IBM Db2 V11.5

Developing Node.JS, Perl, PHP, Python, and Rubyon Rails Applications

2020-08-19

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

© Copyright IBM Corp. 2016, 2020

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:
© (your company name) (year).
Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. _enter the year or years_.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Contents

Tables

Chapter 1. Developing Node.jsl Applications

node-ibm_db driver for Node.js applications

You can use the node-ibm_db driver in your Node.js applications to access IBM® database servers.

Node.js is a software platform that is built on JavaScript. Node.js provides a fast, scalable, lightweight application solution for data-intensive real-time applications.

The node-ibm_db driver is a Node.js binding for IBM database servers. The node-ibm_db driver contains both asynchronous and synchronous interfaces.

You can install the node-ibm_db driver with the following IBM data server client products:

- All supported versions, releases, and fix packs of the IBM Data Server Driver Package product
- All Db2® Cancun Release 10.5.0.4 or later IBM data server products

Resources for the node-ibm_db driver

A list of URLs for the node-ibm_db driver.

Table 1 lists resources that are related to the node-ibm_db driver and required IBM data server products.

Table 1. Resources for the node-ibm_db driver and required IBM data server products

Description	URL
Latest information about the IBM node-ibm_db driver. The link does not point to an IBM site.	https://www.npmjs.org/package/ibm_db[1]
Sample code. The link does not point to an IBM site.	https://github.com/ibmdb/node-ibm_db/[1]
List of installation requirements for Db2 database products.	http://www-01.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.qb.server.doc/doc/r0025127.html
List of installation requirements for theIBM Informix® server.	http://www-01.ibm.com/support/knowledgecenter/SSGU8G_11.50.0/com.ibm.expr.doc/ids_in_004x.htm
IBM Data Server Driver Package software download site	https://www.ibm.com/support/pages/node/387577
IBM node-ibm_db driver forum. The link does not point to an IBM site.	https://groups.google.com/forum/#!forum/node-ibm_db[1]
Reported issues. The link does not point to an IBM site.	https://github.com/ibmdb/node-ibm_db/issues[1]

Installing the node-ibm_db driver on Linux and UNIX systems

You can install the node-ibm_db driver on Linux® and UNIX systems for use with Node.js applications.

[1] Any references to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The content available at those non-IBM websites is not part of any materials relating to the IBM products described herein. Your use of any non-IBM website is at your own risk.

© Copyright IBM Corp. 2016, 2020

Before you begin

You must install one of the listed IBM data server products on the system where you are installing or running the Node.js application:

- All supported versions, releases, and fix packs of the IBM Data Server Driver Package product
- All Db2 Cancun Release 10.5.0.4 or later IBM data server products

Procedure

To install the node-ibm_db driver:

1. Set the **IBM_DB_HOME** environment variable to your IBM data server product installation directory by issuing the following command:

   ```
   export IBM_DB_HOME=DB2HOME
   ```

   where *DB2HOME* is the directory where the IBM data server product is installed. In a IBM Data Server Driver Package environment, *DB2HOME* is the directory in which the client package is installed. For example, if the client package is installed in the `/home/db2inst1/dsdriver` directory, issue the following command:

   ```
   export IBM_DB_HOME=/home/db2inst1/dsdriver
   ```

2. Issue the **npm install** command:

   ```
   npm install ibm_db
   ```

What to do next

Before you run any Node.js application that connects to an IBM database server, you must ensure that the node-ibm_db driver can access the CLI driver by sourcing the `db2profile` script. The script is in the IBM data server product installation directory. To source the script, issue the following command:

```
source DB2HOME/db2profile
```

Verifying the node-ibm_db driver installation

You can test the node-ibm_db driver with a sample Node.js application.

Procedure

To verify the node-ibm_db driver installation:

1. Create a sample Node.js application to test the connection to the sample database.

   Copy the following sample code into a file and save the file as `test1.js`.

   ```
   //test1.js
   var ibmdb = require('ibm_db');

   ibmdb.open("DRIVER={DB2};DATABASE=sample;HOSTNAME=<hostname>;
   UID=<user_id>;PWD=<password>;PORT=<port>;
   PROTOCOL=TCPIP", function (err,conn) {
     if (err) return console.log(err);

     conn.query('select * from staff where id = ?', [10], function (err, data) {
       if (err) console.log(err);

       console.log(data);

       conn.close(function () {
         console.log('done');
       });
     });
   });
   ```

   where:

   - *<hostname>* is the fully qualified host name of your IBM database server.

- *<user_id>* and *<password>* are a valid user ID and password for connecting to the sample database.
- *<port>* is the listener port of the IBM database server.

2. Run the `test1.js` application by issuing the **node test1.js** command:

```
C:\Users\IBM_ADMIN>node test1.js
```

The **node test1.js** command results in the following output.

```
[ { ID: 10,
    NAME: 'Sanders',
    DEPT: 20,
    JOB: 'Mgr  ',
    YEARS: 7,
    SALARY: 98357.5,
    COMM: 1.5537297e-317 } ]
done
```

Chapter 2. Developing Perl Applications

Programming considerations for Perl

Perl database Interface (DBI) is an open standard application programming interface (API) that provides database access for client applications that are written in Perl. Perl DBI defines a set of functions, variables, and conventions that provide a platform-independent database interface.

You can use the IBM Db2 database driver for Perl DBI (the DBD::DB2 driver) available from http://search.cpan.org/~ibmtordb2/ along with the Perl DBI Module available from http://search.cpan.org/~timb/ to create a Perl application that access the IBM database server.

Because Perl is an interpreted language and the Perl DBI module uses dynamic SQL, Perl is an ideal language for quickly creating and revising prototypes of Db2 applications. The Perl DBI module uses an interface that is similar to the CLI and JDBC interfaces, which makes it easy for you to port your Perl prototypes to CLI and JDBC.

The working versions of Perl that works with Db2 are Activestate Perl and 32-bit Strawberry Perl. For 32-bit Strawberry Perl specifically, you must email opendev@us.ibm.com or contact IBM support for the required binaries.

For information about supported Database servers, installation instructions, and prerequisites, see http://search.cpan.org/~ibmtordb2/

Perl downloads and related resources

Several resources are available to help you develop Perl applications that access IBM database servers.

Table 2. Perl downloads and related resources

Downloads	**Related resources**
Perl Database Interface (DBI) Module	http://search.cpan.org/~timb/
DBD::DB2 driver	http://search.cpan.org/~ibmtordb2/
IBM Data Server Driver Package (DS Driver)	http://www.ibm.com/software/data/support/data-server-clients/index.html
DBI API documentation	http://search.cpan.org/~timb/DBI/DBI.pm
Db2 Perl Database Interface for Db2 technote, including readme and installation instructions	http://www.ibm.com/software/data/db2/perl
Perl driver bug reporting system	http://rt.cpan.org/
Reporting bugs to the Open Source team at IBM	opendev@us.ibm.com

Database connections in Perl

The DBD::DB2 driver provides support for standard database connection functions defined by the DBI API.

To enable Perl to load the DBI module, you must include the `use DBI;` line in your application:

The DBI module automatically loads the DBD::DB2 driver when you create a database handle using the **`DBI->connect`** statement with the listed syntax:

```
my $dbhandle = DBI->connect('dbi:DB2:dsn', $userID, $password);
```

where:

© Copyright IBM Corp. 2016, 2020

$dbhandle
represents the database handle returned by the connect statement

dsn

for local connections, represents a Db2 alias cataloged in your Db2 database directory

for remote connections, represents a complete connection string that includes the host name, port number, protocol, user ID, and password for connecting to the remote host

$userID
represents the user ID used to connect to the database

$password
represents the password for the user ID used to connect to the database

For more information about the DBI API, see http://search.cpan.org/~timb/DBI/DBI.pmhttp://search.cpan.org/~timb/DBI/DBI.pm.

Examples

Example 1: Connect to a database on the local host (client and server are on the same workstation)

```
use DBI;

$DATABASE = 'dbname';
$USERID = 'username';
$PASSWORD = 'password';

my $dbh = DBI->connect("dbi:DB2:$DATABASE", $USERID, $PASSWORD, {PrintError => 0})
or die "Couldn't connect to database: " . DBI->errstr;

$dbh->disconnect;
```

Example 2: Connect to a database on the remote host (client and server are on different workstations)

```
use DBI;

$DSN="DATABASE=sample; HOSTNAME=host; PORT=60000; PROTOCOL=TCPIP; UID=username;
PWD=password";

my $dbh = DBI->connect("dbi:DB2:$DSN", $USERID, $PASSWORD, {PrintError => 0})
or die "Couldn't connect to database: " . DBI->errstr;

$dbh->disconnect;
```

Fetching results in Perl

The Perl DBI module provides methods for connecting to a database, preparing and issuing SQL statements, and fetching rows from result sets.

About this task

This procedure fetches results from an SQL query.

Restrictions

Because the Perl DBI module supports only dynamic SQL, you cannot use host variables in your Perl Db2 applications.

Procedure

To fetch results:

1. Create a database handle by connecting to the database with the `DBI->connect` statement.
2. Create a statement handle from the database handle.

For example, you can return the statement handle $sth from the database handle by calling the prepare method and passing an SQL statement as a string argument, as demonstrated in the Perl statement example:

```
my $sth = $dbhandle->prepare(
   'SELECT firstnme, lastname
      FROM employee '
   );
```

3. Issue the SQL statement by calling the execute method on the statement handle. A successful call to the execute method associates a result set with the statement handle.
   For example, you can run the statement prepared in the previous Perl statement by using the listed example:

```
#Note: $rc represents the return code for the execute call
my $rc = $sth->execute();
```

4. Fetch a row from the result set associated with the statement handle by calling the fetchrow method. The Perl DBI returns a row as an array with one value per column.
   For example, you can return all of the rows from the statement handle in the previous example by using the listed Perl statement:

```
while (($firstnme, $lastname) = $sth->fetchrow()) {
   print "$firstnme $lastname\n";
}
```

Examples

The example shows how to connect to a database and issue a SELECT statement from an application written in Perl.

```
#!/usr/bin/perl
use DBI;

my $database='dbi:DB2:sample';
my $user='';
my $password='';

my $dbh = DBI->connect($database, $user, $password)
   or die "Can't connect to $database: $DBI::errstr";

my $sth = $dbh->prepare(
   q{ SELECT firstnme, lastname
      FROM employee }
   )
   or die "Can't prepare statement: $DBI::errstr";

my $rc = $sth->execute
   or die "Can't execute statement: $DBI::errstr";

print "Query will return $sth->{NUM_OF_FIELDS} fields.\n\n";
print "$sth->{NAME}->[0]: $sth->{NAME}->[1]\n";

while (($firstnme, $lastname) = $sth->fetchrow()) {
   print "$firstnme: $lastname\n";
}

# check for problems that might have terminated the fetch early
warn $DBI::errstr if $DBI::err;

$sth->finish;
$dbh->disconnect;
```

Parameter markers in Perl

The Perl DBI module supports executing a prepared statement that includes parameter markers for variable input. To include a parameter marker in an SQL statement, use the question mark (?) character or a colon followed by a name (`:name`).

The Perl code example creates a statement handle that accepts a parameter marker for the WHERE clause of a SELECT statement. The code then executes the statement twice using the input values 25000 and 35000 to replace the parameter marker.

```
   my $sth = $dbhandle->prepare(
      'SELECT firstnme, lastname
         FROM employee
         WHERE salary > ?'
      );

   my $rc = $sth->execute(25000);

.
.
.

   my $rc = $sth->execute(35000);
```

SQLSTATE and SQLCODE variables in Perl

The Perl DBI module provides methods for returning the SQLSTATE and SQLCODE associated with a Perl DBI database or statement handle.

To return the SQLSTATE associated with a Perl DBI database handle or statement handle, call the `state` method. For example, to return the SQLSTATE associated with the database handle $dbhandle, include the `my $sqlstate = $dbhandle->state;` Perl statement in your application:

To return the SQLCODE associated with a Perl DBI database handle or statement handle, call the `err` method. To return the message for an SQLCODE associated with a Perl DBI database handle or statement handle, call the `errstr` method. For example, to return the SQLCODE associated with the database handle $dbhandle, include the `my $sqlcode = $dbhandle->err;` Perl statement in your application:

Perl Restrictions

Some restrictions apply to the support that is available for application development in Perl.

The Perl DBI module supports only dynamic SQL. When you must execute a statement multiple times, you can improve the performance of your Perl applications by issuing a **`prepare`** call to prepare the statement.

For current information about the restrictions on a specific version of the DBD::DB2 driver installation, see the CAVEATS file in the DBD::Db2 driver package.

pureXML and Perl

The DBD::DB2 driver supports Db2 pureXML®. Support for pureXML allows more direct access to your data through the DBD::DB2 driver and helps to decrease application logic by providing more transparent communication between your application and database.

With pureXML support, you can directly insert XML documents into your Db2 database. Your application no longer needs to parse XML documents because the pureXML parser is automatically run when you insert XML data into the database. Having document parsing handled outside your application improves application performance and reduces maintenance efforts. Retrieval of XML stored data with the DBD::DB2 driver is easy as well; you can access the data using a BLOB or record.

For information about the Db2 Perl Database Interface and how to download the latest DBD::DB2 driver, see http://www.ibm.com/software/data/db2/perl.

Examples

The example is a Perl program that uses pureXML:

```
#!/usr/bin/perl
use DBI;
use strict ;

# Use DBD:DB2 module:
#   to create a simple Db2 table with an XML column
#   Add one row of data
#   retreive the XML data as a record or a LOB (based on $datatype).

# NOTE: the Db2 SAMPLE database must already exist.

my $database='dbi:Db2:sample';
my $user='';
my $password='';

my $datatype = "record" ;
#  $datatype = "LOB" ;

my $dbh = DBI->connect($database, $user, $password)
  or die "Can't connect to $database: $DBI::errstr";

# For LOB datatype, LongReadLen = 0 -- no data is retrieved on initial fetch
$dbh->{LongReadLen} = 0 if $datatype eq "LOB" ;

# SQL CREATE TABLE to create test table
my $stmt = "CREATE TABLE xmlTest (id INTEGER, data XML)";
my $sth = $dbh->prepare($stmt);
$sth->execute();

#insert one row of data into table
insertData() ;

# SQL SELECT statement returns home phone element from XML data
$stmt = qq(
    SELECT XMLQUERY ('
    \$d/*:customerinfo/*:phone[\@type = "home"] '
    passing data as "d")
    FROM xmlTest
) ;


# prepare and execute SELECT statement
$sth = $dbh->prepare($stmt);
$sth->execute();

# Print data returned from select statement
if($datatype eq "LOB") {
    printLOB() ;
}
else {
    printRecord() ;
}

# Drop table
$stmt = "DROP TABLE xmlTest" ;
$sth = $dbh->prepare($stmt);
$sth->execute();

warn $DBI::errstr if $DBI::err;

$sth->finish;
$dbh->disconnect;


################


sub printRecord {
    print "output data as record\n" ;

    while( my @row = $sth->fetchrow )
    {
        print $row[0] . "\n";
    }

    warn $DBI::errstr if $DBI::err;
}
```

```
sub printLOB {
    print "output as Blob data\n" ;

    my $offset = 0;
    my $buff="";
    $sth->fetch();
    while( $buff = $sth->blob_read(1,$offset,1000000)) {
        print $buff;
        $offset+=length($buff);
        $buff="";
    }
    warn $DBI::errstr if $DBI::err;
}


sub insertData {

    # insert a row of data
    my $xmlInfo = qq(\'
    <customerinfo xmlns="http://posample.org" Cid="1011">
      <name>Bill Jones</name>
      <addr country="Canada">
        <street>5 Redwood</street>
        <city>Toronto</city>
        <prov-state>Ontario</prov-state>
        <pcode-zip>M6W 1E9</pcode-zip>
      </addr>
      <phone type="work">416-555-9911</phone>
      <phone type="home">416-555-1212</phone>
    </customerinfo>
    \') ;

    my $catID = 1011 ;

    # SQL statement to insert data.
    my $Sql = qq(
     INSERT INTO xmlTest (id, data)
         VALUES($catID, $xmlInfo )
    );

    $sth = $dbh->prepare( $Sql )
      or die "Can't prepare statement: $DBI::errstr";

    my $rc = $sth->execute
      or die "Can't execute statement: $DBI::errstr";

    # check for problems
    warn $DBI::errstr if $DBI::err;
}
```

Running Perl sample programs

Perl sample programs demonstrate how to connect and run database operations with the IBM database server..

