Systems Design and Security

Part 9: Java and MySQL

http://staffwww.dcs.shef.ac.uk/people/A.Simons/

Home ⇒ Teaching ⇒ Lectures ⇒ COM2008/COM3008

Bibliography

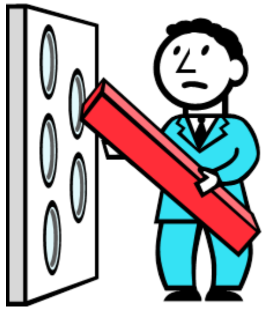
- Database Systems
  - T Connolly and C Begg, Database Systems – a Practical Approach to Design, Implementation and Management, 6th ed., Pearson, 2014.
  - C J Date, An Introduction to Database Systems, 8th ed., Pearson, 2003.
- MySQL and Java
  - K Sharan, Beginning Java 8 APIs, Extensions and Libraries, APress, 2014. -- Java 8.
  - M Matthews, J Cole, J D Gradecki, MySQL and Java Developers Guide, Wiley, 2003. -- Java 1.4
  - take care with older Java versions

Outline

- Installing MySQL and JDBC
- Managing connection resources
- Executing updates and queries
- Object-data conversion
- Commits and transactions
- Injection attack and validation

Programs and Databases

- Weakly-typed databases
  - databases store mostly text, some binary data
  - SQL uses simple types: int, char, date, etc.
  - search results mostly strings, chars, ints
- Strongly-typed programming languages
  - programs use rich, structured object types
  - stronger type-checking at the class-level
- Programmatic access to data
  - programs have to overcome the "impedance mismatch"
  - map string, int data into complex object types
  - vice-versa, when storing objects back in databases



MySQL Database

- MySQL is a robust free database
  - runs under Windows, Unix and Linux
  - uses the standard SQL query language
  - many on-line tutorials are available

- Get a MySQL group account (see later)
  - your lecturer will tell you when these are ready
  - all your group members get shared access to the same DB
  - only accessible from campus network (or use VPN)

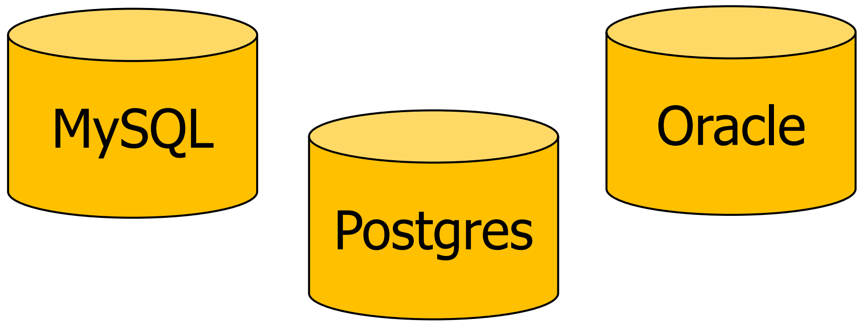
DCS Guide: https://guide.dcs.shef.ac.uk/doku.php?id=students
MySQL Guide, e.g.: http://www.mysqltutorial.org/

Java Database Connectivity

- JDBC developed by Sun Microsystems
  - Java 8 SE comes with JDBC 4.2
  - builds on the simplicity/portability of SQL
  - supports conversion to/from Java objects
  - database failures reported as Java exceptions
- JDBC is the natural API for Java access to data
  - simple to use for non-database programmers
  - hides specific database details from programmers
  - can access any tabular data source, eg: CSV files

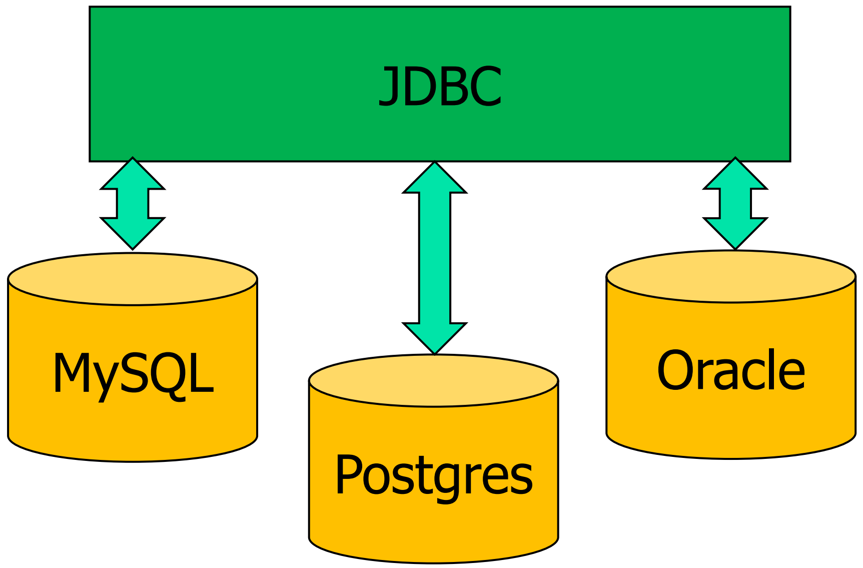
JDBC Architecture

- Assumes multiple vendor RDB implementations
- Assumes SQL used as the common query language



