certificate, which is stored in the keystore 'keystore.jks' along with its associated private key. This entry in the keystore is identified by the alias 'AlwaysEncryptedKey'.

Here is an example of the same using a PKCS12 store type:

```
keytool -genkeypair -keyalg RSA -alias AlwaysEncryptedKey -keystore keystore.pfx -storepass mypassword -
validity 360 -keysize 2048 -storetype pkcs12 -keypass mypassword
```

If the keystore is of type PKCS12, the keytool utility doesn't prompt for a key password and the key password needs to be provided with -keypass option as the SQLServerColumnEncryptionJavaKeyStoreProvider requires that the keystore and the key have the same password.

You can also export a certificate from the Windows Certificate store in .pfx format and use that with the SQLServerColumnEncryptionJavaKeyStoreProvider. The exported certificate can also be imported to the Java Key Store as a JKS keystore type.

After creating the keytool entry, create the column master key metadata in the database, which needs the keystore provider name and the key path. For more information on how to create column master key meta data, see CREATE COLUMN MASTER KEY. For SQLServerColumnEncryptionJavaKeyStoreProvider, the key path is just the alias of the key and the name of the SQLServerColumnEncryptionJavaKeyStoreProvider is 'MSSQL_JAVA_KEYSTORE'. You can also query this name using the getName() public API of the SQLServerColumnEncryptionJavaKeyStoreProvider class.

The T-SQL syntax for creating the column master key is:

```
CREATE COLUMN MASTER KEY [<CMK_name>]
WITH
(
    KEY_STORE_PROVIDER_NAME = N'MSSQL_JAVA_KEYSTORE',
    KEY_PATH = N'<key_alias>'
)
```

For the 'AlwaysEncryptedKey' created above, the column master key definition would be:

```
CREATE COLUMN MASTER KEY [MyCMK]
WITH
(
    KEY_STORE_PROVIDER_NAME = N'MSSQL_JAVA_KEYSTORE',
    KEY_PATH = N'AlwaysEncryptedKey'
)
```

NOTE

The built-in SQL Server management Studio functionality cannot create column master key definitions for the Java Key Store. T-SQL commands must be used programmatically.

Creating a column encryption key for the Java Key Store

The SQL Server Management Studio or any other tool can't be used to create column encryption keys using column master keys in the Java Key Store. The client application must create the column encryption key programmatically using the SQLServerColumnEncryptionJavaKeyStoreProvider class. For more information, see Using column master key store providers for programmatic key provisioning.

Implementing a custom column master key store provider

If you want to store column master keys in a keystore that is not supported by an existing provider, you can implement a custom provider by extending the SQLServerColumnEncryptionKeyStoreProvider Class and registering the provider using the SQLServerConnection.registerColumnEncryptionKeyStoreProviders() method.

```
public class MyCustomKeyStore extends SQLServerColumnEncryptionKeyStoreProvider{
    private String name = "MY_CUSTOM_KEYSTORE";

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

    public String getName()
    {
        return name;
    }

    public byte[] encryptColumnEncryptionKey(String masterKeyPath, String encryptionAlgorithm, byte[]
plainTextColumnEncryptionKey)
    {
        // Logic for encrypting the column encryption key
    }

    public byte[] decryptColumnEncryptionKey(String masterKeyPath, String encryptionAlgorithm, byte[]
encryptedColumnEncryptionKey)
    {
        // Logic for decrypting the column encryption key
    }
}
```

Register the provider:

```
SQLServerColumnEncryptionKeyStoreProvider storeProvider = new MyCustomKeyStore();
Map<String, SQLServerColumnEncryptionKeyStoreProvider> keyStoreMap = new HashMap<String,
SQLServerColumnEncryptionKeyStoreProvider>();
keyStoreMap.put(storeProvider.getName(), storeProvider);
SQLServerConnection.registerColumnEncryptionKeyStoreProviders(keyStoreMap);
```

Using column master key store providers for programmatic key provisioning

When accessing encrypted columns, the Microsoft JDBC Driver for SQL Server transparently finds and calls the right column master key store provider to decrypt column encryption keys. Typically, your normal application code doesn't directly call column master key store providers. You may, however, instantiate and call a provider programmatically to provision and manage Always Encrypted keys. This step may be done to generate an encrypted column encryption key and decrypt a column encryption key as part column master key rotation, for example. For more information, see Overview of Key Management for Always Encrypted.

If you use a custom keystore provider, implementing your own key management tools may be required. When using keys stored in Windows Certificate Store or in Azure Key Vault, you can use existing tools, such as SQL Server Management Studio or PowerShell, to manage and provision keys. When using keys stored in the Java Key Store, you need to provision keys programmatically. The following example illustrates using the SQLServerColumnEncryptionJavaKeyStoreProvider class to encrypt the key with a key stored in the Java Key Store.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerColumnEncryptionJavaKeyStoreProvider;
import com.microsoft.sqlserver.jdbc.SQLServerColumnEncryptionKeyStoreProvider;
import com.microsoft.sqlserver.jdbc.SQLServerException;

/**
 * This program demonstrates how to create a column encryption key programmatically for the Java Key Store.
 */
public class AlwaysEncrypted {
    // Alias of the key stored in the keystore.
    private static String keyAlias = "<proide key alias>";

    // Name by which the column master key will be known in the database.
    private static String columnMasterKeyName = "MyCMK";

    // Name by which the column encryption key will be known in the database.
    private static String columnEncryptionKey = "MyCEK";

    // The location of the keystore.
    private static String keyStoreLocation = "C:\\Dev\\Always Encrypted\\keystore.jks";

    // The password of the keystore and the key.
    private static char[] keyStoreSecret = "********".toCharArray();

    /**
     * Name of the encryption algorithm used to encrypt the value of the column encryption key. The algorithm
for the system providers must be
     * RSA_OAEP.
     */
    private static String algorithm = "RSA_OAEP";

    public static void main(String[] args) {
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=<databaseName>;user=
<user>;password=<password>;columnEncryptionSetting=Enabled;";

        try (Connection connection = DriverManager.getConnection(connectionUrl);
                Statement statement = connection.createStatement();) {

            // Instantiate the Java Key Store provider.
            SQLServerColumnEncryptionKeyStoreProvider storeProvider = new
SQLServerColumnEncryptionJavaKeyStoreProvider(keyStoreLocation,
                    keyStoreSecret);

            byte[] encryptedCEK = getEncryptedCEK(storeProvider);

            /**
             * Create column encryption key For more details on the syntax, see:
             * https://docs.microsoft.com/sql/t-sql/statements/create-column-encryption-key-transact-sql
Encrypted column encryption key first needs
             * to be converted into varbinary_literal from bytes, for which byteArrayToHex() is used.
             */
            String createCEKSQL = "CREATE COLUMN ENCRYPTION KEY "
                    + columnEncryptionKey
                    + " WITH VALUES ( "
                    + " COLUMN_MASTER_KEY = "
                    + columnMasterKeyName
                    + " , ALGORITHM =  '"
                    + algorithm
                    + "' , ENCRYPTED_VALUE =  0x"
                    + byteArrayToHex(encryptedCEK)
                    + " ) ";
            statement.executeUpdate(createCEKSQL);
            System.out.println("Column encryption key created with name : " + columnEncryptionKey);
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static byte[] getEncryptedCEK(SQLServerColumnEncryptionKeyStoreProvider storeProvider) throws
SQLServerException {
        String plainTextKey = "You need to give your plain text";

        // plainTextKey has to be 32 bytes with current algorithm supported
        byte[] plainCEK = plainTextKey.getBytes();

        // This will give us encrypted column encryption key in bytes
        byte[] encryptedCEK = storeProvider.encryptColumnEncryptionKey(keyAlias, algorithm, plainCEK);

        return encryptedCEK;
    }

    public static String byteArrayToHex(byte[] a) {
        StringBuilder sb = new StringBuilder(a.length * 2);
        for (byte b : a)
            sb.append(String.format("%02x", b).toUpperCase());
        return sb.toString();
    }
}
```

Enabling Always Encrypted for application queries

The easiest way to enable the encryption of parameters and the decryption of query results that target encrypted columns is by setting the value of the **columnEncryptionSetting** connection string keyword to **Enabled**.

The following connection string is an example of enabling Always Encrypted in the JDBC driver:

```
String connectionUrl = "jdbc:sqlserver://<server>:<port>;user=<user>;password=<password>;databaseName=
<database>;columnEncryptionSetting=Enabled;";
SQLServerConnection connection = (SQLServerConnection) DriverManager.getConnection(connectionUrl);
```

The following code is an equivalent example using the SQLServerDataSource object:

```
SQLServerDataSource ds = new SQLServerDataSource();
ds.setServerName("<server>");
ds.setPortNumber(<port>);
ds.setUser("<user>");
ds.setPassword("<password>");
ds.setDatabaseName("<database>");
ds.setColumnEncryptionSetting("Enabled");
SQLServerConnection con = (SQLServerConnection) ds.getConnection();
```

Always Encrypted can also be enabled for individual queries. For more information, see Controlling the performance impact of Always Encrypted. Enabling Always Encrypted isn't sufficient for encryption or decryption to succeed. You also need to make sure:

- The application has the *VIEW ANY COLUMN MASTER KEY DEFINITION* and *VIEW ANY COLUMN ENCRYPTION KEY DEFINITION* database permissions, required to access the metadata about Always Encrypted keys in the database. For details, see Permissions in Always Encrypted (Database Engine).
- The application can access the column master key that protects the column encryption keys, which encrypt the queried database columns. To use the Java Key Store provider, you need to provide additional credentials in the connection string. For more information, see Using Java Key Store provider.

Configuring how java.sql.Time values are sent to the server

The **sendTimeAsDatetime** connection property is used to configure how the java.sql.Time value is sent to the server. When set to false, the time value is sent as a SQL Server time type. When set to true, the time value is sent as a datetime type. If a time column is encrypted, the **sendTimeAsDatetime** property must be false, as encrypted columns don't support the conversion from time to datetime. Also note that this property is by default true, so when using encrypted time columns you'll have to set it to false. Otherwise, the driver will throw an exception. Starting with version 6.0 of the driver, the SQLServerConnection class has two methods to configure the value of this property programmatically:

- public void setSendTimeAsDatetime(boolean sendTimeAsDateTimeValue)
- public boolean getSendTimeAsDatetime()

For more information on this property, see Configuring How java.sql.Time Values are Sent to the Server.

Configuring how String values are sent to the server

The **sendStringParametersAsUnicode** connection property is used to configure how String values are sent to SQL Server. If set to true, String parameters are sent to the server in Unicode format. If set to false, String parameters are sent in non-Unicode format, such as ASCII or MBCS, instead of Unicode. The default value for this property is true. When Always Encrypted is enabled and a char/varchar/varchar(max) column is encrypted, the value of **sendStringParametersAsUnicode** must be set to false. If this property is set to true, the driver will throw an exception when decrypting data from an encrypted char/varchar/varchar(max) column that has Unicode characters. For more information on this property, see Setting the Connection Properties.

Retrieving and modifying data in encrypted columns

Once you enable Always Encrypted for application queries, you can use standard JDBC APIs to retrieve or modify data in encrypted database columns. If your application has the required database permissions and can access the column master key, the driver will encrypt any query parameters that target encrypted columns and will decrypt data that is retrieved from encrypted columns.

If Always Encrypted isn't enabled, queries with parameters that target encrypted columns will fail. Queries can still retrieve data from encrypted columns as long as the query has no parameters targeting encrypted columns. However, the driver won't attempt to decrypt any values retrieved from encrypted columns and the application will receive binary encrypted data (as byte arrays).

The following table summarizes the behavior of queries depending on whether Always Encrypted is enabled or not:

QUERY CHARACTERISTIC	ALWAYS ENCRYPTED IS ENABLED AND APPLICATION CAN ACCESS THE KEYS AND KEY METADATA	ALWAYS ENCRYPTED IS ENABLED AND APPLICATION CAN'T ACCESS THE KEYS OR KEY METADATA	ALWAYS ENCRYPTED IS DISABLED
Queries with parameters targeting encrypted columns.	Parameter values are transparently encrypted.	Error	Error
Queries retrieving data from encrypted columns without parameters targeting encrypted columns.	Results from encrypted columns are transparently decrypted. The application receives plaintext values of the JDBC datatypes corresponding to the SQL Server types configured for the encrypted columns.	Error	Results from encrypted columns aren't decrypted. The application receives encrypted values as byte arrays (byte[]).

Inserting and retrieving encrypted data examples

The following examples illustrate retrieving and modifying data in encrypted columns. The examples assume the target table with the following schema and encrypted SSN and BirthDate columns. If you've configured a Column Master Key named "MyCMK" and a Column Encryption Key named "MyCEK" (as described in the preceding keystore providers sections), you can create the table using this script:

```
CREATE TABLE [dbo].[Patients]([PatientId] [int] IDENTITY(1,1),
 [SSN] [char](11) COLLATE Latin1_General_BIN2
 ENCRYPTED WITH (ENCRYPTION_TYPE = DETERMINISTIC,
 ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256',
 COLUMN_ENCRYPTION_KEY = MyCEK) NOT NULL,
 [FirstName] [nvarchar](50) NULL,
 [LastName] [nvarchar](50) NULL,
 [BirthDate] [date]
 ENCRYPTED WITH (ENCRYPTION_TYPE = RANDOMIZED,
 ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256',
 COLUMN_ENCRYPTION_KEY = MyCEK) NOT NULL
 PRIMARY KEY CLUSTERED ([PatientId] ASC) ON [PRIMARY])
 GO
```

For each Java code example, you'll need to insert keystore-specific code in the location noted.

If you're using an Azure Key Vault keystore provider:

```
    String clientID = "<Azure Application ID>";
    String clientKey = "<Azure Application API Key Password>";
    SQLServerColumnEncryptionAzureKeyVaultProvider akvProvider = new
SQLServerColumnEncryptionAzureKeyVaultProvider(clientID, clientKey);
    Map<String, SQLServerColumnEncryptionKeyStoreProvider> keyStoreMap = new HashMap<String,
SQLServerColumnEncryptionKeyStoreProvider>();
    keyStoreMap.put(akvProvider.getName(), akvProvider);
    SQLServerConnection.registerColumnEncryptionKeyStoreProviders(keyStoreMap);
    String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=<databaseName>;user=<user>;password=
<password>;columnEncryptionSetting=Enabled;";
```

If you're using a Windows Certificate Store keystore provider:

```
    String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=<databaseName>;user=<user>;password=
<password>;columnEncryptionSetting=Enabled;";
```

If you're using a Java Key Store keystore provider:

```
    String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=<databaseName>;user=<user>;password=
<password>;columnEncryptionSetting=Enabled;keyStoreAuthentication=JavaKeyStorePassword;keyStoreLocation=<path
to jks or pfx file>;keyStoreSecret=<keystore secret/password>";
```

Inserting data example

This example inserts a row into the Patients table. Note the following items:

- There's nothing specific to encryption in the sample code. The Microsoft JDBC Driver for SQL Server automatically detects and encrypts the parameters that target encrypted columns. This behavior makes encryption transparent to the application.
- The values inserted into database columns, including the encrypted columns, are passed as parameters using SQLServerPreparedStatement. While using parameters is optional when sending values to non-encrypted columns (although, it's highly recommended because it helps prevent SQL injection), it's required for values that target encrypted columns. If the values inserted into the encrypted columns were passed as literals embedded in the query statement, the query would fail because the driver wouldn't be able to determine the values in the target encrypted columns and it wouldn't encrypt the values. As a result, the server would reject them as incompatible with the encrypted columns.
- All values printed by the program will be in plaintext, as the Microsoft JDBC Driver for SQL Server will transparently decrypt the data retrieved from the encrypted columns.
- If you're doing a lookup using a WHERE clause, the value used in the WHERE clause needs to be passed as a parameter so that the driver can transparently encrypt it before sending it to the database. In the following example, the SSN is passed as a parameter but the LastName is passed as a literal as LastName isn't encrypted.
- The setter method used for the parameter targeting the SSN column is setString(), which maps to the char/varchar SQL Server data type. If, for this parameter, the setter method used was setNString(), which maps to nchar/nvarchar, the query would fail, as Always Encrypted doesn't support conversions from encrypted nchar/nvarchar values to encrypted char/varchar values.

```
// <Insert keystore-specific code here>
try (Connection sourceConnection = DriverManager.getConnection(connectionUrl);
        PreparedStatement insertStatement = sourceConnection.prepareStatement("INSERT INTO [dbo].[Patients]
VALUES (?, ?, ?, ?)")) {
    insertStatement.setString(1, "795-73-9838");
    insertStatement.setString(2, "Catherine");
    insertStatement.setString(3, "Abel");
    insertStatement.setDate(4, Date.valueOf("1996-09-10"));
    insertStatement.executeUpdate();
    System.out.println("1 record inserted.\n");
}
// Handle any errors that may have occurred.
catch (SQLException e) {
    e.printStackTrace();
}
```

Retrieving plaintext data example

The following example demonstrates filtering data based on encrypted values and retrieving plaintext data from encrypted columns. Note the following items:

- The value used in the WHERE clause to filter on the SSN column needs to be passed as a parameter so that the Microsoft JDBC Driver for SQL Server can transparently encrypt it before sending it to the database.
- All values printed by the program will be in plaintext, as the Microsoft JDBC Driver for SQL Server will transparently decrypt the data retrieved from the SSN and BirthDate columns.

> **NOTE**
>
> if columns are encrypted using deterministic encryption, queries can perform equality comparisons on them. For more information, see Selecting Deterministic or Randomized encryption in Always Encrypted (Database Engine).

```
// <Insert keystore-specific code here>
try (Connection connection = DriverManager.getConnection(connectionUrl);
        PreparedStatement selectStatement = connection
                .prepareStatement("\"SELECT [SSN], [FirstName], [LastName], [BirthDate] FROM [dbo].[Patients]
WHERE SSN = ?;\"");) {
    selectStatement.setString(1, "795-73-9838");
    ResultSet rs = selectStatement.executeQuery();
    while (rs.next()) {
        System.out.println("SSN: " + rs.getString("SSN") + ", FirstName: " + rs.getString("FirstName") + ",
LastName:"
                + rs.getString("LastName") + ", Date of Birth: " + rs.getString("BirthDate"));
    }
}
// Handle any errors that may have occurred.
catch (SQLException e) {
    e.printStackTrace();
}
```

Retrieving encrypted data example

If Always Encrypted isn't enabled, a query can still retrieve data from encrypted columns, as long as the query has no parameters targeting encrypted columns.

The following example illustrates retrieving binary encrypted data from encrypted columns. Note the following items:

- Since Always Encrypted isn't enabled in the connection string, the query will return encrypted values of SSN and BirthDate as byte arrays (the program converts the values to strings).
- A query retrieving data from encrypted columns with Always Encrypted disabled can have parameters, as long

as none of the parameters target an encrypted column. The following query filters by LastName, which isn't encrypted in the database. If the query filtered by SSN or BirthDate, the query would fail.

```
try (Connection sourceConnection = DriverManager.getConnection(connectionUrl);
        PreparedStatement selectStatement = sourceConnection
                .prepareStatement("SELECT [SSN], [FirstName], [LastName], [BirthDate] FROM [dbo].[Patients]
WHERE LastName = ?;");) {

    selectStatement.setString(1, "Abel");
    ResultSet rs = selectStatement.executeQuery();
    while (rs.next()) {
        System.out.println("SSN: " + rs.getString("SSN") + ", FirstName: " + rs.getString("FirstName") + ",
LastName:"
                + rs.getString("LastName") + ", Date of Birth: " + rs.getString("BirthDate"));
    }
}
// Handle any errors that may have occurred.
catch (SQLException e) {
    e.printStackTrace();
}
```

Avoiding common problems when querying encrypted columns

This section describes common categories of errors when querying encrypted columns from Java applications and a few guidelines on how to avoid them.

Unsupported data type conversion errors

Always Encrypted supports few conversions for encrypted data types. See Always Encrypted (Database Engine) for the detailed list of supported type conversions. Here is what you can do to avoid data type conversion errors. Make sure that:

- you use the proper setter methods when passing values for parameters that target encrypted columns. Ensure that the SQL Server data type of the parameter is exactly the same as the type of the target column or a conversion of the SQL Server data type of the parameter to the target type of the column is supported. API methods have been added to the SQLServerPreparedStatement, SQLServerCallableStatement, and SQLServerResultSet classes to pass parameters corresponding to specific SQL Server data types. For example, if a column isn't encrypted you can use the setTimestamp() method to pass a parameter to a datetime2 or to a datetime column. But when a column is encrypted you'll have to use the exact method representing the type of the column in the database. For example, use setTimestamp() to pass values to an encrypted datetime2 column and use setDateTime() to pass values to an encrypted datetime column. See Always Encrypted API Reference for the JDBC Driver for a complete list of new APIs.
- the precision and scale of parameters targeting columns of the decimal and numeric SQL Server data types is the same as the precision and scale configured for the target column. API methods have been added to the SQLServerPreparedStatement, SQLServerCallableStatement, and SQLServerResultSet classes to accept precision and scale along with data values for parameters/columns representing decimal and numeric data types. See Always Encrypted API Reference for the JDBC Driver for a complete list of new/overloaded APIs.
- the fractional seconds precision/scale of parameters targeting columns of datetime2, datetimeoffset, or time SQL Server data types isn't greater than the fractional seconds precision/scale for the target column in queries that modify values of the target column. API methods have been added to the SQLServerPreparedStatement, SQLServerCallableStatement, and SQLServerResultSet classes to accept fractional seconds precision/scale along with data values for parameters representing these data types. For a complete list of new/overloaded APIs, see Always Encrypted API Reference for the JDBC Driver.

Errors due to incorrect connection properties

This section describes how to configure connection settings properly to use Always Encrypted data. Since encrypted data types support limited conversions, the **sendTimeAsDatetime** and **sendStringParametersAsUnicode** connection settings need proper configuration when using encrypted columns. Make sure that:

- sendTimeAsDatetime connection setting is set to false when inserting data into encrypted time columns. For more information, see Configuring how java.sql.Time Values are sent to the server.
- sendStringParametersAsUnicode connection setting is set to true (or is left as the default) when inserting data into encrypted char/varchar/varchar(max) columns.

Errors due to passing plaintext instead of encrypted values

Any value that targets an encrypted column needs to be encrypted inside the application. An attempt to insert/modify or to filter by a plaintext value on an encrypted column will result in an error similar to this one:

```
com.microsoft.sqlserver.jdbc.SQLServerException: Operand type clash: varchar is incompatible with
varchar(8000) encrypted with (encryption_type = 'DETERMINISTIC', encryption_algorithm_name =
'AEAD_AES_256_CBC_HMAC_SHA_256', column_encryption_key_name = 'MyCEK', column_encryption_key_database_name =
'ae') collation_name = 'SQL_Latin1_General_CP1_CI_AS'
```

To prevent such errors, make sure:

- always Encrypted is enabled for application queries targeting encrypted columns (for the connection string or for a specific query).
- you use prepared statements and parameters to send data targeting encrypted columns. The following example shows a query that incorrectly filters by a literal/constant on an encrypted column (SSN), instead of passing the literal inside as a parameter. This query will fail:

```
ResultSet rs = connection.createStatement().executeQuery("SELECT * FROM Customers WHERE SSN='795-73-9838'");
```

Force encryption on input parameters

The Force Encryption feature enforces encryption of a parameter when using Always Encrypted. If force encryption is used and SQL Server informs the driver that the parameter doesn't need to be encrypted, the query using the parameter will fail. This property provides additional protection against security attacks that involve a compromised SQL Server providing incorrect encryption metadata to the client, which may lead to data disclosure. The set* methods in the SQLServerPreparedStatement and SQLServerCallableStatement classes and the update* methods in the SQLServerResultSet class are overloaded to accept a boolean argument to specify the force encryption setting. If the value of this argument is false, the driver won't force encryption on parameters. If force encryption is set to true, the query parameter is only sent if the destination column is encrypted and Always Encrypted is enabled on the connection or on the statement. Using this property gives an extra layer of security, ensuring that the driver doesn't mistakenly send data to SQL Server as plaintext when it's expected to be encrypted.

For more information on the SQLServerPreparedStatement and SQLServerCallableStatement methods that are overloaded with the force encryption setting, see Always Encrypted API Reference for the JDBC Driver

Controlling the performance impact of Always Encrypted

Because Always Encrypted is a client-side encryption technology, most of the performance overhead is observed on the client side, not in the database. Apart from the cost of encryption and decryption operations, other sources of performance overheads on the client side are:

- Additional round trips to the database to retrieve metadata for query parameters.
- Calls to a column master key store to access a column master key.

This section describes the built-in performance optimizations in the Microsoft JDBC Driver for SQL Server and how you can control the impact of the above two factors on performance.

Controlling round trips to retrieve metadata for query parameters

If Always Encrypted is enabled for a connection, by default the driver will call sys.sp_describe_parameter_encryption for each parameterized query, passing the query statement (without any parameter values) to SQL Server. sys.sp_describe_parameter_encryption analyzes the query statement to find out if any parameters need to be encrypted, and if so, for each one it returns the encryption-related information that will allow the driver to encrypt parameter values. This behavior ensures a high level of transparency to the client application. As long as the application uses parameters to pass values that target encrypted columns to the driver, the application (and the application developer) doesn't need to know which queries access encrypted columns.

Setting Always Encrypted at the query level

To control the performance impact of retrieving encryption metadata for parameterized queries, you can enable Always Encrypted for individual queries instead of setting it up for the connection. This way you can ensure that sys.sp_describe_parameter_encryption is invoked only for queries that you know have parameters targeting encrypted columns. Note, however, that by doing so you reduce the transparency of encryption: if you change encryption properties of your database columns, you may need to change the code of your application to align it with the schema changes.

To control the Always Encrypted behavior of individual queries, you need to configure individual statement objects by passing an Enum, SQLServerStatementColumnEncryptionSetting, which specifies how data will be sent and received when reading and writing encrypted columns for that specific statement. Here are some useful guidelines:

- If most queries a client application sends over a database connection access encrypted columns, use these guidelines:
  - Set the **columnEncryptionSetting** connection string keyword to **Enabled**.
  - Set SQLServerStatementColumnEncryptionSetting.Disabled for individual queries that don't access any encrypted columns. This setting will disable both calling sys.sp_describe_parameter_encryption as well as an attempt to decrypt any values in the result set.
  - Set SQLServerStatementColumnEncryptionSetting.ResultSet for individual queries that don't have any parameters requiring encryption but retrieve data from encrypted columns. This setting will disable calling sys.sp_describe_parameter_encryption and parameter encryption. The query will be able to decrypt the results from encryption columns.
- If most queries a client application sends over a database connection don't access encrypted columns, use these guidelines:
  - Set the **columnEncryptionSetting** connection string keyword to **Disabled**.
  - Set SQLServerStatementColumnEncryptionSetting.Enabled for individual queries that have any parameters that need to be encrypted. This setting will enable both calling sys.sp_describe_parameter_encryption as well as the decryption of any query results retrieved from encrypted columns.
  - Set SQLServerStatementColumnEncryptionSetting.ResultSet for queries that don't have any parameters requiring encryption but retrieve data from encrypted columns. This setting will disable calling sys.sp_describe_parameter_encryption and parameter encryption. The query will be able to decrypt the results from encryption columns.

The SQLServerStatementColumnEncryptionSetting settings can't be used to bypass encryption and gain access to plaintext data. For more information on how to configure column encryption on a statement, see Always Encrypted API Reference for the JDBC Driver.

In the following example, Always Encrypted is disabled for the database connection. The query the application issues has a parameter that targets the LastName column that isn't encrypted. The query retrieves data from the SSN and BirthDate columns that are both encrypted. In such a case, calling sys.sp_describe_parameter_encryption to retrieve encryption metadata isn't required. However, the decryption of the query results needs to be enabled so that the application can receive plaintext values from the two encrypted columns. The SQLServerStatementColumnEncryptionSetting.ResultSet setting is used to ensure that.

```
// Assumes the same table definition as in Section "Retrieving and modifying data in encrypted columns"
// where only SSN and BirthDate columns are encrypted in the database.
String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=<database>;user=<user>;password=
<password>;"
        + "keyStoreAuthentication=JavaKeyStorePassword;"
        + "keyStoreLocation=<keyStoreLocation>"
        + "keyStoreSecret=<keyStoreSecret>;";

String filterRecord = "SELECT FirstName, LastName, SSN, BirthDate FROM " + tableName + " WHERE LastName = ?";

try (SQLServerConnection connection = (SQLServerConnection) DriverManager.getConnection(connectionUrl);
        PreparedStatement selectStatement = connection.prepareStatement(filterRecord,
ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY,
                connection.getHoldability(), SQLServerStatementColumnEncryptionSetting.ResultSetOnly);) {

    selectStatement.setString(1, "Abel");
    ResultSet rs = selectStatement.executeQuery();
    while (rs.next()) {
        System.out.println("First name: " + rs.getString("FirstName"));
        System.out.println("Last name: " + rs.getString("LastName"));
        System.out.println("SSN: " + rs.getString("SSN"));
        System.out.println("Date of Birth: " + rs.getDate("BirthDate"));
    }
}
// Handle any errors that may have occurred.
catch (SQLException e) {
    e.printStackTrace();
}
```

Column encryption key caching

To reduce the number of calls to a column master key store to decrypt column encryption keys, the Microsoft JDBC Driver for SQL Server caches the plaintext column encryption keys in memory. After receiving the encrypted column encryption key value from the database metadata, the driver first tries to find the plaintext column encryption key corresponding to the encrypted key value. The driver calls the keystore containing the column master key only if it cannot find the encrypted column encryption key value in the cache.

You can configure a time-to-live value for the column encryption key entries in the cache using the API, setColumnEncryptionKeyCacheTtl(), in the SQLServerConnection class. The default time-to-live value for the column encryption key entries in the cache is two hours. To turn off caching, use a value of 0. To set any time-to-live value, use the following API:

```
SQLServerConnection.setColumnEncryptionKeyCacheTtl (int columnEncryptionKeyCacheTTL, TimeUnit unit)
```

For example, to set a time-to-live value of 10 minutes, use:

```
SQLServerConnection.setColumnEncryptionKeyCacheTtl (10, TimeUnit.MINUTES)
```

Only DAYS, HOURS, MINUTES, or SECONDS are supported as the time unit.

Copying encrypted data using SQLServerBulkCopy

With SQLServerBulkCopy, you can copy data that is already encrypted and stored in one table to another table without decrypting the data. To do that:

- Make sure the encryption configuration of the target table is identical to the configuration of the source table.

In particular, both tables must have the same columns encrypted and the columns must be encrypted using the same encryption types and the same encryption keys. If any target column is encrypted differently than its corresponding source column, you won't be able to decrypt the data in the target table after the copy operation. The data will be corrupted.

- Configure both database connections to the source table and to the target table without Always Encrypted enabled.
- Set the allowEncryptedValueModifications option. For more information, see Using Bulk Copy with the JDBC Driver.

> **NOTE**
>
> Use caution when specifying AllowEncryptedValueModifications as this option may lead to corrupting the database because the Microsoft JDBC Driver for SQL Server does not check if the data is indeed encrypted or if it is correctly encrypted using the same encryption type, algorithm, and key as the target column.

See Also

Always Encrypted (Database Engine)

Using Statements with SQL

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When you work with data in a SQL Server database by using the Microsoft JDBC Driver for SQL Server and inline SQL statements, there are different classes that you can use. Which class you use depends on the type of SQL statement that you want to run.

If your SQL statement contains no IN parameters, use the SQLServerStatement class, but if it does contain IN parameters, use the SQLServerPreparedStatement class.

NOTE

If you need to use SQL statements that contain both IN and OUT parameters, you must implement them as stored procedures and call them by using the SQLServerCallableStatement class. For more information about using stored procedures, see Using Statements with Stored Procedures.

The following sections describe the different scenarios for working with data in a SQL Server database by using SQL statements.

In This Section

TOPIC	DESCRIPTION
Using an SQL Statement with No Parameters	Describes how to use SQL statements that contain no parameters.
Using an SQL Statement with Parameters	Describes how to use SQL statements that contain parameters.
Using an SQL Statement to Modify Database Objects	Describes how to use SQL statements to modify database objects.
Using an SQL Statement to Modify Data	Describes how to use SQL statements to modify data in a database.

See Also

Using Statements with the JDBC Driver

Using an SQL Statement to Modify Database Objects

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

To modify SQL Server database objects by using an SQL statement, you can use the executeUpdate method of the SQLServerStatement class. The executeUpdate method will pass the SQL statement to the database for processing, and then return a value of 0 because no rows were affected.

To do this, you must first create a SQLServerStatement object by using the createStatement method of the SQLServerConnection class.

NOTE

SQL statements that modify objects within a database are called Data Definition Language (DDL) statements. These include statements such as `CREATE TABLE`, `DROP TABLE`, `CREATE INDEX`, and `DROP INDEX`. For more information about the types of DDL statements that are supported by SQL Server, see SQL Server Books Online.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, an SQL statement is constructed that will create the simple TestTable in the database, and then the statement is run and the return value is displayed.

```
public static void executeUpdateStatement(Connection con) {
    try(Statement stmt = con.createStatement();) {
        String SQL = "CREATE TABLE TestTable (Col1 int IDENTITY, Col2 varchar(50), Col3 int)";
        int count = stmt.executeUpdate(SQL);
        System.out.println("ROWS AFFECTED: " + count);
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

See Also

Using Statements with SQL

Setting the Data Source Properties

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Data sources are the preferred mechanism by which to create JDBC connections in a Java Platform, Enterprise Edition (Java EE) environment. Data sources provide connections, pooled connections, and distributed connections without hard-coding connection properties into Java code. All Microsoft JDBC Driver for SQL Server data sources can set or get the value of any property by using the appropriate setter and getter methods, respectively.

Java EE products, such as application servers and servlet/JSP engines, typically let you configure data sources for database access. Any property listed in the Setting the Connection Properties topic can be specified wherever the configuration lets you enter a property as a property=value pair.

For more information about SQL Server data sources, see the SQLServerDataSource class. For an example of how to use the SQLServerDataSource class to make a connection to a SQL Server database, see Data Source Sample.

See Also

Connecting to SQL Server with the JDBC Driver

Setting the Connection Properties

8/13/2018 • 20 minutes to read • Edit Online

Download JDBC Driver

The connection string properties can be specified in various ways:

- As name=value properties in the connection URL when you connect by using the DriverManager class.
- As name=value properties in the *Properties* parameter of the Connect method in the DriverManager class.
- As values in the appropriate setter method of the data source of the driver. For example:

```
datasource.setServerName(value)
datasource.setDatabaseName(value)
```

Remarks

Property names are case-insensitive, and duplicate property names are resolved in the following order:

1. API arguments (such as user and password)
2. Property collection.
3. Last instance in the connection string.

Also, unknown values are allowed for the property names, and their values are not validated by the JDBC driver for case sensitivity.

Synonyms are allowed and are resolved in order, just as duplicate property names.

The following table lists all the currently available connection string properties for the JDBC driver.

PROPERTY **TYPE** **DEFAULT**	**DESCRIPTION**
accessToken String null	Use this property to connect to a SQL database using an access token. **accessToken** can't be set using the connection URL.
applicationIntent String ReadWrite	Declares the application workload type when connecting to a server. Possible values are **ReadOnly** and **ReadWrite**. For more information, see JDBC Driver Support for High Availability, Disaster Recovery.
applicationName String [<=128 char] null	The application name, or " Microsoft JDBC Driver for SQL Server" if no name is provided. Used to identify the specific application in various SQL Server profiling and logging tools.

PROPERTY TYPE DEFAULT	DESCRIPTION
authentication String NotSpecified	Beginning with Microsoft JDBC Driver 6.0 for SQL Server, this optional property indicates which SQL authentication method to use for connection. Possible values are **ActiveDirectoryIntegrated**, **ActiveDirectoryPassword**, **SqlPassword**, and the default **NotSpecified**. Use **ActiveDirectoryIntegrated** to connect to a SQL Database using integrated Windows authentication. Use **ActiveDirectoryPassword** to connect to a SQL Database using an Azure AD principal name and password. Use **SqlPassword** to connect to a SQL Server using **userName/user** and **password** properties. Use **NotSpecified** if none of these authentication methods are needed. **Important:** If authentication is set to ActiveDirectoryIntegrated, the following two libraries need to be installed: **SQLJDBC_AUTH.DLL** (available in the JDBC driver package) and Azure Active Directory Authentication Library for SQL Server (**ADALSQL.DLL**) It's available in different languages (for both x86 and amd64) from the download center at Microsoft Active Directory Authentication Library for Microsoft SQL Server. The JDBC driver only supports version **1.0.2028.318 and higher** for the ADALSQL.DLL. **Note:** When authentication property is set to any value other than **NotSpecified**, the driver by default uses Secure Sockets Layer (SSL) encryption. For information on how to configure Azure Active Directory authentication visit Connecting to SQL Database By Using Azure Active Directory Authentication.
authenticationScheme String NativeAuthentication	Indicates which kind of integrated security you want your application to use. Possible values are **JavaKerberos** and the default **NativeAuthentication**. When using **authenticationScheme=JavaKerberos**, you must specify the fully qualified domain name (FQDN) in the **serverName** or **serverSpn** property. Otherwise, an error occurs (Server not found in Kerberos database). For more information on using **authenticationScheme**, see Using Kerberos Integrated Authentication to Connect to SQL Server.

PROPERTY TYPE DEFAULT	DESCRIPTION
cancelQueryTimeout int -1	Beginning with Microsoft JDBC Driver 6.4 for SQL Server, this property can be used to cancel **queryTimeout** set on the connection. Query execution hangs and does not throw exception if TCP connection to SQL Server is silently dropped. This property is only applicable if 'queryTimeout' is also set on the connection. The driver waits the total amount of **cancelQueryTimeout** + **queryTimeout** seconds, to drop the connection and close the channel. The default value for this property is -1 and behavior is to wait indefinitely.
columnEncryptionSetting String ["Enabled" | "Disabled"] Disabled	Set to "Enabled" to use the Always Encrypted (AE) feature beginning with Microsoft JDBC Driver 6.0 for SQL Server. When AE is enabled, the JDBC driver transparently encrypts and decrypts sensitive data stored in encrypted database columns in the SQL Server. For more information about **columnEncryptionSetting**, see Using Always Encrypted with the JDBC Driver for more details. **Note:** Always Encrypted is available with SQL Server 2016 or later versions.
databaseName, database String [<=128 char] null	The name of the database to connect to. If not stated, a connection is made to the default database.
disableStatementPooling boolean ["true" | "false"] true	Flag indicating if the statement pooling should be used.
enablePrepareOnFirst... PreparedStatementCall boolean ["true" | "false"] false	*enablePrepareOnFirstPreparedStatementCall* Set to "true" to enable the prepared statement handle creation by calling `sp_prepexec` in first execution of prepared statement. Set to "false" to change the first execution of a prepared statement to call `sp_executesql` and not prepare a statement, once the second execution happens it would call `sp_prepexec` to setup a prepared statement handle.

PROPERTY TYPE DEFAULT	DESCRIPTION
encrypt boolean ["true" | "false"] false	Set to "true" to specify that the SQL Server uses Secure Sockets Layer (SSL) encryption for all the data sent between the client and the server if the server has a certificate installed. The default value is "false". Beginning with Microsoft JDBC Driver 6.0 for SQL Server, there is a new connection setting 'authentication' that uses SSL encryption by default. For more information, see the 'authentication' property.
failoverPartner String null	The name of the failover server used in a database mirroring configuration. This property is used for an initial connection failure to the principal server; after you make the initial connection, this property is ignored. Must be used in conjunction with databaseName property. **Note:** The driver does not support specifying the server instance port number for the failover partner instance as part of the failoverPartner property in the connection string. However, specifying the serverName, instanceName, and portNumber properties of the principal server instance and failoverPartner property of the failover partner instance in the same connection string is supported. If you specify a Virtual Network Name in the **Server** connection property, you cannot use database mirroring. For more information, see JDBC Driver Support for High Availability, Disaster Recovery
fips boolean ["true" | "false"] "false"	For FIPS enabled JVM this property should be **true**.
fipsProvider String null	FIPS provider configured in JVM. For example, BCFIPS or SunPKCS11-NSS. Removed in version 6.4.0 - see the details Here.
gsscredential org.ietf.jgss.GSSCredential null	Beginning with Microsoft JDBC Driver 6.2 for SQL Server, user credentials to be used for Kerberos Constrained Delegation can be passed in this property. This should be used with **integratedSecurity** as **true** and **JavaKerberos authenticationscheme**.

PROPERTY TYPE DEFAULT	DESCRIPTION
hostNameInCertificate String null	The host name to be used in validating the SQL Server SSL certificate. If the hostNameInCertificate property is unspecified or set to null, the Microsoft JDBC Driver for SQL Server uses the **serverName** property value on the connection URL as the host name to validate the SQL Server SSL certificate. **Note:** This property is used in combination with the **encrypt**/**authentication** properties and the **trustServerCertificate** property. This property affects the certificate validation, if and only if the connection uses Secure Sockets Layer (SSL) encryption and the **trustServerCertificate** is set to "false". Make sure the value passed to **hostNameInCertificate** exactly matches the Common Name (CN) or DNS name in the Subject Alternate Name (SAN) in the server certificate for an SSL connection to succeed. For more information, see Understanding SSL Support.
instanceName String [<=128 char] null	The SQL Server instance name to connect to. When it is not specified, a connection is made to the default instance. For the case where both the instanceName and port are specified, see the notes for port. If you specify a Virtual Network Name in the **Server** connection property, you cannot use **instanceName** connection property. See JDBC Driver Support for High Availability, Disaster Recovery for more information.
integratedSecurity boolean ["true"|"false"] false	Set to "true" to indicate that Windows credentials are used by SQL Server on Windows operating systems. If "true," the JDBC driver searches the local computer credential cache for credentials that have already been provided at the computer or network logon. Set to "true" (with **authenticationscheme=JavaKerberos**), to indicate that Kerberos credentials are used by SQL Server. For more information on Kerberos authentication, see Using Kerberos Integrated Authentication to Connect to SQL Server. If "false," the username and password must be supplied.
jaasConfigurationName String SQLJDBCDriver	Beginning with Microsoft JDBC Driver 6.2 for SQL Server, each connection to SQL Server can have its own JAAS Login Configuration file to establish Kerberos connection. Name of the Login Configuration file can be passed through this property. By default, driver sets property `useDefaultCcache = true` for IBM JVMs, and `useTicketCache = true` for other JVMs.

PROPERTY TYPE DEFAULT	DESCRIPTION
keyStoreAuthentication String null	Beginning with Microsoft JDBC Driver 6.0 for SQL Server, this property identifies which key store to seamlessly set up for the connection with Always Encrypted and determines an authentication mechanism used to authenticate to the key store. Microsoft JDBC Driver 6.0 for SQL Server supports setting up of the Java Key Store seamlessly using this property for which you need to set "**keyStoreAuthentication=JavaKeyStorePassword**". Note that to use this property, you also need to set the **keyStoreLocation** and **keyStoreSecret** properties for the Java Key Store. For more information, visit Using Always Encrypted with the JDBC Driver.
keyStoreLocation String null	When **keyStoreAuthentication=JavaKeyStorePassword**, the **keyStoreLocation** property identifies the path to the Java keystore file that stores the column master key to be used with Always Encrypted data. Note that the path must include the keystore filename. For more information, visit Using Always Encrypted with the JDBC Driver.
keyStoreSecret String null	When **keyStoreAuthentication=JavaKeyStorePassword**, the **keyStoreSecret** property identifies the password to use for the keystore as well as for the key. Note that for using the Java Key Store the keystore and the key password must be the same. For more information, visit Using Always Encrypted with the JDBC Driver.
lastUpdateCount boolean ["true" | "false"] true	A "true" value only returns the last update count from an SQL statement passed to the server, and it can be used on single SELECT, INSERT, or DELETE statements to ignore additional update counts caused by server triggers. Setting this property to "false" causes all update counts to be returned, including those returned by server triggers. **Note:** This property only applies when it is used with the executeUpdate methods. All other execute methods return all results and update counts. This property only affects update counts returned by server triggers. It does not affect result sets or errors that result as part of trigger execution.
lockTimeout int -1	The number of milliseconds to wait before the database reports a lock time-out. The default behavior is to wait indefinitely. If it is specified, this value is the default for all statements on the connection. Note that **Statement.setQueryTimeout()** can be used to set the time-out for specific statements. The value can be 0, which specifies no wait.

PROPERTY TYPE DEFAULT	DESCRIPTION
loginTimeout int [0..65535] 15	The number of seconds the driver should wait before timing out a failed connection. A zero value indicates that the timeout is the default system timeout, which is specified as 15 seconds by default. A non-zero value is the number of seconds the driver should wait before timing out a failed connection. If you specify a Virtual Network Name in the **Server** connection property, you should specify a timeout value of three minutes or more to allow sufficient time for a failover connection to succeed. See JDBC Driver Support for High Availability, Disaster Recovery for more information.
multiSubnetFailover Boolean false	Always specify **multiSubnetFailover=true** when connecting to the availability group listener of a SQL Server 2012 (11.x) availability group or a SQL Server 2012 (11.x) Failover Cluster Instance. **multiSubnetFailover=true** configures Microsoft JDBC Driver for SQL Server to provide faster detection of and connection to the (currently) active server. Possible values are true and false. See JDBC Driver Support for High Availability, Disaster Recovery for more information. You can programmatically access the **multiSubnetFailover** connection property with getPropertyInfo, getMultiSubnetFailover, and setMultiSubnetFailover. **Note:** Beginning with Microsoft JDBC Driver 6.0 for SQL Server, it is no longer required to set **multiSubnetFailover** to "true" when connecting to an Availability Group Listener. A new property, **transparentNetworkIPResolution**, which is enabled by default, provides the detection of and connection to the (currently) active server.
packetSize int [-1 | 0 | 512..32767] 8000	The network packet size used to communicate with SQL Server, specified in bytes. A value of -1 indicates using the server default packet size. A value of 0 indicates using the maximum value, which is 32767. If this property is set to a value outside the acceptable range, an exception occurs. **Important:** We do not recommend using the packetSize property when the encryption is enabled (encrypt=true). Otherwise, the driver might raise a connection error. For more information, see the setPacketSize method of the SQLServerDataSource class.
password String [<=128 char] null	The database password, in case of connection with SQL user and password. For Kerberos connection with principal name and password, this property is set to Kerberos Principal password.

PROPERTY TYPE DEFAULT	DESCRIPTION
portNumber, port int [0..65535] 1433	The port where SQL Server is listening. If the port number is specified in the connection string, no request to SQLbrowser is made. When the port and instanceName are both specified, the connection is made to the specified port. However, the **instanceName** is validated and an error is thrown if it does not match the port. **Important:** We recommend that the port number is always specified, as this is more secure than using SQLbrowser.
queryTimeout int -1	The number of seconds to wait before a timeout has occurred on a query. The default value is -1, which means infinite timeout. Setting this to 0 also implies to wait indefinitely.
responseBuffering String ["full" | "adaptive"] adaptive	If this property is set to "adaptive", the minimum possible data is buffered when necessary. The default mode is "adaptive." When this property is set to "full", the entire result set is read from the server when a statement is executed. **Note:** After upgrading the JDBC driver from version 1.2, the default buffering behavior will be "adaptive." If your application has never set the "responseBuffering" property and you want to keep the version 1.2 default behavior in your application, you must set the responseBufferring property to "full" either in the connection properties or by using the setResponseBuffering method of the SQLServerStatement object.
selectMethod String ["direct" | "cursor"] direct	If this property is set to "cursor," a database cursor is created for each query created on the connection for **TYPE_FORWARD_ONLY** and **CONCUR_READ_ONLY** cursors. This property is typically required only if the application generates large result sets that cannot be fully contained in client memory. When this property is set to "cursor," only a limited number of result set rows are retained in client memory. The default behavior is that all result set rows are retained in client memory. This behavior provides the fastest performance when the application is processing all rows.

PROPERTY **TYPE** **DEFAULT**	**DESCRIPTION**
sendStringParameters... AsUnicode boolean ["true" | "false"] true	*sendStringParametersAsUnicode* If the **sendStringParametersAsUnicode** property is set to "true", String parameters are sent to the server in Unicode format. If the **sendStringParametersAsUnicode** property is set to "false", String parameters are sent to the server in non-Unicode format such as ASCII/MBCS instead of Unicode. The default value for the **sendStringParametersAsUnicode** property is "true". **Note:** The **sendStringParametersAsUnicode** property is only checked when sending a parameter value with **CHAR**, **VARCHAR**, or **LONGVARCHAR** JDBC types. The new JDBC 4.0 national character methods, such as the setNString, setNCharacterStream, and setNClob methods of SQLServerPreparedStatement and SQLServerCallableStatement classes, always send their parameter values to the server in Unicode regardless of the setting of this property. For optimal performance with the **CHAR**, **VARCHAR**, and **LONGVARCHAR** JDBC data types, an application should set the **sendStringParametersAsUnicode** property to "false" and use the setString, setCharacterStream, and setClob non-national character methods of the SQLServerPreparedStatement and SQLServerCallableStatement classes. When the application sets the **sendStringParametersAsUnicode** property to "false" and uses a non-national character method to access Unicode data types on the server side (such as **nchar**, **nvarchar** and **ntext**), some data might be lost if the database collation does not support the characters in the String parameters passed by the non-national character method. Note that an application should use the setNString, setNCharacterStream, and setNClob national character methods of the SQLServerPreparedStatement and SQLServerCallableStatement classes for the **NCHAR**, **NVARCHAR**, and **LONGNVARCHAR** JDBC data types.
sendTimeAsDatetime boolean ["true" | "false"] false	This property was added in SQL Server JDBC Driver 3.0. Set to "true" to send java.sql.Time values to the server as SQL Server **datetime** values. Set to "false" to send java.sql.Time values to the server as SQL Server **time** values. The default value for this property may change in a future release. For more information about how the Microsoft JDBC Driver for SQL Server configures java.sql.Time values before sending them to the server, see Configuring How java.sql.Time Values are Sent to the Server.

