UNIT 2:JAVA DATABASE CONNECTIVITY

JDBC (JAVA DATABASE CONNECTIVITY)

JDBC is a Java database connectivity technology (Java Standard Edition platform) from Oracle Corporation.

This technology is an API for the Java programming language that defines how a client may access a database.

It provides methods for querying and updating data in a database. JDBC is oriented towards relational databases.

A JDBC-to-ODBC bridge enables connections to any ODBC-accessible data source in the JVM host environment.

Sun Microsystems released JDBC as part of JDK 1.1 on February 19, 1997.Since then it has been part of the Java Standard Edition.

The JDBC classes are contained in the Java package java.sql & javax.sql.

JAVA as a DATABASE FRONT END

For developing an application, we need three important things, they are frontend, backend and middleware.

Backends are the database management systems which organize & manage the data.

Middleware is software which connects this backend with the frontend.

Frontend is used to design the user interface i.e. screens, reports, menus etc. The data can be accessed through front ends by the user.

Java is a well known front end as it allows to create application interfaces & other required components. Java has a huge collection of classes and interfaces with readymade methods. These all are used to design the front ends.

Java allows user to create frontend not only in the form of simple applications but in the form of applets. Applets are used in online applications as an interface which is featured by GUI.

It strongly supports internet as it is popularly known as internet language. It has drivers like JDBC, JDBC-ODBC bridge etc which supports connection with various backends from different vendors.

Advantages:

- It is an open source, so users do not have to struggle with heavy license fees each year.
- Platform independent.
- Java API's can easily be accessed by developers.
- Java perform supports garbage collection, so memory management is automatic.
- Java always allocates objects on the stack.
- Java embraced the concept of exception specifications.
- Multi-platform support language and support for web-services.
- Using JAVA we can develop dynamic web applications.

- It allows you to create modular programs and reusable codes.

DATABASE CLIENT-SERVER METHODOLOGY

The JDBC API is an application interface of java for connecting java as a front end &DBMS or RDBMS as a backend. The JDBC is a java database connectivity tool.

In short the JDBC API allows programmatic access to relational data from the java programming language.

With JDBC API user can execute SQL statements, retrieve results, & ensure the updations back to data source.

The speciality of JDBC API is that it can interact with multiple data sources in a distributed & heterogeneous environment.

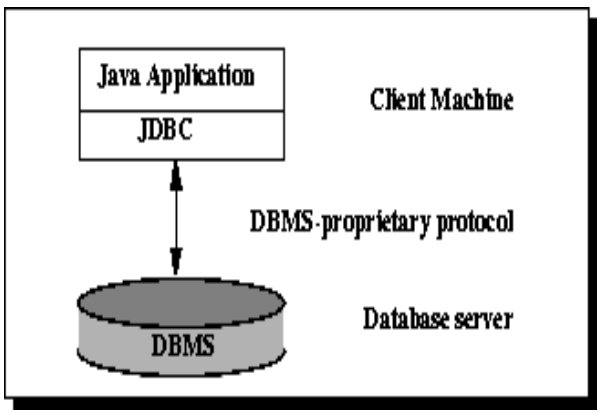
It supports two-tier & three-tier processing models or architecture for accessing database.

TWO-TIER DATABASE DESIGN

In the two-tier model, a Java application talks directly to the data source.

This requires a JDBC driver that can communicate with the particular data source being accessed.

A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user.



The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server.

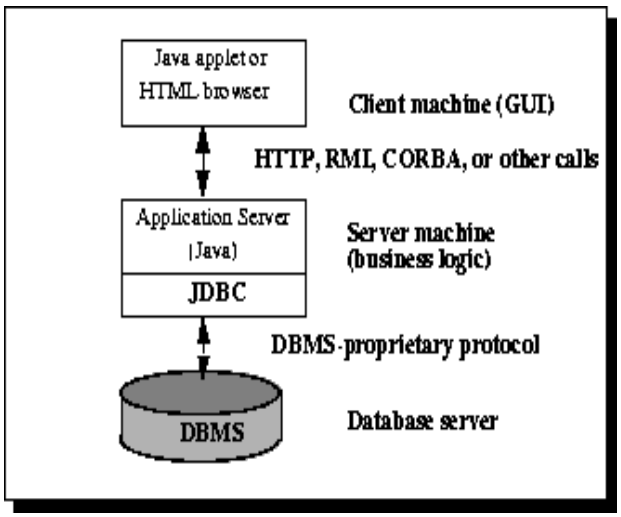
The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

THREE-TIER DATABASE DESIGN

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source.

The data source processes the commands and sends the results back to the middle tier, which then sends them to the user.

MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain



control over access and the kinds of updates that can be made to corporate data.

Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans™,

The Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture.

Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

THE JDBC API(JDBC API Components)

- **DriverManager:**
  This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication subprotocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.

- **Driver:**
  This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.

- **Connection :**
  This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.

- **Statement :**
  You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

- **ResultSet:**
  These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

- **SQLException:**
  This class handles any errors that occur in a database application.

LIMITATIONS Using JDBC (APPLICATIONS Vs APPLETS)

We can compare java applications & applets on the basis of JDBC mainly. The connection of java application is easier than those of the applets.

Following are the major limitations presented in the form of differences between applications & applet.

APPLICATIONS	APPLETS
An application runs stand-alone, with the support of a virtual machine	An applet runs under the control of a browser
An application can have free reign over these resources.	Applet is subjected to more stringent security restrictions in terms of file and network access
Need JDK, JRE, JVM installed on client machine.	Applet is portable and can be executed by any JAVA supported browser.
The java applications i.e. java code is trusted.	Applets can be trusted or untrusted to access the data
Writing program with java applications for database connectivity are simple.	Writing programs with java applets is little complicated than applications.

SECURITY CONSIDERATIONS

JDBC and untrusted applets

JDBC should follow the standard applet security model. Specifically:

- JDBC should assume that normal unsigned applets are untrustworthy
- JDBC should not allow untrusted applets access to local database data
- If a downloaded JDBC Driver registers itself with the JDBC DriverManager, then JDBC should only use that driver to satisfy connection requests from code which has been loaded from the same source as the driver.
- An untrusted applet will normally only be allowed to open a database connection back to the server from which it was downloaded
- JDBC should avoid making any automatic or implicit use of local credentials when making connections to remote database servers.

JDBC and Java applications

For a normal Java application (i.e. all Java code other than untrusted applets) JDBC should happily load drivers from the local classpath and allow the application free access to files, remote servers, etc.

However as with applets, if for some reason an untrusted sun.sql.Driver class is loaded from a remote source, then that Driver should only be used with code loaded from that same source.

Network security

The security of database requests and data transmission on the network, especially in the Internet case, is also an important consideration for the JDBC user. However, keep in mind that we are defining programming interfaces in this specification, not a network protocol. The network protocols used for database access have generally already been fixed by the DBMS vendor or connectivity vendor. JDBC users should verify that the network protocol provides adequate security for their needs before using a JDBC driver and DBMS server.

Security Responsibilities of Drivers

Because JDBC drivers may be used in a variety of different situations, it is important that driver writers follow certain simple security rules to prevent applets from making illegal database connections.

These rules are unnecessary if a driver is downloaded as an applet, because the standard security manager will prevent an applet driver from making illegal connections. However JDBC driver writers should bear in mind that if their driver is "successful" then users may start installing it on their local disks, in which case it becomes a trusted part of the Java environment, and must make sure it is not abused by visiting applets. We therefore urge all JDB driver writers to follow the basic security rules.

JDBC DRIVERS

A JDBC driver is a software component enabling a Java application to interact with a database.

To connect with individual databases, JDBC (the Java Database Connectivity API) requires drivers for each database. The JDBC driver gives out the connection to the database and implements the protocol for transferring the query and result between client and database.

JDBC technology drivers fit into one of four categories.

- JDBC-ODBC bridge
- Native-API Driver
- Network-Protocol Driver(MiddleWare Driver)
- Database-Protocol Driver(Pure Java Driver)

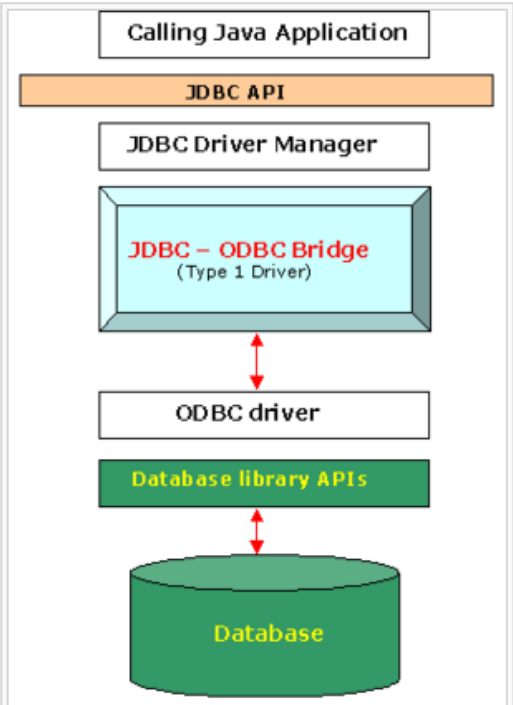
Type 1 Driver - JDBC-ODBC bridge

The JDBC type 1 driver, also known as the JDBC-ODBC bridge, is a database driver implementation that employs the ODBC driver to connect to the database.

