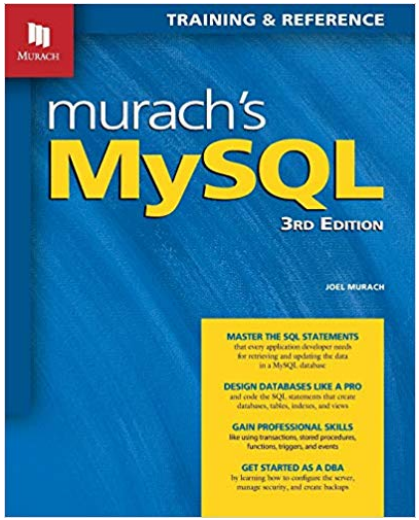
Building host programs

Connecting to a MySQL database
Topic 4 Lesson 7

Adapted from Chapter 1



https://dev.mysql.com/doc/connector-j/8.0/en/

https://dev.mysql.com/doc/connector-python/en/connector-python-reference.html

https://pymysql.readthedocs.io/en/latest/

Embedding SQL

SQL commands can be called from within a host language (e.g., C++ or Java) program. SQL statements can refer to host variables (including special variables used to return status).

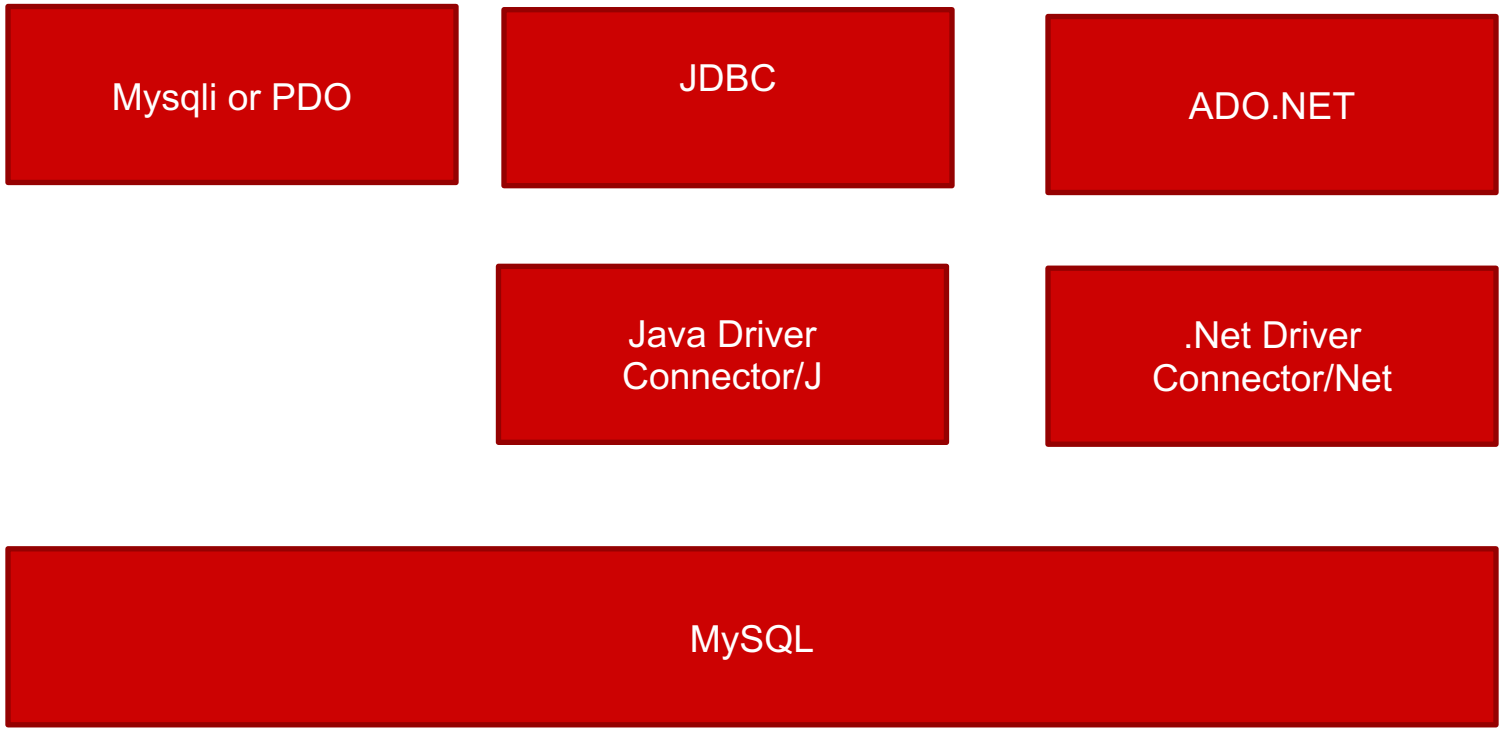
Two main integration approaches:

- Embed SQL in the host language (Embedded SQL, SQLJ). A Preprocessor converts SQL code to host language calls. The output from the preprocessor is then compiled by the host compiler
- Create special API to call SQL commands

JDBC Java Database Connectivity API (for JAVA)
http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/

ODBC Standard database connectivity API Pep 249 – **Python** Database Application specification https://www.python.org/dev/peps/pep-0249/

Embedded SQL



Database (API)s

Add a library with database calls (API)

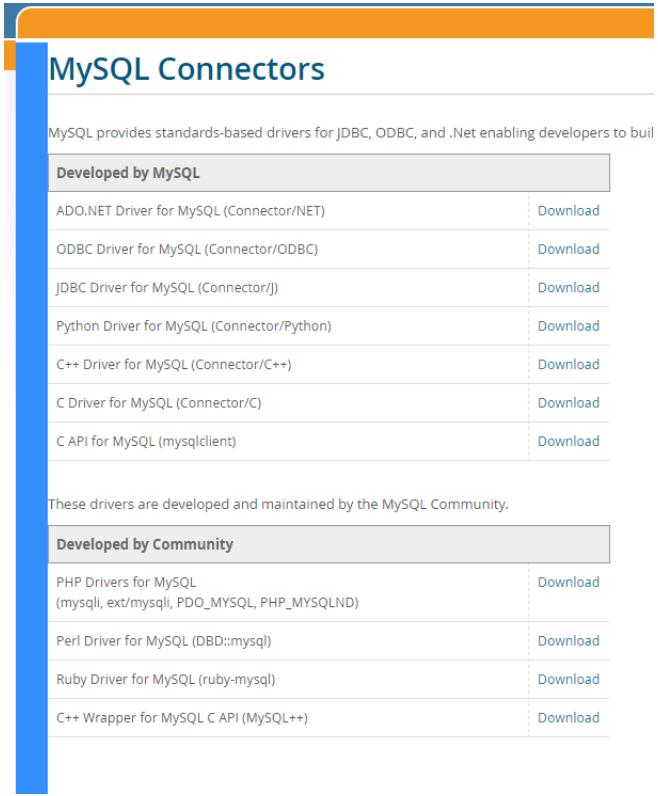
- Special standardized interface: procedures/objects
  - Pass SQL strings from host language, presents result sets in a host language-friendly way

A “driver” traps the calls and translates them into DBMS specific code (Oracle, MySQL, SQL Server etc.)

- database can be across a network

GOAL: applications are independent of database systems and operating systems

Download the desired driver



MySQL Connectors

MySQL provides standards-based drivers for JDBC, ODBC, and .Net enabling developers to buil

Developed by MySQL	
ADO.NET Driver for MySQL (Connector/NET)	Download
ODBC Driver for MySQL (Connector/ODBC)	Download
JDBC Driver for MySQL (Connector/J)	Download
Python Driver for MySQL (Connector/Python)	Download
C++ Driver for MySQL (Connector/C++)	Download
C Driver for MySQL (Connector/C)	Download
C API for MySQL (mysqlclient)	Download

These drivers are developed and maintained by the MySQL Community.

