Progress DataDirect for JDBC for TeamCity

User's Guide

Release 6.0.0

Copyright

Visit the following page online to see Progress Software Corporation’s current Product Documentation Copyright Notice/Trademark Legend: https://www.progress.com/legal/documentation-copyright.

Updated: 2025/05/14

Table of Contents

1

Welcome to the Progress DataDirect for JDBC for TeamCity Driver

The Progress® DataDirect® for JDBC™ for TeamCity™ driver supports SQL read-only access for JDBC applications to TeamCity. The driver provides access to data in TeamCity Enterprise and Community editions (v2018.1 and higher) that can be pulled into a data visualization tool to get important insights into engineering operations. In addition, the driver employs a SQL engine component that provides support to SQL constructs to provide the most comprehensive SQL support and JDBC standard connectivity to BI (Business Intelligence) and ETL (Extract,Transform, Load) tools. To support SQL access to TeamCity services, the driver creates a relational map of the TeamCity data model and translates SQL statements to TeamCity REST API requests.

For an overview of the relational tables exposed by the driver and their corresponding API calls, see Introduction to the TeamCity data model.

The documentation for the driver also includes the *Progress DataDirect for JDBC Drivers Reference*. The reference provides general reference information for all DataDirect drivers for JDBC, including content on troubleshooting, supported SQL escapes, and DataDirect tools.

For the complete documentation set, visit the Progress DataDirect Connectors Documentation Hub: https://docs.progress.com/category/datadirect-teamcity.

For details, see the following topics:

- What's new in this release?
- Requirements
- Installing and setting up the driver
- Driver and DataSource classes
- Connection URL examples

What's new in this release?

Support and certification

Visit the following web pages for the latest support and certification information.

- Release Notes: https://www.progress.com/datadirect-connectors/whats-new#jdbc
- DataDirect Product Compatibility Guide: https://docs.progress.com/bundle/datadirect-product-compatibility/resource/datadirect-product-compatibility.pdf

Changes since 6.0.0 GA

- **Enhancements**
  - The driver has been enhanced to fetch multiple requests simultaneously, thereby improving throughput and performance. You can configure this behavior using the new ReadAhead connection property. See ReadAhead on page 57 for details.

Highlights of 6.0.0 Release

- The driver supports SQL read-only access to TeamCity. See Supported SQL statements and extensions on page 67 for details.
- The driver supports JDBC core functions. For details, refer to "JDBC support" in the *Progress DataDirect for JDBC Drivers Reference*.
- The driver supports TeamCity data types through data type inference. See Data types on page 16 and getTypeInfo() on page 17 for details.
- The driver supports Bearer Token authentication. See Bearer Token Authentication on page 14 for further details.
- The driver supports Basic authentication. See Basic authentication on page 43 for further details.
- The driver supports the handling of large result sets with paging, and the FetchSize on page 52 connection property.
- The driver includes the Progress DataDirect TeamCity Configuration Manager for quick configuration and testing of your driver in a web browser. The tool allows you to:
  - Generate and edit connection URLs
  - Test connect your connection URLs
  - Execute SQL commands for testing

For details, see Generating connection URLs with the Configuration Manager on page 37 and Testing connections and queries on page 38.

Requirements

The driver is compatible with JDBC 2.0, 3.0, and 4.0.

The driver requires a Java Virtual Machine (JVM) that is Java SE 8 or higher, including Oracle JDK, OpenJDK, and IBM SDK (Java) distributions.

Installing and setting up the driver

This section provides you with an overview of the steps required to install and set-up the driver. After completing this procedure, you will be able to begin accessing data with your application.

To begin accessing data with the driver:

1. Install the driver:
   a) After downloading the product, unzip the installer files to a temporary directory.
   b) From the installer directory, run the appropriate installer file to start the installer.
      - **Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.exe`
      - **Non-Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.jar`
   c) Follow the prompts to complete installation.

   The installer program supports multiple installation methods, including command-line and silent installations. For detailed instructions, refer to the *Progress DataDirect for JDBC Drivers Installation Guide*.
2. Set your system CLASSPATH to include the driver `.jar` file. The CLASSPATH is the search string your Java Virtual Machine (JVM) uses to locate JDBC drivers on your computer. The following examples demonstrate setting the CLASSPATH from a command line using the default installation directory.
   - **Windows Example**

     ```
     CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC\lib\60\teamcity.jar
     ```
   - **UNIX/LINUX Example**

     ```
     CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/60/teamcity.jar
     ```
3. Configure your driver using one of the following methods:
   - **Connection URL**: You can begin using the driver immediately by passing a connection URL with your application or tool. The following examples show how to connect using either Basic or Bearer Token authentication.

     Bearer Token

     ```
     jdbc:datadirect:teamcity:ServerName=https://teamcity.company.com;
     AuthenticationMethod=BearerToken;SecurityToken=security_token;
     ```

Note: See Bearer Token Authentication on page 14 for details.

Basic

```
jdbc:datadirect:teamcity:ServerName=https://teamcity.company.com;
AuthenticationMethod=Basic;User=user_name;Password=password;
```

Note: See Basic authentication on page 43 for details.

You can also generate a connection string using the Progress DataDirect TeamCity Configuration Manager. For details, see Generating connection URLs with the Configuration Manager on page 37.

- **Data sources**: The driver also supports connecting using JDBC data sources. A JDBC data source is a Java object, specifically a DataSource object, that defines connection information required for a JDBC driver to connect to the database. See Connecting using data sources for more information.

Note: For most connections, specifying the minimum required connection properties is sufficient to begin accessing data; however, you can provide values for optional properties to use additional supported features and improve performance.

4. Set the values for any optional properties that you want to configure. For additional information on optional features and functionality, see the following resources:
   - Connection URL Examples provides connection string examples that can be used to configure common functionality and features. You can modify and combine these examples to create a string that best suites your environment.
   - Connection Property Descriptions provides a complete list of supported properties by functionality.
   - Performance Considerations describes connection properties that affect performance, along with recommended settings.
5. Connect to your service and begin accessing data with your applications, BI tools, database tools, and more. To help you get started, the following resources guide you through accessing data with some common tools:
   - Progress DataDirect TeamCity Configuration Manager: The TeamCity Configuration Manager is a browser-based tool that allows you to quickly generate connection URLs, test connections, and execute test queries.
   - Tableau: Tableau is a business intelligence software program that allows you to easily create reports and visualized representations of your data.
   - DbVisualizer: DB Visualizer is a database tool that allows you to connect and execute SQL statements against your data.
   - Supported SQL statements and extensions: This section describes the syntax used for SQL statements supported by the driver. You can modify and use the provided examples for your application or tool.

This completes the deployment of the driver.

Driver and DataSource classes

The following are the `Driver` and `DataSource` classes used by the driver:

Driver class:

com.ddtek.jdbc.teamcity.TeamCityDriver

DataSource class:

com.ddtek.jdbcx.teamcity.TeamCityDataSource

Connection URL examples

After setting the CLASSPATH, the connection information needs to be passed in the form of a connection URL. This section provides examples of connection strings configured to use common features and functionality. You can modify and/or combine these examples to create a connection string for your environment.

Note:

- You can also use the DataDirect Configuration Manager tool to generate and test connection URLs. For more information, see "Generating connection URLs with the Configuration Manager."
- Connection property names are case-insensitive. For example, `Password` is the same as `password`.
- For connection properties that support string values, use the following escape sequence to specify values containing leading or trailing spaces and curly brackets: {*value*}. For example: `User={hello }` or `Password={{hello}}`.

Basic Authentication

This string includes only the information required to establish a connection. For detailed descriptions of these properties, see Required properties.

```
jdbc:datadirect:teamcity:ServerName=server_name;User=user_name;Password=password;
[property=value[;...]];
```

where:

server_name

specifies the base URL of the TeamCity service to which you want to issue requests. For example, `https://teamcity.company.com` for enterprise accounts.

Note: Specifying an HTTPS URL for this property enables data encryption.

---

user_name

specifies the user name that is used to connect to your TeamCity service.

password

specifies the password used to connect to your TeamCity service.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

Note: The User and Password properties are not required to be stored in the connection string. They can also be sent separately by the application.

The following example shows how to establish a connection to a TeamCity service:

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:teamcity:ServerName=https://teamcity.company.com;
    User=jsmith;Password=secret;");
```

See also

ServerName on page 59
User on page 63
Password on page 54

Bearer Token Authentication

This string includes only the information required to establish a connection. For detailed descriptions of these properties, see Required properties.

```
jdbc:datadirect:teamcity:ServerName=server_name;AuthenticationMethod=BearerToken;
SecurityToken=security_token;[property=value[;...]];
```

where:

server_name

specifies the base URL of the TeamCity service to which you want to issue requests. For example, `https://teamcity.company.com` for enterprise accounts.

Note: Specifying an HTTPS URL for this property enables data encryption.

security_token

specifies the security token used to connect to your TeamCity service.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

The following example shows how to establish a connection to a TeamCity service:

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:teamcity:ServerName=teamcity.company.com;
    SecurityToken=12a3=bCD/EfGh4Ijk+Lgd8g-44tk3c527831;");
```

See also

ServerName on page 59
SecurityToken on page 58

Proxy server

This string includes the properties you may need to connect through a proxy server with Bearer Token authentication.

```
jdbc:datadirect:teamcity:ServerName=server_name;ProxyHost=proxy_host;
ProxyPassword=proxy_password;ProxyPort=proxy_port;ProxyUser=proxy_user;
AuthenticationMethod=BearerToken;SecurityToken=security_token;
[property=value[;...]];
```

where:

server_name

specifies the base URL of the TeamCity instance to which you want to issue requests. For example, `https://teamcity.company.com` for enterprise accounts.

proxy_host

specifies the proxy server to use for the first connection.

proxy_password

specifies the password needed to connect to a proxy server for the first connection.

proxy_port

specifies the port number where the proxy server is listening for requests for the first connection. The default is `0`.

proxy_user

specifies the user name needed to connect to a proxy server for the first connection.

security_token

specifies the security token used to authenticate to the TeamCity service.

Important: The security token is a confidential value used to authenticate to the server. To prevent unauthorized access, this value must be securely maintained.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

The following example connection string includes the properties required for using a proxy server with Bearer Token authentication.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:teamcity:ServerName=https://teamcity.company.com;
    ProxyHost=pserver;ProxyPassword=secret;ProxyPort=808;
```

```
ProxyUser=jsmith;AuthenticationMethod=bearertoken;
SecurityToken=12a3=bCD/EfGh4Ijk+Lgd8g-44tk3c527831;");
```

See also

Connection property descriptions on page 47

Data types

The following table lists data types supported by the driver and how they are mapped to JDBC data types.

See "getTypeInfo()" for getTypeInfo() results of data types supported by the driver.

Table 1: TeamCity Data Types

TeamCity Data Type	JDBC Data Type
BigInt	BIGINT
Binary	BINARY
Bit	BIT
Boolean	BOOLEAN
Char	CHAR
Date	DATE
Decimal	DECIMAL
Double	DOUBLE
Float	FLOAT
GUID	CHAR
Integer	INTEGER
JSON	VARCHAR
LongVarBinary	LONGVARBINARY
LongVarChar	LONGVARCHAR
NVarChar	NVARCHAR
SmallInt	SMALLINT
Time	TIME
Timestamp	TIMESTAMP

TeamCity Data Type	JDBC Data Type
TinyInt	TINYINT
VarBinary	VARBINARY
VarChar	VARCHAR

getTypeInfo()

The DatabaseMetaData.getTypeInfo() method returns information about data types. The following table provides getTypeInfo() results for supported data types.

Table 2: getTypeInfo() Results

TYPE_NAME = BigInt AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = BigInt MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = 25 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = Binary AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -2 (BINARY) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Binary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32767 SEARCHABLE = 3 SQL_DATA_TYPE = 60 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = Bit AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -7 (BIT) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Bit MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 3 SQL_DATA_TYPE = 14 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Boolean AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 16 (BOOLEAN) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Boolean MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 3 SQL_DATA_TYPE = 16 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Char AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Char MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = 1 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = Date AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = DATE ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Date MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = 91 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Decimal AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Decimal MAXIMUM_SCALE = 1000	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 1000 SEARCHABLE = 3 SQL_DATA_TYPE = 3 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = Double AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Double MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 53 SEARCHABLE = 3 SQL_DATA_TYPE = 8 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE

TYPE_NAME = Float AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 6 (FLOAT) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Float MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 24 SEARCHABLE = 3 SQL_DATA_TYPE = 6 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = GUID AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = GUID MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 36 SEARCHABLE = 3 SQL_DATA_TYPE = 1 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Integer AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Integer MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = 4 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE

TYPE_NAME = JSON AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = JSON MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777215 SEARCHABLE = 3 SQL_DATA_TYPE = 12 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = LongVarBinary AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LongVarBinary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777215 SEARCHABLE = 0 SQL_DATA_TYPE = -4 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = LongVarChar AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LongVarChar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777215 SEARCHABLE = 0 SQL_DATA_TYPE = -1 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = NVarChar AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = -9 (NVARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = NVarChar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32767 SEARCHABLE = 3 SQL_DATA_TYPE = -9 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = SmallInt AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = SmallInt MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 5 SEARCHABLE = 3 SQL_DATA_TYPE = 5 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = Time AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = TIME ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Time MAXIMUM_SCALE = 9	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 12 SEARCHABLE = 3 SQL_DATA_TYPE = 92 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = Timestamp AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = TIMESTAMP ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Timestamp MAXIMUM_SCALE = 9	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 23 SEARCHABLE = 3 SQL_DATA_TYPE = 93 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = TinyInt AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -6 (TINYINT) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = TinyInt MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 3 SEARCHABLE = 3 SQL_DATA_TYPE = -6 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE

TYPE_NAME = VarBinary AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = VarBinary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777215 SEARCHABLE = 3 SQL_DATA_TYPE = 61 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = VarChar AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = VarChar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32767 SEARCHABLE = 3 SQL_DATA_TYPE = 12 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

SQL escape sequences

The driver supports the following SQL escape sequences.

- Date, Time, and Timestamp Escape Sequences
- Scalar Functions
- Outer Join Escape Sequences
- LIKE Escape Character Sequence for Wildcards

Refer to "SQL escape sequences" in the *Progress DataDirect for JDBC Drivers Reference* for information about SQL escape sequences.

Supported scalar functions

The driver supports the scalar functions in the following table. Note that your database system may not support all these functions. Refer to the documentation for your database system to find out which functions are supported by your database.

In addition, you can also determine the supported scalar functions by using DatabaseMetaData methods.

You can use scalar functions in SQL statements with the following syntax:

`{fn scalar-function}`

where:

scalar-function

is a scalar function supported by the drivers, as listed in the following table.

Example:

```
SELECT id, name FROM emp WHERE name LIKE {fn UCASE('Smith')}
```

Refer to "Scalar functions" in the *Progress DataDirect for JDBC Drivers Reference* for more information.

Table 3: Supported Scalar Functions

String Functions	Numeric Functions	Timedate Functions	System Functions
ASCII	ABS	CURDATE	CURSESSIONID
BIT_LENGTH	ACOS	CURRENT_DATE	DATABASE
CHAR	ASIN	CURRENT_TIME	IDENTITY
CHAR_LENGTH	ATAN	CURRENT_TIMESTAMP	USER
CHARACTER_LENGTH	ATAN2	CURTIME	
CONCAT	BITAND	DATEDIFF	
DIFFERENCE	BITOR	DATE_ADD	
HEXTORAW	BITXOR	DATE_SUB	
INSERT	CEILING	DAY	
LCASE	COS	DAYNAME	
LEFT	COT	DAYOFMONTH	
LENGTH	DEGREES	DAYOFWEEK	
LOCATE	EXP	DAYOFYEAR	
LOCATE_2	FLOOR	EXTRACT	

String Functions	Numeric Functions	Timedate Functions	System Functions
LOWER	LOG	HOUR	
LTRIM	LOG10	MINUTE	
OCTET_LENGTH	MOD	MONTH	
RAWTOHEX	PI	MONTHNAME	
REPEAT	POWER	NOW	
REPLACE	RADIANS	QUARTER	
RIGHT	RAND	SECOND	
RTRIM	ROUND	SECONDS_SINCE_MIDNIGHT	
SOUNDEX	ROUNDMAGIC	TIMESTAMPADD	
SPACE	SIGN	TIMESTAMPDIFF	
SUBSTR	SIN	TO_CHAR	
SUBSTRING	SQRT	WEEK	
UCASE	TAN	YEAR	
UPPER	TRUNCATE		

DataDirect tools

Progress DataDirect for JDBC drivers install the set of tools described in this section. For detailed instructions on using these tools, refer to the corresponding topics in the *Progress DataDirect for JDBC Drivers Reference*.

- DataDirect Test allows you to test your JDBC driver and learn the JDBC API.
- DataDirect Connection Pool Manager allows you to pool connections when accessing databases. When your applications use connection pooling, connections are reused rather than created each time a connection is requested. Because establishing a connection is among the most costly operations an application may perform, using Connection Pool Manager to implement connection pooling can significantly improve performance.
- Statement Pool Monitor loads statements into and remove statements from the statement pool as well as generate information to help you troubleshoot statement pooling performance.
- DataDirect Spy logs detailed information about calls your driver makes that can be used for troubleshooting.

Troubleshooting

The *Progress DataDirect for JDBC Drivers Reference* provides information on troubleshooting problems should they occur. Refer to the "Troubleshooting" section in the *Reference* for details.

Additional information

In addition to the content provided in this guide, the documentation set also contains detailed conceptual and reference information that applies to all the drivers. For more information in these topics, refer the *Progress DataDirect for JDBC Drivers Reference* or use the links below to view some common topics:

- "JDBC support" describes support for JDBC interfaces and methods for the Progress DataDirect for JDBC drivers.
- "JDBC extensions" describes the JDBC extensions provided by the com.ddtek.jdbc.extensions package.
- "SQL escape sequences for JDBC" provides an overview of SQL escape sequences for JDBC. In addition, it documents the scalar functions that you use in SQL statements.
- "Security best practices for JDBC applications" describes the security best practices you should employ when developing and deploying your application with the driver.

Contacting Technical Support

Progress DataDirect offers a variety of options to meet your support needs. Please visit our Web site for more details and for contact information:

https://www.progress.com/support

The Progress DataDirect Web site provides the latest support information through our global service network. The SupportLink program provides access to support contact details, tools, patches, and valuable information, including a list of FAQs for each product. In addition, you can search our Knowledgebase for technical bulletins and other information.

When you contact us for assistance, please provide the following information:

- Your number or the serial number that corresponds to the product for which you are seeking support, or a case number if you have been provided one for your issue. If you do not have a SupportLink contract, the SupportLink representative assisting you will connect you with our Sales team.
- Your name, phone number, email address, and organization. For a first-time call, you may be asked for full information, including location.
- The Progress DataDirect product and the version that you are using.
- The type and version of the operating system where you have installed your product.
- Any database, database version, third-party software, or other environment information required to understand the problem.
- A brief description of the problem, including, but not limited to, any error messages you have received, what steps you followed prior to the initial occurrence of the problem, any trace logs capturing the issue, and so

on. Depending on the complexity of the problem, you may be asked to submit an example or reproducible application so that the issue can be re-created.

- A description of what you have attempted to resolve the issue. If you have researched your issue on Web search engines, our Knowledgebase, or have tested additional configurations, applications, or other vendor products, you will want to carefully note everything you have already attempted.
- A simple assessment of how the severity of the issue is impacting your organization.

2

Tutorials

The following sections guide you through using the driver to access your data with some common third-party applications. For information on installing your driver and setting the CLASSPATH, see "Installing and setting-up the driver."

For details, see the following topics:

- Tableau
- DbVisualizer
- Interactive SQL for JDBC (JDBCISQL)

Tableau

After you have installed your driver and defined it on the CLASSPATH, you can use the driver to access your data with Tableau. Tableau is a business intelligence software program that allows you to easily create reports and visualized representations of your data. By using the driver with Tableau, you can improve performance when retrieving data while leveraging the driver's relational mapping tools.

To use the driver to access data with Tableau:

1. Navigate to the `\lib\xx` subdirectory of the Progress DataDirect installation directory; then, locate the `jar` file for your driver:

   ```
   teamcity.jar
   ```

2. Copy the `.jar` file for your driver into the following directory:

Windows: `C:\Program Files\Tableau\Drivers`

Linux: `/opt/tableau/tableau_driver/jdbc`

3. Open Tableau. From the **Connect** menu, select **Other Databases (JDBC)**.
4. In the **Other Databases (JDBC)** dialog, provide values for the following fields; then, click **Sign In**.
   - **URL**: Copy and paste your connection URL into this field. The following examples show how to connect using Bearer Token authentication.

     Bearer Token

     ```
     jdbc:datadirect:teamcity:ServerName=https://teamcity.company.com;
     AuthenticationMethod=BearerToken;SecurityToken=security_token;
     ```

     Note: See Bearer Token Authentication on page 14 for details.

   - **Dialect**: Select **SQL92** (the default) from the drop-down box.
   - **User**: If required by the authentication method being used, enter the user name. Alternatively, this value can be specified with the `User` property in the connection string.
   - **Password**: If required by the authentication method being used, enter the password. Alternatively, this value can be specified with the `Password` property in the connection string.
5. The **Data Source** window appears. In the **Schema** field, select the schema for the service you want to use.
6. In the **Table** field, the tables stored in the selected schema are now exposed and available for selection.

You have successfully accessed your data and are now ready to create reports with Tableau. For detailed information, refer to the Tableau product documentation at: https://www.tableau.com/support/help.

DbVisualizer

After you have installed your driver and defined it on the CLASSPATH, you can use the driver to access your data with the third-party DbVisualizer tool. The following topics guide you through using DbVisualizer to add your driver, connect, and execute SQL statements.

Adding a driver

To add a driver with DbVisualizer:

1. Open DbVisualizer.
2. From the menu, select **Tools**>**Driver Manager**. The Driver Manager window opens.
3. From the Driver Manager menu, select **Driver**>**Create Driver**.
4. Click the  button to navigate to the location of the driver jar file; then, click **OK**. The following are the default locations for the driver:

   Windows

   ```
   C:\Program Files\Progress\DataDirect\JDBC\lib\60\teamcity.jar
   ```

Linux

```
/opt/Progress/DataDirect/JDBC/lib/60/teamcity.jar
```

5. Provide values for the following fields; then, close the Driver Manager window.
   - **Name**: Type an alias for your driver. For example:

     ```
     TeamCity
     ```

   - **URL Format**: Optionally, specify the format of the connection string for your driver. For example:

     ```
     jdbc:datadirect:teamcity:ServerName=<server>
     ```

   - **Driver Class**: From the drop down menu, select the driver class for your driver. For example:

     ```
     com.ddtek.jdbc.teamcity.TeamCityDriver
     ```

You can now use your driver with DbVisualizer. Proceed to "Connecting and executing SQL statements" for information on connecting and executing SQL statements.

Connecting and executing SQL statements

To use the driver to access data with DbVisualizer:

1. Open DbVisualizer.
2. From the menu, select **Database**>**New Connection**. When prompted to use the Connection Wizard, click **OK**.
3. Provide the following information when prompted; then, click **Next** to proceed:
   - **Connection alias**: Type the name to be used when referring to this connection.
   - **Driver**: Select the alias that you provided for your driver from the drop-down menu.
4. Provide values for the following fields; then, click **Finish**.
   - **Database URL**: Copy and paste your connection URL into this field. The following examples show how to connect using Bearer Token authentication.

     Note: You can also generate connection strings using TeamCity Configuration Manager. For more information, see Generating connection URLs with the Configuration Manager on page 37.

     Bearer token grant

     ```
     jdbc:datadirect:teamcity:ServerName=teamcity.company.com;
     AuthenticationMethod=BearerToken;SecurityToken=security_token;
     ```

   - **Database UserId**: If required by the authentication method being used (Basic), enter the user name. Alternatively, this value can be specified with the `User` property in the connection string.
   - **Database Password**: If required by the authentication method being used (Basic), enter the password. Alternatively, this value can be specified with the `Password` property in the connection string.

5. To execute SQL statements, select **SQL Commander**>**New SQL Commander**. A SQL Commander tab opens.
6. Select values for the following fields:
   - **Database Connection**: Select connection alias you provided for the connection from the drop-down menu.
   - **Schema**: Select the schema you want to execute queries against from the drop-down menu.
7. In the SQL Commander tab, enter SQL commands you want to execute; then select **SQL Commander**>**Execute**. For example:

   To select all of the rows from the USERS table:

   ```
   SELECT * FROM USERS
   ```

   To select the URLs for a specified issue:

   ```
   SELECT * FROM AGENTS WHERE AGENTID = <ID>
   ```

   See "Supported SQL statements and extensions" for the supported syntax used to execute SQL statements.

   Note: If you are fetching large sets of data, you may want to limit the results using the Max Rows and Max Chars fields.

You have successfully accessed your data with DbVisualizer.

Interactive SQL for JDBC (JDBCISQL)

After you have installed your driver and defined it on the CLASSPATH, you can use the driver to access your data with Interactive SQL for JDBC (JDBCISQL). JDBCISQL supports a command line interface that allows you to connect to a data source, execute SQL statements and retrieve results for display on a terminal.

To execute commands with JDBCISQL:

1. Start the ISQL tool. Do one of the following:
   - On Windows, double-click the`jdbcisql.bat` file in the *`install_dir`*`\jdbcisql` folder. Or, from a command prompt, navigate to the *`install_dir`*`\jdbcisql` directory and run the `jdbcisql.bat` file.
   - On Linux and UNIX, change to the *`install_dir`*`\isql` directory and run `jdbcisql.sh`.

   The Interactive SQL prompt appears.
2. Type the driver name class; then, press **Enter**:

   ```
   com.ddtek.jdbc.teamcity.TeamCityDriver
   ```

3. Type `connect` followed by the connection URL for the driver; then, press **Enter**. For example:

   ```
   connect jdbc:datadirect:teamcity:HostName=teamcity.company.com;
   User=jsmith;Password=secret;
   ```

   If successful, the tool will return the time required to connect.

4. At the ISQL> prompt, issue a SQL command to query or modify the data source; then, press **Enter**. For example:

   ```
   SELECT * FROM PROJECTS;
   ```

   Note: SQL commands must be terminated by a semi-colon.

   Note: In addition to SQL commands, JDBCISQL supports a set of proprietary commands. Type `Help` at the prompt for a list of supported commands and syntax.

   The results of the command are displayed in the terminal.

5. After you are finished executing queries and commands, you can disconnect from the data source by typing the following; then, pressing **Enter**:

   ```
   DISCONNECT;
   ```

6. Press any key to end the session.

3

Configuring and connecting

This section provides information on how to connect to your data store using either the JDBC Driver Manager or DataDirect JDBC data sources, as well as information on how to implement and use functionality supported by the driver.

After the driver has been installed and defined on your classpath, you can connect from your application to your data in either of the following ways.

- Using the JDBC `DriverManager` by specifying the connection URL in the `DriverManager.getConnection()` method.
- Creating a JDBC data source that can be accessed through the Java Naming Directory Interface (JNDI).

For details, see the following topics:

Setting the classpath

The driver must be defined on your CLASSPATH before you can connect. The CLASSPATH is the search string your Java Virtual Machine (JVM) uses to locate JDBC drivers on your computer. If the driver is not defined on your CLASSPATH, you will receive a `class not found` exception when trying to load the driver. Set your system CLASSPATH to include the driver jar file as shown, where *`install_dir`* is the path to your product installation directory.

```
install_dir/lib/60/teamcity.jar
```

Windows Example

```
CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC\lib\60\teamcity.jar
```

UNIX Example

```
CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/60/teamcity.jar
```

Connecting using the JDBC Driver Manager

One way to connect to a service is through the JDBC DriverManager using the `DriverManager.getConnection()` method. As the following examples show, this method specifies a string containing a connection URL.

Bearer Token authentication

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:teamcity:ServerName=https://teamcity.company.com;
   AuthenticationMethod=BearerToken;SecurityToken=12a3=bCD/4tk3c527831;");
```

Note: See Bearer token authentication on page 44 for details.

Passing the connection URL

After setting the CLASSPATH, the required connection information needs to be passed in the form of a connection URL. The following example includes the properties required for connecting with Bearer Token authentication.

Connection URL Syntax

The connection URL takes the following form:

```
jdbc:datadirect:teamcity:ServerName=server_name;AuthenticationMethod=BearerToken;
SecurityToken=security_token;[property=value[;...]];
```

where:

server_name

specifies the base URL of the TeamCity instance to which you want to issue requests. For example, `https://teamcity.company.com` for enterprise accounts.

security_token

Specifies the security token used to authenticate to TeamCity when Bearer Token authentication is enabled.

Important: The security token is a confidential value used to authenticate to the server. To prevent unauthorized access, this value must be securely maintained.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

The following example connection string includes the properties required for connecting with the Bearer Token authentication.