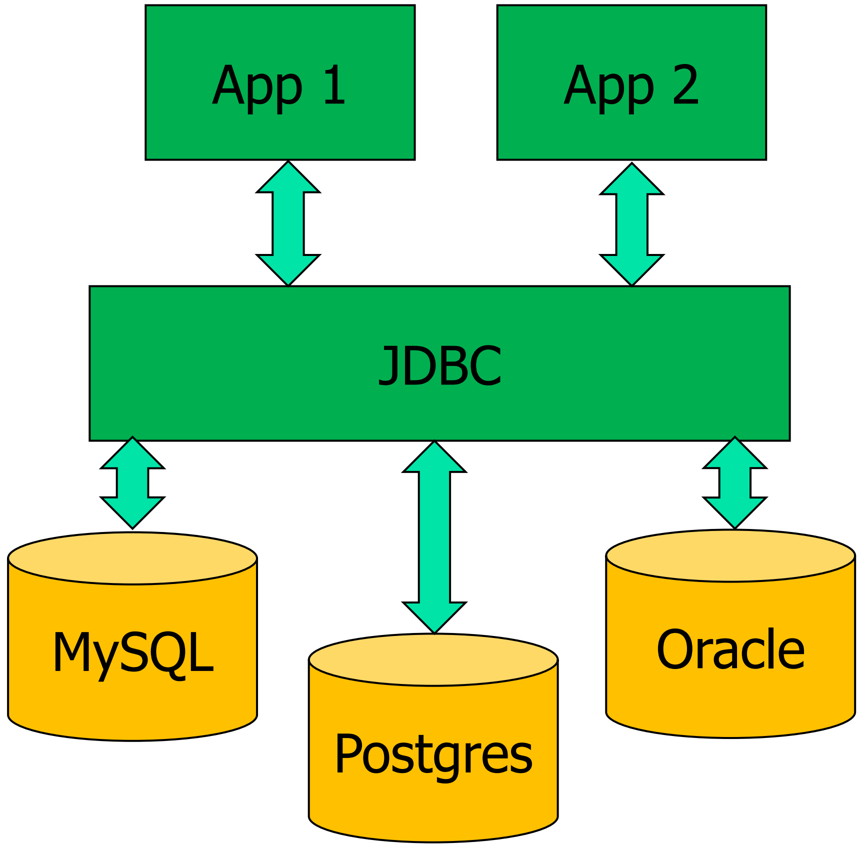
JDBC Architecture

- JDBC defines a set of interfaces
- vendors implement these for their database engine
- known as a JDBC driver
- need to download a specific JDBC driver for each database engine



JDBC Architecture

- run Java apps, that use the JDBC interface
- easy to switch between DB vendors, hardly altering a line of Java code
- just load a different JDBC driver at Java startup
- if you start with toy DBs in Microsoft Access…
- … can scale up later to large Oracle DBs



Overview

Platform-independent Connector/J in your group project folder

- Get the JDBC driver for MySQL
  - from: http://www.mysql.com/products/connector/
  - use the Connector/J version of the driver (for Java)
  - install the jarfile on your project's build path
  - see: http://download.oracle.com/javase/tutorial/jdbc/
- Find/read about the JDBC classes
  - the packages: java.sql, javax.sql contain JDBC classes
  - DriverManager – creates connections to the DB
  - Connection – represents an open connection to the DB
  - Statement – represents a statement to execute on the DB
  - ResultSet – is an iterator over the results of a query

Install the Driver

- Unzip the download bundle
  - the download is: mysql-connector-java-[version].zip
  - contains a jarfile: mysql-connector-java-[version]-bin.jar
  - this contains a driver class: com.mysql.jdbc.Driver
- Java archive (jar) files
  - zipped files containing compiled Java classes, docs, etc.
  - can contain library of add-on software (as in this case)
  - need to tell Java where to find the jarfiles you use
    - JDK: append path to the jarfile to your CLASSPATH
    - Eclipse, IntelliJ, NetBeans: add the jarfile to your build path

Eclipse Build Path

- Select project in the package explorer list
  - Right-click to pop up main menu
  - Select Build Path and submenu Configure Build Path
- Properties Browser opens at Java Build Path
  - Select tab Libraries
  - Click on right-hand button Add External JAR
  - Browse for mysql-connector-java-[version]-bin.jar
  - Open (OK), it should then appear in the list
  - Click on OK to save and close Properties Browser.

