3. Introduction to JDBC

Contents

[1. Introduction 1](#_Toc5413)

[2. Demonstration - Loading the Driver 3](#_Toc18588)

[3. JDBC Interfaces 5](#_Toc2977)

[4. Connections and it's types 5](#_Toc23105)

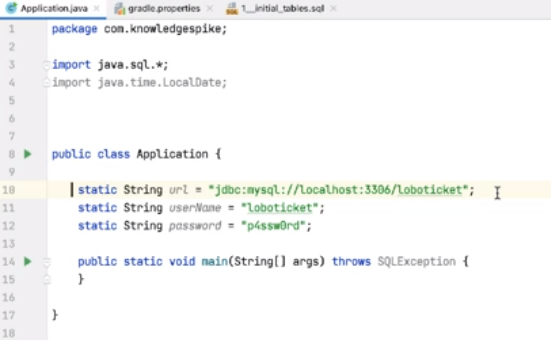
[5. ResultSet and it's types 8](#_Toc22405)

[6. Summary 9](#_Toc21435)

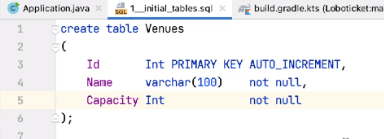
# 1. Introduction

=>slides: Pg. 1-2

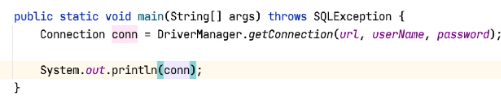
So let's just do a quick demo and show how this works.



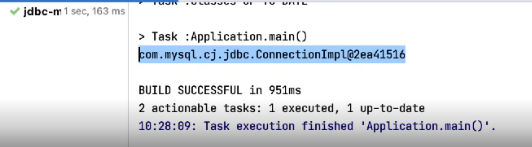
So before I get into the details of using JDBC fully, I just wanted to show you the setup of the application that we'll be using. So all of this will be provided as the code for this course. So we're using STS as the IDE. But also inside here, I've set this application up using Gradle. So we have a Gradle file. If you're not used to Gradle, don't worry too much. There's just some tasks in here you might be interested in. So in Gradle, I have a plugin for something called Flyway. So Flyway is a tool for managing databases, and Flyway uses something called migrations. And the reason I'm telling you this is that in this code, I have a single migration. And this migration contains the code that's needed to create the tables in the database that we are using.



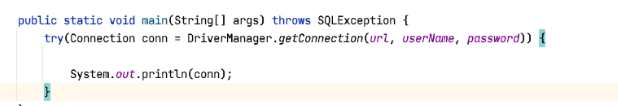
So there are three tables, Venues, Acts, and Gigs, and the database we're using within MySQL is something called loboticket. So by running the flyway, flywayMigrate task, that will allow me to connect MySQL and create the tables that I need for my code. If you don't want to use Flyway, simply grab the SQL and use that to create the tables. So these are the tables that we're using. This is slightly different If you watched the first module from the tables shown in the diagram there. So we have a Venues table, and each venue has a name and a capacity. We have an Acts table, and each act has a name and an optional record label that the act is assigned to, and we have a Gigs table. The Gigs table is related to the Acts and Venues table by the VenueId and the ActId. The Gigs table tells us how many tickets have been sold for the gig, the price of those tickets, and the date that gig will run on. So all the code that we write will execute against this database. So the first thing I want to do very simply is to show you what a connection looks like.



So as we saw on the slide, I can get the connection by calling DriverManager.getConnection. Now there are a couple of overloads to this. I can just pass this the JDBC URL that will connect to that database, but the database I'm using requires a username and password. So I'm going to use the version here where I pass the URL, the username, and password. I'm not going to worry too much about what those values are here, just to point out that they're all defined as static variables in this code. So to getConnection, I can pass url, userName, and password. Once I have that connection, I can print that out just by doing a system.out.println. So from inside STS, I'm going to build this code.



So from within here, if I run this code, we'll see here it prints out the fully qualified class name. So even though we are coding against the interfaces, this prints out com.mysql.cj.jdbc.ConnectionImpl. So this is the class that implements this connection interface for MySQL. So one thing I want to emphasize here, and I'll do this throughout the course, is that anything we get back from JDBC must be closed when we're finished with them. Otherwise, we're in danger of leaking resources. Now it used to be we would do that by using a try‑catch‑finally, but now we can use a try‑with‑resources.



So I can wrap the call to getConnection inside try and then the usage of that connection inside the body of the try. And if I run this, the output is going to be the same. But now when we hit this closing brace, we're going to explicitly close this connection. And we'll see as we go through this, we'll be closing connections, prepared statements, callable statements, and result sets. And we need to do that to make sure we don't leak anything while using JDBC.