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:teamcity:ServerName=https://teamcity.company.com;
AuthenticationMethod=BearerToken;SecurityToken=12a3=bCD/4tk3c527831;");
```

See also

Connection property descriptions on page 47
Connection URL examples on page 13

Generating connection URLs with the Configuration Manager

The driver includes a browser-based tool, Progress DataDirect TeamCity Configuration Manager, that allows you to generate connection URLs, test connections, and execute test queries. This section will guide you through generating and testing a connection URL that can be used by your application.

To generate a connection URL:

1. Open the TeamCity Configuration Manager by double-clicking the driver jar file. Or, in a command line, navigate to the directory containing your driver jar file; then, execute the following command:

   ```
   java -jar teamcity.jar
   ```

   The TeamCity Configuration Manager opens in your default web browser.

2. From the browser window, provide values of the connection properties you want to configure in the corresponding fields. A connection URL will generate in the Connection String field as you provide settings. To view more properties, select the tabs at the top of the page. See "Connection URL examples" for a list of required properties and properties used for different configurations.

   Note: If you do not specify a value for an optional property, the property will be omitted from the string and the default value will be used.

3. Optionally, you can manually edit your string by clicking the Edit button ().

4. At any point during the process, you can click **Test Connect** to attempt to connect to the service using the string generated in the Connection String field. In the **Test Connection** window:
   a) Provide values for any fields required by your service.
   b) Optionally, in the Test Query field, enter any SQL queries you want to execute during the test. For example:

      ```
      SELECT * FROM INFORMATION_SCHEMA.SYSTEM_TABLES
      ```

   c) Click **Execute**.

      If successful, the window displays a confirmation message and, if a query was specified, the results of the query.

   Note: The information you enter in the logon dialog box during a test connect is not saved.

5. To use your string, click the Copy button () and paste the string to a location that can be used by your application.

Testing connections and queries

You can quickly test a connection string and queries using Progress DataDirect TeamCity Configuration Manager.

To test your connection and query:

1. Open the TeamCity Configuration Manager by double-clicking on the driver jar file. Or, in a command line, navigate to the directory containing your driver jar file; then, execute the following command:

   ```
   java -jar teamcity.jar
   ```

   The TeamCity Configuration Manager opens in your default web browser.

2. Provide a connection string to test using one of the following methods:

   - Entering a string: Click the Edit button (); then, paste your string into the Connection String field. If you prefer, you can also type a string directly into this field.
   - Generating a string: Provide values of the connection properties you want to configure into the corresponding fields. The TeamCity Configuration Manager will generate a string in the Connection String field based on the values you specify.

3. Click **Test Connect** to attempt to connect to the service using the string specified in the Connection String field. The **Test Connection** window appears.
4. Provide connection property values for any fields required by your service.
5. To execute a test query with the test connection, enter a SQL query into the Test Query field. For example:

   ```
   SELECT * FROM INFORMATION_SCHEMA.SYSTEM_TABLES
   ```

6. Click **Execute**.

   If successful, the window displays a confirmation message and, if a query was specified, the results of the query.

Connecting using data sources

A *JDBC data source* is a Java object, specifically a `DataSource` object, that defines connection information required for a JDBC driver to connect to the database. Each JDBC driver vendor provides their own data source implementation for this purpose. A Progress DataDirect data source is Progress DataDirect's implementation of a `DataSource` object that provides the connection information needed for the driver to connect to a database.

Because data sources work with the Java Naming Directory Interface (JNDI) naming service, data sources can be created and managed separately from the applications that use them. Because the connection information is defined outside of the application, the effort to reconfigure your infrastructure when a change is made is minimized. For example, if the database is moved to another database server, the administrator need only change the relevant properties of the `DataSource` object. The applications using the database do not need to change because they only refer to the name of the data source.

How data sources are implemented

Data sources are implemented through a `DataSource` class. A data source class implements the following interfaces.

- `javax.sql.DataSource`
- `javax.sql.ConnectionPoolDataSource` (allows applications to use connection pooling)

Refer to "Connection Pool Manager" in the *Progress DataDirect for JDBC Drivers Reference* for more information.

See also

Driver and DataSource classes on page 13

Creating data sources

The following example files provide details on creating and using Progress DataDirect data sources with the Java Naming Directory Interface (JNDI), where *`install_dir`* is the product installation directory.

- *`install_dir`*`/Examples/JNDI/JNDI_LDAP_Example.java` can be used to create a JDBC data source and save it in your LDAP directory using the JNDI Provider for LDAP.
- *`install_dir`*`/Examples/JNDI/JNDI_FILESYSTEM_Example.java` can be used to create a JDBC data source and save it in your local file system using the File System JNDI Provider.

See "Example data source" for an example data source definition for the example files.

To connect using a JNDI data source, the driver needs to access a JNDI data store to persist the data source information. For a JNDI file system implementation, you must download the File System Service Provider from the Oracle Technology Network Java SE Support downloads page, unzip the files to an appropriate location, and add the `fscontext.jar` and `providerutil.jar` files to your CLASSPATH. These steps are not required for LDAP implementations because the LDAP Service Provider is included with supported versions of Java SE.

Example data source

To configure a data source using the example files, you will need to create a data source definition. The content required to create a data source definition is divided into three sections.

First, you will need to import the data source class. For example:

```
import com.ddtek.jdbcx.teamcity.TeamCityDataSource;
```

Next, you will need to set the values and define the data source. For example, the following definition contains the minimum properties required for a connection using the Bearer Token authentication.

Note: The security token is a confidential value used to authenticate to the server. To prevent unauthorized access, this value must be securely maintained.

```
TeamCityDataSource mds = new TeamCityDataSource();
mds.setDescription("My TeamCity Data Source");
mds.setServerName("teamcity.company.com");
mds.AuthenticationMethod("BearerToken");
mds.SecurityToken("12a3=bCD/EfGh4Ijk+Lgd8g-44tk3c527831");
```

Finally, you will need to configure the example application to print out the data source attributes. Note that this code is specific to the driver and should only be used in the example application. For example, you would add the following section for the minimum properties required to establish a connection:

```
if (ds instanceof TeamCityDataSource)
{
TeamCityDataSource jmds = (TeamCityDataSource) ds;
System.out.println("description=" + jmds.getDescription());
System.out.println("servername=" + jmds.getServerName());
System.out.println("authenticationmethod=" + jmds.getAuthenticationMethod());
System.out.println("securitytoken=" + jmds.getSecurityToken());
...
System.out.println();
}
```

Calling a data source in an application

Applications can call a Progress DataDirect data source using a logical name to retrieve the `javax.sql.DataSource` object. This object loads the specified driver and can be used to establish a connection to the database.

Once the data source has been registered with JNDI, it can be used by your JDBC application as shown in the following code example.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("EmployeeDB");
Connection con = ds.getConnection("domino", "spark");
```

In this example, the JNDI environment is first initialized. Next, the initial naming context is used to find the logical name of the data source (`EmployeeDB`). The `Context.lookup()` method returns a reference to a Java object, which is narrowed to a `javax.sql.DataSource` object. Then, the `DataSource.getConnection()` method is called to establish a connection.

Testing a data source connection

You can use DataDirect Test™ to establish and test a data source connection. The screen shots in this section were taken on a Windows system.

Take the following steps to establish a connection.

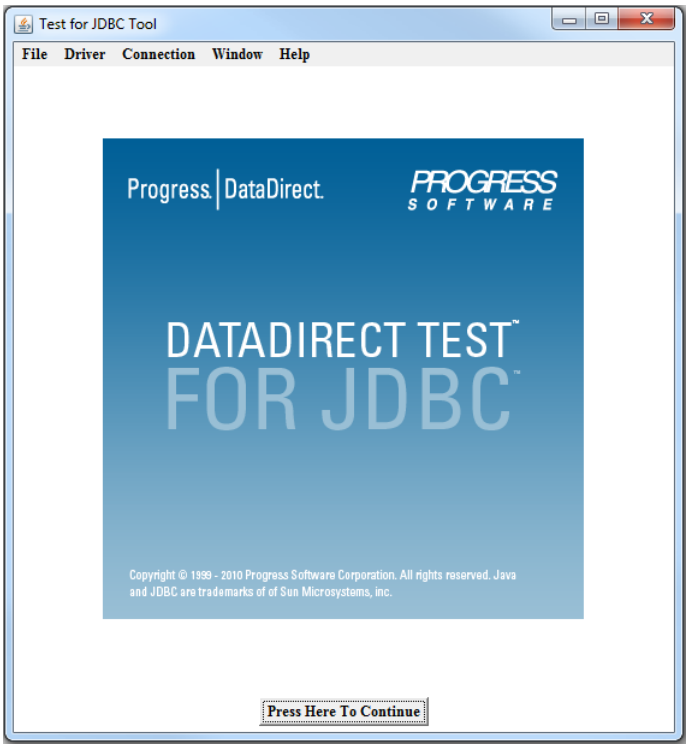
1. Navigate to the installation directory. The default location is:

- Windows systems: `Program Files\Progress\DataDirect\JDBC\testforjdbc`
- UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC/testforjdbc`

Note: For UNIX/Linux, if you do not have access to `/opt`, your home directory will be used in its place.

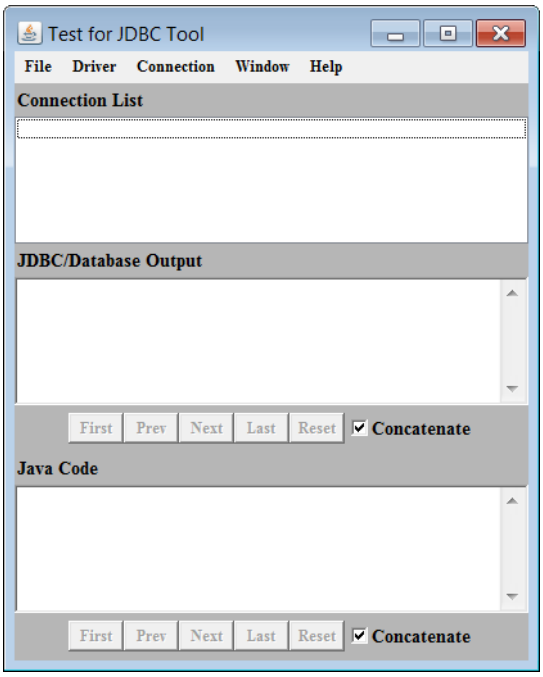
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

   The **Test for JDBC Tool** window appears:



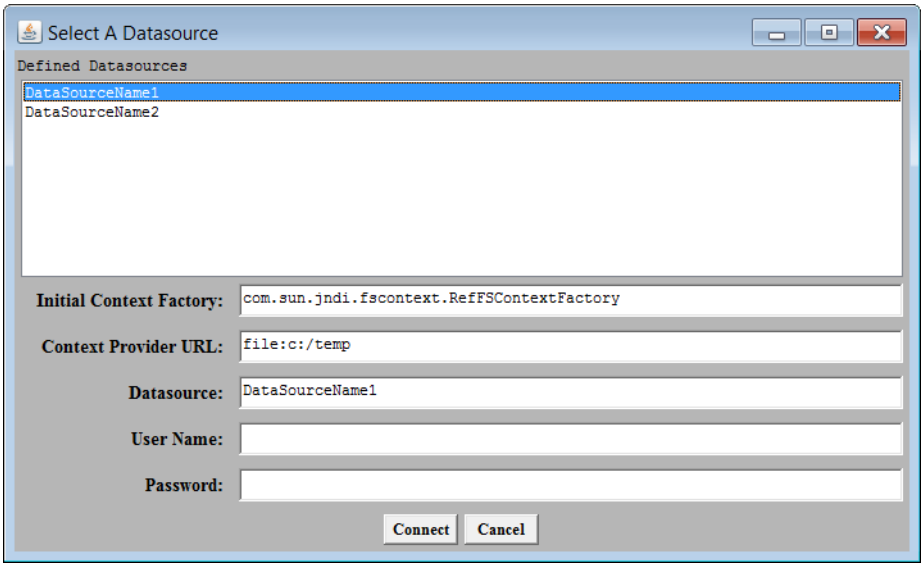
3. Click **Press Here to Continue**.

   The main dialog appears:



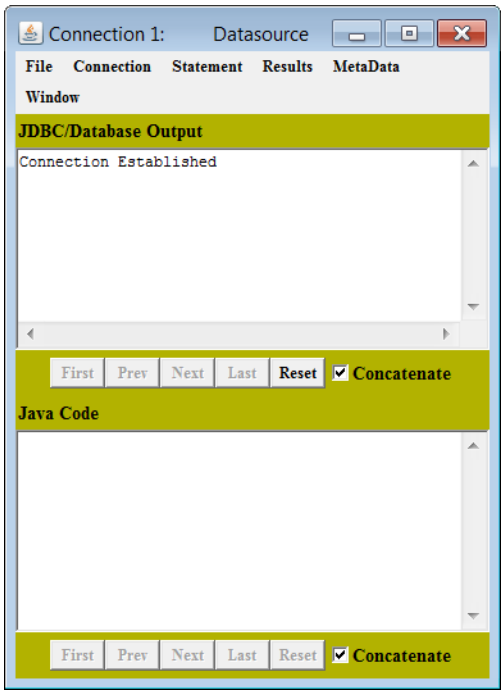
4. From the menu bar, select **Connection > Connect to DB via Data Source**.

   The **Select A Database** dialog appears:



5. Select a datasource template from the **Defined Datasources** field.
6. Provide the following information:
   a) In the **Initial Context Factory**, specify the location of the initial context provider for your application.
   b) In the **Context Provider URL**, specify the location of the context provider for your application.
   c) In the **Datasource** field, specify the name of your datasource.
7. If you are using user ID/password authentication, enter your user ID and password in the corresponding fields.
8. Click **Connect**.

If the connection information is entered correctly, the **JDBC/Database Output** window reports that a connection has been established. If a connection is not established, the window reports an error.



Authentication

The driver supports the following authentication methods:

- *Basic Authentication* authenticates using the specified user IDs and passwords.
- *Bearer Token Authentication* authenticates using security tokens generated by TeamCity.

By default, the driver is configured to use basic authentication (`AuthenticationMethod=Basic`).

See also

AuthenticationMethod on page 50

Basic authentication

This string includes the properties used to connect with basic authentication.

```
jdbc:datadirect:teamcity:ServerName=server_name;User=user_name;
Password=password;property=value[;...]];
```

where:

server_name

specifies the URL of the TeamCity service to which you want to issue requests. For example, `https://teamcity.company.com` for enterprise accounts.

user_name

specifies the user name that is used to connect to the TeamCity service. For example, `jsmith`.

password

specifies the password used to connect to your TeamCity service.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following example connection string includes the properties required for connecting with basic authentication.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:teamcity:ServerName=teamcity.company.com;
   AuthenticationMethod=Basic;User=jsmith;Password=secret;");
```

See also

Bearer Token Authentication on page 14

Bearer token authentication

This string includes the properties used to connect with bearer token authentication.

