Package 'RJDBC'

July 21, 2025

Version 0.2-10

Title Provides Access to Databases Through the JDBC Interface

Author Simon Urbanek <Simon.Urbanek@r-project.org>

Maintainer Simon Urbanek <Simon.Urbanek@r-project.org>

Depends methods, DBI, rJava (>= 0.4-15), R (>= 2.4.0)

Description The RJDBC package is an implementation of R's DBI interface using JDBC as a back-end. This allows R to connect to any DBMS that has a JDBC driver.

License MIT + file LICENSE

URL http://www.rforge.net/RJDBC/

NeedsCompilation no

Repository CRAN

Date/Publication 2022-03-24 16:07:54 UTC

Contents

JDBC *JDBC engine*

Description

JDBC creates a new DBI driver that can be used to start JDBC connections.

findDrivers attempts to find and load all JDBC 4 drivers on the class path.

Usage

```
JDBC (driverClass = "", classPath = "", identifier.quote = NA)
findDrivers(classPath = "", service = "java.sql.Driver", loader = NULL)
```

Arguments

driverClass	name of the Java class of the JDBC driver to load or a one of Java driver instances from findDrivers.
classPath	character vector, class path that needs to be appended in order to load the desired JDBC driver. Usually it is the path to the JAR file containing the driver and all necessary dependencies. It can be a vector and all paths are expanded.
identifier.quote	character to use for quoting identifiers in automatically generated SQL statements or NA if the back-end doesn't support quoted identifiers. See details section below.
service	string, name of the services class (for JDBC always "java.sql.Driver")
loader	Java class loader to use during the look-up or NULL for the default one

Details

JDBC function has two purposes. One is to initialize the Java VM and load a Java JDBC driver (not to be confused with the JDBCDriver R object which is actually a DBI driver). The second purpose is to create a proxy R object which can be used to a call dbConnect which actually creates a connection.

JDBC requires a JDBC driver for a database-backend to be loaded. Usually a JDBC driver is supplied in a Java Archive (jar) file. The path to such a file can be specified in classPath. The driver itself has a Java class name that is used to load the driver (for example the MySQL driver uses com.mysql.jdbc.Driver), this has to be specified in driverClass.

Modern drivers (those supporting JDBC 4) may use Java Service Provider interface for discovery and those can be found using the findDrivers() function which returns a list of drivers. You can pass any of the returned elements as classDriver. Note that the discovery is dynamic, so you can use rJava::.jaddClassPath(...) to add new locations in which Java will look for driver JAR files. However, only drivers providing JSP metadata in their JAR files can be found. JSP was introduced in Java 1.6 so findDrivers() only works on Java 1.6 or higher.

There are currently three different ways to specify drivers:

1. `dbConnect(JDBC("my.Driver.Class", "driver.jar"), ...)` is the most explicit way where the specified driver class is used and expected to be found on the class path. This always works, but the user has to know the full name of the driver class.
2. `dbConnect(JDBC(, "driver.jar"), ...)` omits the driver class which means JDBC will try to find the driver using the `DriverManager`. This *only* works if the JVM has been loaded with the driver when initialized, so this method is discouraged as it is in general very unreliable. The `DriverManager` never updates the list of drivers, so once your driver is not found, there is nothing you can do about it.
3. `dbConnect(JDBC(findDrivers("driver.jar")[[1]]), ...)` uses `findDrivers()` (see details above) to find all available drivers and then passes the needed driver (in this example the first one) to `JDBC()`. You don't need to repeat teh class path in this case as it is already set by `findDrivers()`. It is best to look at the output to see which drivers have been found, but if you pass the list, the first driver is used. Note that if you print the driver you will see the class name so you can also use this information in the first method above instead.

If you have issues loading your driver (e.g., you get ClassNotFound errors), make sure you specify *all* dependencies of your driver, not just the main JAR file. They *all* must be listed on the class path. Also make sure your JVM is supported by the driver, trying to load drivers with too old JVM versions also leads to ClassNotFound errors (as the loader will ignore calsses it cannot load). You can always enable debugging information in the rJava class loader using `.jclassLoader()$setDebug(1L)` for more verbose output that may help in your troubleshooting.

Due to the fact that JDBC can talk to a wide variety of databases, the SQL dialect understood by the database is not known in advance. Therefore the RJDBC implementation tries to adhere to the SQL92 standard, but not all databases are compliant. This affects mainly functions such as `dbWriteTable` that have to automatically generate SQL code. One major ability is the support for quoted identifiers. The SQL92 standard uses double-quotes, but many database engines either don't support it or use other character. The `identifier.quote` parameter allows you to set the proper quote character for the database used. For example MySQL would require `identifier.quote="`"`. If set to `NA`, the ability to quote identifiers is disabled, which poses restrictions on the names that can be used for tables and fields. Other functionality is not affected.

