Day 19 Revisit

JDBC – Java Database Connectivity

It has many interfaces and classes which will help us to interact with any RDBMS

RDBMS – Relational DataBase Management System.

In RDBMS, everything is entity. (Table, View, Sequence, Trigger, Function, Stored Procedure, Materialized Views)

In Java, everything is Object.

JDBC is an API (Application Programming Interface) designed Java developers.

JDBC API is used to interact with any Database (Oracle, MySQL, Postgres, DB2, MS-SQL etc.,)

5 Steps to interact with the Database

1. Load & Register the Driver (The name of the driver class will differ for each database)
2. Establish the connection between java & Database (URL, username, password)
3. Create and Execute the Query using Statement or PreparedStatement
4. Store & process the result using resultset.
5. Close all the resources.

|  |  |
| --- | --- |
| **Driver class Name** |  |
| MySQL | com.mysql.cj.jdbc.Driver |
| Oracle | oracle.jdbc.driver.OracleDriver |
| Postgres | Org.postgresql.Driver |
| **Connection URL /String** |  |
| MySQL | jdbc:mysql://localhost:3306/<db\_name> |
| Oracle | jdbc:oracle:thin:@localhost:1521:xe |
| Postgres | Jdbc:postgres://localhost:5432/<db\_name> |
| **Username/ Password** |  |
| MySQL | root/root |
| Oracle | Admin/manager or hr/hr |
| Postgres | postgres/postgres |
|  |  |

All the classes & interfaces are defined in **java.sql** package

Driver(I), Connection(I), ResultSet(I), Statement(I), PreparedStatement(I), CallableStatement (I)

DatabaseMetaData(I), ResultSetMetaData(I)

DriverManager[C], Date[C], Time[C]

<https://www.postgresqltutorial.com/>

**Agenda**

Joins, Types of Joins

ACID properties of Database (Atomicity, Consistency, Isolation & Durability)

Index & Views

Conn.setAutoCommit(true) ; --- is the default value

Conn.setAutoCommit(false); --- The changes made to the database will not be committed automatically.

Conn.commit();



| **RDBMS** | **JDBC driver** | **URL format** |
| --- | --- | --- |
| MySQL | com.mysql.cj.jdbc.Driver | jdbc:mysql://hostname/databaseName |
| Oracle | oracle.jdbc.driver.OracleDriver | jdbc:oracle:thin:@hostname:portNumber:databaseName |
| SQLServer | com.microsoft.sqlserver.jdbc.SQLServerDriver | jdbc:sqlserver://serverName:portNumber;property=value |
| PostgreSQL | org.postgresql.Driver | jdbc:postgresql://hostname:port/databaseName |

Types of SQL Queries

1. Simple Query ( simple select query, counting the rows/records of the table)
2. Query with parameters (Insert query, update query, delete query)
3. Sub – Query (Query written within another query)
4. Join queries (To combine column from two or more table)