PROPERTY TYPE DEFAULT	DESCRIPTION
serverName, server String null	The computer running SQL Server. You can also specify the Virtual Network Name of a AlwaysOn Availability Groups availability group. See JDBC Driver Support for High Availability, Disaster Recovery for more information.
serverNameAsACE boolean ["true" | "false"] false	Beginning with Microsoft JDBC Driver 6.0 for SQL Server, set to "true" to indicate that the driver should translate the Unicode server name to ASCII compatible encoding (Punycode) for the connection. If this setting is false, the driver connects using the server name as provided by the user. See International Features of the JDBC Driver for more details.
serverPreparedStatement... DiscardThreshold Integer 10	*serverPreparedStatementDiscardThreshold* Beginning with JDBC Driver 6.2 for SQL Server, this property can be used to control how many outstanding prepared statement discard actions (`sp_unprepare`) can be outstanding per connection before a call to clean up the outstanding handles on the server is executed. If this property is set to <= 1, unprepare actions are executed immediately on prepared statement close. If it is set to >1 these calls are batched together to avoid overhead of calling sp_unprepare too often.
serverSpn String null	Beginning in Microsoft JDBC Driver 4.2 for SQL Server, this optional property can be used to specify the Service Principal Name (SPN) for a Java Kerberos connection. It is used in conjunction with **authenticationScheme**. To specify the SPN, it can be in the form of: “MSSQLSvc/fqdn:port@REALM” where fqdn is the fully qualified domain name, port is the port number, and REALM is the Kerberos realm of the SQL Server in upper-case letters. Note: the @REALM is optional if the default realm of the client (as specified in the Kerberos configuration) is the same as the Kerberos realm for the SQL Server. For more information on using **serverSpn** with Java Kerberos, see Using Kerberos Integrated Authentication to Connect to SQL Server.

PROPERTY TYPE DEFAULT	DESCRIPTION
statementPooling... CacheSize int 0	*statementPoolingCacheSize* Beginning with JDBC Driver 6.4 for SQL Server, this property can be used to enable Prepared Statement Handle Caching in the driver. This property defines the size of the cache for statement pooling. This property can only be used in conjunction with **disableStatementPooling** connection proeprty which should be set to "false". Setting **disableStatementPooling** to "true" or **statementPoolingCacheSize** to 0 disables prepared statement handle caching.
socketTimeout int 0	The number of milliseconds to wait before a timeout is occurred on a socket read or accept. The default value is 0, which means infinite timeout.
sslProtocol String TLS	Beginning with JDBC Driver 6.4 for SQL Server, this property can used to specify TLS protocol to be considered during secure connection. Possible values are: **TLS**, **TLSv1**, **TLSv1.1**, and **TLSv1.2**. For more information, see SSLProtocol.
transparentNetwork... IPResolution boolean ["true" | "false"] true	*transparentNetworkIPResolution* Beginning with Microsoft JDBC Driver 6.0 for SQL Server, this property provides faster detection of and connection to the (currently) active server. Possible values are "true" and "false" where "true" is the default value. Prior to Microsoft JDBC Driver 6.0 for SQL Server, an application had to set the connection string to include “multiSubnetFailover=true” to indicate that it was connecting to an AlwaysOn Availability Group. Without setting the **multiSubnetFailover** connection keyword to "true", an application might experience a timeout while connecting to an AlwaysOn Availability Group. Beginning with Microsoft JDBC Driver 6.0 for SQL Server, an application does not need to set multiSubnetFailover to true anymore. **Note:** When transparentNetworkIPResolution=true, the first connection attempt uses 500 ms as the timeout. Any subsequent attempts use the same timeout logic as used by the multiSubnetFailover property.
trustManagerClass String null	The fully qualified class name of a custom `javax.net.ssl.TrustManager` implementation.

PROPERTY TYPE DEFAULT	DESCRIPTION
trustManager... ConstructorArg String null	*trustManagerConstructorArg* An optional argument to pass to the constructor of the TrustManager. If trustManagerClass is specified and an encrypted connection is requested, the custom TrustManager is used rather than the default system JVM keystore-based TrustManager.
trustServerCertificate boolean ["true" | "false"] false	Set to "true" to specify that the Microsoft JDBC Driver for SQL Server does not validate the SQL Server SSL certificate. If "true", the SQL Server SSL certificate is automatically trusted when the communication layer is encrypted using SSL. If "false", the Microsoft JDBC Driver for SQL Server validates the server SSL certificate. If the server certificate validation fails, the driver raises an error and terminate the connection. The default value is "false". Make sure the value passed to **serverName** exactly matches the Common Name (CN) or DNS name in the Subject Alternate Name in the server certificate for an SSL connection to succeed. For more information, see Understanding SSL Support. **Note:** This property is used in combination with the **encrypt/authentication** properties. This property only affects server SSL certificate validation if and only if the connection uses SSL encryption.
trustStore String null	The path (including filename) to the certificate trustStore file. The trustStore file contains the list of certificates that the client trusts. When this property is unspecified or set to null, the driver relies on the trust manager factory's lookup rules to determine which certificate store to use. **The default SunX509 TrustManagerFactory tries to locate the trusted material in the following search order:** A file specified by the "javax.net.ssl.trustStore" Java Virtual Machine (JVM) system property. "<java-home>/lib/security/jssecacerts" file. "<java-home>/lib/security/cacerts" file. For more information, see the SUNX509 TrustManager Interface documentation on the Sun Microsystems Web site. **Note:** This property only affects the certificate trustStore lookup, if and only if the connection uses SSL encryption and the **trustServerCertificate** property is set to "false".

PROPERTY TYPE DEFAULT	DESCRIPTION
trustStorePassword String null	The password used to check the integrity of the trustStore data. If the trustStore property is set but the trustStorePassword property is not set, the integrity of the trustStore is not checked. When both trustStore and trustStorePassword properties are unspecified, the driver uses the JVM system properties, "javax.net.ssl.trustStore" and "javax.net.ssl.trustStorePassword". If the "javax.net.ssl.trustStorePassword" system property is not specified, the integrity of the trustStore is not checked. If the trustStore property is not set but the trustStorePassword property is set, the JDBC driver uses the file specified by the "javax.net.ssl.trustStore" as a trust store and the integrity of the trust store is checked by using the specified trustStorePassword. This might be needed when the client application does not want to store the password in the JVM system property. **Note:** The trustStorePassword property only affects the certificate trustStore lookup, if and only if the connection uses SSL connection and the **trustServerCertificate** property is set to "false".
trustStoreType String JKS	Set this property to specify trust store type to be used for FIPS mode. Possible values are either **PKCS12** or type defined by FIPS provider.
useBulkCopyFor... BatchInsert boolean ["true" | "false"] false	*useBulkCopyForBatchInsert* Beginning with Microsoft JDBC Driver 7.0 for SQL Server, this connection property can be enabled to make use of Bulk Copy API when performing batch insert operations using `java.sql.PreparedStatement` for performance improvement. This feature is functional only when target server is of type **Azure Data Warehouse**. It is disabled by default, set this property to "true" to enable this feature. **Important Note:** This feature only supports fully parameterized INSERT queries. If the INSERT Queries are combined by other SQL queries, or contain data in values, execution will fall back to the basic Batch Insert operation. For more information on how to use this property, see Using Bulk Copy API for Batch Insert Operation

PROPERTY TYPE DEFAULT	DESCRIPTION
userName, user String [<=128 char] null	The database user, in case of connection with SQL user and password. For Kerberos connection with principal name and password, this property is set to Kerberos Principal name.
workstationID String [<=128 char] <empty string>	The workstation ID. Used to identify the specific workstation in various SQL Server profiling and logging tools. If none is specified, the <empty string> is used.
xopenStates boolean ["true" | "false"] false	Set to "true" to specify that the driver returns XOPEN-compliant state codes in exceptions. The default is to return SQL 99 state codes.

NOTE

The Microsoft JDBC Driver for SQL Server takes the server default values for connection properties except for ANSI_DEFAULTS and IMPLICIT_TRANSACTIONS. The Microsoft JDBC Driver for SQL Server automatically sets ANSI_DEFAULTS to ON and IMPLICIT_TRANSACTIONS to OFF.

IMPORTANT

If authentication is set to ActiveDirectoryPassword, the following library needs to be included in classpath: azure-activedirectory-library-for-java. It can be found on Maven Repository. The simplest way to download the library and its dependencies is using Maven:

1. First, install Maven on your system
2. Go to the GitHub page of the driver
3. Download the pom.xml file
4. Run the following Maven command to download the library and its dependencies:
   `mvn dependency:copy-dependencies`

See Also

Connecting to SQL Server with the JDBC Driver
FIPS Mode

Using a Stored Procedure with No Parameters

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The simplest kind of SQL Server stored procedure that you can call is one that contains no parameters and returns a single result set. The Microsoft JDBC Driver for SQL Server provides the SQLServerStatement class, which you can use to call this kind of stored procedure and process the data that it returns.

When you use the JDBC driver to call a stored procedure without parameters, you must use the `call` SQL escape sequence. The syntax for the `call` escape sequence with no parameters is as follows:

`{call procedure-name}`

NOTE
For more information about the SQL escape sequences, see Using SQL Escape Sequences.

As an example, create the following stored procedure in the AdventureWorks sample database:

```
CREATE PROCEDURE GetContactFormalNames
AS
BEGIN
   SELECT TOP 10 Title + ' ' + FirstName + ' ' + LastName AS FormalName
   FROM Person.Contact
END
```

This stored procedure returns a single result set that contains one column of data, which is a combination of the title, first name, and last name of the top 10 contacts that are in the Person.Contact table.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, and the executeQuery method is used to call the GetContactFormalNames stored procedure.

```
public static void executeSprocNoParams(Connection con) throws SQLException {
    try(Statement stmt = con.createStatement();) {

        ResultSet rs = stmt.executeQuery("{call dbo.GetContactFormalNames}");
        while (rs.next()) {
            System.out.println(rs.getString("FormalName"));
        }
    }
}
```

See Also

Using Statements with Stored Procedures

Using Parameter Metadata

8/8/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

To query a SQLServerPreparedStatement or a SQLServerCallableStatement object about the parameters that they contain, the Microsoft JDBC Driver for SQL Server implements the SQLServerParameterMetaData class. This class contains numerous fields and methods that return information in the form of a single value.

To create a SQLServerParameterMetaData object, you can use the getParameterMetaData methods of the SQLServerPreparedStatement and SQLServerCallableStatement classes.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, the getParameterMetaData method of the SQLServerCallableStatement class is used to return a SQLServerParameterMetaData object, and then various methods of the SQLServerParameterMetaData object are used to display information about the type and mode of the parameters that are contained within the HumanResources.uspUpdateEmployeeHireInfo stored procedure.

```
public static void getParameterMetaData(Connection con) {
    try(CallableStatement cstmt = con.prepareCall("{call HumanResources.uspUpdateEmployeeHireInfo(?, ?, ?, ?,
?)}");) {
        ParameterMetaData pmd = cstmt.getParameterMetaData();
        int count = pmd.getParameterCount();
        for (int i = 1; i <= count; i++) {
            System.out.println("TYPE: " + pmd.getParameterTypeName(i) + " MODE: " + pmd.getParameterMode(i));
        }
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

NOTE

There are some limitations when using the SQLServerParameterMetaData class with prepared statements.

With Microsoft JDBC Driver 6.0 (or higher) for SQL Server: When using SQL Server 2008 or 2008 R2, the JDBC driver supports SELECT, DELETE, INSERT, and UPDATE statements as long as these statements does not contain subqueries and/or joins.

MERGE queries are also not supported for SQLServerParameterMetaData class when using SQL Server 2008 or 2008 R2. For SQL Server 2012 and higher versions parameter metadata with complex queries are supported.

Retrieval of parameter metadata for encrypted columns are not supported. **With Microsoft JDBC Driver 4.1 or 4.2 for SQL Server**: The JDBC driver supports SELECT, DELETE, INSERT, and UPDATE statements as long as these statements does not contain subqueries and/or joins. MERGE queries are also not supported for SQLServerParameterMetaData class.

Using Basic Data Types

8/13/2018 • 5 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server uses the JDBC basic data types to convert the SQL Server data types to a format that can be understood by the Java programming language, and vice versa. The JDBC driver provides support for the JDBC 4.0 API, which includes the **SQLXML** data type, and National (Unicode) data types, such as **NCHAR**, **NVARCHAR**, **LONGNVARCHAR**, and **NCLOB**.

Data Type Mappings

The following table lists the default mappings between the basic SQL Server, JDBC, and Java programming language data types:

SQL SERVER TYPES	JDBC TYPES (JAVA.SQL.TYPES)	JAVA LANGUAGE TYPES
bigint	BIGINT	long
binary	BINARY	byte[]
bit	BIT	boolean
char	CHAR	String
date	DATE	java.sql.Date
datetime	TIMESTAMP	java.sql.Timestamp
datetime2	TIMESTAMP	java.sql.Timestamp
datetimeoffset (2)	microsoft.sql.Types.DATETIMEOFFSET	microsoft.sql.DateTimeOffset
decimal	DECIMAL	java.math.BigDecimal
float	DOUBLE	double
image	LONGVARBINARY	byte[]
int	INTEGER	int
money	DECIMAL	java.math.BigDecimal
nchar	CHAR NCHAR (Java SE 6.0)	String
ntext	LONGVARCHAR LONGNVARCHAR (Java SE 6.0)	String

SQL SERVER TYPES	JDBC TYPES (JAVA.SQL.TYPES)	JAVA LANGUAGE TYPES
numeric	NUMERIC	java.math.BigDecimal
nvarchar	VARCHAR NVARCHAR (Java SE 6.0)	String
nvarchar(max)	VARCHAR NVARCHAR (Java SE 6.0)	String
real	REAL	float
smalldatetime	TIMESTAMP	java.sql.Timestamp
smallint	SMALLINT	short
smallmoney	DECIMAL	java.math.BigDecimal
text	LONGVARCHAR	String
time	TIME (1)	java.sql.Time (1)
timestamp	BINARY	byte[]
tinyint	TINYINT	short
udt	VARBINARY	byte[]
uniqueidentifier	CHAR	String
varbinary	VARBINARY	byte[]
varbinary(max)	VARBINARY	byte[]
varchar	VARCHAR	String
varchar(max)	VARCHAR	String
xml	LONGVARCHAR LONGNVARCHAR (Java SE 6.0)	String SQLXML
sqlvariant	SQLVARIANT	Object
geometry	VARBINARY	byte[]
geography	VARBINARY	byte[]

(1) To use java.sql.Time with the time SQL Server type, you must set the **sendTimeAsDatetime** connection property to false.

(2) You can programmatically access values of **datetimeoffset** with DateTimeOffset Class.

The following sections provide examples of how you can use the JDBC Driver and the basic data types. For a more detailed example of how to use the basic data types in a Java application, see Basic Data Types Sample.

Retrieving Data as a String

If you have to retrieve data from a data source that maps to any of the JDBC basic data types for viewing as a string, or if strongly typed data is not required, you can use the getString method of the SQLServerResultSet class, as in the following:

```
try(Statement stmt = con.createStatement();) {
    ResultSet rs = stmt.executeQuery("SELECT lname, job_id FROM employee WHERE (lname = 'Brown')");
    rs.next();
    short empJobID = rs.getShort("job_id");
}
```

Retrieving Data by Data Type

If you have to retrieve data from a data source, and you know the type of data that is being retrieved, use one of the get<Type> methods of the SQLServerResultSet class, also known as the *getter methods*. You can use either a column name or a column index with the get<Type> methods, as in the following:

```
try(Statement stmt = con.createStatement();) {
    ResultSet rs = stmt.executeQuery("SELECT lname, job_id FROM employee WHERE (lname = 'Brown')");
    rs.next();
    short empJobID = rs.getShort("job_id");
}
```

NOTE

The getUnicodeStream and getBigDecimal with scale methods are deprecated and are not supported by the JDBC driver.

Updating Data by Data Type

If you have to update the value of a field in a data source, use one of the update<Type> methods of the SQLServerResultSet class. In the following example, the updateInt method is used in conjunction with the updateRow method to update the data in the data source:

```
try (Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);) {
    ResultSet rs = stmt.executeQuery("SELECT lname, job_id FROM employee WHERE (lname = 'Brown')");
    rs.next();
    int empJobID = rs.getInt(2);
    empJobID++;
    rs.first();
    rs.updateInt(2, empJobID);
    rs.updateRow();
}
```

NOTE

The JDBC driver cannot update a SQL Server column with a column name that is more than 127 characters long. If an update to a column whose name is more than 127 characters is attempted, an exception is thrown.

Updating Data by Parameterized Query

If you have to update data in a data source by using a parameterized query, you can set the data type of the parameters by using one of the set<Type> methods of the SQLServerPreparedStatement class, also known as the *setter methods*. In the following example, the prepareStatement method is used to pre-compile the parameterized query, and then the setString method is used to set the string value of the parameter before the executeUpdate method is called.

```
try(PreparedStatement pstmt = con.prepareStatement("UPDATE employee SET fname = ? WHERE (lname = 'Brown')");)
{
    String name = "Bob";
    pstmt.setString(1, name);
    int rowCount = pstmt.executeUpdate();
}
```

For more information about parameterized queries, see Using an SQL Statement with Parameters.

Passing Parameters to a Stored Procedure

If you have to pass typed parameters into a stored procedure, you can set the parameters by index or name by using one of the set<Type> methods of the SQLServerCallableStatement class. In the following example, the prepareCall method is used to set up the call to the stored procedure, and then the setString method is used to set the parameter for the call before the executeQuery method is called.

```
try(CallableStatement cstmt = con.prepareCall("{call employee_jobid(?)}");) {
    String lname = "Brown";
    cstmt.setString(1, lname);
    ResultSet rs = cstmt.executeQuery();
}
```

NOTE

In this example, a result set is returned with the results of running the stored procedure.

For more information about using the JDBC driver with stored procedures and input parameters, see Using a Stored Procedure with Input Parameters.

Retrieving Parameters from a Stored Procedure

If you have to retrieve parameters back from a stored procedure, you must first register an out parameter by name or index by using the registerOutParameter method of the SQLServerCallableStatement class, and then assign the returned out parameter to an appropriate variable after you run the call to the stored procedure. In the following example, the prepareCall method is used to set up the call to the stored procedure, the registerOutParameter method is used to set up the out parameter, and then the setString method is used to set the parameter for the call before executeQuery method is called. The value that is returned by the out parameter of the stored procedure is retrieved by using the getShort method.

```
try(CallableStatement cstmt = con.prepareCall("{call employee_jobid (?, ?)}");) {
    cstmt.registerOutParameter(2, java.sql.Types.SMALLINT);
    String lname = "Brown";
    cstmt.setString(1, lname);
    ResultSet rs = cstmt.executeQuery();
    short empJobID = cstmt.getShort(2);
}
```

NOTE

In addition to the returned out parameter, a result set might also be returned with the results of running the stored procedure.

For more information about how to use the JDBC driver with stored procedures and output parameters, see Using a Stored Procedure with Output Parameters.

See Also

Understanding the JDBC Driver Data Types

Using Spatial Datatypes

8/8/2018 • 5 minutes to read • Edit Online

Download JDBC Driver

Spatial datatypes (Geometry and Geography) are supported starting JDBC Driver preview release 6.5.0. Spatial datatypes are currently not supported with stored procedures, Table Valued Parameters (TVP), BulkCopy, and Always Encrypted. This page shows various use cases of Geometry and Geography data types with the JDBC Driver. For an overview on spatial datatypes, check Spatial Data Types Overview page.

Creating a Geometry / Geography object

There are two main ways to create a Geometry / Geography object - either convert from a Well-Known Text (WKT) or a Well-Known Binary (WKB).

Creating from WKT

```
String geoWKT = "LINESTRING(1 0, 0 1, -1 0)";
Geometry geomWKT = Geometry.STGeomFromText(geoWKT, 0);
Geography geogWKT = Geography.STGeomFromText(geoWKT, 4326);
```

This will create a LINESTRING Geometry object with Spatial Reference System Identifier (SRID) 0, and a Geography object with SRID 4326.

Creating from WKB

```
byte[] geomWKB =
Hex.decodeHex("00000000010403000000000000000000F03F00000000000000000000000000000000000000000000F03F00000000000
0F0BF0000000000000000010000000100000000010000000FFFFFFFF0000000002".toCharArray());
byte[] geogWKB =
Hex.decodeHex("E61000000104030000000000000000000000000000000000F03F000000000000F03F000000000000000000000000000
00000000000000000F0BF01000000010000000001000000FFFFFFFF0000000002".toCharArray());

Geometry geomWKT = Geometry.deserialize(geomWKB);
Geography geogWKT = Geography.deserialize(geogWKB);
```

This will create a Geometry and Geography object that is equivalent to the ones created from the WKT previously.

Working with a Geometry / Geography object

Assuming the user has a table on SQL Server like below:

```
CREATE TABLE sampleTable (c1 geometry)
```

A sample script to insert a Geometry value would be:

```
String geoWKT = "LINESTRING(1 0, 0 1, -1 0)";
Geometry geomWKT = Geometry.STGeomFromText(geoWKT, 0);
SQLServerPreparedStatement pstmt = (SQLServerPreparedStatement) connection.prepareStatement("insert into
sampleTable values (?)");
pstmt.setGeometry(1, geomWKT);
pstmt.execute();
```

The same can be done for the Geography counterpart, using a Geography column and **setGeography()** method.

To read a Geometry / Geography column:

```
try(SQLServerResultSet rs = (SQLServerResultSet)stmt.executeQuery("select * from geomTable")) {
    while(rs.next()){
        rs.getGeometry(1);
    }
}
```

The same can be done for the Geography counterpart, using a Geography column and **getGeography()** method.

Newly introduced APIs

These are the new public APIs that have been introduced with this addition, in the classes **SQLServerPreparedStatement**, **SQLServerResultSet**, **Geometry**, and **Geography**.

SQLServerPreparedStatement

METHOD	DESCRIPTION
void setGeometry(int n, Geometry x)	Sets the designated parameter to the given microsoft.sql.Geometry Class object.
void setGeography(int n, Geography x)	Sets the designated parameter to the given microsoft.sql.Geography Class object.

SQLServerResultSet

METHOD	DESCRIPTION
Geometry getGeometry(int colunIndex)	Returns the value of the designated column in the current row of this ResultSet object as a com.microsoft.sqlserver.jdbc.Geometry object in the Java programming language.
Geometry getGeometry(String columnName)	Returns the value of the designated column in the current row of this ResultSet object as a com.microsoft.sqlserver.jdbc.Geometry object in the Java programming language.
Geography getGeography(int colunIndex)	Returns the value of the designated column in the current row of this ResultSet object as a com.microsoft.sqlserver.jdbc.Geography object in the Java programming language.
Geography getGeography(String columnName)	Returns the value of the designated column in the current row of this ResultSet object as a com.microsoft.sqlserver.jdbc.Geography object in the Java programming language.

Geometry

METHOD	DESCRIPTION

METHOD	DESCRIPTION
Geometry STGeomFromText(String wkt, int SRID)	Constructor for a Geometry instance from an Open Geospatial Consortium (OGC) Well-Known Text (WKT) representation augmented with any Z (elevation) and M (measure) values carried by the instance.
Geometry STGeomFromWKB(byte[] wkb)	Constructor for a Geometry instance from an Open Geospatial Consortium (OGC) Well-Known Binary (WKB) representation.
Geometries deserialize(byte[] wkb)	Constructor for a Geometry instance from an internal SQL Server format for spatial data.
Geometry parse(String wkt)	Constructor for a Geometry instance from an Open Geospatial Consortium (OGC) Well-Known Text (WKT) representation. Spatial Reference Identifier is defaulted to 0.
Geometry point(double x, double y, int SRID)	Constructor for a Geometry instance that represents a Point instance from its X and Y values and a Spatial Reference Identifier.
String STAsText()	Returns the Open Geospatial Consortium (OGC) Well-Known Text (WKT) representation of a Geometry instance. This text will not contain any Z (elevation) or M (measure) values carried by the instance.
byte[] STAsBinary()	Returns the Open Geospatial Consortium (OGC) Well-Known Binary (WKB) representation of a Geometry instance. This value will not contain any Z or M values carried by the instance.
byte[] serialize()	Returns the bytes that represent an internal SQL Server format of Geometry type.
boolean hasM()	Returns if the object contains an M (measure) value.
boolean hasZ()	Returns if the object contains a Z (elevation) value.
Double getX()	Returns the X coordinate value.
Double getY()	Returns the Y coordinate value.
Double getM()	Returns the M (measure) value of the object.
Double getZ()	Returns the Z (elevation) value of the object.
int getSrid()	Returns the Spatial Reference Identifier (SRID) value.
boolean isNull()	Returns if the Geometry object is null.
int STNumPoints()	Returns the number of points in the Geometry object.
String STGeometryType()	Returns the Open Geospatial Consortium (OGC) type name represented by a geometry instance.

METHOD	DESCRIPTION
String asTextZM()	Returns the Well-Known Text (WKT) representation of the Geometry object.
String toString()	Returns the String representation of the Geometry object.

Geography

METHOD	DESCRIPTION
Geography STGeomFromText(String wkt, int SRID)	Constructor for a Geography instance from an Open Geospatial Consortium (OGC) Well-Known Text (WKT) representation augmented with any Z (elevation) and M (measure) values carried by the instance.
Geography STGeomFromWKB(byte[] wkb)	Constructor for a Geography instance from an Open Geospatial Consortium (OGC) Well-Known Binary (WKB) representation.
Geography deserialize(byte[] wkb)	Constructor for a Geography instance from an internal SQL Server format for spatial data.
Geography parse(String wkt)	Constructor for a Geography instance from an Open Geospatial Consortium (OGC) Well-Known Text (WKT) representation. Spatial Reference Identifier is defaulted to 0.
Geography point(double lat, double lon, int SRID)	Constructor for a Geography instance that represents a Point instance from its latitude and longitude values and a Spatial Reference Identifier.
String STAsText()	Returns the Open Geospatial Consortium (OGC) Well-Known Text (WKT) representation of a Geography instance. This text will not contain any Z (elevation) or M (measure) values carried by the instance.
byte[] STAsBinary())	Returns the Open Geospatial Consortium (OGC) Well-Known Binary (WKB) representation of a Geography instance. This value will not contain any Z or M values carried by the instance.
byte[] serialize()	Returns the bytes that represent an internal SQL Server format of Geography type.
boolean hasM()	Returns if the object contains an M (measure) value.
boolean hasZ()	Returns if the object contains a Z (elevation) value.
Double getLatitude()	Returns the latitude value.
Double getLongitude()	Returns the longitude value.
Double getM()	Returns the M (measure) value of the object.
Double getZ()	Returns the Z (elevation) value of the object.

METHOD	DESCRIPTION
int getSrid()	Returns the Spatial Reference Identifier (SRID) value.
boolean isNull()	Returns if the Geography object is null.
int STNumPoints()	Returns the number of points in the Geography object.
String STGeographyType()	Returns the Open Geospatial Consortium (OGC) type name represented by a geography instance.
String asTextZM()	Returns the Well-Known Text (WKT) representation of the Geography object.
String toString()	Returns the String representation of the Geography object.

Limitations of Spatial Datatypes

1. The spatial sub-datatypes **CircularString**, **CompoundCurve**, **CurvePolygon**, and **FullGlobe** are only supported starting from SQL Server 2012 and above.
2. Always Encrypted cannot be used with spatial datatypes.
3. Stored procedures, TVP, and BulkCopy operations are currently not supported with spatial datatypes.

See Also

Spatial Data Types Sample (JDBC)

Understanding Transactions

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Transactions are groups of operations that are combined into logical units of work. They are used to control and maintain the consistency and integrity of each action in a transaction, despite errors that might occur in the system.

With the Microsoft JDBC Driver for SQL Server, transactions can be either local or distributed. Transactions can also use isolation levels. For more information about the isolation levels supported by the JDBC driver, see Understanding Isolation Levels.

Applications should control transactions by either using Transact-SQL statements or the methods provided by the JDBC driver, but not both. Using both Transact-SQL statements and JDBC API methods on the same transaction might lead to problems, such as a transaction cannot be committed when expected, a transaction is committed or rolled back and a new one starts unexpectedly, or "Failed to resume the transaction" exceptions.

Using Local Transactions

A transaction is considered to be local when it is a single-phase transaction, and it is handled by the database directly. The JDBC driver supports local transactions by using various methods of the SQLServerConnection class, including setAutoCommit, commit, and rollback. Local transactions are typically managed explicitly by the application or automatically by the Java Platform, Enterprise Edition (Java EE) application server.

The following example performs a local transaction that consists of two separate statements in the `try` block. The statements are run against the Production.ScrapReason table in the AdventureWorks sample database, and they are committed if no exceptions are thrown. The code in the `catch` block rolls back the transaction if an exception is thrown.

```
public static void executeTransaction(Connection con) {
    try {
        //Switch to manual transaction mode by setting
        //autocommit to false. Note that this starts the first
        //manual transaction.
        con.setAutoCommit(false);
        Statement stmt = con.createStatement();
        stmt.executeUpdate("INSERT INTO Production.ScrapReason(Name) VALUES('Wrong size')");
        stmt.executeUpdate("INSERT INTO Production.ScrapReason(Name) VALUES('Wrong color')");
        con.commit(); //This commits the transaction and starts a new one.
        stmt.close(); //This turns off the transaction.
        System.out.println("Transaction succeeded. Both records were written to the database.");
    }
    catch (SQLException ex) {
        ex.printStackTrace();
        try {
            System.out.println("Transaction failed.");
            con.rollback();
        }
        catch (SQLException se) {
            se.printStackTrace();
        }
    }
}
```

Using Distributed Transactions

A distributed transaction updates data on two or more networked databases while retaining the important atomic, consistent, isolated, and durable (ACID) properties of transaction processing. Distributed transaction support was added to the JDBC API in the JDBC 2.0 Optional API specification. The management of distributed transactions is typically performed automatically by the Java Transaction Service (JTS) transaction manager in a Java EE application server environment. However, the Microsoft JDBC Driver for SQL Server supports distributed transactions under any Java Transaction API (JTA) compliant transaction manager.

The JDBC driver seamlessly integrates with Microsoft Distributed Transaction Coordinator (MS DTC) to provide true distributed transaction support with SQL Server. MS DTC is a distributed transaction facility provided by Microsoft for Microsoft Windows systems. MS DTC uses proven transaction processing technology from Microsoft to support XA features such as the complete two-phase distributed commit protocol and the recovery of distributed transactions.

For more information about how to use distributed transactions, see Understanding XA Transactions.

See Also

Performing Transactions with the JDBC Driver

Working with a Connection

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The following sections provide examples of the different ways to connect to a SQL Server database by using the SQLServerConnection class of the Microsoft JDBC Driver for SQL Server.

NOTE

If you have problems connecting to SQL Server using the JDBC driver, see Troubleshooting Connectivity for suggestions on how to correct it.

Creating a Connection by Using the DriverManager Class

The simplest approach to creating a connection to a SQL Server database is to load the JDBC driver and call the getConnection method of the DriverManager class, as in the following:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
String connectionUrl = "jdbc:sqlserver://localhost;database=AdventureWorks;integratedSecurity=true;"
Connection con = DriverManager.getConnection(connectionUrl);
```

This technique will create a database connection using the first available driver in the list of drivers that can successfully connect with the given URL.

NOTE

When using the sqljdbc4.jar class library, applications do not need to explicitly register or load the driver by using the Class.forName method. When the getConnection method of the DriverManager class is called, an appropriate driver is located from the set of registered JDBC drivers. For more information, see Using the JDBC Driver.

Creating a Connection by Using the SQLServerDriver Class

If you have to specify a particular driver in the list of drivers for DriverManager, you can create a database connection by using the connect method of the SQLServerDriver class, as in the following:

```
Driver d = (Driver) Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver").newInstance();
String connectionUrl = "jdbc:sqlserver://localhost;database=AdventureWorks;integratedSecurity=true;"
Connection con = d.connect(connectionUrl, new Properties());
```

Creating a Connection by Using the SQLServerDataSource Class

If you have to create a connection by using the SQLServerDataSource class, you can use various setter methods of the class before you call the getConnection method, as in the following:

```
SQLServerDataSource ds = new SQLServerDataSource();
ds.setUser("MyUserName");
ds.setPassword("*****");
ds.setServerName("localhost");
ds.setPortNumber(1433);
ds.setDatabaseName("AdventureWorks");
Connection con = ds.getConnection();
```

Creating a Connection that Targets a Very Specific Data Source

If you have to make a database connection that targets a very specific data source, there are a number of approaches that you can take. Each approach depends on the properties that you set by using the connection URL.

To connect to the default instance on a remote server, use the following:

```
String url = "jdbc:sqlserver://MyServer;integratedSecurity=true;"
```

To connect to a specific port on a server, use the following:

```
String url = "jdbc:sqlserver://MyServer:1533;integratedSecurity=true;"
```

To connect to a named instance on a server, use the following:

```
String url = "jdbc:sqlserver://209.196.43.19;instanceName=INSTANCE1;integratedSecurity=true;"
```

To connect to a specific database on a server, use the following:

```
String url = "jdbc:sqlserver://172.31.255.255;database=AdventureWorks;integratedSecurity=true;"
```

For more connection URL examples, see Building the Connection URL.

Creating a Connection with a Custom Login Time-out

If you have to adjust for server load or network traffic, you can create a connection that has a specific login time-out value described in seconds, as in the following:

```
String url = "jdbc:sqlserver://MyServer;loginTimeout=90;integratedSecurity=true;"
```

Create a Connection with Application-level Identity

If you have to use logging and profiling, you will have to identify your connection as originating from a specific application, as in the following:

```
String url = "jdbc:sqlserver://MyServer;applicationName=MYAPP.EXE;integratedSecurity=true;"
```

Closing a Connection

You can explicitly close a database connection by calling the close method of the SQLServerConnection class, as in the following:

```
con.close();
```

This will release the database resources that the SQLServerConnection object is using, or return the connection to the connection pool in pooled scenarios.

NOTE

Calling the close method will also roll back any pending transactions.

See Also

Connecting to SQL Server with the JDBC Driver

International Features of the JDBC Driver

8/13/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

The internationalization features of the Microsoft JDBC Driver for SQL Server include the following:

- Support for a fully localized experience in the same languages as SQL Server
- Support for the Java language conversions for locale sensitive SQL Server data
- Support for international languages, regardless of operating system
- Support for international domain names (starting with Microsoft JDBC Driver 6.0 for SQL Server)

Handling of Character Data

Character data in Java is handled as Unicode by default; the Java **String** object represents Unicode character data. In the JDBC driver, the only exception to this rule is the ASCII stream getter and setter methods, which are special cases because they use byte streams with the implicit assumption of single well-known code pages (ASCII).

In addition, the JDBC driver provides the **sendStringParametersAsUnicode** connection string property. This property can be used to specify that prepared parameters for character data are sent as ASCII or Multi-byte Character Set (MBCS) instead of Unicode. For more information about the **sendStringParametersAsUnicode** connection string property, see Setting the Connection Properties.

Driver Incoming Conversions

Unicode text data coming from the server does not have to be converted. It is passed directly as Unicode. Non-Unicode data coming from the server is converted from the code page for the data, at the database or column level, to Unicode. The JDBC driver uses the Java Virtual Machine (JVM) conversion routines to perform these conversions. These conversions are performed on all typed String and Character stream getter methods.

If the JVM does not have the proper code page support for the data from the database, the JDBC driver throws a "XXX codepage not supported by the Java environment" exception. To solve this problem, you should install the full international character support required for that JVM. For an example, see the Supported Encodings documentation on Sun Microsystems Web site.

Driver Outgoing Conversions

Character data going from the driver to the server can be ASCII or Unicode. For example, the new JDBC 4.0 national character methods, such as setNString, setNCharacterStream, and setNClob methods of SQLServerPreparedStatement and SQLServerCallableStatement classes, always send their parameter values to the server in Unicode.

On the other hand, the non-national character API methods, such as setString, setCharacterStream, and setClob methods of SQLServerPreparedStatement and SQLServerCallableStatement classes send their values to the server in Unicode only when the **sendStringParametersAsUnicode** property is set to "true", which is the default value.

Non-Unicode Parameters

For optimal performance with **CHAR**, **VARCHAR** or **LONGVARCHAR** type of non-Unicode parameters, set the **sendStringParametersAsUnicode** connection string property to "false" and use the non-national character methods.

Formatting Issues

For date, time, and currencies, all formatting with localized data is performed at the Java language level using the Locale object; and the various formatting methods for **Date**, **Calendar**, and **Number** data types. In the rare case where the JDBC driver must pass along locale sensitive data in a localized format, the proper formatter is used with the default JVM locale.

Collation Support

The JDBC Driver 3.0 supports all the collations supported by SQL Server 2000 (8.x), SQL Server 2005 (9.x), and the new collations or new versions of Windows collation names introduced in SQL Server 2008.

For more information on the collations, see Collation and Unicode Support and Windows Collation Name (Transact-SQL) in SQL Server Books Online.

Using International Domain Names (IDN)

The JDBC Driver 6.0 for SQL Server supports the use of Internationalized Domain Names (IDNs) and can convert a Unicode serverName to ASCII compatible encoding (Punycode) when required during a connection. If the IDNs are stored in the Domain Name System (DNS) as ASCII strings in the Punycode format (specified by RFC 3490), enable the conversion of the Unicode server name by setting the serverNameAsACE property to true. Otherwise, if the DNS service is configured to allow the use of Unicode characters, set the serverNameAsACE property as false (the default). For older versions of the JDBC driver, it is also possible to convert the serverName to Punycode using Java's IDN.toASCII methods before setting that property for a connection.

NOTE

Most resolver software written for non-Windows platforms is based on the Internet DSN standards and is therefore most likely to use the Punycode format for IDNs, while a Windows-based DNS Server on a private network can be configured to allow the use of UTF-8 characters on a per-server basis. For more details, see Unicode character support.

See Also

Overview of the JDBC Driver

Using Advanced Data Types

8/13/2018 • 6 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server uses the JDBC advanced data types to convert the SQL Server data types to a format that can be understood by the Java programming language.

Remarks

The following table lists the default mappings between the advanced SQL Server, JDBC, and Java programming language data types.

SQL SERVER TYPES	JDBC TYPES (JAVA.SQL.TYPES)	JAVA LANGUAGE TYPES
varbinary(max) image	LONGVARBINARY	byte[] (default), Blob, InputStream, String
text varchar(max)	LONGVARCHAR	String (default), Clob, InputStream
ntext nvarchar(max)	LONGVARCHAR LONGNVARCHAR (Java SE 6.0)	String (default), Clob, NClob (Java SE 6.0)
xml	LONGVARCHAR SQLXML (Java SE 6.0)	String (default), InputStream, Clob, byte[], Blob, SQLXML (Java SE 6.0)
Udt[1]	VARBINARY	String (default), byte[], InputStream

[1] The Microsoft JDBC Driver for SQL Server supports sending and retrieving CLR UDTs as binary data but doesn't support manipulation of the CLR metadata.

The following sections provide examples of how you can use the JDBC driver and the advanced data types.

BLOB and CLOB and NCLOB Data Types

The JDBC driver implements all the methods of the java.sql.Blob, java.sql.Clob, and java.sql.NClob interfaces.

NOTE

CLOB values can be used with SQL Server 2005 (9.x) (or later) large-value data types. Specifically, CLOB types can be used with the **varchar(max)** and **nvarchar(max)** data types, BLOB types can be used with **varbinary(max)** and **image** data types, and NCLOB types can be used with **ntext** and **nvarchar(max)**.

Large Value Data Types

In earlier versions of SQL Server, working with large-value data types required special handling. Large-value data types are those that exceed the maximum row size of 8 KB. SQL Server introduces a max specifier for **varchar**, **nvarchar**, and **varbinary** data types to allow storage of values as large as 2^31 bytes. Table columns and Transact-SQL variables can specify **varchar(max)**, **nvarchar(max)**, or **varbinary(max)** data types.

The primary scenarios for working with large-value types involve retrieving them from a database, or adding them to a database. The following sections describe different approaches to accomplish these tasks.

Retrieving Large-Value Types from a Database

When you retrieve a non-binary large-value data type—such as the **varchar(max)** data type—from a database, one approach is to read that data as a character stream. In the following example, the executeQuery method of the SQLServerStatement class is used to retrieve data from the database and return it as a result set. Then the getCharacterStream method of the SQLServerResultSet class is used to read the large-value data from the result set.

```
ResultSet rs = stmt.executeQuery("SELECT TOP 1 * FROM Test1");
rs.next();
Reader reader = rs.getCharacterStream(2);
```

NOTE

This same approach can also be used for the **text**, **ntext**, and **nvarchar(max)** data types.

When you retrieve a binary large-value data type—such as the **varbinary(max)** data type—from a database, there are several approaches that you can take. The most efficient approach is to read the data as a binary stream, as in the following:

```
ResultSet rs = stmt.executeQuery("SELECT photo FROM mypics");
rs.next();
InputStream is = rs.getBinaryStream(2);
```

You can also use the getBytes method to read the data as a byte array, as in the following:

```
ResultSet rs = stmt.executeQuery("SELECT photo FROM mypics");
rs.next();
byte [] b = rs.getBytes(2);
```

NOTE

You can also read the data as a BLOB. However, this is less efficient than the two methods shown previously.

Adding Large-Value Types to a Database

Uploading large data with the JDBC driver works well for the memory-sized cases, and in the larger-than-memory cases, streaming is the primary option. However, the most efficient way to upload large data is through the stream interfaces.

Using a String or bytes is also an option, as in the following:

```
PreparedStatement pstmt = con.prepareStatement("INSERT INTO test1 (c1_id, c2_vcmax) VALUES (?, ?)");
pstmt.setInt(1, 1);
pstmt.setString(2, htmlStr);
pstmt.executeUpdate();
```

NOTE

This approach can also be used for values that are stored in **text**, **ntext**, and **nvarchar(max)** columns.

If you have an image library on the server and must upload entire binary image files to a **varbinary(max)** column, the most efficient method with the JDBC driver is to use streams directly, as in the following:

```
try (PreparedStatement pstmt = con.prepareStatement("INSERT INTO test1 (Col1, Col2) VALUES(?,?)")) {
  File inputFile = new File("CLOBFile20mb.jpg");
  try (FileInputStream inStream = new FileInputStream(inputFile)) {
    int id = 1;
    pstmt.setInt(1,id);
    pstmt.setBinaryStream(2, inStream);
    pstmt.executeUpdate();
  }
}
```

NOTE

Using either the CLOB or BLOB method is not an efficient way to upload large data.

Modifying Large-Value Types in a Database

In most cases, the recommended method for updating or modifying large values on the database is to pass parameters through the SQLServerPreparedStatement and SQLServerCallableStatement classes by using Transact-SQL commands like `UPDATE`, `WRITE`, and `SUBSTRING`.

If you have to replace the instance of a word in a large text file, such as an archived HTML file, you can use a Clob object, as in the following:

```
String SQL = "SELECT * FROM test1;";
try (Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)
     ResultSet rs = stmt.executeQuery(SQL)) {
  rs.next();

  Clob clob = rs.getClob(2);
  long pos = clob.position("dog", 1);
  clob.setString(pos, "cat");
  rs.updateClob(2, clob);
  rs.updateRow();
}
```

Additionally, you could do all the work on the server and just pass parameters to a prepared UPDATE statement.

For more information about large-value types, see "Using Large-Value Types" in SQL Server Books Online.

XML Data Type

SQL Server provides an **xml** data type that lets you store XML documents and fragments in a SQL Server database. The **xml** data type is a built-in data type in SQL Server, and is in some ways similar to other built-in types, such as **int** and **varchar**. As with other built-in types, you can use the **xml** data type as a column type when you create a table; as a variable type, a parameter type, or a function-return type; or in Transact-SQL CAST and CONVERT functions.

In the JDBC driver, the **xml** data type can be mapped as a String, byte array, stream, CLOB, BLOB, or SQLXML object. String is the default. Starting with the JDBC Driver version 2.0, the JDBC driver provides support for the JDBC 4.0 API, which introduces the SQLXML interface. The SQLXML interface defines methods to interact and manipulate XML data. The **SQLXML** data type maps to the SQL Server**xml** data type. For more information about how to read and write XML data from and to the relational database with the **SQLXML** Java data type, see Supporting XML Data.

The implementation of the **xml** data type in the JDBC driver provides support for the following:

- Access to the XML as a standard Java UTF-16 string for most common programming scenarios
- Input of UTF-8 and other 8-bit encoded XML
- Access to the XML as a byte array with a leading BOM when encoded in UTF-16 for interchange with other XML processors and disk files

SQL Server requires a leading BOM for UTF-16-encoded XML. The application must provide this when XML parameter values are supplied as byte arrays. SQL Server always outputs XML values as UTF-16 strings with no BOM or embedded encoding declaration. When XML values are retrieved as byte[], BinaryStream or Blob, a UTF-16 BOM is pre-pended to the value.

For more information about the **xml** data type, see "xml Data Type" in SQL Server Books Online.

User-Defined Data Type

The introduction of user-defined types (UDTs) in SQL Server 2005 (9.x) extends the SQL type system by letting you store objects and custom data structures in a SQL Server database. UDTs can contain multiple data types and can have behaviors, differentiating them from the traditional alias data types that consist of a single SQL Server system data type. UDTs are defined by using any of the languages supported by the Microsoft .NET common language runtime (CLR) that produce verifiable code. This includes Microsoft Visual C# and Visual Basic .NET. The data is exposed as fields and properties of a .NET Framework-based class or structure, and behaviors are defined by methods of the class or structure.

In SQL Server, a UDT can be used as the column definition of a table, as a variable in a Transact-SQL batch, or as an argument of a Transact-SQL function or stored procedure.

For more information about user-defined data types, see "Using and Modifying Instances of User-defined Types" in SQL Server Books Online.

See Also

Understanding the JDBC Driver Data Types

Understanding Data Type Differences

8/13/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

There are a number of differences between the Java programming language data types and SQL Server data types. The Microsoft JDBC Driver for SQL Server helps to facilitate those differences through various types of conversions.

Character Types

The JDBC character string data types are **CHAR**, **VARCHAR**, and **LONGVARCHAR**. The JDBC driver provides support for the JDBC 4.0 API. In the JDBC 4.0, the JDBC character string data types can also be **NCHAR**, **NVARCHAR**, and **LONGNVARCHAR**. These new character string types maintain Java native character types in Unicode format and remove the need to perform any ANSI-to-Unicode or Unicode-to-ANSI conversion.

TYPE	DESCRIPTION
Fixed-length	The SQL Server **char** and **nchar** data types map directly to the JDBC **CHAR** and **NCHAR** types. These are fixed-length types with padding provided by the server in the case where the column has `SET ANSI_PADDING ON`. Padding is always turned on for **nchar**, but for **char**, in the case where the server char columns are not padded, the JDBC driver adds the padding.
Variable-length	The SQL Server **varchar** and **nvarchar** types map directly to the JDBC **VARCHAR** and **NVARCHAR** types, respectively.
Long	The SQL Server **text** and **ntext** types map to the JDBC **LONGVARCHAR** and **LONGNVARCHAR** type, respectively. These are deprecated types beginning in SQL Server 2005 (9.x), so you should use large value types, **varchar(max)** or **nvarchar(max)**, instead. Using the update<Numeric Type> and updateObject (int, java.lang.Object) methods will fail against **text** and **ntext** server columns. However, using the setObject method with a specified character conversion type is supported against **text** and **ntext** server columns.

Binary String Types

The JDBC binary-string types are **BINARY**, **VARBINARY**, and **LONGVARBINARY**.

TYPE	DESCRIPTION

TYPE	DESCRIPTION
Fixed-length	The SQL Server **binary** type maps directly to the JDBC **BINARY** type. This is a fixed-length type with padding provided by the server in the case where the column has SET ANSI_PADDING ON. When the server char columns are not padded, the JDBC driver adds the padding. The SQL Server **timestamp** type is a JDBC **BINARY** type with the fixed length of 8 bytes.
Variable-length	The SQL Server **varbinary** type maps to the JDBC **VARBINARY** type. The **udt** type in SQL Server maps to JDBC as a **VARBINARY** type.
Long	The SQL Server **image** type maps to the JDBC **LONGVARBINARY** type. This type is deprecated beginning in SQL Server 2005 (9.x), so you should use a large value type, **varbinary(max)** instead.

Exact Numeric Types

The JDBC exact numeric types map directly to their corresponding SQL Server types.

TYPE	DESCRIPTION
BIT	The JDBC **BIT** type represents a single bit that can be 0 or 1. This maps to a SQL Server **bit** type.
TINYINT	The JDBC **TINYINT** type represents a single byte. This maps to a SQL Server **tinyint** type.
SMALLINT	The JDBC **SMALLINT** type represents a signed 16-bit integer. This maps to a SQL Server **smallint** type.
INTEGER	The JDBC **INTEGER** type represents a signed 32-bit integer. This maps to a SQL Server **int** type.
BIGINT	The JDBC **BIGINT** type represents a signed 64-bit integer. This maps to a SQL Server **bigint** type.
NUMERIC	The JDBC **NUMERIC** type represents a fixed-precision decimal value that holds values of identical precision. The **NUMERIC** type maps to the SQL Server **numeric** type.
DECIMAL	The JDBC **DECIMAL** type represents a fixed-precision decimal value that holds values of at least the specified precision. The **DECIMAL** type maps to the SQL Server **decimal** type. The JDBC **DECIMAL** type also maps to the SQL Server **money** and **smallmoney** types, which are specific fixed-precision decimal types that are stored in 8 and 4 bytes, respectively.

Approximate Numeric Types

The JDBC approximate numeric types are **REAL**, **DOUBLE**, and **FLOAT**.