IntelliJ and Netbeans

- IntelliJ IDEA
  - Select project in the project list
  - Go to main File menu → Project Structure
  - Go to Project Settings → Libraries
  - Click + icon to add external library, browse for JAR, OK
- NetBeans
  - Right-click on the project for main menu
  - Select Properties → Libraries
  - Right-click Libraries → Add JAR/Folder
  - Under Compile tab, browse for JAR, OK

Check Driver Exists

```
import java.sql.*;
import java.util.*;

public class FindDrivers {
  public static void main(String[] args) throws Exception {
    System.out.println("\nDrivers loaded as properties:");
    System.out.println(System.getProperty("jdbc.drivers"));
    System.out.println("\nDrivers loaded by DriverManager:");
    Enumeration<Driver> list = DriverManager.getDrivers();
    while (list.hasMoreElements())
      System.out.println(list.nextElement());
  }
}
```

Example Output

```
import java.sql.*;
import java.util.*;

public class FindDrivers {
  public static void main(String[] args) throws Exception {
    System.out.println("\nDrivers loaded as properties:");
    System.out.println(System.getProperty("jdbc.drivers"));
    System.out.println("\nDrivers loaded by DriverManager:");
    Enume
    while
      Sys
  }
}
```

```
Drivers loaded as properties:
null

Drivers loaded by DriverManager:
sun.jdbc.odbc.JdbcOdbcDriver@ca0b6
com.mysql.jdbc.Driver@1270b73
```

Using JDBC

- Use the JDBC 4.x library
  - import java.sql.*;   // adequate for most purposes
  - import javax.sql.*;   // only for advanced features
- Four main steps
  - open a connection to the server
  - execute a query/update (one or more)
  - iterate over the results of a query
  - release server resources - otherwise will crash!
    - typically after 20 opened connections!
- Resource management
  - need to get this right, remember to close (see next)

Resources – Old Style

```
Connection con = null; // connection to a database

try {
 con = DriverManager.getConnection(...);


     // use the open connection
     // for one or more queries

}
catch (Exception ex) {
  ex.printStackTrace();
}
finally {
  if (con != null) con.close();
}
```

declare *con* outside the try-block

opening the connection may fail

handle any exception and report the error

finally, close *con* if it was opened

- Explicit release of resources, in a try...catch...finally block

Resources – New Style

```
try (Connection con = DriverManager.getConnection(...)) {

     // use the open connection
     // for one or more queries

}
catch (Exception ex) {
  ex.printStackTrace();
}
```

declare *con* inside () parentheses, before the start of the try-block

automatically closed at the end of the block

- Implicit release of resources, in a try-with-resources block
- Any AutoClosable object can be initialised in the () parentheses
- Automatically closed, whether success or failure occurs
- Since Java 7 / JDK1.7, if your IDE recognises it!
  https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

Open a Connection

```
Connection con = null;  // a Connection object

try {

   con = DriverManager.getConnection(
      "jdbc:mysql://server/dbname", "userID", "password");

    // use the open connection
    // for several queries

}
catch (Exception ex) {
      ex.printStackTrace();
}
finally {
   if (con != null) con.close();
}
```

server/dbname path may include a port number, and dbname is as you set it up

userID and password are the user ID and password for the dbname

All other query code goes in here

Different Styles

- Example database IDs
  - on separate DB server, with DB name "team043"
  - on localhost, on a specific port, with DB name "myDB"

```
“jdbc:mysql://stusql.dcs.shef.ac.uk/team043”

"jdbc:mysql://localhost:3306/myDB"
```

- Example connection methods
  - multi-argument, or single string URL argument

```
Connection con = DriverManager.getConnection(
  “database”, “dbuser”, “dbpassword”);

Connection con = DriverManager.getConnection(
  “database?user=dbuser&password=dbpassword”);
```

Execute an Update

```
Statement stmt = null;

try {
  stmt = con.createStatement();
  int count = stmt.executeUpdate(
    "UPDATE lecturer SET office = 119"
      + " WHERE name = 'A Simons'");
}
catch (SQLException ex) {
  ex.printStackTrace();
}
finally {
  if (stmt != null)
    stmt.close();
}
```

Use executeQuery(sqlString) for SELECT statements

Use executeUpdate(sqlString) for INSERT, DELETE and UPDATE statements

Updates return the number of rows that were updated (or zero)