The driver converts JDBC method calls into ODBC function calls.

The driver is platform-dependent as it makes use of ODBC which in turn depends on native libraries of the underlying operating system the JVM is running upon.

Sun provides a JDBC-ODBC Bridge driver: sun.jdbc.odbc.JdbcOdbcDriver.



Schematic of the JDBC-ODBC bridge

Advantages

- Almost any database for which an ODBC driver is installed can be accessed, and data can be retrieved.

Disadvantages

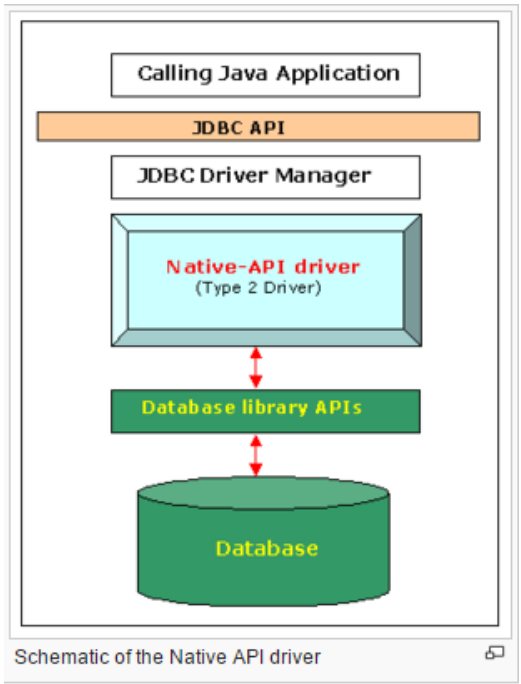
- Performance overhead since the calls have to go through the jdbc Overhead bridge to the ODBC driver, then to the native db connectivity interface (thus may be slower than other types of drivers).
- The ODBC driver needs to be installed on the client machine.
- Not suitable for applets, because the ODBC driver needs to be installed on the client.

Type 2 Driver - Native-API Driver

The JDBC type 2 driver, also known as the Native-API driver, is a database driver implementation that uses the client-side libraries of the database.

The driver converts JDBC method calls into native calls of the database API.

For example: Oracle OCI driver



Schematic of the Native API driver

Advantages

- As there is no implementation of jdbc-odbc bridge, its considerably faster than a type 1 driver.

Disadvantages

- The vendor client library needs to be installed on the client machine.
- Not all databases have a client side library
- This driver is platform dependent
- This driver supports all java applications except Applets

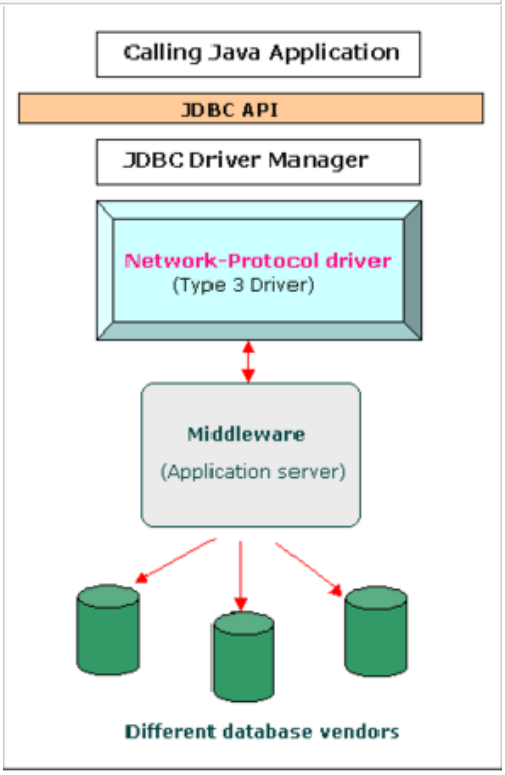
Type 3 Driver - Network-Protocol Driver(MiddleWare Driver)

The JDBC type 3 driver, also known as the Pure Java Driver for Database Middleware, is a database driver implementation which makes use of a middle tier between the calling program and the database.

The middle-tier (application server) converts JDBC calls directly or indirectly into the vendor-specific database protocol.

The type 3 driver is platform-independent as the platform-related differences are taken care of by the middleware.

Exa: IDA Server



Schematic of the Network Protocol driver

Advantages

- Since the communication between client and the middleware server is database independent, there is no need for the database vendor library on the client. The client need not be changed for a new database.
- The middleware server (which can be a full fledged J2EE Application server) can provide typical middleware services like caching (of connections, query results, etc.), load balancing, logging, and auditing.
- A single driver can handle any database, provided the middleware supports it.