```
jdbc:datadirect:teamcity:ServerName=server_name;AuthenticationMethod=BearerToken;
SecurityToken=security_token;property=value[;...]];
```

where:

server_name

specifies the URL of the TeamCity service to which you want to issue requests. For example, `https://teamcity.company.com` for enterprise accounts.

security_token

specifies the security token that is used to connect to your TeamCity service.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

The following example connection string includes the properties required for connecting with bearer token authentication.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:teamcity:ServerName=https://teamcity.company.com;
    AuthenticationMethod=BearerToken;SecurityToken=secret;");
```

See also

Basic authentication on page 43

Performance considerations

FetchSize: FetchSize can be used to adjust the trade-off between throughput and response time. In general, setting larger values for FetchSize will improve throughput, but can reduce response time. You should set FetchSize to suit your environment. Smaller fetch sizes can improve the initial response time of the query. Larger fetch sizes improve overall fetch times at the cost of additional memory.

ReadAhead: The ReadAhead property allows you to issue multiple fetch requests in parallel. By increasing this number, you can improve throughput and performance, but it does so with the following restrictions:

- Larger values can increase the load on the server, which may adversely affect performance of other users. If you encounter issues, decrease the value specified for this option.
- Larger values may result in unnecessary requests if your application only requires the first few rows of results. This may be an issue if your service places limits on the number of web requests.

Caution: Due to potential impacts to other users, we strongly recommend specifying only smaller values for the ReadAhead property. For example, in fully optimized environments, which include exceptionally fast connections and low latency, we recommend a setting of no higher than 5. For typical environments, this value is sometimes lower.

4

Connection property descriptions

You can use connection properties to customize the driver for your environment. This section organizes connection properties according to functionality. You can use connection properties with either the JDBC `DriverManager` or a JDBC data source. For a `DriverManager` connection, a property is expressed as a key value pair and takes the form *`property=value`*. For a data source connection, a property is expressed as a JDBC method and takes the form `set`*`property`*`(`*`value`*`)`.

Note: Connection property names are case-insensitive. For example, `Password` is the same as `password`.

Note: In a JDBC data source, string values must be enclosed in double quotation marks, for example, `setUser("abc@defcorp.com")`.

The following tables describe the connection properties by functionality.

- Authentication properties
- Proxy server properties
- Timeout properties
- Additional properties

Authentication properties

The following table summarizes properties used for Basic and Bearer Token authentication methods.

Property	Data Source Method	Default
ServerName on page 59	`getServerName()` `setServerName(String)`	No default value
AuthenticationMethod on page 50	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	Basic
Password on page 54	`getPassword()` `setPassword(String)`	No default value
SecurityToken on page 58	`getSecurityToken()` `setSecurityToken(String)`	No default value
User on page 63	`getUser()` `setUser(String)`	No default value

Proxy server properties

The following table summarizes proxy server connection properties.

Property	Data Source Method	Default
ProxyHost on page 55	`getProxyHost()` `setProxyHost(String)`	No default value
ProxyPassword on page 55	`getProxyPassword()` `setProxyPassword(String)`	No default value
ProxyPort on page 56	`getProxyPort()` `setProxyPort(Integer)`	`0` which means the default is determined by the ProxyHost property. For HTTP URLs: `80` For HTTPS URLs: `443`
ProxyUser on page 56	`getProxyUser()` `setProxyUser(String)`	No default value

Timeout properties

The following table summarizes timeout connection properties.

Property	Data Source Method	Default
WSRetryCount on page 64	`getWsRetryCount()` `setWsRetryCount(Integer)`	`5`
WSTimeout on page 64	`getWsTimeout()` `setWsTimeout(Integer)`	`120` (seconds)

Additional properties

The following table summarizes additional connection properties.

Property	Data Source Method	Default
DebugRecord on page 51	`getDebugRecord()` `setDebugRecord(String)`	`0` (Disabled)
DefaultQueryOptions on page 52	`getDefaultQueryOptions()` `setDefaultQueryOptions(String)`	No default value
FetchSize on page 52	`getFetchSize()` `setFetchSize(Integer)`	`100` (rows)
LogConfigFile on page 53	`getLogConfigFile()` `setLogConfigFile(String)`	`ddlogging.properties`
ReadAhead on page 57	`getReadAheadThreads()` `setReadAheadThreads(Integer)`	`5`
SpyAttributes on page 59	`getSpyAttributes()` `setSpyAttributes(String)`	No default value
StmtCallLimit on page 62	`getStmtCallLimit()` `setStmtCallLimit(Integer)`	`0`
StmtCallLimitBehavior on page 62	`getStmtCallLimitBehavior()` `setStmtCallLimitBehavior(String)`	`ErrorAlways`

For details, see the following topics:

- AuthenticationMethod
- DebugRecord
- DefaultQueryOptions

AuthenticationMethod

Purpose

Determines which authentication mechanism the driver uses when establishing a connection.

Valid Values

`Basic` | `BearerToken`

Behavior

If set to `Basic`, the driver uses a hashed value, based on the concatenation of the user name and password, for authentication.

If set to `BearerToken`, the driver uses an API Token (configured as `BearerToken`) for authentication.

Data Source Methods

```
public String getAuthenticationMethod()
```

```
public void setAuthenticationMethod(String)
```

Default Value

`Basic`

Data Type

String

See also

Authentication on page 43

DebugRecord

Purpose

Specifies the directory where the driver generates debug record files. When a value is specified, the driver records server requests and responses to a set of files stored in this location. These files assist in troubleshooting by providing a method for Technical Support to reproduce and debug issues for REST services that are not publicly accessible.

Important: Debug record files may capture security-related headers, such as auth or token headers. Before sending Technical Support debug files, review the content to remove any confidential information that may have been recorded.

Valid Values

debug_record_folder

where:

debug_record_folder

is the location of the folder where the debug record files are to be generated. For example, `C:\Temp\MyDebug Folder`.

Notes

- The specified directory must exist.
- You must have write access to the specified directory.
- The contents of the specified directory are deleted every time a connection is established.
- For more information, refer to "Enabling Debug Record Mode" in the *Progress DataDirect for JDBC Drivers Reference*.
- For assistance, contact Technical Support.

Data Source Methods

```
public String getDebugRecord()
```

```
public void setDebugRecord(String)
```

Default Value

No default value

Data Type

String

DefaultQueryOptions

Purpose

Specifies a semicolon-separated list of parameters that are used as default filter values (Where clauses) for SQL queries.

Valid Values

string

where:

string

is a set of parameters and default filter values that you want to apply to SQL queries.

Notes

- A Where clause used in a SQL query overrides the DefaultQueryOptions passed in a connection URL.

Data Source Methods

```
public String getDefaultQueryOptions()
```

```
public void setDefaultQueryOptions(String)
```

Default Value

No default value

Data Type

String

FetchSize

Purpose

Specifies the maximum number of rows that the driver processes before returning data to the application when executing a Select. This value provides a suggestion to the driver as to the number of rows it should internally process before returning control to the application. The driver may fetch fewer rows to conserve memory when processing exceptionally wide rows.

Valid Values

`0` | *x*

where:

x

is a positive integer indicating the number of rows that should be processed.

Behavior

If set to `0`, the driver processes all the rows of the result before returning control to the application. When large data sets are being processed, setting FetchSize to `0` can diminish performance and increase the likelihood of out-of-memory errors.

If set to *x*, the driver limits the number of rows that may be processed for each fetch request before returning control to the application.

Notes

- To optimize throughput and conserve memory, the driver uses an internal algorithm to determine how many rows should be processed based on the width of rows in the result set. Therefore, the driver may process fewer rows than specified by FetchSize when the result set contains exceptionally wide rows. Alternatively, the driver processes the number of rows specified by FetchSize when the result set contains rows of unexceptional width.
- FetchSize can be used to adjust the trade-off between throughput and response time. Smaller fetch sizes can improve the initial response time of the query. Larger fetch sizes can improve overall response times at the cost of additional memory.
- You can use FetchSize to reduce demands on memory and decrease the likelihood of out-of-memory errors. Simply, decrease FetchSize to reduce the number of rows the driver is required to process before returning data to the application.

Data Source Methods

```
public Integer getFetchSize()
```

```
public void setFetchSize(Integer)
```

Default Value

`100`

Data Type

Integer

See also

Performance considerations on page 45

LogConfigFile

Purpose

Specifies the file name, and optionally, the path of the properties file used to initialize driver logging.

Valid Values

`String`

where:

`String`

is the relative or fully qualified path of the properties file to load to initialize driver logging. If you do not specify a path, the driver looks for this file in the current working directory. If the specified file does not exist, the driver continues searching for an appropriate properties file as described in "Using Java Logging" in the *Progress DataDirect for JDBC Drivers Reference*.

Data Source Methods

```
public String getLogConfigFile()
```

```
public void setLogConfigFile(String)
```

Default Value

```
ddlogging.properties
```

Data Type

String

Password

Purpose

A password that is used to connect to the service.

Important: Setting the password using a data source is not recommended. The data source persists all properties, including password, in clear text.

Behavior

password

where:

password

is a valid password. The password is case-sensitive.

Data Source Methods

```
public String getPassword()
```

```
public void setPassword(String)
```

Default Value

No default value

Data Type

String

See also

User on page 63

ProxyHost

Purpose

Identifies a proxy server to use for the first connection.

Valid Values

server_name | *IP_address*

where:

server_name

is the name of the proxy server, which may be qualified with the domain name.

IP_address

is an IP address, specified in either IPv4 or IPv6 format, or a combination of the two.

Data Source Methods

```
public String getProxyHost()
public void setProxyHost(String)
```

Default Value

No default value

Data Type

String

ProxyPassword

Purpose

Specifies the password needed to connect to a proxy server for the first connection.

Valid Values

password

where:

password

is a valid password for that server. Contact your system administrator to obtain a valid password.

Data Source Methods

```
public String getProxyPassword()
public void setProxyPassword(String)
```

Default Value

No default value

Data Type

String

ProxyPort

Purpose

Specifies the port number where the proxy server is listening for HTTP or HTTPS requests for the first connection.

Valid Values

port

where:

port

is the port number on which the proxy server is listening. Contact your system administrator to obtain the correct port.

Data Source Methods

```
public Integer getProxyPort()
```

```
public void setProxyPort(Integer)
```

Default Value

0 which means that the default value is determined by whether the value specified for the ProxyHost property is an HTTP or HTTPS URL.

For HTTP: 80

For HTTPS: 443

Data Type

Integer

ProxyUser

Purpose

Specifies the user name needed to connect to a proxy server for the first connection.

Valid Values

user_name

where:

user_name

is a valid user ID for the proxy server.

Data Source Methods

```
public String getProxyUser()
public void setProxyUser(String)
```

Default Value

No default value

Data Type

String

ReadAhead

Purpose

Specifies the maximum number of fetch requests the driver issues in parallel. By default, the driver queues the next page when processing the current page. This property allows you to fetch multiple requests simultaneously, thereby improving throughput and performance.

Caution: Due to potential impacts to other users on the network, we strongly recommend specifying only smaller values for this property. For example, in fully optimized environments, which include exceptionally fast connections and low latency, we recommend a setting of no higher than `10`. For typical environments, this value should be considerably lower.

Valid Values

`0` | *x*

where:

x

is the maximum number of fetch requests the driver issues in parallel up to 100.

Behavior

If set to `0`, the driver queues the next page while processing the current page.

If set to *x*, the driver executes fetch requests as they are issued until the number of active parallel-requests equals the specified value. When that threshold is met, the driver waits until the results of a request are processed before requesting the next page of data.

Notes

- Specifying larger values for this property generally improves performance; however, with the following warnings:
  - Larger values can increase the load on the server, which may adversely affect performance of other users. If you encounter issues, decrease the value specified for this property.
  - Larger values may result in unnecessary requests if your application only requires the first few rows of results. This may be an issue if your service places limits on the number of web requests.

Data Source Methods

```
public Integer getReadAheadThreads()
```

```
public void setReadAheadThreads(Integer)
```

Default Value

0

Data Type

Integer

SecurityToken

Purpose

Specifies the security token required to make a connection to the server. This property is required when token-based authentication is enabled (AuthenticationMethod=BearerToken); otherwise, this property is ignored.

Important: If setting the security token using a data source, be aware that the SecurityToken property, like all data source properties, is persisted in clear text.

Valid Values

string

where:

string

is the value of the security token assigned to the user.

Data Source Methods

```
public String getSecurityToken()
```

```
public void setSecurityToken(String)
```

Default Value

No default value

Data Type

String

See also

Bearer token authentication on page 44

ServerName

Purpose

Specifies the host name portion of the HTTP endpoint to which you send requests.

Valid Values

string

where:

string

is the host name portion of the HTTP endpoint to which you send requests.

Data Source Methods

`public String getServerName()`

`public void setServerName(String)`

Default Value

No default value

Data Type

String

SpyAttributes

Purpose

Enables DataDirect Spy to log detailed information about calls that are issued by the driver on behalf of the application. DataDirect Spy is not enabled by default.

Valid Values

(*spy_attribute*[;*spy_attribute*]...)

where:

spy_attribute

is any valid DataDirect spy attribute.

Behavior

Attribute	Description
linelimit=*numberofchars*	Sets the maximum number of characters that DataDirect Spy logs on a single line. The default is 0 (no maximum limit).
load=*classname*	Loads the driver specified by *classname*.
log=(file)*filename*	Directs logging to the file specified by *filename*. For Windows, if coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: log=(file)C:\\temp\\spy.log;logIS=yes;logTName=yes.
log=(filePrefix)*file_prefix*	Directs logging to a file prefixed by *file_prefix*. The log file is named *file_prefixX*.log where: *X* is an integer that increments by 1 for each connection on which the prefix is specified. For example, if the attribute log=(filePrefix)C:\\temp\\spy_ is specified on multiple connections, the following logs are created: C:\temp\spy_1.log C:\temp\spy_2.log C:\temp\spy_3.log ... If coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: log=(filePrefix)C:\\temp\\spy_;logIS=yes;logTName=yes.
log=System.out	Directs logging to the Java output standard, System.out.

Attribute	Description
logIS={ yes | no | nosingleread }	Specifies whether DataDirect Spy logs activity on InputStream and Reader objects. When logIS=nosingleread, logging on InputStream and Reader objects is active; however, logging of the single-byte read InputStream.read or single-character Reader.read is suppressed to prevent generating large log files that contain single-byte or single character read messages. The default is no.
logLobs={ yes | no }	Specifies whether DataDirect Spy logs activity on BLOB and CLOB objects.
logTName={ yes | no }	Specifies whether DataDirect Spy logs the name of the current thread. The default is no.
timestamp={ yes | no }	Specifies whether a timestamp is included on each line of the DataDirect Spy log. The default is no.

