Complex Systems Theory and Practice

Lecture 9

Piotr Syga

2021-12-10

Users

- similarly to SQL databases MongoDB allows creating users and managing their privileges
- the privileges are usually handled by roles
- users may authorize using password or X.509 certificate, LDAP or Kerberos – when not using a password the user has to be created in $external database

User creation

```
db.createUser({user: '<name>', pwd:passwordPrompt(), roles:
[{role:'readWrite', db: '<database>'}, { role:'read', db:
'<otherDatabase>'}]})
```

Loging in

```
mongosh --port 27017 -u '<name>' --authenticationDatabase
'<database>' -p
```

Users

Creating a role

```
use admin
db.createRole({role: '<name>', privileges: [{ resource: {cluster: true},
actions: ["killop", "inprog"]}, { resource: {db: , collection: }, actions:
["killCursors"]}], roles: []})
```

The `cluster` resource is used for actions that affect the state of the system.
More examples of managing the users and roles can be found in the manual.

Client-Side Field Level Encryption

- Since MongoDB 4.2 Client-side encryption of selected files has been enabled.
- The encryption supports locally stored keys as well as external KSMs (e.g., Amazon Web Services KMS, Azure Key Vault, Google Cloud Platform KMS).
- The encrypted data as stored as subtype 6 of BinData.
- To force encryption, one has to use `encrypt` option indicating the key used (by UUID) and the algorithm of the encryption inside the field to be encrypted.
- Forced encryption is always executed the encryption on the fields where it is not indicated is optional.

Client-Side Field Level Encryption

Forced encryption example

```
db.getSiblingDB('hr').runCommand({
'collMod' : 'employees', 'validator' : {
'$jsonSchema' : { 'bsonType' : 'object', 'properties' : {
'taxid' :{'encrypt' : { 'keyId' :
[UUID('e114f7ad-ad7a-4a68-81a7-ebcb9ea0953a')], 'algorithm' :
'AEAD_AES_256_CBC_HMAC_SHA_512-Random',}},
'taxid-short' : {'encrypt' : { 'keyId' :
[UUID('33408ee9-e499-43f9-89fe-5f8533870617')], 'algorithm' :
'AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic', 'bsonType'
: 'string' }
}}}}})
```

Security checklist

Important notions to remember while creating external MongoDB database:

- Enable Access Control and Enforce Authentication
- Configure Role-Based Access Control
- Encrypt Communication
- Encrypt the Data
- Limit Network Exposure
- Audit System Activity
- Run MongoDB with a Dedicated User
- Run MongoDB with Secure Configuration Options

Enforcing partial schema

- we can enforce a partial scheme of the document by using a `validator` that works on a collection level
- `validator` may be initiated as an option while creating a new collection or by using `collMod` function
- for each rules we can specify additional options `validationLevel` and `validationAction`

Enforcing partial schema – example

```
db.createCollection('students', {
   validator: {
      $jsonSchema: {
         bsonType: 'object',
         required: [ 'name', 'year', 'major', 'address' ],
         properties: {
            name: {
               bsonType: 'string',
               description: 'must be a string and is required'
            },
            year: {
               bsonType: 'int',
               minimum: 2000,
               maximum: 2099,
               description: 'must be an integer in [ 2000, 2099 ] and is required'
            },
            major: {
               enum: [ 'BDA', 'FT', 'IKw', 'Op','IB', null ],
               description: 'can only be one of the enum values and is required'
            }
         }
      }
   }
})
```

Examples of API

- ODBC
- JDBC
- OLEDB
- ADO.NET data provider
- REST API / CRUD

Functionalities

Main components

- Connector
- Command Executor
- Data Reader
- Data Adapter

Types of queries

Based on the handling of the queries we can derive 3 types:

- Statement
- PreparedStatement
- CallableStatement

Drivers

JDBC requires a driver in order to connect between Java interface and a database. Historically the drivers might be divided into 4 types:

Drivers

JDBC requires a driver in order to connect between Java interface and a database. Historically the drivers might be divided into 4 types:

Type 1 – JDBC – ODBC connector that converts JDBC methods to universal API
Example: `sun.jdbc.odbc.JdbcOdbcDriver`
+: universality
-: deprecated in current JDK, time overhead due to the need of translation, ODBC is required

Type 2 – native API (client-side packages)
Example: Oracle Call Interface
+: less time overhead than Type 1
-: requires client side libraries for a specific DBE that depends on platform and cannot be used as an applet or any 'closed environment app'

Drivers

JDBC requires a driver in order to connect between Java interface and a database. Historically the drivers might be divided into 4 types:

Type 3 – using a mediator
Example: `com.jw.client`, `com.jw.server`, `com.jw.client.JWDriver`, `ids.sql IDS driver`
+: independent of the platform used, independent of the DBE, universal mediator has bindings to each side
-: requires additional entity, communication (and time) overhead, the mediator requires specific bindings

Type 4 – direct communication in Java
Example: `oracle.jdbc.driver.OracleDriver` (:thin:)
+: fast, platform independent, lack of mediator, allows java debugging
-: separate drivers for each DBE, some that are not open and under severe licensing

Oracle drivers, MySQL Connector/J

Example – Select statement

```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
public class JdbcExample {
   public static void main(String[] args) {
      try (
        Connection conn = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/CSdatabase", "user", "pass");
         Statement stmt = conn.createStatement();
      ) {
         String testquery = "Select name, lastname, score From Tests";
         ResultSet res = stmt.executeQuery(testquery);
         int rowCount = 0;
         while(res.next()) {
            String n  = res.getString("name");
            String l  = res.getString("lastname");
            int    score = res.getInt("score");
            /*
                        ...
               */
            ++rowCount;
         }
      } catch(SQLException ex) {
         ex.printStackTrace();
      }
   }
}
```

Example – Update, Insert, Delete

```
import java.sql.*;

public class JdbcExampleB {
   public static void main(String[] args) {
      try (
        Connection conn = DriverManager.getConnection(
               "jdbc:mysql://localhost:3306/CSdatabase", "user", "pass");
         Statement stmt = conn.createStatement();
      ) {
String upquery = "Update Tests Set mark = 'Ok' Where score>50";
int upcounter = stmt.executeUpdate(upquery);
      System.out.println(upcounter + " passed");
String delquery = "Delete From Tests Where score is NULL";
 int delcounter = stmt.executeUpdate(delquery);
String inquery = "Insert Into Tests (name, lastname, score) "
               + "values ('John', 'Doe', 7),"
     + "('Joe','Smith',80)";
stmt.executeUpdate(inquery);
      } catch(SQLException ex) {
         ex.printStackTrace();
      }
   }
}
```

Example – Creating tables

```
import java.sql.*;

public class JdbcExampleC {
   public static void main(String[] args) {
      try (
        Connection conn = DriverManager.getConnection(
         "jdbc:mysql://localhost:3306/CSdatabase", "user", "pass");
         Statement stmt = conn.createStatement();
      ) {
 String cdb = "Create Database Musictmp";
stmt.executeUpdate(cdb);
/* ... */
String ctab =
        "Create Table Musictmp.albums " +
        "(name varchar(50) NOT NULL, " +
        "band int NOT NULL, " +
        "release date, " +
        "PRIMARY KEY (name,band), " +
        "FOREIGN KEY (band) REFERENCES " +
        "tmp.bands(id))";
stmt.executeUpdate(ctab);
      } catch(SQLException ex) {
         ex.printStackTrace();
      }
   }
}
```