# Demonstration - Loading the Driver

=>slides: Pg. 3

So in this module, as the title says, we'll introduce you to JDBC. We'll take a look at a number of things that make up JDBC. We'll take a look at the JDBC interfaces. We look at something called the DriverManager. We'll take a look at something called the JDBC driver. We'll understand what a JDBC URL is, and I will use that to connect to a database. And then we'll see how we can make that initial connection to the database.

=>slides: Pg. 4

So when you write an application using JDBC, you need to connect to the database to get the data. And to connect to the database, we use something called a JDBC driver. Now these drivers are Java classes, either supplied by the database vendor or maybe through open source. And the drivers are specific to a given database. So there'll be a MySQL driver or an Oracle driver or a SQL Server driver, for example. However, if we were to use the driver, we don't use that class explicitly. We would use an interface. And in this case, the interface we would use would be the driver interface. And by using these interfaces, we hide away the details of what the driver is doing under the covers. Similarly, we use that driver to make a connection to the database. But again, we wouldn't use the driver‑specific classes to manage this connection. We'd use the JDBC Connection interface.

The statement interface is **used to create SQL basic statements in Java. I**t provides methods to execute queries with the database. The Statement interface **provides methods to execute queries with the database**. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.The **Statement interface** provides the method to execute the database queries. After making a connection, Java application can interact with database.

To execute CRUD statements against the database, we'd use another interface called PreparedStatement. So PreparedStatement, as we'll see a little later, lets us create what are known as parameterized statements. And the text that goes into a PreparedStatement looks like SQL. So into a PreparedStatement, we can put insert statements, update statements, delete statements, or select statements. And if the PreparedStatement succeeds, it will do one of two things. It will either tell us how many rows in the database have changed, so we do an insert, an update, or a delete, how many rows have we affected. Or, if we use the select, it will return the data we've asked for, if any, in tabular format. Now it may be that our database has stored procedures. And if we want to call those stored procedures, we'd do something called a CallableStatement. And again, we'll see the syntax for this later in the course. But essentially, we use this to execute any stored procedures in the database, passing in parameters to that stored procedure and getting data back from that stored procedure. Like the PreparedStatement, a CallableStatement could return this data in a tabular format. And if either of these do that, we'll get that data back in something known as a ResultSet. Now again, just to emphasize, all of the things we're seeing here are interfaces defined as part of JDBC. So there's a Driver interface, a Connection interface, a PreparedStatement interface, a CallableStatement interface, and a ResultSet interface. And all of these interfaces hide away the details of the underlying JDBC driver that we're using. So, for example, if we're connected to the MySQL database, we'd use the MySQL driver, but we wouldn't see any of the classes inside that driver, only these interfaces.

=>slides: Pg. 5

So how do we use this? So we'll see in a moment how we get a connection. But once we have that connection, the connection lets us create a PreparedStatement. Now I'm not showing the details of the PreparedStatement here. We'll see that in the next few chapters. But we can see here that I'm using the PreparedStatement to do a SELECT. So I'm going to get some data from the database. Notice we are coding against the interface, so I'll be coding against the Connection interface, calling its PreparedStatement method, and that returns me a reference of type PreparedStatement. PreparedStatement is one of the JDBC interfaces. Once I have that PreparedStatements, I can call, for example, its executeQuery method, and that lets me execute select queries, and this returns me a ResultSet. And this ResultSet is the tabular data. But again, notice the ResultSet is the interface. Again, we're coding against the interface. To get the data from the ResultSet, it has a next method. So the ResultSet is cursor‑based, and again, we'll go into more detail on this a little later. So to move through this ResultSet, we call ResultSet.next. Now as we move in, we'll see later, we can access this data. And again, just to emphasize, all of this is coded against the interfaces and not against the implementations of the underlying driver.

=>slides: Pg. 6

So how do we get this connection? So to get the connection, we use something called the DriverManager. And again, we'll go into more detail on this in the next module. So the DriverManager has a method called getConnection, and to this we pass a string. And this string is known as the JDBC URL. And basically, this identifies which database we'd like to connect to. And again, we'll go into more information on this in later modules. Now again, notice here, getConnection returns us something of type Connection, and that connection is an interface type, again the interface defined inside JDBC. =>slides: Pg. 7

However, if we were to print out that connection, we'd see the underlying implementation class. So if I was to do a system.outprintln on that connection variable, we'd see something like this. In this case, again, I'm using the MySQL JDBC driver. So what we get printed out is the driver‑specific class name.