Example

The following value instructs the driver to log all JDBC activity to a file using a maximum of 80 characters for each line.

```
(log=(file)/tmp/spy.log;linelimit=80)
```

Notes

- If coding a path on Windows to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: log=(file)C:\\temp\\spy.log.
- For more information, refer to "Tracking JDBC calls with DataDirect Spy" in the *Progress DataDirect for JDBC Drivers Reference*.

Data Source Methods

```
public String getSpyAttributes()
```

```
public void setSpyAttributes(String)
```

Default Value

No default value

Data Type

String

StmtCallLimit

Purpose

Specifies the maximum number of web service calls the driver can make when executing any single SQL statement or metadata query.

Valid Values

`0` | *x*

where:

x is a positive integer that defines the maximum number of web service calls up to 2147483647 the driver can make when executing any single SQL statement or metadata query.

Behavior

If set to `0`, there is no limit.

If set to *x*, the driver uses this value to set the maximum number of web service calls on a single connection that can be made when executing a SQL statement. This limit can be overridden by changing the `STMT_CALL_LIMIT` session attribute using the `ALTER SESSION` statement. For example, the following statement sets the statement call limit to 10 web service calls:

```
ALTER SESSION SET STMT_CALL_LIMIT=10
```

If the web service call limit is exceeded, the behavior of the driver depends on the value specified for the `StmtCallLimitBehavior` property.

Data Source Methods

```
public Integer getStmtCallLimit()
```

```
public void setStmtCallLimit(Integer)
```

Default Value

`0`

Data Type

Integer

StmtCallLimitBehavior

Purpose

Specifies the behavior of the driver when the maximum web service call limit specified by the StmtCallLimit property is exceeded.

Valid Values

`ReturnResults | ErrorAlways`

Behavior

If set to `ReturnResults`, the driver returns any partial results it received prior to the call limit being exceeded. The driver generates a warning that not all of the results were fetched.

If set to `ErrorAlways`, the driver generates an exception if the maximum web service call limit is exceeded.

Data Source Methods

```
public String getStmtCallLimitBehavior()
```

```
public void setStmtCallLimitBehavior(String)
```

Default Value

`ErrorAlways`

Data Type

String

User

Purpose

Specifies the user name that is used to connect to the service.

Valid Values

`String`

where:

`String`

is a valid user name. The user name is case-insensitive.

Data Source Methods

```
public String getUser()
```

```
public void setUser(String)
```

Default Value

No default value

Data Type

String

See also

Password on page 54

Basic authentication on page 43

WSRetryCount

Purpose

Specifies the number of times the driver retries a timed-out Select request. The timeout period is specified by the WSTimeout connection property.

Valid Values

`0` | *x*

where:

x

is a positive integer

Behavior

If set to `0`, the driver does not retry timed-out requests after the initial unsuccessful attempt.

If set to *x*, the driver retries the timed-out request the specified number of times.

Data Source Methods

```
public Integer getWSRetryCount()
```

```
public void setWSRetryCount(Integer)
```

Default Value

5

Data Type

Integer

See also

WSTimeout on page 64

WSTimeout

Purpose

Specifies the time, in seconds, that the driver waits for a response to a web service request.

Valid Values

`0` | *x*

where:

x

is a positive integer that defines the number of seconds the driver waits for a response to a web service request.

Behavior

If set to `0`, the driver waits indefinitely for a response; there is no timeout.

If set to *x*, the driver uses the value as the default timeout, measured in seconds, for any statement created by the connection.

If a Select request times out and WSRetryCount is set to retry timed-out requests, the driver retries the request the specified number of times.

Data Source Methods

```
public Integer getWSTimeout()
```

```
public void setWSTimeout(Integer)
```

Default Value

`120`

Data Type

Integer

See also

WSRetryCount on page 64

5

Supported SQL statements and extensions

The driver provides support for the SQL statements and the SQL extensions described in this section. SQL extensions are denoted by an (EXT) in the topic title.

For details, see the following topics:

- Alter Session (EXT)
- Explain Plan
- Select
- Subqueries
- SQL expressions

Alter Session (EXT)

Purpose

Changes various attributes of a local or remote session. A local session maintains the state of the overall connection. A remote session maintains the state that pertains to a particular remote data source connection.

Syntax

`ALTER SESSION SET` *`attribute_name=value`*

where:

attribute_name

specifies the name of the attribute to be changed. Attributes apply to either local or remote sessions.

value

specifies the value for that attribute.

The following table lists the local and remote session attributes, and provides descriptions of each.

Table 4: Alter Session Attributes

Attribute Name	Session Type	Description
Current_Schema	Local	Sets the current schema for the local session. The current schema is the schema used when an identifier in a SQL statement is unqualified. The string value must be the name of a schema visible in the local session. For example: `ALTER SESSION SET CURRENT_SCHEMA=TEAMCITY`
Stmt_Call_Limit	Local	Sets the maximum number of Web service calls the driver can make in executing a statement. Setting the Stmt_Call_Limit attribute has the same effect as setting the Statement Call Limit connection option. It sets the default Web service call limit used by any statement on the connection. Executing this command on a statement overrides the previously set Statement Call Limit for the connection. The value specified must be a positive integer or `0`. The value `0` means that no call limit exists. For example: `ALTER SESSION SET STMT_CALL_LIMIT=150`

Ws_Call_Count	Remote	Resets the Web service call count of a remote session to the value specified. The value must be `0` or a positive integer. WS_Call_Count represents the total number of Web service calls made to the remote data source instance for the current session. For example:
		`ALTER SESSION SET teamcity.WS_CALL_COUNT=0`
		The current value of WS_Call_Count can be obtained by referring to the System_Remote_Sessions system table (see SYSTEM_REMOTE_SESSIONS Catalog Table for details). For example:
		`SELECT * from information_schema.system_remote_sessions WHERE session_id = cursessionid()`

Explain Plan

Purpose

Retrieves a detailed list of the elements in the execution plan. It generates a result set with a single column named `OPERATION`. The individual elements that comprise the plan are returned as rows in the result set.

Syntax

```
EXPLAIN PLAN FOR {SELECT ...}
```

The returned list of elements includes the indexes used for performing the query and can be used to optimize the query.

Select

Purpose

Use the Select statement to fetch results from one or more tables.

Syntax

```
SELECT select_clause from_clause
[where_clause]
[groupby_clause]
[having_clause]
[{UNION [ALL | DISTINCT] |
 {MINUS [DISTINCT] | EXCEPT [DISTINCT]} |
 INTERSECT [DISTINCT]} select_statement]
[limit_clause]
```

where:

select_clause

specifies the columns from which results are to be returned by the query. See "Select clause" for a complete explanation.

from_clause

specifies one or more tables on which the other clauses in the query operate. See "From clause" for a complete explanation.

where_clause

is optional and restricts the results that are returned by the query. See "Where clause" for a complete explanation.

groupby_clause

is optional and allows query results to be aggregated in terms of groups. See "Group By clause" for a complete explanation.

having_clause

is optional and specifies conditions for groups of rows (for example, display only the departments that have salaries totaling more than $200,000). See "Having clause" for a complete explanation.

UNION

is an optional operator that combines the results of the left and right Select statements into a single result. See "Union operator" for a complete explanation.

`INTERSECT`

is an optional operator that returns a single result by keeping any distinct values from the results of the left and right Select statements. See "Intersect operator" for a complete explanation.

`EXCEPT | MINUS`

are synonymous optional operators that returns a single result by taking the results of the left Select statement and removing the results of the right Select statement. See "Except and Minus operators" for a complete explanation.

`orderby_clause`

is optional and sorts the results that are returned by the query. See "Order By clause" for a complete explanation.

`limit_clause`

is optional and places an upper bound on the number of rows returned in the result. See "Limit clause" for a complete explanation.

Select clause

Purpose

Use the Select clause to specify with a list of column expressions that identify columns of values that you want to retrieve or an asterisk (*) to retrieve the value of all columns.

Syntax

```
SELECT [{LIMIT offsetnumber | TOP number}] [ALL | DISTINCT] {* | column_expression
[[AS] column_alias] [,column_expression [[AS] column_alias], ...]}
```

where:

`LIMIT` *`offset number`*

creates the result set for the Select statement first and then discards the first number of rows specified by *`offset`* and returns the number of remaining rows specified by *`number`*. To not discard any of the rows, specify `0` for *`offset`*, for example, `LIMIT 0` *`number`*. To discard the first *`offset`* number of rows and return all the remaining rows, specify `0` for *`number`*, for example, `LIMIT` *`offset`*`0`.

`TOP` *`number`*

is equivalent to `LIMIT 0`*`number`*.

`column_expression`

can be simply a column name (for example, `last_name`). More complex expressions may include mathematical operations or string manipulation (for example, `salary * 1.05`). See "SQL expressions" for details. *`column_expression`* can also include aggregate functions. See "Aggregate functions" for details.

column_alias

can be used to give the column a descriptive name. For example, to assign the alias department to the column dep:

```
SELECT dep AS department FROM emp
```

DISTINCT

eliminates duplicate rows from the result of a query. This operator can precede the first column expression. For example:

```
SELECT DISTINCT dep FROM emp
```

Notes

- Separate multiple column expressions with commas (for example, `SELECT last_name, first_name, hire_date`).
- Column names can be prefixed with the table name or table alias. For example, `SELECT emp.last_name or e.last_name`, where `e` is the alias for the table `emp`.
- NULL values are not treated as distinct from each other. The default behavior is that all result rows be returned, which can be made explicit with the keyword `ALL`.

See also

SQL expressions on page 81

Aggregate functions

Aggregate functions can also be a part of a Select clause. Aggregate functions return a single value from a set of rows. An aggregate can be used with a column name (for example, `AVG(salary)`) or in combination with a more complex column expression (for example, `AVG(salary * 1.07)`).

The following table lists supported aggregate functions.

Note: Doubly nested aggregates, such as `SUM(COUNT(col1))`, are currently not permitted by the driver.

Table 5: Aggregate Functions

Aggregate	Returns
AVG	The average of the values in a numeric column expression. For example, `AVG(salary)` returns the average of all salary column values.
COUNT	The number of values in any field expression. For example, `COUNT(name)` returns the number of name values. When using `COUNT` with a field name, `COUNT` returns the number of non-NULL column values. A special example is `COUNT(*)`, which returns the number of rows in the set, including rows with NULL values. **Note:** The driver does not support `COUNT(DISTINCT *)`. For example, `SELECT COUNT(DISTINCT *) FROM mytable` results in a syntax error.

MAX	The maximum value in any column expression. For example, `MAX(salary)` returns the maximum salary column value.
MIN	The minimum value in any column expression. For example, `MIN(salary)` returns the minimum salary column value.
SUM	The total of the values in a numeric column expression. For example, `SUM(salary)` returns the sum of all salary column values.

Example

The following example uses the `COUNT`, `MAX`, and `AVG` aggregate functions:

```
SELECT
     COUNT(amount) AS numOpportunities,
     MAX(amount) AS maxAmount,
     AVG(amount) AS avgAmount
FROM opportunity o INNER JOIN user u
     ON o.ownerId = u.id
WHERE o.isClosed = 'false' AND
     u.name = 'MyName'
```

From clause

Purpose

The From clause indicates the tables to be used in the Select statement.

Syntax

`FROM` *`table_name`* `[`*`table_alias`*`] [,...]`

where:

`table_name`

is the name of a table or a subquery. Multiple tables define an implicit inner join among those tables. Multiple table names must be separated by a comma. For example:

```
SELECT * FROM emp, dep
```

Subqueries can be used instead of table names. Subqueries must be enclosed in parentheses. See "Subquery in a From clause" for an example.

`table_alias`

is a name used to refer to a table in the rest of the Select statement. When you specify an alias for a table, you can prefix all column names of that table with the table alias.

Example

The following example specifies two table aliases, `e` for `emp` and `d` for `dep`:

```
SELECT e.name, d.deptName
FROM emp e, dep d
WHERE e.deptId = d.id
```

table_alias is a name used to refer to a table in the rest of the Select statement. When you specify an alias for a table, you can prefix all column names of that table with the table alias. For example, given the table specification:

```
FROM emp E
```

you may refer to the last_name field as E.last_name. Table aliases must be used if the Select statement joins a table to itself. For example:

```
SELECT * FROM emp E, emp F WHERE E.mgr_id = F.emp_id
```

The equal sign (=) includes only matching rows in the results.

Join in a From clause

Purpose

You can use a Join as a way to associate multiple tables within a Select statement. Joins may be either explicit or implicit. For example, the following is the example from the previous section restated as an explicit inner join:

```
SELECT * FROM emp INNER JOIN dep ON id=empId
SELECT e.name, d.deptName
FROM emp e INNER JOIN dep d ON e.deptId = d.id;
```

whereas the following is the same statement as an implicit inner join:

```
SELECT * FROM emp, dep WHERE emp.deptID=dep.id
```

Note: The ON clause in a join expression must evaluate to a true or false value.

Syntax

```
FROM table_name {RIGHT OUTER | INNER | LEFT OUTER | CROSS | FULL OUTER} JOIN table.key
 ON search-condition
```

Example

In this example, two tables are joined using LEFT OUTER JOIN. T1, the first table named includes nonmatching rows.

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.key = T2.key
```

If you use a CROSS JOIN, no ON expression is allowed for the join.

Subquery in a From clause

Subqueries can be used in the From clause in place of table references (*table_name*).

Example

```
SELECT * FROM (SELECT * FROM emp WHERE sal > 10000) new_emp, dept WHERE
new_emp.deptno = dept.deptno
```

See also

Subqueries on page 79

Where clause

Purpose

Specifies the conditions that rows must meet to be retrieved.

Syntax

```
WHERE expr1 rel_operator expr2
```

where:

expr1

is either a column name, literal, or expression.

expr2

is either a column name, literal, expression, or subquery. Subqueries must be enclosed in parentheses.

rel_operator

is the relational operator that links the two expressions.

Example

The following Select statement retrieves the first and last names of employees that make at least $20,000.