Before you begin

Before running the Perl sample programs, you must install the latest DBD::DB2 driver for Perl DBI. For information about how to obtain the latest driver, see http://search.cpan.org/~ibmtordb2/.

About this task

The Perl sample programs for Db2 database are available in the `sqllib/samples/perl` directory.

Procedure

To run a Perl sample program through the Perl interpreter:

- Enter the interpreter name and the program name (including the file extension):
  - If connecting locally on the server:

```
perl dbauth.pl
```

- If connecting from a remote client:

```
perl dbauth.pl sample <userid> <password>
```

Some of the sample programs require you to run support files. For example, the `tbsel` sample program requires several tables that are created by the `tbselcreate.db2` CLP script. The `tbselinit` script (UNIX), or the `tbselinit.bat` batch file (Windows), first calls `tbseldrop.db2` to drop the tables if they exist, and then calls `tbselcreate.db2` to create them. Therefore, to run the `tbsel` sample program, issue the listed commands:

- If connecting locally on the server:

```
tbselinit
perl tbsel.pl
```

- If connecting from a remote client:

```
tbselinit
perl tbsel.pl sample <userid> <password>
```

Note: For a remote client, you must modify the connect statement in the `tbselinit` or `tbselinit.bat` file to hardcode your user ID and password: `db2 connect to sample user <userid> using <password>`

Executing routines from Perl applications

Db2 client applications can access routines (stored procedures and user-defined functions) that are created by supported host languages or by SQL procedures. For example, the sample program `spclient.pl` can access the SQL procedures spserver shared library, if it exists in the database.

Before you begin

To build a host language routine, you must have the appropriate compiler set up on the server. SQL procedures do not require a compiler. The shared library can be built on the server only, and not from a remote client.

Procedure

To create SQL procedures in a shared library and then accesses the procedures from a Perl application:

1. Create and catalog the SQL procedures in the library. For example, go to the `samples/sqlpl` directory on the server, and run the listed commands to create and catalog the SQL procedures in the `spserver` library:

```
db2 connect to sample
db2 -td@ -vf spserver.db2
```

2. Go back to the `perl` samples directory (this can be on a remote client workstation), and run the Perl interpreter on the client program to access the `spserver` shared library:

   - If connecting locally on the server:

```
perl spclient
```

   - If connecting from a remote client:

```
perl spclient sample <userid> <password>
```

Chapter 3. Developing PHP applications

PHP application development for IBM Database servers

PHP: Hypertext Preprocessor (PHP) is an interpreted programming language that is widely used for developing web applications. PHP is a popular language for web development because it is easy to learn, focuses on practical solutions, and supports the most commonly required functionality in web applications.

PHP is a modular language that enables you to customize the available functionality through the use of extensions. These extensions can simplify tasks such as reading, writing, and manipulating XML, creating SOAP clients and servers, and encrypting communications between server and browser. The most popular extensions for PHP, however, provide read and write access to databases so that you can easily create a dynamic database-driven website.

IBM provides the following PHP extensions for accessing IBM Database servers:

ibm_db2
A procedural application programming interface (API) that, in addition to the normal create, read, update, and write database operations, also offers extensive access to the database metadata. You can compile the ibm_db2 extension with either PHP 4 or PHP 5.

pdo_ibm
A driver for the PHP Data Objects (PDO) extension that offers access to IBM Database servers through the standard object-oriented database interface that is introduced in PHP 5.1.

The most recent versions of the ibm_db2 and pdo_ibm extensions are also available from the PHP Extension Community Library (PECL) at http://pecl.php.net/.

PHP downloads and related resources

Many resources are available to help you develop PHP applications for IBM Database servers.

Table 3. PHP downloads and related resources

Downloads	
Complete PHP source code [1]	http://www.php.net/downloads.php
The ibm_db2 extension from the PHP Extension Community Library (PECL)	http://pecl.php.net/package/ibm_db2
The pdo_ibm extension from the PHP Extension Community Library	http://pecl.php.net/package/pdo_ibm
Compiled ibm_db2 extension for Windows	http://windows.php.net/downloads/pecl/releases/ibm_db2/
Compiled pdo_ibm extension for Windows	http://windows.php.net/downloads/pecl/releases/pdo_ibm/
IBM Data Server Driver Package (DS Driver)	https://www.ibm.com/support/pages/node/387577
Zend Server	http://www.zend.com/en/products/server/downloads
PHP Manual	http://www.php.net/docs.php
ibm_db2 API documentation	http://www.php.net/ibm_db2%20

© Copyright IBM Corp. 2016, 2020

Table 3. PHP downloads and related resources (continued)	
Downloads	
PDO API documentation	http://php.net/manual/en/book.pdo.php
PHP website	http://www.php.net/

1. Includes the Windows binary files. Most Linux distributions come with PHP already precompiled.

Setting up the PHP environment for IBM Data Servers products

You can set up the PHP environment on Linux, UNIX, or Windows operating systems by installing a precompiled binary version of PHP and enabling support for IBM Data Servers products.

About this task

For the easiest installation and configuration experience on Linux, UNIX, or Windows operating systems, you can download and install Zend Server for use in production systems at http://www.zend.com/en/products/server/downloads. Packaging details are available at http://www.zend.com/en/products/server/editions.

On Windows, precompiled binary versions of PHP are available for download from http://www.php.net/downloads.php. Most Linux distributions include a precompiled version of PHP. On UNIX operating systems that do not include a precompiled version of PHP, you can compile your own version of PHP.

For more information about installing and configuring PHP, see http://www.php.net/manual/en/install.php.

Setting up the PHP environment for IBM Data Server products on Windows

Before you can connect to an IBM database server and run SQL statements, you must set up the PHP environment.

Before you begin

You must have the following required software installed on your system:

- PHP version 5 or later
- If your PHP application will connect to a remote IBM database, one of the following products on the computer where your application will run:
  - The IBM Data Server Client product
  - The IBM Data Server Runtime Client product
  - The IBM Data Server Driver Package product
  - The IBM Data Server Driver for ODBC and CLI product

  If your PHP application connects to an IBM database server on the local computer, no additional IBM data server products are required.

Procedure

To install the ibm_db2 and pdo_ibm php extensions:

1. Copy the ibm_db2 and pdo_ibm extension files into the `\ext\` subdirectory of your PHP installation directory.
   The ibm_db2 and pdo_ibm extension files can be obtained from the following sources:
   - IBM Data Server product installation path
   - PHP Extension Community Library (PECL)
     - For the ibm_db2 extension file, see http://windows.php.net/downloads/pecl/releases/ibm_db2/
     - For the pdo_ibm extension file, see http://windows.php.net/downloads/pecl/releases/pdo_ibm/

Note: If you installed the IBM Data Server Driver for ODBC and CLI software, you must obtain the ibm_db2 and pdo_ibm extension files separately from the PHP Extension Community Library (PECL).

- If you have thread safe PHP environment, copy the following extension files from the IBM Data Server product installation path into the `\ext\` subdirectory of your PHP installation directory:
  - `php_ibm_db2_X_X_XXX_ts.dll`
  - `php_pdo_ibm_X_X_XXX_ts.dll`
- If you have non-thread safe PHP environment, copy the following extension files from the IBM Data Server product installation path into the `\ext\` subdirectory of your PHP installation directory:
  - `php_ibm_db2_X_X_XXX_nts.dll`
  - `php_pdo_ibm_X_X_XXX_nts.dll`

2. Open the `php.ini` file in an editor of your choice. Edit the extension entry in the `php.ini` file in the PHP installation directory to reference the PHP driver.
   - For the thread safe PHP environment:

     ```
     extension=php_pdo.dll
     extension=php_ibm_db2_X_X_XXX_ts.dll
     extension=php_pdo_ibm_X_X_XXX_ts.dll
     ```

   - For the non-thread safe PHP environment:

     ```
     extension=php_pdo.dll
     extension=php_ibm_db2_X_X_XXX_nts.dll
     extension=php_pdo_ibm_X_X_XXX_nts.dll
     ```

3. If the PHP application that is connecting to an IBM database server is running in the HTTP server environment, restart the HTTP Server so the new configuration settings take effect.

Setting up the PHP environment for IBM Data Server products on Linux or UNIX
Before you can connect to an IBM database server and run SQL statements, you must set up the PHP environment.

Before you begin

You must have the following required software installed on your system:

- PHP version 5 or later
- If your PHP application connects to a remote IBM database, the computer that runs your PHP application requires one of the following products:
  - The IBM Data Server Client product
  - The IBM Data Server Runtime Client product
  - The IBM Data Server Driver Package product
  - The IBM Data Server Driver for ODBC and CLI product

  If your PHP application connects to an IBM database server on the local computer, no additional IBM data server products are required.

Procedure

To install the ibm_db2 and pdo_ibm php extensions:

1. Using the export command, set the environment variable IBM_DB_HOME.

   ```
   $export IBM_DB_HOME=DB2HOME
   ```

   The DB2HOME is the directory where the IBM Data Server product is installed. For example:

   ```
   $ export IBM_DB_HOME=/home/db2inst1/sqllib
   ```

2. Using one of the following three methods, install the ibm_db and pdo_ibm extensions.
   - Use the **pecl install** command included in the PHP Extension Community Library (PECL).
     – To install the ibm_db2 extension:

```
$ pecl install ibm_db2
```

     – To install the pdo_ibm extension:

```
$ pecl install pdo_ibm
```

   - Use the commands included in the source code:
     a. Extract the source archive.
     b. Run the following commands from the extracted directory:

```
$ phpize --clean
$ phpize
$ ./configure
$ make
$ make install
```

     c. If you are installing the pdo_ibm extension, you must run the following configure command:

```
$./configure --with-PDO_IBM=DB2HOME
```

        The DB2HOME variable is the directory where the IBM Data Server product is installed.
   - Use the compiled extensions included with the IBM Data Server products:
     a. You must determine whether your PHP environment is threadsafe or not threadsafe by issuing the following command:

```
$ php -info| grep "Thread Safe"
```

     b. The IBM data server client and IBM Data Server Driver Package software are shipped with two types of PHP drivers:
        – Threadsafe: `ibm_db2_XX_ts.so` and `pdo_ibm_XX_ts.so`
        – Not threadsafe: `ibm_db2_XX_nts.so` and `pdo_ibm_XX_nts.so`

        Using the **cp** command, copy the appropriate PHP driver share library files to the installed PHP extension directory as `ibm_db2.so` and `pdo_ibm.so` files.

        For a 32-bit PHP driver:

```
$ cp DB2HOME/php/php32/ibm_db2_XX_[ts/nts].so <local_php_directory>/php/lib/php/extensions/ibm_db2.so
$ cp DB2HOME/php/php32/pdo_ibm_XX_[ts/nts].so <local_php_directory>/php/lib/php/extensions/pdo_ibm.so
```

        For a 64-bit PHP driver:

```
$ cp DB2HOME/php/php64/ibm_db2_XX_[ts/nts].so <local_php_directory>/php/lib/php/extensions/ibm_db2.so
$ cp DB2HOME/php/php64/pdo_ibm_XX_[ts/nts].so <local_php_directory>/php/lib/php/extensions/pdo_ibm.so
```

        The DB2HOME variable is the directory where the IBM Data Server product is installed.
3. Open the `php.ini` file in an editor of your choice. Edit the extension entry in the `php.ini` file in the *<local_php_directory>*/`php/lib` directory to reference the PHP driver:

```
extension=pdo.so
extension=ibm_db2.so
extension=pdo_ibm.so
```

4. Ensure that the PHP driver can access the `libdb2.so` CLI driver file by setting the LD_LIBRARY_PATH variable for Linux and UNIX operating systems other than the AIX operating system. For AIX operating system, you must set LIBPATH variable.

- For a 32-bit Linux and UNIX operating systems other than the AIX operating system, use the **export** command to set the *IBM_DB_HOME*/lib32 directory to the LD_LIBRARY_PATH variable.

```
export LD_LIBRARY_PATH=DB2HOME/lib32
```

- For a 32-bit AIX operating system, use the **export** command to set the *IBM_DB_HOME*/lib32 directory to the: LIBPATH variable.

```
export LIBPATH=DB2HOME/lib32
```

- For a 64-bit Linux and UNIX operating systems other than the AIX operating system, use the **export** command to set the LD_LIBRARY_PATH variable to the *IBM_DB_HOME*/lib64 directory.

```
export LD_LIBRARY_PATH=DB2HOME/lib64
```

- For a 64-bit AIX operating system, use the **export** command to set the LD_LIBRARY_PATH variable to the *IBM_DB_HOME*/lib64 directory.

```
export LIBPATH=DB2HOME/lib64
```

5. Optional: If the PHP application that is connecting to an IBM database server is running in the HTTP server environment:
   a) Add the LD_LIBRARY_PATH variable in the `httpd.conf` file.
      For a 32-bit architecture, set LD_LIBRARY_PATH to the DB2HOME\lib32 directory. For a 64-bit architecture set LD_LIBRARY_PATH to the DB2HOME\lib64 directory.
   b) Restart the HTTP server so the new configuration settings take effect.

Application development in PHP (ibm_db2)

The ibm_db2 extension provides a variety of useful PHP functions for accessing and manipulating data in an IBM data server database. The extension includes functions for connecting to a database, executing and preparing SQL statements, fetching rows from result sets, calling stored procedures, handling errors, and retrieving metadata.

Connecting to an IBM data server database in PHP (ibm_db2)

Before you can issue SQL statements to create, update, delete, or retrieve data, you must connect to a database from your PHP application. You can use the ibm_db2 API to connect to an IBM data server database through either a cataloged connection or a direct TCP/IP connection. To improve performance, you can also create a persistent connection.