Data Manipulation

Command	Action	Example
CREATE	creates a table	CREATE TABLE student (…);
INSERT	inserts records	INSERT INTO student VALUES (…);
UPDATE	modifies records	UPDATE student SET name=‘Jill Smith’ WHERE id=4;
DELETE	deletes records	DELETE FROM student WHERE name=‘Joe Bloggs’;
DESCRIBE	info about a table	DESCRIBE student;

SHOW	info about system	SHOW DATABASES;
		SHOW TABLES [FROM db];

DROP	delete tables	DROP TABLE student;

Execute a Query

```
Statement stmt = null; // a SQL statement object

try {
   stmt = con.createStatement();
   ResultSet res =
      stmt.executeQuery("SELECT * FROM lecturer");

      // do what you like with the results;
      // could convert into objects

   res.close();
}
catch (SQLException ex) {
   ex.printStackTrace();
}
finally {
   if (stmt != null) stmt.close();
}
```

Statement objects are created by the open Connection

executeQuery() accepts a SQL query and returns a ResultSet

Remember to close Statement objects when finished

SQL Data Access

Command	Action	Example
SELECT	displays a table	SELECT * FROM student;
SELECT	projects columns	SELECT id, name FROM student;
WHERE	selects rows	SELECT id, name FROM student WHERE id=4;
WHERE	inner join	SELECT name FROM lecturer, module WHERE module.lec = lecturer.id
INNER JOIN	inner join	SELECT name FROM lecturer INNER JOIN module ON lec = lecturer.id
USING	natural join	SELECT name FROM module, student USING id

Iterate over Results

```
ResultSet res = stmt.executeQuery(
     "SELECT * FROM lecturer WHERE name='A Simons'");
  while (res.next()) {
    int id = res.getInt(1);               // col 1 as int
    String name = res.getString(2);       // col 2 as string
    int office = res.getInt(3);           // col 3 as int

    Lecturer teacher = new Lecturer(id, name, office);

    // do something with teacher
    // eg: store object in a list
  }

res.close();
```

ResultSet objects work like iterators

extract columns by integer index, or by column name, supplied as a string

getX() for each of several Java types X

Result Set

- Provides access to results of DB queries one row at a time
- Is a reference to the actual data, one row at a time
- The actual set of results may be very large

```
ResultSet res = stmt.executeQuery(
     "SELECT * FROM lecturer WHERE name='A Simons'");
  while (res.next()) {
    int id = res.getInt("id");     // access by col name
    String name = res.getString("name");
    int office = res.getInt("office");

    Lecturer teacher =
      new Lecturer(id, name, office);

    // do something with teacher
  }
```

id	name	office
15	A Simons	119
23	A Stratton	118

Data Conversion

```
class ResultSet {

  public int getInt(int column);
  public String getString(int columnIndex);
  public float getFloat(int columnIndex);
  public double getDouble(int columnIndex);
  public Date getDate(int columnIndex);
  public Time getTime(int columnIndex);
  …
  public int getInt(String columnLabel);
  public String getString(String columnLabel);
  …
  public Object getObject(int columnIndex);
  public Object getObject(String columnLabel);
}
```

Use specific access method where the type is known

Access using a column index or column name

Use the getObject() method if type is unknown

Performs a default type conversion, result is in a var of type Object

Lab 1: Execute Query

Run a Poll

- Sketch all the Java for a query
  - connect to the DB
  - create the Statement
  - execute a query seeking all Lecturers whose first initial is 'A' (assume a single name-field)
  - iterate through the ResultSet
  - return the result as a List<Lecturer>
- Remember to manage resources
  - always close the ResultSet
  - always close the Statement
  - always close the Connection

When to Commit?

- Commit changes immediately
  - good strategy in most cases
  - saves important updates, as these happen
  - fewer concurrent data access issues
    - avoid dirty reads – reading uncommitted changes
    - avoid non-repeatable reads – rows altered while reading
    - avoid phantom reads – rows inserted while reading
- Defer committing changes
  - execute a batch of insert, update jobs for efficiency
  - group sets of changes that logically belong together
    - notion of transactions (see later)

Control Auto-Commit

- Default JDBC setting
  - by default, every executeUpdate() commits immediately to the database
- Custom JDBC setting
  - can control the commit-point manually
  - check that DB engine supports delayed commit

```
con.setAutoCommit(false);      // turn off auto-commit
con.commit();                  // commit manually
con.setAutoCommit(true);       // turn on again
```

Manual Commit

```
con.setAutoCommit(false);
…
Statement stmt = null;
try {
   stmt = con.createStatement();
   int count = stmt.executeUpdate(
     "UPDATE lecturer SET office = 119"
        + " WHERE name = 'A Simons'");
   con.commit();
}
catch (SQLException ex) {
   ex.printStackTrace();
}
finally {
   if (stmt != null) stmt.close();
}
```

Updates are only committed at the manual commit-point.