Types of Statements in JDBC API

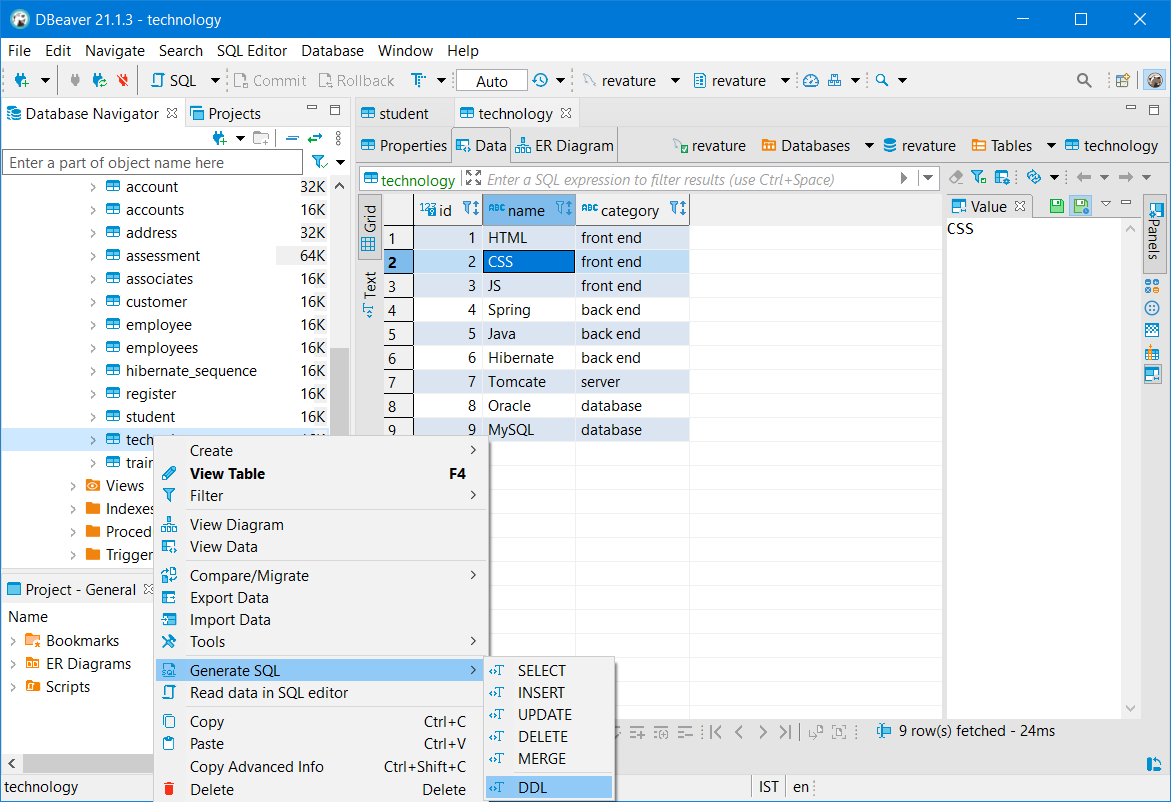
1. Statement [I] -- For Executing simple queries
2. PreparedStatement [I] – For Executing Queries with parameter
3. CallableStatement [I] – For executing Function & Stored Procedures

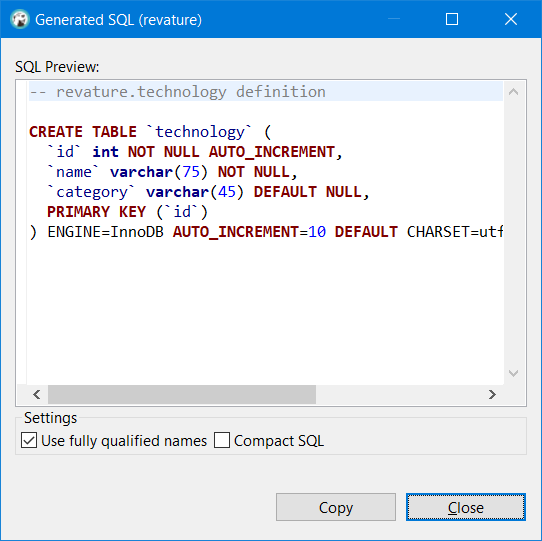
Few important methods in JDBC Statement/PreparedStatement Interfaces

Execute() - return a Boolean value

executeUpdate() – return primitive int value (0 or no or row affected)