Before you begin

Before connecting to an IBM data server database through the ibm_db2 extension, you must set up the PHP environment on your system and enable the ibm_db2 extension.

Procedure

To return a connection resource that you can use to call SQL statements, call one of the listed connection functions:

Table 4. ibm_db2 connection functions

Function	Description
db2_connect	Creates a non-persistent connection.
db2_pconnect	Creates a persistent connection. A persistent connection remains open between PHP requests, which allows subsequent PHP script requests to reuse the connection if they have an identical set of credentials.

The database values that you pass as arguments to these functions can specify either a cataloged database name or a complete database connection string for a direct TCP/IP connection. You can specify optional arguments that control when transactions are committed, the case of the column names that are returned, and the cursor type.

If the connection attempt fails, you can retrieve diagnostic information by calling the `db2_conn_error` or `db2_stmt_errormsg` function.

When you create a connection by calling the `db2_connect` function, PHP closes the connection to the database when one of the listed events occurs:

- You call the `db2_close` function for the connection
- You set the connection resource to NULL
- The PHP script finishes

When you create a connection by calling the `db2_pconnect` function, PHP ignores any calls to the `db2_close` function for the specified connection resource, and keeps the connection to the database open for subsequent PHP scripts.

For more information about the ibm_db2 API, see http://www.php.net/docs.php.

Example

Connect to a cataloged database.

```
<?php
$database = "sample";
$user = "db2inst1";
$password = "";

$conn = db2_connect($database, $user, $password);

if ($conn) {
echo "Connection succeeded.";
db2_close($conn);
}
else {
echo "Connection failed.";
}
?>
```

What to do next

If the connection attempt is successful, you can use the connection resource when you call ibm_db2 functions that execute SQL statements. Next, prepare and execute SQL statements.

Trusted contexts in PHP applications (ibm_db2)
Starting in Version 9.5 Fix Pack 3 (or later), the ibm_db2 extension supports trusted contexts by using connection string keywords.

Trusted contexts provide a way of building much faster and more secure three-tier applications. The user's identity is always preserved for auditing and security purposes. When you need secure connections, trusted contexts improve performance because you do not have to get new connections.

Examples

Enable trusted contexts, switch users, and get the current user ID.

```
<?php

$database = "SAMPLE";
$hostname = "localhost";
$port = 50000;
$authID = "db2inst1";
$auth_pass = "ibmdb2";

$tc_user = "tcuser";
$tc_pass = "tcpassword";
```

```
$dsn = "DATABASE=$database;HOSTNAME=$hostname;PORT=$port;PROTOCOL=TCPIP;UID=$authID;PWD=$auth_pass;";
$options = array ("trustedcontext" => DB2_TRUSTED_CONTEXT_ENABLE);

$tc_conn = db2_connect($dsn, "", "", $options);
if($tc_conn) {
    echo "Explicit Trusted Connection succeeded.\n";

    if(db2_get_option($tc_conn, "trustedcontext")) {
        $userBefore = db2_get_option($tc_conn, "trusted_user");

        //Do some work as user 1.

        //Switching to trusted user.
        $parameters = array("trusted_user" => $tc_user, "trusted_password" => $tcuser_pass);
        $res = db2_set_option ($tc_conn, $parameters, 1);

        $userAfter = db2_get_option($tc_conn, "trusted_user");
        //Do more work as trusted user.

        if($userBefore != $userAfter) {
            echo "User has been switched." . "\n";
        }
    }

    db2_close($tc_conn);
}
else {
    echo "Explicit Trusted Connection failed.\n";
}

?>
```

Executing SQL statements in PHP (ibm_db2)

After connecting to a database, use functions available in the ibm_db2 API to prepare and execute SQL statements. The SQL statements can contain static text, XQuery expressions, or parameter markers that represent variable input.

Executing a single SQL statement in PHP (ibm_db2)

To prepare and execute a single SQL statement that accepts no input parameters, use the db2_exec function. A typical use of the db2_exec function is to set the default schema for your application in a common include file or base class.

Before you begin

To avoid the security threat of SQL injection attacks, use the db2_exec function only to execute SQL statements composed of static strings. Interpolation of PHP variables representing user input into the SQL statement can expose your application to SQL injection attacks.

Obtain a connection resource by calling one of the connection functions in the ibm_db2 API. Refer to “Connecting to an IBM data server database in PHP (ibm_db2)” on page 17.

Procedure

To prepare and execute a single SQL statement, call the db2_exec function, passing the listed arguments:

connection
A valid database connection resource returned from the db2_connect or db2_pconnect function.

statement
A string that contains the SQL statement. This string can include an XQuery expression that is called by the XMLQUERY function.

options
Optional: An associative array that specifies statement options:

DB2_ATTR_CASE
For compatibility with database systems that do not follow the SQL standard, this option sets the case in which column names will be returned to the application. By default, the case is set to DB2_CASE_NATURAL, which returns column names as they are returned by the database. You

can set this parameter to DB2_CASE_LOWER to force column names to lowercase, or to DB2_CASE_UPPER to force column names to upper case.

DB2_ATTR_CURSOR
This option sets the type of cursor that ibm_db2 returns for result sets. By default, `ibm_db2` returns a forward-only cursor (DB2_FORWARD_ONLY) which returns the next row in a result set for every call to `db2_fetch_array`, `db2_fetch_assoc`, `db2_fetch_both`, `db2_fetch_object`, or `db2_fetch_row`. You can set this parameter to DB2_SCROLLABLE to request a scrollable cursor so that the `ibm_db2` fetch functions accept a second argument specifying the absolute position of the row that you want to access within the result set.

If the function call succeeds, it returns a statement resource that you can use in subsequent function calls related to this query.

If the function call fails (returns `False`), you can use the `db2_stmt_error` or `db2_stmt_errormsg` function to retrieve diagnostic information about the error.

For more information about the `ibm_db2` API, see http://www.php.net/docs.php.

Example

Example 1: Executing a single SQL statement.

```
<?php
$conn = db2_connect("sample", "db2inst1", "");
$sql = "SELECT * FROM DEPT";
$stmt = db2_exec($conn, $sql);
db2_close($conn);
?>
```

Example 2: Executing an XQuery expression

```
<?php
$xquery = '$doc/customerinfo/phone';
$stmt = db2_exec($conn, "select xmlquery('$xquery'
PASSING INFO AS \"doc\") from customer");?>
```

What to do next

If the SQL statement selected rows using a scrollable cursor, or inserted, updated, or deleted rows, you can call the `db2_num_rows` function to return the number of rows that the statement returned or affected. If the SQL statement returned a result set, you can begin fetching rows.

Preparing and executing SQL statements with variable input in PHP (ibm_db2)
To prepare and execute an SQL statement that includes variable input, use the `db2_prepare`, `db2_bind_param`, and `db2_execute` functions. Preparing a statement improves performance because the database server creates an optimized access plan for data retrieval that it can reuse if the statement is executed again.

Before you begin

Obtain a connection resource by calling one of the connection functions in the ibm_db2 API. Refer to "Connecting to an IBM data server database in PHP (ibm_db2)" on page 17.

Procedure

To prepare and execute an SQL statement that includes parameter markers:

1. Call the `db2_prepare` function, passing the listed arguments:

   connection
   A valid database connection resource returned from the `db2_connect` or `db2_pconnect` function.

statement
A string that contains the SQL statement, including question marks (?) as parameter markers for any column or predicate values that require variable input. This string can include an XQuery expression that is called the XMLQUERY function. You can only use parameter markers as a place holder for column or predicate values. The SQL compiler is unable to create an access plan for a statement that uses parameter markers in place of column names, table names, or other SQL identifiers.

options
Optional: An associative array that specifies statement options:

DB2_ATTR_CASE
For compatibility with database systems that do not follow the SQL standard, this option sets the case in which column names will be returned to the application. By default, the case is set to DB2_CASE_NATURAL, which returns column names as they are returned by the database. You can set this parameter to DB2_CASE_LOWER to force column names to lowercase, or to DB2_CASE_UPPER to force column names to upper case.

DB2_ATTR_CURSOR
This option sets the type of cursor that ibm_db2 returns for result sets. By default, `ibm_db2` returns a forward-only cursor (DB2_FORWARD_ONLY) which returns the next row in a result set for every call to `db2_fetch_array`, `db2_fetch_assoc`, `db2_fetch_both`, `db2_fetch_object`, or `db2_fetch_row`. You can set this parameter to DB2_SCROLLABLE to request a scrollable cursor so that the ibm_db2 fetch functions accept a second argument specifying the absolute position of the row that you want to access within the result set.

If the function call succeeds, it returns a statement handle resource that you can use in subsequent function calls that are related to this query.

If the function call fails (returns `False`), you can use the `db2_stmt_error` or `db2_stmt_errormsg` function to retrieve diagnostic information about the error.

2. Optional: For each parameter marker in the SQL string, call the `db2_bind_param` function, passing the listed arguments. Binding input values to parameter markers ensures that each input value is treated as a single parameter, which prevents SQL injection attacks against your application.

   stmt
   A prepared statement returned by the call to the `db2_prepare` function.

   parameter-number
   An integer that represents the position of the parameter marker in the SQL statement.

   variable-name
   A string that specifies the name of the PHP variable to bind to the parameter specified by *parameter-number*.

3. Call the `db2_execute` function, passing the listed arguments:

   stmt
   A prepared statement returned by the `db2_prepare` function.

   parameters
   Optional: An array that contains the values to use in place of the parameter markers, in order.

   For more information about the `ibm_db2` API, see http://www.php.net/docs.php.

Example

Prepare and execute a statement that includes variable input.

```
$sql = "SELECT firstnme, lastname FROM employee WHERE bonus > ? AND bonus < ?";
$stmt = db2_prepare($conn, $sql);
if (!$stmt) {
  // Handle errors
}

// Explicitly bind parameters
db2_bind_param($stmt, 1, $_POST['lower']);
```

```
db2_bind_param($stmt, 2, $_POST['upper']);

db2_execute($stmt);
// Process results

// Invoke prepared statement again using dynamically bound parameters
db2_execute($stmt, array($_POST['lower'], $_POST['upper']));
```

What to do next

If the SQL statement returns one or more result sets, you can begin fetching rows from the statement resource.

Inserting large objects in PHP (ibm_db2)

When you insert a large object into the database, rather than loading all of the data for a large object into a PHP string and passing it to the IBM data server database through an INSERT statement, you can insert large objects directly from a file on your PHP server.

Before you begin

Obtain a connection resource by calling one of the connection functions in the ibm_db2 API.

Procedure

To insert a large object into the database directly from a file:

1. Call the db2_prepare function to prepare an INSERT statement with a parameter marker that represents the large object column.
2. Set the value of a PHP variable to the path and name of the file that contains the data for the large object. The path can be relative or absolute, and is subject to the access permissions of the PHP executable file.
3. Call the db2_bind_param function to bind the parameter marker to the variable. The third argument to this function is a string representing the name of the PHP variable that holds the path and name of the file. The fourth argument is DB2_PARAM_FILE, which tells the ibm_db2 extension to retrieve the data from a file.
4. Call the db2_execute function to issue the INSERT statement.

Example

Insert a large object into the database.

```
$stmt = db2_prepare($conn, "INSERT INTO animal_pictures(picture) VALUES (?)");

$picture = "/opt/albums/spook/grooming.jpg";
$rc = db2_bind_param($stmt, 1, "picture", DB2_PARAM_FILE);
$rc = db2_execute($stmt);
```

Reading query result sets

Fetching rows or columns from result sets in PHP (ibm_db2)

When you run a statement that returns one or more result sets, use one of the functions available in the ibm_db2 extension to iterate through the returned rows of each result set. If your result set includes columns that contain large data, you can retrieve the data on a column-by-column basis to avoid large memory usage.

Before you begin

You must have a statement resource returned by either the db2_exec or db2_execute function that has one or more associated result sets.

Procedure

To fetch data from a result set:

1. Fetch data from a result set by calling one of the fetch functions.

Table 5. ibm_db2 fetch functions

Function	Description
	Returns an array,. The columns are 0-indexed by columnindexed. position, representing a row in a result set
	Returns an array,. indexed by column name, representing a row in a result set
	Returns an array, indexed by both column name and position, representing a row in a result set
	Sets the result set. Use this function to pointer to the next rowiterate through a result or requested rowset.
	Returns an object with. The properties of the properties representingobject map to the names columns in the fetchedof the columns in the rowresult set.

These functions accept the listed arguments:

stmt
A valid statement resource.

row_number
The number of the row that you want to retrieve from the result set. Row numbering begins with 1. Specify a value for this optional parameter if you requested a scrollable cursor when you called the db2_exec or db2_prepare function. With the default forward-only cursor, each call to a fetch method returns the next row in the result set.

2. Optional: If you called the db2_fetch_row function, for each iteration over the result set, retrieve a value from the specified column by calling the db2_result function. You can specify the column by either passing an integer that represents the position of the column in the row (starting with 0), or a string that represents the name of column.
3. Continue fetching rows until the fetch function returns False, which indicates that you have reached the end of the result set.

For more information about the ibm_db2 extension, see http://www.php.net/docs.php.

Example

Example 1: Fetch rows from a result set by calling the db2_fetch_object function

```
<?php
$conn = db2_connect("sample", "db2inst1", "password");
$sql = 'SELECT FIRSTNME, LASTNAME FROM EMPLOYEE WHERE EMPNO = ?';
$stmt = db2_prepare($conn, $sql);
db2_execute($stmt, array('000010'));
while ($row = db2_fetch_object($stmt)) {
  print "Name:
```

```
  {$row->FIRSTNME} {$row->LASTNAME}

  ";
  }
db2_close($conn);
?>
```

Example 2: Fetch rows from a result set by calling the db2_fetch_row function

```
<?php
$conn = db2_connect("sample", "db2inst1", "password");
$sql = 'SELECT FIRSTNME, LASTNAME FROM EMPLOYEE WHERE EMPNO = ?';
$stmt = db2_prepare($conn, $sql);
db2_execute($stmt, array('000010'));
while (db2_fetch_row($stmt)) {
  $fname = db2_result($stmt, 0);
  $lname = db2_result($stmt, 'LASTNAME');
  print "
  Name: $fname $lname

  ";
  }
db2_close($conn);
?>
```

Example 3: Fetch rows from a result set by calling the db2_fetch_both function

```
<?php
$conn = db2_connect("sample", "db2inst1", "password");
$sql = 'SELECT FIRSTNME, LASTNAME FROM EMPLOYEE WHERE EMPNO = ?';
$stmt = db2_prepare($conn, $sql);
db2_execute($stmt, array('000010'));
while ($row = db2_fetch_both($stmt)) {
  print "
  NAME: $row[0] $row[1]

  ";
  print "
  NAME: " . $row['FIRSTNME'] . " " . $row['LASTNAME'] . "

  ";
  }
db2_close($conn);
?>
```

What to do next

When you are ready to close the connection to the database, call the db2_close function. If you attempt to close a persistent connection that you created by using db2_pconnect, the close request returns TRUE, and the IBM data server client connection remains available for the next caller.