As of RDJBC 0.2-2 JDBC-specific stored procedure calls starting with `{call` and `{?= call` are supported in the statements.

Value

`JDBC` returns a `JDBCDriver` object that can be used in calls to `dbConnect`.

`findDrivers` returns a list of Java object references to instances of JDBC drivers that were found. The list can be empty if no drivers were found. Elements can be used as the `driverClass` in calls to `JDBC`.

See Also

`dbConnect`

Examples

```
## Not run:
drv <- JDBC("com.mysql.jdbc.Driver",
```

```
  "/etc/jdbc/mysql-connector-java-3.1.14-bin.jar", "`")
conn <- dbConnect(drv, "jdbc:mysql://localhost/test")
dbListTables(conn)
data(iris)
dbWriteTable(conn, "iris", iris)
dbGetQuery(conn, "select count(*) from iris")
d <- dbReadTable(conn, "iris")

## End(Not run)
```

JDBCConnection-class *Class "JDBCConnection"*

Description

Class representing a (DBI) database connection which uses JDBC to connect to a database.

Generators

Objects can be created by call to dbConnect of a JDBC driver.

Slots

jc: Java object representing the connection.

identifier.quote: Quote character to use for quoting identifiers it automatically generated SQL statements or NA if the back-end doesn't support quoted identifiers. Usually the value is inherited from the "JDBCDriver".

Extends

Class "DBIConnection-class", directly. Class "DBIObject-class", by class "DBIConnection", distance 2.

Methods

No methods defined with class "JDBCConnection" in the signature.

See Also

JDBC, "JDBCDriver"

JDBCConnection-methods

Methods for the class 'JDBCConnect' in Package 'RJDBC'

Description

Methods for the class 'JDBCConnection' in Package 'RJDBC'.

`dbSendQuery` and `dbSendUpdate` submit a SQL query to the database. The difference between the two is only that `dbSendUpdate` is used with DBML queries and thus doesn't return any result set.

`dbGetTables` and `dbGetFields` are similar to `dbListTables` and `dbListFields` but the result is a data frame with all available details (whereas the latter return only a character vector of the names).

Usage

```
dbSendUpdate (conn, statement, ...)
dbGetTables (conn, ...)
dbGetFields (conn, ...)
```

Arguments

`conn`	connection object
`statement`	SQL statement to execute
`...`	additional arguments to prepared statement substituted for "?"

Details

Some notable enhancements to the DBI API:

`dbSendUpdate` supports vectorized arguments which is far more efficient than using scalar updates. Example: `dbSendUpdate(c, "INSERT INTO myTable VALUES(?, ?)", rnorm(1000), runif(1000))` performs a single JDBC `batchExecute()` call. Additional parameter `max.batch=10000L` is an integer that specifies the maximum batch size supported by the DBMS.

`dbSendQuery` and `dbSendUpdate` accept both `...` (populated frist) as well as `list=` (populated as second). Only unnamed arguments are used from `...` (assuming that those are function arguments and no data) while all elements are used from `list=`.

`dbGetQuery` is a shorthand for `sendQuery` + `fetch`. Parameters `n=-1`, `block=2048L` and `use.label=TRUE` are passed through to `fetch()` others to `dbSendQuery`.

`dbListTables` and `dbGetTables` have the arguments `(conn, pattern="%", schema=NULL)`. `dbExistsTable` is just a wrapper for `dbGetTables`.

`dbWriteTable` is defined as `(conn, name, value, overwrite=FALSE, append=FALSE, force=FALSE, ..., max.batch=10000L)` and is just a short-hand for the corresponding `dbSendUpdate()` statements. Since it is only a convenience wrapper, it is strongly recommended to use `dbSendUpdate()` in any real use-cases as you have far more control over the shape and properties of the table if you issue the `CREATE TABLE` statement according to your DBMS' capabilities.