Developed by Community	
PHP Drivers for MySQL (mysqli, ext/mysqli, PDO_MYSQL, PHP_MYSQLND)	Download
Perl Driver for MySQL (DBD::mysql)	Download
Ruby Driver for MySQL (ruby-mysql)	Download
C++ Wrapper for MySQL C API (MySQL++)	Download

GO TO:
https://www.mysql.com/products/connector/

MySQL Drivers

- Connector/ODBC provides driver support for connecting to MySQL using the Open Database Connectivity (ODBC) API.
- Connector/Net enables developers to create .NET applications that connect to MySQL. Connector/Net implements a fully functional ADO.NET interface and provides support for use with ADO.NET
- Connector/J provides driver support for connecting to MySQL from **Java** applications using the standard Java Database Connectivity (JDBC) API.
- Connector/Python provides driver support for connecting to MySQL from **Python** applications using an API that is compliant with the Python DB API version 2.0. http://dev.mysql.com/doc/connector-python/en/
- Connector/C++ enables C++ applications to connect to MySQL.
- Connector/C is a standalone replacement for the MySQL Client Library (libmysqlclient), to be used for C applications.

JDBC Processing (Java)

Steps to submit a database query:

Load the JDBC driver
Connect to the data source
Execute SQL statements

JDBC Architecture

Application or the client (initiates and terminates connections, submits SQL statements)
Driver manager (loads the JDBC driver)
Driver (connects to data source, transmits requests and returns/translates results and error codes)
Data source (processes SQL statements)

JDBC Driver Class

All drivers are managed by the Java DriverManager class

To load a JDBC driver in Java host code:

Class.forName(“oracle/jdbc.driver.Oracledriver”); /Oracle

Class.forName("com.mysql.jdbc.Driver"); /MySQL

When starting the Java application:

-Djdbc.drivers=oracle/jdbc.driver

Or provide the driver in the CLASSPATH directory

For a description of the flags that can be passed to driver:

https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-reference-configuration-properties.html

Connecting to a DB via JDBC

Interact with a data source through sessions. Each connection identifies a logical session.
JDBC URL: jdbc:<subprotocol>:<otherParameters>

Example:

```
//Define URL of database server for
 // database named mysql on the localhost
 // with the default port number 3306.
 String url =
    "jdbc:mysql://localhost:3306/mysql";

//Get a connection to the database for a user named root with a xxxx password.
Connection con = DriverManager.getConnection( url,"root", “xxxx");

//Display URL and connection information
 System.out.println("URL: " + url);
 System.out.println("Connection: " + con);
```

Connection class interface

public int **getTransactionIsolation()** and
void **setTransactionIsolation**(int level)
Sets isolation level for the current connection.

public boolean **getReadOnly()** and void **setReadOnly(boolean b)**
Specifies whether transactions in this connection are readonly

public boolean **getAutoCommit()**
and void **setAutoCommit(boolean b)**
If autocommit is set, then each SQL statement is considered its own transaction. Otherwise, a transaction is committed using commit(), or aborted using rollback().

public boolean **isClosed()**
Checks whether connection is still open.

Executing SQL statements

Three different methods to execute SQL statements:

- **Statement** (both static and dynamic SQL statements)
- **PreparedStatement** (semi-static SQL statements)
- **CallableStatment** (stored procedures)

PreparedStatement class: Precompiled, parameterized SQL statements:

- Structure of the SQL statement is fixed
- Values of parameters are determined at run-time

Prepared stmt: pass and define arguments

String sql="INSERT INTO Sailors VALUES(?,?,?,?)";
PreparedStatment pstmt=**con.prepareStatement**(sql);
pstmt.clearParameters();
pstmt.setInt(1,sid);
pstmt.setString(2,sname);
pstmt.setInt(3, rating);
pstmt.setFloat(4,age);
// No return rows use executeUpdate()
int numRows = **pstmt.executeUpdate();**

Parameters are positional

Result set (cursor)

PreparedStatement.executeUpdate only returns the number of affected records

PreparedStatement.executeQuery returns data, encapsulated in a ResultSet object (a cursor)

```
ResultSet rs=pstmt.executeQuery(sql);
// rs is now a cursor
While (rs.next()) {
// process the data  }
```

ResultSet: Cursor with seek functionality

A ResultSet is a very powerful cursor:

previous(): moves one row back
absolute(int num): moves to the row with the specified number
relative (int num): moves forward or backward
first() and **last**()

Functionality not available for MySQL cursors

Java to SQL types and get methods

SQL Type	Java class	Result Set get method
BIT	Boolean	getBoolean()
CHAR	String	getString()
VARCHAR	String	getString()
DOUBLE	Double	getDouble()
FLOAT	Double	getDouble()
INTEGER	Integer	getInt()
REAL	Double	getFloat()
DATE	Java.sql.Date	getDate()
TIME	Java.sql.Time	getTime()
TIMESTAMP	Java.sql.Timestamp	getTimestamp()

JDBC: Processing errors and exceptions

Most of java.sql can throw an error and set SQLException when an error occurs

An SQLException can occur both in the driver and the database. When such an exception occurs, an object of type SQLException will be passed to the catch clause.