Fetching large objects in PHP (ibm_db2)

When you fetch a large object from a result set, rather than treating the large object as a PHP string, you can save system resources by fetching large objects directly into a file on your PHP server.

Before you begin

Obtain a connection resource by calling one of the connection functions in the ibm_db2 API.

Procedure

To fetch a large object from the database directly into a file:

1. Create a PHP variable representing a stream. For example, assign the return value from a call to the fopen function to a variable.
2. Create a SELECT statement by calling the db2_prepare function.
3. Bind the output column for the large object to the PHP variable representing the stream by calling the db2_bind_param function. The third argument to this function is a string representing the name of

the PHP variable that holds the path and name of the file. The fourth argument is DB2_PARAM_FILE, which tells the ibm_db2 extension to write the data into a file.

4. Issue the SQL statement by calling the `db2_execute` function.
5. Retrieve the next row in the result set by calling an ibm_db2 fetch function (for example, `db2_fetch_object`).

   For more information about the ibm_db2 API, see http://www.php.net/docs.php.

Example

Fetch a large object from the database.

```
$stmt = db2_prepare($conn, "SELECT name, picture FROM animal_pictures");
$picture = fopen("/opt/albums/spook/grooming.jpg", "wb");
$rc = db2_bind_param($stmt, 1, "nickname", DB2_CHAR, 32);
$rc = db2_bind_param($stmt, 2, "picture", DB2_PARAM_FILE);
$rc = db2_execute($stmt);
$rc = db2_fetch_object($stmt);
```

Calling stored procedures in PHP (ibm_db2)

To call a stored procedure from a PHP application, you prepare and execute an SQL CALL statement. The procedure that you call can include input parameters (IN), output parameters (OUT), and input and output parameters (INOUT).

Before you begin

Obtain a connection resource by calling one of the connection functions in the `ibm_db2` API. Refer to "Connecting to an IBM data server database in PHP (ibm_db2)" on page 17.

Procedure

To call a stored procedure:

1. Call the `db2_prepare` function, passing the listed arguments:

   connection
   A valid database connection resource returned from `db2_connect` or `db2_pconnect`.

   statement
   A string that contains the SQL CALL statement, including parameter markers (?) for any input or output parameters

   options
   Optional: A associative array that specifies the type of cursor to return for result sets. You can use this parameter to request a scrollable cursor on database servers that support this type of cursor. By default, a forward-only cursor is returned.
2. For each parameter marker in the CALL statement, call the `db2_bind_param` function, passing the listed arguments:

   stmt
   The prepared statement returned by the call to the `db2_prepare` function.

   parameter-number
   An integer that represents the position of the parameter marker in the SQL statement.

   variable-name
   The name of the PHP variable to bind to the parameter specified by *parameter-number*.

   parameter-type
   A constant that specifies whether to bind the PHP variable to the SQL parameter as an input parameter (DB2_PARAM_IN), an output parameter (DB2_PARAM_OUT), or a parameter that accepts input and returns output (DB2_PARAM_INOUT).

   This step binds each parameter marker to the name of a PHP variable that will hold the output.
3. Call the `db2_execute` function, passing the prepared statement as an argument.

For more information about the ibm_db2 API, see http://www.php.net/docs.php.

Example

Prepare and execute an SQL CALL statement.

```
$sql = 'CALL match_animal(?, ?)';
$stmt = db2_prepare($conn, $sql);

$second_name = "Rickety Ride";
$weight = 0;

db2_bind_param($stmt, 1, "second_name", DB2_PARAM_INOUT);
db2_bind_param($stmt, 2, "weight", DB2_PARAM_OUT);

print "Values of bound parameters _before_ CALL:\n";
print "  1: {$second_name} 2: {$weight}\n";

db2_execute($stmt);

print "Values of bound parameters _after_ CALL:\n";
print "  1: {$second_name} 2: {$weight}\n";
```

What to do next

If the procedure call returns one or more result sets, you can begin fetching rows from the statement resource.

Retrieving multiple result sets from a stored procedure in PHP (ibm_db2)
When a single call to a stored procedure returns more than one result set, you can use the db2_next_result function of the ibm_db2 API to retrieve the result sets.

Before you begin

You must have a statement resource returned by the db2_exec or db2_execute function that has multiple result sets.

Procedure

To retrieve multiple result sets:

1. Fetch rows from the first result set returned from the procedure by calling one of the ibm_db2 fetch functions, passing the statement resource as an argument. (The first result set that is returned from the procedure is associated with the statement resource.)

Table 6. ibm_db2 fetch functions

Function	Description
	Returns an array,. The columns are 0-indexed by columnindexed. position, representing a row in a result set
	Returns an array,. indexed by column name, representing a row in a result set
	Returns an array, indexed by both column name and position, representing a row in a result set

Table 6. ibm_db2 fetch functions (continued)	
Function	**Description**
	Sets the result set. Use this function to pointer to the next rowiterate through a result or requested rowset.
	Returns an object with. The properties of the properties representingobject map to the names columns in the fetchedof the columns in the rowresult set.

2. Retrieve the subsequent result sets by passing the original statement resource as the first argument to the db2_next_result function. You can fetch rows from the statement resource until no more rows are available in the result set.

   The db2_next_result function returns False when no more result sets are available or if the procedure did not return a result set.

   For more information about the ibm_db2 API, see http://www.php.net/docs.php.

Example

Retrieve multiple result sets from a stored procedure.

```
$stmt = db2_exec($conn, 'CALL multiResults()');

print "Fetching first result set\n";
while ($row = db2_fetch_array($stmt)) {
    // work with row
}

print "\nFetching second result set\n";
$result_2 = db2_next_result($stmt);
if ($result_2) {
  while ($row = db2_fetch_array($result_2)) {
    // work with row
  }
}

print "\nFetching third result set\n";
$result_3 = db2_next_result($stmt);
if ($result_3) {
  while ($row = db2_fetch_array($result_3)) {
    // work with row
  }
}
```

What to do next

When you are ready to close the connection to the database, call the db2_close function. If you attempt to close a persistent connection that you created by using db2_pconnect, the close request returns TRUE, and the persistent IBM data server client connection remains available for the next caller.

Commit modes in PHP applications (ibm_db2)

You can control how groups of SQL statements are committed by specifying a commit mode for a connection resource. The ibm_db2 extension supports two commit modes: autocommit and manual commit.

You must use a regular connection resource returned by the db2_connect function to control database transactions in PHP. Persistent connections always use autocommit mode.

autocommit mode

In autocommit mode, each SQL statement is a complete transaction, which is automatically committed. Autocommit mode helps prevent locking escalation issues that can impede the

performance of highly scalable Web applications. By default, the ibm_db2 extension opens every connection in autocommit mode.

You can turn on autocommit mode after disabling it by calling db2_autocommit($*conn*, DB2_AUTOCOMMIT_ON), where *conn* is a valid connection resource.

Calling the db2_autocommit function might affect the performance of your PHP scripts because it requires additional communication between PHP and the database management system.

manual commit mode

In manual commit mode, the transaction ends when you call the db2_commit or db2_rollback function. This means that all statements executed on the same connection between the start of a transaction and the call to the commit or rollback function are treated as a single transaction.

Manual commit mode is useful if you might have to roll back a transaction that contains one or more SQL statements. If you issue SQL statements in a transaction, and the script ends without explicitly committing or rolling back the transaction, the ibm_db2 extension automatically rolls back any work performed in the transaction.

You can turn off autocommit mode when you create a database connection by using the "AUTOCOMMIT" => DB2_AUTOCOMMIT_OFF setting in the db2_connect options array. You can also turn off autocommit mode for an existing connection resource by calling db2_autocommit($*conn*, DB2_AUTOCOMMIT_OFF), where *conn* is a valid connection resource.

For more information about the ibm_db2 API, see http://www.php.net/docs.php.

Examples

End the transaction when db2_commit or db2_rollback is called.

```
$conn = db2_connect('SAMPLE', 'db2inst1', 'ibmdb2', array(
  'AUTOCOMMIT' => DB2_AUTOCOMMIT_ON));

// Issue one or more SQL statements within the transaction
$result = db2_exec($conn, 'DELETE FROM TABLE employee');
if ($result === FALSE) {
  print '<p>Unable to complete transaction!</p>';
  db2_rollback($conn);
}
else {
  print '<p>Successfully completed transaction!</p>';
  db2_commit($conn);
}
```

Error-handling functions in PHP applications (ibm_db2)

Sometimes errors happen when you attempt to connect to a database or issue an SQL statement. The username or password might be incorrect, a table or column name might be misspelled, or the SQL statement might be invalid. The ibm_db2 API provides error-handling functions to help you recover gracefully from the error situations.

Connection errors

Use one of the listed functions to retrieve diagnostic information if a connection attempt fails.

Table 7. ibm_db2 functions for handling connection errors

Function	Description
db2_conn_error	Retrieves the SQLSTATE returned by the last connection attempt
db2_conn_errormsg	Retrieves a descriptive error message appropriate for an application error log

SQL errors

Use one of the listed functions to retrieve diagnostic information if an attempt to prepare or execute an SQL statement or to fetch a result from a result set fails.

Table 8. ibm_db2 functions for handling SQL errors

Function	Description
db2_stmt_error	Retrieves the SQLSTATE returned by the last attempt to prepare or execute an SQL statement or to fetch a result from a result set
db2_stmt_errormsg	Retrieves a descriptive error message appropriate for an application error log

For more information about the ibm_db2 API, see http://www.php.net/docs.php.

Tip: To avoid security vulnerabilities that might result from directly displaying the raw SQLSTATE returned from the database, and to offer a better overall user experience in your web application, use a switch structure to recover from known error states or return custom error messages. For example:

```
switch($this->state):
    case '22001':
        // More data than allowed for the defined column
        $message = "You entered too many characters for this value.";
        break;
```

Examples

Example 1: Handle connection errors

```
$connection = db2_connect($database, $user, $password);
if (!$connection) {
    $this->state = db2_conn_error();
    return false;
}
```

Example 2: Handle SQL errors

```
$stmt = db2_prepare($connection, "DELETE FROM employee
WHERE firstnme = ?");
if (!$stmt) {
    $this->state = db2_stmt_error();
    return false;
}
```

Example 3: Handle SQL errors that result from executing prepared statements

```
$success = db2_execute($stmt, array('Dan');
if (!$success) {
    $this->state = db2_stmt_error($stmt);
    return $false;
}
```

Database metadata retrieval functions in PHP (ibm_db2)

You can use functions in the ibm_db2 API to retrieve metadata for databases served by Db2, IBM Cloudscape, and, through Db2 Connect, Db2 for z/OS® and Db2 for i.

Some classes of applications, such as administration interfaces, must dynamically reflect the structure and SQL objects contained in arbitrary databases. One approach to retrieving metadata about a database is to issue SELECT statements directly against the system catalog tables; however, the schema of the system catalog tables might change between versions of Db2, or the schema of the system catalog tables on Db2 might differ from the schema of the system catalog tables on Db2 for z/OS. Rather than laboriously maintaining these differences in your application code, you can use PHP functions available in the ibm_db2 extension to retrieve database metadata.

Before calling these functions, you must set up the PHP environment and have a connection resource returned by the db2_connect or db2_pconnect function.

Important: Calling metadata functions uses a significant amount of space. If possible, cache the results of your calls for use in subsequent calls.

Table 9. ibm_db2 metadata retrieval functions

Function	Description
db2_client_info	Returns a read-only object with information about the IBM data server client
db2_column_privileges	Returns a result set listing the columns and associated privileges for a table
db2_columns	Returns a result set listing the columns and associated metadata for a table
db2_foreign_keys	Returns a result set listing the foreign keys for a table
db2_primary_keys	Returns a result set listing the primary keys for a table
db2_procedure_columns	Returns a result set listing the parameters for one or more stored procedures
db2_procedures	Returns a result set listing the stored procedures registered in the database
db2_server_info	Returns a read-only object with information about the database management system software and configuration
db2_special_columns	Returns a result set listing the unique row identifiers for a table
db2_statistics	Returns a result set listing the indexes and statistics for a table
db2_table_privileges	Returns a result set listing tables and their associated privileges in the database

Most of the ibm_db2 database metadata retrieval functions return result sets with columns defined for each function. To retrieve rows from the result sets, use the ibm_db2 functions that are available for this purpose.

The db2_client_info and db2_server_info functions directly return a single object with read-only properties. You can use the properties of these objects to create an application that behaves differently depending on the database management system to which it connects. For example, rather than encoding a limit of the lowest common denominator for all possible database management systems, a Web-based database administration application built on the ibm_db2 extension could use the db2_server_info()->MAX_COL_NAME_LEN property to dynamically display text fields for naming columns with maximum lengths that correspond to the maximum length of column names on the database management system to which it is connected.

For more information about the ibm_db2 API, see http://www.php.net/docs.php.

Examples

Example 1: Display a list of columns and associated privileges for a table

```
<?php
$conn = db2_connect('sample', 'db2inst1', 'ibmdb2');
```

```
if ($conn) {
    $stmt = db2_column_privileges($conn, NULL, NULL, 'DEPARTMENT');
    $row = db2_fetch_array($stmt);
    print $row[2] . "\n";
    print $row[3] . "\n";
    print $row[7];
    db2_close($conn);
}
else {
    echo db2_conn_errormsg();
    printf("Connection failed\n\n");
}
?>
```

Example 2: Display a list of primary keys for a table

```
<?php
$conn = db2_connect('sample', 'db2inst1', 'ibmdb2');

if ($conn) {
    $stmt = db2_primary_keys($conn, NULL, NULL, 'DEPARTMENT');
    while ($row = db2_fetch_array($stmt)) {
        echo "TABLE_NAME:\t" . $row[2] . "\n";
        echo "COLUMN_NAME:\t" . $row[3] . "\n";
        echo "KEY_SEQ:\t" . $row[4] . "\n";
    }

    db2_close($conn);
}
else {
    echo db2_conn_errormsg();
    printf("Connection failed\n\n");
}
?>
```

Example 3: Display a list of parameters for one or more stored procedures

```
<?php
$conn = db2_connect('sample', 'db2inst1', 'ibmdb2');

if ($conn) {
    $stmt = db2_procedures($conn, NULL, 'SYS%', '%%');

    $row = db2_fetch_assoc($stmt);
    var_dump($row);

    db2_close($conn);
}
else {
    echo "Connection failed.\n";
}
?>
```

Example 4: Display a list of the indexes and statistics for a table

```
<?php
$conn = db2_connect('sample', 'db2inst1', 'ibmdb2');

if ($conn) {
    echo "Test DEPARTMENT table:\n";
    $result = db2_statistics($conn, NULL, NULL, "EMPLOYEE", 1);
    while ($row = db2_fetch_assoc($result)) {
        var_dump($row);
    }

    echo "Test non-existent table:\n";
    $result = db2_statistics($conn,NULL,NULL,"NON_EXISTENT_TABLE",1);
    $row = db2_fetch_array($result);
    if ($row) {
        echo "Non-Empty\n";
    } else {
        echo "Empty\n";
    }

    db2_close($conn);
}
else {
```

```
    echo 'no connection: ' . db2_conn_errormsg();
}
?>
```

Example 5: Display a list of tables and their associated privileges in the database

```
<?php
$conn = db2_connect('sample', 'db2inst1', 'ibmdb2');

if ($conn) {
    $stmt = db2_table_privileges($conn, NULL, "%%", "DEPARTMENT");
    while ($row = db2_fetch_assoc($stmt)) {
        var_dump($row);
    }
    db2_close($conn);
}
else {
    echo db2_conn_errormsg();
    printf("Connection failed\n\n");
}
?>
```

Application development in PHP (PDO)

The PDO_IBM extension provides a variety of useful PHP functions for accessing and manipulating data through the standard object-oriented database interface introduced in PHP 5.1. The extension includes functions for connecting to a database, executing and preparing SQL statements, fetching rows from result sets, managing transactions, calling stored procedures, handling errors, and retrieving metadata.