# JDBC Interfaces

=>slides: Pg. 8

So now that we've seen how to load the driver, and we will go into more detail on this in the next module, let's just see how all this ties together. So we use a DriverManager to load the driver. That DriverManager provides implementations of the JDBC interfaces. And remember, we always code against the interfaces. These implementations, these classes, know how to talk to the specific database. So in the example that we just saw, we're using the MySQL driver and therefore the MySQL implementations so it knows how to talk to the MySQL database. As I've said, you could be using Oracle or SQL Server or any other database out there that has a suitable JDBC driver.

=>slides: Pg. 9

We've seen that to load the driver, we call it DriverManager.getConnection, and the example here shows the single‑parameter version to which we pass the JDBC URL. And again, we'll go into more detail on this URL in a later module.

=>slides: Pg. 10

This loads the driver code, and this driver code contains the JDBC classes. Now for example, we saw this ConnectionImpl class, and that ConnectionImpl class implements the JDBC Connection interface. So we have something like this.

=>slides: Pg. 11

We have a set of JDBC interfaces, and we have a set of JDBC implementations loaded by the driver. So our driver interface will have some DriverImpl class. The connection interface will have a ConnectionImpl class. Statement, StatementImpl class. PreparedStatement, PreparedStatementImpl class. CallableStatement and ResultSet similarly will have a CallableStatementImpl and ResultSetImpl. Now these implementation classes won't just be called DriverImpl or ConnectionImpl. They'll be fully qualified class names. And it doesn't matter what the implementation names are. They will be decided by whoever has written the driver. Again, the important thing is that we code against the interfaces, so Driver, Connection, PreparedStatement, CallableStatement, and ResultSet.

=>slides: Pg. 12

Okay, so what have we seen in this module? We've seen that we code against these five specific interfaces. We've seen how we use DriverManager to load the driver. And then we've talked about how this loaded driver code has classes that implement these interfaces that we are going to use.

# Connections and it's types

=>slides: Pg. 13

The JDBC classes are contained in the Java Package **java.sql** and **javax.sql**.  
JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database.
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver

**Type-1 driver**

Type-1 driver or JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. Type-1 driver is also called Universal driver because it can be used to connect to any of the databases.

* As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured.
* The ODBC bridge driver is needed to be installed in individual client machines.
* Type-1 driver isn’t written in java, that’s why it isn’t a portable driver.
* This driver software is built-in with JDK so no need to install separately.
* It is a database independent driver.

**Type-2 driver**

The Native API driver uses the client -side libraries of the database. This driver converts JDBC method calls into native calls of the database API. In order to interact with different database, this driver needs their local API, that’s why data transfer is much more secure as compared to type-1 driver.

* Driver needs to be installed separately in individual client machines
* The Vendor client library needs to be installed on client machine.
* Type-2 driver isn’t written in java, that’s why it isn’t a portable driver
* It is a database dependent driver.

**Type-3 driver**

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation.

* Type-3 drivers are fully written in Java, hence they are portable drivers.
* No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.
* Network support is required on client machine.
* Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.
* Switch facility to switch over from one database to another database.

**Type-4 driver**

Type-4 driver is also called native protocol driver. This driver interact directly with database. It does not require any native database library, that is why it is also known as Thin Driver.

* Does not require any native library and Middleware server, so no client-side or server-side installation.
* It is fully written in Java language, hence they are portable drivers.

**Which Driver to use When?**

* If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is type-4.
* If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
* Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
* The type 1 driver is not considered a deployment-level
* driver, and is typically used for development and testing purposes only.

# ResultSet and it's types

=>slides: Pg. 14

This slide shows the position of ResultSet in the JDBC Framework. ****ResultSet**** can be obtained by executing SQL Query using ****Statement****, ****PreparedStatement**** or ****CallableStatement****.

****AutoCloseable****, ****Wrapper**** are super interfaces of ResultSet.

=>slides: Pg. 14

There are two types of result sets namely, forward only and, bidirectional.

****Forward only ResultSet:**** The ResultSet object whose cursor moves only in one direction is known as forward only ResultSet. By default, JDBC result sets are forward-only result sets.

****Bidirectional ResultSet:**** A bi-directional ResultSet object is the one whose cursor moves in both forward and backward directions.

# Summary

=>slides: Pg. 15

I've also talked about the fact that all the code we're going to use is in STS. There's a Gradle script that allows us to manage the code. And specifically, there's code there to set up a database. The Gradle script uses Flyway to do this, but Feel free just to take that SQL that's there and use that to create the databases any way that you want to if you want to follow along with these demonstrations.

=>slides: Pg. 14

We've seen that we're going to use a ticket database, and this has gigs, acts, and venues. We've seen how these are related using keys. The database also has stored procedures. I haven't showed you these yet. When we come to look at CallableStatement in module 7, we'll see those stored procedures then.

=>slides: Pg. 15

Okay, so what's next? So in the next module, we're going to go into more detail into how we connect to a database.

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