4. Connecting to a Database

Contents

[1. Introduction 1](#_Toc23111)

[2. Closing Connections 3](#_Toc20864)

[3. Demonstration - Closing Connections 3](#_Toc3449)

[4. Demonstration: JDBC – Create, Select And Delete MySQL Database 5](#_Toc3859)

[CREATING DATABASE 5](#_Toc23144)

[SELECT DATABASE 6](#_Toc30192)

[DELETING DATABASE 6](#_Toc8433)

[5. Summary 6](#_Toc1173)

# 1. Introduction

=>slides: Pg. 1

This is the Connecting to a Database module. So in the previous module, we looked at the JDBC interfaces, and we saw things like PreparedStatement and CallableStatement. And we also looked briefly at how we connect to a database to get hold of those interfaces, and so allow us to access the data within the database. In this module, we'll go into more detail in how we create and manage that connection.

=>slides: Pg. 2

So in the previous module, we glossed over the URL, this JDBC URL. In this module, we'll go into more detail as to what makes up that URL, and look at some examples of JDBC URLs. We'll also look at gaining more detail at how we create that initial connection to the database, and see that once we have that connection, we can then run statements against the database.

=>slides: Pg. 3

So we saw this in the previous module. To get a connection, we define the URL that looks something like this, and we'll see in a moment what this means. We then call DriverManager.getConnection. The DriverManager takes that URL and essentially offers it to classes on its class path, loading the first one that recognizes that URL. So in this case the URL starts with jdbc:mysql, so hopefully the MySQL JDBC driver will recognize the URL, load itself into memory, and then create a connection to the database defined by the rest of the URL. In earlier versions of Java, we had to load these classes explicitly ourselves before we called getConnection, but that's changed, so the classes now load themselves when they see a URL that they recognize.

=>slides: Pg. 4

So what are the parts of this URL? So a JDBC URL always starts with the four characters jdbc. This, if you like, is the protocol for the URL.

=>slides: Pg. 5

The characters between the two colons are known as the subprotocol, and this is the vendor or product name. So in this case, it's mysql, that could be Oracle or Postgres or SQL Server or some vendor‑specific name that the driver defined for that vendor will recognize.

=>slides: Pg. 6

And then the last part of the URL is the database‑specific connection string. So here we're saying, connect to a mysql database, running on localhost, listening on port 3306. You don't need the port if it's the default port. And then for the MySQL JDBC URL, the last part of this connection string is the name of the schema within the MySQL database that we want to connect to.

=>slides: Pg. 7

So here are some examples of JDBC URLs. So the second one is the one that we've seen, jdbc:mysql://localhost:3306/loboticket. The top one would allow us to connect to PostgreSQL, and again, with that same database, so in this case the URL is jdbc:postgresql. The third one is for the Oracle Thin driver, so it's jdbc:oracle, and then thin and the address of the database. And then, just as another example, the bottom one is for the derby in‑memory database. So whether it's an in‑memory database or a full‑blown enterprise database, the URLs look similar. In this case, jdbc:derby is the name of the vendor or the name of the product, and then the name of the database coming at the end of the URL, in this case.

=>slides: Pg. 8

So how do we use this? And again, we saw this in the previous module. We call DriverManager.getConnection and pass in the value of this URL. The DriverManager looks for a class that recognizes this URL, and that class obviously has to be on the class path. In this case, the MySQL driver would recognize the URL. Once it was loaded into memory, it would then try and connect to the database using the final part of the URL, so localhost:3306/loboticket. So one thing that I didn't point out in the previous module, but that is very, very important, is the fact that many methods in JDBC throw SQL exceptions. So getConnection, if it fails, will throw a SQL exception, so what we're doing here is just handling that exception on the main method. Within your code, you would need to handle that exception whichever way you felt fit. That might just be logging errors, that might be ending the application, that might be telling the user that they can't connect to the database for whatever reason. But remember, for all of these things, whether it's getConnection, whether it's when we create PreparedStatements, CallableStatements, execute queries, they will all throw a SQL exception, and you will need some way of handling that.

=>slides: Pg. 9

Okay, so getConnection has overloaded methods. On the previous slide, we saw we can just pass it the URL. The database I'm using for these demonstrations also requires a username and a password, and you can pass those here into getConnection. So my username is loboticket, and my password is the very strong p4ssw0rd. And again, this will work by getting me a connection to that database.

=>slides: Pg. 10

Okay, so once we have this, how do we execute this code? So from the command line, I'd run java, telling it what the class path is, and the class path in this case is dot, so the current directory in which my class files are, and that lets me pick up the com.app.application class. And then also on the class path I have the jar file containing the mysql driver, and at the time of recording this class, that driver is in mysql‑connector‑java‑8.0.23.jar. If we don't specify the JAR file on the class path, then the DriverManager is not able to load the driver that we specify in the URL.