Connecting to an IBM data server database with PHP (PDO)

Before you can issue SQL statements to create, update, delete, or retrieve data, you must connect to a database. You can use the PHP Data Objects (PDO) interface for PHP to connect to an IBM data server database through either a cataloged connection or a direct TCP/IP connection. To improve performance, you can also create a persistent connection.

Before you begin

You must set up the PHP 5.1 (or later) environment on your system and enable the PDO and PDO_IBM extensions.

About this task

This procedure returns a connection object to an IBM data server database. This connection stays open until you set the PDO object to NULL, or the PHP script finishes.

Procedure

To connect to an IBM data server database:

1. Create a connection to the database by calling the PDO constructor within a `try{}` block. Pass a *DSN* value that specifies `ibm:` for the PDO_IBM extension, followed by either a cataloged database name or a complete database connection string for a direct TCP/IP connection.
   - (Windows): By default, the PDO_IBM extension uses connection pooling to minimize connection resources and improve connection performance.
   - (Linux and UNIX): To create a persistent connection, pass `array(PDO::ATTR_PERSISTENT => TRUE)` as the *driver_options* (fourth) argument to the PDO constructor.
2. Optional: Set error handling options for the PDO connection in the fourth argument to the PDO constructor:
   - By default, PDO sets an error message that can be retrieved through `PDO::errorInfo()` and an SQLCODE that can be retrieved through `PDO::errorCode()` when any error occurs; to request this mode explicitly, set `PDO::ATTR_ERRMODE => PDO::ERRMODE_SILENT`

- To issue a PHP E_WARNING when any error occurs, in addition to setting the error message and SQLCODE, set PDO::ATTR_ERRMODE => PDO::ERRMODE_WARNING
- To throw a PHP exception when any error occurs, set PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION

3. Catch any exception thrown by the try{} block in a corresponding catch {} block.

   For more information about the PDO API, see http://php.net/manual/en/book.pdo.php.

Example

Connect to an IBM data server database over a persistent connection.

```
try {
  $connection = new PDO("ibm:SAMPLE", "db2inst1", "ibmdb2", array(
    PDO::ATTR_PERSISTENT => TRUE,
    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION)
  );
}
catch (Exception $e) {
  echo($e->getMessage());
}
```

What to do next

Next, you prepare and execute SQL statements.

Executing SQL statements in PHP (PDO)
After connecting to a database, use methods available in the PDO API to prepare and execute SQL statements. The SQL statements can contain static text or parameter markers that represent variable input.

Executing a single SQL statement in PHP (PDO)
To prepare and execute a single SQL statement that accepts no input parameters, use the PDO::exec or PDO::query method. Use the PDO::exec method to execute a statement that returns no result set. Use the PDO::query method to execute a statement that returns one or more result sets.

Before you begin

Important: To avoid the security threat of SQL injection attacks, use the PDO::exec or PDO::query method only to execute SQL statements composed of static strings. Interpolation of PHP variables representing user input into the SQL statement can expose your application to SQL injection attacks.

Obtain a connection object by calling the PDO constructor.

Procedure

To prepare and execute a single SQL statement that accepts no input parameters, call one of the listed methods:

- To execute an SQL statement that returns no result set, call the PDO::exec method on the PDO connection object, passing in a string that contains the SQL statement. For example, a typical use of PDO::exec is to set the default schema for your application in a common include file or base class.

  If the SQL statement succeeds (successfully inserts, modifies, or deletes rows), the PDO::exec method returns an integer value representing the number of rows that were inserted, modified, or deleted.

  To determine if the PDO::exec method failed (returned FALSE or 0), use the === operator to strictly test the returned value against FALSE.
- To execute an SQL statement that returns one or more result sets, call the PDO::query method on the PDO connection object, passing in a string that contains the SQL statement. For example, you might want to call this method to execute a static SELECT statement.

If the method call succeeds, it returns a PDOStatement resource that you can use in subsequent method calls.

If the method call fails (returns FALSE), you can use the `PDO::errorCode` and `PDO::errorInfo` method to retrieve diagnostic information about the error.

For more information about the PDO API, see http://php.net/manual/en/book.pdo.php.

Example

Example 1: Call the `PDO::exec` method to set the default schema for your application

```
$conn = new PDO('ibm:SAMPLE', 'db2inst1', 'ibmdb2');
$result = $conn->exec('SET SCHEMA myapp');
if ($result === FALSE) {
  print "Failed to set schema: " . $conn->errorMsg();
}
```

Example 2: Call the `PDO::query` method to issue an SQL SELECT statement

```
$conn = new PDO('ibm:SAMPLE', 'db2inst1', 'ibmdb2');
$result = $conn->query('SELECT firstnme, lastname FROM employee');
if (!$result) {
  print "<p>Could not retrieve employee list: " . $conn->errorMsg(). "</p>";
}
while ($row = $result->fetch()) {
  print "<p>Name: {$row[0]} {$row[1]}</p>";
}
```

What to do next

If you called the `PDO::query` method to create a PDOStatement object, you can begin retrieving rows from the object by calling the `PDOStatement::fetch` or `PDOStatement::fetchAll` method.

Preparing and executing SQL statements in PHP (PDO)

To prepare and execute an SQL statement that includes variable input, use the `PDO::prepare`, `PDOStatement::bindParam`, and `PDOStatement::execute` methods. Preparing a statement improves performance because the database server creates an optimized access plan for data retrieval that it can reuse if the statement is executed again.

Before you begin

Obtain a connection object by calling the PDO constructor. Refer to "Connecting to an IBM data server database with PHP (PDO)" on page 32.

Procedure

To prepare and execute an SQL statement that includes parameter markers:

1. Call the `PDO::prepare` method, passing the listed arguments:

   statement
   A string that contains the SQL statement, including question marks (?) or named variables (`:name`) as parameter markers for any column or predicate values that require variable input. You can only use parameter markers as a place holder for column or predicate values. The SQL compiler is unable to create an access plan for a statement that uses parameter markers in place of column names, table names, or other SQL identifiers. You cannot use both question mark (?) parameter markers and named parameter markers (`:name`) in the same SQL statement.

   driver_options
   Optional: An array that contains statement options:

   PDO::ATTR_CURSOR
   This option sets the type of cursor that PDO returns for result sets. By default, PDO returns a forward-only cursor (PDO::CURSOR_FWDONLY), which returns the next row in a result set for

every call to PDOStatement::fetch(). You can set this parameter to PDO::CURSOR_SCROLL to request a scrollable cursor.

If the function call succeeds, it returns a PDOStatement object that you can use in subsequent method calls that are related to this query.

If the function call fails (returns False), you can use the PDO::errorCode or PDO::errorInfo method to retrieve diagnostic information about the error.

2. Optional: For each parameter marker in the SQL string, call the PDOStatement::bindParam method, passing the listed arguments. Binding input values to parameter markers ensures that each input value is treated as a single parameter, which prevents SQL injection attacks against your application.

   parameter
   A parameter identifier. For question mark parameter markers (?), this is an integer that represents the 1-indexed position of the parameter in the SQL statement. For named parameter markers (:name), this is a string that represents the parameter name.

   variable
   The value to use in place of the parameter marker

3. Call the PDOStatement::execute method, optionally passing an array that contains the values to use in place of the parameter markers, either in order for question mark parameter markers, or as a :name => value associative array for named parameter markers.

   For more information about the PDO API, see http://php.net/manual/en/book.pdo.php.

Example

Prepare and execute a statement that includes variable input.

```
$sql = "SELECT firstnme, lastname FROM employee WHERE bonus > ? AND bonus < ?";
$stmt = $conn->prepare($sql);
if (!$stmt) {
  // Handle errors
}

// Explicitly bind parameters
$stmt->bindParam(1, $_POST['lower']);
$stmt->bindParam(2, $_POST['upper']);

$stmt->execute($stmt);

// Invoke statement again using dynamically bound parameters
$stmt->execute($stmt, array($_POST['lower'], $_POST['upper']));
```

What to do next

If the SQL statement returns one or more result sets, you can begin fetching rows from the statement resource by calling the PDOStatement::fetch or PDOStatement::fetchAll method.

Inserting large objects in PHP (PDO)

When you insert a large object into the database, rather than loading all of the data for a large object into a PHP string and passing it to the IBM data server database through an INSERT statement, you can insert large objects directly from a file on your PHP server.

Before you begin

Obtain a connection object by calling the PDO constructor.

Procedure

To insert a large object into the database directly from a file:

1. Call the PDO::prepare method to create a PDOStatement object from an INSERT statement with a parameter marker that represents the large object column.

2. Create a PHP variable that represents a stream (for example, by assigning the value returned by the `fopen` function to variable).
3. Call the `PDOStatement::bindParam` method, passing the listed arguments to bind the parameter marker to the PHP variable that represents the stream of data for the large object:

   parameter
   A parameter identifier. For question mark parameter markers (?), this is an integer that represents the 1-indexed position of the parameter in the SQL statement. For named parameter markers (`:name`), this is a string that represents the parameter name.

   variable
   The value to use in place of the parameter marker

   data_type
   The PHP constant, `PDO::PARAM_LOB`, which tells the PDO extension to retrieve the data from a file.
4. Call the `PDOStatement::execute` method to issue the INSERT statement.

Example

Insert a large object into the database.

```
$stmt = $conn->prepare("INSERT INTO animal_pictures(picture) VALUES (?)");
$picture = fopen("/opt/albums/spook/grooming.jpg", "rb");
$stmt->bindParam(1, $picture, PDO::PARAM_LOB);
$stmt->execute();
```

Reading query result sets

Fetching rows or columns from result sets in PHP (PDO)
After executing a statement that returns one or more result sets, use one of the methods available in the PDO API to iterate through the returned rows. The PDO API also provides methods to fetch a single column from one or more rows in the result set.

Before you begin

You must have a statement resource returned by either the `PDO::query` or `PDOStatement::execute` method that has one or more associated result sets.

Procedure

To fetch data from a result set:

1. Fetch data from a result set by calling one of the fetch methods:
   - To return a single row from a result set as an array or object, call the `PDOStatement::fetch` method.
   - To return all of the rows from the result set as an array of arrays or objects, call the `PDOStatement::fetchAll` method.

   By default, PDO returns each row as an array indexed by the column name and 0-indexed column position in the row. To request a different return style, specify one of the `PDO::FETCH_*` constants as the first parameter when you call the `PDOStatement::fetch` method:

   `PDO::FETCH_ASSOC`
   Returns an array indexed by column name as returned in your result set.

   `PDO::FETCH_BOTH` (default)
   Returns an array indexed by both column name and 0-indexed column number as returned in your result set

   `PDO::FETCH_BOUND`
   Returns TRUE and assigns the values of the columns in your result set to the PHP variables to which they were bound with the `PDOStatement::bindParam` method.

PDO::FETCH_CLASS
Returns a new instance of the requested class, mapping the columns of the result set to named properties in the class.

PDO::FETCH_INTO
Updates an existing instance of the requested class, mapping the columns of the result set to named properties in the class.

PDO::FETCH_LAZY
Combines PDO::FETCH_BOTH and PDO::FETCH_OBJ, creating the object variable names as they are accessed.

PDO::FETCH_NUM
Returns an array indexed by column number as returned in your result set, starting at column 0.

PDO::FETCH_OBJ
Returns an anonymous object with property names that correspond to the column names returned in your result set.

If you requested a scrollable cursor when you called the PDO::query or PDOStatement::execute method, you can pass the listed optional parameters that control which rows are returned to the caller:

- One of the PDO::FETCH_ORI_* constants that represents the fetch orientation of the fetch request:

  PDO::FETCH_ORI_NEXT (default)
  Fetches the next row in the result set.

  PDO::FETCH_ORI_PRIOR
  Fetches the previous row in the result set.

  PDO::FETCH_ORI_FIRST
  Fetches the first row in the result set.

  PDO::FETCH_ORI_LAST
  Fetches the last row in the result set.

  PDO::FETCH_ORI_ABS
  Fetches the absolute row in the result set. Requires a positive integer as the third argument to the PDOStatement::fetch method.

  PDO::FETCH_ORI_REL
  Fetches the relative row in the result set. Requires a positive or negative integer as the third argument to the PDOStatement::fetch method.

- An integer requesting the absolute or relative row in the result set, corresponding to the fetch orientation requested in the second argument to the PDOStatement::fetch method.

2. Optional: Fetch a single column from one or more rows in a result set by calling one of the listed methods:

   - To return a single column from a single row in the result set:

     Call the PDOStatement::fetchColumn method, specifying the column you want to retrieve as the first argument of the method. Column numbers start at 0. If you do not specify a column, the PDOStatement::fetchColumn returns the first column in the row.

   - To return an array that contains a single column from all of the remaining rows in the result set:

     Call the PDOStatement::fetchAll method, passing the PDO::FETCH_COLUMN constant as the first argument, and the column you want to retrieve as the second argument. Column numbers start at 0. If you do not specify a column, calling PDOStatement::fetchAll(PDO::FETCH_COLUMN) returns the first column in the row.

   For more information about the PDO API, see http://php.net/manual/en/book.pdo.php.

Example

Return an array indexed by column number.

```
$stmt = $conn->query("SELECT firstnme, lastname FROM employee");
while ($row = $stmt->fetch(PDO::FETCH_NUM)) {
  print "Name: <p>{$row[0] $row[1]}</p>";
}
```

What to do next

When you are ready to close the connection to the database, set the PDO object to NULL. The connection closes automatically when the PHP script finishes.

Fetching large objects in PHP (PDO)

When you fetch a large object from a result set, rather than treating the large object as a PHP string, you can save system resources by fetching large objects directly into a file on your PHP server.

Before you begin

Obtain a connection object by calling the PDO constructor.

