

This session will teach you how to access a database using Ruby. The *Ruby DBI* module provides a database-independent interface for Ruby scripts similar to that of the Perl DBI module.

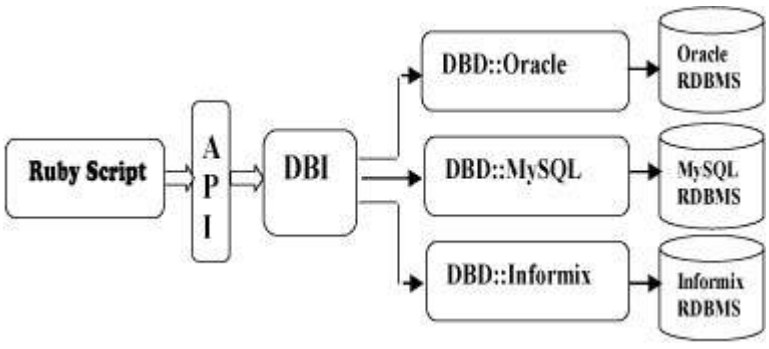
DBI stands for Database independent interface for Ruby which means DBI provides an abstraction layer between the Ruby code and the underlying database, allowing you to switch database implementations really easily. It defines a set of methods, variables, and conventions that provide a consistent database interface, independent of the actual database being used.

DBI can interface with the following:

- ADO *ActiveXDataObjects*
- DB2
- Frontbase
- mSQL
- MySQL
- ODBC
- Oracle
- OCI8 *Oracle*
- PostgreSQL
- Proxy/Server
- SQLite
- SQLRelay

Architecture of a DBI Application

DBI is independent of any database available in backend. You can use DBI whether you are working with Oracle, MySQL or Informix, etc. This is clear from the following architure diagram.



The general architecture for Ruby DBI uses two layers:

- The database interface *DBI* layer. This layer is database independent and provides a set of common access methods that are used the same way regardless of the type of database server with which you're communicating.
- The database driver *DBD* layer. This layer is database dependent; different drivers provide access to different database engines. There is one driver for MySQL, another for PostgreSQL, another for InterBase, another for Oracle, and so forth. Each driver interprets requests from the DBI layer and maps them onto requests appropriate for a given type of database server.

Prerequisites:

If you want to write Ruby scripts to access MySQL databases, you'll need to have the Ruby MySQL module installed.

This module acts as a DBD as explained above and can be downloaded from http://www.tmtm.org/en/mysql/ruby/

Obtaining and Installing Ruby/DBI:

You can download and install the Ruby DBI module from the following location:

http://rubyforge.org/projects/ruby-dbi/

Before starting this installation make sure you have root privilege. Now, follow the following steps:

Step 1

```
$ tar zxf dbi-0.2.0.tar.gz
```

Step 2

Go in distribution directory *dbi-0.2.0* and configure it using the *setup.rb* script in that directory. The most general configuration command looks like this, with no arguments following the config argument. This command configures the distribution to install all drivers by default.

```
$ ruby setup.rb config
```

To be more specific, provide a --with option that lists the particular parts of the distribution you want to use. For example, to configure only the main DBI module and the MySQL DBD-level driver, issue the following command:

```
$ ruby setup.rb config --with=dbi,dbd_mysql
```

Step 3

Final step is to build the driver and install it using the following commands:

```
$ ruby setup.rb setup
$ ruby setup.rb install
```

Database Connection:

Assuming we are going to work with MySQL database, before connecting to a database make sure of the following:

- You have created a database TESTDB.
- You have created EMPLOYEE in TESTDB.
- This table is having fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB
- Ruby Module DBI is installed properly on your machine.
- You have gone through MySQL tutorial to understand MySQL Basics.

Following is the example of connecting with MySQL database "TESTDB"

```
#!/usr/bin/ruby -w
```

```
require "dbi"

begin
     # connect to the MySQL server
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     # get server version string and display it
     row = dbh.select_one("SELECT VERSION()")
     puts "Server version: " + row[0]
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

While running this script, it's producing the following result at my Linux machine.