Similar behaviour for all updates using this open Connection, unless you switch back

Transactions

- Transaction
  - a database transaction is a single, complete unit of work, which must either execute completely, or not at all
  - must be ACID: Atomic (all or nothing), Consistent (data integrity), Isolated (serializable), Durable (permanent)
- Protect against brief loss of service
  - wrap a set of updates in a transaction, if all updates must happen together (e.g. credit/debit in a money transfer)
  - the transaction must succeed, or fail, as a whole
  - upon failure, the database must rollback (forget temporary changes) or revert (recover the "before image" of the data)

Transaction Support

- MySQL support for transactions is engine-dependent
  - InnoDB engine (the default) supports ACID transactions
  - MyIASM engine only has atomic actions (auto commit=true)
  - BDB (Berkley DB) engine also supports transactions
- Can specify storage engine when defining a table
  - check this engine is available for your DB
  - use the ENGINE keyword in MySQL:

```
CREATE TABLE account(
  accno INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  holder VARCHAR(50),
  balance INT
) ENGINE=BDB;
```

or:

```
CREATE TABLE lecturer
(…) ENGINE=InnoDB;
```

Transactions in JDBC

```
con.setAutoCommit(false);     // turn off auto-commit
Statement credit = null;      // SQL statement objects
Statement debit = null;
try {
  credit = con.createStatement();
  debit = con.createStatement();
  debit.executeUpdate(“UPDATE account …”);
  credit.executeUpdate(“UPDATE account …”);
  con.commit();                // manually commit when ready
}
catch (SQLException ex) {
  if (con != null) con.rollback(); // if transaction fails
}
finally {
  if (credit != null) credit.close();
  if (debit != null) debit.close();
}
```

SQL Injection!

- Control access to data
  - ensure users have restricted views of data
  - ensure users have appropriate authorisation
- Validate all inputs
  - biggest mistake is failure to validate all inputs
  - allows SQL injection faults, could kill the DB
- Encrypt the data
  - if you really must!

injecting an extra SQL command
in a simple text entry field

Enter student ID:

```
0011234567; drop table
student
```

Prepared Statements

- For common queries with similar patterns
- Pre-compile a prepared statement with ? parameters
- Protects against injection: actual values ≠ SQL commands

```
// prepare beforehand

PreparedStatement pstmt = con.prepareStatement(
  "SELECT id, name FROM lecturer WHERE name=?");

// use later

pstmt.setString(1, "A Simons");
ResultSet res = pstmt.executeQuery();
…
pstmt.close();
```

A parameter in the pstmt is shown as ?

setX(nth, val) sets the nth param with val

Repeated Actions

```
// prepared statement with two parameters
// column 1 = null because of auto-increment

PreparedStatement pstmt = con.prepareStatement(
  "INSERT INTO lecturer VALUES (null, ?, ?)");

// iterate over a list of lecturers in memory and
// insert these into the database

for (Lecturer lect : department.getLecturers()) {
  pstmt.clearParameters();
  pstmt.setString(1, lect.getName());
  pstmt.setInt(2, lect.getOffice());
  int count = pstmt.executeUpdate();
}
…
pstmt.close();
```

Each parameter in the pstmt is shown as ?

clearParameters() clears all old values

setX(nth, val) sets the nth param with val

Concurrency Control

- Default strategy
  - JDBC uses the default strategy for the underlying DB
- Custom strategy
  - get and set using methods of the Connection class

```
int level = con.getTransactionIsolation();            // find the default setting
con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
                                                       // serialize transactions
```

Constants for levels of concurrency control:

TRANSACTION_NONE, TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, TRANSACTION_REPEATABLE_READ, TRANSACTION_SERIALIZABLE

Lab 2: Input Validation

Run a Poll

- Sketch the Java for your input validation
  - assume you have a Java Swing JTextField queryField;
  - this has a getText() method that returns the text that was entered into the queryField, as a String
  - you need to determine whether this String is healthy
- How will you check the input?
  - what String API functions can you use to look for SQL-injection attempts?
  - should you raise an exception if an attack is being made?
  - can you clean up the text String from the queryField?