executeQuery() - return resultset





**CREATE** **TABLE** `technology` (

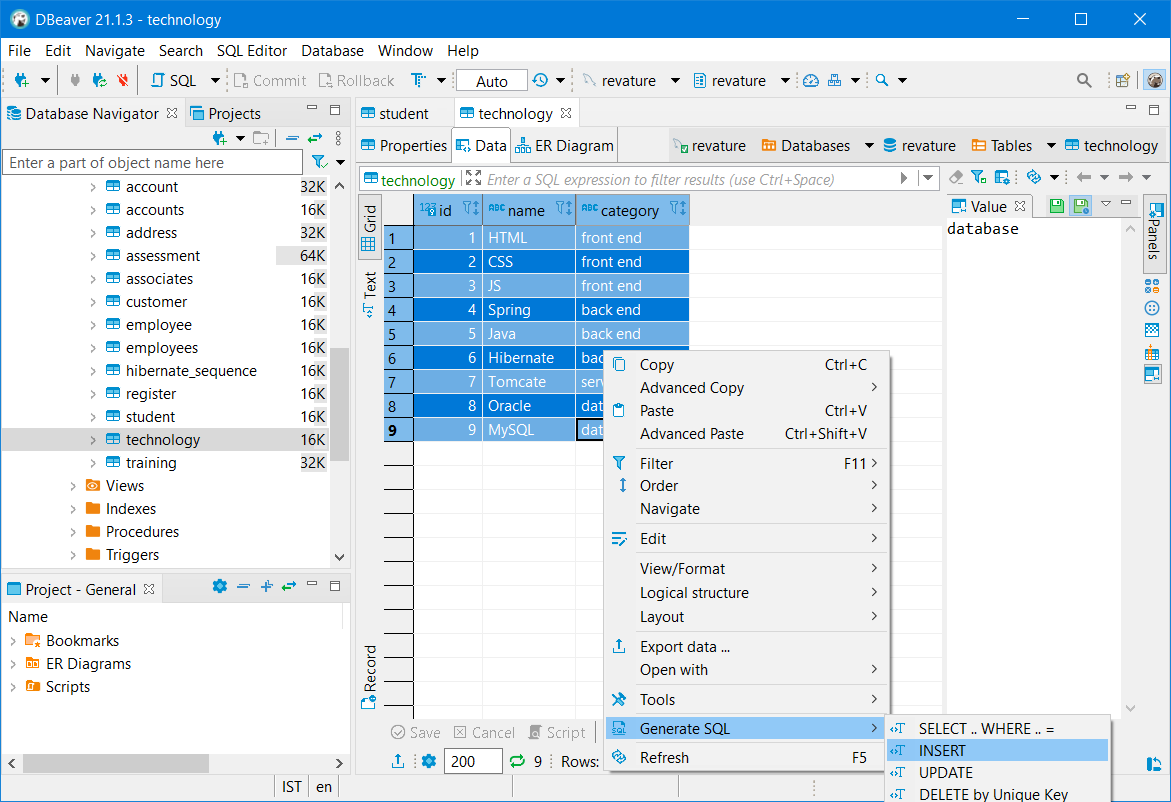
`id` **int** **NOT** **NULL** **AUTO\_INCREMENT**,

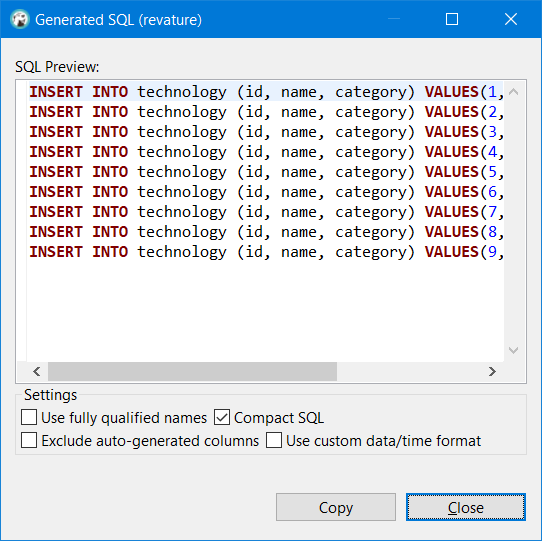
`name` **varchar**(75) **NOT** **NULL**,

`category` **varchar**(45) **DEFAULT** **NULL**,

**PRIMARY** **KEY** (`id`)

)