```
Server version: 5.0.45
```

If a connection is established with the data source, then a Database Handle is returned and saved into **dbh** for further use otherwise **dbh** is set to nil value and *e.err* and *e::errstr* return error code and an error string respectively.

Finally, before coming out it, ensure that database connection is closed and resources are released.

INSERT Operation:

INSERT operation is required when you want to create your records into a database table.

Once a database connection is established, we are ready to create tables or records into the database tables using **do** method or **prepare** and **execute** method.

Using do Statement:

Statements that do not return rows can be issued by invoking the **do** database handle method. This method takes a statement string argument and returns a count of the number of rows affected by the statement.

```
dbh.do("DROP TABLE IF EXISTS EMPLOYEE")
dbh.do("CREATE TABLE EMPLOYEE (
     FIRST_NAME  CHAR(20) NOT NULL,
     LAST_NAME  CHAR(20),
     AGE INT,
     SEX CHAR(1),
     INCOME FLOAT )" );
```

Similar way you can execute SQL *INSERT* statement to create a record into EMPLOYEE table.

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # connect to the MySQL server
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     dbh.do( "INSERT INTO EMPLOYEE(FIRST_NAME,
                  LAST_NAME,
                  AGE,
     SEX,
     INCOME)
```

```
            VALUES ('Mac', 'Mohan', 20, 'M', 2000)" )
      puts "Record has been created"
      dbh.commit
rescue DBI::DatabaseError => e
      puts "An error occurred"
      puts "Error code:    #{e.err}"
      puts "Error message: #{e.errstr}"
      dbh.rollback
ensure
      # disconnect from server
      dbh.disconnect if dbh
end
```

Using *prepare* and *execute*:

You can use *prepare* and *execute* methods of DBI class to execute SQL statement through Ruby code.

Record creation takes following steps:

- Preparing SQL statement with INSERT statement. This will be done using **prepare** method.
- Executing SQL query to select all the results from the database. This will be done using **execute** method.
- Releasing Statement handle. This will be done using **finish** API
- If everything goes fine, then **commit** this operation otherwise you can **rollback** complete transaction.

Following is the syntax to use these two methods:

```
sth = dbh.prepare(statement)
sth.execute
   ... zero or more SQL operations ...
sth.finish
```

These two methods can be used to pass **bind** values to SQL statements. There may be a case when values to be entered is not given in advance. In such case, binding values are used. A question mark (**?**) is used in place of actual value and then actual values are passed through execute API.

Following is the example to create two records in EMPLOYEE table:

```
#!/usr/bin/ruby -w

require "dbi"

begin
      # connect to the MySQL server
      dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                        "testuser", "test123")
      sth = dbh.prepare( "INSERT INTO EMPLOYEE(FIRST_NAME,
                  LAST_NAME,
                  AGE,
       SEX,
      INCOME)
                  VALUES (?, ?, ?, ?, ?)" )
      sth.execute('John', 'Poul', 25, 'M', 2300)
      sth.execute('Zara', 'Ali', 17, 'F', 1000)
      sth.finish
      dbh.commit
      puts "Record has been created"
rescue DBI::DatabaseError => e
      puts "An error occurred"
      puts "Error code:    #{e.err}"
      puts "Error message: #{e.errstr}"
```

```
     dbh.rollback
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

If there are multiple INSERTs at a time, then preparing a statement first and then executing it multiple times within a loop is more efficient than invoking do each time through the loop

READ Operation:

READ Operation on any database means to fetch some useful information from the database.

Once our database connection is established, we are ready to make a query into this database. We can use either **do** method or **prepare** and **execute** methods to fetch values from a database table.

Record fetching takes following steps:

- Preparing SQL query based on required conditions. This will be done using **prepare** method.
- Executing SQL query to select all the results from the database. This will be done using **execute** method.
- Fetching all the results one by one and printing those results. This will be done using **fetch** method.
- Releasing Statement handle. This will be done using **finish** method.

Following is the procedure to query all the records from EMPLOYEE table having salary more than 1000.

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # connect to the MySQL server
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     sth = dbh.prepare("SELECT * FROM EMPLOYEE
                          WHERE INCOME > ?")
     sth.execute(1000)

     sth.fetch do |row|
        printf "First Name: %s, Last Name : %s\n", row[0], row[1]
        printf "Age: %d, Sex : %s\n", row[2], row[3]
        printf "Salary :%d \n\n", row[4]
     end
     sth.finish
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

This will produce the following result:

```
First Name: Mac, Last Name : Mohan
Age: 20, Sex : M
Salary :2000

First Name: John, Last Name : Poul
Age: 25, Sex : M
```

```
Salary :2300
```

There are more shot cut methods to fetch records from the database. If you are interested then go through Fetching the Result otherwise proceed to next section.

Update Operation:

UPDATE Operation on any database means to update one or more records which are already available in the database. Following is the procedure to update all the records having SEX as 'M'. Here we will increase AGE of all the males by one year. This will take three steps

- Preparing SQL query based on required conditions. This will be done using **prepare** method.
- Executing SQL query to select all the results from the database. This will be done using **execute** method.
- Releasing Statement handle. This will be done using **finish** method.
- If everything goes fine then **commit** this operation otherwise you can **rollback** complete transaction.

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # connect to the MySQL server
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     sth = dbh.prepare("UPDATE EMPLOYEE SET AGE = AGE + 1
                          WHERE SEX = ?")
     sth.execute('M')
     sth.finish
     dbh.commit
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

DELETE Operation:

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where AGE is more than 20. This operation will take following steps.

- Preparing SQL query based on required conditions. This will be done using **prepare** method.
- Executing SQL query to delete required records from the database. This will be done using **execute** method.
- Releasing Statement handle. This will be done using **finish** method.
- If everything goes fine then **commit** this operation otherwise you can **rollback** complete transaction.

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # connect to the MySQL server
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
```

```
                      "testuser", "test123")
     sth = dbh.prepare("DELETE FROM EMPLOYEE
                        WHERE AGE > ?")
     sth.execute(20)
     sth.finish
     dbh.commit
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

Performing Transactions:

Transactions are a mechanism that ensures data consistency. Transactions should have the following four properties:

- **Atomicity:** Either a transaction completes or nothing happens at all.
- **Consistency:** A transaction must start in a consistent state and leave the system is a consistent state.
- **Isolation:** Intermediate results of a transaction are not visible outside the current transaction.
- **Durability:** Once a transaction was committed, the effects are persistent, even after a system failure.

The DBI provides two methods to either *commit* or *rollback* a transaction. There is one more method called *transaction* which can be used to implement transactions. There are two simple approaches to implement transactions:

Approach I:

The first approach uses DBI's *commit* and *rollback* methods to explicitly commit or cancel the transaction:

```
   dbh['AutoCommit'] = false # Set auto commit to false.
   begin
     dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
             WHERE FIRST_NAME = 'John'")
     dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
             WHERE FIRST_NAME = 'Zara'")
     dbh.commit
   rescue
     puts "transaction failed"
     dbh.rollback
   end
   dbh['AutoCommit'] = true
```

Approach II:

The second approach uses the *transaction* method. This is simpler, because it takes a code block containing the statements that make up the transaction. The *transaction* method executes the block, then invokes *commit* or *rollback* automatically, depending on whether the block succeeds or fails:

```
   dbh['AutoCommit'] = false # Set auto commit to false.
   dbh.transaction do |dbh|
     dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
             WHERE FIRST_NAME = 'John'")
     dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
             WHERE FIRST_NAME = 'Zara'")
```

```
   end
   dbh['AutoCommit'] = true
```

COMMIT Operation:

Commit is the operation, which gives a green signal to database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call **commit** method.

```
    dbh.commit
```

ROLLBACK Operation:

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use **rollback** method.

Here is a simple example to call **rollback** method.

```
    dbh.rollback
```

Disconnecting Database:

To disconnect Database connection, use disconnect API.

```
   dbh.disconnect
```

If the connection to a database is closed by the user with the disconnect method, any outstanding transactions are rolled back by the DBI. However, instead of depending on any of DBI's implementation details, your application would be better off calling commit or rollback explicitly.

Handling Errors:

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.

If a DBI method fails, DBI raises an exception. DBI methods may raise any of several types of exception but the two most important exception classes are *DBI::InterfaceError* and *DBI::DatabaseError*.

Exception objects of these classes have three attributes named *err*, *errstr*, and *state*, which represent the error number, a descriptive error string, and a standard error code. The attributes are explained below:

- **err:** Returns an integer representation of the occurred error or *nil* if this is not supported by the DBD.The Oracle DBD for example returns the numerical part of an *ORA-XXXX* error message.
- **errstr:** Returns a string representation of the occurred error.
- **state:** Returns the SQLSTATE code of the occurred error.The SQLSTATE is a five-character-long string. Most DBDs do not support this and return nil instead.

You have seen following code above in most of the examples:

```
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # disconnect from server
     dbh.disconnect if dbh
end
```

To get debugging information about what your script is doing as it executes, you can enable tracing. To do this, you must first load the dbi/trace module and then call the *trace* method that controls the trace mode and output destination:

```
require "dbi/trace"
..............

trace(mode, destination)
```

The mode value may be 0 *off*, 1, 2, or 3, and the destination should be an IO object. The default values are 2 and STDERR, respectively.

Code Blocks with Methods

There are some methods which creates handles. These methods can be invoked with a code block. The advantage of using code block along with methods is that they provide the handle to the code block as its parameter and automatically clean up the handle when the block terminates. There are few examples to understand the concept

- **DBI.connect :** This method generates a database handle and it is recommended to call *disconnect* at the end of the block to disconnect the database.
- **dbh.prepare :** This method generates a statement handle and it is recommended to *finish* at the end of the block. Within the block, you must invoke *execute* method to execute the statement.
- **dbh.execute :** This method is similar except we don't need to invoke execute within the block. The statement handle is automatically executed.

Example 1:

DBI.connect can take a code block, passes the database handle to it, and automatically disconnects the handle at the end of the block as follows.

```
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                  "testuser", "test123") do |dbh|
```

Example 2:

dbh.prepare can take a code block, passes the statement handle to it, and automatically calls finish at the end of the block as follows.

```
dbh.prepare("SHOW DATABASES") do |sth|
       sth.execute
       puts "Databases: " + sth.fetch_all.join(", ")
end
```

Example 3:

dbh.execute can take a code block, passes the statement handle to it, and automatically calls finish at the end of the block as follows:

```
dbh.execute("SHOW DATABASES") do |sth|
   puts "Databases: " + sth.fetch_all.join(", ")
end
```

DBI *transaction* method also takes a code block which has been described in above.

Driver-specific Functions and Attributes:

The DBI lets database drivers provide additional database-specific functions, which can be called by the user through the *func* method of any Handle object.

Driver-specific attributes are supported and can be set or gotten using the **[]=** or **[]** methods.

DBD::Mysql implements the following driver-specific functions:

S.N.	Functions with Description
1	**dbh.func:** *createdb, db_name* Creates a new database
2	**dbh.func:** *dropdb, db_name* Drops a database
3	**dbh.func:** *reload* Performs a reload operation
4	**dbh.func:** *shutdown* Shut down the server
5	**dbh.func:** *insert_id* **=> Fixnum** Returns the most recent AUTO_INCREMENT value for a connection.
6	**dbh.func:** *client_info* **=> String** Returns MySQL client information in terms of version.
7	**dbh.func:** *client_version* **=> Fixnum** Returns client information in terms of version. It's similar to :client_info but it return a fixnum instead of sting.
8	**dbh.func:** *host_info* **=> String** Returns host information
9	**dbh.func:** *proto_info* **=> Fixnum** Returns protocol being used for the communication.
10	**dbh.func:** *server_info* **=> String** Returns MySQL server information in terms of version.
11	**dbh.func:** *stat* **=> String** Returns current state of the database.
12	**dbh.func:** *thread_id* **=> Fixnum** Return current thread ID.

Example:

```
#!/usr/bin/ruby

require "dbi"
begin
   # connect to the MySQL server
   dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                     "testuser", "test123")
   puts dbh.func(:client_info)
   puts dbh.func(:client_version)
   puts dbh.func(:host_info)
   puts dbh.func(:proto_info)
   puts dbh.func(:server_info)
   puts dbh.func(:thread_id)
   puts dbh.func(:stat)
rescue DBI::DatabaseError => e
   puts "An error occurred"
   puts "Error code:    #{e.err}"
   puts "Error message: #{e.errstr}"
```

```
ensure
   dbh.disconnect if dbh
end
```

This will produce the following result:

```
5.0.45
50045
Localhost via UNIX socket
10
5.0.45
150621
Uptime: 384981  Threads: 1  Questions: 1101078  Slow queries: 4 \
Opens: 324  Flush tables: 1  Open tables: 64  \
Queries per second avg: 2.860
```