=>slides: Pg. 11

And we saw, again, in the previous module that when we run this, it will print out the driver‑specific class name, so in this case, com.mysql.cj.jdbc.ConnectionImpl. Okay, so let's see how we run this.

# Closing Connections

=>slides: Pg. 12

So as we said in the previous module, but just to emphasize here, database connections are native resources, and this means they must be closed by the application after use. It used to be that to do this, you'd use a try‑finally block and always close the connection in the finally. But this was slightly awkward as the close method itself could also throw an exception. So you ended up wrapping the code in the finally block in a try‑catch block, and the code looked very messy. Luckily, we now have try‑with‑resources. So we can close our connections using the try‑with‑resources syntax. Now these closes cascade if you like. So within a connection, if I create a statement, within that statement, I get back a ResultSet. Closing the connection will also close the statement, and closing this statement will also close the ResultSet. So while you could close all of these explicitly, closing the connection will close things implicitly for you.

=>slides: Pg. 13

Our code looks something like this. We wrap the DriveManager.getConnection inside a try. And then inside the body of the try, we can call PreparedStatement executeQuery, iterate over the ResultSet that we get back from their query. The try block closes. When it closes, that closes the connection. And closing the connection closes the PreparedStatement. And closing the statement closes the ResultSet. But you have to do this. You have to make sure that we close ResultSets, statements, and connections when we are finished with them. So let's just quickly take a look at the code and see how we do this.

# Demonstration - Closing Connections

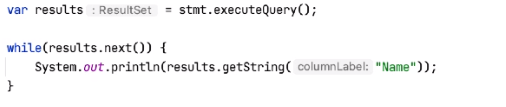
=>slides: Pg. 14



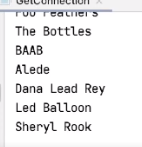
So this is the code where we left it. We have the DriverManager.getConnection wrapped inside the try‑with‑resources. So what I'm going to show you very briefly is how to use the connection to get a PreparedStatement and that PreparedStatement to get some results. And we'll do this two ways. So we said as well as closing the Connection, we should also close the PreparedStatement and the ResultSet. And we can also do that using try‑with‑resources.



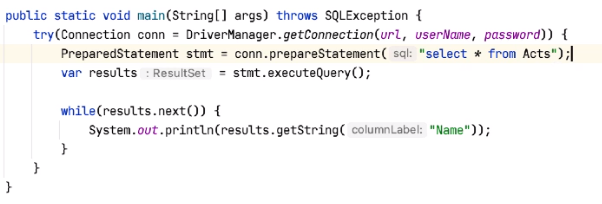
So inside here, I can say something like this, so PreparedStatement = conn.prepareStatement, and I can pass this the sql for the statement. We're going to select all the data from the Acts table. So we can say select \* from Acts. So now, as well as having the connection inside try‑with‑resources, we have the statement inside try‑with‑resources. So I can use that statement.



I can say statement. I can say var results = stat.executeQuery, and that will give me back a ResultSet. Let me remove this System.out.println here and then do a while results.next to iterate over the ResultSet. Then once I iterate over this, I can print out all of the details from the results that we get back. In this case, I'll just print out the name of the act. Don't worry too much now about how PreparedStatement works or how the ResultSet works. We'll cover those in some details as we go through the rest of this course.



If I run this now, it will print out the name of the act and the database, so the Foo Feathers, The Bottles, Alede, Dana Lead Rey, and so on and so forth. And this code currently is using try‑with‑resources. Try‑with‑resources will close the statement, which will cause the ResultSet to close and will close the connection.

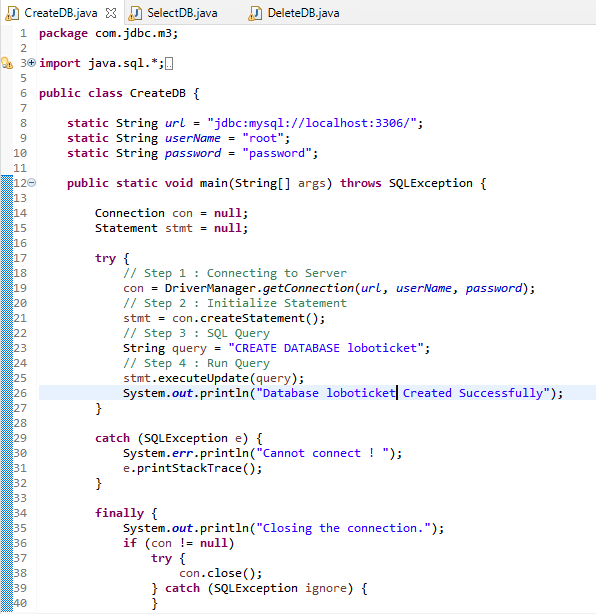


I can also move this PreparedStatement call and put that inline inside the try‑with‑resources block. Run the code again, and it works the same way. We get the same output, but now I'm relying on the connection being closed. Then when the connection is closed, it will close the PreparedStatement. When the PreparedStatement closes, it will close the ResultSet. Notice as well that as I'm using Java 11 here, we can use var here for the variable type. So we can either set the variable types explicitly, as I've done with the PreparedStatement, or we can use var to set the type implicitly within the code.