MySQL Accounts

- Issuing group accounts
  - MySQL accounts are generated annually for group-work by the DCS support team
  - they send a list of new accounts to your lecturer who then has to allocate to team-leaders
  - team leaders watch out for this in your email inbox!
- Interpreting login data
  - the email will mention a DBname and a DBpassword
  - e.g. DBname=team043 and DBpassword=5a94128d
  - note DBname is used also as your team's userID.
  - need a DBname, userID and DBpassword to connect

MySQL Client

- MySQL command-line client (shell, monitor)
  - be connected to the University's VPN
  - in Linux, open a terminal or console

```
% mysql -hstusql.dcs.shef.ac.uk –uteam043 -p team043
Enter password: *********
Welcome to the MySQL monitor.  …
```

  - -h specifies the host, the database server
  - -u specifies the userid, your team name
  - -p prompts for your password on the next line (why?)
  - the last argument is the database name (why?)

Using the Client

- Type any SQL instructions at the prompt

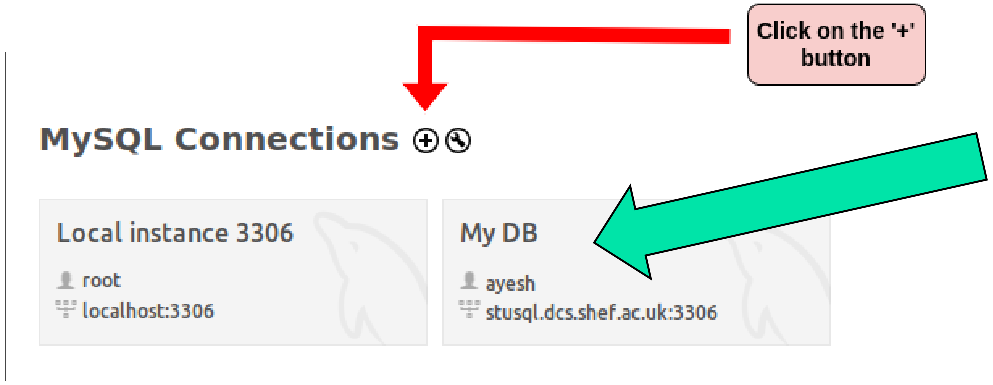
```
% mysql -hstusql.dcs.shef.ac.uk –uteam043 -p team043
Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g
…
mysql> CREATE TABLE Student (
     ->   regno INTEGER NOT NULL PRIMARY KEY,
     ->   forename VARCHAR(30),
     ->   surname VARCHAR(30));
Query OK, 0 rows affected (0.42 sec)

mysql>
```

prompts with -> until the command is terminated

MySQL Workbench

- MySQL GUI-based client, for Windows (see Lab for Linux)
  - https://dev.mysql.com/downloads/workbench/
  - Version: MySQL Workbench 8.0.34
  - Remember, connection to DCS servers requires VPN first