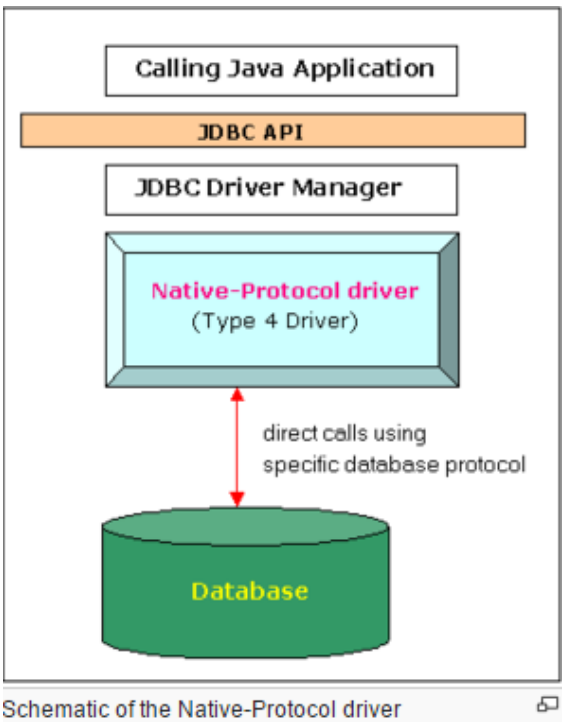
Disadvantages

- Requires database-specific coding to be done in the middle tier.
- The middleware layer added may result in additional latency, but is typically overcome by using better middleware services.

Type 4 Driver - Database-Protocol Driver(Pure Java Driver)

The JDBC type 4 driver, also known as the Direct to Database Pure Java Driver, is a database driver implementation that converts JDBC calls directly into a vendor-specific database protocol.

Written completely in Java, type 4 drivers are thus platform independent.



Schematic of the Native-Protocol driver

Advantages

- Completely implemented in Java to achieve platform independence.
- These drivers don't translate the requests into an intermediary format (such as ODBC).
- The client application connects directly to the database server. No translation or middleware layers are used, improving performance.
- The JVM can manage all aspects of the application-to-database connection; this can facilitate debugging.

Disadvantages

- Drivers are database dependent, as different database vendors use widely different (and usually proprietary) network protocols.

CORBA (COMMON OBJECT REQUEST BROKER ARCHITECTURE)

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) designed to facilitate the communication of systems that are deployed on diverse platforms.

CORBA enables collaboration between systems on different operating systems, programming languages, and computing hardware. CORBA has many of the same design goals as object-oriented programming: encapsulation and reuse.

CORBA enables software written in different languages and running on different computers to work with each other seamlessly.

CORBA normalizes the method-call semantics between application objects residing either in the same address-space (application) or in remote address-spaces (same host, or remote host on a network).

CORBA uses an interface definition language (IDL) to specify the interfaces that objects present to the outer world. CORBA then specifies a mapping from IDL to a specific implementation language.

The CORBA specification dictates there shall be an ORB through which an application would interact with other objects. This is how it is implemented in practice:

- The application simply initializes the ORB, and accesses an internal Object Adapter,which maintains things like reference counting, object (and reference) instantiation policies, and object lifetime policies.
- The Object Adapter is used to register instances of the generated code classes. Generated code classes are the result of compiling the user IDL code, which translates the high-level interface definition into an OS- and language-specific class base for use by the user application. This step is necessary in order to enforce CORBA semantics and provide a clean user process for interfacing with the CORBA infrastructure.

A language mapping requires the developer to create IDL code that represents the interfaces to his objects.

Typically, a CORBA implementation comes with a tool called an IDL compiler which converts the user's IDL code into some language-specific generated code.

A traditional compiler then compiles the generated code to create the linkable-object files for the application.

CORBA's benefits include-

language- and OS-independence, freedom from technology-linked implementations, strong data-typing, high level of tunability, and freedom from the details of distributed data transfers.

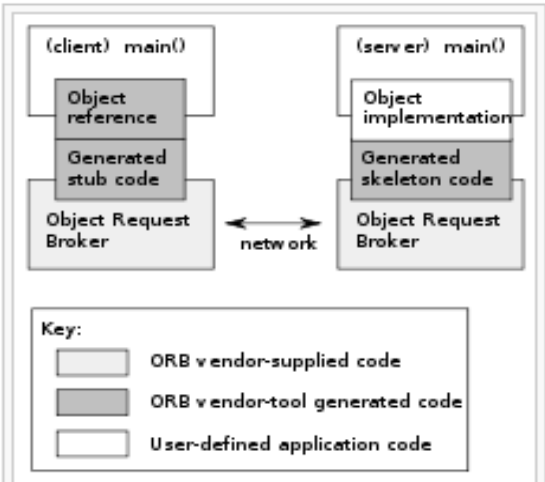


Illustration of the autogeneration of the infrastructure code from an interface defined using the CORBA IDL