

QDL DB

Introduction

The QDL database extension. A module that allows access to various databases. This is designed not to be a full fledged database application, but a tools module that allows for all the basic access to various databases with the grunt work of converting between types as well as having a way to work seamlessly with native SQL types. Typically you would write your database application using this module.

Loading the module

To load the module, invoke

```
q := module_load('edu.uiuc.ncsa.qdl.extensions.database.QDLDBModule', 'java')
module_import(q)
```

Supported functions

Name	Description	Comment
connect(cfg.)	Open a connection to the database	All other requests will fail until this is called
execute(stmt{, args.})	Execute a statement with no result	
read(stmt{, args.})	Execute a statement with a result	
update(stmt{, args.})	Update an entry in the database	

Variables

types. - a stem of names and integer values,

```
types.
{NUMERIC:2, FLOAT:6, BLOB:2004, LONGVARCHAR:-1, CLOB:2005, ARRAY:2003, BINARY:-2,
CHAR:1, BIGINT:-5, TIME:92, BIT:-7, DATE:91, REF:2006, SQLXML:2009, SMALLINT:5,
TIMESTAMP:93, VARCHAR:12, REAL:7, VARBINARY:-3, DOUBLE:8, STRUCT:2002, TINYINT:-6,
INTEGER:4}
```

These are internal values and should not be altered.

Arguments

The statement in each of the calls above is a string. It may be either hard coded such as

```
select * from my_table where id='42'
```

which would be issued as

```
db#read('select * from my_table where id=\'42\');
```

Or it may be prepared with ? signs replacing the arguments and a list of arguments and possibly their types supplied.

QDL tries to be helpful, in that if you supply no SQL type, it will be inferred, so a string will be treated as if it is a string.

That said, databases can have any number of oddities so if you need a specific SQL type (e.g. you have a column that is a tinyint) then by all means specify it. In this case, if the argument were a tiny int, you would issue

```
db#read('select * from my_table where id=?', [42, types.TINYINT]);
```

Prepared statements are also extremely useful so you don't have to do a lot of escaping of quotes. For instance to do a search using a regex might look like

```
db#read('select client_id from oauth2.clients where client_id regexp ?',  
['.*123.*'])  
{client_id:oa4mp:/client_id/7142f3461239deb57d98ba3a4636}
```

Connecting to the database

The basic way to do this is to create a stem of values and pass that to the connect function. Supported values are

Name	Description	Comment
username	The user name	
password	The password	
database	The name of the database	In Derby this is the path
schema	The schema	
host	The host	
port	The port	Standard ports are 3306 for maria DB and mysql and 5432 for postgres. Derby does not use ports
parameters	Specific connection parameters	These are very vendor specific
useSSL	Use SSL for the connection	Make <i>sure</i> you have set up SSL correctly first!
bootPassword	The boot password	Derby only
inMemory	Run in memory only	Derby. Note that this database will

		vanish as soon as you exit QDL.
type	The type of the connection	One of mysql, mariadb, postgres or derby

An Example

```

cfg.'username' := 'qdl-user';
cfg.'password' := 'w00fity';
cfg.'schema' := 'qdl_test';
cfg.'database' := 'qdl_test';
cfg.'host' := 'localhost';
cfg.'port' := 3306;
cfg.'type' := 'mariadb';
db#connect(cfg.);
true

```

This indicates that a connection to the database was made. Here is a test to count the rows in one of the tables

```

db#read('select count(*) from transactions');
{count(*):161}

```

The result is a stem. Note that the database engine itself returned the name of the result as '**count(*)**' and this may vary by vendor. In any case, there are 161 entries in the given table for the given database.