```
SELECT last_name, first_name FROM emp WHERE salary >= 20000
```

See also

SQL expressions on page 81
Subqueries on page 79

Group By clause

Purpose

Specifies the names of one or more columns by which the returned values are grouped. This clause is used to return a set of aggregate values.

Syntax

```
GROUP BY column_expression [,...]
```

where:

column_expression

is either a column name or a SQL expression. Multiple values must be separated by a comma. If *column_expression* is a column name, it must match one of the column names specified in the Select clause. Also, the Group By clause must include all non-aggregate columns specified in the Select list.

Example

The following example totals the salaries in each department:

```
SELECT dept_id, sum(salary) FROM emp GROUP BY dept_id
```

This statement returns one row for each distinct department ID. Each row contains the department ID and the sum of the salaries of the employees in the department.

See also

SQL expressions on page 81
Subqueries on page 79

Having clause

Purpose

Specifies conditions for groups of rows (for example, display only the departments that have salaries totaling more than $200,000). This clause is valid only if you have already defined a Group By clause.

Syntax

HAVING *expr1 rel_operator expr2*

where:

expr1 | expr2

can be column names, constant values, or expressions. These expressions do not have to match a column expression in the Select clause. See "SQL expressions" for details regarding SQL expressions.

rel_operator

is the relational operator that links the two expressions.

Example

The following example returns only the departments that have salaries totaling more than $200,000:

```
SELECT dept_id, sum(salary) FROM emp GROUP BY dept_id HAVING sum(salary) > 200000
```

See also

SQL expressions on page 81
Subqueries on page 79

Union operator

Purpose

Combines the results of two Select statements into a single result. The single result is all the returned rows from both Select statements. By default, duplicate rows are not returned. To return duplicate rows, use the All keyword (`UNION ALL`).

Syntax

```
select_statement
UNION [ALL | DISTINCT] | {MINUS [DISTINCT] | EXCEPT [DISTINCT]} | INTERSECT
[DISTINCT]select_statement
```

Notes

- When using the Union operator, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
UNION
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
UNION
SELECT salary, last_name FROM raises
```

Intersect operator

Purpose

Intersect operator returns a single result set. The result set contains rows that are returned by both Select statements. Duplicates are returned unless the Distinct operator is added.

Syntax

```
select_statement
INTERSECT [DISTINCT]
select_statement
```

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

- When using the Intersect operator, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
INTERSECT [DISTINCT]
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
INTERSECT
SELECT salary, last_name FROM raises
```

Except and Minus operators

Purpose

Return the rows from the left Select statement that are not included in the result of the right Select statement.

Syntax

```
select_statement
{EXCEPT [DISTINCT] | MINUS [DISTINCT]}
select_statement
```

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

- When using one of these operators, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
EXCEPT
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
EXCEPT
SELECT salary, last_name FROM raises
```

Order By clause

Purpose

The Order By clause specifies how the rows are to be sorted.

Syntax

`ORDER BY` *`sort_expression`* `[DESC | ASC] [,...]`

where:

`sort_expression`

is either the name of a column, a column alias, a SQL expression, or the positioned number of the column or expression in the select list to use.

The default is to perform an ascending (ASC) sort.

Example

To sort by `last_name` and then by `first_name`, you could use either of the following Select statements:

```
SELECT emp_id, last_name, first_name FROM emp
ORDER BY last_name, first_name
```

or

```
SELECT emp_id, last_name, first_name FROM emp
ORDER BY 2,3
```

In the second example, `last_name` is the second item in the Select list, so `ORDER BY 2,3` sorts by `last_name` and then by `first_name`.

See also

SQL expressions on page 81

Limit clause

Purpose

Places an upper bound on the number of rows returned in the result.

Syntax

`LIMIT` *`number_of_rows`* `[OFFSET` *`offset_number`*`]`

where:

number_of_rows

specifies a maximum number of rows in the result. A negative number indicates no upper bound.

OFFSET

specifies how many rows to skip at the beginning of the result set. *offset_number* is the number of rows to skip.

Notes

- In a compound query, the Limit clause can appear only on the final Select statement. The limit is applied to the entire query, not to the individual Select statement to which it is attached.

Example

The following example returns a maximum of 20 rows.

```
SELECT last_name, first_name FROM emp WHERE salary > 20000 ORDER BY dept_idc LIMIT
20
```

Subqueries

A query is an operation that retrieves data from one or more tables or views. In this reference, a top-level query is called a Select statement, and a query nested within a Select statement is called a subquery.

A subquery is a query expression that appears in the body of another expression such as a Select, an Update, or a Delete statement. In the following example, the second Select statement is a subquery:

```
SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM dept)
```

IN predicate

Purpose

The In predicate specifies a set of values against which to compare a result set. If the values are being compared against a subquery, only a single column result set is returned.

Syntax

value [NOT] IN (*value1*, *value2*,...)

OR

value [NOT] IN (*subquery*)

Example

```
SELECT * FROM emp WHERE deptno IN
(SELECT deptno FROM dept WHERE dname <> 'Sales')
```

EXISTS predicate

Purpose

The Exists predicate is true only if the cardinality of the subquery is greater than 0; otherwise, it is false.

Syntax

```
EXISTS (subquery)
```

Example

```
SELECT empno, ename, deptno FROM emp e WHERE EXISTS
(SELECT deptno FROM dept WHERE e.deptno = dept.deptno)
```

UNIQUE predicate

Purpose

The Unique predicate is used to determine whether duplicate rows exist in a virtual table (one returned from a subquery).

Syntax

```
UNIQUE (subquery)
```

Example

```
SELECT * FROM dept d WHERE UNIQUE
(SELECT deptno FROM emp e WHERE e.deptno = d.deptno)
```

Correlated subqueries

Purpose

A correlated subquery is a subquery that references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent Select statement.

A correlated subquery answers a multiple-part question in which the answer depends on the value in each row processed by the parent statement. For example, you can use a correlated subquery to determine which employees earn more than the average salaries for their departments. In this case, the correlated subquery specifically computes the average salary for each department.

Syntax

```
SELECT select_list
   FROM table1 t_alias1
   WHERE expr rel_operator
   (SELECT column_list
       FROM table2 t_alias2
       WHERE t_alias1.columnrel_operatort_alias2.column)
```

Notes

- Correlated column names in correlated subqueries must be explicitly qualified with the table name of the parent.

Example A

The following statement returns data about employees whose salaries exceed their department average. This statement assigns an alias to emp, the table containing the salary information, and then uses the alias in a correlated subquery:

```
SELECT deptno, ename, sal FROM emp x WHERE sal >
  (SELECT AVG(sal) FROM emp WHERE x.deptno = deptno)
  ORDER BY deptno
```

Example B

This is an example of a correlated subquery that returns row values:

```
SELECT * FROM dept "outer" WHERE 'manager' IN
  (SELECT managername FROM emp
  WHERE "outer".deptno = emp.deptno)
```

Example C

This is an example of finding the department number (deptno) with multiple employees:

```
SELECT * FROM dept main WHERE 1 <
  (SELECT COUNT(*) FROM emp WHERE deptno = main.deptno)
```

Example D

This is an example of correlating a table with itself:

```
SELECT deptno, ename, sal FROM emp x WHERE sal >
  (SELECT AVG(sal) FROM emp WHERE x.deptno = deptno)
```

SQL expressions

An expression is a combination of one or more values, operators, and SQL functions that evaluate to a value. You can use expressions in the Where, and Having of Select statements; and in the Set clauses of Update statements.

Expressions enable you to use mathematical operations as well as character string manipulation operators to form complex queries.

The driver supports both unquoted and quoted identifiers. An unquoted identifier must start with an ASCII alpha character and can be followed by zero

Quoted identifiers must be enclosed in double quotation marks (""). A quoted identifier can contain any Unicode character including the space character. The driver recognizes the Unicode escape sequence \uxxxx as a Unicode character. You can specify a double quotation mark in a quoted identifier by escaping it with a double quotation mark.

The maximum length of both quoted and unquoted identifiers is 128 characters.

Valid expression elements are:

- Column names
- Literals
- Operators
- Functions

Column names

The most common expression is a simple column name. You can combine a column name with other expression elements.

Literals

Literals are fixed data values. For example, in the expression `PRICE * 1.05`, the value 1.05 is a constant. Literals are classified into types, including the following:

- Binary
- Character string
- Date
- Floating point
- Integer
- Numeric
- Time
- Timestamp

The following table describes the literal format for supported SQL data types.

Table 6: Literal Syntax Examples

SQL Type	Literal Syntax	Example
BIGINT	*n* where *n* is any valid integer value in the range of the INTEGER data type	`12 or -34 or 0`
BOOLEAN	Min Value: 0 Max Value: 1	`0` `1`
DATE	DATE'*date*'	`'2010-05-21'`
DATETIME	TIMESTAMP'*ts*'	`'2010-05-21 18:33:05.025'`

SQL Type	Literal Syntax	Example

DECIMAL	*n.f*
	where:
	n
	is the integral part
	f
	is the fractional part
		`0.25`
		`3.1415`
		`-7.48`

DOUBLE	*n.f*E*x* where: *n* is the integral part *f* is the fractional part *x* is the exponent	`1.2E0` or `2.5E40` or `-3.45E2` or `5.67E-4`
INTEGER	*n* where *n* is a valid integer value in the range of the INTEGER data type	`12` or `-34` or `0`
LONGVARBINARY	'*hex_value*'	`'000482ff'`
LONGVARCHAR	'*value*'	`'This is a string literal'`
TIME	TIME'*time*'	`'2010-05-21 18:33:05.025'`
VARCHAR	'*value*'	`'This is a string literal'`

Character string literals

Text specifies a character string literal. A character string literal must be enclosed in single quotation marks. To represent one single quotation mark within a literal, you must enter two single quotation marks. When the data in the fields is returned to the client, trailing blanks are stripped.

A character string literal can have a maximum length of 32 KB, that is, (32*1024) bytes.

Example

```
'Hello'
'Jim''s friend is Joe'
```

Numeric literals

Unquoted numeric values are treated as numeric literals. If the unquoted numeric value contains a decimal point or exponent, it is treated as a real literal; otherwise, it is treated as an integer literal.

Example
```
+1894.1204
```

Binary literals

Binary literals are represented with single quotation marks. The valid characters in a binary literal are 0-9, a-f, and A-F.

Example
```
'00af123d'
```

Date/Time literals

Date and time literal values are enclosed in single quotion marks ('*value*').

- The format for a Date literal is DATE'*date*'.
- The format for a Time literal is TIME'*time*'.
- The format for a Timestamp literal is TIMESTAMP'*ts*'.

Integer literals

Integer literals are represented by a string of numbers that are not enclosed in quotation marks and do not contain decimal points.

Notes

- Integer constants must be whole numbers; they cannot contain decimals.
- Integer literals can start with sign characters (+/-).

Example
```
1994 or -2
```

Operators

This section describes the operators that can be used in SQL expressions.

Note: Numeric operators are restricted to numeric types. Numeric operators do not support non-numeric types.

Unary operator

A unary operator operates on only one operand.

operator operand

Binary operator

A binary operator operates on two operands.

operand1 operator operand2

If an operator is given a null operand, the result is always null. The only operator that does not follow this rule is concatenation (||).

Arithmetic operators

You can use an arithmetic operator in an expression to negate, add, subtract, multiply, and divide numeric values. The result of this operation is also a numeric value. The + and - operators are also supported in date/time fields to allow date arithmetic. The following table lists the supported arithmetic operators.

Table 7: Arithmetic Operators

Operator	Purpose	Example
+ -	Denotes a positive or negative expression. These are unary operators.	`SELECT * FROM emp WHERE comm = -1`
* /	Multiplies, divides. These are binary operators.	`UPDATE emp SET sal = sal + sal * 0.10`
+ -	Adds, subtracts. These are binary operators.	`SELECT sal + comm FROM emp WHERE empno > 100`

Concatenation operator

The concatenation operator manipulates character strings. The following table lists the only supported concatenation operator.

Table 8: Concatenation Operator

Operator	Purpose	Example
||	Concatenates character strings.	`SELECT 'Name is' || ename FROM emp`

The result of concatenating two character strings is the data type VARCHAR.

Comparison operators

Comparison operators compare one expression to another. The result of such a comparison can be TRUE, FALSE, or UNKNOWN (if one of the operands is NULL). The driver considers the UNKNOWN result as FALSE.

The following table lists the supported comparison operators.

Table 9: Comparison Operators

Operator	Purpose	Example
=	Equality test.	`SELECT * FROM emp WHERE sal = 1500`
!=<>	Inequality test.	`SELECT * FROM emp WHERE sal != 1500`

Operator	Purpose	Example
><	"Greater than" and "less than" tests.	`SELECT * FROM emp WHERE sal > 1500 SELECT * FROM emp WHERE sal < 1500`
>=<=	"Greater than or equal to" and "less than or equal to" tests.	`SELECT * FROM emp WHERE sal >= 1500 SELECT * FROM emp WHERE sal <= 1500`
LIKE	% and _ wildcards can be used to search for a pattern in a column. The percent sign denotes zero, one, or multiple characters, while the underscore denotes a single character. The right-hand side of a LIKE expression must evaluate to a string or binary.	`SELECT * FROM emp WHERE ENAME LIKE 'J%'`

ESCAPE clause in LIKE operator
LIKE 'pattern string' ESCAPE 'c'
	The Escape clause is supported in the LIKE predicate to indicate the escape character. Escape characters are used in the pattern string to indicate that any wildcard character that is after the escape character in the pattern string should be treated as a regular character.
	The default escape character is backslash (\).
		`SELECT * FROM emp WHERE ENAME LIKE 'J%\_%' ESCAPE '\'`
		This matches all records with names that start with letter 'J' and have the '_' character in them.
		`SELECT * FROM emp WHERE ENAME LIKE 'JOE\_JOHN' ESCAPE '\'`
		This matches only records with name 'JOE_JOHN'.

[NOT] IN	"Equal to any member of" test.	`SELECT * FROM emp WHERE job IN ('CLERK','ANALYST') SELECT * FROM emp WHERE sal IN (SELECT sal FROM emp WHERE deptno = 30)`
[NOT] BETWEEN x AND y	"Greater than or equal to x" and "less than or equal to y."	`SELECT * FROM emp WHERE sal BETWEEN 2000 AND 3000`
EXISTS	Tests for existence of rows in a subquery.	`SELECT empno, ename, deptno FROM emp e WHERE EXISTS (SELECT deptno FROM dept WHERE e.deptno = dept.deptno)`
IS [NOT] NULL	Tests whether the value of the column or expression is NULL.	`SELECT * FROM emp WHERE ename IS NOT NULL SELECT * FROM emp WHERE ename IS NULL`

Logical operators

A logical operator combines the results of two component conditions to produce a single result or to invert the result of a single condition. The following table lists the supported logical operators.

Table 10: Logical Operators

Operator	Purpose	Example
NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	`SELECT * FROM emp WHERE NOT (job IS NULL)` `SELECT * FROM emp WHERE NOT (sal BETWEEN 1000 AND 2000)`
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE; otherwise, returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' AND deptno = 10`
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE; otherwise, returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' OR deptno = 10`

Example

In the Where clause of the following Select statement, the AND logical operator is used to ensure that managers earning more than $1000 a month are returned in the result:

```
SELECT * FROM emp WHERE jobtitle = manager AND sal > 1000
```

Operator precedence

As expressions become more complex, the order in which the expressions are evaluated becomes important. The following table shows the order in which the operators are evaluated. The operators in the first line are evaluated first, then those in the second line, and so on. Operators in the same line are evaluated left to right in the expression.You can change the order of precedence by using parentheses. Enclosing expressions in parentheses forces them to be evaluated together.