Procedure

To fetch a large object from the database directly into a file:

1. Create a PHP variable representing a stream. For example, assign the return value from a call to the `fopen` function to a variable.
2. Create a PDOStatement object from an SQL statement by calling the `PDO::prepare` method.
3. Bind the output column for the large object to the PHP variable representing the stream by calling the `PDOStatement::bindColumn` method. The second argument is a string representing the name of the PHP variable that holds the path and name of the file. The third argument is a PHP constant, PDO::PARAM_LOB, which tells the PDO extension to write the data into a file. You must call the `PDOStatement::bindColumn` method to assign a different PHP variable for every column in the result set.
4. Issue the SQL statement by calling the `PDOStatement::execute` method.
5. Call `PDOStatement::fetch(PDO::FETCH_BOUND)` to retrieve the next row in the result set, binding the column output to the PHP variables that you associated when you called the `PDOStatement::bindColumn` method.

Example

Fetch a large object from the database directly into a file.

```
$stmt = $conn->prepare("SELECT name, picture FROM animal_pictures");
$picture = fopen("/opt/albums/spook/grooming.jpg", "wb");
$stmt->bindColumn('NAME', $nickname, PDO::PARAM_STR, 32);
$stmt->bindColumn('PICTURE', $picture, PDO::PARAM_LOB);
$stmt->execute();
$stmt->fetch(PDO::FETCH_BOUND);
```

Calling stored procedures in PHP (PDO)

To call a stored procedure from a PHP application, you execute an SQL CALL statement. The procedure that you call can include input parameters (IN), output parameters (OUT), and input and output parameters (INOUT).

Before you begin

Obtain a connection object by calling the PDO constructor.

About this task

This procedure prepares and executes an SQL CALL statement. For more information, also see the topic about preparing and executing SQL statements.

Procedure

To call a stored procedure:

1. Call the PDO::prepare method to prepare a CALL statement with parameter markers that represent the OUT and INOUT parameters.
2. For each parameter marker in the CALL statement, call the PDOStatement::bindParam method to bind each parameter marker to the name of the PHP variable that will hold the output value of the parameter after the CALL statement has been issued. For INOUT parameters, the value of the PHP variable is passed as the input value of the parameter when the CALL statement is issued.
   a) Set the third parameter, *data_type*, to one of the PDO::PARAM_* constants that specifies the type of data being bound:

      PDO::PARAM_NULL
      Represents the SQL NULL data type.

      PDO::PARAM_INT
      Represents SQL integer types.

      PDO::PARAM_LOB
      Represents SQL large object types.

      PDO::PARAM_STR
      Represents SQL character data types.

      For an INOUT parameter, use the bitwise OR operator to append PDO::PARAM_INPUT_OUTPUT to the type of data being bound.

   b) Set the fourth parameter, *length*, to the maximum expected length of the output value.
3. Call the PDOStatement::execute method, passing the prepared statement as an argument.

   For more information about the PDO API, see http://php.net/manual/en/book.pdo.php.

Example

Prepare and execute an SQL CALL statement.

```
$sql = 'CALL match_animal(?, ?)';
$stmt = $conn->prepare($sql);

$second_name = "Rickety Ride";
$weight = 0;

$stmt->bindParam(1, $second_name, PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT, 32);
$stmt->bindParam(2, $weight, PDO::PARAM_INT, 10);

print "Values of bound parameters _before_ CALL:\n";
print "  1: {$second_name} 2: {$weight}\n";

$stmt->execute();

print "Values of bound parameters _after_ CALL:\n";
print "  1: {$second_name} 2: {$weight}\n";
```

Retrieving multiple result sets from a stored procedure in PHP (PDO)

When a single call to a stored procedure returns more than one result set, you can use the PDOStatement::nextRow method of the PDO API to retrieve the result sets.

Before you begin

You must have a PDOStatement object returned by calling a stored procedure with the `PDO::query` or `PDOStatement::execute` method.

Procedure

To retrieve multiple result sets:

1. Fetch rows from the first result set returned from the procedure by calling one of the PDO fetch methods. (The first result set that is returned from the procedure is associated with the PDOStatement object returned by the CALL statement.)
   - To return a single row from a result set as an array or object, call the `PDOStatement::fetch` method.
   - To return all of the rows from the result set as an array of arrays or objects, call the `PDOStatement::fetchAll` method.

   Fetch rows from the PDOStatement object until no more rows are available in the first result set.
2. Retrieve the subsequent result sets by calling the `PDOStatement::nextRowset` method to return the next result set. You can fetch rows from the PDOStatement object until no more rows are available in the result set.

   The `PDOStatement::nextRowset` method returns False when no more result sets are available or the procedure did not return a result set.

   For more information about the PDO API, see http://php.net/manual/en/book.pdo.php.

Example

Retrieve multiple result sets from a stored procedure.

```
$sql = 'CALL multiple_results()';
$stmt = $conn->query($sql);
do {
   $rows = $stmt->fetchAll(PDO::FETCH_NUM);
   if ($rows) {
       print_r($rows);
   }
} while ($stmt->nextRowset());
```

What to do next

When you are ready to close the connection to the database, set the PDO object to NULL. The connection closes automatically when the PHP script finishes.

Commit modes in PHP (PDO)
You can control how groups of SQL statements are committed by specifying a commit mode for a connection resource. The PDO extension supports two commit modes: autocommit and manual commit.

autocommit mode
In autocommit mode, each SQL statement is a complete transaction, which is automatically committed. Autocommit mode helps prevent locking escalation issues that can impede the performance of highly scalable Web applications. By default, the PDO extension opens every connection in autocommit mode.

manual commit mode
In manual commit mode, the transaction begins when you call the `PDO::beginTransaction` method, and it ends when you call either the `PDO::commit` or `PDO::rollBack` method. This means that any statements executed (on the same connection) between the start of a transaction and the call to the commit or rollback method are treated as a single transaction.

Manual commit mode is useful if you might have to roll back a transaction that contains one or more SQL statements. If you issue SQL statements in a transaction and the script ends without explicitly committing or rolling back the transaction, PDO automatically rolls back any work performed in the transaction.

After you commit or rollback the transaction, PDO automatically resets the database connection to autocommit mode.

For more information about the PDO API, see http://php.net/manual/en/book.pdo.php.

Examples

End the transaction when `PDO::commit` or `PDO::rollBack` is called.

```
$conn = new PDO('ibm:SAMPLE', 'db2inst1', 'ibmdb2', array(
  PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
  // PDO::ERRMODE_EXCEPTION means an SQL error throws an exception
try {
  // Issue these SQL statements in a transaction within a try{} block
  $conn->beginTransaction();

  // One or more SQL statements

  $conn->commit();
}
catch (Exception $e) {
  // If something raised an exception in our transaction block of statements,
  // roll back any work performed in the transaction
  print '<p>Unable to complete transaction!</p>';
  $conn->rollBack();
}
```

Handling errors and warnings in PHP (PDO)

Sometimes errors happen when you attempt to connect to a database or issue an SQL statement. The password for your connection might be incorrect, a table you referred to in a SELECT statement might not exist, or the SQL statement might be invalid. PDO provides error-handling methods to help you recover gracefully from the error situations.

Before you begin

You must set up the PHP environment on your system and enable the PDO and PDO_IBM extensions.

About this task

PDO gives you the option of handling errors as warnings, errors, or exceptions. However, when you create a new PDO connection object, PDO always throws a PDOException object if an error occurs. If you do not catch the exception, PHP prints a backtrace of the error information that might expose your database connection credentials, including your user name and password.

This procedure catches a PDOException object and handles the associated error.

Procedure

1. To catch a PDOException object and handle the associated error:
   a) Wrap the call to the PDO constructor in a `try` block.
   b) Following the `try` block, include a `catch` block that catches the PDOException object.
   c) Retrieve the error message associated with the error by invoking the `Exception::getMessage` method on the PDOException object.
2. To retrieve the SQLSTATE associated with a PDO or PDOStatement object, invoke the `errorCode` method on the object.
3. To retrieve an array of error information associated with a PDO or PDOStatement object, invoke the `errorInfo` method on the object. The array contains a string representing the SQLSTATE as the first element, an integer representing the SQL or CLI error code as the second element, and a string containing the full text error message as the third element.

   For more information about the PDO API, see http://php.net/manual/en/book.pdo.php.

Chapter 4. Developing Python applications

Python, SQLAlchemy, and Django Framework application development for IBM Database servers

Python is a general purpose, high-level scripting language that is well suited for rapid application development. Python emphasizes code readability and supports various programming paradigms, including procedural, object-oriented, aspect-oriented, meta, and functional programming. The Python language is managed by the Python Software Foundation.

The listed extensions are available for accessing IBM Database servers from a Python application:

ibm_db
This API is defined by IBM and provides the best support for advanced features. In addition to issuing SQL queries, calling stored procedures, and using pureXML, you can access metadata information.

ibm_db_dbi
This API implements Python Database API Specification v2.0. Because the ibm_db_dbi API conforms to the specification, it does not offer some of the advanced features that the ibm_db API supports. If you have an application with a driver that supports Python Database API Specification v2.0, you can easily switch to ibm_db. The ibm_db and ibm_db_dbi APIs are packaged together.

ibm_db_sa
This adapter supports SQLAlchemy, which offers a flexible way to access IBM Database servers. SQLAlchemy is a popular open source Python SQL toolkit and object-to-relational mapper (ORM).

ibm_db_django
This adapter provides access to IBM Database servers from Django. Django is a popular web framework used to build high-performing, elegant web applications quickly.

If you want to connect your Python applications to Db2 for IBM i V5R4 and later servers, you must have PTF SI27256 applied to those servers.

Python downloads and related resources

Many resources are available to help you develop Python applications for IBM data servers.

Table 10. Python downloads and related resources

Downloads	
Python [2]	http://www.python.org/download/
SQLAlchemy	http://www.sqlalchemy.org/download.html
Django	http://www.djangoproject.com/download/
ibm_db and ibm_db_dbi extensions (including source code)	http://pypi.python.org/pypi/ibm_db/
	https://github.com/ibmdb/python-ibmdb
ibm_db_sa adapter for SQLAlchemy 0.4	https://github.com/ibmdb/python-ibmdbsa
	http://pypi.python.org/pypi/ibm_db_sa
ibm_db_django adaptor for Django 1.0.x and 1.1	https://github.com/ibmdb/python-ibmdb-django
	http://pypi.python.org/pypi/ibm_db_django
setuptools program	http://pypi.python.org/pypi/setuptools

[2] Includes Windows binaries. Most Linux distributions come with Python already precompiled.

© Copyright IBM Corp. 2016, 2020

Table 10. Python downloads and related resources (continued)	
Downloads	
IBM Data Server Driver Package (DS Driver)	https://www.ibm.com/support/pages/node/387577
API documentation	
ibm_db API documentation	https://github.com/ibmdb/python-ibmdb/wiki/APIs
Python Database API Specification v2.0	http://www.python.org/dev/peps/pep-0249/
SQLAlchemy documentation	
Quick Getting Started Steps for ibm_db_sa	https://github.com/ibmdb/python-ibmdbsa/blob/master/ibm_db_sa/README.md
SQLAlchemy Documentation	http://www.sqlalchemy.org/docs/index.html
Django documentation	
Getting Started steps for ibm_db_django	https://github.com/ibmdb/python-ibmdb-django/blob/master/README.md
Django Documentation	http://www.djangoproject.com
Additional resources	
Python Programming Language website	http://www.python.org/
The Python SQL Toolkit and Object Relational Mapper website	http://www.sqlalchemy.org/

Setting up the Python environment for IBM database servers

Before you can connect to an IBM database server and run SQL statements, you must set up the Python environment by installing the ibm_db (Python) driver and, optionally, the ibm_db_sa (SQLAlchemy) or ibm_db_django (Django) adapter.

Before you begin

Ensure that the following software is installed on your system:

- Python 2.5 or later. For Linux operating systems, you also require the `python2.5-dev` package.
- The **`setuptools`** program or the **`distribute`** program. The **`setuptools`** program is available at http://pypi.python.org/pypi/setuptools, and the **`distribute`** program is available at http://pypi.python.org/pypi/distribute. You can use the **`setuptools`** program or the **`distribute`** program to download, build, install, upgrade, and uninstall Python packages.
- If your Python application will connect to a remote IBM database, one of the following products on the computer where your application will run:
  - The IBM Data Server Client product
  - The IBM Data Server Runtime Client product
  - The IBM Data Server Driver Package product
  - The IBM Data Server Driver for ODBC and CLI product

  If your Python application connects to an IBM database server on the local computer, no additional IBM data server products are required.

Procedure

To set up the Python environment:

1. Using the following method, install the ibm_db Python driver:
   - Install from the remote repository:
     a. If you want to avoid automatic installation of the clidriver and would like to use an existing copy of the driver, you can set **IBM_DB_HOME**.

        Set the **IBM_DB_HOME** environment variable by using the **export** command:

        ```
        $ export IBM_DB_HOME=DB2HOME
        ```

        where *DB2HOME* is the directory where the IBM data server product is installed.

        For example, issue the following command to set the **IBM_DB_HOME** environment variable:

        ```
        $ export IBM_DB_HOME=/home/db2inst1/<dsdriver installation>/clidriver
        ```

     b. Issue the following command:

        ```
        $ pip install ibm_db
        ```

        or

        ```
        $ easy_install ibm_db
        ```

2. Optional: Using the following method, install the ibm_db_sa SQLAlchemy adapter or ibm_db_django Django adapter:
   - Install from the remote repository:
     – To install the SQLAlchemy adapter, issue the following command:

       ```
       $ pip install ibm_db_sa
       ```

       or

       ```
       $ easy_install ibm_db_sa
       ```

     – To install the django adapter, issue the following command:

       ```
       $ pip install ibm_db_django
       ```

       or

       ```
       $ easy_install ibm_db_django
       ```

3. Ensure that the Python driver can access the `libdb2.so` CLI driver file:
   - For 32-bit Linux and UNIX operating systems other than the AIX operating system, set the **LD_LIBRARY_PATH** variable to the *IBM_DB_HOME*`/lib32` directory by issuing the **export** command:

     ```
     $ export LD_LIBRARY_PATH=IBM_DB_HOME/lib32
     ```

   - For 64-bit Linux and UNIX operating systems other than the AIX operating system, set the **LD_LIBRARY_PATH** variable to the *IBM_DB_HOME*`/lib64` directory by issuing the **export** command:

     ```
     $ export LD_LIBRARY_PATH=IBM_DB_HOME/lib64
     ```

   - For a 32-bit AIX operating system, set the **LIBPATH** variable to the*IBM_DB_HOME*`/lib32` directory by issuing the **export** command:

     ```
     $ export LIBPATH=IBM_DB_HOME/lib32
     ```

- For a 64-bit AIX operating system, set the **LIBRARY_PATH** variable to the *IBM_DB_HOME*/lib64 directory by issuing the **export** command:

```
$ export LIBPATH=IBM_DB_HOME/lib64
```

What to do next

Test the ibm_db Python driver, the ibm_db_sa SQLAlchemy adapter, and the ibm_db_django Django adapter connection by using the test applications.

Verifying the Python driver, SQLAlchemy adapter, and Django adapter installation

When the installation of the Python driver and optional adapters are complete, it is a good practice to test the new Python environment to verify that installation is working.