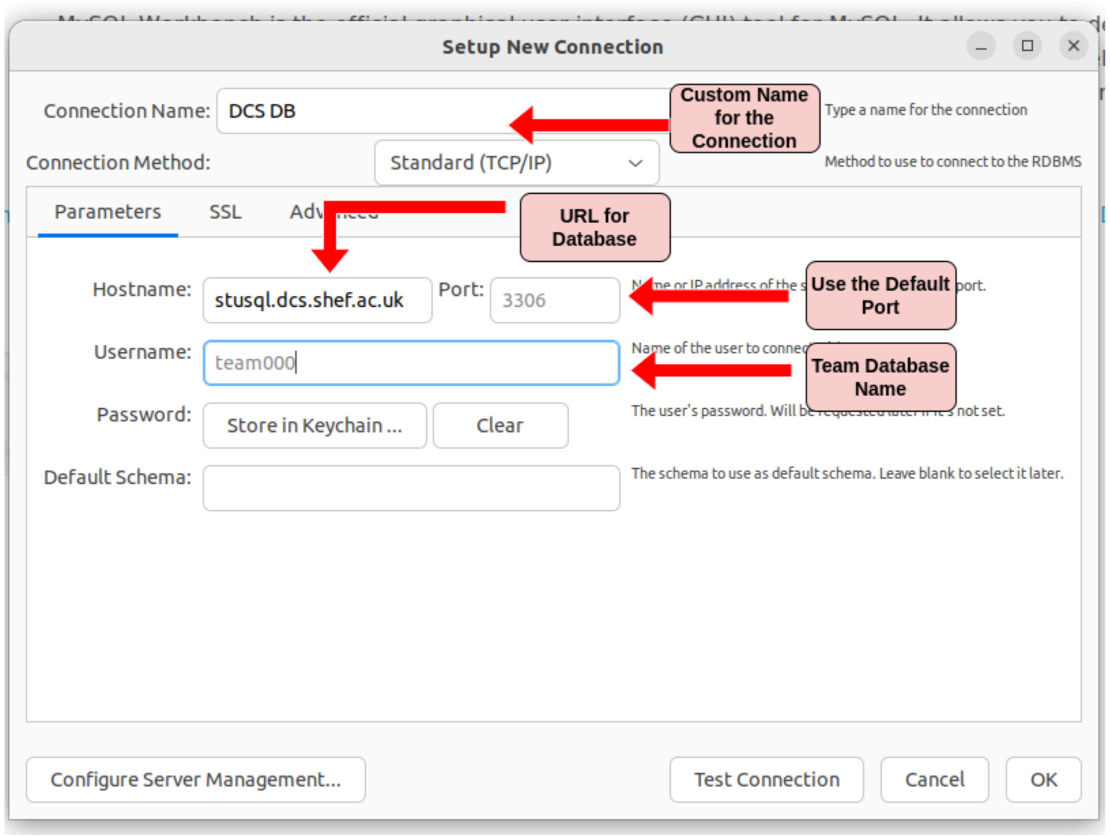
Local instance 3306
root
localhost:3306

My DB
ayesh
stusql.dcs.shef.ac.uk:3306

The first time you use, you won't see the list of MySQL connections shown here.

But click on (+) to get the following...

Configuring


Setup New Connection

Connection Name: DCS DB — Type a name for the connection

Connection Method: Standard (TCP/IP) — Method to use to connect to the RDBMS

Parameters | SSL | Advanced

Hostname: stusql.dcs.shef.ac.uk Port: 3306 — Name or IP address of the server host and TCP/IP port.

Username: team000 — Name of the user to connect with.

Password: Store in Keychain ... Clear — The user's password. Will be requested later if it's not set.

Default Schema: — The schema to use as default schema. Leave blank to select it later.