SQLWarning is a subclass of SQLException

- Not as severe as an error
- They are not thrown
- Code has to explicitly test for a warning

Example of try and catch for error handling

```
try {
            stmt=con.createStatement();
            warning=con.getWarnings();
            while(warning != null) {
                        // handle SQLWarnings;
                        warning = warning.getNextWarning():
             }
      con.clearWarnings();
     stmt.executeUpdate(queryString);
      warning = con.getWarnings();
                …
   }    //end try
   catch( SQLException SQLe) {
    // handle the exception
     System.out.println( SQLe.getMessage());}
```

Examining meta data for the DB

DatabaseMetaData object gives information about the database system catalog.

```
DatabaseMetaData md = con.getMetaData();
// print information about the driver:
System.out.println(
        “Name:” + md.getDriverName() +
         “version: ” + md.getDriverVersion());
```

Metadata- print out tables and fields

```
DatabaseMetaData md=con.getMetaData();
ResultSet trs=md.getTables(null,null,null,null);
String tableName;
While(trs.next()) {
        tableName = trs.getString("TABLE_NAME");
        System.out.println("Table: " + tableName);
        //print all attributes
        ResultSet crs = md.getColumns(null,null,tableName, null);
        while (crs.next()) {
                System.out.println(crs.getString("COLUMN_NAME" + ", ");
         }

   }
```

http://docs.oracle.com/javase/10/docs/api/java/sql/DatabaseMetaData.html

Connect, Process, check for errors

```
Connection con = // connect
      DriverManager.getConnection(url, ”login", ”pass");
Statement stmt = con.createStatement(); // set up stmt
String query = "SELECT name, rating FROM Sailors";
ResultSet rs = stmt.executeQuery(query);
try { // handle exceptions
        // loop through result tuples
         while (rs.next()) {
               String s = rs.getString(“name");
                Int n = rs.getFloat(“rating");
                System.out.println(s + " " + n);
           }
    } catch(SQLException ex) {
        System.out.println(ex.getMessage () +
          ex.getSQLState () + ex.getErrorCode ());
}
```

Connect

Get multiset

Process with cursor

Catch Errors

Java documentation

For documentation refer to:
https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-examples.html

Java Summary

APIs such as JDBC introduce a layer of abstraction between application and DBMS

Embedded SQL allows execution of parameterized static queries within a host language

Dynamic SQL allows execution of completely ad hoc queries within a host language

Cursor mechanism allows retrieval of one record at a time and bridges impedance mismatch between host language and SQL

Building MySQL python applications

Topic 4 Lesson 8

Python database objects

2 main classes for processing database queries

- **Connection object**
  - Connection to the database
  - Object created via the connection (.connection) method
- **Cursor object**
  - Query statement execution
    - Method to execute a statement (.execute)
  - Result to the results
    - Method to retrieve row of data from the results (variations of fetch)
  - Cursor object created by the cursor method (.cursor) of the connection object.
  - Method to run a MySQL procedure (.callproc)

Process for accessing database

1. Import the MySQL API module
2. Acquire a connection to a specific database
3. Issue SQL statements and stored procedures.
4. Close the connection

Database (API)s

Add a library with database calls (API)

- Special standardized interface: procedures/objects
  - Pass SQL strings from host language, presents result sets in a host language-friendly way

A “driver” traps the calls and translates them into DBMS specific code (Oracle, MySQL, SQL Server etc.)

- database can be across a network

GOAL: applications are independent of database systems and operating systems

Python connection library

Mysqlclient: a wrapper around the mysql-connector-c C library. You should have a development C environment set up to compile C code to use this library.