Before you begin

You must have the following software installed on your system:

- Python 2.5 or later. For Linux operating systems, you also require the python2.5-dev package.
- If your Python application connects to a remote IBM database, the computer that runs your Python application requires one of the following products:
  - IBM Data Server Client
  - IBM Data Server Runtime Client
  - IBM Data Server Driver Package
  - IBM Data Server Driver for ODBC and CLI
- If your Python application connects to local IBM database, no additional IBM Data Server products are required.
- The Python environment must be configured for the listed driver and adapters:
  - ibm_db Python driver
  - ibm_db_sa SQLAlchemy adapter
  - ibm_db_django Django adapter

Procedure

To verify that your Python installation is successful:

1. Using the **python** command, start the Python interpreter.

```
$ python
```

2. Using the listed code, test the ibm_db Python driver:

```
import ibm_db
ibm_db_conn = ibm_db.connect('database', 'user', 'password')
import ibm_db_dbi
conn = ibm_db_dbi.Connection(ibm_db_conn)
conn.tables('SYSCAT', '%')
```

   You must specify a valid database name (database), user ID (user), and password (password) in the code. Successful connection indicates valid ibm_db Python driver installation.

3. Optional: Using the listed code, test the ibm_db_sa SQLAlchemy adapter:

```
import sqlalchemy
from sqlalchemy import *
import ibm_db_sa.ibm_db_sa
db2 = sqlalchemy.create_engine('ibm_db_sa://user:password@host.name.com:50000/database')
metadata = MetaData()
users = Table('STAFF', metadata,
Column('ID', Integer, primary_key = True),
Column('NAME', String(9), nullable = False),
Column('DEPT', Integer, nullable = False),
```

```
Column('JOB', String(5), nullable = False)
)
```

You must specify a valid database name (database), user ID (user), and password (password) in the `sqlalchemy.create_engine` argument string. Successful connection indicates valid ibm_db_django Django adapter installation.

4. Optional: Using the listed code, test the ibm_db_django Django adapter:
   a. Using the **`django-admin.py startproject`** command, create a new Django project:

```
django-admin.py startproject myproj
```

   b. Using the editor of your choice, edit DATABASES dictionary in the `settings.py` file to configure access to the IBM database server:

```
DATABASES = {
  'default': {
  'ENGINE'   : 'ibm_db_django',
  'NAME'     : 'database',
  'USER'     : 'user',
  'PASSWORD' : 'password',
  'HOST'     : 'localhost',
  'PORT'     : '50000',
  'PCONNECT' :  True,      #Optional property. It is true by default
  }
}
```

   You must specify a valid database name (database), user ID (user), password (password), host name (localhost), and port number (50000) in the `settings.py` file entry.

   c. Using the editor of your choice, add the following tuple of strings in the INSTALLED_APPS section of the `settings.py` file:

```
'django.contrib.flatpages',
'django.contrib.redirects',
'django.contrib.comments',
'django.contrib.admin',
```

   d. Using the `manage.py` application, verify the Django configuration:

```
python manage.py test
```

Application development in Python with ibm_db

The ibm_db API provides a variety of Python functions for accessing and manipulating data in an IBM data server database, including functions for connecting to a database, preparing and issuing SQL statements, fetching rows from result sets, calling stored procedures, committing and rolling back transactions, handling errors, and retrieving metadata.

Connecting to an IBM database server in Python

Before you can run SQL statements to create, update, delete, or retrieve data, you must connect to a database. You can use the ibm_db API to connect to a database through either a cataloged or uncataloged connection. To improve performance, you can also create a persistent connection.

Before you begin

- Setting up the Python environment for IBM Database servers.
- Issue the `import ibm_db` command from your Python script.

Procedure

Call one of the listed functions to establish connection to an IBM database server:

Table 11. ibm_db connection functions	
Function	**Description**
	Creates a nonpersistent. connection
	Creates a persistent. A persistent connection connectionremains open after the initial Python script request, which allows subsequent Python requests to reuse the connection. The subsequent Python connect requests must have an identical set of credentials.

The database value that you pass as an argument to these functions can be either a cataloged database name or a complete database connection string for a direct TCP/IP connection. You can specify optional arguments that control the timing of committing transactions, the case of the column names that are returned, and the cursor type.

If the connection attempt fails, you can retrieve diagnostic information by calling the `ibm_db.conn_error` or `ibm_db.conn_errormsg` function.

For more information about the ibm_db API, see https://github.com/ibmdb/python-ibmdb/wiki/APIs.

Example

Example 1: Connect to a local or cataloged database

```
import ibm_db
conn = ibm_db.connect("database","username","password")
```

Example 2: Connect to an uncataloged database

```
import ibm_db
ibm_db.connect("DATABASE=name;HOSTNAME=host;PORT=60000;PROTOCOL=TCPIP;UID=username;
                PWD=password;", "", "")
```

What to do next

If the connection attempt is successful, you can use the connection resource when you call ibm_db functions that execute SQL statements. Next, you prepare and execute SQL statements.

Executing SQL statements in Python

After connecting to a database, use functions available in the ibm_db API to prepare and execute SQL statements. The SQL statements can contain static text, XQuery expressions, or parameter markers that represent variable input.

Preparing and executing a single SQL statement in Python

To prepare and execute a single SQL statement, use the `ibm_db.exec_immediate` function. To avoid the security threat of SQL injection attacks, use the `ibm_db.exec_immediate` function only to execute SQL statements that are composed of static strings. Interpolation of Python variables representing user input into the SQL statement can expose your application to SQL injection attacks.

Before you begin

Obtain a connection resource by calling one of the connection functions in the ibm_db API. For more information, see "Connecting to an IBM database server in Python" on page 47.

Procedure

To prepare and execute a single SQL statement, call the `ibm_db.exec_immediate` function, passing the listed arguments:

connection
A valid database connection resource that is returned from the `ibm_db.connect` or `ibm_db.pconnect` function.

statement
A string that contains the SQL statement. This string can include an XQuery expression that is called by the XMLQUERY function.

options
Optional: A dictionary that specifies the type of cursor to return for result sets. You can use this parameter to request a scrollable cursor for database servers that support this type of cursor. By default, a forward-only cursor is returned.

If the function call fails (returns `False`), you can use the `ibm_db.stmt_error` or `ibm_db.stmt_errormsg` function to retrieve diagnostic information about the error.

If the function call succeeds, you can use the `ibm_db.num_rows` function to return the number of rows that the SQL statement returned or affected. If the SQL statement returns a result set, you can begin fetching the rows.

For more information about the `ibm_db` API, see https://github.com/ibmdb/python-ibmdb/wiki/APIs.

Example

Example 1: Execute a single SQL statement

```
import ibm_db
conn = ibm_db.connect("database","username","password")
stmt = ibm_db.exec_immediate(conn, "UPDATE employee SET bonus = '1000' WHERE job = 'MANAGER'")
print "Number of affected rows: ", ibm_db.num_rows(stmt)
```

Example 2: Execute an XQuery expression

```
import ibm_db
conn = ibm_db.connect("database","username","password")
if conn:
    sql = "SELECT XMLSERIALIZE(XMLQUERY('for $i in $t/address where $i/city = \"Olathe\" return <zip>
                    {$i/zip/text()}</zip>' passing c.xmlcol as \"t\") AS CLOB(32k)) FROM xml_test c WHERE id = 1"
    stmt = ibm_db.exec_immediate(conn, sql)
    result = ibm_db.fetch_both(stmt)
    while( result ):
        print "Result from XMLSerialize and XMLQuery:", result[0]
        result = ibm_db.fetch_both(stmt)
```

What to do next

If the SQL statement returns one or more result sets, you can begin fetching rows from the statement resource.

Preparing and executing SQL statements with variable input in Python

To prepare and execute an SQL statement that includes variable input, use the `ibm_db.prepare`, `ibm_db.bind_param`, and `ibm_db.execute` functions. Preparing a statement improves performance because the database server creates an optimized access plan for data retrieval that it can reuse if the statement is executed again.

Before you begin

Obtain a connection resource by calling one of the connection functions in the `ibm_db` API. Refer to "Connecting to an IBM database server in Python" on page 47.

Procedure

To prepare and execute an SQL statement that includes parameter markers:

1. Call the `ibm_db.prepare` function, passing the listed arguments:

   connection
   A valid database connection resource that is returned from the `ibm_db.connect` or `ibm_db.pconnect` function.

   statement
   A string that contains the SQL statement, including question marks (?) as parameter markers for column or predicate values that require variable input. This string can include an XQuery expression that is called by the XMLQUERY function.

   options
   Optional: A dictionary that specifies the type of cursor to return for result sets. You can use this parameter to request a scrollable cursor for database servers that support this type of cursor. By default, a forward-only cursor is returned.

   If the function call succeeds, it returns a statement handle resource that you can use in subsequent function calls that are related to the query.

   If the function call fails (returns `False`), you can use the `ibm_db.stmt_error` or `ibm_db.stmt_errormsg` function to retrieve diagnostic information about the error.
2. Optional: For each parameter marker in the SQL string, call the `ibm_db.bind_param` function, passing the listed arguments. Binding input values to parameter markers ensures that each input value is treated as a single parameter, which prevents SQL injection attacks.

   stmt
   The prepared statement that is returned by the call to the `ibm_db.prepare` function.

   parameter-number
   An integer that represents the position of the parameter marker in the SQL statement.

   variable
   The value to use in place of the parameter marker.
3. Call the `ibm_db.execute` function, passing the listed arguments:

   stmt
   A prepared statement that is returned from `ibm_db.prepare`.

   parameters
   A tuple of input parameters that match parameter markers that are contained in the prepared statement.

   For more information about the `ibm_db` API, see https://github.com/ibmdb/python-ibmdb/wiki/APIs.

Example

Prepare and execute a statement that includes variable input.

```
import ibm_db
conn = ibm_db.connect("database","username","password")
sql = "SELECT EMPNO, LASTNAME FROM EMPLOYEE WHERE EMPNO > ? AND EMPNO < ?"
stmt = ibm_db.prepare(conn, sql)
max = 50
min = 0
# Explicitly bind parameters
ibm_db.bind_param(stmt, 1, min)
ibm_db.bind_param(stmt, 2, max)
ibm_db.execute(stmt)
# Process results

# Invoke prepared statement again using dynamically bound parameters
param = max, min,
ibm_db.execute(stmt, param)
```

What to do next

If the SQL statement returns one or more result sets, you can begin fetching rows from the statement resource.

Fetching rows or columns from result sets in Python

After executing a statement that returns one or more result sets, use one of the functions available in the ibm_db API to iterate through the returned rows. If your result set includes columns that contain large data (such as BLOB or CLOB data), you can retrieve the data on a column-by-column basis to avoid large memory usage.

Before you begin

You must have a statement resource that is returned by either the `ibm_db.exec_immediate` or `ibm_db.execute` function that has one or more associated result sets.

Procedure

To fetch data from a result set:

1. Fetch data from a result set by calling one of the fetch functions.

Table 12. ibm_db fetch functions

Function	Description
	Returns a tuple, which is. The columns are 0-indexed by columnindexed. position, representing a row in a result set
	Returns a dictionary,. which is indexed by column name, representing a row in a result set
	Returns a dictionary,. which is indexed by both column name and position, representing a row in a result set
	Sets the result set. Use this function to pointer to the next rowiterate through a result or requested rowset.

These functions accept the listed arguments:

stmt
A valid statement resource.

row_number
The number of the row that you want to retrieve from the result set. Specify a value for this parameter if you requested a scrollable cursor when you called the `ibm_db.exec_immediate` or `ibm_db.prepare` function. With the default forward-only cursor, each call to a fetch method returns the next row in the result set.

2. Optional: If you called the `ibm_db.fetch_row` function, for each iteration through the result set, retrieve a value from a specified column by calling the `ibm_db.result` function. You can specify the column by passing either an integer that represents the position of the column in the row (starting with 0) or a string that represents the name of the column.

3. Continue fetching rows until the fetch method returns False, which indicates that you have reached the end of the result set.

   For more information about the ibm_db API, see https://github.com/ibmdb/python-ibmdb/wiki/APIs.

Example

Example 1: Fetch rows from a result set by calling the `ibm_db.fetch_both` function

```
import ibm_db

conn = ibm_db.connect("database","username","password")
sql = "SELECT * FROM EMPLOYEE"
stmt = ibm_db.exec_immediate(conn, sql)
dictionary = ibm_db.fetch_both(stmt)
while dictionary != False:
    print "The ID is : ",  dictionary["EMPNO"]
    print "The Name is : ", dictionary[1]
    dictionary = ibm_db.fetch_both(stmt)
```

Example 2: Fetch rows from a result set by calling the `ibm_db.fetch_tuple` function

```
import ibm_db

conn = ibm_db.connect("database","username","password")
sql = "SELECT * FROM EMPLOYEE"
stmt = ibm_db.exec_immediate(conn, sql)
tuple = ibm_db.fetch_tuple(stmt)
while tuple != False:
    print "The ID is : ", tuple[0]
    print "The name is : ", tuple[1]
    tuple = ibm_db.fetch_tuple(stmt)
```

Example 3: Fetch rows from a result set by calling the `ibm_db.fetch_assoc` function

```
import ibm_db

conn = ibm_db.connect("database","username","password")
sql = "SELECT * FROM EMPLOYEE"
stmt = ibm_db.exec_immediate(conn, sql)
dictionary = ibm_db.fetch_assoc(stmt)
while dictionary != False:
    print "The ID is : ", dictionary["EMPNO"]
    print "The name is : ", dictionary["FIRSTNME"]
    dictionary = ibm_db.fetch_assoc(stmt)
```

Example 4: Fetch columns from a result set

```
import ibm_db

conn = ibm_db.connect("database","username","password")
sql = "SELECT * FROM EMPLOYEE"
stmt = ibm_db.exec_immediate(conn, sql)
while ibm_db.fetch_row(stmt) != False:
    print "The Employee number is : ",  ibm_db.result(stmt, 0)
    print "The last name is : ", ibm_db.result(stmt, "LASTNAME")
```

What to do next

When you are ready to close the connection to the database, call the `ibm_db.close` function. If you attempt to close a persistent connection that you created with `ibm_db.pconnect`, the close request returns True, and the connection remains available for the next caller.

Calling stored procedures in Python

To call a stored procedure from a Python application, use `ibm_db.callproc` function. The procedure that you call can include input parameters (IN), output parameters (OUT), and input and output parameters (INOUT).

Before you begin

Obtain a connection resource by calling one of the connection functions in the ibm_db API.

Procedure

Call the `ibm_db.callproc` function by passing the listed arguments:

connection
A valid database connection resource that is returned from the `ibm_db.connect` or `ibm_db.pconnect` function.

procname
A valid stored procedure name

parameters
A tuple of parameters that matches the parameters that are declared in the stored procedure.