TYPE	DESCRIPTION
REAL	The JDBC **REAL** type has seven digits of precision (single precision) and maps directly to the SQL Server **real** type.
DOUBLE	The JDBC **DOUBLE** type has 15 digits of precision (double precision) and maps to the SQL Server **float** type. The JDBC **FLOAT** type is a synonym of **DOUBLE**. Because there can be confusion between **FLOAT** and **DOUBLE**, **DOUBLE** is preferred.

Datetime Types

The JDBC **TIMESTAMP** type maps to the SQL Server **datetime** and **smalldatetime** types. The **datetime** type is stored in two 4-byte integers. The **smalldatetime** type holds the same information (date and time), but with less accuracy, in two 2-byte small integers.

NOTE

The SQL Server **timestamp** type is a fixed-length binary-string type. It does not map to any of the JDBC time types: **DATE**, **TIME**, or **TIMESTAMP**.

Custom Type Mapping

The custom type mapping feature of JDBC that uses the SQLData interfaces for the JDBC advanced types (UDTs, Struct, and so on). is not implemented in the JDBC driver.

See Also

Understanding the JDBC Driver Data Types

Using Multiple Result Sets

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When working with inline SQL or SQL Server stored procedures that return more than one result set, the Microsoft JDBC Driver for SQL Server provides the getResultSet method in the SQLServerStatement class for retrieving each set of data returned. In addition, when running a statement that returns more than one result set, you can use the execute method of the SQLServerStatement class, because it will return a **boolean** value that indicates if the value returned is a result set or an update count.

If the execute method returns **true**, the statement that was run has returned one or more result sets. You can access the first result set by calling the getResultSet method. To determine if more result sets are available, you can call the getMoreResults method, which returns a **boolean** value of **true** if more result sets are available. If more result sets are available, you can call the getResultSet method again to access them, continuing the process until all result sets have been processed. If the getMoreResults method returns **false**, there are no more result sets to process.

If the execute method returns **false**, the statement that was run has returned an update count value, which you can retrieve by calling the getUpdateCount method.

NOTE

For more information about update counts, see Using a Stored Procedure with an Update Count.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, and an SQL statement is constructed that, when run, returns two result sets:

```
public static void executeStatement(Connection con) {
    try (Statement stmt = con.createStatement();) {
        String SQL = "SELECT TOP 10 * FROM Person.Contact; SELECT TOP 20 * FROM Person.Contact";

        boolean results = stmt.execute(SQL);
        int rsCount = 0;

        // Loop through the available result sets.
        do {
            if (results) {
                ResultSet rs = stmt.getResultSet();
                rsCount++;

                // Show data from the result set.
                System.out.println("RESULT SET #" + rsCount);
                while (rs.next()) {
                    System.out.println(rs.getString("LastName") + ", " + rs.getString("FirstName"));
                }
            }
            System.out.println();
            results = stmt.getMoreResults();
        } while (results);
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

In this case, the number of result sets returned is known to be two. However, the code is written so that if an unknown number of result sets were returned, such as when calling a stored procedure, they would all be processed. To see an example of calling a stored procedure that returns multiple result sets along with update values, see Handling Complex Statements.

NOTE

When you make the call to the getMoreResults method of the SQLServerStatement class, the previously returned result set is implicitly closed.

See Also

Using Statements with the JDBC Driver

Using Holdability

8/8/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

By default, a result set created within a transaction is kept open after the transaction is committed to the database, or when it is rolled back. However, it is sometimes useful for the result set to be closed, after the transaction has been committed. To do this, the Microsoft JDBC Driver for SQL Server supports the use of result set holdability.

Result set holdability can be set by using the setHoldability method of the SQLServerConnection class. When setting the holdability by using the setHoldability method, the result set holdability constants of `ResultSet.HOLD_CURSORS_OVER_COMMIT` or `ResultSet.CLOSE_CURSORS_AT_COMMIT` can be used.

The JDBC driver also supports setting holdability when creating one of the Statement objects. When creating the Statement objects that have overloads with result set holdability parameters, the holdability of statement object must match the connection's holdability. When they don't match, an exception is thrown. It's because SQL Server supports the holdability only at the connection level.

The holdability of a result set is the holdability of the SQLServerConnection object that is associated with the result set at the time when the result set is created for server-side cursors only. It does not apply to client-side cursors. All result sets with client-side cursors will always have the holdability value of `ResultSet.HOLD_CURSORS_OVER_COMMIT`.

For server cursors, when connected to SQL Server 2005 or later, setting the holdability affects only the holdability of new result sets that are yet to be created on that connection. It means that setting holdability has no impact on the holdability of any result sets that were previously created and are already open on that connection.

In the following example, the result set holdability is set while performing a local transaction consisting of two separate statements in the `try` block. The statements are run against the Production.ScrapReason table in the AdventureWorks sample database. First, the example switches to manual transaction mode by setting the auto-commit to `false`. Once auto-commit mode is disabled, no SQL Statements will be committed until the application calls the commit method explicitly. The code in the catch block rolls back the transaction if an exception is thrown.

```
public static void executeTransaction(Connection con) {
    try (Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);)
{
        con.setAutoCommit(false);
        con.setHoldability(ResultSet.HOLD_CURSORS_OVER_COMMIT);

        stmt.executeUpdate("INSERT INTO Production.ScrapReason(Name) VALUES('Bad part')");
        ResultSet rs = stmt.executeQuery("SELECT * FROM Production.ScrapReason");
        con.commit();
        System.out.println("Transaction succeeded.");

        // Display results.
        while (rs.next()) {
            System.out.println(rs.getString(2));
        }
    }
    catch (SQLException ex) {
        ex.printStackTrace();
        try {
            System.out.println("Transaction failed.");
            con.rollback();
        }
        catch (SQLException se) {
            se.printStackTrace();
        }
    }
}
```

See Also

Performing Transactions with the JDBC Driver

Connecting using Azure Active Directory Authentication

8/2/2018 • 9 minutes to read • Edit Online

Download JDBC Driver

This article provides information on how to develop Java applications to use the Azure Active Directory authentication feature with Microsoft JDBC Driver 6.0 (or higher) for SQL Server.

You can use Azure Active Directory (AAD) authentication, which is a mechanism of connecting to Azure SQL Database v12 using identities in Azure Active Directory. Use Azure Active Directory authentication to centrally manage identities of database users and as an alternative to SQL Server authentication. The JDBC Driver allows you to specify your Azure Active Directory credentials in the JDBC connection string to connect to Azure SQL DB. For information on how to configure Azure Active Directory authentication visit Connecting to SQL Database By Using Azure Active Directory Authentication.

Two new connection properties have been added to support Azure Active Directory Authentication:

- **authentication**: Use this property to indicate which SQL authentication method to use for connection. Possible values are: **ActiveDirectoryIntegrated**, **ActiveDirectoryPassword**, **SqlPassword**, and the default **NotSpecified**.
  - Use 'authentication=ActiveDirectoryIntegrated' to connect to a SQL Database using integrated Windows authentication. To use this authentication mode, you need to federate the on-premise Active Directory Federation Services (ADFS) with AAD in the cloud. Once this is set up as well as a Kerberos ticket, you can access Azure SQL DB without being prompted for credentials when you're logged in a domain joined machine.
  - Use 'authentication=ActiveDirectoryPassword' to connect to a SQL Database using an Azure AD principal name and password.
  - Use 'authentication=SqlPassword' to connect to a SQL Server using userName/user and password properties.
  - Use 'authentication=NotSpecified' or leave it as default when none of these authentication methods are needed.
- **accessToken**: Use this property to connect to a SQL database using an access token. accessToken can only be set using the Properties parameter of the getConnection() method in the DriverManager class. It can't be used in the connection URL.

For more information, see the authentication property on the Setting the Connection Properties page.

Client Setup Requirements

Please make sure that the following components are installed on the client machine:

- Java 7 or above
- Microsoft JDBC Driver 6.0 (or higher) for SQL Server
- If you're using the access token-based authentication mode, you need azure-activedirectory-library-for-java and its dependencies to run the examples from this article. For more information, see **Connecting using Access Token** section.
- If you're using the ActiveDirectoryPassword authentication mode, you need azure-activedirectory-library-for-java and its dependencies. For more information, see **Connecting using ActiveDirectoryPassword**

Authentication Mode section.

- If you're using the ActiveDirectoryIntegrated mode, you need azure-activedirectory-library-for-java and its dependencies. For more information, see **Connecting using ActiveDirectoryIntegrated Authentication Mode** section.

Connecting using ActiveDirectoryIntegrated Authentication Mode

With version 6.4, Microsoft JDBC Driver adds support for ActiveDirectoryIntegrated Authentication using a Kerberos ticket on multiple platforms (Windows/Linux and Mac). For more information, see Set Kerberos ticket on Windows, Linux And Mac for more details. Alternatively, on Windows, sqljdbc_auth.dll can also be used for ActiveDirectoryIntegrated Authentication with JDBC Driver.

NOTE

If you are using an older version of the driver, check this link for the respective dependencies that are required to use this authentication mode.

The following example shows how to use 'authentication=ActiveDirectoryIntegrated' mode. Run this example on a domain joined machine that is federated with Azure Active Directory. A contained database user representing your Azure AD principal, or one of the groups, you belong to, must exist in the database, and must have the CONNECT permission.

Before building and running the example, on the client machine (on which, you want to run the example), download the azure-activedirectory-library-for-java library and its dependencies, and include them in the Java build path

Replace the server/database name with your server/database name in the following lines before executing the example:

```
ds.setServerName("aad-managed-demo.database.windows.net"); // replace 'aad-managed-demo' with your server name
ds.setDatabaseName("demo"); // replace with your database name
```

The example to use ActiveDirectoryIntegrated authentication mode:

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerDataSource;

public class AADIntegrated {
    public static void main(String[] args) throws Exception {

        SQLServerDataSource ds = new SQLServerDataSource();
        ds.setServerName("aad-managed-demo.database.windows.net"); // Replace with your server name
        ds.setDatabaseName("demo"); // Replace with your database name
        ds.setAuthentication("ActiveDirectoryIntegrated");
        ds.setHostNameInCertificate("*.database.windows.net");

        try (Connection connection = ds.getConnection();
                Statement stmt = connection.createStatement();) {

            ResultSet rs = stmt.executeQuery("SELECT SUSER_SNAME()");
            if (rs.next()) {
                System.out.println("You have successfully logged on as: " + rs.getString(1));
            }
        }
    }
}
```

Running this example on a client machine automatically uses your Kerberos ticket and no password is required. If a connection is established, you should see the following message:

```
You have successfully logged on as: <your domain user name>
```

Set Kerberos ticket on Windows, Linux And Mac

You need to set up a Kerberos ticket linking your current user to a Windows domain account. A summary of key steps is included below.

Windows

JDK comes with `kinit`, which you can use to get a TGT from Key Distribution Center (KDC) on a domain joined machine that is federated with Azure Active Directory.

Step 1: Ticket Granting Ticket retrieval

- **Run on**: Windows
- **Action**:
    - Use the command `kinit username@DOMAIN.COMPANY.COM` to get a TGT from KDC, then it will prompt you for your domain password.
    - Use `klist` to see the available tickets. If the kinit was successful, you should see a ticket from krbtgt/DOMAIN.COMPANY.COM@ DOMAIN.COMPANY.COM.

NOTE

You may need to specify a `.ini` file with `-Djava.security.krb5.conf` for your application to locate KDC.

Linux and Mac

Requirements

Access to a Windows domain-joined machine to query your Kerberos Domain Controller.

Step 1: Find Kerberos KDC

- **Run on**: Windows command line
- **Action**: `nltest /dsgetdc:DOMAIN.COMPANY.COM` (where "DOMAIN.COMPANY.COM" maps to your domain's

name)

- **Sample Output**
  ```
  DC: \\co1-red-dc-33.domain.company.com Address: \\2111:4444:2111:33:1111:ecff:ffff:3333 ... The command completed successfully
  ```
- **Information to extract** The DC name, in this case `co1-red-dc-33.domain.company.com`

Step 2: Configuring KDC in krb5.conf

- **Run on**: Linux/Mac
- **Action**: Edit the /etc/krb5.conf in an editor of your choice. Configure the following keys

  ```
  [libdefaults]
    default_realm = DOMAIN.COMPANY.COM

  [realms]
  DOMAIN.COMPANY.COM = {
     kdc = co1-red-dc-28.domain.company.com
  }
  ```

  Then save the krb5.conf file and exit

NOTE

Domain must be in ALL CAPS.

Step 3: Testing the Ticket Granting Ticket retrieval

- **Run on**: Linux/Mac
- **Action**:
  - Use the command `kinit username@DOMAIN.COMPANY.COM` to get a TGT from KDC, then it will prompt you for your domain password.
  - Use `klist` to see the available tickets. If the kinit was successful, you should see a ticket from krbtgt/DOMAIN.COMPANY.COM@ DOMAIN.COMPANY.COM.

Connecting using ActiveDirectoryPassword Authentication Mode

The following example shows how to use 'authentication=ActiveDirectoryPassword' mode.

Before building and running the example:

1. On the client machine (on which, you want to run the example), download the azure-activedirectory-library-for-java library and its dependencies, and include them in the Java build path
2. Locate the following lines of code and replace the server/database name with your server/database name.
   ```
   java ds.setServerName("aad-managed-demo.database.windows.net"); // replace 'aad-managed-demo' with your server name ds.setDatabaseName("demo"); // replace with your database name
   ```
3. Locate the following lines of code and replace user name, with the name of the AAD user you want to connect as.
   ```
   java ds.setUser("bob@cqclinic.onmicrosoft.com"); // replace with your user name ds.setPassword("password"); // replace with your password
   ```

The example to use ActiveDirectoryPassword authentication mode:

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerDataSource;

public class AADUserPassword {

    public static void main(String[] args) throws Exception{

        SQLServerDataSource ds = new SQLServerDataSource();
        ds.setServerName("aad-managed-demo.database.windows.net"); // Replace with your server name
        ds.setDatabaseName("demo"); // Replace with your database
        ds.setUser("bob@cqclinic.onmicrosoft.com"); // Replace with your user name
        ds.setPassword("password"); // Replace with your password
        ds.setAuthentication("ActiveDirectoryPassword");
        ds.setHostNameInCertificate("*.database.windows.net");

        try (Connection connection = ds.getConnection();
                Statement stmt = connection.createStatement();) {

            ResultSet rs = stmt.executeQuery("SELECT SUSER_SNAME()");
            if (rs.next()) {
                System.out.println("You have successfully logged on as: " + rs.getString(1));
            }
        }
    }
}
```

If connection is established, you should see the following message as output:

```
You have successfully logged on as: <your user name>
```

NOTE

A contained user database must exist and a contained database user representing the specified Azure AD user or one of the groups, the specified Azure AD user belongs to, must exist in the database, and must have the CONNECT permission (except for Azure Active Directory server admin or group)

Connecting using Access Token

Applications/services can retrieve an access token from the Azure Active Directory and use that to connect to SQL Azure Database.

NOTE

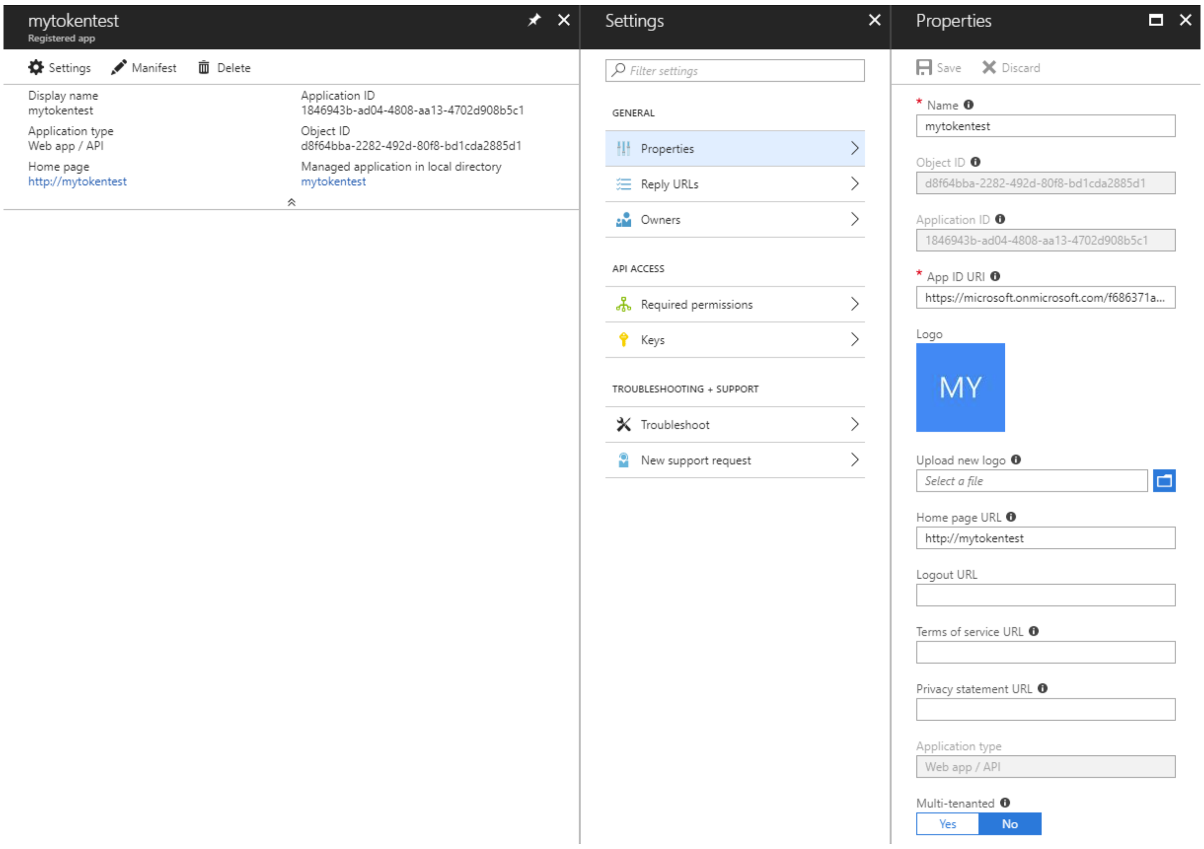
accessToken can only be set using the Properties parameter of the getConnection() method in the DriverManager class. It can't be used in the connection string.

The example below contains a simple Java application that connects to Azure SQL Database using access token-based authentication. Before building and running the example, perform the following steps:

1. Create an application account in Azure Active Directory for your service.

    a. Sign in to the Azure portal.
    b. Click on Azure Active Directory in the left-hand navigation.
    c. Click the "App registrations" tab.

d. In the drawer, click "New application registration".
e. Enter mytokentest as a friendly name for the application, select "Web App/API".
f. We don't need SIGN-ON URL. Just provide anything: "http://mytokentest".
g. Click "Create" at the bottom.
h. While still in the Azure portal, click the "Settings" tab of your application, and open the "Properties" tab.
i. Find the "Application ID" (AKA Client ID) value and copy it aside, you need this later when configuring your application (for example, 1846943b-ad04-4808-aa13-4702d908b5c1). See the following snapshot.
j. Under section “Keys”, create a key by filling in the name field, selecting the duration of the key, and saving the configuration (leave the value field empty). After saving, the value field should be filled automatically, copy the generated value. This is the client Secret.
k. Click Azure Active Directory on the left side panel. Under "App Registrations", find the "End points" tab. Copy the URL under "OATH 2.0 TOKEN ENDPOINT", this is your STS URL.


mytokentest
Registered app
Display name
mytokentest
Application type
Web app / API
Home page
http://mytokentest
Application ID
1846943b-ad04-4808-aa13-4702d908b5c1
Object ID
d8f64bba-2282-492d-80f8-bd1cda2885d1
Managed application in local directory
mytokentest

Properties
Name: mytokentest
Object ID: d8f64bba-2282-492d-80f8-bd1cda2885d1
Application ID: 1846943b-ad04-4808-aa13-4702d908b5c1
App ID URI: https://microsoft.onmicrosoft.com/f686371a...
Home page URL: http://mytokentest
Logout URL:
Terms of service URL:
Privacy statement URL:
Application type: Web app / API
Multi-tenanted: Yes [✓ No]

2. Sign in to your Azure SQL Server’s user database as an Azure Active Directory admin and using a T-SQL command provision a contained database user for your application principal. For more information, see the Connecting to SQL Database or SQL Data Warehouse By Using Azure Active Directory Authentication for more details on how to create an Azure Active Directory admin and a contained database user.

```
CREATE USER [mytokentest] FROM EXTERNAL PROVIDER
```

3. On the client machine (on which, you want to run the example), download the azure-activedirectory-library-for-java library and its dependencies, and include them in the Java build path. Note that the azure-activedirectory-library-for-java is only needed to run this specific example. The example uses the APIs from this library to retrieve the access token from Azure AAD. If you already have an access token, you can skip this step. Note that you also need to remove the section in the example that retrieves access token.

In the following example, replace the STS URL, Client ID, Client Secret, server and database name with your values.

```java
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import com.microsoft.sqlserver.jdbc.SQLServerDataSource;

// The azure-activedirectory-library-for-java is needed to retrieve the access token from the AD.
import com.microsoft.aad.adal4j.AuthenticationContext;
import com.microsoft.aad.adal4j.AuthenticationResult;
import com.microsoft.aad.adal4j.ClientCredential;

public class AADTokenBased {

    public static void main(String[] args) throws Exception {

        // Retrieve the access token from the AD.
        String spn = "https://database.windows.net/";
        String stsurl = "https://login.microsoftonline.com/..."; // Replace with your STS URL.
        String clientId = "1846943b-ad04-4808-aa13-4702d908b5c1"; // Replace with your client ID.
        String clientSecret = "..."; // Replace with your client secret.

        AuthenticationContext context = new AuthenticationContext(stsurl, false,
Executors.newFixedThreadPool(1));
        ClientCredential cred = new ClientCredential(clientId, clientSecret);

        Future<AuthenticationResult> future = context.acquireToken(spn, cred, null);
        String accessToken = future.get().getAccessToken();

        System.out.println("Access Token: " + accessToken);

        // Connect with the access token.
        SQLServerDataSource ds = new SQLServerDataSource();

        ds.setServerName("aad-managed-demo.database.windows.net"); // Replace with your server name.
        ds.setDatabaseName("demo"); // Replace with your database name.
        ds.setAccessToken(accessToken);
        ds.setHostNameInCertificate("*.database.windows.net");

        try (Connection connection = ds.getConnection();
                Statement stmt = connection.createStatement();) {

            ResultSet rs = stmt.executeQuery("SELECT SUSER_SNAME()");
            if (rs.next()) {
                System.out.println("You have successfully logged on as: " + rs.getString(1));
            }
        }
    }
}
```

If the connection is successful, you should see the following message as output:

```
Access Token: <your access token>
You have successfully logged on as: <your client ID>
```

Understanding Data Type Conversions

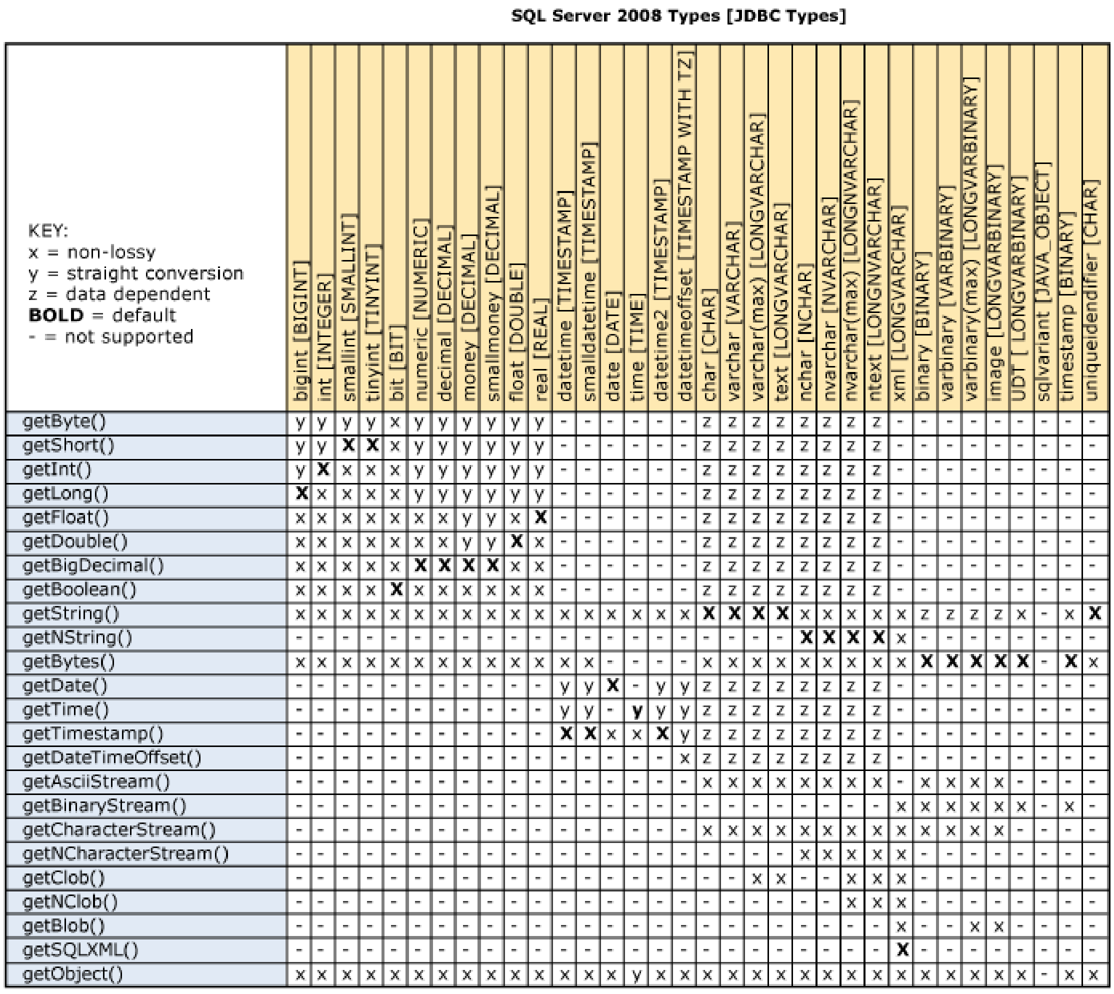
8/13/2018 • 7 minutes to read • Edit Online

Download JDBC Driver

To facilitate the conversion of Java programming language data types to SQL Server data types, the Microsoft JDBC Driver for SQL Server provides data type conversions as required by the JDBC specification. For added flexibility, all types are convertible to and from **Object**, **String**, and **byte[]** data types.

Getter Method Conversions

Based on the SQL Server data types, the following chart contains the JDBC driver's conversion map for the get<Type>() methods of the SQLServerResultSet class, and the supported conversions for the get<Type> methods of the SQLServerCallableStatement class.



SQL Server 2008 Types [JDBC Types]

KEY:
x = non-lossy
y = straight conversion
z = data dependent
BOLD = default
- = not supported

	bigint [BIGINT]	int [INTEGER]	smallint [SMALLINT]	tinyint [TINYINT]	bit [BIT]	numeric [NUMERIC]	decimal [DECIMAL]	money [DECIMAL]	smallmoney [DECIMAL]	float [DOUBLE]	real [REAL]	datetime [TIMESTAMP]	smalldatetime [TIMESTAMP]	date [DATE]	time [TIME]	datetime2 [TIMESTAMP]	datetimeoffset [TIMESTAMP WITH TZ]	char [CHAR]	varchar [VARCHAR]	varchar(max) [LONGVARCHAR]	text [LONGVARCHAR]	nchar [NCHAR]	nvarchar [NVARCHAR]	nvarchar(max) [LONGNVARCHAR]	ntext [LONGNVARCHAR]	xml [LONGVARCHAR]	binary [BINARY]	varbinary [VARBINARY]	varbinary(max) [LONGVARBINARY]	image [LONGVARBINARY]	UDT [LONGVARBINARY]	sqlvariant [JAVA_OBJECT]	timestamp [BINARY]	uniqueidentifier [CHAR]
getByte()	y	y	y	y	x	y	y	y	y	y	y	-	-	-	-	-	-	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getShort()	y	y	**X**	**X**	x	y	y	y	y	y	y	-	-	-	-	-	-	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getInt()	y	**X**	x	x	x	y	y	y	y	y	y	-	-	-	-	-	-	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getLong()	**X**	x	x	x	x	y	y	y	y	y	y	-	-	-	-	-	-	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getFloat()	x	x	x	x	x	x	x	y	y	x	**X**	-	-	-	-	-	-	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getDouble()	x	x	x	x	x	x	x	y	y	**X**	x	-	-	-	-	-	-	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getBigDecimal()	x	x	x	x	x	**X**	**X**	**X**	**X**	x	x	-	-	-	-	-	-	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getBoolean()	x	x	x	x	**X**	x	x	x	x	x	x	-	-	-	-	-	-	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getString()	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	**X**	**X**	**X**	**X**	x	x	x	x	x	z	z	z	z	x	-	x	**X**
getNString()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**X**	**X**	**X**	**X**	x	-	-	-	-	-	-	-	-
getBytes()	x	x	x	x	x	x	x	x	x	x	x	x	x	-	-	-	-	x	x	x	x	x	x	x	x	x	**X**	**X**	**X**	**X**	**X**	-	**X**	x
getDate()	-	-	-	-	-	-	-	-	-	-	-	y	y	**X**	-	y	y	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getTime()	-	-	-	-	-	-	-	-	-	-	-	y	y	-	**y**	y	y	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getTimestamp()	-	-	-	-	-	-	-	-	-	-	-	**X**	**X**	x	x	**X**	y	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getDateTimeOffset()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	z	z	z	z	z	z	z	z	-	-	-	-	-	-	-	-	-
getAsciiStream()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	x	x	x	x	x	x	-	x	x	x	x	-	-	-	-
getBinaryStream()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	x	x	x	x	-	x	-
getCharacterStream()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	x	x	x	x	x	x	x	x	x	x	x	-	-	-	-
getNCharacterStream()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	x	x	x	-	-	-	-	-	-	-	-
getClob()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	-	-	x	x	x	-	-	-	-	-	-	-	-
getNClob()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	x	-	-	-	-	-	-	-	-
getBlob()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	-	-	x	x	-	-	-	-
getSQLXML()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**X**	-	-	-	-	-	-	-	-
getObject()	x	x	x	x	x	x	x	x	x	x	x	x	x	x	y	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	-	x	x

There are three categories of conversions that are supported by the JDBC driver's getter methods:

- **Non-Lossy (x)**: Conversions for cases where the getter type is the same or smaller than the underlying server type. For example, when calling getBigDecimal on an underlying server decimal column, no conversion is necessary.
- **Converted (y)**: Conversions from numeric server types to Java language types where the conversion is regular and follows Java language conversion rules. For these conversions, precision is always truncated—never rounded—and overflow is handled as modulo of the destination type, which is smaller. For example, calling getInt on an underlying **decimal** column that contains "1.9999" will return "1", or if the underlying

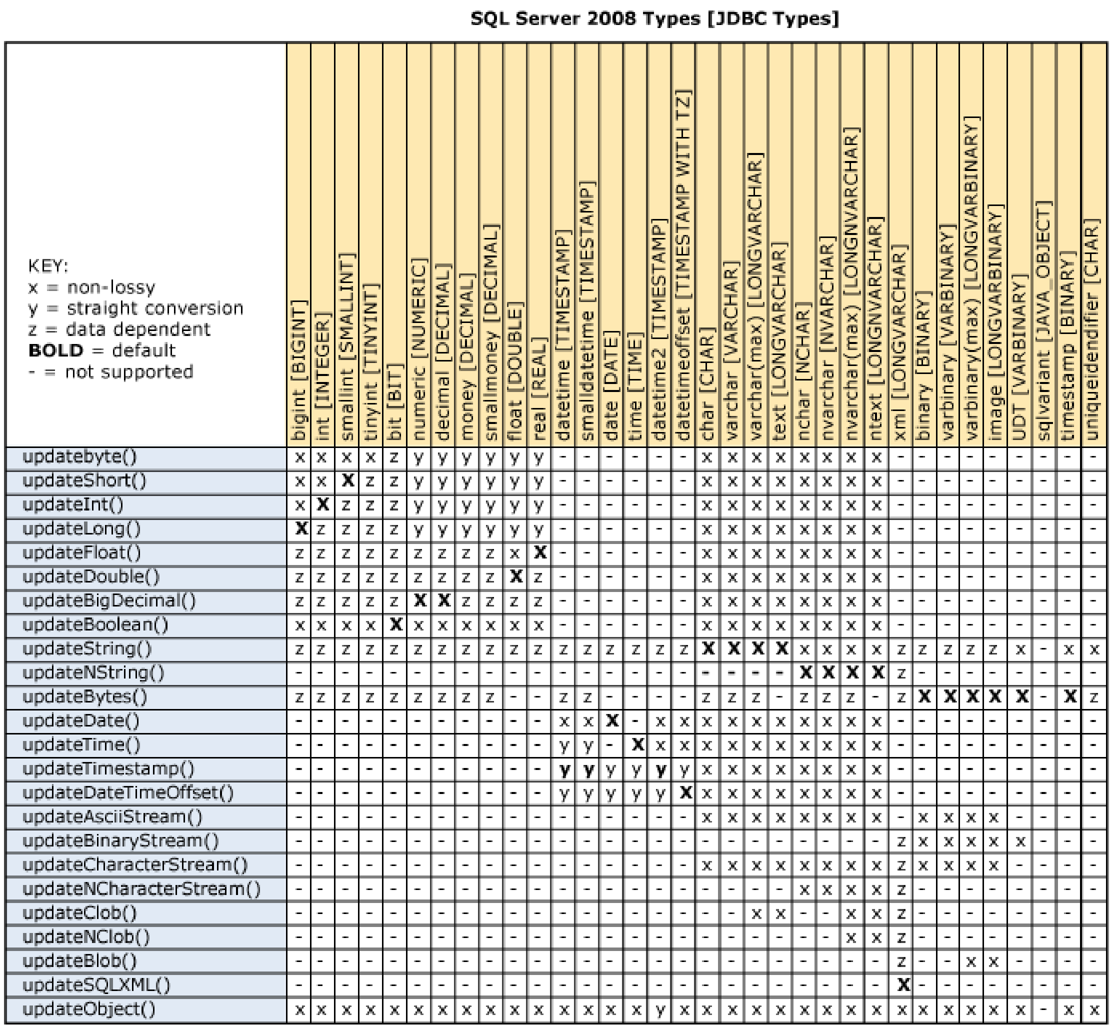
decimal value is "3000000000" then the **int** value overflows to "-1294967296".

- **Data Dependent (z)**: Conversions from underlying character types to numeric types require that the character types contain values that can be converted into that type. No other conversions are performed. If the value is too large for the getter type, the value isn't valid. For example, if getInt is called on a varchar(50) column that contains "53", the value is returned as an **int**; but if the underlying value is "xyz" or "3000000000", an error is thrown.

If getString is called on a **binary**, **varbinary**, **varbinary(max)**, or **image** column data type, the value is returned as a hexadecimal string value.

Updater Method Conversions

For the Java typed data passed to the update<Type>() methods of the SQLServerResultSet class, the following conversions apply.



SQL Server 2008 Types [JDBC Types]

KEY:
x = non-lossy
y = straight conversion
z = data dependent
BOLD = default
- = not supported

	bigint [BIGINT]	int [INTEGER]	smallint [SMALLINT]	tinyint [TINYINT]	bit [BIT]	numeric [NUMERIC]	decimal [DECIMAL]	money [DECIMAL]	smallmoney [DECIMAL]	float [DOUBLE]	real [REAL]	datetime [TIMESTAMP]	smalldatetime [TIMESTAMP]	date [DATE]	time [TIME]	datetime2 [TIMESTAMP]	datetimeoffset [TIMESTAMP WITH TZ]	char [CHAR]	varchar [VARCHAR]	varchar(max) [LONGVARCHAR]	text [LONGVARCHAR]	nchar [NCHAR]	nvarchar [NVARCHAR]	nvarchar(max) [LONGNVARCHAR]	ntext [LONGNVARCHAR]	xml [LONGVARCHAR]	binary [BINARY]	varbinary [VARBINARY]	varbinary(max) [LONGVARBINARY]	image [LONGVARBINARY]	UDT [VARBINARY]	sqlvariant [JAVA_OBJECT]	timestamp [BINARY]	uniqueidentifier [CHAR]
updatebyte()	x	x	x	x	z	y	y	y	y	y	y	-	-	-	-	-	-	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateShort()	x	x	**X**	z	z	y	y	y	y	y	y	-	-	-	-	-	-	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateInt()	x	**X**	z	z	z	y	y	y	y	y	y	-	-	-	-	-	-	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateLong()	**X**	z	z	z	z	y	y	y	y	y	y	-	-	-	-	-	-	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateFloat()	z	z	z	z	z	z	z	z	z	x	**X**	-	-	-	-	-	-	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateDouble()	z	z	z	z	z	z	z	z	z	**X**	z	-	-	-	-	-	-	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateBigDecimal()	z	z	z	z	z	**X**	**X**	z	z	z	z	-	-	-	-	-	-	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateBoolean()	x	x	x	x	**X**	x	x	x	x	x	x	-	-	-	-	-	-	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateString()	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	**X**	**X**	**X**	**X**	x	x	x	x	z	z	z	z	z	x	-	x	x
updateNString()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**X**	**X**	**X**	**X**	z	-	-	-	-	-	-	-	-
updateBytes()	z	z	z	z	z	z	z	z	z	-	-	z	z	-	-	-	-	z	z	z	-	z	z	z	-	z	**X**	**X**	**X**	**X**	**X**	-	**X**	z
updateDate()	-	-	-	-	-	-	-	-	-	-	-	x	x	**X**	-	x	x	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateTime()	-	-	-	-	-	-	-	-	-	-	-	y	y	-	**X**	x	x	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateTimestamp()	-	-	-	-	-	-	-	-	-	-	-	**y**	**y**	y	y	**y**	y	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateDateTimeOffset()	-	-	-	-	-	-	-	-	-	-	-	y	y	y	y	y	**X**	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
updateAsciiStream()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	x	x	x	x	x	x	-	x	x	x	x	-	-	-	-
updateBinaryStream()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	z	x	x	x	x	x	-	-	-
updateCharacterStream()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	x	x	x	x	x	x	z	x	x	x	x	-	-	-	-
updateNCharacterStream()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	x	x	z	-	-	-	-	-	-	-	-
updateClob()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	-	-	x	x	z	-	-	-	-	-	-	-	-
updateNClob()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	z	-	-	-	-	-	-	-	-
updateBlob()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	z	-	-	x	x	-	-	-	-
updateSQLXML()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**X**	-	-	-	-	-	-	-	-
updateObject()	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	y	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	-	x	x

There are three categories of conversions supported by the JDBC driver's updater methods:

- **Non-Lossy (x)**: Conversions for cases where the updater type is the same or smaller than the underlying server type. For example, when calling updateBigDecimal on an underlying server decimal column, no conversion is necessary.

- **Converted (y)**: Conversions from numeric server types to Java language types where the conversion is regular and follows Java language conversion rules. For these conversions, precision is always truncated (never rounded) and overflow is handled as modulo of the destination (the smaller) type. For example, calling updateDecimal on an underlying **int** column that contains "1.9999" will return "1", or if the underlying **decimal** value is "3000000000" then the **int** value overflows to "-1294967296".

- **Data Dependent (z)**: Conversions from underlying source data types to destination data types require that the contained values can be converted into the destination types. No other conversions are performed. If the value is too large for the getter type, the value isn't valid. For example, if updateString is called on an int column that contains "53", the update succeeds; but if the underlying String value is "foo" or "3000000000", an error is thrown.

When updateString is called on a **binary**, **varbinary**, **varbinary(max)**, or **image** column data type, it handles the String value as a hexadecimal string value.

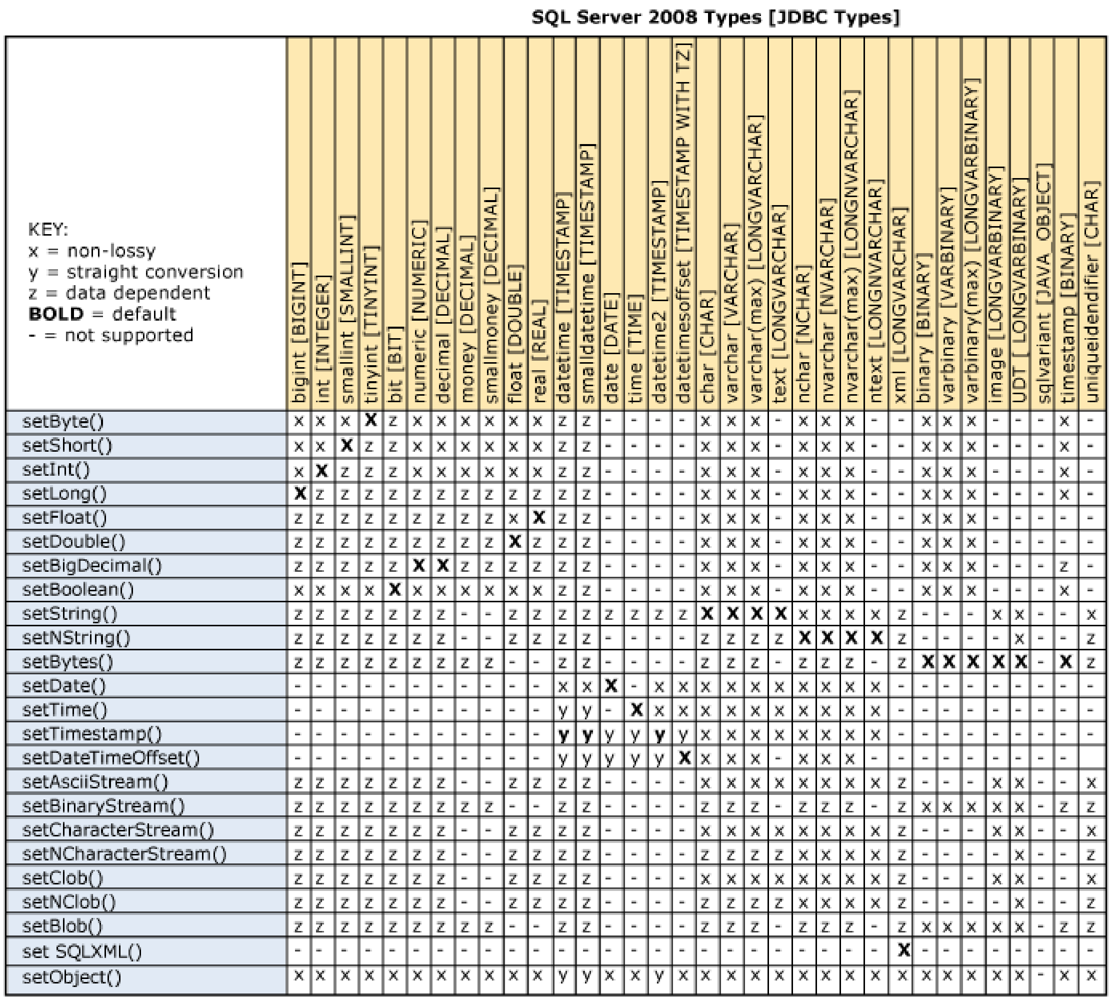
When the SQL Server column data type is **XML**, the data value must be a valid **XML**. When calling updateBytes, updateBinaryStream, or updateBlob methods, the data value should be the hexadecimal string representation of the XML characters. For example:

```
<hello>world</hello> = 0x3C68656C6C6F3E776F726C643C2F68656C6C6F3E
```

Note that a byte-order mark (BOM) is required if the XML characters are in specific character encodings.

Setter Method Conversions

For the Java typed data passed to the set<Type>() methods of the SQLServerPreparedStatement class and the SQLServerCallableStatement class, the following conversions apply.



SQL Server 2008 Types [JDBC Types]

KEY:
x = non-lossy
y = straight conversion
z = data dependent
BOLD = default
- = not supported

	bigint [BIGINT]	int [INTEGER]	smallint [SMALLINT]	tinyint [TINYINT]	bit [BIT]	numeric [NUMERIC]	decimal [DECIMAL]	money [DECIMAL]	smallmoney [DECIMAL]	float [DOUBLE]	real [REAL]	datetime [TIMESTAMP]	smalldatetime [TIMESTAMP]	date [DATE]	time [TIME]	datetime2 [TIMESTAMP]	datetimeoffset [TIMESTAMP WITH TZ]	char [CHAR]	varchar [VARCHAR]	varchar(max) [LONGVARCHAR]	text [LONGVARCHAR]	nchar [NCHAR]	nvarchar [NVARCHAR]	nvarchar(max) [LONGNVARCHAR]	ntext [LONGNVARCHAR]	xml [LONGVARCHAR]	binary [BINARY]	varbinary [VARBINARY]	varbinary(max) [LONGVARBINARY]	image [LONGVARBINARY]	UDT [LONGVARBINARY]	sqlvariant [JAVA_OBJECT]	timestamp [BINARY]	uniqueidendifier [CHAR]
setByte()	x	x	x	**X**	z	x	x	x	x	x	x	z	z	-	-	-	-	x	x	x	-	x	x	x	-	-	x	x	x	-	-	-	x	-
setShort()	x	x	**X**	z	z	x	x	x	x	x	x	z	z	-	-	-	-	x	x	x	-	x	x	x	-	-	x	x	x	-	-	-	x	-
setInt()	x	**X**	z	z	z	x	x	x	x	x	x	z	z	-	-	-	-	x	x	x	-	x	x	x	-	-	x	x	x	-	-	-	x	-
setLong()	**X**	z	z	z	z	z	z	z	z	z	z	z	z	-	-	-	-	x	x	x	-	x	x	x	-	-	x	x	x	-	-	-	x	-
setFloat()	z	z	z	z	z	z	z	z	z	x	**X**	z	z	-	-	-	-	x	x	x	-	x	x	x	-	-	x	x	x	-	-	-	-	-
setDouble()	z	z	z	z	z	z	z	z	z	**X**	z	z	z	-	-	-	-	x	x	x	-	x	x	x	-	-	x	x	x	-	-	-	-	-
setBigDecimal()	z	z	z	z	z	**X**	**X**	z	z	z	z	z	z	-	-	-	-	x	x	x	-	x	x	x	-	-	x	x	x	-	-	-	z	-
setBoolean()	x	x	x	x	**X**	x	x	x	x	x	x	z	z	-	-	-	-	x	x	x	-	x	x	x	-	-	x	x	x	-	-	-	x	-
setString()	z	z	z	z	z	z	z	-	-	z	z	z	z	z	z	z	z	**X**	**X**	**X**	**X**	x	x	x	x	z	-	-	-	x	x	-	-	x
setNString()	z	z	z	z	z	z	z	-	-	z	z	z	z	-	-	-	-	z	z	z	z	**X**	**X**	**X**	**X**	z	-	-	-	-	x	-	-	z
setBytes()	z	z	z	z	z	z	z	z	z	-	-	z	z	-	-	-	-	z	z	z	-	z	z	z	-	z	**X**	**X**	**X**	**X**	**X**	-	**X**	z
setDate()	-	-	-	-	-	-	-	-	-	-	-	x	x	**X**	-	x	x	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
setTime()	-	-	-	-	-	-	-	-	-	-	-	y	y	-	**X**	x	x	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
setTimestamp()	-	-	-	-	-	-	-	-	-	-	-	**y**	**y**	y	y	**y**	y	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
setDateTimeOffset()	-	-	-	-	-	-	-	-	-	-	-	y	y	y	y	y	**X**	x	x	x	-	x	x	x	-	-	-	-	-	-	-	-	-	
setAsciiStream()	z	z	z	z	z	z	z	-	-	z	z	z	z	-	-	-	-	x	x	x	x	x	x	x	x	z	-	-	-	x	x	-	-	x
setBinaryStream()	z	z	z	z	z	z	z	z	z	-	-	z	z	-	-	-	-	z	z	z	-	z	z	z	-	z	x	x	x	x	x	-	z	z
setCharacterStream()	z	z	z	z	z	z	z	-	-	z	z	z	z	-	-	-	-	x	x	x	x	x	x	x	x	z	-	-	-	x	x	-	-	x
setNCharacterStream()	z	z	z	z	z	z	z	-	-	z	z	z	z	-	-	-	-	z	z	z	z	x	x	x	x	z	-	-	-	-	x	-	-	z
setClob()	z	z	z	z	z	z	z	-	-	z	z	z	z	-	-	-	-	x	x	x	x	x	x	x	x	z	-	-	-	x	x	-	-	x
setNClob()	z	z	z	z	z	z	z	-	-	z	z	z	z	-	-	-	-	z	z	z	z	x	x	x	x	z	-	-	-	-	x	-	-	z
setBlob()	z	z	z	z	z	z	z	z	z	-	-	z	z	-	-	-	-	z	z	z	-	z	z	z	-	z	x	x	x	x	x	-	z	z
set SQLXML()	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**X**	-	-	-	-	-	-	-	-
setObject()	x	x	x	x	x	x	x	x	x	x	x	y	y	x	x	y	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	-	x	x

The server tries any conversions and returns errors on failure.

In the case of the **String** data type, if the value exceeds the length of **VARCHAR**, it maps to **LONGVARCHAR**. Similarly, **NVARCHAR** maps to **LONGNVARCHAR** if the value exceeds the supported length of **NVARCHAR**. The same is true for **byte[]**. Values longer than **VARBINARY** become **LONGVARBINARY**.

There are two categories of conversions that are supported by the JDBC driver's setter methods:

- **Non-Lossy (x)**: Conversions for numeric cases where the setter type is the same or smaller than the underlying server type. For example, when calling setBigDecimal on an underlying server **decimal** column, no conversion is necessary. For numeric to character cases, the Java **numeric** data type is converted to a **String**. For example, calling setDouble with a value of "53" on a varchar(50) column produces a character value "53" in that destination column.

- **Converted (y)**: Conversions from a Java **numeric** type to an underlying server **numeric** type that is smaller. This conversion is regular and follows SQL Server conversion conventions. Precision is always truncated (never rounded) and overflow throws an unsupported conversion error. For example, using updateDecimal with a value of "1.9999" on an underlying integer column results in a "1" in the destination column; but if "3000000000" is passed, the driver throws an error.

- **Data Dependent (z)**: Conversions from a Java **String** type to the underlying SQL Server data type depends on the following conditions: The driver sends the **String** value to SQL Server and SQL Server performs conversions, if necessary. If the sendStringParametersAsUnicode is set to true and the underlying SQL Server data type is **image**, SQL Server doesn't allow converting **nvarchar** to **image** and throws an SQLServerException. If the sendStringParametersAsUnicode is set to false and the underlying SQL Server data type is **image**, SQL Server allows converting **varchar** to **image** and doesn't throw an exception.

SQL Server performs the conversions and passes errors back to the JDBC driver when there are problems.

When the SQL Server column data type is **XML**, the data value must be a valid **XML**. When calling updateBytes, updateBinaryStream, or updateBlob methods, the data value should be the hexadecimal string representation of the XML characters. For example:

```
<hello>world</hello> = 0x3C68656C6C6F3E776F726C643C2F68656C6C6F3E
```

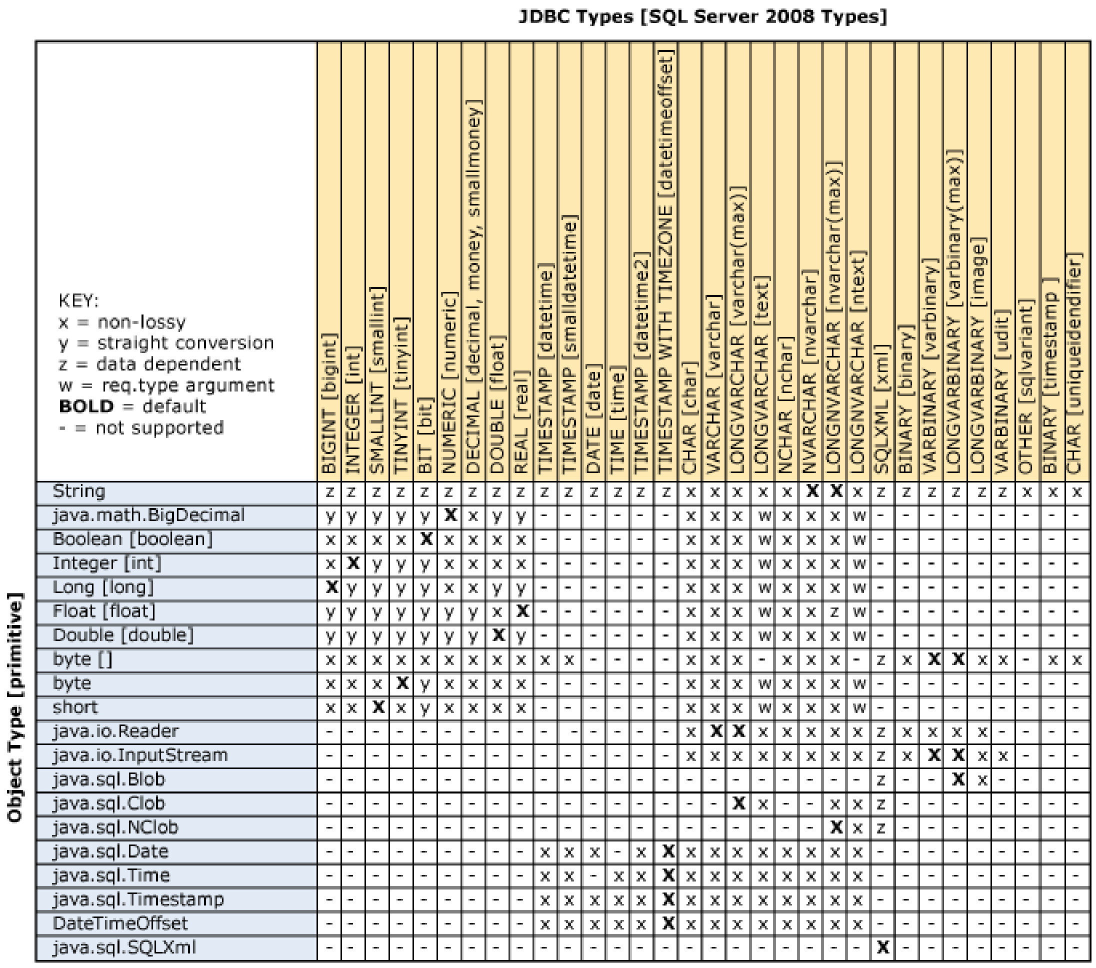
Note that a byte-order mark (BOM) is required if the XML characters are in specific character encodings.

Conversions on setObject

NOTE

Microsoft JDBC Drivers 4.2 (and higher) for SQL Server supports JDBC 4.1 and 4.2. For more detail on 4.1 and 4.2 datatype mappings and conversions see JDBC 4.1 Compliance for the JDBC Driver and JDBC 4.2 Compliance for the JDBC Driver, in addition to the information below.

For the Java typed data passed to the setObject(<Type>) methods of the SQLServerPreparedStatement class, the following conversions apply.



JDBC Types [SQL Server 2008 Types]

KEY:
x = non-lossy
y = straight conversion
z = data dependent
w = req.type argument
BOLD = default
- = not supported

Object Type [primitive]	BIGINT [bigint]	INTEGER [int]	SMALLINT [smallint]	TINYINT [tinyint]	BIT [bit]	NUMERIC [numeric]	DECIMAL [decimal, money, smallmoney]	DOUBLE [float]	REAL [real]	TIMESTAMP [datetime]	TIMESTAMP [smalldatetime]	DATE [date]	TIME [time]	TIMESTAMP [datetime2]	TIMESTAMP WITH TIMEZONE [datetimeoffset]	CHAR [char]	VARCHAR [varchar]	LONGVARCHAR [varchar(max)]	LONGVARCHAR [text]	NCHAR [nchar]	NVARCHAR [nvarchar]	LONGNVARCHAR [nvarchar(max)]	LONGNVARCHAR [ntext]	SQLXML [xml]	BINARY [binary]	VARBINARY [varbinary]	LONGVARBINARY [varbinary(max)]	LONGVARBINARY [image]	VARBINARY [udt]	OTHER [sqlvariant]	BINARY [timestamp]	CHAR [uniqueidendifier]
String	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	x	x	x	x	x	**X**	**X**	x	z	z	z	z	z	z	x	x	x
java.math.BigDecimal	y	y	y	y	y	**X**	x	y	y	-	-	-	-	-	-	x	x	x	w	x	x	x	w	-	-	-	-	-	-	-	-	-
Boolean [boolean]	x	x	x	x	**X**	x	x	x	x	-	-	-	-	-	-	x	x	x	w	x	x	x	w	-	-	-	-	-	-	-	-	-
Integer [int]	x	**X**	y	y	y	x	x	x	x	-	-	-	-	-	-	x	x	x	w	x	x	x	w	-	-	-	-	-	-	-	-	-
Long [long]	**X**	y	y	y	y	x	x	y	y	-	-	-	-	-	-	x	x	x	w	x	x	x	w	-	-	-	-	-	-	-	-	-
Float [float]	y	y	y	y	y	y	y	x	**X**	-	-	-	-	-	-	x	x	x	w	x	x	z	w	-	-	-	-	-	-	-	-	-
Double [double]	y	y	y	y	y	y	y	**X**	y	-	-	-	-	-	-	x	x	x	w	x	x	x	w	-	-	-	-	-	-	-	-	-
byte []	x	x	x	x	x	x	x	x	x	x	x	-	-	-	-	x	x	x	-	x	x	x	-	z	x	**X**	**X**	x	x	-	x	x
byte	x	x	x	**X**	y	x	x	x	x	-	-	-	-	-	-	x	x	x	w	x	x	x	w	-	-	-	-	-	-	-	-	-
short	x	x	**X**	x	y	x	x	x	x	-	-	-	-	-	-	x	x	x	w	x	x	x	w	-	-	-	-	-	-	-	-	-
java.io.Reader	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	**X**	**X**	x	x	x	x	x	z	x	x	x	x	-	-	-	-
java.io.InputStream	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	x	x	x	x	x	x	x	x	z	x	**X**	**X**	x	x	-	-	-
java.sql.Blob	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	z	-	-	**X**	x	-	-	-	-
java.sql.Clob	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**X**	x	-	-	x	x	z	-	-	-	-	-	-	-	-
java.sql.NClob	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**X**	x	z	-	-	-	-	-	-	-	-
java.sql.Date	-	-	-	-	-	-	-	-	-	x	x	x	-	x	**X**	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
java.sql.Time	-	-	-	-	-	-	-	-	-	x	x	-	x	x	**X**	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
java.sql.Timestamp	-	-	-	-	-	-	-	-	-	x	x	x	x	x	**X**	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
DateTimeOffset	-	-	-	-	-	-	-	-	-	x	x	x	x	x	**X**	x	x	x	x	x	x	x	x	-	-	-	-	-	-	-	-	-
java.sql.SQLXml	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**X**	-	-	-	-	-	-	-	-

The setObject method with no specified target type uses the default mapping. In the case of the **String** data type, if the value exceeds the length of **VARCHAR**, it maps to **LONGVARCHAR**. Similarly, **NVARCHAR** maps to **LONGNVARCHAR** if the value exceeds the supported length of **NVARCHAR**. The same is true for **byte[]**. Values longer than **VARBINARY** become **LONGVARBINARY**.

There are three categories of conversions that are supported by the JDBC driver's setObject methods:

- **Non-Lossy (x)**: Conversions for numeric cases where the setter type is the same or smaller than the underlying server type. For example, when calling setBigDecimal on an underlying server **decimal** column, no conversion is necessary. For numeric to character cases, the Java **numeric** data type is converted to a **String**. For example, calling setDouble with a value of "53" on a varchar(50) column will produce a character value "53" in that destination column.

- **Converted (y)**: Conversions from a Java **numeric** type to an underlying server **numeric** type that is smaller. This conversion is regular and follows SQL Server conversion conventions. Precision is always truncated—never rounded—and overflow throws an unsupported conversion error. For example, using updateDecimal with a value of "1.9999" on an underlying integer column results in a "1" in the destination column; but if "3000000000" is passed, the driver throws an error.

- **Data Dependent (z)**: Conversions from a Java **String** type to the underlying SQL Server data type depends on the following conditions: The driver sends the **String** value to SQL Server and SQL Server performs conversions, if necessary. If the sendStringParametersAsUnicode connection property is set to true and the underlying SQL Server data type is **image**, SQL Server doesn't allow converting **nvarchar** to **image** and throws an SQLServerException. If the sendStringParametersAsUnicode is set to false and the underlying SQL Server data type is **image**, SQL Server allows converting **varchar** to **image** and doesn't throw an exception.

SQL Server performs the bulk of the set conversions and passes errors back to the JDBC driver when there are problems. Client-side conversions are the exception and are performed only in the case of **date**, **time**, **timestamp**,

Boolean, and **String** values.

When the SQL Server column data type is **XML**, the data value must be a valid **XML**. When calling setObject(byte[], SQLXML), setObject(inputStream, SQLXML), or setObject(Blob, SQLXML) methods, the data value should be the hexadecimal string representation of the XML characters. For example:

```
<hello>world</hello> = 0x3C68656C6C6F3E776F726C643C2F68656C6C6F3E
```

Note that a byte-order mark (BOM) is required if the XML characters are in specific character encodings.

See Also

Understanding the JDBC Driver Data Types

Understanding Concurrency Control

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Concurrency control refers to the various techniques that are used to preserve the integrity of the database when multiple users are updating rows at the same time. Incorrect concurrency can lead to problems such as dirty reads, phantom reads, and non-repeatable reads. The Microsoft JDBC Driver for SQL Server provides interfaces to all the concurrency techniques used by SQL Server to resolve these issues.

NOTE

For more information about SQL Server concurrency, see "Managing Concurrent Data Access" in SQL Server Books Online.

Remarks

The JDBC driver supports the following concurrency types:

CONCURRENCY TYPE	CHARACTERISTICS	ROW LOCKS	DESCRIPTION
CONCUR_READ_ONLY	Read Only	No	Updates through the cursor are not allowed, and no locks are held on the rows that make up the result set.
CONCUR_UPDATABLE	Optimistic Read Write	No	Database assumes row contention is unlikely, but possible. Row integrity is checked with a timestamp comparison.
CONCUR_SS_SCROLL_LOCKS	Pessimistic Read Write	Yes	Database assumes row contention is likely. Row integrity is ensured with row locking.

CONCURRENCY TYPE	CHARACTERISTICS	ROW LOCKS	DESCRIPTION
CONCUR_SS_OPTIMISTIC_CC	Optimistic Read Write	No	Database assumes row contention is unlikely, but possible. Row integrity is verified with a timestamp comparison. For SQL Server 2005 (9.x) and later, the server will change this to CONCUR_SS_OPTIMISTIC_CCVAL if the table does not contain a timestamp column. For SQL Server 2000 (8.x), if the underlying table has a timestamp column, OPTIMISTIC WITH ROW VERSIONING is used even if OPTIMISTIC WITH VALUES is specified. If OPTIMISTIC WITH ROW VERSIONING is specified and the table does not have timestamps, OPTIMISTIC WITH VALUES is used.
CONCUR_SS_OPTIMISTIC_CCVAL	Optimistic Read Write	No	Database assumes row contention is unlikely, but possible. Row integrity is checked with a row data comparison.

Result Sets That Are Not Updateable

An updatable result set is a result set in which rows can be inserted, updated, and deleted. In the following cases, SQL Server cannot create an updatable cursor. The exception generated is, "Result set is not updatable."

CAUSE	DESCRIPTION	REMEDY
Statement is not created by using JDBC 2.0 (or later) syntax	JDBC 2.0 introduced new methods to create statements. If JDBC 1.0 syntax is used, the result set defaults to read-only.	Specify result set type and concurrency when creating the statement.
Statement is created by using TYPE_SCROLL_INSENSITIVE	SQL Server creates a static snapshot cursor. This is disconnected from the underlying table rows to help protect the cursor from row updates by other users.	Use TYPE_SCROLL_SENSITIVE, TYPE_SS_SCROLL_KEYSET, TYPE_SS_SCROLL_DYNAMIC, or TYPE_FORWARD_ONLY with CONCUR_UPDATABLE to avoid creating a static cursor.
Table design precludes a KEYSET or DYNAMIC cursor	The underlying table does not have unique keys to enable SQL Server to uniquely identify a row.	Add unique keys to the table to provide unique identification of each row.

See Also

Using a Stored Procedure with Input Parameters

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

A SQL Server stored procedure that you can call is one that contains one or more IN parameters, which are parameters that can be used to pass data into the stored procedure. The Microsoft JDBC Driver for SQL Server provides the SQLServerPreparedStatement class, which you can use to call this kind of stored procedure and to process the data that it returns.

When you use the JDBC driver to call a stored procedure with IN parameters, you must use the `call` SQL escape sequence together with the prepareCall method of the SQLServerConnection class. The syntax for the `call` escape sequence with IN parameters is as follows:

```
{call procedure-name[([parameter][,[parameter]]...)]}
```

NOTE

For more information about the SQL escape sequences, see Using SQL Escape Sequences.

When you construct the `call` escape sequence, specify the IN parameters by using the ? (question mark) character. This character acts as a placeholder for the parameter values that will be passed into the stored procedure. To specify a value for a parameter, you can use one of the setter methods of the SQLServerPreparedStatement class. The setter method that you can use is determined by the data type of the IN parameter.

When you pass a value to the setter method, you must specify not only the actual value that will be used in the parameter, but also the ordinal placement of the parameter in the stored procedure. For example, if your stored procedure contains a single IN parameter, its ordinal value will be 1. If the stored procedure contains two parameters, the first ordinal value will be 1, and the second ordinal value will be 2.

As an example of how to call a stored procedure that contains an IN parameter, use the uspGetEmployeeManagers stored procedure in the AdventureWorks sample database. This stored procedure accepts a single input parameter named EmployeeID, which is an integer value, and it returns a recursive list of employees and their managers based on the specified EmployeeID. The Java code for calling this stored procedure is as follows:

```
public static void executeSprocInParams(Connection con) throws SQLException {
    try(PreparedStatement pstmt = con.prepareStatement("{call dbo.uspGetEmployeeManagers(?)}"); ) {

        pstmt.setInt(1, 50);
        ResultSet rs = pstmt.executeQuery();

        while (rs.next()) {
            System.out.println("EMPLOYEE:");
            System.out.println(rs.getString("LastName") + ", " + rs.getString("FirstName"));
            System.out.println("MANAGER:");
            System.out.println(rs.getString("ManagerLastName") + ", " + rs.getString("ManagerFirstName"));
            System.out.println();
        }
    }
}
```

See Also

Using Statements with Stored Procedures

Using Database Metadata

8/8/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

To query a database for information about what it supports, the Microsoft JDBC Driver for SQL Server implements the SQLServerDatabaseMetaData class. This class contains numerous methods that return information in the form of a single value, or as a result set.

To create a SQLServerDatabaseMetaData object, you can use the getMetaData method of the SQLServerConnection class to get information about the database that it is connected to.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, the getMetaData method of the SQLServerConnection class is used to return a SQLServerDatabaseMetadata object, and then various methods of the SQLServerDatabaseMetaData object are used to display information about the driver, driver version, database name, and database version.

```
public static void getDatabaseMetaData(Connection con) {
    try {
        DatabaseMetaData dbmd = con.getMetaData();
        System.out.println("dbmd:driver version = " + dbmd.getDriverVersion());
        System.out.println("dbmd:driver name = " + dbmd.getDriverName());
        System.out.println("db name = " + dbmd.getDatabaseProductName());
        System.out.println("db ver = " + dbmd.getDatabaseProductVersion());
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

See Also

Handling Metadata with the JDBC Driver

SQLXML Interface

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The JDBC driver provides support for the JDBC 4.0 API, which introduces the java.sql.SQLXML interface. The SQLXML interface defines methods to interact and manipulate XML data. The **SQLXML** data type maps to the SQL Server**xml** data type.

The SQLXML interface provides methods for accessing the XML value as a **String**, a **Reader** or **Writer**, or as a **Stream**. The XML value may also be accessed through a **Source** or set as a **Result**, which are used with XML Parser APIs such as Document Object Model (DOM), Simple API for XML (SAX), and Streaming API for XML (StAX), as well as with XSLT transforms and XPath.

Remarks

The following table describes the methods defined in the SQLXML interface:

METHOD SYNTAX	METHOD DESCRIPTION
void free()	This method frees the SQLXML object and releases the resources that it holds.
InputStream getBinaryStream()	Returns an input stream to read data from the SQLXML.
Reader getCharacterStream()	Returns the **XML** data as a java.io.Reader object or as a stream of characters.
T extends Source T getSource(Class<T> sourceClass)	Returns a **Source** for reading the **XML** value specified by this **SQLXML** object. **Note:** The getSource method supports the following sources: javax.xml.transform.dom.DOMSource, javax.xml.transform.sax.SAXSource, javax.xml.transform.stax.StAXSource, and java.io.InputStream.
String getString()	Returns a string representation of the **XML** value designated by this SQLXML object.
OutputStream setBinaryStream()	Retrieves a stream that can be used to write the **XML** value that this SQLXML object represents.
Writer setCharacterStream()	Returns a stream to be used to write the **XML** value that this SQLXML object represents.
T extends Result T setResult(Class<T> resultClass)	Returns a **Result** for setting the **XML** value specified by this **SQLXML** object. **Note:** The setResult method supports the following sources: javax.xml.transform.dom.DOMResult, javax.xml.transform.sax.SAXResult, javax.xml.transform.stax.StaxResult, and java.io.OutputStream.

METHOD SYNTAX	METHOD DESCRIPTION
void setString(String value)	Sets the XML value designated by this SQLXML object to the specified **String** representation.

The applications can read and write XML values to or from an SQLXML object only once.

When the free() method is called, a SQLXML object becomes invalid and is neither readable nor writeable. If the application tries to invoke a method on that SQLXML object other than the free() method, an exception is thrown.

The SQLXML object becomes neither readable nor writable when the application calls any of the following getter methods: getSource, getCharacterStream, getBinaryStream, and getString.

The SQLXML object becomes neither writeable nor readable when the application calls any of the following setter methods: setResult, setCharacterStream, setBinaryStream, and setString.

See Also

Supporting XML Data

Handling Complex Statements

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

When you use the Microsoft JDBC Driver for SQL Server, you might have to handle complex statements, including statements that are dynamically generated at runtime. Complex statements often perform a variety of tasks, including updates, inserts, and deletes. These types of statements might also return multiple result sets and output parameters. In these situations, the Java code that runs the statements might not know in advance the types and number of objects and data returned.

To effectively process complex statements, the JDBC driver provides a number of methods to query the objects and data that is returned so your application can correctly process them. The key to processing complex statements is the execute method of the SQLServerStatement class. This method returns a **boolean** value. When the value is true, the first result returned from the statements is a result set. When the value is false, the first result returned was an update count.

When you know the type of object or data that was returned, you can use either the getResultSet or the getUpdateCount method to process that data. To proceed to the next object or data that is returned from the complex statement, you can call the getMoreResults method.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, a complex statement is constructed that combines a stored procedure call with a SQL statement, the statements are run, and then a `do` loop is used to process all the result sets and updated counts that are returned.

```
public static void executeComplexStatement(Connection con) {
    try (Statement stmt = con.createStatement();) {
        String sqlStringWithUnknownResults = "{call dbo.uspGetEmployeeManagers(50)}; SELECT TOP 10 * FROM
Person.Contact";
        boolean results = stmt.execute(sqlStringWithUnknownResults);
        int count = 0;
        do {
            if (results) {
                ResultSet rs = stmt.getResultSet();
                System.out.println("Result set data displayed here.");
            }
            else {
                count = stmt.getUpdateCount();
                if (count >= 0) {
                    System.out.println("DDL or update data displayed here.");
                }
                else {
                    System.out.println("No more results to process.");
                }
            }
            results = stmt.getMoreResults();
        }
        while (results || count != -1);
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

See Also

Using Statements with the JDBC Driver

Using Connection Pooling

8/8/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server provides support for Java Platform, Enterprise Edition (Java EE) connection pooling. The JDBC driver implements the JDBC 3.0 required interfaces to enable the driver to participate in any connection-pooling implementation that is provided by middleware vendors and is JDBC 3.0-compliant. Middleware such as Java EE application servers frequently provides compliant connection-pooling facilities. The JDBC driver will participate in pooled connections in these environments.

NOTE

Although the JDBC driver supports Java EE connection pooling, it does not provide its own pooling implementation. The driver relies on third-party Java Application Servers to manage the connections.

Remarks

The classes for the connection pooling implementation are as follows.

CLASS	IMPLEMENTS	DESCRIPTION
com.microsoft.sqlserver.jdbc. SQLServerXADataSource	javax.sql.ConnectionPoolDataSource and javax.sql.XADataSource	We recommend that you use the SQLServerXADataSource class for all your Java EE server needs, because it implements all the JDBC 3.0 pooling and XA interfaces.
com.microsoft.sqlserver.jdbc. SQLServerConnectionPoolDataSource	javax.sql.ConnectionPoolDataSource	This class is a connection factory that enables the Java EE application server to populate its connection pool with physical connections. If the configuration of your Java EE vendor requires a class that implements javax.sql.ConnectionPoolDataSource, specify the class name as SQLServerConnectionPoolDataSource. We generally recommend that you use the SQLServerXADataSource class instead, because it implements both pooling and XA interfaces, and has been verified in more Java EE server configurations.

JDBC application code should always close connections explicitly to derive the most benefit from pooling. When the application explicitly closes a connection, the pooling implementation can reuse the connection immediately. If the connection is not closed, other applications cannot reuse it. Applications can use the `finally` construct to make sure that pooled connections are closed even if an exception occurs.

NOTE

The JDBC driver does not currently call the sp_reset_connection stored procedure when it returns the connection to the pool. Instead, the driver relies on third-party Java Application Servers to return the connections back to their original states.

See Also

Connecting to SQL Server with the JDBC Driver

Always Encrypted API Reference for the JDBC Driver

8/15/2018 • 15 minutes to read • Edit Online

Download JDBC Driver

Always Encrypted allows clients to encrypt sensitive data inside client applications and never reveal the encryption keys to SQL Server. An Always Encrypted enabled driver installed on the client computer achieves this functionality by automatically encrypting and decrypting sensitive data in the SQL Server client application. The driver encrypts the data in sensitive columns before passing the data to SQL Server, and automatically rewrites queries so that the semantics to the application are preserved. Similarly, the driver transparently decrypts data stored in encrypted database columns that are in query results. For more information, see Always Encrypted (Database Engine) and Using Always Encrypted with the JDBC Driver.

NOTE

Always Encrypted is supported only by Microsoft JDBC Driver 6.0 or higher for SQL Server with SQL Server 2016.

Always Encrypted API References

There are several new additions and modifications to the JDBC driver API for use in client applications that use Always Encrypted.

SQLServerConnection Class

NAME	DESCRIPTION
New connection string keyword: columnEncryptionSetting	columnEncryptionSetting=Enabled enables Always Encrypted functionality for the connection and columnEncryptionSetting=Disabled disables it. Accepted values are Enabled/Disabled. The default is Disabled.

New methods:
 	
`public static void setColumnEncryptionTrustedMasterKeyPaths(Map<String, List\<String>> trustedKeyPaths)`
 	
`public static void updateColumnEncryptionTrustedMasterKeyPaths(String server, List\<String> trustedKeyPaths)`
 	
`public static void removeColumnEncryptionTrustedMasterKeyPaths(String server)`
	Allows you to set/update/remove a list of trusted key paths for a database server. If while processing an application query the driver receives a key path that's not on the list, the query will fail. This property provides additional protection against security attacks that involve a compromised SQL Server sending fake key paths, which may lead to leaking key store credentials.

New method: `public static Map<String, List\<String>> getColumnEncryptionTrustedMasterKeyPaths()`	Returns a list of trusted key paths for a database server.

NAME	DESCRIPTION
New method: `public static void registerColumnEncryptionKeyStoreProviders (Map\ <String, SQLServerColumnEncryptionKeyStoreProvider> clientKeyStoreProviders)`	Allows you to register custom key store providers. It's a dictionary that maps key store provider names to key store provider implementations. To use the JVM key store, you need to instantiate a SQLServerColumnEncryptionJVMKeyStoreProvider object with JVM keystore credentials and register it with the driver. The name for this provider must be 'MSSQL_JVM_KEYSTORE'. To use the Azure Key Vault store, you need to instantiate a SQLServerColumnEncryptionAzureKeyStoreProvider object and register it with the driver. The name for this provider must be 'AZURE_KEY_VAULT'.
`public final boolean getSendTimeAsDatetime()`	Returns the setting of the sendTimeAsDatetime connection property.
`public void setSendTimeAsDatetime(boolean sendTimeAsDateTimeValue)`	Modifies the setting of the sendTimeAsDatetime connection property.

SQLServerConnectionPoolProxy Class

NAME	DESCRIPTION
`public final boolean getSendTimeAsDatetime()`	Returns the setting of the sendTimeAsDatetime connection property.
`public void setSendTimeAsDatetime(boolean sendTimeAsDateTimeValue)`	Modifies the setting of the sendTimeAsDatetime connection property.

SQLServerDataSource Class

NAME	DESCRIPTION
`public void setColumnEncryptionSetting(String columnEncryptionSetting)`	Enables/disables Always Encrypted functionality for the data source object. The default is Disabled.
`public String getColumnEncryptionSetting()`	Retrieves the Always Encrypted functionality setting for the data source object.
`public void setKeyStoreAuthentication(String keyStoreAuthentication)`	Sets the name that identifies a key store. Only value supported is the **JavaKeyStorePassword** for identifying the Java Key Store. The default is null.
`public String getKeyStoreAuthentication()`	Gets the value of the keyStoreAuthentication setting for the data source object.
`public void setKeyStoreSecret(String keyStoreSecret)`	Sets the password for the Java keystore. The password for the keystore and the key must be the same. Note that keyStoreAuthentication must be set with **JavaKeyStorePassword**.

NAME	DESCRIPTION
`public void setKeyStoreLocation(String keyStoreLocation)`	Sets the location including the file name for the Java keystore. Note that keyStoreAuthentication must be set with **JavaKeyStorePassword**.
`public String getKeyStoreLocation()`	Retrieves the keyStoreLocation for the Java Key Store.

SQLServerColumnEncryptionJavaKeyStoreProvider Class

The implementation of the key store provider for Java Key Store. This class enables using certificates stored in the Java keystore as column master keys.

Constructors

NAME	DESCRIPTION
`public SQLServerColumnEncryptionJavaKeyStoreProvider (String keyStoreLocation, char[] keyStoreSecret)`	Key store provider for the Java Key Store.

Methods

NAME	DESCRIPTION
`public byte[] decryptColumnEncryptionKey (String masterKeyPath, String encryptionAlgorithm, byte[] encryptedColumnEncryptionKey)`	Decrypts the specified encrypted value of a column encryption key. The encrypted value is expected to be encrypted using the certificate with the specified key path and using the specified algorithm. **The format of the key path should be one of the following:** Thumbprint:<certificate_thumbprint> Alias:<certificate_alias> (Overrides SQLServerColumnEncryptionKeyStoreProvider. decryptColumnEncryptionKey(String, String, Byte[]).)
`public byte[] encryptColumnEncryptionKey (String masterKeyPath, String encryptionAlgorithm, byte[] plainTextColumnEncryptionKey)`	Encrypts a column encryption key using the certificate with the specified key path and using the specified algorithm. **The format of the key path should be one of the following:** Thumbprint:<certificate_thumbprint> Alias:<certificate_alias> (Overrides SQLServerColumnEncryptionKeyStoreProvider. encryptColumnEncryptionKey(String, String, Byte[]).)
`public void setName (String name)`	Sets the name of this key store provider.
`public String getName ()`	Gets the name of this key store provider.

SQLServerColumnEncryptionAzureKeyVaultProvider Class

The implementation of the key store provider for Azure Key Vault. This class enables using keys stored in the Azure Key Vault as column master keys.

Constructors

NAME	DESCRIPTION
`public SQLServerColumnEncryptionAzureKeyVaultProvider (String clientId, String clientKey)`	Key store provider for Azure Key Vault. You need to provide the identifier and the key of the client requesting the token to authenticate to Azure Key Vault.

Methods

NAME	DESCRIPTION
`public byte[] decryptColumnEncryptionKey (String masterKeyPath, String encryptionAlgorithm, byte[] encryptedColumnEncryptionKey)`	Decryptes an encrypted column encryption key (CEK). This decryption is accomplished with an RSA encryption algorithm that uses the asymmetric key specified by the master key path. (Overrides SQLServerColumnEncryptionKeyStoreProvider. decryptColumnEncryptionKey(String, String, Byte[]).)
`public byte[] encryptColumnEncryptionKey (String masterKeyPath, String encryptionAlgorithm, byte[] columnEncryptionKey)`	Encrypts a column encryption key, by giving the specified column master key to the specified algorithm. (Overrides SQLServerColumnEncryptionKeyStoreProvider. encryptColumnEncryptionKey(String, String, Byte[]).)
`public void setName (String name)`	Sets the name of this key store provider.
`public String getName ()`	Gets the name of this key store provider.

SQLServerKeyVaultAuthenticationCallback Interface

This interface contains one method for Azure Key Vault authentication, which is to be implemented by user.

Methods

NAME	DESCRIPTION
`public String getAccessToken(String authority, String resource, String scope);`	The method needs to be overridden. The method is used to get access token to Azure Key Vault.

SQLServerColumnEncryptionKeyStoreProvider Class

Extend this class to implement a custom key store provider.

NAME	DESCRIPTION
SQLServerColumnEncryptionKeyStoreProvider	Base class for all key store providers. A custom provider must derive from this class and override its member functions and then register it using SQLServerConnection. registerColumnEncryptionKeyStoreProviders().

Methods

NAME	DESCRIPTION
`public abstract byte[] decryptColumnEncryptionKey (String masterKeyPath, String encryptionAlgorithm, byte [] encryptedColumnEncryptionKey)`	Base class method for decrypting the specified encrypted value of a column encryption key. The encrypted value is expected to be encrypted using the column master key with the specified key path and the specified algorithm.
`public abstract byte[] encryptColumnEncryptionKey (String masterKeyPath, String encryptionAlgorithm, byte[] columnEncryptionKey)`	Base class method for encrypting a column encryption key using the column master key with the specified key path and using the specified algorithm.
`public abstract void setName(String name)`	Sets the name of this key store provider.
`public abstract String getName()`	Gets the name of this key store provider.

New or overloaded methods in **SQLServerPreparedStatement** Class

NAME	DESCRIPTION

`public void setBigDecimal(int parameterIndex, BigDecimal x, int precision, int scale)`
 	
`public void setObject(int parameterIndex, Object x, int targetSqlType, Integer precision, int scale)`
 	
`public void setObject(int parameterIndex, Object x, SQLType targetSqlType, Integer precision, Integer scale)`
 	
`public void setTime(int parameterIndex, java.sql.Time x, int scale)`
 	
`public void setTimestamp(int parameterIndex, java.sql.Timestamp x, int scale)`
`public void setDateTimeOffset(int parameterIndex, microsoft.sql.DateTimeOffset x, int scale)`
	These methods are overloaded with a precision or a scale argument or both to support Always Encrypted for specific data types that require precision and scale information.

`public void setMoney(int parameterIndex, BigDecimal x)`
 	
`public void setSmallMoney(int parameterIndex, BigDecimal x)`
 	
`public void setUniqueIdentifier(int parameterIndex, String guid)`
 	
`public void setDateTime(int parameterIndex, java.sql.Timestamp x)`
 	
`public void setSmallDateTime(int parameterIndex, java.sql.Timestamp x)`
	These methods are added to support Always Encrypted for the data types money, smallmoney, uniqueidentifier, datetime and smalldatetime.
	
	Note that the existing `setTimestamp()` method is used for sending parameter values to the encrypted datetime2 column. For encrypted datetime and smalldatetime columns use the new methods `setDateTime()` and `setSmallDateTime()` respectively.

`public final void setBigDecimal(int parameterIndex, BigDecimal x, int precision, int scale, boolean forceEncrypt)`
 	
`public final void setMoney(int parameterIndex, BigDecimal x, boolean forceEncrypt)`
 	
`public final void setSmallMoney(int parameterIndex, BigDecimal x, boolean forceEncrypt)`
 	
`public final void setBoolean(int parameterIndex, boolean x, boolean forceEncrypt)`
	Sets the designated parameter to the given java value.
	
	If the boolean forceEncrypt is set to true, the query parameter will only be set if the designation column is encrypted and Always Encrypted is enabled on the connection or on the statement.
	
	If the boolean forceEncrypt is set to false, the driver won't force encryption on parameters.

`boolean x, boolean forceEncrypt)`

NAME	DESCRIPTION
`public final void setByte(int parameterIndex, byte x, boolean forceEncrypt)`	
`public final void setBytes(int parameterIndex, byte x[], boolean forceEncrypt)`	
`public final void setUniqueIdentifier(int parameterIndex, String guid, boolean forceEncrypt)`	
`public final void setDouble(int parameterIndex, double x, boolean forceEncrypt)`	
`public final void setFloat(int parameterIndex, float x, boolean forceEncrypt)`	
`public final void setInt(int parameterIndex, int value, boolean forceEncrypt)`	
`public final void setLong(int parameterIndex, long x, boolean forceEncrypt)`	
`public final setObject(int parameterIndex, Object x, int targetSqlType, Integer precision, int scale, boolean forceEncrypt)`	
`public final void setObject(int parameterIndex, Object x, SQLType targetSqlType, Integer precision, Integer scale, boolean forceEncrypt)`	
`public final void setShort(int parameterIndex, short x, boolean forceEncrypt)`	
`public final void setString(int parameterIndex, String str, boolean forceEncrypt)`	
`public final void setNString(int parameterIndex, String value, boolean forceEncrypt)`	
`public final void setTime(int parameterIndex, java.sql.Time x, int scale, boolean forceEncrypt)`	
`public final void setTimestamp(int parameterIndex, java.sql.Timestamp x, int scale, boolean forceEncrypt)`	
`public final void setDateTimeOffset(int parameterIndex, microsoft.sql.DateTimeOffset x, int scale, boolean forceEncrypt)`	
`public final void setDateTime(int parameterIndex, java.sql.Timestamp x, boolean forceEncrypt)`	
`public final void setSmallDateTime(int parameterIndex, java.sql.Timestamp x, boolean forceEncrypt)`	
`public final void setDate(int parameterIndex, java.sql.Date x, java.util.Calendar cal, boolean forceEncrypt)`	
`public final void setTime(int parameterIndex, java.sql.Time x, java.util.Calendar cal, boolean forceEncrypt)`	
`public final void setTimestamp(int parameterIndex, java.sql.Timestamp x, java.util.Calendar cal, boolean forceEncrypt)`	

New or overloaded methods in **SQLServerCallableStatement** Class

NAME	DESCRIPTION

`public void registerOutParameter(int parameterIndex, int sqlType, int precision, int scale)`
 	
`public void registerOutParameter(int parameterIndex, SQLType sqlType, int precision, int scale)`
 	
`public void registerOutParameter(String parameterName, int sqlType, int precision, int scale)`
 	
`public void registerOutParameter(String parameterName, SQLType sqlType, int precision, int scale)`
`public void setBigDecimal(String parameterName, BigDecimal bd, int precision, int scale)`
 	
`public void setTime(String parameterName, java.sql.Time t, int scale)`
 	
`public void setTimestamp(String parameterName, java.sql.Timestamp t, int scale)`
 	
`public void setDateTimeOffset(String parameterName, microsoft.sql.DateTimeOffset t, int scale)`
 	
`public final void setObject(String sCol, Object x, int targetSqlType, Integer precision, int scale)`
	These methods are overloaded with a precision or a scale argument or both to support Always Encrypted for specific data types that require precision and scale information.

NAME	DESCRIPTION

`public void setDateTime(String parameterName, java.sql.Timestamp x)`
 	
`public void setSmallDateTime(String parameterName, java.sql.Timestamp x)`
 	
`public void setUniqueIdentifier(String parameterName, String guid)`
 	
`public void setMoney(String parameterName, BigDecimal bd)`
 	
`public void setSmallMoney(String parameterName, BigDecimal bd)`
 	
`public Timestamp getDateTime(int index)`
 	
`public Timestamp getDateTime(String sCol)`
 	
`public Timestamp getDateTime(int index, Calendar cal)`
 	
`public Timestamp getSmallDateTime(int index)`
 	
`public Timestamp getSmallDateTime(String sCol)`
 	
`public Timestamp getSmallDateTime(int index, Calendar cal)`
 	
`public Timestamp getSmallDateTime(String name, Calendar cal)`
 	
`public BigDecimal getMoney(int index)`
 	
`public BigDecimal getMoney(String sCol)`
 	
`public BigDecimal getSmallMoney(int index)`
 	
`public BigDecimal getSmallMoney(String sCol)`
	These methods are added to support Always Encrypted for the data types money, smallmoney, uniqueidentifier, datetime and smalldatetime.
	
	Note that the existing `setTimestamp()` method is used for sending parameter values to the encrypted datetime2 column. For encrypted datetime and smalldatetime columns use the new methods `setDateTime()` and `setSmallDateTime()` respectively.

`public void setObject(String parameterName, Object o, int n, int m, boolean forceEncrypt)`
 	
`public void setObject(String parameterName, Object obj, SQLType jdbcType, int scale, boolean forceEncrypt)`
 	
`public void setDate(String parameterName, java.sql.Date x, Calendar c, boolean forceEncrypt)`
 	
`public void setTime(String parameterName, java.sql.Time t, int scale, boolean forceEncrypt)`
 	
`public void setTime(String parameterName, java.sql.Time x, Calendar c, boolean forceEncrypt)`
 	
`public void setDateTime(String parameterName, java.sql.Timestamp x, boolean forceEncrypt)`
 	
`public void setDateTimeOffset(String parameterName, microsoft.sql.DateTimeOffset t, int scale, boolean forceEncrypt)`
	Sets the designated parameter to the given java value.
	
	If the boolean forceEncrypt is set to true, the query parameter will only be set if the designation column is encrypted and Always Encrypted is enabled on the connection or on the statement.
	
	If the boolean forceEncrypt is set to false, the driver won't force encryption on parameters.

NAME	DESCRIPTION
`public void setSmallDateTime(String parameterName, java.sql.Timestamp x, boolean forceEncrypt)`	
`public void setTimestamp(String parameterName, java.sql.Timestamp t, int scale, boolean forceEncrypt)`	
`public void setTimestamp(String parameterName, java.sql.Timestamp x, boolean forceEncrypt)`	
`public void setUniqueIdentifier(String parameterName, String guid, boolean forceEncrypt)`	
`public void setBytes(String parameterName, byte[] b, boolean forceEncrypt)`	
`public void setByte(String parameterName, byte b, boolean forceEncrypt)`	
`public void setString(String parameterName, String s, boolean forceEncrypt)`	
`public final void setNString(String parameterName, String value, boolean forceEncrypt)<br /><br /> public void setMoney(String parameterName, BigDecimal bd, boolean forceEncrypt)`	
`public void setSmallMoney(String parameterName, BigDecimal bd, boolean forceEncrypt)`	
`public void setBigDecimal(String parameterName, BigDecimal bd, int precision, int scale, boolean forceEncrypt)`	
`public void setDouble(String parameterName, double d, boolean forceEncrypt)`	
`public void setFloat(String parameterName, float f, boolean forceEncrypt)`	
`public void setInt(String parameterName, int i, boolean forceEncrypt)`	
`public void setLong(String parameterName, long l, boolean forceEncrypt)`	
`public void setShort(String parameterName, short s, boolean forceEncrypt)`	
`public void setBoolean(String parameterNames, boolean b, boolean forceEncrypt)`	
`public void setTimeStamp(String sCol, java.sql.Timestamp x, Calendar c, Boolean forceEncrypt)`	

New or overloaded methods in **SQLServerResultSet** Class

NAME	DESCRIPTION

`public String getUniqueIdentifier(int columnIndex)`
`public String getUniqueIdentifier(String columnLabel)`
`public java.sql.Timestamp getDateTime(int columnIndex)`
`public java.sql.Timestamp getDateTime(String columnName)`
`public java.sql.Timestamp getDateTime(int columnIndex, Calendar cal)`
`public java.sql.Timestamp getDateTime(String colName, Calendar cal)`
`public java.sql.Timestamp getSmallDateTime(int columnIndex)`
`public java.sql.Timestamp getSmallDateTime(String columnName)`
`public java.sql.Timestamp getSmallDateTime(int columnIndex, Calendar cal)`
`public java.sql.Timestamp getSmallDateTime(String colName, Calendar cal)`
`public BigDecimal getMoney(int columnIndex)`
`public BigDecimal getMoney(String columnName)`
`public BigDecimal getSmallMoney(int columnIndex)`
`public BigDecimal getSmallMoney(String columnName)`
`public void updateMoney(String columnName, BigDecimal x)`
`public void updateSmallMoney(String columnName, BigDecimal x)`
`public void updateDateTime(int index, java.sql.Timestamp x)`
`public void updateSmallDateTime(int index, java.sql.Timestamp x)`
	These methods are added to support Always Encrypted for the data types money, smallmoney, uniqueidentifier, datetime, and smalldatetime.
	
	Note that the existing `updateTimestamp()` method is used for updating encrypted datetime2 columns. For encrypted datetime and smalldatetime columns use the new methods `updateDateTime()` and `updateSmallDateTime()` respectively.

`public void updateBoolean(int index, boolean x, boolean forceEncrypt)`
`public void updateByte(int index, byte x, boolean forceEncrypt)`
`public void updateShort(int index, short x, boolean forceEncrypt)`
`public void updateInt(int index, int x, boolean forceEncrypt)`
`public void updateLong(int index, long x, boolean forceEncrypt)`
	Update the designated column to the given java value.
	
	If the boolean forceEncrypt is set to true, the column will only be set if it's encrypted and Always Encrypted is enabled on the connection or on the statement.
	
	If the boolean forceEncrypt is set to false, the driver won't force encryption on parameters.

NAME	DESCRIPTION
`public void updateFloat(int index, float x, boolean forceEncrypt)`	
`public void updateDouble(int index, double x, boolean forceEncrypt)`	
`public void updateMoney(int index, BigDecimal x, boolean forceEncrypt)`	
`public void updateMoney(String columnName, BigDecimal x, boolean forceEncrypt)`	
`public void updateSmallMoney(int index, BigDecimal x, boolean forceEncrypt)`	
`public void updateSmallMoney(String columnName, BigDecimal x, boolean forceEncrypt)`	
`public void updateBigDecimal(int index, BigDecimal x, Integer precision, Integer scale, boolean forceEncrypt)`	
`public void updateString(int columnIndex, String stringValue, boolean forceEncrypt)`	
`public void updateNString(int columnIndex, String nString, boolean forceEncrypt)`	
`public void updateNString(String columnLabel, String nString, boolean forceEncrypt)`	
`public void updateBytes(int index, byte x[], boolean forceEncrypt) <br/><br/> public void updateDate(int index, java.sql.Date x, boolean forceEncrypt)`	
`public void updateTime(int index, java.sql.Time x, Integer scale, boolean forceEncrypt)`	
`public void updateTimestamp(int index, java.sql.Timestamp x, int scale, boolean forceEncrypt)`	
`public void updateDateTime(int index, java.sql.Timestamp x, Integer scale, boolean forceEncrypt)`	
`public void updateSmallDateTime(int index, java.sql.Timestamp x, Integer scale, boolean forceEncrypt)`	
`public void updateDateTimeOffset(int index, microsoft.sql.DateTimeOffset x, Integer scale, boolean forceEncrypt)`	
`public void updateUniqueIdentifier(int index, String x, boolean forceEncrypt)`	
`public void updateObject(int index, Object x, int precision, int scale, boolean forceEncrypt)`	
`public void updateObject(int index, Object obj, SQLType targetSqlType, int scale, boolean forceEncrypt)`	
`public void updateBoolean(String columnName, boolean x, boolean forceEncrypt)`	
`public void updateByte(String columnName, byte x, boolean forceEncrypt)`	

NAME	DESCRIPTION

`public void updateShort(String columnName, short x, boolean forceEncrypt)`
 	
`public void updateInt(String columnName, int x, boolean forceEncrypt)`
 	
`public void updateLong(String columnName, long x, boolean forceEncrypt)`
 	
`public void updateFloat(String columnName, float x, boolean forceEncrypt)`
 	
`public void updateDouble(String columnName, double x, boolean forceEncrypt) <br/><br/> public void updateBigDecimal(String columnName, BigDecimal x, boolean forceEncrypt)`
 	
`public void updateBigDecimal(String columnName, BigDecimal x, Integer precision, Integer scale, boolean forceEncrypt)`
 	
`public void updateString(String columnName, String x, boolean forceEncrypt)`
 	
`public void updateBytes(String columnName, byte x[], boolean forceEncrypt)`
 	
`public void updateDate(String columnName, java.sql.Date x, boolean forceEncrypt)`
 	
`public void updateTime(String columnName, java.sql.Time x, int scale, boolean forceEncrypt)`
 	
`public void updateTimestamp(String columnName, java.sql.Timestamp x, int scale, boolean forceEncrypt)`
 	
`public void updateDateTime(String columnName, java.sql.Timestamp x, int scale, boolean forceEncrypt)`
 	
`public void updateSmallDateTime(String columnName, java.sql.Timestamp x, int scale, boolean forceEncrypt)`
 	
`public void updateDateTimeOffset(String columnName, microsoft.sql.DateTimeOffset x, int scale, boolean forceEncrypt)`
 	
`public void updateUniqueIdentifier(String columnName, String x, boolean forceEncrypt)`
 	
`public void updateObject(String columnName, Object x, int precision, int scale, boolean forceEncrypt)`
 	
`public void updateObject(String columnName, Object obj, SQLType targetSqlType, int scale, boolean forceEncrypt)`	

New types in **microsoft.sql.Types** class

NAME	DESCRIPTION
DATETIME, SMALLDATETIME, MONEY, SMALLMONEY, GUID	Use these types as the target SQL types when sending parameter values to **encrypted** datetime, smalldatetime, money, smallmoney, uniqueidentifier columns using `setObject()/updateObject()` API methods.

SQLServerStatementColumnEncryptionSetting Enum

Specifies how data will be sent and received when reading and writing encrypted columns. Depending on your specific query, performance impact may be reduced by bypassing the Always Encrypted driver's processing when non-encrypted columns are being used. Note that these settings can't be used to bypass encryption and gain access to plaintext data.

Syntax

```
Public enum  SQLServerStatementColumnEncryptionSetting
```

Members

NAME	DESCRIPTION
UseConnectionSetting	Specifies that the command should default to the Always Encrypted setting in the connection string.
Enabled	Enables Always Encrypted for the query.
ResultSetOnly	Specifies that only the results of the command should be processed by the Always Encrypted routine in the driver. Use this value when the command has no parameters that require encryption.
Disabled	Disables Always Encrypted for the query.

The statement level setting for AE is added to the SQLServerConnection class and to the SQLServerConnectionPoolProxy class. The following methods in these classes are overloaded with the new setting.

NAME	DESCRIPTION
`public Statement createStatement(int nType, int nConcur, int statementHoldability, SQLServerStatementColumnEncryptionSetting stmtColEncSetting)`	Creates a Statement object that will generate ResultSet objects with the given type, concurrency, holdability, and column encryption setting.
`public CallableStatement prepareCall(String sql, int nType, int nConcur, int statementHoldability, SQLServerStatementColumnEncryptionSetting stmtColEncSetiing)`	Creates a CallableStatement object with the given column encryption setting that will generate ResultSet objects with the given type, concurrency, and holdability.
`public PreparedStatement prepareStatement(String sql, int autogeneratedKeys, SQLServerStatementColumnEncryptionSetting stmtColEncSetting)`	Creates a PreparedStatement object with the given column encryption setting that has the capability to retrieve auto-generated keys.
`public PreparedStatement prepareStatement(String sql, String[] columnNames, SQLServerStatementColumnEncryptionSetting stmtColEncSetting)`	Creates a PreparedStatement object with the given column encryption setting that will generate ResultSet objects with the given column names.
`public PreparedStatement prepareStatement(String sql, int[] columnIndexes, SQLServerStatementColumnEncryptionSetting stmtColEncSetting`	Creates a PreparedStatement object with the given column encryption setting that will generate ResultSet objects with the given column indexes.

NAME	DESCRIPTION
`public PreparedStatement prepareStatement(String sql, int nType, int nConcur, int nHold, SQLServerStatementColumnEncryptionSetting stmtColEncSetting)`	Creates a PreparedStatement object with the given column encryption setting that will generate ResultSet objects with the given type, concurrency, and holdability.

NOTE

If Always Encrypted is disabled for a query and the query has parameters that need to be encrypted (parameters that correspond to encrypted columns), the query will fail.

If Always Encrypted is disabled for a query and the query returns results from encrypted columns, the query will return encrypted values. The encrypted values will have the varbinary datatype.

See Also

Using Always Encrypted with the JDBC Driver

Using Kerberos Integrated Authentication to Connect to SQL Server

8/8/2018 • 6 minutes to read • Edit Online

Download JDBC Driver

Beginning in Microsoft JDBC Driver 4.0 for SQL Server, an application can use the **authenticationScheme** connection property to indicate that it wants to connect to a database using type 4 Kerberos integrated authentication. See Setting the Connection Properties for more information on connection properties. For more information on Kerberos, see Microsoft Kerberos.

When using integrated authentication with the Java **Krb5LoginModule**, you can configure the module using Class Krb5LoginModule.

The Microsoft JDBC Driver for SQL Server sets the following properties for IBM Java VMs:

- **useDefaultCcache = true**
- **moduleBanner = false**

The Microsoft JDBC Driver for SQL Server sets the following properties for all other Java VMs:

- **useTicketCache = true**
- **doNotPrompt = true**

Remarks

Prior to Microsoft JDBC Driver 4.0 for SQL Server, applications could specify integrated authentication (using Kerberos or NTLM, depending on which is available) by using the **integratedSecurity** connection property and by referencing **sqljdbc_auth.dll**, as described in Building the Connection URL.

Beginning in Microsoft JDBC Driver 4.0 for SQL Server, an application can use the **authenticationScheme** connection property to indicate that it wants to connect to a database using Kerberos integrated authentication using the pure Java Kerberos implementation:

- If you want integrated authentication using **Krb5LoginModule**, you must still specify the **integratedSecurity=true** connection property. You would then also specify the **authenticationScheme=JavaKerberos** connection property.

- To continue using integrated authentication with **sqljdbc_auth.dll**, just specify **integratedSecurity=true** connection property (and optionally **authenticationScheme=NativeAuthentication**).

- If you specify **authenticationScheme=JavaKerberos** but do not also specify **integratedSecurity=true**, the driver will ignore the **authenticationScheme** connection property and it will expect to find user name and password credentials in the connection string.

When using a datasource to create connections, you can programmatically set the authentication scheme using setAuthenticationScheme and (optionally) set the SPN for Kerberos connections using **setServerSpn**.

A new logger has been added to support Kerberos authentication: com.microsoft.sqlserver.jdbc.internals.KerbAuthentication. For more information, see Tracing Driver Operation.

The following guidelines will help you to configure Kerberos:

1. Set **AllowTgtSessionKey** to 1 in the registry for Windows. For more information, see Kerberos protocol registry entries and KDC configuration keys in Windows Server 2003.
2. Make sure that the Kerberos configuration (krb5.conf in UNIX environments), points to the correct realm and KDC for your environment.
3. Initialize the TGT cache by using kinit or logging into the domain.
4. When an application that uses **authenticationScheme=JavaKerberos** runs on the Windows Vista or Windows 7 operating systems, you should use a standard user account. However if you run the application under an administrator's account, the application must run with administrator privileges.

> **NOTE**
>
> The serverSpn connection attribute is only supported by Microsoft JDBC Drivers 4.2 and higher.

Service Principal Names

A service principal name (SPN) is the name by which a client uniquely identifies an instance of a service.

You can specify the SPN using the **serverSpn** connection property, or simply let the driver build it for you (the default). This property is in the form of: "MSSQLSvc/fqdn:port@REALM" where fqdn is the fully-qualified domain name, port is the port number, and REALM is the Kerberos realm of the SQL Server in upper-case letters. The realm portion of this property is optional if your Kerberos configuration's default realm is the same realm as that of the Server and is not included by default. If you wish to support a cross-realm authentication scenario where the default realm in the Kerberos configuration is different than the realm of the Server, then you must set the SPN with the serverSpn property.

For example, your SPN might look like: "MSSQLSvc/some-server.zzz.corp.contoso.com:1433@ZZZZ.CORP.CONTOSO.COM"

For more information about service principal names (SPNs), see:

- How to use Kerberos authentication in SQL Server
- Using Kerberos with SQL Server

> **NOTE**
>
> Before 6.2 release of JDBC driver, for proper use of Cross Realm Kerberos, you would need to explicitly set the **serverSpn**.
>
> As of the 6.2 release, the driver will be able to build the **serverSpn** by default, even when using Cross Realm Kerberos. Although one can use **serverSpn** explicitly too.

Creating a Login Module Configuration File

You can optionally specify a Kerberos configuration file. If a configuration file is not specified, the following settings are in effect:

Sun JVM
com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;

IBM JVM
com.ibm.security.auth.module.Krb5LoginModule required useDefaultCcache = true;

If you decide to create a login module configuration file, the file must follow this format:

```
<name> {
    <LoginModule> <flag> <LoginModule options>;
    <optional_additional_LoginModules, flags_and_options>;
};
```

A login configuration file consists of one or more entries, each specifying which underlying authentication technology should be used for a particular application or applications. For example,

```
SQLJDBCDriver {
   com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;
};
```

So, each login module configuration file entry consists of a name followed by one or more LoginModule-specific entries, where each LoginModule-specific entry is terminated by a semicolon and the entire group of LoginModule-specific entries is enclosed in braces. Each configuration file entry is terminated by a semicolon.

In addition to allowing the driver to acquire Kerberos credentials using the settings specified in the login module configuration file, the driver can use existing credentials. This can be useful when your application needs to create connections using more than one user's credentials.

The driver will attempt to use existing credentials if they are available, before attempting to login using the specified login module. Thus, when using the Subject.doAs method for executing code under a specific context, a connection will be created with the credentials passed to the Subject.doAs call.

For more information, see JAAS Login Configuration File and Class Krb5LoginModule.

Beginning in Microsoft JDBC Driver 6.2, name of login module configuration file can optionally be passed using connection property jaasConfigurationName, this allows each connection to have its own login configuration.

Creating a Kerberos Configuration File

For more information about Kerberos configuration files, see Kerberos Requirements.

This is a sample domain configuration file, where YYYY and ZZZZ are domain names at your site.