Pymysql: pure python implementation. It tends to be available quicker for the newer versions of python.

mysql-connection-python: Developed from the MySQL group at Oracle. Another pure python implementation.

mysql-connector: Original connector from MySQL

Python mysql.connector example

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# Simple MySQL database connection

import flask
import mysql.connector


def main(config):
    output = []
    cnx = mysql.connector.connect(**config)

    cur = cnx.cursor()
    cur2 = cnx.cursor()
    reb = 'rebels'
    movie_id = 1
    stmt_select = "select * from characters order by character_name"

    cur.execute(stmt_select)
    for row in cur.fetchall():
        output.append('{0:20s}  {1:15s} {2:15s}
            {3:15s}'.format(row[0], row[1], row[2], row[3]))
    cur.close()
```

Python mysql.connector (cont.)

```
    s2 = 'SELECT * FROM movies WHERE movie_id = {}'.format(movie_id)
        cur2.execute(s2)
        for row in cur.fetchall():
            print(row)

        cur2.callproc('track_planet', args=['Endor'])

        for result in cur2.stored_results():
            print(result.fetchall())

        cur2.close()

        return output


    if __name__ == '__main__':
        config = {
            'host': 'localhost',
            'port': 3306,
            'database': 'starwarsfinal',
            'user': 'root',
            'password': 'root',
            'charset': 'utf8',
            'use_unicode': True,
            'get_warnings': True,
        }

        out = main(config)
        print('\n'.join(out))
```

Example pymysql (connect & retrieve data)

```
import pymysql


    cnx = pymysql.connect(host='localhost', user='root', password='root',
                          db='lotrfinal', charset='utf8mb4',
    cursorclass=pymysql.cursors.DictCursor)

    cur = cnx.cursor()
    stmt_select = "select * from lotr_character order by
    character_name"

    cur.execute(stmt_select)

    rows = cur.fetchall()
```

Pymysql provides different cursors

- Pymysql.cursors.SSDictCursor : an unbuffered cursor, useful for queries that returns many rows or for connections on remote servers. Instead of copying every row of data to the buffer, this will fetch rows as needed
- Pymyql.cursors.DictCursor: returns the result as a dictionary, where the key is the field name and the value is the field value
- Pymysql.cursors.SSDictCursor: an unbuffered cursor, which returns the results as a dictionary {field_name: field_value}

Example pymysql (process cursor)

```
for row in rows:
        print(row) # prints each field as a key value pair
        print(row["character_name"], row['species'])
        #reference field by name
        c_name_var = row["character_name"]
        # get specific values
    cur.close()
```

Prepared statements

Any SQL statement can be made into a prepared statement by using the character string %s to specify a value that will be provided at execution time:

Example:

```
species = ‘elf’
cursor = cnx.cursor()
query = "SELECT character_name FROM lotr_character WHERE species=%s”
cursor.execute(query, species)
# ... retrieve data ...
```

Tuples affected by the query

The cursor method rowcount returns the number of tuples affected or returned by the SQL statement. For example, if cur is the cursor result of a SELECT statement

```
print("The query returned {} rows".format(cur.rowcount))
```

Prints the number of rows returned.