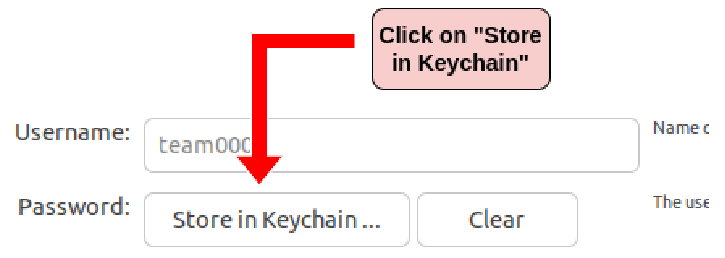
Give the connection any name you like

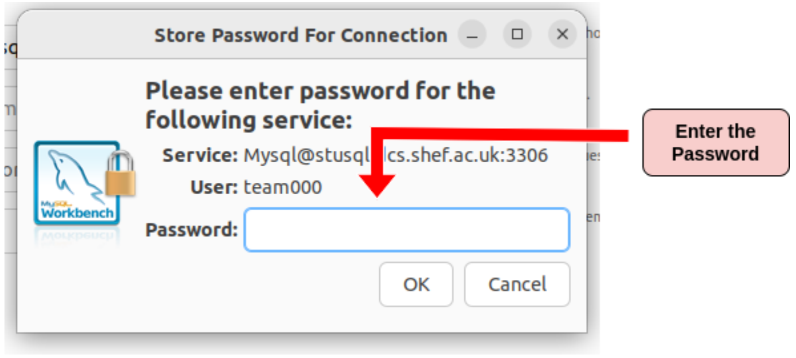
Use the standard TCP/IP

Host is stusql.dcs.shef.ac.uk
Port is 3306

Username is your team DB name

Security



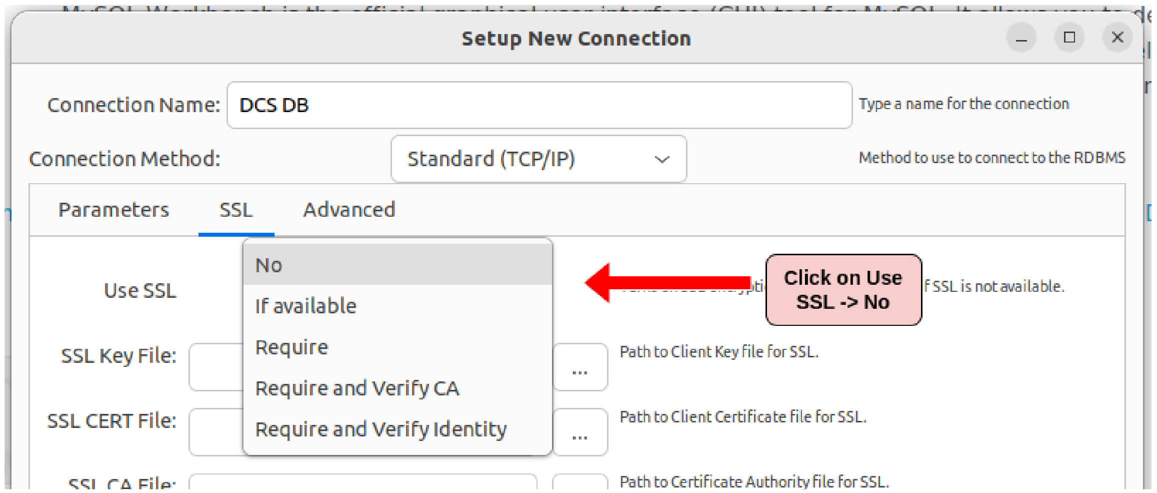


Please enter password for the following service:

Service: Mysql@stusql.dcs.shef.ac.uk:3306

User: team000

Password:



Connection Name: DCS DB — Type a name for the connection

Connection Method: Standard (TCP/IP) — Method to use to connect to the RDBMS

Use SSL: No — if SSL is not available.

SSL Key File: — Path to Client Key file for SSL.

SSL CERT File: — Path to Client Certificate file for SSL.

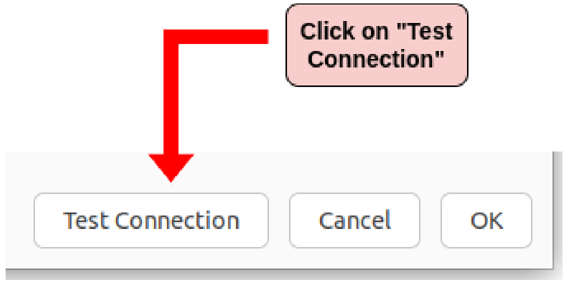
SSL CA File: — Path to Certificate Authority file for SSL.

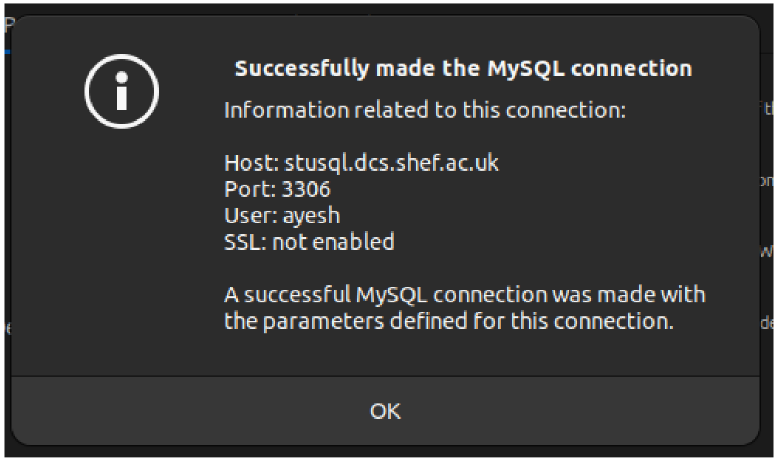
Click on "store in keychain" to enter team password

Click on SSL tab and select No

(stustore does not support SSL connections)

Connecting



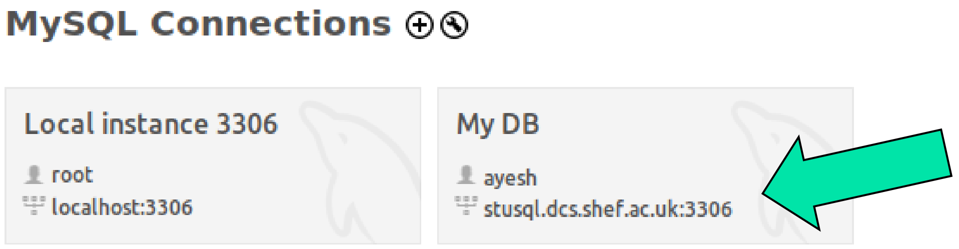


Successfully made the MySQL connection

Information related to this connection:

Host: stusql.dcs.shef.ac.uk
Port: 3306
User: ayesh
SSL: not enabled

A successful MySQL connection was made with the parameters defined for this connection.



Click on Test Connection to finish set-up

Next time, just click on the button for the configured DB

Summary

- Java and databases have different strengths of type
- MySQL is the best free database to use on Windows and Linux
- JDBC is the bridge between Java and any database, using a suitable driver which is easy to install
- Connections, statements, result sets must always be closed using auto-closing resource management or the try-catch-finally idiom
- Prepared statements, manual commit and robust transactions are possible with some DB engines
- Concurrency control is possible, but default settings are OK
- Guard against SQL injection through input validation!