PreparedStatement – a recall

We can create (almost) universal queries for Update, Insert, ...

```
String AlmostUniversalUpdateStatement =
        "Update ?" +
        "Set col1 = ? " +
        "Where ID = ?";

try {
con.setAutoCommit(false);
upStmt = con.prepareStatement(AlmostUniversalUpdateStatement);
upStmt.setString(1, 'Tab1');
upStmt.setInt(2, 100);
upStmt.setInt(3, 1);
upStmt.executeUpdate();
upStmt.setString(1, 'Tab1');
upStmt.setInt(2, 73);
upStmt.setInt(3, 2);
upStmt.executeUpdate();
con.commit();
 } catch (SQLException e ) {
        /* ... */
}
```

Handling errors – Oracle docs

```
public static void printSQLException(SQLException ex) {
    for (Throwable e : ex) {
        if (e instanceof SQLException) {
            if (ignoreSQLException(
                ((SQLException)e).
                getSQLState()) == false) {
                e.printStackTrace(System.err);
                System.err.println("SQL State: " +
                    ((SQLException)e).getSQLState());

                System.err.println("Error Code: " +
                    ((SQLException)e).getErrorCode());

                System.err.println("Message: " + e.getMessage());

                Throwable t = ex.getCause();
                while(t != null) {
                    System.out.println("Cause: " + t);
                    t = t.getCause();
                }
            }
        }
    }
}
```

Further reading

- datatypes, tables and their properites, arrays, structures and maps
- procedures
- transactions
- Process, Runtime (e.g. msqldump)
- SQLXML

Packeges

Aside from the interpreter and DBE we require specific database libs, e.g.:

- `python-mysqldb`
- `python-mysqldb-dbg`

Connector and the interfaces are also availible on python's and SQL dialect pages.

Connecting the database

```
#!/usr/bin/python

import _mysql
import sys

try:
    con = _mysql.connect('localhost', 'user', 'pass', 'CSdatabase')

    con.query("SELECT VERSION()")
    result = con.use_result()

    print "MySQL version: %s" % \
        result.fetch_row()[0]

except _mysql.Error, e:

    print "Error %d: %s" % (e.args[0], e.args[1])
    sys.exit(1)

finally:

    if con:
        con.close()
```

Cursors

```
#!/usr/bin/python

import MySQLdb as mdb
import sys

try:
    con = _mysql.connect('localhost', 'user', 'pass', 'CSdatabase')

    kursor = con.cursor()
    kursor.execute("Select SecurityNo, salary From staff")
    person, sal = kursor.fetchone()
    while person is not None:
        print "Employees: %s %d" % (person, sal)
        person, sal = kursor.fetchone()

except mdb.Error, e:

    print "Error %d: %s" % (e.args[0], e.args[1])
    sys.exit(1)

finally:

    if con:
        con.close()
```

Cursors

```
#!/usr/bin/python

import MySQLdb as mdb
import sys

try:
    con = _mysql.connect('localhost', 'user', 'pass', 'CSdatabase')

    kursor = con.cursor()
    kursor.execute("Select SecurityNo, salary From staff")
    person, sal = kursor.fetchone()
    while person is not None:
        print "Employees: %s %d" % (person, sal)
        person, sal = kursor.fetchone()

except mdb.Error, e:

    print "Error %d: %s" % (e.args[0], e.args[1])
    sys.exit(1)

finally:

    if con:
        con.close()
```

What happens to the `kursor` after the last row is fetched?

`kursor.close()`

Cursors – useful methods

- `cursor.description`
- `cursor.rowcount`
- `cursor.statement`
- `cursor.column_names`
- `cursor.callproc`

cursor.execute

```
stmt = "SELECT * FROM staff WHERE SecurityNo = %(p)s"
cursor.execute(stmt, { 'p': '66010100000' })

stmt = "INSERT INTO staff (pesel, salary) VALUES (%s, %s)"
data = [
  ('77010100000', 4000),
  ('88010100000', 3000),
  ('99010100000', 2000),
]
cursor.executemany(stmt, data)
```

Further reading

- arrays, assosciative tables
- procedures, prepared statemnts
- transactions
- runtime

Selected languages

- C++
- Haskell
- NodeJS
- Julia
- Go