```
[libdefaults]
default_realm = YYYY.CORP.CONTOSO.COM
dns_lookup_realm = false
dns_lookup_kdc = true
ticket_lifetime = 24h
forwardable = yes

[domain_realm]
.yyyy.corp.contoso.com = YYYY.CORP.CONTOSO.COM
.zzzz.corp.contoso.com = ZZZZ.CORP.CONTOSO.COM

[realms]
        YYYY.CORP.CONTOSO.COM = {
  kdc = krbtgt/YYYY.CORP. CONTOSO.COM @ YYYY.CORP. CONTOSO.COM
  default_domain = YYYY.CORP. CONTOSO.COM
}

        ZZZZ.CORP. CONTOSO.COM = {
  kdc = krbtgt/ZZZZ.CORP. CONTOSO.COM @ ZZZZ.CORP. CONTOSO.COM
  default_domain = ZZZZ.CORP. CONTOSO.COM
}
```

Enabling the Domain Configuration File and the Login Module

Configuration File

You can enable a domain configuration file with -Djava.security.krb5.conf. You can enable a login module configuration file with **-Djava.security.auth.login.config**.

For example, when you start your application, you could use this command line:

```
Java.exe -Djava.security.auth.login.config=SQLJDBCDriver.conf -Djava.security.krb5.conf=krb5.ini
<APPLICATION_NAME>
```

Verifying that SQL Server Can be Accessed via Kerberos

Run the following query in SQL Server Management Studio:

```
select auth_scheme from sys.dm_exec_connections where session_id=\@\@spid
```

Make sure that you have the necessary permission to run this query.

Constrained Delegation

Beginning in Microsoft JDBC Driver 6.2, the driver supports Kerberos Constrained Delegation. The delegated credential can be passed in as org.ietf.jgss.GSSCredential object, these credentials are used by driver to establish connection.

```
Properties driverProperties = new Properties();
GSSCredential impersonatedUserCredential = [userCredential]
driverProperties.setProperty("integratedSecurity", "true");
driverProperties.setProperty("authenticationScheme", "JavaKerberos");
driverProperties.put("gsscredential", impersonatedUserCredential);
Connection conn = DriverManager.getConnection(CONNECTION_URI, driverProperties);
```

Kerberos Connection using Principal Names and Password

Beginning in Microsoft JDBC Driver 6.2, the driver can establish Kerberos connection using the Principal Name and Password passed in connection string.

```
jdbc:sqlserver://servername=server_name;integratedSecurity=true;authenticationScheme=JavaKerberos;userName=us
er@REALM;password=****
```

The username property does not require REALM if user belongs to the default_realm set in krb5.conf file. When `userName` and `password` is set along with `integratedSecurity=true;` and `authenticationScheme=JavaKerberos;` property, the connection is established with value of userName as Kerberos Principal along with the password supplied.

See Also

Connecting to SQL Server with the JDBC Driver

Using a Stored Procedure with an Update Count

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

To modify data in a SQL Server database by using a stored procedure, the Microsoft JDBC Driver for SQL Server provides the SQLServerCallableStatement class. By using the SQLServerCallableStatement class, you can call stored procedures that modify data that is in the database and return a count of the number of rows affected, also referred to as the update count.

After you have set up the call to the stored procedure by using the SQLServerCallableStatement class, you can then call the stored procedure by using either the execute or the executeUpdate method. The executeUpdate method will return an **int** value that contains the number of rows affected by the stored procedure, but the execute method doesn't. If you use the execute method and want to get the count of the number of rows affected, you can call the getUpdateCount method after you run the stored procedure.

NOTE

If you want the JDBC driver to return all update counts, including update counts returned by any triggers that may have fired, set the lastUpdateCount connection string property to "false". For more information about the lastUpdateCount property, see Setting the Connection Properties.

As an example, create the following table and stored procedure, and also insert sample data in the AdventureWorks sample database:

```
CREATE TABLE TestTable
   (Col1 int IDENTITY,
    Col2 varchar(50),
    Col3 int);

CREATE PROCEDURE UpdateTestTable
   @Col2 varchar(50),
   @Col3 int
AS
BEGIN
   UPDATE TestTable
   SET Col2 = @Col2, Col3 = @Col3
END;
INSERT INTO dbo.TestTable (Col2, Col3) VALUES ('b', 10);
```

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, the execute method is used to call the UpdateTestTable stored procedure, and then the getUpdateCount method is used to return a count of the rows that are affected by the stored procedure.

```
public static void executeUpdateStoredProcedure(Connection con) {
    try(CallableStatement cstmt = con.prepareCall("{call dbo.UpdateTestTable(?, ?)}");) {
        cstmt.setString(1, "A");
        cstmt.setInt(2, 100);
        cstmt.execute();
        int count = cstmt.getUpdateCount();
        System.out.println("ROWS AFFECTED: " + count);
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

See Also

Using Statements with Stored Procedures

Understanding Row Locking

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server uses SQL Server row locks. These implement concurrency controls among multiple users who are performing modifications in a database at the same time. By default, transactions and locks are managed on a per-connection basis. For example, if an application opens two JDBC connections, locks that are acquired by one connection cannot be shared with the other connection. Neither connection can acquire locks that would conflict with locks held by the other connection.

NOTE

If row locking is used, all rows in the fetch buffer are locked, so a very large setting for the fetch size can affect concurrency.

Locking is used to assure transactional integrity and database consistency. Locking prevents users from reading data that is being changed by other users, and prevents multiple users from changing the same data at the same time. If locking is not used, data within the database might become logically incorrect, and queries run against that data might produce unexpected results.

NOTE

For more information about row locking in SQL Server, see "Locking in the Database Engine" in SQL Server Books Online.

See Also

Managing Result Sets with the JDBC Driver

JDBC Driver Support for High Availability, Disaster Recovery

8/13/2018 • 7 minutes to read • Edit Online

Download JDBC Driver

This topic discusses Microsoft JDBC Driver for SQL Server support for high-availability, disaster recovery -- AlwaysOn Availability Groups. For more information about AlwaysOn Availability Groups, see SQL Server 2012 (11.x) Books Online.

Beginning in version 4.0 of the Microsoft JDBC Driver for SQL Server, you can specify the availability group listener of a (high-availability, disaster-recovery) availability group (AG) in the connection property. If a Microsoft JDBC Driver for SQL Server application is connected to an AlwaysOn database that fails over, the original connection is broken and the application must open a new connection to continue work after the failover. The following connection properties were added in Microsoft JDBC Driver 4.0 for SQL Server:

- **multiSubnetFailover**
- **applicationIntent**

Specify multiSubnetFailover=true when connecting to the availability group listener of an availability group or a Failover Cluster Instance. Note that **multiSubnetFailover** is false by default. Use **applicationIntent** to declare the application workload type. See sections below for more details.

Beginning in version 6.0 of the Microsoft JDBC Driver for SQL Server, a new connection property **transparentNetworkIPResolution** (TNIR) is added for transparent connection to Always On availability groups or to a server which has multiple IP addresses associated. When **transparentNetworkIPResolution** is true, the driver attempts to connect to the first IP address available. If the first attempt fails, the driver tries to connect to all IP addresses in parallel until the timeout expires, discarding any pending connection attempts when one of them succeeds.

Please note that:

- transparentNetworkIPResolution is true by default
- transparentNetworkIPResolution is ignored if multiSubnetFailover is true
- transparentNetworkIPResolution is ignored if database mirroring is used
- transparentNetworkIPResolution is ignored if there are more than 64 IP addresses
- When transparentNetworkIPResolution is true, the first connection attempt uses a timeout value of 500ms. Rest of the connection attempts follow the same logic as in the multiSubnetFailover feature.

NOTE

If you are using Microsoft JDBC Driver 4.2 (or lower) for SQL Server and if **multiSubnetFailover** is false, the Microsoft JDBC Driver for SQL Server attempts to connect to the first IP address. If the Microsoft JDBC Driver for SQL Server cannot establish a connection with first IP address, the connection fails. The Microsoft JDBC Driver for SQL Server will not attempt to connect to any subsequent IP address associated with the server.

NOTE

Increasing connection timeout and implementing connection retry logic will increase the probability that an application will connect to an availability group. Also, because a connection can fail because of an availability group failover, you should implement connection retry logic, retrying a failed connection until it reconnects.

Connecting With MultiSubnetFailover

Always specify **multiSubnetFailover=true** when connecting to the availability group listener of a SQL Server 2012 (11.x) availability group or a SQL Server 2012 (11.x) Failover Cluster Instance. **multiSubnetFailover** enables faster failover for all Availability Groups and failover cluster instances in SQL Server 2012 (11.x) and will significantly reduce failover time for single and multi-subnet AlwaysOn topologies. During a multi-subnet failover, the client will attempt connections in parallel. During a subnet failover, the Microsoft JDBC Driver for SQL Server will aggressively retry the TCP connection.

The **multiSubnetFailover** connection property indicates that the application is being deployed in an availability group or Failover Cluster Instance and that the Microsoft JDBC Driver for SQL Server will try to connect to the database on the primary SQL Server instance by trying to connect to all the IP addresses. When **MultiSubnetFailover=true** is specified for a connection, the client retries TCP connection attempts faster than the operating system's default TCP retransmit intervals. This enables faster reconnection after failover of either an AlwaysOn Availability Group or an AlwaysOn Failover Cluster Instance, and is applicable to both single- and multi-subnet Availability Groups and Failover Cluster Instances.

For more information about connection string keywords in the Microsoft JDBC Driver for SQL Server, see Setting the Connection Properties.

Specifying **multiSubnetFailover=true** when connecting to something other than an availability group listener or Failover Cluster Instance may result in a negative performance impact, and is not supported.

If the security manager is not installed, the Java Virtual Machine caches virtual IP addresses (VIPs) for a finite period of time, by default, defined by your JDK implementation and the Java properties networkaddress.cache.ttl and networkaddress.cache.negative.ttl. If the JDK security manager is installed, the Java Virtual Machine will cache VIPs, and will not refresh the cache by default. You should set "time-to-live" (networkaddress.cache.ttl) to one day for the Java Virtual Machine cache. If you don't change the default value to one day (or so), the old value will not be purged from the Java Virtual Machine cache when a VIP is added or updated. For more information about networkaddress.cache.ttl and networkaddress.cache.negative.ttl, see http://download.oracle.com/javase/6/docs/technotes/guides/net/properties.html.

Use the following guidelines to connect to a server in an availability group or Failover Cluster Instance:

- The driver will generate an error if the **instanceName** connection property is used in the same connection string as the **multiSubnetFailover** connection property. This reflects the fact that SQL Browser is not used in an availability group. However, if the **portNumber** connection property is also specified, the driver will ignore **instanceName** and use **portNumber**.
- Use the **multiSubnetFailover** connection property when connecting to a single subnet or multi-subnet, it will improve performance for both.
- To connect to an availability group, specify the availability group listener of the availability group as the server in your connection string. For example, jdbc:sqlserver://VNN1.
- Connecting to a SQL Server instance configured with more than 64 IP addresses will cause a connection failure.
- Behavior of an application that uses the **multiSubnetFailover** connection property is not affected based on the type of authentication: SQL Server Authentication, Kerberos Authentication, or Windows

Authentication.

- Increase the value of **loginTimeout** to accommodate for failover time and reduce application connection retry attempts.
- Distributed transactions are not supported.

  If read-only routing is not in effect, connecting to a secondary replica location in an availability group will fail in the following situations:

1. If the secondary replica location is not configured to accept connections.
2. If an application uses **applicationIntent=ReadWrite** (discussed below) and the secondary replica location is configured for read-only access.

   A connection will fail if a primary replica is configured to reject read-only workloads and the connection string contains **ApplicationIntent=ReadOnly**.

Upgrading to Use Multi-Subnet Clusters from Database Mirroring

If you upgrade a Microsoft JDBC Driver for SQL Server application that currently uses database mirroring to a multi-subnet scenario, you should remove the **failoverPartner** connection property and replace it with **multiSubnetFailover** set to **true** and replace the server name in the connection string with a availability group listener. If a connection string uses **failoverPartner** and **multiSubnetFailover=true**, the driver will generate an error. However, if a connection string uses **failoverPartner** and **multiSubnetFailover=false** (or **ApplicationIntent=ReadWrite**), the application will use database mirroring.

The driver will return an error if database mirroring is used on the primary database in the AG, and if **multiSubnetFailover=true** is used in the connection string that connects to a primary database instead of to an availability group listener.

Specifying Application Intent

The keyword **ApplicationIntent** can be specified in your connection string. The assignable values are **ReadWrite** or **ReadOnly**. The default is **ReadWrite**.

When **ApplicationIntent=ReadOnly**, the client requests a read workload when connecting. The server enforces the intent at connection time, and during a **USE** database statement.

The **ApplicationIntent** keyword does not work with legacy read-only databases.

Targets of ReadOnly

When a connection chooses **ReadOnly**, the connection is assigned to any of the following special configurations that might exist for the database:

- Always On
  - A database can allow or disallow read workloads on the targeted Always On database. This choice is controlled by using the **ALLOW_CONNECTIONS** clause of the **PRIMARY_ROLE** and **SECONDARY_ROLE** Transact-SQL statements.
- Geo-Replication
- Read Scale-Out

If none of those special targets are available, the regular database is read from.

The **ApplicationIntent** keyword enables *read-only routing*.

Read-Only Routing

Read-only routing is a feature that can ensure the availability of a read-only replica of a database. To enable read-only routing, all of the following apply:

- You must connect to an Always On Availability Group availability group listener.
- The **ApplicationIntent** connection string keyword must be set to **ReadOnly**.
- The Availability Group must be configured by the database administrator to enable read-only routing.

Multiple connections each using read-only routing might not all connect to the same read-only replica. Changes in database synchronization or changes in the server's routing configuration can result in client connections to different read-only replicas. You can ensure that all read-only requests connect to the same read-only replica. Ensure this sameness by *not* passing an availability group listener to the **Server** connection string keyword. Instead, specify the name of the read-only instance.

Read-only routing may take longer than connecting to the primary. The longer wait is because read-only routing first connects to the primary, and then looks for the best available readable secondary. Due to these multiple staps, you should increase your login timeout to to at least 30 seconds.

New Methods Supporting multiSubnetFailover and applicationIntent

The following methods give you programmatic access to the **multiSubnetFailover**, **applicationIntent** and **transparentNetworkIPResolution** connection string keywords:

- SQLServerDataSource.getApplicationIntent
- SQLServerDataSource.setApplicationIntent
- SQLServerDataSource.getMultiSubnetFailover
- SQLServerDataSource.setMultiSubnetFailover
- SQLServerDriver.getPropertyInfo
- SQLServerDataSource.setTransparentNetworkIPResolution
- SQLServerDataSource.getTransparentNetworkIPResolution

  The **getMultiSubnetFailover**, **setMultiSubnetFailover**, **getApplicationIntent**, **setApplicationIntent**, **getTransparentNetworkIPResolution** and **setTransparentNetworkIPResolution** methods are also added to SQLServerDataSource Class, SQLServerConnectionPoolDataSource Class, and SQLServerXADataSource Class.

SSL Certificate Validation

An availability group consists of multiple physical servers. Microsoft JDBC Driver 4.0 for SQL Server added support for **Subject Alternate Name** in SSL certificates so multiple hosts can be associated with the same certificate. For more information on SSL, see Understanding SSL Support.

See Also

Connecting to SQL Server with the JDBC Driver
Setting the Connection Properties

Using Result Set Metadata

8/8/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

To query a result set for information about the columns that it contains, the Microsoft JDBC Driver for SQL Server implements the SQLServerResultSetMetaData class. This class contains numerous methods that return information in the form of a single value.

To create a SQLServerResultSetMetaData object, you can use the getMetaData method of the SQLServerResultSet class.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, the getMetaData method of the SQLServerResultSet class is used to return a SQLServerResultSetMetaData object, and then various methods of the SQLServerResultSetMetaData object are used to display information about the name and data type of the columns contained within the result set.

```
public static void getResultSetMetaData(Connection con) {
    try(Statement stmt = con.createStatement();) {
        String SQL = "SELECT TOP 10 * FROM Person.Contact";

        ResultSet rs = stmt.executeQuery(SQL);
        ResultSetMetaData rsmd = rs.getMetaData();

        // Display the column name and type.
        int cols = rsmd.getColumnCount();
        for (int i = 1; i <= cols; i++) {
            System.out.println("NAME: " + rsmd.getColumnName(i) + " " + "TYPE: " + rsmd.getColumnTypeName(i));
        }
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

See Also

Handling Metadata with the JDBC Driver

Understanding XA Transactions

8/13/2018 • 10 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server provides support for Java Platform, Enterprise Edition/JDBC 2.0 optional distributed transactions. JDBC connections obtained from the SQLServerXADataSource class can participate in standard distributed transaction processing environments such as Java Platform, Enterprise Edition (Java EE) application servers.

WARNING

Microsoft JDBC Driver 4.2 (and higher) for SQL includes new timeout options for the existing feature for automatic rollback of unprepared transactions. See Configuring server-side timeout settings for automatic rollback of unprepared transactions later in this topic for more detail.

Remarks

The classes for the distributed transaction implementation are as follows:

CLASS	IMPLEMENTS	DESCRIPTION
com.microsoft.sqlserver.jdbc.SQLServerXADataSource	javax.sql.XADataSource	The class factory for distributed connections.
com.microsoft.sqlserver.jdbc.SQLServerXAResource	javax.transaction.xa.XAResource	The resource adaptor for the transaction manager.

NOTE

XA distributed transaction connections default to the Read Committed isolation level.

Guidelines and Limitations when Using XA Transactions

The following additional guidelines apply to tightly coupled transactions:

- When you use XA transactions together with Microsoft Distributed Transaction Coordinator (MS DTC), you may notice that the current version of MS DTC doesn't support tightly coupled XA branch behavior. For example, MS DTC has a one-to-one mapping between an XA branch transaction ID (XID) and an MS DTC transaction ID and the work that is performed by loosely coupled XA branches is isolated from one another.

  The hotfix provided at MSDTC and Tightly Coupled Transactions enables the support for tightly coupled XA branches where multiple XA branches with same global transaction ID (GTRID) are mapped to a single MS DTC transaction ID. This support enables multiple tightly coupled XA branches to see one another's changes in the resource manager, such as SQL Server.

- A SSTRANSTIGHTLYCPLD flag allows the applications to use tightly coupled XA transactions, which have different XA branch transaction IDs (BQUAL) but have the same global transaction ID (GTRID) and format ID (FormatID). In order to use that feature, you must set the SSTRANSTIGHTLYCPLD on the flags parameter of the XAResource.start method:

```
xaRes.start(xid, SQLServerXAResource.SSTRANSTIGHTLYCPLD);
```

Configuration Instructions

The following steps are required if you want to use XA data sources together with Microsoft Distributed Transaction Coordinator (MS DTC) for handling distributed transactions.

> **NOTE**
>
> The JDBC distributed transaction components are included in the xa directory of the JDBC driver installation. These components include the xa_install.sql and sqljdbc_xa.dll files.

Running the MS DTC Service

The MS DTC service should be marked **Automatic** in Service Manager to make sure that it is running when the SQL Server service is started. To enable MS DTC for XA transactions, you must follow these steps:

On Windows Vista and later:

1. Click the **Start** button, type **dcomcnfg** in the Start **Search** box, and then press ENTER to open **Component Services**. You can also type %windir%\system32\comexp.msc in the **StartSearch** box to open **Component Services**.
2. Expand Component Services, Computers, My Computer, and then Distributed Transaction Coordinator.
3. Right-click **Local DTC** and then select **Properties**.
4. Click the **Security** tab on the **Local DTC Properties** dialog box.
5. Select the **Enable XA Transactions** check box, and then click **OK**. This will cause a MS DTC service restart.
6. Click **OK** again to close the **Properties** dialog box, and then close **Component Services**.
7. Stop and then restart SQL Server to make sure that it syncs up with the MS DTC changes.

Configuring the JDBC Distributed Transaction Components

You can configure the JDBC driver distributed transaction components by following these steps:

1. Copy the new sqljdbc_xa.dll from the JDBC driver installation directory to the Binn directory of every SQL Server computer that will participate in distributed transactions.

   > **NOTE**
   >
   > If you are using XA transactions with a 32-bit SQL Server, use the sqljdbc_xa.dll file in the x86 folder, even if the SQL Server is installed on a x64 processor. If you are using XA transactions with a 64-bit SQL Server on the x64 processor, use the sqljdbc_xa.dll file in the x64 folder.

2. Execute the database script xa_install.sql on every SQL Server instance that will participate in distributed transactions. This script installs the extended stored procedures that are called by sqljdbc_xa.dll. These extended stored procedures implement distributed transaction and XA support for the Microsoft JDBC Driver for SQL Server. You'll need to run this script as an administrator of the SQL Server instance.
3. To grant permissions to a specific user to participate in distributed transactions with the JDBC driver, add the user to the SqlJDBCXAUser role.

You can configure only one version of the sqljdbc_xa.dll assembly on each SQL Server instance at a time. Applications may need to use different versions of the JDBC driver to connect to the same SQL Server instance by using the XA connection. In that case, sqljdbc_xa.dll, which comes with the newest JDBC driver, must be installed on the SQL Server instance.

There are three ways to verify the version of sqljdbc_xa.dll is currently installed on the SQL Server instance:

1. Open the LOG directory of SQL Server computer that will participate in distributed transactions. Select and open the SQL Server "ERRORLOG" file. Search for "Using 'SQLJDBC_XA.dll' version ..." phrase in the "ERRORLOG" file.

2. Open the Binn directory of SQL Server computer that will participate in distributed transactions. Select sqljdbc_xa.dll assembly.

   - On Windows Vista or later: Right-click sqljdbc_xa.dll and then select Properties. Then click the **Details** tab. The **File Version** field shows the version of sqljdbc_xa.dll that is currently installed on the SQL Server instance.

3. Set the logging functionality as shown in the code example in the next section. Search for "Server XA DLL version:..." phrase in the output log file.

Configuring server-side timeout settings for automatic rollback of unprepared transactions

WARNING

This server-side option is new with Microsoft JDBC Driver 4.2 (and higher) for SQL Server. To get the updated behavior, make sure the sqljdbc_xa.dll on the server is updated. For more information on setting client side timeouts, see XAResource.setTransactionTimeout().

There are two registry settings (DWORD values) to control the timeout behavior of distributed transactions:

- **XADefaultTimeout** (in seconds): The default timeout value to be used when the user does not specify any timeout. The default is 0.

- **XAMaxTimeout** (in seconds): The maximum value of the timeout that a user can set. The default is 0.

These settings are SQL Server instance specific and should be created under the following registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Microsoft SQL Server\MSSQL\<version>.<instance_name>\XATimeout
```

NOTE

For 32-bit SQL Server running in 64-bit machines, the registry settings should be created under the following key:
`HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Microsoft SQL Server\MSSQL\<version>.<instance_name>\XATimeout`

A timeout value is set for each transaction when it's started and the transaction is rolled back by the SQL Server if the timeout expires. The timeout is determined depending on these registry settings and depending on what the user has specified through XAResource.setTransactionTimeout(). A few examples on how these timeout values are interpreted as follows:

- `XADefaultTimeout = 0`, `XAMaxTimeout = 0`

  Means no default timeout will be used, and no maximum timeout will be enforced on clients. In this case the transactions will have a timeout only if the client sets a timeout using XAResource.setTransactionTimeout.

- `XADefaultTimeout = 60`, `XAMaxTimeout = 0`

  Means all transactions will have a 60 seconds timeout if the client doesn't specify any timeout. If the client specifies a timeout, then that timeout value will be used. No maximum value for timeout is enforced.

- `XADefaultTimeout = 30`, `XAMaxTimeout = 60`

  Means all transactions will have a 30 seconds timeout if the client doesn't specify any timeout. If client specifies any timeout, then the client's timeout will be used as long as it is less than 60 seconds (the max value).

- `XADefaultTimeout = 0`, `XAMaxTimeout = 30`

  Means all transactions will have a 30 seconds timeout (the max value) if the client does not specify any timeout. If the client specifies any timeout, then the client's timeout will be used as long as it is less than 30 seconds (the max value).

Upgrading sqljdbc_xa.dll

When you install a new version of the JDBC driver, you should also use sqljdbc_xa.dll from the new version to upgrade sqljdbc_xa.dll on the server.

> **IMPORTANT**
>
> You should upgrade sqljdbc_xa.dll during a maintenance window or when there are no MS DTC transactions in process.

1. Unload sqljdbc_xa.dll using the Transact-SQL command **DBCC sqljdbc_xa (FREE)**.

2. Copy the new sqljdbc_xa.dll from the JDBC driver installation directory to the Binn directory of every SQL Server computer that will participate in distributed transactions.

   The new DLL will be loaded when an extended procedure in sqljdbc_xa.dll is called. You don't need to restart SQL Server to load the new definitions.

Configuring the User-Defined Roles

To grant permissions to a specific user to participate in distributed transactions with the JDBC driver, add the user to the SqlJDBCXAUser role. For example, use the following Transact-SQL code to add a user named 'shelby' (SQL standard login user named 'shelby') to the SqlJDBCXAUser role:

```
USE master
GO
EXEC sp_grantdbaccess 'shelby', 'shelby'
GO
EXEC sp_addrolemember [SqlJDBCXAUser], 'shelby'
```

SQL user-defined roles are defined per database. To create your own role for security purposes, you'll have to define the role in each database, and add users in a per database manner. The SqlJDBCXAUser role is strictly defined in the master database because it's used to grant access to the SQL JDBC extended stored procedures that reside in master. You'll have to first grant individual users access to master, and then grant them access to the SqlJDBCXAUser role while you're logged into the master database.

Example

```
import java.net.Inet4Address;
import java.sql.*;
import java.util.Random;
import javax.sql.XAConnection;
import javax.transaction.xa.*;
import com.microsoft.sqlserver.jdbc.*;
```

```
public class testXA {

    public static void main(String[] args) throws Exception {

        // Create variables for the connection string.
        String prefix = "jdbc:sqlserver://";
        String serverName = "localhost";
        int portNumber = 1433;
        String databaseName = "AdventureWorks";
        String user = "UserName";
        String password = "*****";

        String connectionUrl = prefix + serverName + ":" + portNumber + ";databaseName=" + databaseName +
";user="
                + user + ";password=" + password;

        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

        try (Connection con = DriverManager.getConnection(connectionUrl); Statement stmt =
con.createStatement()) {
            stmt.executeUpdate("CREATE TABLE XAMin (f1 int, f2 varchar(max))");

        }
        // Create the XA data source and XA ready connection.
        SQLServerXADataSource ds = new SQLServerXADataSource();
        ds.setUser(user);
        ds.setPassword(password);
        ds.setServerName(serverName);
        ds.setPortNumber(portNumber);
        ds.setDatabaseName(databaseName);

        XAConnection xaCon = ds.getXAConnection();
        try (Connection con = xaCon.getConnection()) {

            // Get a unique Xid object for testing.
            XAResource xaRes = null;
            Xid xid = null;
            xid = XidImpl.getUniqueXid(1);

            // Get the XAResource object and set the timeout value.
            xaRes = xaCon.getXAResource();
            xaRes.setTransactionTimeout(0);

            // Perform the XA transaction.
            System.out.println("Write -> xid = " + xid.toString());
            xaRes.start(xid, XAResource.TMNOFLAGS);
            PreparedStatement pstmt = con.prepareStatement("INSERT INTO XAMin (f1,f2) VALUES (?, ?)");
            pstmt.setInt(1, 1);
            pstmt.setString(2, xid.toString());
            pstmt.executeUpdate();

            // Commit the transaction.
            xaRes.end(xid, XAResource.TMSUCCESS);
            xaRes.commit(xid, true);
        }
        xaCon.close();

        // Open a new connection and read back the record to verify that it worked.
        try (Connection con = DriverManager.getConnection(connectionUrl); Statement stmt =
con.createStatement();
                ResultSet rs = stmt.executeQuery("SELECT * FROM XAMin")) {
            rs.next();
            System.out.println("Read -> xid = " + rs.getString(2));
            stmt.executeUpdate("DROP TABLE XAMin");
        }
    }
}
```

```
class XidImpl implements Xid {

    public int formatId;
    public byte[] gtrid;
    public byte[] bqual;

    public byte[] getGlobalTransactionId() {
        return gtrid;
    }

    public byte[] getBranchQualifier() {
        return bqual;
    }

    public int getFormatId() {
        return formatId;
    }

    XidImpl(int formatId, byte[] gtrid, byte[] bqual) {
        this.formatId = formatId;
        this.gtrid = gtrid;
        this.bqual = bqual;
    }

    public String toString() {
        int hexVal;
        StringBuffer sb = new StringBuffer(512);
        sb.append("formatId=" + formatId);
        sb.append(" gtrid(" + gtrid.length + ")={0x");
        for (int i = 0; i < gtrid.length; i++) {
            hexVal = gtrid[i] & 0xFF;
            if (hexVal < 0x10)
                sb.append("0" + Integer.toHexString(gtrid[i] & 0xFF));
            else
                sb.append(Integer.toHexString(gtrid[i] & 0xFF));
        }
        sb.append("} bqual(" + bqual.length + ")={0x");
        for (int i = 0; i < bqual.length; i++) {
            hexVal = bqual[i] & 0xFF;
            if (hexVal < 0x10)
                sb.append("0" + Integer.toHexString(bqual[i] & 0xFF));
            else
                sb.append(Integer.toHexString(bqual[i] & 0xFF));
        }
        sb.append("}");
        return sb.toString();
    }

    // Returns a globally unique transaction id.
    static byte[] localIP = null;
    static int txnUniqueID = 0;

    static Xid getUniqueXid(int tid) {

        Random rnd = new Random(System.currentTimeMillis());
        txnUniqueID++;
        int txnUID = txnUniqueID;
        int tidID = tid;
        int randID = rnd.nextInt();
        byte[] gtrid = new byte[64];
        byte[] bqual = new byte[64];
        if (null == localIP) {
            try {
                localIP = Inet4Address.getLocalHost().getAddress();
            } catch (Exception ex) {
                localIP = new byte[] {0x01, 0x02, 0x03, 0x04};
            }
        }
        System.arraycopy(localIP, 0, gtrid, 0, 4);
        System.arraycopy(localIP, 0, bqual, 0, 4);

        // Bytes 4 -> 7 - unique transaction id.
        // Bytes 8 ->11 - thread id.
        // Bytes 12->15 - random number generated by using seed from current time in milliseconds.
        for (int i = 0; i <= 3; i++) {
            gtrid[i + 4] = (byte) (txnUID % 0x100);
            bqual[i + 4] = (byte) (txnUID % 0x100);
            txnUID >>= 8;
            gtrid[i + 8] = (byte) (tidID % 0x100);
            bqual[i + 8] = (byte) (tidID % 0x100);
            tidID >>= 8;
            gtrid[i + 12] = (byte) (randID % 0x100);
            bqual[i + 12] = (byte) (randID % 0x100);
            randID >>= 8;
        }
        return new XidImpl(0x1234, gtrid, bqual);
    }
}
```

See Also

Performing Transactions with the JDBC Driver

Using Auto Generated Keys

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server supports the optional JDBC 3.0 APIs to retrieve automatically generated row identifiers. The main value of this feature is to provide a way to make IDENTITY values available to an application that is updating a database table without a requiring a query and a second round-trip to the server.

Because SQL Server doesn't support pseudo columns for identifiers, updates that have to use the auto-generated key feature must operate against a table that contains an IDENTITY column. SQL Server allows only a single IDENTITY column per table. The result set that is returned by getGeneratedKeys method of the SQLServerStatement class will have only one column, with the returned column name of GENERATED_KEYS. If generated keys are requested on a table that has no IDENTITY column, the JDBC driver will return a null result set.

As an example, create the following table in the AdventureWorks sample database:

```
CREATE TABLE TestTable
   (Col1 int IDENTITY,
    Col2 varchar(50),
    Col3 int);
```

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, an SQL statement is constructed that will add data to the table, and then the statement is run and the IDENTITY column value is displayed.

```
public static void executeInsertWithKeys(Connection con) {
    try(Statement stmt = con.createStatement();) {
        String SQL = "INSERT INTO TestTable (Col2, Col3) VALUES ('S', 50)";
        int count = stmt.executeUpdate(SQL, Statement.RETURN_GENERATED_KEYS);
        ResultSet rs = stmt.getGeneratedKeys();

        ResultSetMetaData rsmd = rs.getMetaData();
        int columnCount = rsmd.getColumnCount();
        if (rs.next()) {
            do {
                for (int i=1; i<=columnCount; i++) {
                    String key = rs.getString(i);
                    System.out.println("KEY " + i + " = " + key);
                }
            } while(rs.next());
        }
        else {
            System.out.println("NO KEYS WERE GENERATED.");
        }
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

See Also

Using Statements with the JDBC Driver

Using Database Mirroring (JDBC)

8/13/2018 • 5 minutes to read • Edit Online

Download JDBC Driver

Database mirroring is primarily a software solution for increasing database availability and data redundancy. The Microsoft JDBC Driver for SQL Server provides implicit support for database mirroring, so that the developer doesn't need to write any code or take any other action when it has been configured for the database.

Database mirroring, which is implemented on a per-database basis, keeps a copy of a SQL Server production database on a standby server. This server is either a hot or warm standby server, depending on the configuration and state of the database mirroring session. A hot standby server supports rapid failover without a loss of committed transactions, and a warm standby server supports forcing service (with possible data loss).

The production database is called the *principal* database, and the standby copy is called the *mirror* database. The principal database and mirror database must reside on separate instances of SQL Server (server instances), and they should reside on separate computers, if it's possible.

The production server instance, referred to as the principal server, communicates with the standby server instance, referred to as the mirror server. The principal and mirror servers act as partners within a database mirroring session. If the principal server fails, the mirror server can make its database into the principal database through a process called *failover*. For example, Partner_A and Partner_B are two partner servers, with the principal database initially on Partner_A as principal server, and the mirror database residing on Partner_B as the mirror server. If Partner_A goes offline, the database on Partner_B can fail over to become the current principal database. When Partner_A rejoins the mirroring session, it becomes the mirror server and its database becomes the mirror database.

In the case where the Partner_A server is irreparably damaged, a Partner_C server can be brought online to act as the mirror server for Partner_B, which is now the principal server. However, in this scenario, the client application must include programming logic to ensure that the connection string properties are updated with the new server names used in the database mirroring configuration. Otherwise, the connection to the servers may fail.

Alternative database mirroring configurations offer different levels of performance and data safety, and support different forms of failover. For more information, see "Overview of Database Mirroring" in SQL Server Books Online.

Programming Considerations

When the principal database server fails, the client application receives errors in response to API calls, which indicate that the connection to the database has been lost. When this occurs, any uncommitted changes to the database are lost and the current transaction is rolled back. If this occurs, the application should close the connection (or release the data source object) and try to reopen it. On connection, the new connection is transparently redirected to the mirror database, which now acts as the principal server, without the client having to modify the connection string or data source object.

When a connection is initially established, the principal server sends the identity of its failover partner to the client that will be used when failover occurs. When an application tries to establish an initial connection with a failed principal server, the client doesn't know the identity of the failover partner. To allow clients the opportunity to cope with this scenario, the failoverPartner connection string property, and optionally the setFailoverPartner data source method, allows the client to specify the identity of the failover partner on its own. The client property is used only in this scenario; if the principal server is available, it isn't used.

NOTE

When a failoverPartner is specified in either the connection string or with a data source object, the databaseName property must also be set or else an exception will be thrown. If the failoverPartner and databaseName are not specified explicitly, the application will not attempt to failover when the principal database server fails. In other words, the transparent redirection only works for connections that explicitly specify the failoverPartner and databaseName. For more information about failoverPartner and other connection string properties, see Setting the Connection Properties.

If the failover partner server supplied by the client doesn't refer to a server acting as a failover partner for the specified database, and if the server/database referred to is in a mirroring arrangement, the connection is refused by the server. Although the SQLServerDataSource class provides the getFailoverPartner method, this method only returns the name of the failover partner specified in the connection string or the setFailoverPartner method. To retrieve the name of the actual failover partner that is currently being used, use the following Transact-SQL statement:

```
SELECT m.mirroring_role_DESC, m.mirroring_state_DESC,
m.mirroring_partner_instance FROM sys.databases as db,
sys.database_mirroring AS m WHERE db.name = 'MirroringDBName'
AND db.database_id = m.database_id
```

NOTE

You will need to change this statement to use the name of your mirroring database.

You should consider caching the partner information to update the connection string or devise a retry strategy in case the first attempt at making a connection fails.

Example

In the following example, an attempt is first made to connect to the principle server. If that fails and an exception is thrown, an attempt is made to connect to the mirror server, which may have been promoted to the new principle server. Note the use of the failoverPartner property in the connection string.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class ClientFailover {
    public static void main(String[] args) {

        String connectionUrl = "jdbc:sqlserver://serverA:1433;"
                + "databaseName=AdventureWorks;integratedSecurity=true;"
                + "failoverPartner=serverB";

        // Establish the connection to the principal server.
        try (Connection con = DriverManager.getConnection(connectionUrl);
                Statement stmt = con.createStatement();) {
            System.out.println("Connected to the principal server.");

            // Note that if a failover of serverA occurs here, then an
            // exception will be thrown and the failover partner will
            // be used in the first catch block below.

            // Execute a SQL statement that inserts some data.

            // Note that the following statement assumes that the
            // TestTable table has been created in the AdventureWorks
            // sample database.
            stmt.executeUpdate("INSERT INTO TestTable (Col2, Col3) VALUES ('a', 10)");
        }
        catch (SQLException se) {
            System.out.println("Connection to principal server failed, " + "trying the mirror server.");
            // The connection to the principal server failed,
            // try the mirror server which may now be the new
            // principal server.
            try (Connection con = DriverManager.getConnection(connectionUrl);
                    Statement stmt = con.createStatement();) {
                System.out.println("Connected to the new principal server.");
                stmt.executeUpdate("INSERT INTO TestTable (Col2, Col3) VALUES ('a', 10)");
            }
            // Handle any errors that may have occurred.
            catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

See Also

Connecting to SQL Server with the JDBC Driver

National Character Set Support

5/3/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The JDBC driver provides support for the JDBC 4.0 API, which includes new national character set conversion API methods. This support includes new setter, getter, and updater methods for **NCHAR**, **NVARCHAR**, **LONGNVARCHAR**, and **NCLOB** JDBC types.

The following is a list of new getter, setter, and updater methods to support the national character set conversion:

- SQLServerPreparedStatement: setNString, setNCharacterStream, setNClob.
- SQLServerCallableStatement: getNClob, getNString, getNCharacterStream, setNString, setNCharacterStream, setNClob.
- SQLServerResultSet: getNClob, getNString, getNCharacterStream, updateNClob, updateNString, updateNCharacterStream.

NOTE

You must set the classpath to include the sqljdbc4.jar file to use these methods in your application.

In order to send String parameters to the server in Unicode format, the applications should either use the new JDBC 4.0 national character methods; or set the **sendStringParametersAsUnicode** connection property to "**true**" when using the non-national character methods. The recommended way is to use the new JDBC 4.0 national character methods where possible. For more information about the **sendStringParametersAsUnicode** connection property, see Setting the Connection Properties.

See Also

Understanding the JDBC Driver Data Types

Understanding Cursor Types

8/13/2018 • 11 minutes to read • Edit Online

Download JDBC Driver

Operations in a relational database act on a complete set of rows. The set of rows returned by a SELECT statement consists of all the rows that satisfy the conditions in the WHERE clause of the statement. This complete set of rows returned by the statement is known as the result set. Applications cannot always work effectively with the entire result set as a unit. These applications need a mechanism to work with one row or a small block of rows at a time. Cursors are an extension to result sets that provide that mechanism.

Cursors extend result set processing by doing the following:

- Allowing positioning at specific rows of the result set.
- Retrieving one row or block of rows from the current position in the result set.
- Supporting data modifications to the row at the current position in the result set.
- Supporting different levels of visibility to changes made by other users to the database data that is presented in the result set.

NOTE

For a full description of the SQL Server cursor types, see the "Cursor Types (Database Engine)" topic in SQL Server Books Online.

The JDBC specification provides support for forward-only and scrollable cursors that are sensitive or insensitive to changes made by other jobs, and can be read-only or updatable. This functionality is provided by the Microsoft JDBC Driver for SQL ServerSQLServerResultSet class.

Remarks

The JDBC driver supports the following cursor types:

RESULT SET (CURSOR) TYPE	SQL SERVER CURSOR TYPE	CHARACTERISTICS	SELECT METHOD	RESPONSE BUFFERING	DESCRIPTION

RESULT SET (CURSOR) TYPE	SQL SERVER CURSOR TYPE	CHARACTERISTICS	SELECT METHOD	RESPONSE BUFFERING	DESCRIPTION
TYPE_FORWARD_ONLY (CONCUR_READ_ONLY)	N/A	Forward-only, read-only	direct	full	The application has to make a single (forward) pass through the result set. This is the default behavior and behaves the same as a TYPE_SS_DIRECT_FORWARD_ONLY cursor. The driver reads the entire result set from the server into a memory during the statement execution time.
TYPE_FORWARD_ONLY (CONCUR_READ_ONLY)	N/A	Forward-only, read-only	direct	adaptive	The application has to make a single (forward) pass through the result set. It behaves the same as a TYPE_SS_DIRECT_FORWARD_ONLY cursor. The driver reads rows from the server as the application requests them and thus minimizes the client-side memory usage.
TYPE_FORWARD_ONLY (CONCUR_READ_ONLY)	Fast Forward	Forward-only, read-only	cursor	N/A	The application has to make a single (forward) pass through the result set by using a server cursor. It behaves the same as a TYPE_SS_SERVER_CURSOR_FORWARD_ONLY cursor. Rows are retrieved from the server in blocks that are specified by the fetch size.
TYPE_FORWARD_ONLY (CONCUR_UPDA	Dynamic (Forward-only)	Forward-only, updatable	N/A	N/A	The application has to make a single (forward)

RESULT SET (CURSOR) TYPE	SQL SERVER CURSOR TYPE	CHARACTERISTICS	SELECT METHOD	RESPONSE BUFFERING	DESCRIPTION
(CONCUR_UPDATABLE)					single (forward) pass through the result set to update one or more rows. Rows are retrieved from the server in blocks that are specified by the fetch size. By default, the fetch size is fixed when the application calls the setFetchSize method of the SQLServerResultSet object. **Note:** The JDBC driver provides an adaptive buffering feature that allows you to retrieve statement execution results from the SQL Server as the application needs them, rather than all at once. For example, if the application should retrieve a large data that is too large to fit entirely in application memory, adaptive buffering allows the client application to retrieve such a value as a stream. The default behavior of the driver is "**adaptive**". However, in order to get the adaptive buffering for the forward-only updatable result sets, the application has to explicitly call the setResponseBuffering method of the SQLServerStatem

RESULT SET (CURSOR) TYPE	SQL SERVER CURSOR TYPE	CHARACTERISTICS	SELECT METHOD	RESPONSE BUFFERING	DESCRIPTION
					ent object by providing a **String** value **"adaptive"**. For an example code, see Updating Large Data Sample.
TYPE_SCROLL_INSENSITIVE	Static	Scrollable, not updateable. External row updates, inserts, and deletes are not visible.	N/A	N/A	The application requires a database snapshot. The result set is not updatable. Only CONCUR_READ_ONLY is supported. All other concurrency types will cause an exception when used with this cursor type. Rows are retrieved from the server in blocks that are specified by the fetch size.
TYPE_SCROLL_SENSITIVE (CONCUR_READ_ONLY)	Keyset	Scrollable, read-only. External row updates are visible, and deletes appear as missing data. External row inserts are not visible.	N/A	N/A	The application has to see changed data for existing rows only. Rows are retrieved from the server in blocks that are specified by the fetch size.

RESULT SET (CURSOR) TYPE	SQL SERVER CURSOR TYPE	CHARACTERISTICS	SELECT METHOD	RESPONSE BUFFERING	DESCRIPTION
TYPE_SCROLL_SENSITIVE (CONCUR_UPDATABLE, CONCUR_SS_SCROLL_LOCKS, CONCUR_SS_OPTIMISTIC_CC, CONCUR_SS_OPTIMISTIC_CCVAL)	Keyset	Scrollable, updatable. External and internal row updates are visible, and deletes appear as missing data; inserts are not visible.	N/A	N/A	The application may change data in the existing rows by using the ResultSet object. The application must also be able to see the changes to rows made by others from outside the ResultSet object. Rows are retrieved from the server in blocks that are specified by the fetch size.
TYPE_SS_DIRECT_FORWARD_ONLY	N/A	Forward-only, read-only	N/A	full or adaptive	Integer value = 2003. Provides a read-only client side cursor that is fully buffered. No server cursor is created. Only CONCUR_READ_ONLY concurrency type is supported. All other concurrency types cause an exception when used with this cursor type.

RESULT SET (CURSOR) TYPE	SQL SERVER CURSOR TYPE	CHARACTERISTICS	SELECT METHOD	RESPONSE BUFFERING	DESCRIPTION
TYPE_SS_SERVER_CURSOR_FORWARD_ONLY	Fast Forward	Forward-only	N/A	N/A	Integer value = 2004. Fast, accesses all data using a server cursor. It is updatable when used with CONCUR_UPDATABLE concurrency type. Rows are retrieved from the server in blocks that are specified by the fetch size. In order to get the adaptive buffering for this case, the application has to explicitly call the setResponseBuffering method of the SQLServerStatement object by providing a **String** value "**adaptive**". For an example code, see Updating Large Data Sample.
TYPE_SS_SCROLL_STATIC	Static	Other users' updates are not reflected.	N/A	N/A	Integer value = 1004. Application requires a database snapshot. This is the SQL Server-specific synonym for the JDBC TYPE_SCROLL_INSENSITIVE and has the same concurrency setting behavior. Rows are retrieved from the server in blocks that are specified by the fetch size.

RESULT SET (CURSOR) TYPE	SQL SERVER CURSOR TYPE	CHARACTERISTICS	SELECT METHOD	RESPONSE BUFFERING	DESCRIPTION
TYPE_SS_SCROLL_KEYSET (CONCUR_READ_ONLY)	Keyset	Scrollable, read-only. External row updates are visible, and deletes appear as missing data. External row inserts are not visible.	N/A	N/A	Integer value = 1005. Application has to see changed data for existing rows only. This is the SQL Server-specific synonym for the JDBC TYPE_SCROLL_SENSITIVE and has the same concurrency setting behavior. Rows are retrieved from the server in blocks that are specified by the fetch size.
TYPE_SS_SCROLL_KEYSET (CONCUR_UPDATABLE, CONCUR_SS_SCROLL_LOCKS, CONCUR_SS_OPTIMISTIC_CC, CONCUR_SS_OPTIMISTIC_CCVAL)	Keyset	Scrollable, updatable. External and internal row updates are visible, and deletes appear as missing data; inserts are not visible.	N/A	N/A	Integer value = 1005. Application has to change data or see changed data for existing rows. This is the SQL Server-specific synonym for the JDBC TYPE_SCROLL_SENSITIVE and has the same concurrency setting behavior. Rows are retrieved from the server in blocks that are specified by the fetch size.

RESULT SET (CURSOR) TYPE	SQL SERVER CURSOR TYPE	CHARACTERISTICS	SELECT METHOD	RESPONSE BUFFERING	DESCRIPTION
TYPE_SS_SCROLL_DYNAMIC (CONCUR_READ_ONLY)	Dynamic	Scrollable, read-only. External row updates and inserts are visible, and deletes appear as transient missing data in the current fetch buffer.	N/A	N/A	Integer value = 1006. Application must see changed data for existing rows, and see inserted and deleted rows during the lifetime of the cursor. Rows are retrieved from the server in blocks that are specified by the fetch size.
TYPE_SS_SCROLL_DYNAMIC (CONCUR_UPDATABLE, CONCUR_SS_SCROLL_LOCKS, CONCUR_SS_OPTIMISTIC_CC, CONCUR_SS_OPTIMISTIC_CCVAL)	Dynamic	Scrollable, updatable. External and internal row updates and inserts are visible, and deletes appear as transient missing data in the current fetch buffer.	N/A	N/A	Integer value = 1006. The application may change data for existing rows, or insert or delete rows by using the ResultSet object. The application must also be able to see changes, inserts, and deletes made by others from outside the ResultSet object. Rows are retrieved from the server in blocks that are specified by the fetch size.

Cursor Positioning

The TYPE_FORWARD_ONLY, TYPE_SS_DIRECT_FORWARD_ONLY, and TYPE_SS_SERVER_CURSOR_FORWARD_ONLY cursors support only the next positioning method.

The TYPE_SS_SCROLL_DYNAMIC cursor does not support the absolute and getRow methods. The absolute method can be approximated by a combination of calls to the first and relative methods for dynamic cursors.

The getRow method is supported by TYPE_FORWARD_ONLY, TYPE_SS_DIRECT_FORWARD_ONLY, TYPE_SS_SERVER_CURSOR_FORWARD_ONLY, TYPE_SS_SCROLL_KEYSET, and TYPE_SS_SCROLL_STATIC cursors only. The getRow method with all forward-only cursor types returns the number of rows read so far through the cursor.

NOTE

When an application makes an unsupported cursor positioning call, or an unsupported call to the getRow method, an exception is thrown with the message, "The requested operation is not supported with this cursor type."

Only the TYPE_SS_SCROLL_KEYSET and the equivalent TYPE_SCROLL_SENSITIVE cursors expose deleted rows. If the cursor is positioned on a deleted row, column values are unavailable, and the rowDeleted method returns "true". Calls to get<Type> methods throw an exception with the message, "Cannot get value from a deleted row". Deleted rows cannot be updated. If you try to call an update<Type> method on a deleted row, an exception is thrown with the message, "A deleted row cannot be updated". The TYPE_SS_SCROLL_DYNAMIC cursor has the same behavior until the cursor is moved out of the current fetch buffer.

Forward and dynamic cursors expose deleted rows in a similar way, but only while the cursors remain accessible in the fetch buffer. For forward cursors, this is fairly straightforward. For dynamic cursors it more complex when the fetch size is greater than 1. An application can move the cursor forward and backward within the window that is defined by the fetch buffer, but the deleted row will disappear when the original fetch buffer in which it was updated is left. If an application does not want to see transient deleted rows by using dynamic cursors, a fetch relative (0) should be used.

If the key values of a TYPE_SS_SCROLL_KEYSET or TYPE_SCROLL_SENSITIVE cursor row are updated with the cursor, the row retains its original position in the result set, regardless of whether the updated row meets the cursor's selection criteria. If the row was updated outside the cursor, a deleted row will appear at the row's original position, but the row will appear in the cursor only if another row with the new key values was present in the cursor, but has since been deleted.

For dynamic cursors, updated rows will retain their position within the fetch buffer until the window that is defined by the fetch buffer is left. Updated rows might subsequently reappear at different positions within the result set, or might disappear completely. Applications that have to avoid transient inconsistencies in the result set should use a fetch size of 1 (the default is 8 rows with CONCUR_SS_SCROLL_LOCKS concurrency and 128 rows with other concurrencies).

Cursor Conversion

SQL Server can sometimes choose to implement a cursor type other than the one requested, which is referred to as an implicit cursor conversion (or cursor degradation). For more information about implicit cursor conversion, see the "Using Implicit Cursor Conversions" topic in SQL Server Books Online.

With SQL Server 2000 (8.x), when you update the data through the ResultSet.TYPE_SCROLL_SENSITIVE and ResultSet.CONCUR_UPDATABLE result set, an exception is thrown with a message "The cursor is READ ONLY". This exception occurs because the SQL Server 2000 (8.x) has done an implicit cursor conversion for that result set and did not return the updatable cursor that has been requested.

To work around this problem, you can do one of the following two solutions:

- Ensure that the underlying table has a primary key
- Use SQLServerResultSet.TYPE_SS_SCROLL_DYNAMIC instead of ResultSet.TYPE_SCROLL_SENSITIVE while creating a statement.

Cursor Updating

In-place updates are supported for cursors where the cursor type and concurrency support updates. If the cursor is not positioned on an updatable row in the result set (no get<Type> method call succeeded), a call to an update<Type> method will throw an exception with the message, "The result set has no current row." The JDBC specification states that an exception arises when an update method is called for a column of a cursor that is

CONCUR_READ_ONLY. In situations where the row is not updatable, such as because of an optimistic concurrency conflict such as a competing update or deletion, the exception might not arise until insertRow, updateRow, or deleteRow is called.

After a call to update<Type>, the affected column cannot be accessed by get<Type> until updateRow or cancelRowUpdates has been called. This avoids problems where a column is updated by using a different type from the type returned by the server, and subsequent getter calls could invoke client side type conversions that give inaccurate results. Calls to get<Type> will throw an exception with the message, "Updated columns cannot be accessed until updateRow() or cancelRowUpdates() has been called."

NOTE

If the updateRow method is called when no columns have been updated, the JDBC driver will throw an exception with the message, "updateRow() called when no columns have been updated."

After moveToInsertRow has been called, an exception will be thrown if any method other than get<Type>, update<Type>, insertRow, and cursor positioning methods (including moveToCurrentRow) are called on the result set. The moveToInsertRow method effectively places the result set into insert mode, and cursor positioning methods terminate insert mode. Relative cursor positioning calls move the cursor relative to the position it was at before moveToInsertRow was called. After cursor positioning calls, the eventual destination cursor position becomes the new cursor position.

If the cursor positioning call made while in insert mode does not succeed, the cursor position after the failed call is the original cursor position before moveToInsetRow was called. If insertRow fails, the cursor remains on the insert row and the cursor remains in insert mode.

Columns in the insert row are initially in an uninitialized state. Calls to the update<Type> method set the column state to initialized. A call to the get<Type> method for an uninitialized column throws an exception. A call to the insertRow method returns all the columns in the insert row to an uninitialized state.

If any columns are uninitialized when the insertRow method is called, the default value for the column is inserted. If there is no default value but the column is nullable, then NULL is inserted. If there is no default value and the column is not nullable, the server will return an error and an exception will be thrown.

NOTE

Calls to the getRow method returns 0 when in insert mode.

The JDBC driver does not support positioned updates or deletes. According to the JDBC specification, the setCursorName method has no effect and the getCursorName method will throw an exception if called.

Read-only and static cursors are never updatable.

SQL Server restricts server cursors to a single result set. If a batch or stored procedure contains multiple statements, then a forward-only read-only client cursor must be used.

See Also

Managing Result Sets with the JDBC Driver

Programming with SQLXML

5/3/2018 • 5 minutes to read • Edit Online

Download JDBC Driver

This section describes how to use the Microsoft JDBC Driver for SQL Server API methods to store and retrieve an XML document in and from a relational database with **SQLXML** objects.

This section also contains information about the types of SQLXML objects and provides a list of important guidelines and limitations when using SQLXML objects.

Reading and Writing XML Data with SQLXML Objects

The following list describes how to use the Microsoft JDBC Driver for SQL Server API methods to read and write XML data with SQLXML objects:

- To create a SQLXML object, use the createSQLXML method of the SQLServerConnection class. Note that this method creates a SQLXML object without any data. To add **xml** data to SQLXML object, call one of the following methods that are specified in the SQLXML interface: setResult, setCharacterStream, setBinaryStream, or setString.
- To retrieve the SQLXML object itself, use the getSQLXML methods of the SQLServerResultSet class or the SQLServerCallableStatement class.
- To retrieve the **xml** data from a SQLXML object, use one of the following methods that are specified in the SQLXML interface: getSource, getCharacterStream, getBinaryStream, or getString.
- To update the **xml** data in a SQLXML object, use the updateSQLXML method of the SQLServerResultSet class.
- To store a SQLXML object in a database table column of type **xml**, use the setSQLXML methods of the SQLServerPreparedStatement class or the SQLServerCallableStatement class.

  The example code in SQLXML Data Type Sample demonstrates how to perform these common API tasks.

Readable and Writable SQLXML Objects

The following table lists which types of SQLXML objects are supported by the setter, getter, and updater methods provided by the JDBC API. The columns in the table refer to the following:

- The **Method Name** column lists the supported getter, setter, and updater methods in the JDBC API.
- The **Getter SQLXML Object** column represents a SQLXML object, which is created by either the getSQLXML method of the SQLServerCallableStatement class or the getSQLXML method of the SQLServerResultSet class.
- The **Setter SQLXML Object** column represents a SQLXML object, which is created by the createSQLXML method of the SQLServerConnection class. Note that the setter methods below accept only a SQLXML object created by the createSQLXML method.

METHOD NAME	GETTER SQLXML OBJECT (READABLE)	SETTER SQLXML OBJECT (WRITABLE)
CallableStatement.setSQLXML()	Not Supported	Supported
CallableStatement.setObject()	Not Supported	Supported
PreparedStatement.setSQLXML()	Not Supported	Supported
PreparedStatement.setObject()	Not Supported	Supported
ResultSet.updateSQLXML()	Not Supported	Supported
ResultSet.updateObject()	Not Supported	Supported
ResultSet.getSQLXML()	Supported	Not Supported
CallableStatement.getSQLXML()	Supported	Not Supported

As shown in the table above, the setter SQLXML methods will not work with the readable SQLXML objects; similarly, the getter methods will not work with the writable SQLXML objects.

If the application invokes the setObject method by specifying a scale or a length parameter with a SQLXML object, the scale or length parameter is ignored.

Guidelines and Limitations when Using SQLXML Objects

Applications can use SQLXML objects to read and write the XML data from and to the database. The following list provides information about specific limitations and guidance when using SQLXML objects:

- A SQLXML object can be valid only for the duration of the transaction in which it was created.
- A SQLXML object received from a getter method can only be used to read data.
- A SQLXML object created by the connection object can only be used to write data.
- The application can invoke only one getter method on a readable SQLXML object to read data. After the getter method is invoked, all other getter or setter methods on the same SQLXML object fail.
- The application can invoke only the free method on the SQLXML object after it is read or written to. However, it is still possible to process the returned stream or source as long as the underlying column or parameter is active. If the underlying column or parameter becomes inactive, the stream or source associated with the SQLXML object will be closed. If the underlying column or parameter is no longer valid, the underlying data will not be available for the Stream, Simple API for XML (SAX), and Streaming API for XML (StAX) getters.
- The application can invoke only one setter method on a writable SQLXML object. After the setter method is invoked, all other setter or getter methods on the same SQLXML object fail.
- To set data on the SQLXML object, the application must use the appropriate setter method and the functions in the returned object.
- The getSQLXML methods of the SQLServerCallableStatement class and the SQLServerResultSet class returns **null** data if the underlying column is **null**.
- The setter objects can be valid through the connection they are created within.

- Applications are not allowed to set a **null** value by using the setter methods provided by the SQLXML interface. The applications can set an empty string ("") by using the setter methods provided in the SQLXML interface. To set a **null** value, the applications should call one of the following:
    - The setNull methods of the SQLServerCallableStatement class and SQLServerPreparedStatement class.
    - The setObject methods of the SQLServerCallableStatement class and SQLServerPreparedStatement class.
    - The setSQLXML methods of the SQLServerCallableStatement class and SQLServerPreparedStatement class with a **null** parameter value.
- When working with XML documents, we recommend using Simple API for XML (SAX) and Streaming API for XML (StAX) parsers instead of Document Object Model (DOM) parsers for performance reasons.

  XML parsers cannot handle empty values. However, SQL Server allows applications to retrieve and store empty values from and to database columns of the XML data type. That means that when parsing the XML data, if the underlying value is empty, an exception is thrown by the parser. For DOM outputs, the JDBC driver catches that exception and throws an error. For SAX and Stax outputs, the error comes from the parser directly.

Adaptive Buffering and SQLXML Support

The binary and character streams returned by the SQLXML object obey the adaptive or full buffering modes. On the other hand, if the XML parsers are not streams, they will not obey the adaptive or full settings. For more information about adaptive buffering, see Using Adaptive Buffering.

See Also

Supporting XML Data

Using a Stored Procedure with a Return Status

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

A SQL Server stored procedure that you can call is one that returns a status or a result parameter. This is typically used to indicate the success or failure of the stored procedure. The Microsoft JDBC Driver for SQL Server provides the SQLServerCallableStatement class, which you can use to call this kind of stored procedure and to process the data that it returns.

When you call this kind of stored procedure by using the JDBC driver, you have to use the `call` SQL escape sequence in conjunction with the prepareCall method of the SQLServerConnection class. The syntax for the `call` escape sequence with a return status parameter is the following:

```
{[?=]call procedure-name[([parameter][,[parameter]]...)]}
```

NOTE

For more information about the SQL escape sequences, see Using SQL Escape Sequences.

When you construct the `call` escape sequence, specify the return status parameter by using the ? (question mark) character. This character acts as a placeholder for the parameter value that will be returned from the stored procedure. To specify a value for a return status parameter, you must specify the data type of the parameter by using the registerOutParameter method of the SQLServerCallableStatement class, before executing the stored procedure.

NOTE

When using the JDBC driver with a SQL Server database, the value that you specify for the return status parameter in the registerOutParameter method will always be an integer, which you can specify by using the java.sql.Types.INTEGER data type.

In addition, when you pass a value to the registerOutParameter method for a return status parameter, you must specify not only the data type to be used for the parameter, but also the parameter's ordinal placement in the stored procedure call. In the case of the return status parameter, its ordinal position will always be 1 because it is always the first parameter in the call to the stored procedure. Although the SQLServerCallableStatement class provides support for using the parameter's name to indicate the specific parameter, you can use only a parameter's ordinal position number for return status parameters.

As an example, create the following stored procedure in the AdventureWorks sample database:

```
CREATE PROCEDURE CheckContactCity
   (@cityName CHAR(50))
AS
BEGIN
   IF ((SELECT COUNT(*)
   FROM Person.Address
   WHERE City = @cityName) > 1)
   RETURN 1
ELSE
   RETURN 0
END
```

This stored procedure returns a status value of 1 or 0, depending on whether the city that is specified in the cityName parameter is found in the Person.Address table.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, and the execute method is used to call the CheckContactCity stored procedure:

```
public static void executeStoredProcedure(Connection con) {
    try(CallableStatement cstmt = con.prepareCall("{? = call dbo.CheckContactCity(?)}");) {
        cstmt.registerOutParameter(1, java.sql.Types.INTEGER);
        cstmt.setString(2, "Atlanta");
        cstmt.execute();
        System.out.println("RETURN STATUS: " + cstmt.getInt(1));
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

See Also

Using Statements with Stored Procedures

Using an SQL Statement with No Parameters

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

To work with data in a SQL Server database by using an SQL statement that contains no parameters, you can use the executeQuery method of the SQLServerStatement class to return a SQLServerResultSet that will contain the requested data. To do this, you must first create a SQLServerStatement object by using the createStatement method of the SQLServerConnection class.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, an SQL statement is constructed and run, and then the results are read from the result set.

```
public static void executeStatement(Connection con) {
    try(Statement stmt = con.createStatement();) {
        String SQL = "SELECT LastName, FirstName FROM Person.Contact ORDER BY LastName";
        ResultSet rs = stmt.executeQuery(SQL);

        while (rs.next()) {
            System.out.println(rs.getString("LastName") + ", " + rs.getString("FirstName"));
        }
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

For more information about using result sets, see Managing Result Sets with the JDBC Driver.

See Also

Using Statements with SQL

Connecting to an Azure SQL database

8/2/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

This article discusses issues when using the Microsoft JDBC Driver for SQL Server to connect to a Azure SQL Database. For more information about connecting to a Azure SQL Database, see:

- SQL Azure Database
- How to: Connect to SQL Azure Using JDBC
- Connecting using Azure Active Directory Authentication

Details

When connecting to a Azure SQL Database, you should connect to the master database to call **SQLServerDatabaseMetaData.getCatalogs**.
Azure SQL Database doesn't support returning the entire set of catalogs from a user database. **SQLServerDatabaseMetaData.getCatalogs** use the sys.databases view to get the catalogs. Please refer to the discussion of permissions in sys.databases (SQL Azure Database) to understand **SQLServerDatabaseMetaData.getCatalogs** behavior on a Azure SQL Database.

Connections Dropped

When connecting to a Azure SQL Database, idle connections may be terminated by a network component (such as a firewall) after a period of inactivity. There are two types of idle connections, in this context:

- Idle at the TCP layer, where connections can be dropped by any number of network devices.
- Idle by the SQL Azure Gateway, where TCP **keepalive** messages might be occurring (making the connection not idle from a TCP perspective), but not had an active query in 30 minutes. In this scenario, the Gateway will determine that the TDS connection is idle at 30 minutes and terminate the connection.

To avoid dropping idle connections by a network component, the following registry settings (or their non-Windows equivalents) should be set on the operating system where the driver is loaded:

REGISTRY SETTING	RECOMMENDED VALUE
HKEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \ Services \ Tcpip \ Parameters \ KeepAliveTime	30000
HKEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \ Services \ Tcpip \ Parameters \ KeepAliveInterval	1000
HKEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \ Services \ Tcpip \ Parameters \ TcpMaxDataRetransmissions	10

Restart the computer for the registry settings to take effect.

To accomplish this when running in Windows Azure create a startup task to add the registry keys. For example, add the following Startup task to the service definition file:

```
<Startup>
    <Task commandLine="AddKeepAlive.cmd" executionContext="elevated" taskType="simple">
    </Task>
</Startup>
```

Then add a AddKeepAlive.cmd file to your project. Set the "Copy to Output Directory" setting to Copy always. The following is a sample AddKeepAlive.cmd file:

```
if exist keepalive.txt goto done
time /t > keepalive.txt
REM Workaround for JDBC keep alive on SQL Azure
REG ADD HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters /v KeepAliveTime /t REG_DWORD /d
30000 >> keepalive.txt
REG ADD HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters /v KeepAliveInterval /t
REG_DWORD /d 1000 >> keepalive.txt
REG ADD HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters /v TcpMaxDataRetransmissions /t
REG_DWORD /d 10 >> keepalive.txt
shutdown /r /t 1
:done
```

Appending the Server Name to the UserId in the Connection String

Prior to the 4.0 version of the Microsoft JDBC Driver for SQL Server, when connecting to an Azure SQL Database, you were required to append the server name to the UserId in the connection string. For example, user@servername. Beginning in version 4.0 of the Microsoft JDBC Driver for SQL Server, it's no longer necessary to append @servername to the UserId in the connection string.

Using Encryption Requires Setting hostNameInCertificate

When connecting to an Azure SQL Database, you should specify **hostNameInCertificate** if you specify **encrypt=true**. (If the server name in the connection string is *shortName.domainName*, set the **hostNameInCertificate** property to **.domainName*.)

For example:

```
jdbc:sqlserver://abcd.int.mscds.com;databaseName=
myDatabase;user=myName;password=myPassword;encrypt=true;hostNameInCertificate= *.int.mscds.com;
```

See Also

Connecting to SQL Server with the JDBC Driver

Using an SQL Statement to Modify Data

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

To modify the data that is contained in a SQL Server database by using an SQL statement, you can use the executeUpdate method of the SQLServerStatement class. The executeUpdate method will pass the SQL statement to the database for processing, and then return a value that indicates the number of rows that were affected.

To do this, you must first create a SQLServerStatement object by using the createStatement method of the SQLServerConnection class.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, an SQL statement is constructed that adds new data to the table, and then the statement is run and the return value is displayed.

```
public static void executeUpdateStatement(Connection con) {
    try(Statement stmt = con.createStatement();) {
        String SQL = "INSERT INTO TestTable (Col2, Col3) VALUES ('a', 10)";
        int count = stmt.executeUpdate(SQL);
        System.out.println("ROWS AFFECTED: " + count);
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

NOTE

If you must use an SQL statement that contains parameters to modify the data in a SQL Server database, you should use the executeUpdate method of the SQLServerPreparedStatement class.

If the column that you are trying to insert data into contains special characters such as spaces, you must provide the values to be inserted, even if they are default values. If you do not, the insert operation will fail.

If you want the JDBC driver to return all update counts, including update counts returned by any triggers that may have fired, set the lastUpdateCount connection string property to "false". For more information about the lastUpdateCount property, see Setting the Connection Properties.

See Also

Using Statements with SQL

Building the Connection URL

8/13/2018 • 4 minutes to read • Edit Online

Download JDBC Driver

The general form of the connection URL is

```
jdbc:sqlserver://[serverName[\instanceName][:portNumber]][;property=value[;property=value]]
```

where:

- **jdbc:sqlserver://** (Required) is known as the subprotocol and is constant.
- **serverName** (Optional) is the address of the server to connect to. This could be a DNS or IP address, or it could be localhost or 127.0.0.1 for the local computer. If not specified in the connection URL, the server name must be specified in the properties collection.
- **instanceName** (Optional) is the instance to connect to on serverName. If not specified, a connection to the default instance is made.
- **portNumber** (Optional) is the port to connect to on serverName. The default is 1433. If you're using the default, you don't have to specify the port, nor its preceding ':', in the URL.

  NOTE

  For optimal connection performance, you should set the portNumber when you connect to a named instance. This will avoid a round trip to the server to determine the port number. If both a portNumber and instanceName are used, the portNumber will take precedence and the instanceName will be ignored.

- **property** (Optional) is one or more option connection properties. For more information, see Setting the Connection Properties. Any property from the list can be specified. Properties can only be delimited by using the semicolon (';'), and they can't be duplicated.

Caution

For security purposes, you should avoid building the connection URLs based on user input. You should only specify the server name and driver in the URL. For user name and password values, use the connection property collections. For more information about security in your JDBC applications, see Securing JDBC Driver Applications.

Connection Examples

Connect to the default database on the local computer by using a user name and password:

```
jdbc:sqlserver://localhost;user=MyUserName;password=*****;
```

NOTE

Although the previous example uses a username and password in the connection string, you should use integrated security as it is more secure. For more information, see the Connecting with Integrated Authentication section later in this topic.

The following connection string shows an example of how to connect to a SQL Server database using integrated authentication and Kerberos from an application running on any operating system supported by the Microsoft

JDBC Driver for SQL Server:

```
jdbc:sqlserver://;servername=server_name;integratedSecurity=true;authenticationScheme=JavaKerberos
```

Connect to the default database on the local computer by using integrated authentication:

`jdbc:sqlserver://localhost;integratedSecurity=true;`

Connect to a named database on a remote server:

`jdbc:sqlserver://localhost;databaseName=AdventureWorks;integratedSecurity=true;`

Connect on the default port to the remote server:

`jdbc:sqlserver://localhost:1433;databaseName=AdventureWorks;integratedSecurity=true;`

Connect by specifying a customized application name:

`jdbc:sqlserver://localhost;databaseName=AdventureWorks;integratedSecurity=true;applicationName=MyApp;`

Named and Multiple SQL Server Instances

SQL Server allows for the installation of multiple database instances per server. Each instance is identified by a specific name. To connect to a named instance of SQL Server, you can either specify the port number of the named instance (preferred), or you can specify the instance name as a JDBC URL property or a **datasource** property. If no instance name or port number property is specified, a connection to the default instance is created. See the following examples:

To use a port number, do the following:

`jdbc:sqlserver://localhost:1433;integratedSecurity=true;<more properties as required>;`

To use a JDBC URL property, do the following:

`jdbc:sqlserver://localhost;instanceName=instance1;integratedSecurity=true;<more properties as required>;`

Escaping Values in the Connection URL

You might have to escape certain parts of the connection URL values because of the inclusion of special characters such as spaces, semicolons, and quotation marks. The JDBC driver supports escaping these characters if they are enclosed in braces. For example, {;} escapes a semicolon.

Escaped values can contain special characters (especially '=', ';', '[]', and space) but cannot contain braces. Values that must be escaped and contain braces should be added to a properties collection.

NOTE

White space inside the braces is literal and not trimmed.

Connecting with Integrated Authentication On Windows

The JDBC driver supports the use of Type 2 integrated authentication on Windows operating systems through the integratedSecurity connection string property. To use integrated authentication, copy the sqljdbc_auth.dll file to a directory on the Windows system path on the computer where the JDBC driver is installed.

The sqljdbc_auth.dll files are installed in the following location:

<installation directory>\sqljdbc_*<version>**<language>*\auth\

For any operating system supported by the Microsoft JDBC Driver for SQL Server, see Using Kerberos Integrated Authentication to Connect to SQL Server for a description of a feature added in Microsoft JDBC Driver 4.0 for SQL Server that allows an application to connect to a database using integrated authentication with Type 4 Kerberos.

NOTE

If you are running a 32-bit Java Virtual Machine (JVM), use the sqljdbc_auth.dll file in the x86 folder, even if the operating system is the x64 version. If you are running a 64-bit JVM on a x64 processor, use the sqljdbc_auth.dll file in the x64 folder.

Alternatively you can set the java.libary.path system property to specify the directory of the sqljdbc_auth.dll. For example, if the JDBC driver is installed in the default directory, you can specify the location of the DLL by using the following virtual machine (VM) argument when the Java application is started:

```
-Djava.library.path=C:\Microsoft JDBC Driver 6.4 for SQL Server\sqljdbc_<version>\enu\auth\x86
```

Connecting with IPv6 Addresses

The JDBC driver supports the use of IPv6 addresses with the connection properties collection, and with the serverName connection string property. The initial serverName value, such as jdbc:*sqlserver*://*serverName*, isn't supported for IPv6 addresses in connection strings. Using a name for *serverName* instead of a raw IPv6 address will work in every case in the connection. The following examples provide more information.

To use the serverName property

```
jdbc:sqlserver://;serverName=3ffe:8311:eeee:f70f:0:5eae:10.203.31.9\\instance1;integratedSecurity=true;
```

To use the properties collection

```
Properties pro = new Properties();
```

```
pro.setProperty("serverName", "serverName=3ffe:8311:eeee:f70f:0:5eae:10.203.31.9\\instance1");
```

```
Connection con = DriverManager.getConnection("jdbc:sqlserver://;integratedSecurity=true;", pro);
```

See Also

Connecting to SQL Server with the JDBC Driver

Using Savepoints

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Savepoints offer a mechanism to roll back portions of transactions. Within SQL Server, you can create a savepoint by using the SAVE TRANSACTION savepoint_name statement. Later, you run a ROLLBACK TRANSACTION savepoint_name statement to roll back to the savepoint instead of rolling back to the start of the transaction.

Savepoints are useful in situations where errors are unlikely to occur. The use of a savepoint to roll back part of a transaction in the case of an infrequent error can be more efficient than having each transaction test to see if an update is valid before making the update. Updates and rollbacks are expensive operations, so savepoints are effective only if the probability of encountering the error is low and the cost of checking the validity of an update beforehand is relatively high.

The Microsoft JDBC Driver for SQL Server supports the use of savepoints through the setSavepoint method of the SQLServerConnection class. By using the setSavepoint method, you can create a named or unnamed savepoint within the current transaction, and the method will return a SQLServerSavepoint object. Multiple savepoints can be created within a transaction. To roll back a transaction to a given savepoint, you can pass the SQLServerSavepoint object to the rollback (java.sql.Savepoint) method.

In the following example, a savepoint is used while performing a local transaction consisting of two separate statements in the `try` block. The statements are run against the Production.ScrapReason table in the AdventureWorks sample database, and a savepoint is used to roll back the second statement. This results in only the first statement being committed to the database.

```
public static void executeTransaction(Connection con) {
    try(Statement stmt = con.createStatement();) {
        con.setAutoCommit(false);
        stmt.executeUpdate("INSERT INTO Production.ScrapReason(Name) VALUES('Correct width')");
        Savepoint save = con.setSavepoint();
        stmt.executeUpdate("INSERT INTO Production.ScrapReason(Name) VALUES('Wrong width')");
        con.rollback(save);
        con.commit();
        System.out.println("Transaction succeeded.");
    }
    catch (SQLException ex) {
        ex.printStackTrace();
        try {
            System.out.println("Transaction failed.");
            con.rollback();
        }
        catch (SQLException se) {
            se.printStackTrace();
        }
    }
}
```

See Also

Performing Transactions with the JDBC Driver

Using Table-Valued Parameters

8/8/2018 • 12 minutes to read • Edit Online

Download JDBC Driver

Table-valued parameters provide an easy way to marshal multiple rows of data from a client application to SQL Server without requiring multiple round trips or special server-side logic for processing the data. You can use table-valued parameters to encapsulate rows of data in a client application and send the data to the server in a single parameterized command. The incoming data rows are stored in a table variable that can then be operated on by using Transact-SQL.

Column values in table-valued parameters can be accessed using standard Transact-SQL SELECT statements. Table-valued parameters are strongly typed and their structure is automatically validated. The size of table-valued parameters is limited only by server memory.

NOTE

Support for Table-Valued Parameters is available starting with Microsoft JDBC Driver 6.0 for SQL Server.

You cannot return data in a table-valued parameter. Table-valued parameters are input-only; the OUTPUT keyword is not supported.

For more information about table-valued parameters, see the following resources.

RESOURCE	DESCRIPTION
Table-Valued Parameters (Database Engine) in SQL Server Books Online	Describes how to create and use table-valued parameters
User-Defined Table Types in SQL Server Books Online	Describes user-defined table types that are used to declare table-valued parameters
The Microsoft SQL Server Database Engine section of CodePlex	Contains samples that demonstrate how to use SQL Server features and functionality

Passing Multiple Rows in Previous Versions of SQL Server

Before table-valued parameters were introduced to SQL Server 2008, the options for passing multiple rows of data to a stored procedure or a parameterized SQL command were limited. A developer could choose from the following options for passing multiple rows to the server:

- Use a series of individual parameters to represent the values in multiple columns and rows of data. The amount of data that can be passed by using this method is limited by the number of parameters allowed. SQL Server procedures can have, at most, 2100 parameters. Server-side logic is required to assemble these individual values into a table variable or a temporary table for processing.
- Bundle multiple data values into delimited strings or XML documents and then pass those text values to a procedure or statement. This requires the procedure or statement to include the logic necessary for validating the data structures and unbundling the values.
- Create a series of individual SQL statements for data modifications that affect multiple rows. Changes can

be submitted to the server individually or batched into groups. However, even when submitted in batches that contain multiple statements, each statement is executed separately on the server.

- Use the bcp utility program or the SQLServerBulkCopy object to load many rows of data into a table. Although this technique is very efficient, it does not support server-side processing unless the data is loaded into a temporary table or table variable.

Creating Table-Valued Parameter Types

Table-valued parameters are based on strongly-typed table structures that are defined by using Transact-SQL `CREATE TYPE` statements. You have to create a table type and define the structure in SQL Server before you can use table-valued parameters in your client applications. For more information about creating table types, see User-Defined Table Types in SQL Server Books Online.

```
CREATE TYPE dbo.CategoryTableType AS TABLE
    ( CategoryID int, CategoryName nvarchar(50) )
```

After creating a table type, you can declare table-valued parameters based on that type. The following Transact-SQL fragment demonstrates how to declare a table-valued parameter in a stored procedure definition. Note that the `READONLY` keyword is required for declaring a table-valued parameter.

```
CREATE PROCEDURE usp_UpdateCategories
    (@tvpNewCategories dbo.CategoryTableType READONLY)
```

Modifying Data with Table-Valued Parameters (Transact-SQL)

Table-valued parameters can be used in set-based data modifications that affect multiple rows by executing a single statement. For example, you can select all the rows in a table-valued parameter and insert them into a database table, or you can create an update statement by joining a table-valued parameter to the table you want to update.

The following Transact-SQL UPDATE statement demonstrates how to use a table-valued parameter by joining it to the Categories table. When you use a table-valued parameter with a JOIN in a FROM clause, you must also alias it, as shown here, where the table-valued parameter is aliased as "ec":

```
UPDATE dbo.Categories
    SET Categories.CategoryName = ec.CategoryName
    FROM dbo.Categories INNER JOIN @tvpEditedCategories AS ec
    ON dbo.Categories.CategoryID = ec.CategoryID;
```

This Transact-SQL example demonstrates how to select rows from a table-valued parameter to perform an INSERT in a single set-based operation.

```
INSERT INTO dbo.Categories (CategoryID, CategoryName)
    SELECT nc.CategoryID, nc.CategoryName FROM @tvpNewCategories AS nc;
```

Limitations of Table-Valued Parameters

There are several limitations to table-valued parameters:

- You cannot pass table-valued parameters to user defined functions.

- Table-valued parameters can only be indexed to support UNIQUE or PRIMARY KEY constraints. SQL

Server does not maintain statistics on table-valued parameters.

- Table-valued parameters are read-only in Transact-SQL code. You cannot update the column values in the rows of a table-valued parameter and you cannot insert or delete rows. To modify the data that is passed to a stored procedure or parameterized statement in table-valued parameter, you must insert the data into a temporary table or into a table variable.
- You cannot use ALTER TABLE statements to modify the design of table-valued parameters.
- You can stream large objects in a table-valued parameter.

Configuring a Table-Valued Parameter

Beginning with Microsoft JDBC Driver 6.0 for SQL Server, table-valued parameters are supported with a parameterized statement or a parameterized stored procedure. Table-valued parameters can be populated from a SQLServerDataTable, from a ResultSet or from a user provided implementation of the ISQLServerDataRecord interface. When setting a table-valued parameter for a prepared query, you must specify a type name which must match the name of a compatible type previously created on the server.

The following two code fragments demonstrate how to configure a table-valued parameter with a SQLServerPreparedStatement and with a SQLServerCallableStatement to insert data. Here sourceTVPObject can be a SQLServerDataTable, or a ResultSet or an ISQLServerDataRecord object. The examples assume connection is an active Connection object.

```
// Using table-valued parameter with a SQLServerPreparedStatement.
SQLServerPreparedStatement pStmt =
    (SQLServerPreparedStatement) connection.prepareStatement(“INSERT INTO dbo.Categories SELECT * FROM ?”);
pStmt.setStructured(1, "dbo.CategoryTableType", sourceTVPObject);
pStmt.execute();
```

```
// Using table-valued parameter with a SQLServerCallableStatement.
SQLServerCallableStatement pStmt =
    (SQLServerCallableStatement) connection.prepareCall("exec usp_InsertCategories ?");
pStmt.setStructured(1, "dbo.CategoryTableType", sourceTVPObject);;
pStmt.execute();
```

NOTE

See Section **Table-Valued Parameter API for the JDBC Driver** below for a complete list of APIs available for setting the table-valued parameter.

Passing a Table-Valued Parameter as a SQLServerDataTable Object

Beginning with Microsoft JDBC Driver 6.0 for SQL Server, the SQLServerDataTable class represents an in-memory table of relational data. This example demonstrates how to construct a table-valued parameter from in-memory data using the SQLServerDataTable object. The code first creates a SQLServerDataTable object, defines its schema and populates the table with data. The code then configures a SQLServerPreparedStatement that passes this data table as a table-valued parameter to SQL Server.

```
/* Assumes connection is an active Connection object. */

// Create an in-memory data table.
SQLServerDataTable sourceDataTable = new SQLServerDataTable();

// Define metadata for the data table.
sourceDataTable.addColumnMetadata("CategoryID" ,java.sql.Types.INTEGER);
sourceDataTable.addColumnMetadata("CategoryName" ,java.sql.Types.NVARCHAR);

// Populate the data table.
sourceDataTable.addRow(1, "CategoryNameValue1");
sourceDataTable.addRow(2, "CategoryNameValue2");

// Pass the data table as a table-valued parameter using a prepared statement.
SQLServerPreparedStatement pStmt =
        (SQLServerPreparedStatement) connection.prepareStatement(
            "INSERT INTO dbo.Categories SELECT * FROM ?;");
pStmt.setStructured(1, "dbo.CategoryTableType", sourceDataTable);
pStmt.execute();
```

NOTE

See Section **Table-Valued Parameter API for the JDBC Driver** below for a complete list of APIs available for setting the table-valued parameter.

Passing a Table-Valued Parameter as a ResultSet Object

This example demonstrates how to stream rows of data from a ResultSet to a table-valued parameter. The code first retrieves data from a source table in a creates a SQLServerDataTable object, defines its schema and populates the table with data. The code then configures a SQLServerPreparedStatement that passes this data table as a table-valued parameter to SQL Server.

```
/* Assumes connection is an active Connection object. */

// Create the source ResultSet object. Here SourceCategories is a table defined with the same schema as
Categories table.
ResultSet sourceResultSet = connection.createStatement().executeQuery("SELECT * FROM SourceCategories");

// Pass the source result set as a table-valued parameter using a prepared statement.
SQLServerPreparedStatement pStmt =
        (SQLServerPreparedStatement) connection.prepareStatement(
                "INSERT INTO dbo.Categories SELECT * FROM ?;");
pStmt.setStructured(1, "dbo.CategoryTableType", sourceResultSet);
pStmt.execute();
```

NOTE

See Section **Table-Valued Parameter API for the JDBC Driver** below for a complete list of APIs available for setting the table-valued parameter.

Passing a Table-Valued Parameter as an ISQLServerDataRecord Object

Beginning with Microsoft JDBC Driver 6.0 for SQL Server, a new interface ISQLServerDataRecord is available for streaming data (depending on how the user provides the implementation for it) using a table-valued parameter. The following example demonstrates how to implement the ISQLServerDataRecord interface and how to pass it as a table-valued parameter. For simplicity, the following example passes just one row with hardcoded values to the table-valued parameter. Ideally, the user would implement this interface to stream rows from any source, for example from text files.

```
class MyRecords implements ISQLServerDataRecord
{
    int currentRow = 0;
    Object[] row = new Object[2];

    MyRecords(){
        // Constructor. This implementation has just one row.
        row[0] = new Integer(1);
        row[1] = "categoryName1";
    }

    public int getColumnCount(){
        // Return the total number of columns, for this example it is 2.
        return 2;
    }

    public SQLServerMetaData getColumnMetaData(int columnIndex) {
        // Return the column metadata.
        if (1 == columnIndex)
            return new SQLServerMetaData("CategoryID", java.sql.Types.INTEGER);
        else
            return new SQLServerMetaData("CategoryName", java.sql.Types.NVARCHAR);
    }

    public Object[] getRowData(){
        // Return the columns in the current row as an array of objects. This implementation has just one row.
        return row;
    }

    public boolean next(){
        // Move to the next row. This implementation has just one row, after processing the first row, return
false.
        currentRow++;
        if (1 == currentRow)
            return true;
        else
            return false;
    }
}

// Following code demonstrates how to pass MyRecords object as a table-valued parameter.
MyRecords sourceRecords = new MyRecords();
SQLServerPreparedStatement pStmt =
        (SQLServerPreparedStatement) connection.prepareStatement(
                "INSERT INTO dbo.Categories SELECT * FROM ?;");
pStmt.setStructured(1, "dbo.CategoryTableType", sourceRecords);
pStmt.execute();
```

NOTE

See Section **Table-Valued Parameter API for the JDBC Driver** below for a complete list of APIs available for setting the table-valued parameter.

Table-Valued Parameter API for the JDBC Driver

SQLServerMetaData

This class represents metadata for a column. It is used in the ISQLServerDataRecord interface to pass column metadata to the table-valued parameter. The methods in this class are:

NAME	DESCRIPTION
public SQLServerMetaData(String columnName, int sqlType, int precision, int scale, boolean useServerDefault, boolean isUniqueKey, SQLServerSortOrder sortOrder, int sortOrdinal)	Initializes a new instance of SQLServerMetaData with the specified column name, sql type, precision, scale and server default. This form of the constructor supports table-valued parameters by allowing you to specify if the column is unique in the table-valued parameter, the sort order for the column, and the ordinal of the sort column. useServerDefault - specifies if this column should use the default server value; Default value is false. isUniqueKey - indicates if the column in the table-valued parameter is unique; Default value is false. sortOrder - indicates the sort order for a column; Default value is SQLServerSortOrder.Unspecified. sortOrdinal - specifies ordinal of the sort column; sortOrdinal starts from 0; Default value is -1.
public SQLServerMetaData(String columnName, int sqlType)	Initializes a new instance of SQLServerMetaData using the column name and the sql type.
public SQLServerMetaData(String columnName, int sqlType, int precision, int scale)	Initializes a new instance of SQLServerMetaData using the column name, sql type, precision and scale.
Public SQLServerMetaData(SQLServerMetaData sqlServerMetaData)	Initializes a new instance of SQLServerMetaData fron another SQLServerMetaData object.
public String getColumName()	Retrieves the column name.
public int getSqlType()	Retrieves the java sql Type.
public int getPrecision()	Retrieves the precision of the type passed to the column.
public int getScale()	Retrieves the scale of the type passed to the column.
public SQLServerSortOrder getSortOrder()	Retrieves the sort order.
public int getSortOrdinal()	Retrieves the sort ordinal.
public boolean isUniqueKey()	Returns whether the column is unique.
public boolean useServerDefault()	Returns wheher the column uses the default server value.

SQLServerSortOrder

An Enum that defines the sort order. Possible values are Ascending, Descending and Unspecified.

SQLServerDataTable

This class represents an in-memory data table to be used with table-valued parameters. The methods in this class are:

NAME	DESCRIPTION
Public SQLServerDataTable()	Initializes a new instance of SQLServerDataTable.
public Iterator<Entry<Integer, Object[]>> getIterator()	Retrieves an iterator on the rows of the data table.

NAME	DESCRIPTION
public void addColumnMetadata(String columnName, int sqlType)	Adds metadata for the specified column.
public void addColumnMetadata(SQLServerDataColumn column)	Adds metadata for the specified column.
public void addRow(Object... values)	Adds one row of data to the data table.
public Map<Integer, SQLServerDataColumn> getColumnMetadata()	Retrieves column meta data of this data table.
public void clear()	Clears this data table.

SQLServerDataColumn

This class represents a column of the in-memory data table represented by SQLServerDataTable. The methods in this class are:

NAME	DESCRIPTION
public SQLServerDataColumn(String columnName, int sqlType)	Initializes a new instance of SQLServerDataColumn with the column name and type.
public String getColumnName()	Retrieves the column name.
public int getColumnType()	Retrieves the column type.

ISQLServerDataRecord

This class represents an interface that users can implement to stream data to a table-valued parameter. The methods in this interface are:

NAME	DESCRIPTION
public SQLServerMetaData getColumnMetaData(int column);	Retrieves the column meta data of the given column index.
public int getColumnCount();	Retrieves the total number of columns.
public Object[] getRowData();	Retrieves the data for the current row as an array of Objects.
public boolean next();	Moves to the next row. Returns True if the move is successful and there is a next row, false otherwise.

SQLServerPreparedStatement

The following methods have been added to this class to support passing of table-valued parameters.

NAME	DESCRIPTION
public final void setStructured(int parameterIndex, String tvpName, SQLServerDataTable tvpDataTbale)	Populates a table valued parameter with a data table. parameterIndex is the parameter index, tvpName is the name of the table-valued parameter, and tvpDataTable is the source data table object.

NAME	DESCRIPTION
public final void setStructured(int parameterIndex, String tvpName, ResultSet tvpResultSet)	Populates a table valued parameter with a ResultSet retrieved from anther table. parameterIndex is the parameter index, tvpName is the name of the table-valued parameter, and tvpResultSet is the source result set object.
public final void setStructured(int parameterIndex, String tvpName, ISQLServerDataRecord tvpDataRecord)	Populates a table valued parameter with an ISQLServerDataRecord object. ISQLServerDataRecord is used for streaming data and the user decides how to use it. parameterIndex is the parameter index, tvpName is the name of the table-valued parameter, and tvpDataRecord is an ISQLServerDataRecord object.

SQLServerCallableStatement

The following methods have been added to this class to support passing of table-valued parameters.

NAME	DESCRIPTION
public final void setStructured(String paratemeterName, String tvpName, SQLServerDataTable tvpDataTable)	Populates a table valued parameter passed to a stored procedure with a data table. paratemeterName is the name of the parameter, tvpName is the name of the type TVP, and tvpDataTable is the data table object.
public final void setStructured(String paratemeterName, String tvpName, ResultSet tvpResultSet)	Populates a table valued parameter passed to a stored procedure with a ResultSet retrieved from another table. paratemeterName is the name of the parameter, tvpName is the name of the type TVP, and tvpResultSet is the source result set object.
public final void setStructured(String paratemeterName, String tvpName, ISQLServerDataRecord tvpDataRecord)	Populates a table valued parameter passed to a stored procedure with an ISQLServerDataRecord object. ISQLServerDataRecord is used for streaming data and the user decides how to use it. paratemeterName is the name of the parameter, tvpName is the name of the type TVP, and tvpDataRecord is an ISQLServerDataRecord object.