Table 11: Operator Precedence

Precedence	Operator
1	+ (Positive), - (Negative)
2	*(Multiply), / (Division)
3	+ (Add), - (Subtract)
4	|| (Concatenate)
5	=, >, <, >=, <=, <>, != (Comparison operators)
6	NOT, IN, LIKE

Precedence	Operator
7	AND
8	OR

Example A

The query in the following example returns employee records for which the department number is 1 or 2 and the salary is greater than $1000:

```
SELECT * FROM emp WHERE (deptno = 1 OR deptno = 2) AND sal > 1000
```

Because parenthetical expressions are forced to be evaluated first, the OR operation takes precedence over AND.

Example B

In the following example, the query returns records for all the employees in department 1, but only employees whose salary is greater than $1000 in department 2.

```
SELECT * FROM emp WHERE deptno = 1 OR deptno = 2 AND sal > 1000
```

The AND operator takes precedence over OR, so that the search condition in the example is equivalent to the expression `deptno = 1 OR (deptno = 2 AND sal > 1000)`.

Functions

The driver supports a number of functions that you can use in expressions, including String, Numeric, Timedate, and System functions.

Refer to "Scalar functions" in the *Progress DataDirect for JDBC Drivers Reference* for more information.

Conditions

A condition specifies a combination of one or more expressions and logical operators that evaluates to either TRUE, FALSE, or UNKNOWN. You can use a condition in the Where clause of the Delete, Select, and Update statements; and in the Having clauses of Select statements. The following describes supported conditions.

Table 12: Conditions

Condition	Description
Simple comparison	Specifies a comparison with expressions or subquery results. `= , !=, <>, < , >, <=, <=`
Group comparison	Specifies a comparison with any or all members in a list or subquery. `[= , !=, <>, < , >, <=, <=] [ANY, ALL, SOME]`

Condition	Description
Membership	Tests for membership in a list or subquery. `[NOT] IN`
Range	Tests for inclusion in a range. `[NOT] BETWEEN`
NULL	Tests for nulls. `IS NULL, IS NOT NULL`
EXISTS	Tests for existence of rows in a subquery. `[NOT] EXISTS`
LIKE	Specifies a test involving pattern matching. `[NOT] LIKE`
Compound	Specifies a combination of other conditions. `CONDITION [AND/OR] CONDITION`

6

Introduction to the TeamCity data model

The TeamCity data model is defined using a collection of standard JSON documents that contain the data, identifiers, and object relationships for a given service. These documents are stored on URL endpoints that are accessible using sets of proprietary REST API calls. To expose TeamCity resources to SQL applications, the driver maps TeamCity endpoints to a set of relational parent and child tables. The following sections describe the relational tables exposed by the driver along with their corresponding TeamCity REST API call.

For details, see the following topics:

- BUILDQUEUEREVISIONS
- BUILDQUEUES
- BUILDQUEUESNAPSHOTDEPENDENCIES
- BUILDREVISIONS
- BUILDS
- BUILDSNAPSHOTDEPENDENCIES
- BUILDSTATISTICS
- BUILDTEMPLATES
- BUILDTYPES
- FLAKYTESTS
- LASTCHANGES
- MUTES
- PROJECTDETAILS
- PROJECTFEATURES
- PROJECTS
- PROPERTY
- REVISIONS
- SCOPEBUILDTYPES
- SNAPSHOTDEPENDENCIESBUILD
- SUBPROJECTS
- TARGETTESTS
- TESTFAILUREDETAILS
- TESTFAILURES
- USERDETAILS
- USERGROUPS
- USERPROPERTIES
- USERS

AGENTDETAILS

Endpoint

<*hostname*>app/rest/agents/id:{agentId}

Columns

The AGENTDETAILS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
AGENTID*	Integer
AUTHCOMMENTTIMESTAMP	Timestamp(3)
AUTHORIZED	Boolean
AUTHORIZEDHREF	VarChar(2000)
AUTHORIZEDINFOSTATUS	Boolean
AUTHORIZEDUSERID	Integer
AUTHORIZEDUSERNAME	VarChar(255)
CONNECTED	Boolean
ENABLED	Boolean
ENABLEDCOMMENTTIMESTAMP	Timestamp(3)
ENABLEDHREF	VarChar(2000)
ENABLEDINFOSTATUS	Boolean
ENABLEDUSERFULLNAME	VarChar(255)
ENABLEDUSERID	Integer
ENABLEDUSERNAME	VarChar(255)
IP	VarChar(255)
POOLHREF	VarChar(255)
POOLID	Integer
POOLNAME	VarChar(255)
UPTODATE	Boolean

AGENTS

Endpoint

<*hostname*>app/rest/agents

Columns

The AGENTS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
AGENTID*	Integer
HREF	VarChar(2000)
NAME	VarChar(2000)
TYPEID	Integer
WEBURL	VarChar(2000)

BUILDARTIFACTDEPENDENCIES

Endpoint

<*hostname*>app/rest/builds/id:{buildId}

Parent Table

BUILDDETAILS

Columns

The BUILDARTIFACTDEPENDENCIES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDDETAILS_BUILDID*	Integer	References: BUILDDETAILS.BUILDID
POSITION*	Integer	
BUILDBRANCHNAME	VarChar(255)	
BUILDNUMBER	Integer	
BUILDSTATE	VarChar(255)	

Column Name	Data Type	Notes
BUILDSTATUS	VarChar(255)	
BUILDTYPEID	VarChar(255)	
DEFAULTBRANCH	Boolean	
HREF	VarChar(2000)	
ID	Integer	
WEBURL	VarChar(2000)	

BUILDARTIFACTS

Endpoint

<*hostname*>app/rest/builds/{buildId}/artifacts

Columns

The BUILDARTIFACTS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
BUILDID*	Integer
CHILDRENHREF	VarChar(2000)
CONTENTHREF	VarChar(2000)
HREF	VarChar(2000)
MODIFICATIONTIME	VarChar(255)
NAME	VarChar(255)
SIZE	Integer

BUILDCONFIGURATIONS

Endpoint

<*hostname*>app/rest/buildTypes/{buildTypeId}/builds

Columns

The BUILDCONFIGURATIONS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
BUILDTYPEID*	VarChar(2000)
BUILDCONFIGURATIONBUILDTYPEID	VarChar(2000)
COUNT_	Integer
HREF	VarChar(2000)
NUMBER	Integer
STATE	VarChar(255)
STATUS	VarChar(255)
WEBURL	VarChar(2000)

BUILDDETAILS

Endpoint

<*hostname*>app/rest/builds/id:{buildId}

Columns

The BUILDDETAILS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
BUILDID*	Integer
AGENTHREF	VarChar(2000)
AGENTID	Integer
AGENTNAME	VarChar(255)
AGENTTYPEID	Integer
AGENTWEBURL	VarChar(2000)
ARTIFACTDEPENDENCIESCOUNT	Integer
BUILDTYPEHREF	VarChar(2000)

Column Name	Data Type
BUILDTYPEID	VarChar(255)
BUILDTYPENAME	VarChar(255)
BUILDTYPEPROJECTID	VarChar(255)
BUILDTYPEPROJECTNAME	VarChar(255)
BUILDTYPEWEBURL	VarChar(2000)
CHANGESHREF	VarChar(2000)
FINISHDATE	Timestamp(3)
LASTCHANGESCOUNT	Integer
PROBLEMOCCURRENCESCOUNT	Integer
PROBLEMOCCURRENCESHREF	VarChar(2000)
QUEUEDDATE	Timestamp(3)
REVISIONCOUNT	Integer
SNAPSHOTDEPENDENCIESCOUNT	Integer
STARTDATE	Timestamp(3)
STATUSTEXT	VarChar(255)
TRIGGEREDDATE	Timestamp(3)
TRIGGEREDTYPE	VarChar(255)

BUILDFAILUREDETAILS

Endpoint

<*hostname*>app/rest/problemOccurrences/{buildFailureId}

Columns

The BUILDFAILUREDETAILS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
BUILDFAILUREID*	VarChar(255)

Column Name	Data Type
ADDITIONALDATA	VarChar(20000)
BUILDHREF	VarChar(2000)
BUILDID	Integer
BUILDNUMBER	Integer
BUILDSTATE	VarChar(255)
BUILDSTATUS	VarChar(255)
BUILDTYPEID	VarChar(255)
BUILDWEBURL	VarChar(2000)
DETAILS	VarChar(20000)
PROBLEMHREF	VarChar(255)
PROBLEMID	Integer
PROBLEMIDENTITY	VarChar(255)
PROBLEMTYPE	VarChar(255)

BUILDFAILURES

Endpoint

<*hostname*>app/rest/problemOccurrences

Columns

The BUILDFAILURES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
BUILDFAILUREID*	VarChar(255)
CURRENTLYINVESTIGATED	Boolean
CURRENTLYMUTED	Boolean
HREF	VarChar(255)
IDENTITY	VarChar(255)

Column Name	Data Type
PROJECTID	VarChar(255)
TYPE	VarChar(255)

BUILDLASTCHANGES

Endpoint

<*hostname*>app/rest/builds/id:{buildId}

Parent Table

BUILDDETAILS

Columns

The BUILDLASTCHANGES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDDETAILS_BUILDID*	Integer	References: BUILDDETAILS.BUILDID
POSITION*	Integer	
CHANGEID	Integer	
CHANGEVERSION	VarChar(255)	
DATE	Timestamp(3)	
HREF	VarChar(2000)	
USERNAME	VarChar(255)	
WEBURL	VarChar(2000)	

BUILDQUEUE

Endpoint

<*hostname*>app/rest/buildQueue

Columns

The BUILDQUEUE table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
BUILDQUEUEID*	Integer
ARTIFACTSHREF	VarChar(2000)
BRANCHNAME	VarChar(255)
BUILDTYPE_BUILDTYPEID	VarChar(255)
BUILDTYPEDESCRIPTION	VarChar(2000)
BUILDTYPEHREF	VarChar(2000)
BUILDTYPEID	VarChar(255)
BUILDTYPENAME	VarChar(255)
BUILDTYPEPROJECTID	VarChar(255)
BUILDTYPEPROJECTNAME	VarChar(2000)
BUILDTYPEWEBURL	VarChar(2000)
CHANGESHREF	VarChar(2000)
COMPATIBLEAGENTSHREF	VarChar(2000)
DEFAULTBRANCH	Boolean
HREF	VarChar(2000)
LASTCHANGESCOUNT	Integer
PROPERTIESCOUNT	Integer
QUEUEDDATE	VarChar(255)
REVISIONSCOUNT	Integer
SNAPSHOTDEPENDENCIESCOUNT	Integer
STATE	VarChar(255)
TRIGGEREDBUILDHREF	VarChar(2000)
TRIGGEREDBUILDID	Integer
TRIGGEREDBUILDSTATE	VarChar(255)

Column Name	Data Type
TRIGGEREDBUILDTYPE	VarChar(255)
TRIGGEREDBUILDTYPEDESCRIPTION	VarChar(2000)
TRIGGEREDBUILDTYPEHREF	VarChar(2000)
TRIGGEREDBUILDTYPEID	VarChar(255)
TRIGGEREDBUILDTYPENAME	VarChar(2000)
TRIGGEREDBUILDTYPEPROJECTID	VarChar(255)
TRIGGEREDBUILDTYPEPROJECTNAME	VarChar(2000)
TRIGGEREDBUILDTYPEWEBURL	VarChar(2000)
TRIGGEREDBUILDWEBURL	VarChar(2000)
TRIGGEREDDATE	VarChar(255)
TRIGGEREDTYPE	VarChar(255)
WAITREASON	VarChar(2000)
WEBURL	VarChar(2000)

BUILDQUEUEDETAILS

Endpoint

<*hostname*>app/rest/buildQueue/id:{buildQueueId}

Columns

The BUILDQUEUEDETAILS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
BUILDQUEUEID*	Integer
ARTIFACTSHREF	VarChar(2000)
BUILDTYPEDESCRIPTION	VarChar(2000)
BUILDTYPEHREF	VarChar(2000)
BUILDTYPEID	VarChar(255)

Column Name	Data Type
BUILDTYPENAME	VarChar(255)
BUILDTYPEPROJECTID	VarChar(255)
BUILDTYPEPROJECTNAME	VarChar(2000)
BUILDTYPEWEBURL	VarChar(2000)
CHANGESHREF	VarChar(2000)
COMPATIBLEAGENTSHREF	VarChar(2000)
LASTCHANGESCOUNT	Integer
PROPERTIESCOUNT	Integer
QUEUEDDATE	Timestamp(3)
REVISIONSCOUNT	Integer
SNAPSHOTDEPENDENCIESCOUNT	Integer
TRIGGEREDBUILDHREF	VarChar(2000)
TRIGGEREDBUILDID	Integer
TRIGGEREDBUILDSTATE	VarChar(255)
TRIGGEREDBUILDTYPE	VarChar(255)
TRIGGEREDBUILDTYPEDESCRIPTION	VarChar(2000)
TRIGGEREDBUILDTYPEHREF	VarChar(2000)
TRIGGEREDBUILDTYPEID	VarChar(255)
TRIGGEREDBUILDTYPENAME	VarChar(2000)
TRIGGEREDBUILDTYPEPROJECTID	VarChar(255)
TRIGGEREDBUILDTYPEPROJECTNAME	VarChar(2000)
TRIGGEREDBUILDTYPEWEBURL	VarChar(2000)
TRIGGEREDBUILDWEBURL	VarChar(2000)
TRIGGEREDDATE	Timestamp(3)
TRIGGEREDTYPE	VarChar(255)
WAITREASON	VarChar(2000)

BUILDQUEUELASTCHANGES

Endpoint

<*hostname*>app/rest/buildQueue/id:{buildQueueId}

Parent Table

BUILDQUEUEDETAILS

Columns

The BUILDQUEUELASTCHANGES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDQUEUEDETAILS_BUILDQUEUEID*	Integer	References: BUILDQUEUEDETAILS .BUILDQUEUEID
POSITION*	Integer	
CHANGEID	Integer	
CHANGEVERSION	VarChar(255)	
DATE	Timestamp(3)	
HREF	VarChar(2000)	
USERNAME	VarChar(255)	
WEBURL	VarChar(2000)	