**INSERT** **INTO** revature.technology (id, name, category) **VALUES**(1, 'HTML', 'front End');

**INSERT** **INTO** revature.technology (id, name, category) **VALUES**(2, 'CSS', 'front end');

**INSERT** **INTO** revature.technology (id, name, category) **VALUES**(3, 'JS', 'front end');

**INSERT** **INTO** revature.technology (id, name, category) **VALUES**(4, 'Spring', 'back end');

**INSERT** **INTO** revature.technology (id, name, category) **VALUES**(5, 'Java', 'back end');

**INSERT** **INTO** revature.technology (id, name, category) **VALUES**(6, 'Hibernate', 'back end');

**INSERT** **INTO** revature.technology (id, name, category) **VALUES**(7, 'Tomcate', 'server');

**INSERT** **INTO** revature.technology (id, name, category) **VALUES**(8, 'Oracle', 'database');

**INSERT** **INTO** revature.technology (id, name, category) **VALUES**(9, 'MySQL', 'database');

**INSERT** **INTO** technology (id, name, category) **VALUES**(1, 'HTML', 'front end');

**INSERT** **INTO** technology (id, name, category) **VALUES**(2, 'CSS', 'front end');

**INSERT** **INTO** technology (id, name, category) **VALUES**(3, 'JS', 'front end');

**INSERT** **INTO** technology (id, name, category) **VALUES**(4, 'Spring', 'back end');

**INSERT** **INTO** technology (id, name, category) **VALUES**(5, 'Java', 'back end');

**INSERT** **INTO** technology (id, name, category) **VALUES**(6, 'Hibernate', 'back end');

**INSERT** **INTO** technology (id, name, category) **VALUES**(7, 'Tomcat', 'server');

**INSERT** **INTO** technology (id, name, category) **VALUES**(8, 'Oracle', 'database');

**INSERT** **INTO** technology (id, name, category) **VALUES**(9, 'MySQL', 'database');

**Statement and PreparedStatement Interfaces**

Statement interface is used for executing static SQL statements. PreparedStatement interface is used for executing pre-compiled SQL statements.

FileInputStream fileStream = new FileInputStream("pathtopropertiesfile");

Properties properties = new Properties();

properties.load(fileStream);

URL = properties.getProperty("URL");

CONNECTION\_PASSWORD = properties.getProperty("CONNECTION\_PASSWORD");

CONNECTION\_USERNAME = properties.getProperty("CONNECTION\_USERNAME");

## JDBC Connection

When making a JDBC connection to a database you may have noticed the code typically looks like this.

DriverManager.getConnection(URL, CONNECTION\_USERNAME, CONNECTION\_PASSWORD);

But, where are we getting those values? Well up until now you have probably seen them as plain text Strings. As you can imagine this is not secure and requires you to change the code anytime the password etc. is updated.

To solve this problem developers began to collect all of this information into properties files. A properties file stores information as key value pairs, each on their own line; the file has a .properties extension. For instance you might have,

URL=jdbc:postgresql://localhost:5432/PubHub

CONNECTION\_USERNAME=user

CONNECTION\_PASSWORD=password

Then you can use the Properties class and FileInputStream to use these properties in your class.

FileInputStream fileStream = new FileInputStream("pathtopropertiesfile");

Properties properties = new Properties();

properties.load(fileStream);

URL = properties.getProperty("URL");

CONNECTION\_PASSWORD = properties.getProperty("CONNECTION\_PASSWORD");

CONNECTION\_USERNAME = properties.getProperty("CONNECTION\_USERNAME");

Great! So this solves some of our problem, but we are still saving our information as plain text. Now it's just in a different kind of file.

**Aside**: If you are using this method to read in properties for a web application that will be deployed on a server, you may need to use a different approach. When reading in a file, if you use a relative path it will be relative to the working directory. This can be unpredictable when it comes to servers, so instead we can use this method in those instances.

Properties prop = new Properties();

try {

InputStream dbProps = ConnectionUtil.class.getClassLoader().getResourceAsStream("database.properties");

prop.load(dbProps);

} catch (Exception e) {

LogUtil.logException(e, ConnectionUtil.class);

}

So the next step is to include the information as System Environment Variables instead of just plain text.

Unfortunately Java on its own will not parse our System environment variables into the application.properties file. When you begin using certain frameworks like Spring you may be able to change your .properties file to look like this.