See Also

Overview of the JDBC Driver

Supporting XML Data

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

SQL Server provides an **xml** data type that lets you store XML documents and fragments in a SQL Server database. The **xml** data type is a built-in data type in SQL Server, and is in some ways similar to other built-in types, such as **int** and **varchar**. Like other built-in types, you can use the **xml** data type as: a variable type, a parameter type, a function-return type, or a column type when you create a table; or in Transact-SQL CAST and CONVERT functions. In the JDBC driver, the **xml** data type can be mapped as a String, byte array, stream, CLOB, BLOB, or SQLXML object. String is the default mapping.

The JDBC driver provides support for the JDBC 4.0 API, which introduces the SQLXML interface. The SQLXML interface defines methods to interact with and manipulate XML data. The **SQLXML** is a JDBC 4.0 data type and it maps to the SQL Server**xml** data type. Therefore, in order to use the SQLXML data type in your applications, you must set the classpath to include the sqljdbc4.jar file. If the application tries to use the sqljdbc3.jar when accessing the SQLXML object and its methods, an exception is thrown.

IMPORTANT

SQL Server always validates the XML data before storing it in the database column. Applications can use **SQLXML** data type, because the JDBC driver maps it to the **xml** data type automatically. The **SQLXML** support is available in sqljdbc4.jar. See System Requirements for the JDBC Driver for the list of JRE versions supported by the Microsoft JDBC Driver for SQL Server.

The topics in this section describe the SQLXML interface and how to program against the **SQLXML** data type by using the JDBC API methods.

In This Section

TOPIC	DESCRIPTION
SQLXML Interface	Describes the SQLXML interface and its methods.
Programming with SQLXML	Describes how to use the Microsoft JDBC Driver for SQL Server API methods to store and retrieve an XML data in and from a relational database with the **SQLXML** Java data type. Also contains information about the types of SQLXML objects and provides a list of important guidelines and limitations when using SQLXML objects.

See Also

Understanding the JDBC Driver Data Types

Using an SQL Statement with Parameters

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

To work with data in a SQL Server database by using an SQL statement that contains IN parameters, you can use the executeQuery method of the SQLServerPreparedStatement class to return a SQLServerResultSet that will contain the requested data. To do this, you must first create a SQLServerPreparedStatement object by using the prepareStatement method of the SQLServerConnection class.

When you construct your SQL statement, the IN parameters are specified by using the ? (question mark) character, which acts as a placeholder for the parameter values that will later be passed into the SQL statement. To specify a value for a parameter, you can use one of the setter methods of the SQLServerPreparedStatement class. The setter method that you use is determined by the data type of the value that you want to pass into the SQL statement.

When you pass a value to the setter method, you must specify not only the actual value to be used in the SQL statement, but also the parameter's ordinal placement in the SQL statement. For example, if your SQL statement contains a single parameter, its ordinal value will be 1. If the statement contains two parameters, the first ordinal value will be 1, while the second ordinal value will be 2.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, an SQL prepared statement is constructed and run with a single String parameter value, and then the results are read from the result set.

```
public static void executeStatement(Connection con) {
    try(PreparedStatement pstmt = con.prepareStatement("SELECT LastName, FirstName FROM Person.Contact WHERE
LastName = ?");) {
        pstmt.setString(1, "Smith");
        ResultSet rs = pstmt.executeQuery();

        while (rs.next()) {
            System.out.println(rs.getString("LastName") + ", " + rs.getString("FirstName"));
        }
    }
    // Handle any errors that may have occurred.
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

See Also

Using Statements with SQL

Understanding Isolation Levels

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

Transactions specify an isolation level that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions. Isolation levels are described in terms of which concurrency side effects, such as dirty reads or phantom reads, are allowed.

Transaction isolation levels control the following:

- Whether locks are taken when data is read, and what type of locks are requested.
- How long the read locks are held.
- Whether a read operation referencing rows modified by another transaction:
  - Block until the exclusive lock on the row is freed.
  - Retrieve the committed version of the row that existed at the time the statement or transaction started.
  - Read the uncommitted data modification.

Choosing a transaction isolation level doesn't affect the locks that are acquired to protect data modifications. A transaction always gets an exclusive lock on any data it modifies and holds that lock until the transaction completes, regardless of the isolation level set for that transaction. For read operations, transaction isolation levels primarily define the level of protection from the effects of modifications made by other transactions.

A lower isolation level increases the ability of many users to access data at the same time, but increases the number of concurrency effects, such as dirty reads or lost updates, that users might encounter. Conversely, a higher isolation level reduces the types of concurrency effects that users might encounter, but requires more system resources and increases the chances that one transaction will block another. Choosing the appropriate isolation level depends on balancing the data integrity requirements of the application against the overhead of each isolation level. The highest isolation level, serializable, guarantees that a transaction will retrieve exactly the same data every time it repeats a read operation, but it does this by performing a level of locking that is likely to impact other users in multi-user systems. The lowest isolation level, read uncommitted, can retrieve data that has been modified but not committed by other transactions. All concurrency side effects can happen in read uncommitted, but there's no read locking or versioning, so overhead is minimized.

Remarks

The following table shows the concurrency side effects allowed by the different isolation levels.

ISOLATION LEVEL	DIRTY READ	NON REPEATABLE READ	PHANTOM
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes

ISOLATION LEVEL	DIRTY READ	NON REPEATABLE READ	PHANTOM
Snapshot	No	No	No
Serializable	No	No	No

Transactions must be run at an isolation level of at least repeatable read to prevent lost updates that can occur when two transactions each retrieve the same row, and then later update the row based on the originally retrieved values. If the two transactions update rows using a single UPDATE statement and don't base the update on the previously retrieved values, lost updates can't occur at the default isolation level of read committed.

To set the isolation level for a transaction, you can use the setTransactionIsolation method of the SQLServerConnection class. This method accepts an **int** value as its argument, which is based on one of the connection constants as in the following:

```
con.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
```

To use the new snapshot isolation level of SQL Server, you can use one of the `SQLServerConnection` constants:

```
con.setTransactionIsolation(SQLServerConnection.TRANSACTION_SNAPSHOT);
```

or you can use:

```
con.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED + 4094);
```

For more information about SQL Server isolation levels, see "Isolation Levels in the Database Engine" in SQL Server Books Online.

See Also

Performing Transactions with the JDBC Driver

Wrappers and Interfaces

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server supports interfaces that allow you create a proxy of a class, and wrappers that let you access extensions to the JDBC API that are specific to the Microsoft JDBC Driver for SQL Server through a proxy interface.

Wrappers

The Microsoft JDBC Driver for SQL Server supports the java.sql.Wrapper interface. This interface provides a mechanism to access extensions to the JDBC API that are specific to the Microsoft JDBC Driver for SQL Server through a proxy interface.

The java.sql.Wrapper interface defines two methods: **isWrapperFor** and **unwrap**. The **isWrapperFor** method checks whether the specified input object implements this interface. The **unwrap** method returns an object that implements this interface to allow access to the Microsoft JDBC Driver for SQL Server specific methods.

isWrapperFor and **unwrap** methods are exposed as follows:

- isWrapperFor Method (SQLServerCallableStatement)
- unwrap Method (SQLServerCallableStatement)
- isWrapperFor Method (SQLServerConnectionPoolDataSource)
- unwrap Method (SQLServerConnectionPoolDataSource)
- isWrapperFor Method (SQLServerDataSource)
- unwrap Method (SQLServerDataSource)
- isWrapperFor Method (SQLServerPreparedStatement)
- unwrap Method (SQLServerPreparedStatement)
- isWrapperFor Method (SQLServerStatement)
- unwrap Method (SQLServerStatement)
- isWrapperFor Method (SQLServerXADataSource)
- unwrap Method (SQLServerXADataSource)

Interfaces

Beginning in SQL Server JDBC Driver 3.0, interfaces are available for an application server to access a driver specific method from the associated class. The application server can wrap the class by creating a proxy, exposing the Microsoft JDBC Driver for SQL Server-specific functionality from an interface. The Microsoft JDBC Driver for SQL Server supports interfaces that have the Microsoft JDBC Driver for SQL Server specific methods and constants so an application server can create a proxy of the class.

The interfaces derive from standard Java interfaces so you can use the same object once it is unwrapped to access driver specific functionality or generic Microsoft JDBC Driver for SQL Server functionality.

The following interfaces are added:

- ISQLServerCallableStatement
- ISQLServerConnection
- ISQLServerDataSource
- ISQLServerPreparedStatement
- ISQLServerResultSet
- ISQLServerStatement

Example

Description

This sample demonstrates how to access to a Microsoft JDBC Driver for SQL Server-specific function from a DataSource object. This DataSource class may have been wrapped by an application server. To access the JDBC driver-specific function or constant, you can unwrap the datasource to an ISQLServerDataSource interface and use the functions declared in this interface.

Code

```
import javax.sql.*;
import java.sql.*;
import com.microsoft.sqlserver.jdbc.*;

public class UnWrapTest {
   public static void main(String[] args) {
      // This is a test.  This DataSource object could be something from an appserver
      // which has wrapped the real SQLServerDataSource with its own wrapper
      SQLServerDataSource ds = new SQLServerDataSource();
      checkSendStringParametersAsUnicode(ds);
   }

   // Unwrap to the ISQLServerDataSource interface to access the getSendStringParametersAsUnicode function
   static void checkSendStringParametersAsUnicode(DataSource ds) {
      try {
         final ISQLServerDataSource sqlServerDataSource = ds.unwrap(ISQLServerDataSource.class);
         boolean sendStringParametersAsUnicode = sqlServerDataSource.getSendStringParametersAsUnicode();

         System.out.println("Send string as parameter value is:-" + sendStringParametersAsUnicode);

      } catch (SQLException sqlE) {
         System.out.println("Exception:-" + sqlE);
      }
   }
}
```

See Also

Understanding the JDBC Driver Data Types

Using Bulk Copy with the JDBC Driver

8/2/2018 • 30 minutes to read • Edit Online

Download JDBC Driver

Microsoft SQL Server includes a popular command-line utility named **bcp** for quickly bulk copying large files into tables or views in SQL Server databases. The SQLServerBulkCopy class allows you to write code solutions in Java that provide similar functionality. There are other ways to load data into a SQL Server table (INSERT statements, for example) but SQLServerBulkCopy offers a significant performance advantage over them.

The SQLServerBulkCopy class can be used to write data only to SQL Server tables. But the data source isn't limited to SQL Server; any data source can be used, as long as the data can be read with a ResultSet, RowSet, or ISQLServerBulkRecord implementation.

Using the SQLServerBulkCopy class, you can perform:

- A single bulk copy operation
- Multiple bulk copy operations
- A bulk copy operation with a transaction

NOTE

When using the Microsoft JDBC Driver 4.1 for SQL Server or earlier (which does not support the SQLServerBulkCopy class), you can execute the SQL Server Transact-SQL BULK INSERT statement instead.

Bulk copy example setup

The SQLServerBulkCopy class can be used to write data only to SQL Server tables. The code samples shown in this article use the SQL Server sample database, AdventureWorks. To avoid altering the existing tables code samples write data to tables that you must create first.

The BulkCopyDemoMatchingColumns and BulkCopyDemoDifferentColumns tables are both based on the AdventureWorks Production.Products table. In code samples that use these tables, data is added from the Production.Products table to one of these sample tables. The BulkCopyDemoDifferentColumns table is used when the sample illustrates how to map columns from the source data to the destination table; BulkCopyDemoMatchingColumns is used for most other samples.

A few of the code samples demonstrate how to use one SQLServerBulkCopy class to write to multiple tables. For these samples, the BulkCopyDemoOrderHeader and BulkCopyDemoOrderDetail tables are used as the destination tables. These tables are based on the Sales.SalesOrderHeader and Sales.SalesOrderDetail tables in AdventureWorks.

NOTE

The SQLServerBulkCopy code samples are provided to demonstrate the syntax for using SQLServerBulkCopy only. If the source and destination tables are located in the same SQL Server instance, it is easier and faster to use a Transact-SQL INSERT ... SELECT statement to copy the data.

Table setup

To create the tables necessary for the code samples to run correctly, you must run the following Transact-SQL statements in a SQL Server database.

```
USE AdventureWorks

IF EXISTS (SELECT * FROM dbo.sysobjects
 WHERE id = object_id(N'[dbo].[BulkCopyDemoMatchingColumns]')
 AND OBJECTPROPERTY(id, N'IsUserTable') = 1)
    DROP TABLE [dbo].[BulkCopyDemoMatchingColumns]

CREATE TABLE [dbo].[BulkCopyDemoMatchingColumns]([ProductID] [int] IDENTITY(1,1) NOT NULL,
    [Name] [nvarchar](50) NOT NULL,
    [ProductNumber] [nvarchar](25) NOT NULL,
 CONSTRAINT [PK_ProductID] PRIMARY KEY CLUSTERED
(
    [ProductID] ASC
) ON [PRIMARY]) ON [PRIMARY]

IF EXISTS (SELECT * FROM dbo.sysobjects
 WHERE id = object_id(N'[dbo].[BulkCopyDemoDifferentColumns]')
 AND OBJECTPROPERTY(id, N'IsUserTable') = 1)
    DROP TABLE [dbo].[BulkCopyDemoDifferentColumns]

CREATE TABLE [dbo].[BulkCopyDemoDifferentColumns]([ProdID] [int] IDENTITY(1,1) NOT NULL,
    [ProdNum] [nvarchar](25) NOT NULL,
    [ProdName] [nvarchar](50) NOT NULL,
 CONSTRAINT [PK_ProdID] PRIMARY KEY CLUSTERED
(
    [ProdID] ASC
) ON [PRIMARY]) ON [PRIMARY]

IF EXISTS (SELECT * FROM dbo.sysobject
 WHERE id = object_id(N'[dbo].[BulkCopyDemoOrderHeader]')
 AND OBJECTPROPERTY(id, N'IsUserTable') = 1)
    DROP TABLE [dbo].[BulkCopyDemoOrderHeader]

CREATE TABLE [dbo].[BulkCopyDemoOrderHeader]([SalesOrderID] [int] IDENTITY(1,1) NOT NULL,
    [OrderDate] [datetime] NOT NULL,
    [AccountNumber] [nvarchar](15) NULL,
 CONSTRAINT [PK_SalesOrderID] PRIMARY KEY CLUSTERED
(
    [SalesOrderID] ASC
) ON [PRIMARY]) ON [PRIMARY]

IF EXISTS (SELECT * FROM dbo.sysobjects
 WHERE id = object_id(N'[dbo].[BulkCopyDemoOrderDetail]')
 AND OBJECTPROPERTY(id, N'IsUserTable') = 1)
    DROP TABLE [dbo].[BulkCopyDemoOrderDetail]

CREATE TABLE [dbo].[BulkCopyDemoOrderDetail]([SalesOrderID] [int] NOT NULL,
    [SalesOrderDetailID] [int] NOT NULL,
    [OrderQty] [smallint] NOT NULL,
    [ProductID] [int] NOT NULL,
    [UnitPrice] [money] NOT NULL,
 CONSTRAINT [PK_LineNumber] PRIMARY KEY CLUSTERED
(
    [SalesOrderID] ASC,
    [SalesOrderDetailID] ASC
) ON [PRIMARY]) ON [PRIMARY]
```

Single bulk copy operations

The simplest approach to performing a SQL Server bulk copy operation is to perform a single operation against a database. By default, a bulk copy operation is performed as an isolated operation: the copy operation occurs in a non-transacted way, with no opportunity for rolling it back.

NOTE

If you need to roll back all or part of the bulk copy when an error occurs, you can either use a SQLServerBulkCopy-managed transaction, or perform the bulk copy operation within an existing transaction.
For more information, see Transaction and bulk copy operations

The general steps to perform a bulk copy operation are:

1. Connect to the source server and obtain the data to be copied. Data can also come from other sources, if it can be retrieved from a ResultSet object or an ISQLServerBulkRecord implementation.

2. Connect to the destination server (unless you want **SQLServerBulkCopy** to establish a connection for you).

3. Create a SQLServerBulkCopy object, setting any necessary properties via **setBulkCopyOptions**.

4. Call the **setDestinationTableName** method to indicate the target table for the bulk insert operation.

5. Call one of the **writeToServer** methods.

6. Optionally, update properties via **setBulkCopyOptions** and call **writeToServer** again as necessary.

7. Call **close**, or wrap the bulk copy operations within a try-with-resources statement.

Caution

We recommend that the source and target column data types match. If the data types do not match, SQLServerBulkCopy attempts to convert each source value to the target data type. Conversions can affect performance, and also can result in unexpected errors. For example, a double data type can be converted to a decimal data type most of the time, but not always.

Example

The following application demonstrates how to load data using the SQLServerBulkCopy class. In this example, a ResultSet is used to copy data from the Production.Product table in the SQL Server AdventureWorks database to a similar table in the same database.

IMPORTANT

This sample will not run unless you have created the work tables as described in Table setup. This code is provided to demonstrate the syntax for using SQLServerBulkCopy only. If the source and destination tables are located in the same SQL Server instance, it is easier and faster to use a Transact-SQL INSERT … SELECT statement to copy the data.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerBulkCopy;

public class BulkCopySingle {
    public static void main(String[] args) {
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";
        String destinationTable = "dbo.BulkCopyDemoMatchingColumns";
        int countBefore, countAfter;
        ResultSet rsSourceData;

        try (Connection sourceConnection = DriverManager.getConnection(connectionUrl);
                Connection destinationConnection = DriverManager.getConnection(connectionUrl);
                Statement stmt = sourceConnection.createStatement();
                SQLServerBulkCopy bulkCopy = new SQLServerBulkCopy(destinationConnection)) {

            // Empty the destination table.
            stmt.executeUpdate("DELETE FROM " + destinationTable);

            // Perform an initial count on the destination table.
            countBefore = getRowCount(stmt, destinationTable);

            // Get data from the source table as a ResultSet.
            rsSourceData = stmt.executeQuery("SELECT ProductID, Name, ProductNumber FROM Production.Product");

            // In real world applications you would
            // not use SQLServerBulkCopy to move data from one table to the other
            // in the same database. This is for demonstration purposes only.

            // Set up the bulk copy object.
            // Note that the column positions in the source
            // table match the column positions in
            // the destination table so there is no need to
            // map columns.
            bulkCopy.setDestinationTableName(destinationTable);

            // Write from the source to the destination.
            bulkCopy.writeToServer(rsSourceData);

            // Perform a final count on the destination
            // table to see how many rows were added.
            countAfter = getRowCount(stmt, destinationTable);
            System.out.println((countAfter - countBefore) + " rows were added.");
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static int getRowCount(Statement stmt,
            String tableName) throws SQLException {
        ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM " + tableName);
        rs.next();
        int count = rs.getInt(1);
        rs.close();
        return count;
    }
}
```

Performing a bulk copy operation using Transact-SQL

The following example illustrates how to use the executeUpdate method to execute the BULK INSERT statement.

NOTE

The file path for the data source is relative to the server. The server process must have access to that path in order for the bulk copy operation to succeed.

```
try (Connection con = DriverManager.getConnection(connectionUrl);
        Statement stmt = con.createStatement()) {
    // Perform the BULK INSERT
    stmt.executeUpdate(
            "BULK INSERT Northwind.dbo.[Order Details] " + "FROM 'f:\\mydata\\data.tbl' " + "WITH (
FORMATFILE='f:\\mydata\\data.fmt' )");
}
```

Multiple bulk copy operations

You can perform multiple bulk copy operations using a single instance of a SQLServerBulkCopy class. If the operation parameters change between copies (for example, the name of the destination table), you must update them prior to any subsequent calls to any of the writeToServer methods, as demonstrated in the following example. Unless explicitly changed, all property values remain the same as they were on the previous bulk copy operation for a given instance.

NOTE

Performing multiple bulk copy operations using the same instance of SQLServerBulkCopy is usually more efficient than using a separate instance for each operation.

If you perform several bulk copy operations using the same SQLServerBulkCopy object, there are no restrictions on whether source or target information is equal or different in each operation. However, you must ensure that column association information is properly set each time you write to the server.

IMPORTANT

This sample will not run unless you have created the work tables as described in Table setup. This code is provided to demonstrate the syntax for using SQLServerBulkCopy only. If the source and destination tables are located in the same SQL Server instance, it is easier and faster to use a Transact-SQL INSERT … SELECT statement to copy the data.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerBulkCopy;
import com.microsoft.sqlserver.jdbc.SQLServerBulkCopyOptions;

public class BulkCopyMultiple {
    public static void main(String[] args) {
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";
        String destinationHeaderTable = "dbo.BulkCopyDemoOrderHeader";
        String destinationDetailTable = "dbo.BulkCopyDemoOrderDetail";
        int countHeaderBefore, countDetailBefore, countHeaderAfter, countDetailAfter;
        ResultSet rsHeader, rsDetail;
        try (Connection sourceConnection1 = DriverManager.getConnection(connectionUrl);
                Connection sourceConnection2 = DriverManager.getConnection(connectionUrl);
                Statement stmt = sourceConnection1.createStatement();
                PreparedStatement preparedStmt1 = sourceConnection1.prepareStatement(
                        "SELECT [SalesOrderID], [OrderDate], [AccountNumber] FROM [Sales].[SalesOrderHeader]
WHERE [AccountNumber] = ?;");
                PreparedStatement preparedStmt2 = sourceConnection2.prepareStatement(
                        "SELECT [Sales].[SalesOrderDetail].[SalesOrderID], [SalesOrderDetailID], [OrderQty],
[ProductID], [UnitPrice] FROM "
                                + "[Sales].[SalesOrderDetail] INNER JOIN [Sales].[SalesOrderHeader] ON "
                                + "[Sales].[SalesOrderDetail].[SalesOrderID] = [Sales].[SalesOrderHeader].
[SalesOrderID] WHERE [AccountNumber] = ?;");
                SQLServerBulkCopy bulkCopy = new SQLServerBulkCopy(connectionUrl);) {

            // Empty the destination tables.
            stmt.executeUpdate("DELETE FROM " + destinationHeaderTable);
            stmt.executeUpdate("DELETE FROM " + destinationDetailTable);

            // Perform an initial count on the destination
            // table with matching columns.
            countHeaderBefore = getRowCount(stmt, destinationHeaderTable);

            // Perform an initial count on the destination
            // table with different column positions.
            countDetailBefore = getRowCount(stmt, destinationDetailTable);

            // Get data from the source table as a ResultSet.
            // The Sales.SalesOrderHeader and Sales.SalesOrderDetail
            // tables are quite large and could easily cause a timeout
            // if all data from the tables is added to the destination.
            // To keep the example simple and quick, a parameter is
            // used to select only orders for a particular account
            // as the source for the bulk insert.
            preparedStmt1.setString(1, "10-4020-000034");
            rsHeader = preparedStmt1.executeQuery();

            // Get the Detail data in a separate connection.
            preparedStmt2.setString(1, "10-4020-000034");
            rsDetail = preparedStmt2.executeQuery();

            // Create the SQLServerBulkCopySQLServerBulkCopy object.
            SQLServerBulkCopyOptions copyOptions = new SQLServerBulkCopyOptions();
            copyOptions.setBulkCopyTimeout(100);
            bulkCopy.setBulkCopyOptions(copyOptions);
            bulkCopy.setDestinationTableName(destinationHeaderTable);

            // Guarantee that columns are mapped correctly by
            // defining the column mappings for the order.
            bulkCopy.addColumnMapping("SalesOrderID", "SalesOrderID");
            bulkCopy.addColumnMapping("OrderDate", "OrderDate");
            bulkCopy.addColumnMapping("AccountNumber", "AccountNumber");

            // Write rsHeader to the destination.
            bulkCopy.writeToServer(rsHeader);

            // Set up the order details destination.
            bulkCopy.setDestinationTableName(destinationDetailTable);

            // Clear the existing column mappings
            bulkCopy.clearColumnMappings();

            // Add order detail column mappings.
            bulkCopy.addColumnMapping("SalesOrderID", "SalesOrderID");
            bulkCopy.addColumnMapping("SalesOrderDetailID", "SalesOrderDetailID");
            bulkCopy.addColumnMapping("OrderQty", "OrderQty");
            bulkCopy.addColumnMapping("ProductID", "ProductID");
            bulkCopy.addColumnMapping("UnitPrice", "UnitPrice");
            // Write rsDetail to the destination.
            bulkCopy.writeToServer(rsDetail);

            // Perform a final count on the destination
            // tables to see how many rows were added.
            countHeaderAfter = getRowCount(stmt, destinationHeaderTable);
            countDetailAfter = getRowCount(stmt, destinationDetailTable);

            System.out.println((countHeaderAfter - countHeaderBefore) + " rows were added to the Header
table.");
            System.out.println((countDetailAfter - countDetailBefore) + " rows were added to the Detail
table.");
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static int getRowCount(Statement stmt,
            String tableName) throws SQLException {
        ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM " + tableName);
        rs.next();
        int count = rs.getInt(1);
        rs.close();
        return count;
    }
}
```

Transaction and bulk copy operations

Bulk copy operations can be performed as isolated operations or as part of a multiple step transaction. This latter option enables you to perform more than one bulk copy operation within the same transaction, as well as perform other database operations (such as inserts, updates, and deletes) while still being able to commit or roll back the entire transaction.

By default, a bulk copy operation is performed as an isolated operation. The bulk copy operation occurs in a non-transacted way, with no opportunity for rolling it back. If you need to roll back all or part of the bulk copy when an error occurs, you can use a SQLServerBulkCopy-managed transaction or perform the bulk copy operation within an existing transaction.

Performing a non-transacted bulk copy operation

The following application shows what happens when a non-transacted bulk copy operation encounters an error partway through the operation.

In the example, the source table and destination table each include an Identity column named **ProductID**. The code first prepares the destination table by deleting all rows and then inserting a single row whose **ProductID** is known to exist in the source table. By default, a new value for the Identity column is generated in the destination table for each row added. In this example, an option is set when the connection is opened that forces the bulk load process to use the Identity values from the source table instead.

The bulk copy operation is executed with the **BatchSize** property set to 10. When the operation encounters the invalid row, an exception is thrown. In this first example, the bulk copy operation is non-transacted. All batches copied up to the point of the error are committed; the batch containing the duplicate key is rolled back, and the bulk copy operation is halted before processing any other batches.

NOTE

This sample will not run unless you have created the work tables as described in Table setup. This code is provided to demonstrate the syntax for using SQLServerBulkCopy only. If the source and destination tables are located in the same SQL Server instance, it is easier and faster to use a Transact-SQL INSERT … SELECT statement to copy the data.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerBulkCopy;
import com.microsoft.sqlserver.jdbc.SQLServerBulkCopyOptions;

public class BulkCopyNonTransacted {
    public static void main(String[] args) {
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";
        String destinationTable = "dbo.BulkCopyDemoMatchingColumns";
        int countBefore, countAfter;
        ResultSet rsSourceData;

        try (Connection sourceConnection = DriverManager.getConnection(connectionUrl);
                Statement stmt = sourceConnection.createStatement();
                SQLServerBulkCopy bulkCopy = new SQLServerBulkCopy(connectionUrl)) {

            // Empty the destination table.
            stmt.executeUpdate("DELETE FROM " + destinationTable);

            // Add a single row that will result in duplicate key
            // when all rows from source are bulk copied.
            // Note that this technique will only be successful in
            // illustrating the point if a row with ProductID = 446
            // exists in the AdventureWorks Production.Products table.
            // If you have made changes to the data in this table, change
            // the SQL statement in the code to add a ProductID that
            // does exist in your version of the Production.Products
            // table. Choose any ProductID in the middle of the table
            // (not first or last row) to best illustrate the result.
            stmt.executeUpdate("SET IDENTITY_INSERT " + destinationTable + " ON;" + "INSERT INTO " +
destinationTable
                    + "([ProductID], [Name] ,[ProductNumber]) VALUES(446, 'Lock Nut 23','LN-3416'); SET
IDENTITY_INSERT " + destinationTable
                    + " OFF");

            // Perform an initial count on the destination table.
            countBefore = getRowCount(stmt, destinationTable);

            // Get data from the source table as a ResultSet.
            rsSourceData = stmt.executeQuery("SELECT ProductID, Name, ProductNumber FROM Production.Product");

            // Set up the bulk copy object using the KeepIdentity option and BatchSize = 10.
            SQLServerBulkCopyOptions copyOptions = new SQLServerBulkCopyOptions();
            copyOptions.setKeepIdentity(true);
            copyOptions.setBatchSize(10);

            bulkCopy.setBulkCopyOptions(copyOptions);
            bulkCopy.setDestinationTableName(destinationTable);

            // Write from the source to the destination.
            // This should fail with a duplicate key error
            // after some of the batches have been copied.
            try {
                bulkCopy.writeToServer(rsSourceData);
            }
            catch (SQLException e) {
                e.printStackTrace();
            }

            // Perform a final count on the destination
            // table to see how many rows were added.
            countAfter = getRowCount(stmt, destinationTable);
            System.out.println((countAfter - countBefore) + " rows were added.");
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static int getRowCount(Statement stmt,
            String tableName) throws SQLException {
        ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM " + tableName);
        rs.next();
        int count = rs.getInt(1);
        rs.close();
        return count;
    }
}
```

Performing a dedicated build copy operation in a transaction

By default, a bulk copy operation is its own transaction. When you want to perform a dedicated bulk copy operation, create a new instance of SQLServerBulkCopy with a connection string. In this scenario, the bulk copy operation creates, and then commits or rolls back the transaction. You can explicitly specify the **UseInternalTransaction** option in **SQLServerBulkCopyOptions** to explicitly cause a bulk copy operation to execute in its own transaction, causing each batch of the bulk copy operation to execute within a separate transaction.

NOTE

Since different batches are executed in different transactions, if an error occurs during the bulk copy operation, all the rows in the current batch will be rolled back, but rows from previous batches will remain in the database.

When you specify the **UseInternalTransaction** option in **BulkCopyNonTransacted**, the bulk copy operation is included in a larger, external transaction. When the primary key violation error occurs, the entire transaction is rolled back and no rows are added to the destination table.

```
SQLServerBulkCopyOptions copyOptions = new SQLServerBulkCopyOptions();
copyOptions.setKeepIdentity(true);
copyOptions.setBatchSize(10);
copyOptions.setUseInternalTransaction(true);
```

Using existing transactions

You can pass a Connection object that has transactions enabled as a parameter in a SQLServerBulkCopy constructor. In this situation, the bulk copy operation is performed in an existing transaction, and no change is made to the transaction state (that is, it's neither committed nor aborted). This allows an application to include the bulk copy operation in a transaction with other database operations. If you need to roll back the entire bulk copy operation because an error occurs, or if the bulk copy should execute as part of a larger process that can be rolled back, you can perform the rollback on the Connection object at any point after the bulk copy operation.

The following application is similar to **BulkCopyNonTransacted**, with one exception: in this example, the bulk copy operation is included in a larger, external transaction. When the primary key violation error occurs, the entire transaction is rolled back and no rows are added to the destination table.

NOTE

This sample will not run unless you have created the work tables as described in Table setup. This code is provided to demonstrate the syntax for using SQLServerBulkCopy only. If the source and destination tables are located in the same SQL Server instance, it is easier and faster to use a Transact-SQL INSERT … SELECT statement to copy the data.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerBulkCopy;
import com.microsoft.sqlserver.jdbc.SQLServerBulkCopyOptions;

public class BulkCopyExistingTransactions {
    public static void main(String[] args) {
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";
        String destinationTable = "dbo.BulkCopyDemoMatchingColumns";
        int countBefore, countAfter;
        ResultSet rsSourceData;
        SQLServerBulkCopyOptions copyOptions;

        try (Connection sourceConnection = DriverManager.getConnection(connectionUrl);
                Connection destinationConnection = DriverManager.getConnection(connectionUrl);
                Statement stmt = sourceConnection.createStatement();
                SQLServerBulkCopy bulkCopy = new SQLServerBulkCopy(destinationConnection);) {

            // Empty the destination table.
            stmt.executeUpdate("DELETE FROM " + destinationTable);

            // Add a single row that will result in duplicate key
            // when all rows from source are bulk copied.
            // Note that this technique will only be successful in
            // illustrating the point if a row with ProductID = 446
            // exists in the AdventureWorks Production.Products table.
            // If you have made changes to the data in this table, change
            // the SQL statement in the code to add a ProductID that
            // does exist in your version of the Production.Products
            // table. Choose any ProductID in the middle of the table
            // (not first or last row) to best illustrate the result.
            stmt.executeUpdate("SET IDENTITY_INSERT " + destinationTable + " ON;" + "INSERT INTO " +
destinationTable
                    + "([ProductID], [Name] ,[ProductNumber]) VALUES(446, 'Lock Nut 23','LN-3416'); SET
IDENTITY_INSERT " + destinationTable
                    + " OFF");

            // Perform an initial count on the destination table.
            countBefore = getRowCount(stmt, destinationTable);

            // Get data from the source table as a ResultSet.
            rsSourceData = stmt.executeQuery("SELECT ProductID, Name, ProductNumber FROM Production.Product");

            // Set up the bulk copy object inside the transaction.
            destinationConnection.setAutoCommit(false);

            copyOptions = new SQLServerBulkCopyOptions();
            copyOptions.setKeepIdentity(true);
            copyOptions.setBatchSize(10);

            bulkCopy.setBulkCopyOptions(copyOptions);
            bulkCopy.setDestinationTableName(destinationTable);
            // Write from the source to the destination.
            // This should fail with a duplicate key error.
            try {
                bulkCopy.writeToServer(rsSourceData);
                destinationConnection.commit();
            }
            catch (SQLException e) {
                e.printStackTrace();
            }

            // Perform a final count on the destination
            // table to see how many rows were added.
            countAfter = getRowCount(stmt, destinationTable);
            System.out.println((countAfter - countBefore) + " rows were added.");
        }
        catch (Exception e) {
            // Handle any errors that may have occurred.
            e.printStackTrace();
        }
    }

    private static int getRowCount(Statement stmt,
            String tableName) throws SQLException {
        ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM " + tableName);
        rs.next();
        int count = rs.getInt(1);
        rs.close();
        return count;
    }
}
```

Bulk Copy from a CSV File

The following application demonstrates how to load data using the SQLServerBulkCopy class. In this example, a CSV file is used to copy data exported from the Production.Product table in the SQL Server AdventureWorks database to a similar table in the database.

IMPORTANT

This sample will not run unless you have created the work tables as described in Table setup to get it.

1. Open **SQL Server Management Studio** and connect to the SQL Server with the AdventureWorks database.
2. Expand the databases, right-click the AdventureWorks database, select **Tasks** and **Export Data**...
3. For the Data Source, select the **Data source** that allows you to connect to your SQL Server (e.g. SQL Server Native Client 11.0), check the configuration and then **Next**
4. For the Destination, Select the **Flat File Destination** and enter a **File Name** with a destination such as C:\Test\TestBulkCSVExample.csv. Check that the **Format** is Delimited, the **Text qualifier** is none, and enable **Column names in the first data row**, and then select **Next**
5. Select **Write a query to specify the data to transfer** and **Next**. Enter your **SQL Statement** SELECT ProductID, Name, ProductNumber FROM Production.Product, and **Next**
6. Check the configuration: You can leave the Row delimiter as {CR}{LF} and Column Delimiter as Comma {,}. Select **Edit Mappings**... and check that the data **Type** is correct for each column (e.g. integer for ProductID and Unicode string for the others).
7. Skip ahead to **Finish** and run the export.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import com.microsoft.sqlserver.jdbc.SQLServerBulkCSVFileRecord;
import com.microsoft.sqlserver.jdbc.SQLServerBulkCopy;

public class BulkCopyCSV {
    public static void main(String[] args) {
        String connectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=AdventureWorks;user=
<user>;password=<password>";
        String destinationTable = "dbo.BulkCopyDemoMatchingColumns";
        int countBefore, countAfter;

        // Get data from the source file by loading it into a class that implements ISQLServerBulkRecord.
        // Here we are using the SQLServerBulkCSVFileRecord implementation to import the example CSV file.
        try (Connection destinationConnection = DriverManager.getConnection(connectionUrl);
                Statement stmt = destinationConnection.createStatement();
                SQLServerBulkCopy bulkCopy = new SQLServerBulkCopy(destinationConnection);
                SQLServerBulkCSVFileRecord fileRecord = new
SQLServerBulkCSVFileRecord("C:\\Test\\TestBulkCSVExample.csv", true);) {

            // Set the metadata for each column to be copied.
            fileRecord.addColumnMetadata(1, null, java.sql.Types.INTEGER, 0, 0);
            fileRecord.addColumnMetadata(2, null, java.sql.Types.NVARCHAR, 50, 0);
            fileRecord.addColumnMetadata(3, null, java.sql.Types.NVARCHAR, 25, 0);

            // Empty the destination table.
            stmt.executeUpdate("DELETE FROM " + destinationTable);

            // Perform an initial count on the destination table.
            countBefore = getRowCount(stmt, destinationTable);

            // Set up the bulk copy object.
            // Note that the column positions in the source
            // data reader match the column positions in
            // the destination table so there is no need to
            // map columns.
            bulkCopy.setDestinationTableName(destinationTable);

            // Write from the source to the destination.
            bulkCopy.writeToServer(fileRecord);

            // Perform a final count on the destination
            // table to see how many rows were added.
            countAfter = getRowCount(stmt, destinationTable);
            System.out.println((countAfter - countBefore) + " rows were added.");
        }
        // Handle any errors that may have occurred.
        catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static int getRowCount(Statement stmt,
            String tableName) throws SQLException {
        ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM " + tableName);
        rs.next();
        int count = rs.getInt(1);
        rs.close();
        return count;
    }
}
```

Bulk copy with Always Encrypted columns

Beginning with Microsoft JDBC Driver 6.0 for SQL Server, bulk copy is supported with Always Encrypted columns.

Depending on the bulk copy options, and the encryption type of the source and destination tables the JDBC driver may transparently decrypt and then encrypt the data or it may send the encrypted data as is. For example, when bulk copying data from an encrypted column to an unencrypted column, the driver transparently decrypts data before sending to SQL Server. Similarly when bulk copying data from an unencrypted column (or from a CSV file) to an encrypted column, the driver transparently encrypts data before sending to SQL Server. If both source and destination is encrypted, then depending on the **allowEncryptedValueModifications** bulk copy option, the driver would send data as is or would decrypt the data and encrypt it again before sending to SQL Server.

For more information, see the **allowEncryptedValueModifications** bulk copy option below, and Using Always Encrypted with the JDBC Driver.

IMPORTANT

Limitation of the Microsoft JDBC Driver 6.0 for SQL Server, when bulk copying data from a CSV file to encrypted columns:

Only the Transact-SQL default string literal format is supported for the date and time types

DATETIME and SMALLDATETIME datatypes are not supported

Bulk copy API for JDBC driver

SQLServerBulkCopy

Lets you efficiently bulk load a SQL Server table with data from another source.

Microsoft SQL Server includes a popular command-prompt utility named bcp for moving data from one table to another, whether on a single server or between servers. The SQLServerBulkCopy class lets you write code solutions in Java that provide similar functionality. There are other ways to load data into a SQL Server table (INSERT statements, for example), but SQLServerBulkCopy offers a significant performance advantage over them.

The SQLServerBulkCopy class can be used to write data only to SQL Server tables. However, the data source isn't limited to SQL Server; any data source can be used, as long as the data can be read with a ResultSet instance or ISQLServerBulkRecord implementation.

CONSTRUCTOR	DESCRIPTION
SQLServerBulkCopy(Connection)	Initializes a new instance of the SQLServerBulkCopy class using the specified open instance of SQLServerConnection. If the Connection has transactions enabled, the copy operations will be performed within that transaction.
SQLServerBulkCopy(String connectionURL)	Initializes and opens a new instance of SQLServerConnection based on the supplied connectionURL. The constructor uses the SQLServerConnection to initialize a new instance of the SQLServerBulkCopy class.

PROPERTY	DESCRIPTION

PROPERTY	DESCRIPTION
String DestinationTableName	Name of the destination table on the server. If DestinationTableName hasn't been set when writeToServer is called, a SQLServerException is thrown. DestinationTableName is a three-part name (<database>.<owningschema>.<name>). You can qualify the table name with its database and owning schema if you choose. However, if the table name uses an underscore ("_") or any other special characters, you must escape the name using surrounding brackets. For more information, see "Identifiers" in SQL Server Books Online.
ColumnMappings	Column mappings define the relationships between columns in the data source and columns in the destination. If mappings aren't defined the columns are mapped implicitly based on ordinal position. For this to work, source and target schemas must match. If they don't, an Exception will be thrown. If the mappings isn't empty, not every column present in the data source has to be specified. Those not mapped are ignored. You can refer to source and target columns by either name or ordinal.

METHOD	DESCRIPTION
Void addColumnMapping((int sourceColumn, int destinationColumn)	Adds a new column-mapping, using ordinals to specify both source and destination columns.
Void addColumnMapping ((int sourceColumn, String destinationColumn)	Adds a new column-mapping, using an ordinal for the source column and a column name for the destination column.
Void addColumnMapping ((String sourceColumn, int destinationColumn)	Adds a new column-mapping, using a column name to describe the source column and an ordinal to specify the destination column.
Void addColumnMapping (String sourceColumn, String destinationColumn)	Adds a new column-mapping, using column names to specify both source and destination columns.
Void clearColumnMappings()	Clears the contents of the column mappings.
Void close()	Closes the SQLServerBulkCopy instance.
SQLServerBulkCopyOptions getBulkCopyOptions()	Retrieves the current set of SQLServerBulkCopyOptions.
String getDestinationTableName()	Retrieve the current destination table name.
Void setBulkCopyOptions(SQLServerBulkCopyOptions copyOptions)	Updates the behavior of the SQLServerBulkCopy instance according to the options supplied.
Void setDestinationTableName(String tableName)	Sets the name of the destination table.

METHOD	DESCRIPTION
Void writeToServer(ResultSet sourceData)	Copies all rows in the supplied ResultSet to a destination table specified by the DestinationTableName property of the SQLServerBulkCopy object.
Void writeToServer(RowSet sourceData)	Copies all rows in the supplied RowSet to a destination table specified by the DestinationTableName property of the SQLServerBulkCopy object.
Void writeToServer(ISQLServerBulkRecord sourceData)	Copies all rows in the supplied ISQLServerBulkRecord implementation to a destination table specified by the DestinationTableName property of the SQLServerBulkCopy object.

SQLServerBulkCopyOptions

A collection of settings that control how the writeToServer methods behave in an instance of SQLServerBulkCopy.

CONSTRUCTOR	DESCRIPTION
SQLServerBulkCopyOptions()	Initializes a new instance of the SQLServerBulkCopyOptions class using defaults for all of the settings.

Getters and setters exist for the following options:

OPTION	DESCRIPTION	DEFAULT
Boolean CheckConstraints	Check constraints while data is being inserted.	False - constraints aren't checked
Boolean FireTriggers	When specified, cause the server to fire the insert triggers for the rows being inserted into the database.	False - no triggers are fired
Boolean KeepIdentity	Preserve source identity values.	False - identity values are assigned by the destination
Boolean KeepNulls	Preserve null values in the destination table regardless of the settings for default values.	False - null values are replaced by default values where applicable.
Boolean TableLock	Obtain a bulk update lock for the duration of the bulk copy operation.	False - row locks are used.
Boolean UseInternalTransaction	When specified, each batch of the bulk-copy operation will occur within a transaction. If SQLServerBulkCopy is using an existing connection (as specified by the constructor), a SQLServerException will occur. If SQLServerBulkCopy created a dedicated connection, a transaction will be enabled.	False - no transaction

OPTION	DESCRIPTION	DEFAULT
Int BatchSize	Number of rows in each batch. At the end of each batch, the rows in the batch are sent to the server. A batch is complete when BatchSize rows have been processed or there are no more rows to send to the destination data source. If the SQLServerBulkCopy instance has been declared without the UseInternalTransaction option in effect, rows are sent to the server BatchSize rows at a time, but no transaction-related action is taken. If UseInternalTransaction is in effect, each batch of rows is inserted as a separate transaction.	0 - indicates that each writeToServer operation is a single batch
Int BulkCopyTimeout	Number of seconds for the operation to complete before it times out. A value of 0 indicates no limit; the bulk copy will wait indefinitely.	60 seconds.
Boolean allowEncryptedValueModifications	This option is available with Microsoft JDBC Driver 6.0 (or higher) for SQL Server. When specified, **allowEncryptedValueModifications** enables bulk copying of encrypted data between tables or databases, without decrypting the data. Typically, an application would select data from encrypted columns from one table without decrypting the data (the app would connect to the database with the column encryption setting keyword set to disabled) and then would use this option to bulk insert the data, which is still encrypted. For more information, see Using Always Encrypted with the JDBC Driver. Use caution when specifying **allowEncryptedValueModifications** as this may lead to corrupting the database because the driver doesn't check if the data is indeed encrypted, or if it is correctly encrypted using the same encryption type, algorithm and key as the target column.	

Getters and setters:

METHODS	DESCRIPTION
Boolean isCheckConstraints()	Indicates whether constraints are to be checked while data is being inserted or not.

METHODS	DESCRIPTION
Void setCheckConstraints(Boolean checkConstraints)	Sets whether constraints are to be checked while data is being inserted or not.
Boolean isFireTriggers()	Indicates if the server should fire the insert triggers for the rows being inserted into the database.
Void setFireTriggers(Boolean fireTriggers)	Sets whether the server should be set to fire triggers for the rows being inserted into the database.
Boolean isKeepIdentity()	Indicates whether or not to preserve any source identity values.
Void setKeepIdentity(Boolean keepIdentity)	Sets whether or not to preserve identity values.
Boolean isKeepNulls()	Indicates whether to preserve null values in the destination table regardless of the settings for default values, or if they should be replaced by the default values (where applicable).
Void setKeepNulls(Boolean keepNulls)	Sets whether to preserve null values in the destination table regardless of the settings for default values, or if they should be replaced by the default values (where applicable).
Boolean isTableLock()	Indicates whether SQLServerBulkCopy should obtain a bulk update lock for the duration of the bulk copy operation.
Void setTableLock(Boolean tableLock)	Sets whether SQLServerBulkCopy should obtain a bulk update lock for the duration of the bulk copy operation.
Boolean isUseInternalTransaction()	Indicates whether each batch of the bulk-copy operation will occur within a transaction.
Void setUseInternalTranscation(Boolean useInternalTransaction)	Sets whether each batch of the bulk-copy operations will occur within a transaction or not.
Int getBatchSize()	Gets the number of rows in each batch. At the end of each batch, the rows in the batch are sent to the server
Void setBatchSize(int batchSize)	Sets the number of rows in each batch. At the end of each batch, the rows in the batch are sent to the server.
Int getBulkCopyTimeout()	Gets the number of seconds for the operation to complete before it times out.
Void setBulkCopyTimeout(int timeout)	Sets the number of seconds for the operation to complete before it times out.
boolean isAllowEncryptedValueModifications()	Indicates whether allowEncryptedValueModifications setting is enabled or disabled.
void setAllowEncryptedValueModifications(boolean allowEncryptedValueModifications)	Configures the allowEncryptedValueModifications setting that is used for bulk copy with Always Encrypted columns.

ISQLServerBulkRecord

The ISQLServerBulkRecord interface can be used to create classes that read in data from any source (such as a file) and allow a SQLServerBulkCopy instance to bulk load a SQL Server table with that data.

INTERFACE METHODS	DESCRIPTION
Set<Integer> getColumnOrdinals()	Get the ordinals for each of the columns represented in this data record.
String getColumnName(int column)	Get the name of the given column.
Int getColumnType(int column)	Get the JDBC data type of the given column.
Int getPrecision(int column)	Get the precision for the given column.
Object[] getRowData()	Gets the data for the current row as an array of Objects. Each Object must match the Java language Type that is used to represent the indicated JDBC data type for the given column. For more information, see 'Understanding the JDBC Driver Data Types' for the appropriate mappings.
Int getScale(int column)	Get the scale for the given column.
Boolean isAutoIncrement(int column)	Indicates whether the column represents an identity column.
Boolean next()	Advances to the next data row.

SQLServerBulkCSVFileRecord

A simple implementation of the ISQLServerBulkRecord interface that can be used to read in the basic Java data types from a delimited file where each line represents a row of data.

Implementation Notes and Limitations:

1. The maximum amount of data allowed in any given row is limited by the available memory because the data is read one line at a time.
2. Streaming of large data types such as varchar(max), varbinary(max), nvarchar(max), sqlxml, ntext isn't supported.
3. The delimiter specified for the CSV file shouldn't appear anywhere in the data and should be escaped properly if it is a restricted character in Java regular expressions.
4. In the CSV file implementation, double quotes are treated as part of the data. For example, the line hello,"world","hello,world" would be treated as having four columns with the values hello, "world", "hello and world" if the delimiter is a comma.
5. New line characters are used as row terminators and aren't allowed anywhere in the data.

CONSTRUCTOR	DESCRIPTION
SQLServerBulkCSVFileRecord(String fileToParse, String encoding, String delimiter, Boolean firstLineIsColumnNamesSQLServerBulkCSVFileRecord(String, String, String, boolean)	Initializes a new instance of the SQLServerBulkCSVFileRecord class that will parse each line in the fileToParse with the provided delimiter and encoding. If firstLineIsColumnNames is set to True, the first line in the file will be parsed as column names. If encoding is NULL, the default encoding will be used.

CONSTRUCTOR	DESCRIPTION
SQLServerBulkCSVFileRecord(String fileToParse, String encoding, Boolean firstLineIsColumnNamesSQLServerBulkCSVFileRecord(String, String, boolean)	Initializes a new instance of the SQLServerBulkCSVFileRecord class that will parse each line in the fileToParse with a comma as the delimiter and provided encoding. If firstLineIsColumnNames is set to True, the first line in the file will be parsed as column names. If encoding is NULL, the default encoding will be used.
SQLServerBulkCSVFileRecord(String fileToParse, Boolean firstLineIsColumnNamesSQLServerBulkCSVFileRecord(String, boolean)	Initializes a new instance of the SQLServerBulkCSVFileRecord class that will parse each line in the fileToParse with a comma as the delimiter and default encoding. If firstLineIsColumnNames is set to True, the first line in the file will be parsed as column names.