```
The query returned 2 rows
```

Starting Points

For pymysql:
https://pypi.org/project/PyMySQL/
https://www.tutorialspoint.com/python3/python_database_access.htm
https://pymysql.readthedocs.io/en/latest/modules/index.html

For mysqlclient-python:
https://pypi.org/project/mysqlclient/

For mysql-connection
https://dev.mysql.com/doc/connector-python/en/connector-python-versions.html

For a comparison of the approaches
https://wiki.openstack.org/wiki/PyMySQL_evaluation

Python Summary

There are many different libraries for connecting a python application to a MySQL database. Pymysql is written entirely in python and does not require a C development environment. It also provides 3 different types of cursor objects.

Handling OUT and INOUT parameters to python from MySQL requires the use of wrapper parameter that runs on the DB server. It extracts the values from the session variables into a cursor.

Building MySQL R applications

Topic 4 Lesson 9

R connection to the client server model

As of today, there is no standard driver to connect an R program to a MySQL database

The DBI package separates the connectivity to the DBMS into a “front-end” and a “back-end”. Applications use only the exposed front-end API. The back-end facilities that communicate with specific DBMSs (SQLite, MySQL, PostgreSQL, MonetDB, etc.) are provided by drivers (other packages) that get invoked automatically through R’s S4 methods.

R to MySQL

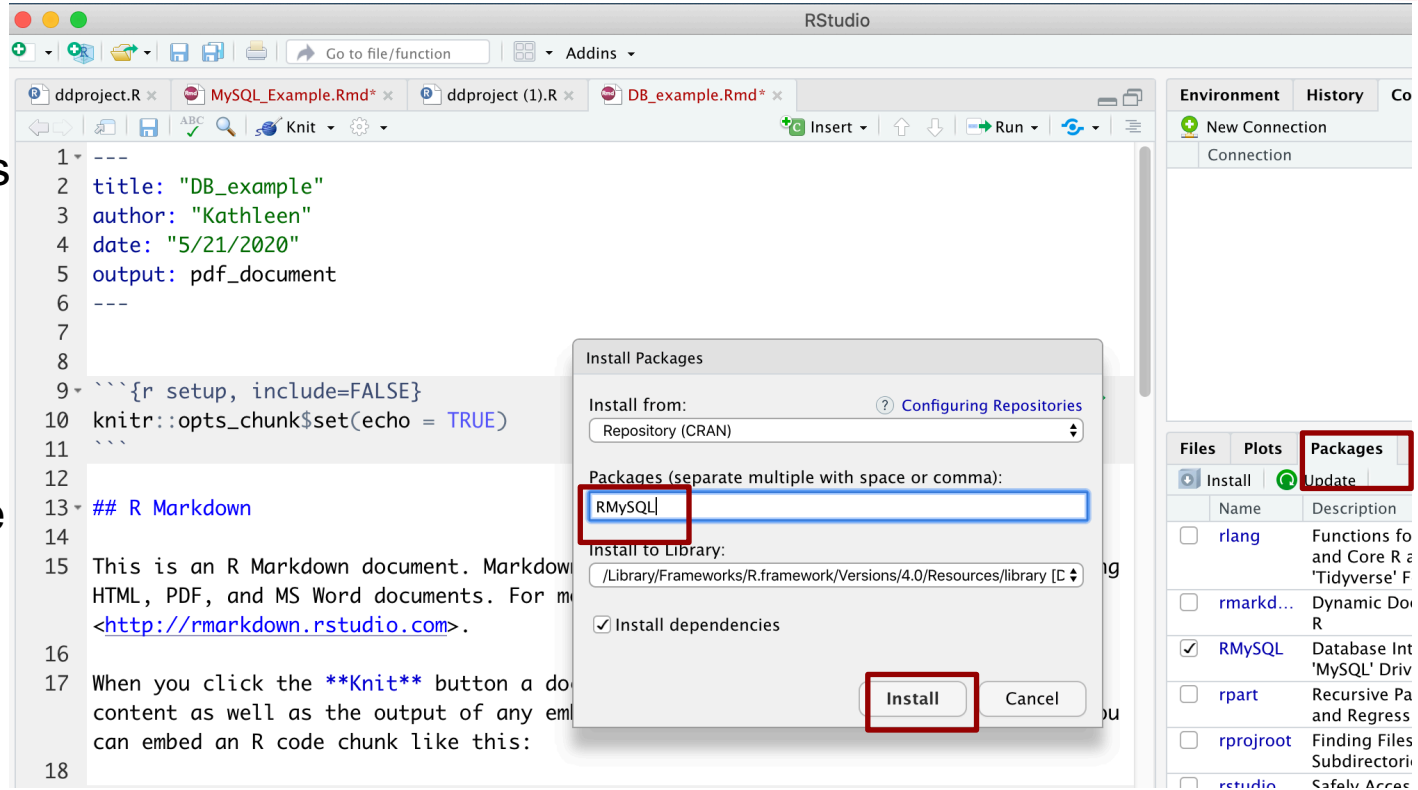
There are a few packages that do connect a R script to a MySQL database, RMySQL, RODBC, RMariaDB and RJDBC. (RMariaDB and RMySQL are supported) – both provide the same interface. All such libraries automatically include the DBI package.

Just like any other R package you must install the package locally so we can access its methods from our R script

Installing RMySQL

Issue the command: install.packages ("RMySQL") to install

You can also use the packages tab to install the library with the popup window



Connecting to the database

We use the dbConnect method to connect to the database. You need to provide your credentials, the hostname of the database and the port to connect. It returns an object with class MySQLConnection

EXAMPLE:

```
library(RMySQL)
mydb = dbConnect(MySQL(), user='user',
  password='password', dbname='database_name',
  host='host', port = 3306)
```

Documentation can be found:
https://cran.rproject.org/web/packages/RMySQL/RMySQL.pdf

Requesting and Accessing data

REQUEST data: dbSendQuery(con, sql) will retrieve data from the database a chunk at a time.

Parameters:

con is the value returned from DbConnect

sql is the query you wish to run on the database

Output: returns a MySQLConnection class

Accessing the requested data

ACCESS data: dbFetch(MySQLResult, n)

Parameters:

MySQLResult is the return variable from dbSendQuery

n maximum number of records to retrieve

Clean up: when done with the results free the allocated space with dbClearResult(res)

Requesting and Accessing data

REQUEST:dbGetQuery(con, sql) will retrieve data from the database

con parameter is the value returned from DbConnect

sql parameter is the query you wish to run on the database

It will automatically fetch all data locally and clear the space for the data. This should be used when the size of the returning data is small (will not exceed virtual memory of R program).

Example code (1)

```
library(RMySQL)
library(tidyverse)
globalUsername <- "root"
globalPass <- ”password"


# 1Settings
db_user <- 'root'
db_password <- ‘password'
db_name <- 'lotrfinal_1'

db_host <- '127.0.0.1' # for local access
db_port <- 3306

# 2. Connect to the db
mydb <-  dbConnect(MySQL(), user = db_user, password = db_password,
                   dbname = db_name, host = db_host, port = db_port)
```

Example code (2)

```
db_table <- 'lotr_character'

s <- str_c("select * from ", db_table)
# 2. Read from the db

rs <- dbSendQuery(mydb, s)
df <-  fetch(rs, n = -1)  #-1 represents to read all data
df
dbClearResult(rs)
dbDisconnect(mydb)
```

Example code (3)

```
# fetch chunks of data when dealing with large results

res <- dbSendQuery(con, "SELECT * FROM lotr_character")
while(!dbHasCompleted(res)){
  chunk <- dbFetch(res, n = 5)
  print(chunk)
  print("Next chunk")
  print(nrow(chunk))
}
dbClearResult(res)
```

Reading a table from the database

You can read a table from the database.

dbReadTable(con, name, row.names, check.names = TRUE, ...)

con – DBConnect object

name – name of the table

row.names - A string or an index specifying the column in the DBMS table to use as row.names in the output data.frame. Defaults to using the row_names column if present. Set to NULL to never use row names.

check.names - if TRUE, the default, column names will be converted to valid R identifiers

Writing a data frame to the database

You can write a data frame as a table to the database. This is useful for storing a data frame to permanent storage.

dbWriteTable(conn, name, value, field_types = NULL, row_names = TRUE, overwrite = FALSE, append = FALSE, ..., allow.keywords = FALSE)

con – DBConnect object
name – name of the table
value data frame to be stored as the table

Package RMariaDB

- For compatibility, the functions for connecting, retrieving and storing data in the database are the same as RMySQL.
- It also provides functions for managing transactions
- The data structures have different classes in the RMariaDB package the object returned from connect is a MariaDBConnection object
- MariaDBResult objects are returned from dbGetQuery.
- Please refer to the documentation:

https://cran.r-project.org/web/packages/RMariaDB/RMariaDB.pdf

A full tutorial can be found at https://programminghistorian.org/en/lessons/getting-started-with-mysql-using-r#selecting-data-from-a-table-with-sql-using-r

Summary

- R has the data frame object that an analogous to the structure of a relational table. We use a data frame object to accept data or pass data to/from the database.
- The MySQLConnection class is the object that tracks all information necessary for a connection (RMySQL)
- The MySQLResult class is the object that represents the data retrieved from the database (RMySQL)