`dbReadTable` is just a shorthand for `dbGetQuery(c, "SELECT * from <table>")`

Methods

dbBegin `signature(conn = "JDBCConnection", ...)`

dbCommit `signature(conn = "JDBCConnection", ...)`

dbDataType `signature(dbObj = "JDBCConnection", obj = "ANY", ...)`

dbDisconnect `signature(conn = "JDBCConnection", ...)`

dbExistsTable `signature(conn = "JDBCConnection", name = "character", ...)`

dbGetException `signature(conn = "JDBCConnection", ...)`

dbGetFields `signature(conn = "JDBCConnection", ...)`

dbGetInfo `signature(conn = "JDBCConnection", ...)`

dbGetQuery `signature(conn = "JDBCConnection", statement = "character", ...)`

dbGetTables `signature(conn = "JDBCConnection", ...)`

dbListFields `signature(conn = "JDBCConnection", ...)`

dbListResults `signature(conn = "JDBCConnection", ...)`

dbListTables `signature(conn = "JDBCConnection", ...)`

dbReadTable `signature(conn = "JDBCConnection", ...)`

dbRemoveTable `signature(conn = "JDBCConnection", ...)`

dbRollback `signature(conn = "JDBCConnection", ...)`

dbSendQuery `signature(conn = "JDBCConnection",statement = "character", ...)`

dbSendUpdate `signature(conn = "JDBCConnection",statement = "character", ...)`

dbWriteTable `signature(conn = "JDBCConnection", ...)`

`JDBCDriver-class` *Class "JDBCDriver"*

Description

A DBI driver that uses any JDBC driver to access databases.

Generators

Objects can be created by calls to `JDBC` or `dbDriver`.

Slots

`identifier.quote`**:** Quote character to use for identifiers in automatically generated SQL statements or `NA` if quoted identifiers are not supported by the back-end.

`jdrv`**:** Java object reference to an instance of the driver if the driver can be instantiated by a default constructor. This object is only used as a fall-back when the driver manager fails to find a driver.

Extends

Class "DBIDriver-class", directly. Class "DBIObject-class", by class "DBIDriver", distance 2.

Methods

No methods defined with class "JDBCDriver" in the signature.

See Also

JDBC

JDBCDriver-methods *Methods for the class JDBCDriver in Package 'RJDBC'*

Description

Methods for the class 'JDBCDriver' in Package 'RJDBC'.

Most prominent method is `dbConnect`, it creates a new JDBC connection using the specified driver. From RJDBC 0.2-9 on the driver takes precedence over DriverManager, because DriverManager is static and is not capable of finding drivers dynamically. DriverManager is now only used if the driver is a NULL-driver, i.e., `JDBC(NULL)`.

There are only two positional properties `user=''` and `password=''` neither of which will be set if empty. All other arguments are treated as additional properties passed to the connection (except when DriverManager is used).

Additional arguments to `dbConnect()` properties are set

`dbListConnections` always return `NULL` with a warning, because JDBC connections are not tracked.

`dbGetInfo` returns very basic information, because the JDBC driver is not loaded until a connection is created.

`dbUnloadDriver` is a no-op in the current implementation, because drivers are never removed from the JVM.

Methods

dbConnect `signature(drv = "JDBCDriver", ...)`

dbListConnections `signature(drv = "JDBCDriver", ...)`

dbGetInfo `signature(drv = "JDBCDriver", ...)`

dbUnloadDriver `signature(drv = "JDBCDriver", ...)`

JDBCResult-class *Class "JDBCResult"*

Description

Representation of a DBI result set returned from a JDBC connection.

Generators

Objects can be created by call to dbSendQuery.

Slots

jr: Java reference to the JDBC result set

md: Java reference to the JDBC result set meta data

env: Environment holding cached objects (currently the result helper object used by fetch())

stat: Java reference to the JDBC statement which generated this result

Extends

Class "DBIResult-class", directly. Class "DBIObject-class", by class "DBIResult", distance 2.

Methods

No methods defined with class "JDBCResult" in the signature.

See Also

JDBC, dbSendQuery

JDBCResult-methods *Methods for the class JDBCResult in Package 'RJDBC' ~~*

Description

Methods for the class 'JDBCResult' in Package 'RJDBC'.

fetch retrieves the content of the result set in the form of a data frame. If n is -1 then the current implementation fetches 32k rows first and then (if not sufficient) continues with chunks of 512k rows, appending them. If the size of the result set is known in advance, it is most efficient to set n to that size.

Additional argument block can be used to inform the driver that pre-fetching of a certain block of records is desirable (see setFetchSize() in JDBC) leading to possibly faster pulls of large queries. The default is to pre-fetch 2048 records. Note that some databases (like Oracle) don't support a fetch size of more than 32767. If set to NA or anything less than 1 then the fetch size is not changed. This is only a hint, drivers are free to ignore it.

Finally, use.label logical argument determines whether column lables are used for naming (TRUE, default) or column names should be used (FALSE) since some database drivers do not implement labels correctly.

dbClearResult releases the result set.

dbColumnInfo returns basic information about the columns (fields) in the result set.

dbGetInfo returns an empty list.

dbListResults returns an empty list and warns that JDBC doesn't track results.

Methods

fetch signature(res = "JDBCResult", ...)

dbClearResult signature(res = "JDBCResult", ...)

dbColumnInfo signature(res = "JDBCResult", ...)

dbGetInfo signature(res = "JDBCResult", ...)

dbListResults signature(res = "JDBCResult", ...)

Index