BUILDQUEUEPROPERTY

Endpoint

<*hostname*>app/rest/buildQueue/id:{buildQueueId}

Parent Table

BUILDQUEUEDETAILS

Columns

The BUILDQUEUEPROPERTY table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDQUEUEDETAILS_BUILDQUEUEID*	Integer	References: BUILDQUEUEDETAILS .BUILDQUEUEID
POSITION*	Integer	
INHERITED	Boolean	
NAME	VarChar(255)	
VALUE	VarChar(255)	

BUILDQUEUEREVISIONS

Endpoint

<*hostname*>app/rest/buildQueue/id:{buildQueueId}

Parent Table

BUILDQUEUEDETAILS

Columns

The BUILDQUEUEREVISIONS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDQUEUEDETAILS_BUILDQUEUEID*	Integer	References: BUILDQUEUEDETAILS .BUILDQUEUEID
POSITION*	Integer	
VCSBRANCHNAME	VarChar(255)	
VCSROOTID	VarChar(255)	
VCSROOTINSTANCEHREF	VarChar(2000)	
VCSROOTINSTANCEID	Integer	
VCSROOTINSTANCENAME	VarChar(255)	
VERSION	VarChar(255)	

BUILDQUEUES

Endpoint

<*hostname*>app/rest/buildQueue

Columns

The BUILDQUEUES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
BUILDQUEUEID*	Integer
BUILDTYPEID	VarChar(255)
HREF	VarChar(2000)
PERSONAL	Boolean
STATE	VarChar(255)
WEBURL	VarChar(2000)

BUILDQUEUESNAPSHOTDEPENDENCIES

Endpoint

<*hostname*>app/rest/buildQueue/id:{buildQueueId}

Parent Table

BUILDQUEUEDETAILS

Columns

The BUILDQUEUESNAPSHOTDEPENDENCIES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDQUEUEDETAILS_BUILDQUEUEID*	Integer	References: BUILDQUEUEDETAILS .BUILDQUEUEID
POSITION*	Integer	
BRANCHNAME	VarChar(255)	

Column Name	Data Type	Notes
BUILDNUMBER	Integer	
BUILDSTATE	VarChar(255)	
BUILDSTATUS	VarChar(255)	
BUILDTYPEID	VarChar(255)	
DEFAULTBRANCH	Boolean	
FAILEDTOSTART	Boolean	
HREF	VarChar(2000)	
ID	Integer	
WEBURL	VarChar(2000)	

BUILDREVISIONS

Endpoint

<*hostname*>app/rest/builds/id:{buildId}

Parent Table

BUILDDETAILS

Columns

The BUILDREVISIONS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDDETAILS_BUILDID*	Integer	References: BUILDDETAILS.BUILDID
POSITION*	Integer	
VCSBRANCHNAME	VarChar(255)	
VCSHREF	VarChar(2000)	
VCSID	Integer	
VCSNAME	VarChar(255)	

Column Name	Data Type	Notes
VCSROOTID	VarChar(255)	
VERSION	VarChar(255)	

BUILDS

Endpoint

<*hostname*>app/rest/builds

Columns

The BUILDS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
BUILDID*	Integer
BUILDTYPEID	VarChar(255)
HREF	VarChar(2000)
NUMBER	Integer
STATE	VarChar(255)
STATUS	VarChar(255)
WEBURL	VarChar(2000)

BUILDSNAPSHOTDEPENDENCIES

Endpoint

<*hostname*>app/rest/builds/id:{buildId}

Parent Table

BUILDDETAILS

Columns

The BUILDSNAPSHOTDEPENDENCIES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDDETAILS_BUILDID*	Integer	References: BUILDDETAILS.BUILDID
POSITION*	Integer	
BUILDBRANCHNAME	VarChar(255)	
BUILDNUMBER	Integer	
BUILDSTATE	VarChar(255)	
BUILDSTATUS	VarChar(255)	
BUILDTYPEID	VarChar(255)	
DEFAULTBRANCH	Boolean	
HREF	VarChar(2000)	
ID	Integer	
WEBURL	VarChar(2000)	

BUILDSTATISTICS

Endpoint

<*hostname*>app/rest/builds/{buildId}/statistics

Columns

The BUILDSTATISTICS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
BUILDID*	Integer
NAME	VarChar(255)
VALUE	Double

BUILDTEMPLATES

Endpoint

<*hostname*>app/rest/projects/{projectId}

Parent Table

PROJECTDETAILS

Columns

The BUILDTEMPLATES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
PROJECTDETAILS_PROJECTID*	VarChar(255)	References: PROJECTDETAILS.PROJECTID
POSITION*	Integer	
HREF	VarChar(2000)	
ID	VarChar(255)	
NAME	VarChar(255)	
PROJECTID	VarChar(255)	
PROJECTNAME	VarChar(255)	
TEMPLATEFLAG	Boolean	

BUILDTYPES

Endpoint

<*hostname*>app/rest/buildTypes

Columns

The BUILDTYPES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
BUILDTYPEID*	VarChar(255)
DESCRIPTION	VarChar(2000)

Column Name	Data Type
HREF	VarChar(2000)
NAME	VarChar(2000)
PAUSED	Boolean
PROJECTID	VarChar(255)
PROJECTNAME	VarChar(2000)
WEBURL	VarChar(2000)

FLAKYTESTS

Endpoint

<*hostname*>app/rest/testOccurrences

Columns

The FLAKYTESTS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
PROJECTID*	VarChar(64)
DETAILS	VarChar(20000000)
DURATION	Integer
HREF	VarChar(2000)
NAME	VarChar(255)
STATUS	VarChar(255)
TESTID	VarChar(255)

LASTCHANGES

Endpoint

<*hostname*>app/rest/buildQueue

Parent Table

BUILDQUEUE

Columns

The LASTCHANGES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDQUEUE_BUILDQUEUEID*	Integer	References: BUILDQUEUE.BUILDQUEUEID
POSITION*	Integer	
CHANGEID	Integer	
CHANGEVERSION	VarChar(255)	
DATE	VarChar(255)	
HREF	VarChar(2000)	
USERNAME	VarChar(255)	
WEBURL	VarChar(2000)	

MUTES

Endpoint

<*hostname*>app/rest/mutes

Columns

The MUTES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
MUTEID*	Integer
ASSIGNMENTHREF	VarChar(2000)
ASSIGNMENTID	Integer
ASSIGNMENTNAME	VarChar(255)
ASSIGNMENTTEXT	VarChar(871)
ASSIGNMENTTIMESTAMP	Timestamp(3)
ASSIGNMENTUSERNAME	VarChar(255)

Column Name	Data Type
HREF	VarChar(2000)
RESOLUTIONTYPE	VarChar(13)
SCOPEBUILDTYPESCOUNT	Integer
SCOPEPARENTPROJECTID	VarChar(255)
SCOPEPROJECTHREF	VarChar(2000)
SCOPEPROJECTID	VarChar(255)
SCOPEPROJECTNAME	VarChar(255)
SCOPEPROJECTWEBURL	VarChar(2000)
TARGETTESTSCOUNT	Integer
TARGETTESTSDEFAULT	Boolean

PROJECTDETAILS

Endpoint

<*hostname*>app/rest/projects/{projectId}

Columns

The PROJECTDETAILS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
PROJECTID*	VarChar(255)
BUILDTYPECOUNT	Integer
BUILDTYPES_BUILDTYPES	JSON(16777215)
DESCRIPTION	VarChar(255)
HREF	VarChar(255)
NAME	VarChar(255)
PARAMETERS_PARAMETERSPROPERTY	JSON(16777215)
PARAMETERSCOUNT	Integer

Column Name	Data Type
PARAMETERSHREF	VarChar(2000)
PROJECTFEATURESCOUNT	Integer
PROJECTFEATURESHREF	VarChar(2000)
SUBPROJECTSCOUNT	Integer
TEMPLATESCOUNT	Integer
VCSROOTSCOUNT	Integer
VCSROOTSHREF	VarChar(2000)
WEBURL	VarChar(2000)

PROJECTFEATURES

Endpoint

<*hostname*>app/rest/projects/{projectId}

Parent Table

PROJECTDETAILS

Columns

The PROJECTFEATURES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
PROJECTDETAILS_PROJECTID*	VarChar(255)	References: PROJECTDETAILS.PROJECTID
POSITION*	Integer	
HREF	VarChar(2000)	
ID	VarChar(255)	
TYPE	VarChar(255)	

PROJECTS

Endpoint

<*hostname*>app/rest/projects

Columns

The PROJECTS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
PROJECTID*	VarChar(255)
ARCHIVED	Boolean
DESCRIPTION	VarChar(255)
HREF	VarChar(2000)
NAME	VarChar(255)
PARENTPROJECTID	VarChar(255)
WEBURL	VarChar(2000)

PROPERTY

Endpoint

<*hostname*>app/rest/buildQueue

Parent Table

BUILDQUEUE

Columns

The PROPERTY table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDQUEUE_BUILDQUEUEID*	Integer	References: BUILDQUEUE.BUILDQUEUEID
POSITION*	Integer	
INHERITED	Boolean	

Column Name	Data Type	Notes
NAME	VarChar(255)	
VALUE	VarChar(255)	

REVISIONS

Endpoint

<*hostname*>app/rest/buildQueue

Parent Table

BUILDQUEUE

Columns

The REVISIONS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDQUEUE_BUILDQUEUEID*	Integer	References: BUILDQUEUE.BUILDQUEUEID
POSITION*	Integer	
VCSBRANCHNAME	VarChar(255)	
VCSROOTID	VarChar(255)	
VCSROOTINSTANCEHREF	VarChar(2000)	
VCSROOTINSTANCEID	Integer	
VCSROOTINSTANCENAME	VarChar(255)	
VERSION	VarChar(255)	

SCOPEBUILDTYPES

Endpoint

<*hostname*>app/rest/mutes

Parent Table

MUTES

Columns

The SCOPEBUILDTYPES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
MUTES_MUTEID*	Integer	References: MUTES.MUTEID
POSITION*	Integer	
SCOPEBUILDTYPEHREF	VarChar(2000)	
SCOPEBUILDTYPEID	VarChar(99)	
SCOPEBUILDTYPENAME	VarChar(67)	
SCOPEBUILDTYPEPAUSED	Boolean	
SCOPEBUILDTYPEPROJECTID	VarChar(70)	
SCOPEBUILDTYPEPROJECTNAME	VarChar(136)	
SCOPEBUILDTYPEWEBURL	VarChar(2000)	

SNAPSHOTDEPENDENCIESBUILD

Endpoint

<*hostname*>app/rest/buildQueue

Parent Table

BUILDQUEUE

Columns

The SNAPSHOTDEPENDENCIESBUILD table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
BUILDQUEUE_BUILDQUEUEID*	Integer	References: BUILDQUEUE.BUILDQUEUEID
POSITION*	Integer	
BRANCHNAME	VarChar(255)	
BUILDID	Integer	
BUILDNUMBER	Integer	

Column Name	Data Type	Notes
BUILDSTATE	VarChar(255)	
BUILDSTATUS	VarChar(255)	
BUILDTYPEID	VarChar(255)	
DEFAULTBRANCH	Boolean	
FAILEDTOSTART	Boolean	
HREF	VarChar(2000)	
WEBURL	VarChar(2000)	

SUBPROJECTS

Endpoint

<*hostname*>app/rest/projects/{projectId}

Parent Table

PROJECTDETAILS

Columns

The SUBPROJECTS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
PROJECTDETAILS_PROJECTID*	VarChar(255)	References: PROJECTDETAILS.PROJECTID
POSITION*	Integer	
ARCHIVED	Boolean	
DESCRIPTION	VarChar(255)	
HREF	VarChar(255)	
ID	VarChar(255)	
NAME	VarChar(255)	
PARENTPROJECTID	VarChar(255)	
WEBURL	VarChar(255)	

TARGETTESTS

Endpoint

<*hostname*>app/rest/mutes

Parent Table

MUTES

Columns

The TARGETTESTS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
MUTES_MUTEID*	Integer	References: MUTES.MUTEID
POSITION*	Integer	
TARGETTESTHREF	VarChar(2000)	
TARGETTESTID	BigInt	
TARGETTESTNAME	VarChar(318)	

TESTFAILUREDETAILS

Endpoint

<*hostname*>/app/rest/testOccurrences/{testFailureId}

Columns

The TESTFAILUREDETAILS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
TESTFAILUREID*	VarChar(255)
BUILDHREF	VarChar(2000)
BUILDID	Integer
BUILDNUMBER	Integer
BUILDSTATE	VarChar(255)

Column Name	Data Type
BUILDSTATUS	VarChar(255)
BUILDTYPEID	VarChar(255)
BUILDWEBURL	VarChar(2000)
DETAILS	VarChar(20000)
FAILEDSTATUS	VarChar(255)
FAILEDTESTDURATION	Integer
FAILEDTESTHREF	VarChar(2000)
FAILEDTESTID	VarChar(255)
FAILEDTESTNAME	VarChar(255)
TESTHREF	VarChar(255)
TESTID	BigInt
TESTNAME	VarChar(255)

TESTFAILURES

Endpoint

<*hostname*>app/rest/testOccurrences

Columns

The TESTFAILURES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
TESTFAILUREID*	VarChar(255)
DURATION	Integer
HREF	VarChar(2000)
NAME	VarChar(255)
PROJECTID	VarChar(255)
STATUS	VarChar(255)

USERDETAILS

Endpoint

<*hostname*>app/rest/users/id:{userId}

Columns

The USERDETAILS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
USERID*	Integer
EMAIL	VarChar(2000)
GROUPCOUNT	Integer
LASTLOGIN	VarChar(2000)
PROPERTIESCOUNT	Integer
PROPERTIESHREF	VarChar(2000)
ROLES_USERROLES	JSON(16777215)

USERGROUPS

Endpoint

<*hostname*>app/rest/users/id:{userId}

Parent Table

USERDETAILS

Columns

The USERGROUPS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
USERDETAILS_USERID*	Integer	References: USERDETAILS.USERID
POSITION*	Integer	
GROUPDESCRIPTION	VarChar(2000)	

Column Name	Data Type	Notes
GROUPHREF	VarChar(2000)	
GROUPKEY	VarChar(2000)	
GROUPNAME	VarChar(2000)	

USERPROPERTIES

Endpoint

<*hostname*>app/rest/users/id:{userId}

Parent Table

USERDETAILS

Columns

The USERPROPERTIES table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type	Notes
USERDETAILS_USERID*	Integer	References: USERDETAILS.USERID
POSITION*	Integer	
USERPROPERTYNAME	VarChar(2000)	
USERPROPERTYVALUE	VarChar(2000)	

USERS

Endpoint

<*hostname*>app/rest/users

Columns

The USERS table contains the following columns. Columns marked with an asterisk comprise the primary key.

Column Name	Data Type
USERID*	Integer
FULLNAME	VarChar(43)

Column Name	Data Type
HREF	VarChar(33)
USERNAME	VarChar(30)