URL=${URL}

CONNECTION\_USERNAME=${CONNECTION\_USERNAME}

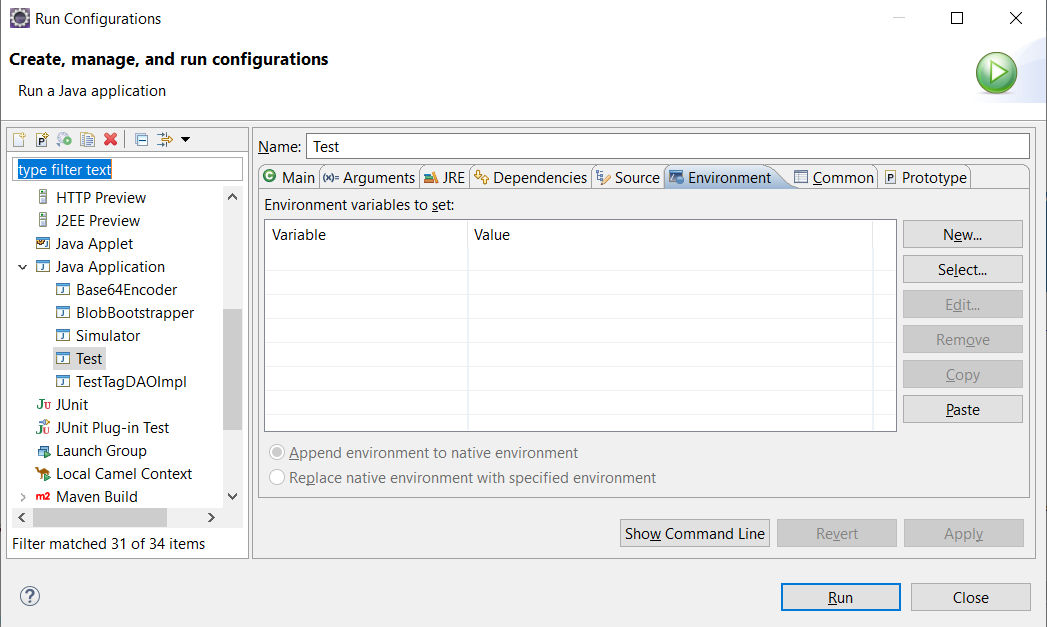
CONNECTION\_PASSWORD=${CONNECTION\_PASSWORD}

However for now we can simply use our properties to read in values from the System through the System library.

URL=url

CONNECTION\_USERNAME=connectionUsername

CONNECTION\_PASSWORD=connectionPassword

Then on the system we would need to make sure to create url, connectionUsername, and connectionPassword respectively. You can do this in Eclipse if you don't want to set them system wide. Simply go to Run Configurations and tab over to the Environment tab.

Then in our Java class we'll use our values to read in from the System variables.

FileInputStream fileStream = new FileInputStream("pathtopropertiesfile");

Properties properties = new Properties();

properties.load(fileStream);

String url = properties.getProperty("URL");

String password = properties.getProperty("CONNECTION\_PASSWORD");

String username = properties.getProperty("CONNECTION\_USERNAME");

URL = System.getenv(url);

CONNECTION\_PASSWORD = System.getenv(password);

CONNECTION\_USERNAME = System.getenv(username);

And then we should be good to go!

Design pattern used in Database Interaction – DAO

DAO – Data Access Object ( It’s a Class used to access the content of the database table)

DAO – this is a class where we write the code to perform CRUD operation on a particular table

Student – Entity Bean Class , StudentDAO – It’s a class where CRUD operation Code on Student table will be written (findAll(), findById(int id), save(Object obj) –[Insert/Update], edit(Object obj), delete(int id))

DAO class in some places called as repository (In Spring boot)

## Read Phenomena

* Dirty Read: reading data that is uncommitted
* Non-repeatable read: when a row is read twice in a transaction and the values are different
* Phantom Read: reading data that is being added or modified by a running transaction

Joins help us to combine the data from two or more tables.

Types of Joins

1. Inner Join
2. Outer Join
3. Cross Join
4. Self Join
5. Natural Join