METHOD	DESCRIPTION
Void addColumnMetadata(int positionInFile, String columnName, int jdbcType, int precision, int scale)	Adds metadata for the given column in the file.
Void close()	Releases any resources associated with the file reader.
Void setTimestampWithTimezoneFormat(DateTim eFormatter dateTimeFormatter	Sets the format for parsing Timestamp data from the file as java.sql.Types.TIMESTAMP_WITH_TIMEZONE.
Void setTimestampWithTimezoneFormat(String dateTimeFormat)setTimeWithTimezoneFormat(DateTimeFormatter)	Sets the format for parsing Time data from the file as java.sql.Types.TIME_WITH_TIMEZONE.
Void setTimeWithTimezoneFormat(DateTimeForm atter dateTimeFormatter)	Sets the format for parsing Time data from the file as java.sql.Types.TIME_WITH_TIMEZONE.
Void setTimeWithTimezoneFormat(String timeFormat)	Sets the format for parsing Time data from the file as java.sql.Types.TIME_WITH_TIMEZONE.

See Also

Overview of the JDBC Driver

Using Bulk Copy API for Batch Insert Operation

8/2/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

Microsoft JDBC Driver 7.0 for SQL Server supports using Bulk Copy API for batch insert operations for Azure Data Warehouse. This feature allows users to enable driver to perform Bulk Copy operation underneath when executing batch insert operations. The driver aims to achieve improvement in performance while inserting the same data as the driver would have with regular batch insert operation. The driver parses the user's SQL Query, leveraging the Bulk Copy API in lieu of the usual batch insert operation. Below are various ways to enable the Bulk Copy API for batch insert feature, as well as the list of its limitations. This page also contains a small sample code that demonstrates a usage and the performance increase as well.

This feature is only applicable to PreparedStatement and CallableStatement's `executeBatch()` & `executeLargeBatch()` APIs.

Pre-Requisites

There are two prerequisites to enable Bulk Copy API for batch insert.

- The server must be Azure Data Warehouse.
- The query must be an insert query (the query may contain comments, but the query must start with the INSERT keyword for this feature to come into effect).

Enabling Bulk Copy API for batch insert

There are three ways to enable Bulk Copy API for batch insert.

1. Enabling with connection property

Adding **useBulkCopyForBatchInsert=true;** to the connection string enables this feature.

```
Connection connection = DriverManager.getConnection("jdbc:sqlserver://<server>:<port>;userName=
<user>;password=<password>;database=<database>;useBulkCopyForBatchInsert=true;");
```

2. Enabling with setUseBulkCopyForBatchInsert() method from SQLServerConnection object

Calling **SQLServerConnection.setUseBulkCopyForBatchInsert(true)** enables this feature.

SQLServerConnection.getUseBulkCopyForBatchInsert() retrieves the current value for **useBulkCopyForBatchInsert** connection property.

The value for **useBulkCopyForBatchInsert** stays constant for each PreparedStatement at the time of its initialization. Any subsequent calls to **SQLServerConnection.setUseBulkCopyForBatchInsert()** will not affect the already created PreparedStatement with regard to its value.

3. Enabling with setUseBulkCopyForBatchInsert() method from SQLServerDataSource object

Similar to above, but using SQLServerDataSource to create a SQLServerConnection object. Both methods achieve the same result.

Known limitations

There are currently these limitations that apply to this feature.

- Insert queries that contain non-parameterized values (for example, `INSERT INTO TABLE VALUES (?, 2`)), are not supported. Wildcards (?) are the only supported parameters for this function.
- Insert queries that contain INSERT-SELECT expressions (for example, `INSERT INTO TABLE SELECT * FROM TABLE2`), are not supported.
- Insert queries that contain multiple VALUE expressions (for example, `INSERT INTO TABLE VALUES (1, 2) (3, 4)`), are not supported.
- Insert queries that are followed by the OPTION clause, joined with multiple tables, or followed by another query, are not supported.
- Due to the limitations of Bulk Copy API, `DATETIME` , `SMALLDATETIME`, `GEOMETRY` , and `GEOGRAPHY` data types, are not supported for this feature.

If the query fails because of non "SQL server" related errors, the driver will log the error message and fallback to the original logic for batch insert.

Example

Below is an example code that demonstrates the use case for a batch insert operation against Azure DW of a thousand rows, for both (regular vs Bulk Copy API) scenarios.

```
    public static void main(String[] args) throws Exception
    {
        String tableName = "batchTest";
        String tableNameBulkCopyAPI = "batchTestBulk";

        String azureDWconnectionUrl = "jdbc:sqlserver://<server>:<port>;databaseName=<database>;user=
<user>;password=<password>";

        try (Connection con = DriverManager.getConnection(azureDWconnectionUrl); // connects to an Azure Data
Warehouse.
                Statement stmt = con.createStatement();
                PreparedStatement pstmt = con.prepareStatement("insert into " + tableName + " values (?,
?)");) {

            String dropSql = "if exists (select * from dbo.sysobjects where id = object_id(N'[dbo].[" +
tableName + "]') and OBJECTPROPERTY(id, N'IsUserTable') = 1) DROP TABLE [" + tableName + "]";
            stmt.execute(dropSql);

            String createSql = "create table " + tableName + " (c1 int, c2 varchar(20))";
            stmt.execute(createSql);

            System.out.println("Starting batch operation using regular batch insert operation.");
            long start = System.currentTimeMillis();
            for (int i = 0; i < 1000; i++) {
                pstmt.setInt(1, i);
                pstmt.setString(2, "test" + i);
                pstmt.addBatch();
            }
            pstmt.executeBatch();

            long end = System.currentTimeMillis();

            System.out.println("Finished. Time taken : " + (end - start) + " milliseconds.");
        }

        try (Connection con = DriverManager.getConnection(azureDWconnectionUrl +
";useBulkCopyForBatchInsert=true"); // connects to an Azure Data Warehouse, with useBulkCopyForBatchInsert
connection property set to true.
                Statement stmt = con.createStatement();
                PreparedStatement pstmt = con.prepareStatement("insert into " + tableNameBulkCopyAPI + "
values (?, ?)");) {

            String dropSql = "if exists (select * from dbo.sysobjects where id = object_id(N'[dbo].[" +
tableNameBulkCopyAPI + "]') and OBJECTPROPERTY(id, N'IsUserTable') = 1) DROP TABLE [" + tableNameBulkCopyAPI +
"]";
            stmt.execute(dropSql);

            String createSql = "create table " + tableNameBulkCopyAPI + " (c1 int, c2 varchar(20))";
            stmt.execute(createSql);

            System.out.println("Starting batch operation using Bulk Copy API.");
            long start = System.currentTimeMillis();
            for (int i = 0; i < 1000; i++) {
                pstmt.setInt(1, i);
                pstmt.setString(2, "test" + i);
                pstmt.addBatch();
            }
            pstmt.executeBatch();

            long end = System.currentTimeMillis();

            System.out.println("Finished. Time taken : " + (end - start) + " milliseconds.");
        }
    }
```

Result:

```
Starting batch operation using regular batch insert operation.
Finished. Time taken : 104132 milliseconds.
Starting batch operation using Bulk Copy API.
Finished. Time taken : 1058 milliseconds.
```

See Also

Improving Performance and Reliability with the JDBC Driver

Using a Stored Procedure with Output Parameters

8/13/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

A SQL Server stored procedure that you can call is one that returns one or more OUT parameters, which are parameters that the stored procedure uses to return data back to the calling application. The Microsoft JDBC Driver for SQL Server provides the SQLServerCallableStatement class, which you can use to call this kind of stored procedure and process the data that it returns.

When you call this kind of stored procedure by using the JDBC driver, you must use the `call` SQL escape sequence together with the prepareCall method of the SQLServerConnection class. The syntax for the `call` escape sequence with OUT parameters is the following:

`{call procedure-name[([parameter][,[parameter]]...)]}`

> **NOTE**
> For more information about the SQL escape sequences, see Using SQL Escape Sequences.

When you construct the `call` escape sequence, specify the OUT parameters by using the ? (question mark) character. This character acts as a placeholder for the parameter values that will be returned from the stored procedure. To specify a value for an OUT parameter, you must specify the data type of each parameter by using the registerOutParameter method of the SQLServerCallableStatement class before you run the stored procedure.

The value that you specify for the OUT parameter in the registerOutParameter method must be one of the JDBC data types contained in java.sql.Types, which in turn maps to one of the native SQL Server data types. For more information about the JDBC and SQL Server data types, see Understanding the JDBC Driver Data Types.

When you pass a value to the registerOutParameter method for an OUT parameter, you must specify not only the data type to be used for the parameter, but also the parameter's ordinal placement or the parameter's name in the stored procedure. For example, if your stored procedure contains a single OUT parameter, its ordinal value will be 1; if the stored procedure contains two parameters, the first ordinal value will be 1, and the second ordinal value will be 2.

> **NOTE**
> The JDBC driver does not support the use of CURSOR, SQLVARIANT, TABLE, and TIMESTAMP SQL Server data types as OUT parameters.

As an example, create the following stored procedure in the AdventureWorks sample database:

```
CREATE PROCEDURE GetImmediateManager
   @employeeID INT,
   @managerID INT OUTPUT
AS
BEGIN
   SELECT @managerID = ManagerID
   FROM HumanResources.Employee
   WHERE EmployeeID = @employeeID
END
```

This stored procedure returns a single OUT parameter (managerID), which is an integer, based on the specified IN parameter (employeeID), which is also an integer. The value that is returned in the OUT parameter is the ManagerID based on the EmployeeID that is contained in the HumanResources.Employee table.

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, and the execute method is used to call the GetImmediateManager stored procedure:

```
public static void executeStoredProcedure(Connection con) throws SQLException {
    try(CallableStatement cstmt = con.prepareCall("{call dbo.GetImmediateManager(?, ?)}");) {
        cstmt.setInt(1, 5);
        cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
        cstmt.execute();
        System.out.println("MANAGER ID: " + cstmt.getInt(2));
    }
}
```

This example uses the ordinal positions to identify the parameters. Alternatively, you can identify a parameter by using its name instead of its ordinal position. The following code example modifies the previous example to demonstrate how to use named parameters in a Java application. Note that parameter names correspond to the parameter names in the stored procedure's definition:

```
public static void executeStoredProcedure(Connection con) throws SQLException {
    try(CallableStatement cstmt = con.prepareCall("{call dbo.GetImmediateManager(?, ?)}"); ) {
        cstmt.setInt("employeeID", 5);
        cstmt.registerOutParameter("managerID", java.sql.Types.INTEGER);
        cstmt.execute();
        System.out.println("MANAGER ID: " + cstmt.getInt("managerID"));
    }
}
```

NOTE

These examples use the execute method of the SQLServerCallableStatement class to run the stored procedure. This is used because the stored procedure did not also return a result set. If it did, the executeQuery method would be used.

Stored procedures can return update counts and multiple result sets. The Microsoft JDBC Driver for SQL Server follows the JDBC 3.0 specification, which states that multiple result sets and update counts should be retrieved before the OUT parameters are retrieved. That is, the application should retrieve all of the ResultSet objects and update counts before retrieving the OUT parameters by using the CallableStatement.getter methods. Otherwise, the ResultSet objects and update counts that haven't already been retrieved will be lost when the OUT parameters are retrieved. For more information about update counts and multiple result sets, see Using a Stored Procedure with an Update Count and Using Multiple Result Sets.

See Also

Using Statements with Stored Procedures

Performing Batch Operations

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

To improve performance when multiple updates to a SQL Server database are occurring, the Microsoft JDBC Driver for SQL Server provides the ability to submit multiple updates as a single unit of work, also referred to as a batch.

The SQLServerStatement, SQLServerPreparedStatement, and SQLServerCallableStatement classes can all be used to submit batch updates. The addBatch method is used to add a command. The clearBatch method is used to clear the list of commands. The executeBatch method is used to submit all commands for processing. Only Data Definition Language (DDL) and Data Manipulation Language (DML) statements that return a simple update count can be run as part of a batch.

The executeBatch method returns an array of **int** values that correspond to the update count of each command. If one of the commands fails, a BatchUpdateException is thrown, and you should use the getUpdateCounts method of the BatchUpdateException class to retrieve the update count array. If a command fails, the driver continues processing the remaining commands. However, if a command has a syntax error, the statements in the batch fail.

NOTE

If you do not have to use update counts, you can first issue a SET NOCOUNT ON statement to SQL Server. This will reduce network traffic and additionally enhance the performance of your application.

As an example, create the following table in the AdventureWorks sample database:

```
CREATE TABLE TestTable
   (Col1 int IDENTITY,
    Col2 varchar(50),
    Col3 int);
```

In the following example, an open connection to the AdventureWorks sample database is passed in to the function, the addBatch method is used to create the statements to be executed, and the executeBatch method is called to submit the batch to the database.

```
public static void executeBatchUpdate(Connection con) {
   try {
      Statement stmt = con.createStatement();
      stmt.addBatch("INSERT INTO TestTable (Col2, Col3) VALUES ('X', 100)");
      stmt.addBatch("INSERT INTO TestTable (Col2, Col3) VALUES ('Y', 200)");
      stmt.addBatch("INSERT INTO TestTable (Col2, Col3) VALUES ('Z', 300)");
      int[] updateCounts = stmt.executeBatch();
      stmt.close();
   }
   catch (Exception e) {
      e.printStackTrace();
   }
}
```

See Also

User Defined Types

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

User-defined types (UDTs) were introduced in SQL Server 2005 (9.x) to allow a developer to extend the server's scalar type system by storing common language runtime (CLR) objects in a SQL Server database. UDTs can contain multiple elements and can have behaviors, unlike the traditional alias data types, that consist of a single SQL Server system data type. Previously, UDTs were restricted to a maximum size of 8 kilobytes. In SQL Server 2008, support was added for UDTs larger than 64 kilobytes. Beginning in version 3.0, the JDBC Driver also supports UDTs larger than 64 kilobytes when you specify the UserDefined format.

There is no behavior change for UDTs that are less than or equal to 8,000 bytes, but larger UDTs are supported and report their size as "unlimited".

See Also

Understanding the JDBC Driver Data Types

Configuring How java.sql.Time Values are Sent to the Server

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

If you use a java.sql.Time object or the java.sql.Types.TIME JDBC type to set a parameter, you can configure how the java.sql.Time value is sent to the server; either as a SQL Server **time** type or as a **datetime** type.

This scenario applies when using one of the following methods:

- SQLServerCallableStatement.registerOutParameter(int, int)
- SQLServerCallableStatement.registerOutParameter(int, int, int)
- SQLServerCallableStatement.setTime
- SQLServerPreparedStatement.setTime
- SQLServerCallableStatement.setObject
- SQLServerPreparedStatement.setObject

  You can configure how the java.sql.Time value is sent by using the **sendTimeAsDatetime** connection property. For more information, see Setting the Connection Properties.

  You can programmatically modify the value of the **sendTimeAsDatetime** connection property with SQLServerDataSource.setSendTimeAsDatetime.

  Versions of SQL Server earlier than SQL Server 2008 don't support the **time** data type, so applications using java.sql.Time typically store java.sql.Time values either as **datetime** or **smalldatetime** SQL Server data types.

  If you want to use the **datetime** and **smalldatetime** SQL Server data types when working with java.sql.Time values, you should set the **sendTimeAsDatetime** connection property to **true**. If you want to use the **time** SQL Server data type when working with java.sql.Time values, you should set the **sendTimeAsDatetime** connection property to **false**.

  Be aware that sending java.sql.Time values into a parameter whose data type can also store the date, that default dates are different depending on whether the java.sql.Time value is sent as a **datetime** (1/1/1970) or **time** (1/1/1900) value. For more information about data conversions when sending data to a SQL Server, see Using Date and Time Data.

  In SQL Server JDBC Driver 3.0, **sendTimeAsDatetime** is true by default. In a future release, the **sendTimeAsDatetime** connection property may be set to false by default.

  To ensure that your application continues to work as expected regardless of the default value of the **sendTimeAsDatetime** connection property, you can:

- Use java.sql.Time when working with the **time** SQL Server data type.
- Use java.sql.Timestamp when working with the **datetime**, **smalldatetime**, and **datetime2** SQL Server data types.

SendTimeAsDatetime must be false for encrypted columns as encrypted columns don't support the conversion from time to datetime. Beginning with Microsoft JDBC Driver 6.0 for SQL Server, the SQLServerConnection class has the following two methods to set/get the value of the sendTimeAsDatetime property.

```
public boolean getSendTimeAsDatetime()
public void setSendTimeAsDatetime(boolean sendTimeAsDateTimeValue)
```

See Also

Understanding the JDBC Driver Data Types

Using SQL Escape Sequences

8/8/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

The Microsoft JDBC Driver for SQL Server supports the use of SQL escape sequences, as defined by the JDBC API. Escape sequences are used within an SQL statement to tell the driver that the escaped part of the SQL string should be handled differently. When the JDBC driver processes the escaped part of an SQL string, it translates that part of the string into SQL code that SQL Server understands.

There are five types of escape sequences that the JDBC API requires, and all are supported by the JDBC driver:

- LIKE wildcard literals
- Function handling
- Date and time literals
- Stored procedure calls
- Outer joins
- Limit escape syntax

The escape sequence syntax used by the JDBC driver is the following:

`{keyword ...parameters...}`

NOTE
SQL escape processing is always turned on for the JDBC driver.

The following sections describe the five types of escape sequences and how they are supported by the JDBC driver.

LIKE Wildcard Literals

The JDBC driver supports the `{escape 'escape character'}` syntax for using LIKE clause wildcards as literals. For example, the following code will return values for col3, where the value of col2 literally begins with an underscore (and not its wildcard usage).

```
ResultSet rst = stmt.executeQuery("SELECT col3 FROM test1 WHERE col2
LIKE '\\_%' {escape '\\'}");
```

NOTE
The escape sequence must be at the end of the SQL statement. For multiple SQL statements in a command string, the escape sequence needs to be at the end of each relevant SQL statement.

Function Handling

The JDBC driver supports function escape sequences in SQL statements with the following syntax:

```
{fn functionName}
```

where `functionName` is a function supported by the JDBC driver. For example:

```
SELECT {fn UCASE(Name)} FROM Employee
```

The following table lists the various functions that are supported by the JDBC driver when using a function escape sequence:

STRING FUNCTIONS	NUMERIC FUNCTIONS	DATETIME FUNCTIONS	SYSTEM FUNCTIONS
ASCII	ABS	CURDATE	DATABASE
CHAR	ACOS	CURTIME	IFNULL
CONCAT	ASIN	DAYNAME	USER
DIFFERENCE	ATAN	DAYOFMONTH	
INSERT	ATAN2	DAYOFWEEK	
LCASE	CEILING	DAYOFYEAR	
LEFT	COS	EXTRACT	
LENGTH	COT	HOUR	
LOCATE	DEGREES	MINUTE	
LTRIM	EXP	MONTH	
REPEAT	FLOOR	MONTHNAME	
REPLACE	LOG	NOW	
RIGHT	LOG10	QUARTER	
RTRIM	MOD	SECOND	
SOUNDEX	PI	TIMESTAMPADD	
SPACE	POWER	TIMESTAMPDIFF	
SUBSTRING	RADIANS	WEEK	
UCASE	RAND	YEAR	
	ROUND		
	SIGN		
	SIN		
	SQRT		
	TAN		
	TRUNCATE		

NOTE

If you try to use a function that the database does not support, an error will occur.

Date and Time Literals

The escape syntax for date, time, and timestamp literals is the following:

```
{literal-type 'value'}
```

where `literal-type` is one of the following:

LITERAL TYPE	DESCRIPTION	VALUE FORMAT
d	Date	yyyy-mm-dd
t	Time	hh:mm:ss [1]
ts	TimeStamp	yyyy-mm-dd hh:mm:ss[.f...]

For example:

```
UPDATE Orders SET OpenDate={d '2005-01-31'}
WHERE OrderID=1025
```

Stored Procedure Calls

The JDBC driver supports the `{? = call proc_name(?,...)}` and `{call proc_name(?,...)}` escape syntax for stored procedure calls, depending on whether you need to process a return parameter.

A procedure is an executable object stored in the database. Generally, it is one or more SQL statements that have been precompiled. The escape sequence syntax for calling a stored procedure is the following:

```
{[?=]call procedure-name[([parameter][,[parameter]]...)]}
```

where `procedure-name` specifies the name of a stored procedure and `parameter` specifies a stored procedure parameter.

For more information about using the `call` escape sequence with stored procedures, see Using Statements with Stored Procedures.

Outer Joins

The JDBC driver supports the SQL92 left, right, and full outer join syntax. The escape sequence for outer joins is the following:

```
{oj outer-join}
```

where outer-join is:

```
table-reference {LEFT | RIGHT | FULL} OUTER JOIN
{table-reference | outer-join} ON search-condition
```

where `table-reference` is a table name and `search-condition` is the join condition you want to use for the tables.

For example:

```
SELECT Customers.CustID, Customers.Name, Orders.OrderID, Orders.Status
   FROM {oj Customers LEFT OUTER JOIN
      Orders ON Customers.CustID=Orders.CustID}
   WHERE Orders.Status='OPEN'
```

The following outer join escape sequences are supported by the JDBC driver:

- Left outer joins
- Right outer joins
- Full outer joins
- Nested outer joins

Limit Escape Syntax

NOTE

The LIMIT escape syntax is only supported by Microsoft JDBC Driver 4.2 (or higher) for SQL Server when using JDBC 4.1 or higher.

The escape syntax for LIMIT is as follows:

```
LIMIT <rows> [OFFSET <row offset>]
```

The escape syntax has two parts: <*rows*> is mandatory and specifies the number of rows to return. OFFSET and <*row offset*> are optional and specify the number of rows to skip before beginning to return rows. The JDBC driver supports only the mandatory part by transforming the query to use TOP instead of LIMIT. SQL Server does not support the LIMIT clause. **The JDBC driver does not support the optional <row offset> and the driver will throw an exception if it is used**.

See Also

Using Statements with the JDBC Driver

Using Statements with Stored Procedures

8/13/2018 • 2 minutes to read • Edit Online

Download JDBC Driver

A stored procedure is a database procedure, similar to a procedure in other programming languages, which is contained within the database itself. In SQL Server, stored procedures can be created by using Transact-SQL, or by using the common language runtime (CLR) and one of the Visual Studio programming languages such as Visual Basic or C#. Generally, SQL Server stored procedures can do the following:

- Accept input parameters and return multiple values in the form of output parameters to the calling procedure or batch.
- Contain programming statements that perform operations in the database, including calling other procedures.
- Return a status value to a calling procedure or batch to indicate success or failure (and the reason for failure).

NOTE

For more information about SQL Server stored procedures, see "Understanding Stored Procedures" in SQL Server Books Online.

To work with data in a SQL Server database by using a stored procedure, the Microsoft JDBC Driver for SQL Server provides the SQLServerStatement, SQLServerPreparedStatement, and SQLServerCallableStatement classes. Which class you use depends on whether IN (input) or OUT (output) parameters are required by the stored procedure. If the stored procedure requires no IN or OUT parameters, you can use the SQLServerStatement class; if the stored procedure will be called multiple times, or requires only IN parameters, you can use the SQLServerPreparedStatement class. If the stored procedure requires both IN and OUT parameters, you should use the SQLServerCallableStatement class. It is only when the stored procedure requires OUT parameters that you will need the overhead of using the SQLServerCallableStatement class.

NOTE

Stored procedures can also return update counts and multiple result sets. For more information, see Using a Stored Procedure with an Update Count and Using Multiple Result Sets.

When you use the JDBC driver to call a stored procedure with parameters, you must use the `call` SQL escape sequence together with the prepareCall method of the SQLServerConnection class. The complete syntax for the `call` escape sequence is as follows:

```
{[?=]call procedure-name[([parameter][,[parameter]]...)]}
```

NOTE

For more information about the `call` and other SQL escape sequences, see Using SQL Escape Sequences.

The topics in this section describe the ways that you can call SQL Server stored procedures by using the JDBC driver and the `call` SQL escape sequence.

In This Section

TOPIC	DESCRIPTION
Using a Stored Procedure with No Parameters	Describes how to use the JDBC driver to run stored procedures that contain no input or output parameters.
Using a Stored Procedure with Input Parameters	Describes how to use the JDBC driver to run stored procedures that contain input parameters.
Using a Stored Procedure with Output Parameters	Describes how to use the JDBC driver to run stored procedures that contain output parameters.
Using a Stored Procedure with a Return Status	Describes how to use the JDBC driver to run stored procedures that contain return status values.
Using a Stored Procedure with an Update Count	Describes how to use the JDBC driver to run stored procedures that return update counts.

See Also

Using Statements with the JDBC Driver

Azure Key Vault Sample Version 6.0.0

8/8/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

Sample application using Azure Key Vault feature

This application is runnable using JDBC Driver 6.0.0 and Azure-Keyvault (version 0.9.7), Adal4j (version 1.3.0), and their dependencies. The underlying dependencies can be resolved by adding these libraries to the pom file of the project as described here:

```
import java.net.URISyntaxException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

import com.microsoft.aad.adal4j.AuthenticationContext;
import com.microsoft.aad.adal4j.AuthenticationResult;
import com.microsoft.aad.adal4j.ClientCredential;

import com.microsoft.sqlserver.jdbc.SQLServerColumnEncryptionAzureKeyVaultProvider;
import com.microsoft.sqlserver.jdbc.SQLServerColumnEncryptionKeyStoreProvider;
import com.microsoft.sqlserver.jdbc.SQLServerConnection;
import com.microsoft.sqlserver.jdbc.SQLServerException;
import com.microsoft.sqlserver.jdbc.SQLServerKeyVaultAuthenticationCallback;

public class AKV_600 {

    static String connectionUrl =
"jdbc:sqlserver://localhost;integratedSecurity=true;database=test;columnEncryptionSetting=enabled";
    static String applicationClientID = "Your Client ID";
    static String applicationKey = "Your Application Key";
    static String keyID = "Your Key ID";
    static String cmkName = "AKV_CMK_JDBC";
    static String cekName = "AKV_CEK_JDBC";
    static String akvTable = "akvTable";

    static String createTableSQL = "create table " + akvTable + " ("
            + "PlainNvarcharMax nvarchar(max) null,"
            + "RandomizedNvarcharMax nvarchar(max) COLLATE Latin1_General_BIN2 ENCRYPTED WITH (ENCRYPTION_TYPE
= RANDOMIZED, ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256', COLUMN_ENCRYPTION_KEY = "
            + cekName + ") NULL,"
            + "DeterministicNvarcharMax nvarchar(max) COLLATE Latin1_General_BIN2 ENCRYPTED WITH
(ENCRYPTION_TYPE = DETERMINISTIC, ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256', COLUMN_ENCRYPTION_KEY = "
            + cekName + ") NULL" + ");";

    static ExecutorService service = Executors.newFixedThreadPool(2);

    private static SQLServerColumnEncryptionAzureKeyVaultProvider tryAuthenticationCallback()
            throws URISyntaxException, SQLServerException {
        SQLServerKeyVaultAuthenticationCallback authenticationCallback = new
SQLServerKeyVaultAuthenticationCallback() {
```

```
            @Override
            public String getAccessToken(String authority, String resource,
                    String scope) {
                AuthenticationResult result = null;
                try {
                    AuthenticationContext context = new AuthenticationContext(
                            authority, false, service);
                    ClientCredential cred = new ClientCredential(
                            applicationClientID, applicationKey);
                    Future<AuthenticationResult> future = context
                            .acquireToken(resource, cred, null);
                    result = future.get();
                } catch (Exception e) {
                    e.printStackTrace();
                }
                return result.getAccessToken();
            }
        };

        SQLServerColumnEncryptionAzureKeyVaultProvider akvProvider = new
SQLServerColumnEncryptionAzureKeyVaultProvider(
                authenticationCallback, service);
        return akvProvider;
    }

    public static void main(String[] args)
            throws ClassNotFoundException, Exception {
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        try (Connection connection = DriverManager.getConnection(connectionUrl);
                Statement statement = connection.createStatement()) {
            statement.execute("DBCC FREEPROCCACHE");
            SQLServerColumnEncryptionAzureKeyVaultProvider akvProvider = tryAuthenticationCallback();
            setupKeyStoreProviders(akvProvider.getName(), akvProvider);
            testAKV(akvProvider.getName(), akvProvider, connection, statement);
        }
    }

    private static void testAKV(String CUSTOM_AKV_PROVIDER_NAME,
            SQLServerColumnEncryptionKeyStoreProvider akvProvider,
            Connection connection, Statement statement)
            throws SQLException, SQLServerException, InterruptedException {

        dropTable(statement);
        dropKeys(statement);

        System.out.println("createCMK");
        createCMK(CUSTOM_AKV_PROVIDER_NAME, statement);

        System.out.println("createCEK");
        createCEK(akvProvider, statement);

        System.out.println("create Table");
        statement.execute(createTableSQL);

        System.out.println("populate");
        populateCharNormalCase(connection);

        System.out.println("run the test");
        testChar(statement);
    }

    private static void setupKeyStoreProviders(String CUSTOM_AKV_PROVIDER_NAME,
            SQLServerColumnEncryptionKeyStoreProvider akvProvider)
            throws SQLServerException {
        Map<String, SQLServerColumnEncryptionKeyStoreProvider> map1 = new HashMap<String,
SQLServerColumnEncryptionKeyStoreProvider>();
        map1.put(CUSTOM_AKV_PROVIDER_NAME, akvProvider);
        SQLServerConnection.registerColumnEncryptionKeyStoreProviders(map1);
    }

    private static void dropTable(Statement statement) throws SQLException {
        statement.executeUpdate("if object_id('" + akvTable
                + "','U') is not null" + " drop table " + akvTable);
    }

    private static void dropKeys(Statement statement) throws SQLException {
        statement.executeUpdate(
                "if exists (SELECT name from sys.column_encryption_keys where name='"
                        + cekName + "')" + " begin"
                        + " drop column encryption key " + cekName + " end");
        statement.executeUpdate(
                "if exists (SELECT name from sys.column_master_keys where name='"
                        + cmkName + "')" + " begin" + " drop column master key "
                        + cmkName + " end");
    }

    private static void createCMK(String CUSTOM_AKV_PROVIDER_NAME,
            Statement statement) throws SQLException {
        String _createColumnMasterKeyTemplate = String.format(
                "CREATE COLUMN MASTER KEY [%s] WITH ( KEY_STORE_PROVIDER_NAME = '%s', KEY_PATH = '%s');",
                cmkName, CUSTOM_AKV_PROVIDER_NAME, keyID);
        statement.execute(_createColumnMasterKeyTemplate);
    }

    private static void createCEK(
            SQLServerColumnEncryptionKeyStoreProvider storeProvider,
            Statement statement) throws SQLServerException, SQLException {
        String letters = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
        byte[] valuesDefault = letters.getBytes();
        byte[] key = storeProvider.encryptColumnEncryptionKey(keyID, "RSA_OAEP",
                valuesDefault);
        String cekSql = "CREATE COLUMN ENCRYPTION KEY " + cekName
                + " WITH VALUES " + "(COLUMN_MASTER_KEY = " + cmkName
                + ", ALGORITHM = 'RSA_OAEP', ENCRYPTED_VALUE = 0x"
                + bytesToHexString(key, key.length) + ")" + ";";
        statement.execute(cekSql);
    }

    final static char[] hexChars = {'0', '1', '2', '3', '4', '5', '6', '7', '8',
            '9', 'A', 'B', 'C', 'D', 'E', 'F'};

    private static String bytesToHexString(byte[] b, int length) {
        StringBuilder sb = new StringBuilder(length * 2);
        for (int i = 0; i < length; i++) {
            int hexVal = b[i] & 0xFF;
            sb.append(hexChars[(hexVal & 0xF0) >> 4]);
            sb.append(hexChars[(hexVal & 0x0F)]);
        }
        return sb.toString();
    }

    private static void populateCharNormalCase(Connection connection)
            throws SQLException {
        String sql = "insert into " + akvTable + " values(?,?,?)";
        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            for (int i = 1; i <= 3; i++) {
                pstmt.setNString(i, "hello world");
            }
            pstmt.execute();
        }
    }

    private static void testChar(Statement statement) throws SQLException {
        try (ResultSet rs = statement
                .executeQuery("select * from " + akvTable);) {
            int numberOfColumns = rs.getMetaData().getColumnCount();
            while (rs.next()) {
                testGetString(rs, numberOfColumns);
            }
        }
    }

    private static void testGetString(ResultSet rs, int numberOfColumns)
            throws SQLException {
        for (int i = 1; i <= numberOfColumns; i = i + 3) {
            String stringValue1 = "" + rs.getString(i);
            String stringValue2 = "" + rs.getString(i + 1);
            String stringValue3 = "" + rs.getString(i + 2);
            System.out.println(stringValue1);
            System.out.println(stringValue2);
            System.out.println(stringValue3);
        }
    }
}
```

See Also

Azure Key Vault Sample Version 7.0.0
Azure Key Vault Sample Version 6.2.2

Azure Key Vault Sample Version 6.2.2

8/8/2018 • 3 minutes to read • Edit Online

Download JDBC Driver

Sample application using Azure Key Vault feature

This application is runnable using JDBC Driver 6.2.2 and 6.4.0 and Azure-Keyvault (version 1.0.0), Adal4j (version 1.4.0), and their dependencies. The underlying dependencies can be resolved by adding these libraries to the pom file of the project as described here:

```
import java.net.URISyntaxException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.HashMap;
import java.util.Map;
import com.microsoft.sqlserver.jdbc.SQLServerColumnEncryptionAzureKeyVaultProvider;
import com.microsoft.sqlserver.jdbc.SQLServerColumnEncryptionKeyStoreProvider;
import com.microsoft.sqlserver.jdbc.SQLServerConnection;
import com.microsoft.sqlserver.jdbc.SQLServerException;

public class AKV_6_2_2 {

    static String connectionUrl =
"jdbc:sqlserver://localhost;integratedSecurity=true;database=test;columnEncryptionSetting=enabled";
    static String applicationClientID = "Your Client ID";
    static String applicationKey = "Your Application Key";
    static String keyID = "Your Key ID";
    static String cmkName = "AKV_CMK_JDBC";
    static String cekName = "AKV_CEK_JDBC";
    static String akvTable = "akvTable";

    static String createTableSQL = "create table " + akvTable + " ("
            + "PlainNvarcharMax nvarchar(max) null,"
            + "RandomizedNvarcharMax nvarchar(max) COLLATE Latin1_General_BIN2 ENCRYPTED WITH (ENCRYPTION_TYPE
= RANDOMIZED, ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256', COLUMN_ENCRYPTION_KEY = "
            + cekName + ") NULL,"
            + "DeterministicNvarcharMax nvarchar(max) COLLATE Latin1_General_BIN2 ENCRYPTED WITH
(ENCRYPTION_TYPE = DETERMINISTIC, ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256', COLUMN_ENCRYPTION_KEY = "
            + cekName + ") NULL" + ");";

    public static void main(String[] args)
            throws ClassNotFoundException, Exception {
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        try (Connection connection = DriverManager.getConnection(connectionUrl);
                Statement statement = connection.createStatement()) {
            statement.execute("DBCC FREEPROCCACHE");
            SQLServerColumnEncryptionAzureKeyVaultProvider akvProvider = new
SQLServerColumnEncryptionAzureKeyVaultProvider(
                    applicationClientID, applicationKey);
            setupKeyStoreProviders(akvProvider.getName(), akvProvider);
            testAKV(akvProvider.getName(), akvProvider, connection, statement);
        }
    }

    private static void testAKV(String CUSTOM_AKV_PROVIDER_NAME,
            SQLServerColumnEncryptionKeyStoreProvider akvProvider,
            Connection connection, Statement statement)
            throws SQLException, SQLServerException, InterruptedException {

        dropTable(statement);
        dropKeys(statement);

        System.out.println("createCMK");
        createCMK(CUSTOM_AKV_PROVIDER_NAME, statement);

        System.out.println("createCEK");
        createCEK(akvProvider, statement);

        System.out.println("create Table");
        statement.execute(createTableSQL);

        System.out.println("populate");
        populateCharNormalCase(connection);

        System.out.println("run the test");
        testChar(statement);
    }

    private static void setupKeyStoreProviders(String CUSTOM_AKV_PROVIDER_NAME,
            SQLServerColumnEncryptionKeyStoreProvider akvProvider)
            throws SQLServerException {
        Map<String, SQLServerColumnEncryptionKeyStoreProvider> map1 = new HashMap<String,
SQLServerColumnEncryptionKeyStoreProvider>();
        map1.put(CUSTOM_AKV_PROVIDER_NAME, akvProvider);
        SQLServerConnection.registerColumnEncryptionKeyStoreProviders(map1);
    }

    private static void dropTable(Statement statement) throws SQLException {
        statement.executeUpdate("if object_id('" + akvTable
                + "','U') is not null" + " drop table " + akvTable);
    }

    private static void dropKeys(Statement statement) throws SQLException {
        statement.executeUpdate(
                "if exists (SELECT name from sys.column_encryption_keys where name='"
                        + cekName + "')" + " begin"
                        + " drop column encryption key " + cekName + " end");
        statement.executeUpdate(
                "if exists (SELECT name from sys.column_master_keys where name='"
                        + cmkName + "')" + " begin" + " drop column master key "
                        + cmkName + " end");
    }

    private static void createCMK(String CUSTOM_AKV_PROVIDER_NAME,
            Statement statement) throws SQLException {
        String _createColumnMasterKeyTemplate = String.format(
                "CREATE COLUMN MASTER KEY [%s] WITH ( KEY_STORE_PROVIDER_NAME = '%s', KEY_PATH = '%s');",
                cmkName, CUSTOM_AKV_PROVIDER_NAME, keyID);
        statement.execute(_createColumnMasterKeyTemplate);
    }

    private static void createCEK(
            SQLServerColumnEncryptionKeyStoreProvider storeProvider,
            Statement statement) throws SQLServerException, SQLException {
        String letters = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
        byte[] valuesDefault = letters.getBytes();
        byte[] key = storeProvider.encryptColumnEncryptionKey(keyID, "RSA_OAEP",
                valuesDefault);
        String cekSql = "CREATE COLUMN ENCRYPTION KEY " + cekName
                + " WITH VALUES " + "(COLUMN_MASTER_KEY = " + cmkName
                + ", ALGORITHM = 'RSA_OAEP', ENCRYPTED_VALUE = 0x"
                + bytesToHexString(key, key.length) + ")" + ";";
        statement.execute(cekSql);
    }
    final static char[] hexChars = {'0', '1', '2', '3', '4', '5', '6', '7', '8',
            '9', 'A', 'B', 'C', 'D', 'E', 'F'};

    private static String bytesToHexString(byte[] b, int length) {
        StringBuilder sb = new StringBuilder(length * 2);
        for (int i = 0; i < length; i++) {
            int hexVal = b[i] & 0xFF;
            sb.append(hexChars[(hexVal & 0xF0) >> 4]);
            sb.append(hexChars[(hexVal & 0x0F)]);
        }
        return sb.toString();
    }

    private static void populateCharNormalCase(Connection connection)
            throws SQLException {
        String sql = "insert into " + akvTable + " values(?,?,?)";
        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            for (int i = 1; i <= 3; i++) {
                pstmt.setNString(i, "hello world");
            }
            pstmt.execute();
        }
    }

    private static void testChar(Statement statement) throws SQLException {
        try (ResultSet rs = statement
                .executeQuery("select * from " + akvTable);) {
            int numberOfColumns = rs.getMetaData().getColumnCount();
            while (rs.next()) {
                testGetString(rs, numberOfColumns);
            }
        }
    }

    private static void testGetString(ResultSet rs, int numberOfColumns)
            throws SQLException {
        for (int i = 1; i <= numberOfColumns; i = i + 3) {
            String stringValue1 = "" + rs.getString(i);
            String stringValue2 = "" + rs.getString(i + 1);
            String stringValue3 = "" + rs.getString(i + 2);
            System.out.println(stringValue1);
            System.out.println(stringValue2);
            System.out.println(stringValue3);
        }
    }
}
```

See Also

Azure Key Vault Sample Version 7.0.0

Azure Key Vault Sample Version 6.0.0

Azure Key Vault Sample Version 7.0.0

8/8/2018 • 4 minutes to read • Edit Online

Download JDBC Driver

Sample application using Azure Key Vault feature

This application is runnable using JDBC Driver 7.0 and above and Azure-Keyvault (version 1.0.0), Adal4j (version 1.6.0), and their dependencies. The underlying dependencies can be resolved by adding these libraries to the pom file of the project as described here:

```
import java.net.URISyntaxException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

import com.microsoft.aad.adal4j.AuthenticationContext;
import com.microsoft.aad.adal4j.AuthenticationResult;
import com.microsoft.aad.adal4j.ClientCredential;
import com.microsoft.sqlserver.jdbc.SQLServerColumnEncryptionAzureKeyVaultProvider;
import com.microsoft.sqlserver.jdbc.SQLServerColumnEncryptionKeyStoreProvider;
import com.microsoft.sqlserver.jdbc.SQLServerConnection;
import com.microsoft.sqlserver.jdbc.SQLServerException;
import com.microsoft.sqlserver.jdbc.SQLServerKeyVaultAuthenticationCallback;

public class AKV_7_0_0 {

    static String connectionUrl =
"jdbc:sqlserver://localhost;integratedSecurity=true;database=test;columnEncryptionSetting=enabled";
    static String applicationClientID = "Your Client ID";
    static String applicationKey = "Your Application Key";
    static String keyID = "Your Key ID";
    static String cmkName = "AKV_CMK_JDBC";
    static String cekName = "AKV_CEK_JDBC";
    static String akvTable = "akvTable";

    static String createTableSQL = "create table " + akvTable + " ("
            + "PlainNvarcharMax nvarchar(max) null,"
            + "RandomizedNvarcharMax nvarchar(max) COLLATE Latin1_General_BIN2 ENCRYPTED WITH (ENCRYPTION_TYPE
= RANDOMIZED, ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256', COLUMN_ENCRYPTION_KEY = "
            + cekName + ") NULL,"
            + "DeterministicNvarcharMax nvarchar(max) COLLATE Latin1_General_BIN2 ENCRYPTED WITH
(ENCRYPTION_TYPE = DETERMINISTIC, ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256', COLUMN_ENCRYPTION_KEY = "
            + cekName + ") NULL" + ");";

    public static void main(String[] args)
            throws ClassNotFoundException, Exception {
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        try (Connection connection = DriverManager.getConnection(connectionUrl);
                Statement statement = connection.createStatement()) {
            statement.execute("DBCC FREEPROCCACHE");
            System.out.println("Create SQLServerColumnEncryptionAzureKeyVaultProvider with
'authenticationCallback' and 'executorService(null)'");
            /* Constructor added in 6.0.0 driver version and removed in 6.2.2 driver, now added back in 7.0.0
driver
             * [Supports SQLServerKeyVaultAuthenticationCallback in 7.0 for backwards compatibility]
             * This constructor is marked @deprecated since it no longer uses 'ExecutorService' parameter
passed. */
            @SuppressWarnings("deprecation")
            SQLServerColumnEncryptionAzureKeyVaultProvider akvProvider1 = new
SQLServerColumnEncryptionAzureKeyVaultProvider(
                    tryAuthenticationCallback(), null);
            setupKeyStoreProviders(akvProvider1.getName(), akvProvider1);
            testAKV(akvProvider1.getName(), akvProvider1, connection, statement);

            statement.execute("DBCC FREEPROCCACHE");
            System.out.println("Create SQLServerColumnEncryptionAzureKeyVaultProvider with
'authenticationCallback'");
            /* Constructor added in 7.0.0 driver version [Supports SQLServerKeyVaultAuthenticationCallback in
7.0 for backwards compatibility]
             * This constructor is recommended to replace the above deprecated constructor */
            SQLServerColumnEncryptionAzureKeyVaultProvider akvProvider2 = new
SQLServerColumnEncryptionAzureKeyVaultProvider(
                    tryAuthenticationCallback());
            testAKV(akvProvider2.getName(), akvProvider2, connection, statement);

            statement.execute("DBCC FREEPROCCACHE");
            System.out.println("Create SQLServerColumnEncryptionAzureKeyVaultProvider with 'clientId' and
'clientKey'");
            /* Constructor added in 6.2.2 driver version [Continued Support] */
            SQLServerColumnEncryptionAzureKeyVaultProvider akvProvider3 = new
SQLServerColumnEncryptionAzureKeyVaultProvider(
                    applicationClientID, applicationKey);
            testAKV(akvProvider3.getName(), akvProvider3, connection, statement);
            System.exit(0);
        }
    }

    private static SQLServerKeyVaultAuthenticationCallback tryAuthenticationCallback()
            throws URISyntaxException, SQLServerException {
        SQLServerKeyVaultAuthenticationCallback authenticationCallback = new
SQLServerKeyVaultAuthenticationCallback() {

            @Override
            public String getAccessToken(String authority, String resource,
                    String scope) {
                AuthenticationResult result = null;
                try {
                    ExecutorService service = Executors.newFixedThreadPool(1);
                    AuthenticationContext context = new AuthenticationContext(
                            authority, false, service);
                    ClientCredential cred = new ClientCredential(
                            applicationClientID, applicationKey);
                    Future<AuthenticationResult> future = context
                            .acquireToken(resource, cred, null);
                    result = future.get();
                    service.shutdown();
                } catch (Exception e) {
                    e.printStackTrace();
                }
                return result.getAccessToken();
            }
        };

        return authenticationCallback;

    }

    private static void testAKV(String CUSTOM_AKV_PROVIDER_NAME,
            SQLServerColumnEncryptionKeyStoreProvider akvProvider,
            Connection connection, Statement statement)
            throws SQLException, InterruptedException {

        dropTable(statement);
        dropKeys(statement);

        System.out.println("createCMK");
        createCMK(CUSTOM_AKV_PROVIDER_NAME, statement);

        System.out.println("createCEK");
        createCEK(akvProvider, statement);

        System.out.println("create Table");
        statement.execute(createTableSQL);

        System.out.println("populate");
        populateCharNormalCase(connection);

        System.out.println("run the test");
        testChar(statement);
    }

    private static void setupKeyStoreProviders(String CUSTOM_AKV_PROVIDER_NAME,
            SQLServerColumnEncryptionKeyStoreProvider akvProvider)
            throws SQLServerException {
        Map<String, SQLServerColumnEncryptionKeyStoreProvider> map1 = new HashMap<String,
SQLServerColumnEncryptionKeyStoreProvider>();
        map1.put(CUSTOM_AKV_PROVIDER_NAME, akvProvider);
        SQLServerConnection.registerColumnEncryptionKeyStoreProviders(map1);
    }

    private static void dropTable(Statement statement) throws SQLException {
        statement.executeUpdate("if object_id('" + akvTable
                + "','U') is not null" + " drop table " + akvTable);
    }

    private static void dropKeys(Statement statement) throws SQLException {
        statement.executeUpdate(
                "if exists (SELECT name from sys.column_encryption_keys where name='"
                        + cekName + "')" + " begin"
                        + " drop column encryption key " + cekName + " end");
        statement.executeUpdate(
                "if exists (SELECT name from sys.column_master_keys where name='"
                        + cmkName + "')" + " begin" + " drop column master key "
                        + cmkName + " end");
    }

    private static void createCMK(String CUSTOM_AKV_PROVIDER_NAME,
            Statement statement) throws SQLException {
        String _createColumnMasterKeyTemplate = String.format(
                "CREATE COLUMN MASTER KEY [%s] WITH ( KEY_STORE_PROVIDER_NAME = '%s', KEY_PATH = '%s');",
                cmkName, CUSTOM_AKV_PROVIDER_NAME, keyID);
        statement.execute(_createColumnMasterKeyTemplate);
    }

    private static void createCEK(
            SQLServerColumnEncryptionKeyStoreProvider storeProvider,
            Statement statement) throws SQLServerException, SQLException {
        String letters = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
        byte[] valuesDefault = letters.getBytes();
        byte[] key = storeProvider.encryptColumnEncryptionKey(keyID, "RSA_OAEP",
                valuesDefault);
        String cekSql = "CREATE COLUMN ENCRYPTION KEY " + cekName
                + " WITH VALUES " + "(COLUMN_MASTER_KEY = " + cmkName
                + ", ALGORITHM = 'RSA_OAEP', ENCRYPTED_VALUE = 0x"
                + bytesToHexString(key, key.length) + ")" + ";";
        statement.execute(cekSql);
    }
    final static char[] hexChars = {'0', '1', '2', '3', '4', '5', '6', '7', '8',
            '9', 'A', 'B', 'C', 'D', 'E', 'F'};

    private static String bytesToHexString(byte[] b, int length) {
        StringBuilder sb = new StringBuilder(length * 2);
        for (int i = 0; i < length; i++) {
            int hexVal = b[i] & 0xFF;
            sb.append(hexChars[(hexVal & 0xF0) >> 4]);
            sb.append(hexChars[(hexVal & 0x0F)]);
        }
        return sb.toString();
    }

    private static void populateCharNormalCase(Connection connection)
            throws SQLException {
        String sql = "insert into " + akvTable + " values(?,?,?)";
        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            for (int i = 1; i <= 5; i++) { //Insert 5 rows
                for (int j = 1; j <= 3; j++) {
                    pstmt.setNString(j, "Row " + i + " Column " + j);
                }
                pstmt.execute();
            }
        }
    }

    /**
     * Rerieves the table
     *
     * @throws SQLException
     */
    private static void testChar(Statement statement) throws SQLException {
        try (ResultSet rs = statement
                .executeQuery("select * from " + akvTable);) {
            int numberOfColumns = rs.getMetaData().getColumnCount();
            while (rs.next()) {
                for (int i = 1; i <= numberOfColumns; i++) {
                    System.out.println(rs.getString(i));
                }
            }
        }
    }
}
```

See Also

Azure Key Vault Sample Version 6.2.2

Azure Key Vault Sample Version 6.0.0