# Demonstration: JDBC – Create, Select And Delete MySQL Database

=>slides: Pg. 15

### **CREATING DATABASE**



There are 4 steps to execute any query against database.

**Step 1 :** ****Connecting to Server****. You can connect to server using following line of code.

con = DriverManager.getConnection(dburl, dbuser, dbpass);

**Step 2 :** : ****Initialize Statement****. Statement class is used for carrying your query to server and execute query.

stmt=con.createStatement();

**Step 3 :** : ****SQL Query****. This is your sql query which is used to create database.

String query="CREATE DATABASE STOREDB";

**Step 4 :** : ****Run Query****. Finally execute your query using statement object.

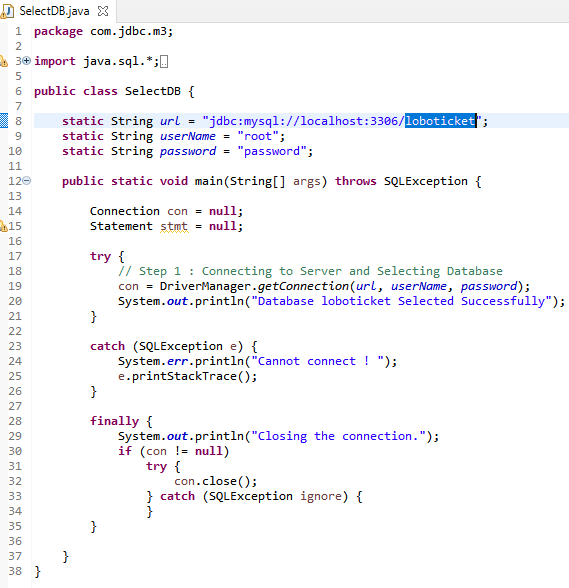
stmt.executeUpdate(query);

### **SELECT DATABASE**

It is necessary to select right database before creating table or connecting your program. There may be dozens of databases resides in server so you need to select your correct database before executing query. You have learned and created STOREDB just now. It’s time to learn how to select your STOREDB database using JDBC.

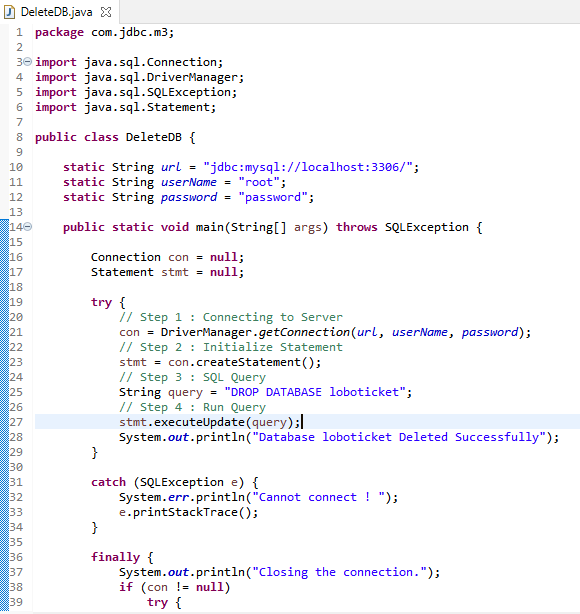
Selecting Database is very easy process. Just write your database name in connecting url and that’s it.

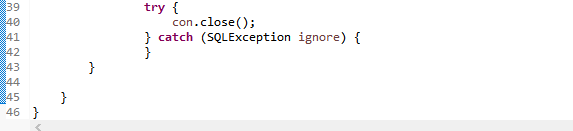
*static final String dburl = “jdbc:mysql://localhost/*loboticket*“;*



### **DELETING DATABASE**

Till now, you have learned how to create and select database using JDBC. Now, final task is deleting database.





# Summary

=>slides: Pg. 16

So what have we seen? We've seen that the JDBC URL has a specific format. So it starts with jdbc, the protocol, then it has a subprotocol, which is going to be vendor specific, and then it has the connection part of the protocol that allows the driver to connect to the database. We use the URL to load the driver, and then the driver connects to the database. And then remember that we need to use something like try‑with‑resources to make sure that we close the connection. We can also use try‑with‑resources to close the PreparedStatement and the ResultSet. But remember that closing the connection will close the PreparedStatement, and closing the PreparedStatement will close the ResultSet.

=>slides: Pg. 16

Okay, so now that we've seen how to connect to a database, we're next going to take a look at how we use Prepared statements in more detail.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*