Example

To call a stored procedure with the `ibm_db.callproc` function:

```
import ibm_db

conn = ibm_db.connect("sample", "username", "password")
if conn:
  name = "Peaches"
  second_name = "Rickety Ride"
  weight = 0

  print "Values of bound parameters _before_ CALL:"
  print "  1: %s 2: %s 3: %d\n" % (name, second_name, weight)

  stmt, name, second_name, weight = ibm_db.callproc(conn, 'match_animal', (name, second_name, weight))
  if stmt is not None:
    print "Values of bound parameters _after_ CALL:"
    print "  1: %s 2: %s 3: %d\n" % (name, second_name, weight)
```

What to do next

If the procedure call returns one or more result sets, you can begin fetching rows from the statement resource.

Retrieving multiple result sets from a stored procedure in Python

When a single call to a stored procedure returns more than one result set, you can use the `ibm_db.next_result` function of the ibm_db API to retrieve the result sets.

Before you begin

You must have a statement resource returned by the `ibm_db.exec_immediate` or `ibm_db.execute` function that has multiple result sets.

Procedure

To retrieve multiple result sets:

1. Fetch rows from the first result set returned from the procedure by calling one of the listed ibm_db fetch functions, passing the statement resource as an argument. (The first result set that is returned from the procedure is associated with the statement resource.)

Table 13. ibm_db fetch functions	
Function	**Description**
	Returns a tuple, which is. The columns are 0-indexed by columnindexed. position, representing a row in a result set
	Returns a dictionary,. which is indexed by column name, representing a row in a result set
	Returns a dictionary,. which is indexed by both column name and position, representing a row in a result set
	Sets the result set. Use this function to pointer to the next rowiterate through a result or requested rowset.

2. Retrieve the subsequent result sets by passing the original statement resource as the first argument to the `ibm_db.next_result` function. You can fetch rows from the statement resource until no more rows are available in the result set.

   The `ibm_db.next_result` function returns False when no more result sets are available or if the procedure did not return a result set.

   For more information about the ibm_db API, see https://github.com/ibmdb/python-ibmdb/wiki/APIs.

Example

Retrieve multiple result sets from a stored procedure.

```
import ibm_db
conn = ibm_db.connect( "sample", "user", "password" )
if conn:
    sql = 'CALL sp_multi()'
    stmt = ibm_db.exec_immediate(conn, sql)
    row = ibm_db.fetch_assoc(stmt)
    while row != False :
        print "The value returned : ", row
        row = ibm_db.fetch_assoc(stmt)

    stmt1 = ibm_db.next_result(stmt)
    while stmt1 != False:
        row = ibm_db.fetch_assoc(stmt1)
        while row != False :
            print "The value returned : ", row
            row = ibm_db.fetch_assoc(stmt1)
        stmt1 = ibm_db.next_result(stmt)
```

What to do next

When you are ready to close the connection to the database, call the `ibm_db.close` function. If you attempt to close a persistent connection that you created by using `ibm_db.pconnect`, the close request returns True, and the IBM data server client connection remains available for the next caller.

Commit modes in Python applications

You can control how groups of SQL statements are committed by specifying a commit mode for a connection resource. The ibm_db API supports two commit modes: autocommit and manual commit.

Autocommit mode

In autocommit mode, each SQL statement is a complete transaction, which is automatically committed. Autocommit mode helps prevent locking escalation issues that can impede the performance of highly scalable web applications. By default, the ibm_db API opens every connection in autocommit mode.

If autocommit mode is disabled, you can enable the autocommit mode by calling `ibm_db.autocommit(`*conn*`, ibm_db.SQL_AUTOCOMMIT_ON)`, where *conn* is a valid connection resource.

Calling the `ibm_db.autocommit` function might affect the performance of your Python scripts because it requires additional communication between Python and the database management system.

Manual commit mode

In manual commit mode, the transaction ends when you call the `ibm_db.commit` or `ibm_db.rollback` function. This means that all statements executed on the same connection between the start of a transaction and the call to the commit or rollback function are treated as a single transaction.

Manual commit mode is useful if you might have to roll back a transaction that contains one or more SQL statements. If you execute SQL statements in a transaction and the script ends without explicitly committing or rolling back the transaction, the ibm_db extension automatically rolls back any work that is performed in the transaction.

You can turn off autocommit mode when you create a database connection by using the `{ ibm_db.SQL_ATTR_AUTOCOMMIT: ibm_db.SQL_AUTOCOMMIT_OFF }` setting in the `ibm_db.connect` or `ibm_db.pconnect` options array. You can also turn off autocommit mode for a connection resource by calling `ibm_db.autocommit(`*conn*`, ibm_db.SQL_AUTOCOMMIT_OFF)`, where *conn* is a valid connection resource.

For more information about the ibm_db API, see https://github.com/ibmdb/python-ibmdb/wiki/APIs.

Examples

Turn off autocomit mode and end the transaction when `ibm_db.commit` or `ibm_db.rollback` is called.

```
import ibm_db

array = { ibm_db.SQL_ATTR_AUTOCOMMIT : ibm_db.SQL_AUTOCOMMIT_OFF }
conn = ibm_db.pconnect("SAMPLE", "user", "password", array)
sql = "DELETE FROM EMPLOYEE"
try:
   stmt = ibm_db.exec_immediate(conn, sql)
except:
   print "Transaction couldn't be completed."
   ibm_db.rollback(conn)
else:
   ibm_db.commit(conn)
   print "Transaction complete."
```

Error-handling functions in Python

Sometimes errors happen when you attempt to connect to a database or issue an SQL statement. The user name or password might be incorrect, a table or column name might be misspelled, or the SQL statement might be invalid. The ibm_db API provides error-handling functions to help you recover gracefully from the error situations.

Connection errors

Use one of the listed functions to retrieve diagnostic information if a connection attempt fails.

Table 14. ibm_db functions for handling connection errors	
Function	**Description**
`ibm_db.conn_error`	Retrieves the SQLSTATE returned by the last connection attempt
`ibm_db. conn_errormsg`	Retrieves a descriptive error message appropriate for an application error log

SQL errors

Use one of the listed functions to retrieve diagnostic information if an attempt to prepare or execute an SQL statement or to fetch a result from a result set fails.

Table 15. ibm_db functions for handling SQL errors	
Function	**Description**
`ibm_db.stmt_error`	Retrieves the SQLSTATE returned by the last attempt to prepare or execute an SQL statement or to fetch a result from a result set
`ibm_db.stmt_errormsg`	Retrieves a descriptive error message appropriate for an application error log

For more information about the ibm_db API, see https://github.com/ibmdb/python-ibmdb/wiki/APIs.

Examples

Example 1: Handle connection errors

```
import ibm_db
try:
    conn = ibm_db.connect("database","username","password")
except:
    print "no connection:", ibm_db.conn_errormsg()
else:
    print "The connection was successful"
```

Example 2: Handle SQL errors

```
import ibm_db
conn = ibm_db.connect("database","username","password")
sql = "DELETE FROM EMPLOYEE"
try:
   stmt = ibm_db.exec_immediate(conn, sql)
except:
   print "Transaction couldn't be completed:" , ibm_db.stmt_errormsg()
else:
   print "Transaction complete."
```

Database metadata retrieval functions in Python

You can use functions in the ibm_db API to retrieve metadata for IBM databases.

Before calling these functions, you must set up the Python environment, issue `import_db` in your Python script, and obtain a connection resource by calling the `ibm_db.connect` or `ibm_db.pconnect` function.

Important: Calling metadata functions uses a significant amount of space. If possible, cache the results of your calls for use in subsequent calls.

Table 16. ibm_db metadata retrieval functions	
Function	**Description**

Table 16. ibm_db metadata retrieval functions (continued)	
Function	**Description**
	Returns a result set listing the columns and associated privileges for a table
	Returns a result set listing the columns and associated metadata for a table
	Returns a result set listing the foreign keys for a table
	Returns a result set listing the primary keys for a table
	Returns a result set listing the parameters for one or more stored procedures
	Returns a result set listing the stored procedures registered in a database
	Returns a result set listing the unique row identifier columns for a table
	Returns a result set listing the index and statistics for a table
	Returns a result set listing the tables in a database and the associated privileges

For more information about the ibm_db API, see https://github.com/ibmdb/python-ibmdb/wiki/APIs.

Examples

Example 1: Display information about the IBM data server client

```
import ibm_db

conn = ibm_db.connect("sample", "user", "password")
client = ibm_db.client_info(conn)

if client:
    print "DRIVER_NAME: string(%d) \"%s\"" % (len(client.DRIVER_NAME), client.DRIVER_NAME)
    print "DRIVER_VER: string(%d) \"%s\"" % (len(client.DRIVER_VER), client.DRIVER_VER)
    print "DATA_SOURCE_NAME: string(%d) \"%s\"" % (len(client.DATA_SOURCE_NAME), client.DATA_SOURCE_NAME)
    print "DRIVER_ODBC_VER: string(%d) \"%s\"" % (len(client.DRIVER_ODBC_VER), client.DRIVER_ODBC_VER)
    print "ODBC_VER: string(%d) \"%s\"" % (len(client.ODBC_VER), client.ODBC_VER)
    print "ODBC_SQL_CONFORMANCE: string(%d) \"%s\"" % (len(client.ODBC_SQL_CONFORMANCE), client.ODBC_SQL_CONFORMANCE)
    print "APPL_CODEPAGE: int(%s)" % client.APPL_CODEPAGE
    print "CONN_CODEPAGE: int(%s)" % client.CONN_CODEPAGE
    ibm_db.close(conn)
else:
    print "Error."
```

Example 2: Display information about the IBM data server

```
import ibm_db

conn = ibm_db.connect("sample", "user", "password")
server = ibm_db.server_info(conn)

if server:
    print "DBMS_NAME: string(%d) \"%s\"" % (len(server.DBMS_NAME), server.DBMS_NAME)
    print "DBMS_VER: string(%d) \"%s\"" % (len(server.DBMS_VER), server.DBMS_VER)
    print "DB_NAME: string(%d) \"%s\"" % (len(server.DB_NAME), server.DB_NAME)
    ibm_db.close(conn)
else:
    print "Error."
```

Chapter 5. Developing Ruby on Rails applications

The IBM_DB Ruby driver and Rails adapter

Collectively known as the IBM_DB gem, the IBM_DB Ruby driver, and Rails adapter allows Ruby applications to access the IBM database servers.

Ruby applications that are connecting to Db2 for z/OS servers and Db2 for IBM i servers requires the use of the Db2 Connect license.

The IBM_DB Ruby adapter allows any database-backed Rails application to interface with IBM data servers.

For more information about IBM Ruby projects, see https://github.com/ibmdb/ruby-ibmdb[3]

For a list of installation requirements for Db2 database products, see ../../com.ibm.db2.luw.qb.server.doc/doc/r0025127.dita

For a list of installation requirements for IBM Informix server, see http://www-01.ibm.com/support/knowledgecenter/SSGU8G_11.50.0/com.ibm.expr.doc/ids_in_004x.htm

For information about downloading an IBM Data Server Driver Package, see https://www.ibm.com/support/pages/node/387577.

Getting started with Ruby on Rails

Before you can develop Ruby on Rails applications for IBM Database servers, you must set up the Rails environment with an IBM data server client.

Procedure

To set up the Ruby on Rails environment with an IBM data server client:

1. Download and install the latest version of Ruby from http://www.ruby-lang.org/en/downloads/.
2. Install the Rails gem and its dependencies by issuing the gem installation command:

```
gem install rails --include-dependencies
```

What to do next

You are now ready to install the IBM_DB Ruby driver and Rails adapter as a gem.

Installing the IBM_DB Ruby driver and Rails adapter as a Ruby gem

The IBM_DB Ruby driver and Rails adapter is available as a Ruby gem for installation in the IBM data server clients. Ruby Gems is the standard packaging and installation framework for libraries and applications in the Ruby runtime environment. A single file for each bundle is called a gem, which complies to the standardized package format. This package is then distributed and stored in a central repository, allowing simultaneous deployment of multiple versions of the same library or application.

[3] Any references to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The content available at those non-IBM websites is not part of any materials relating to the IBM products described herein. Your use of any non-IBM website is at your own risk.

© Copyright IBM Corp. 2016, 2020

Before you begin

Similar to package management and bundles (.rpm, .deb) used in Linux distributions, these gems can also be queried, installed, uninstalled, and manipulated through the gem utility.

The gem utility can: seamlessly query the remote Gemcutter central repository; and look up and install any of the many readily available utilities. When the IBM_DB gem is installed, the IBM_DB Ruby driver and Rails adapter is immediately accessible from any application in the Ruby runtime environment, through the **require** command:

```
require 'ibm_db'
```

or on Windows:

```
require 'mswin32/ibm_db'
```

Procedure

To install the IBM_DB Ruby driver and Rails adapter as a Ruby gem:

1. On all supported platforms, issue the **gem install** command to install the IBM_DB Ruby driver and Rails adapter:

   ```
   $ gem install ibm_db
   ```

2. Before running any Ruby script that connects to the IBM database server, you must ensure that the IBM_DB Ruby driver can access the CLI driver on Linux or UNIX platforms by adding the `libdb2.so` file path to the **LD_LIBRARY_PATH** environmental variable. If the IBM_DB Ruby driver cannot access the CLI driver, the `missing libraries - libdb2.so.1` error message is returned to your Ruby program.

   When using the IBM Data Server Driver Package software, the `libdb2.so` file is in the `odbc_cli_driver/linux/clidriver/lib` directory.

   In the IBM data server product environment, `libdb2.so` is in the `sqllib/lib/` path.

Verifying the IBM_DB Ruby driver installation with the interactive Ruby shell

To verify the IBM_DB Ruby driver installation, use the interactive Ruby shell (irb) to connect to the database and issue a query.

Procedure

- To verify the Ruby driver installation with the interactive Ruby shell, run the listed commands:

```
C:\>irb
irb(main):001:0> require 'mswin32/ibm_db'
#If you are using Linux based platform issue require 'ibm_db')
=>true
irb(main):002:0> conn = IBM_DB.connect 'devdb','username','password'
=> #<IBM_DB::Connection:0x2dddf40>
#Here ‘devdb’ is the database cataloged in client’s
#database directory or db entry in the db2dsdriver.cfg file.
#To connect to a remote database you
#will need to specify all the necessary attributes like
#hostname, port etc as follows.
#IBM_DB.connect('DRIVER={IBM DB2 ODBC=DRIVER};DATABASE=devdb;HOSTNAME=myhost;
PORT=60000;PROTOCOL=TCPIP;UID=uname;PWD=password;','','')
irb(main):003:0> stmt = IBM_DB.exec conn,'select * from staff'
=> #<IBM_DB::Statement:0x2beaabc>
irb(main):004:0> IBM_DB.fetch_assoc stmt
#Fetches the first row of the result set
```

Verifying the IBM_DB Rails adapter installation

To verify that the IBM_DB Rails adapter is installed correctly, build and run a sample Rails application.

Procedure

1. Create a new Rails application by issuing the following command:

   ```
   C:\>rails new newapp --database=ibm_db
   create
   ```

```
create app/controllers
create app/helpers
create app/models
create app/views/layouts
create config/environments
create config/initializers
create db
[......]
create log/server.log
create log/production.log
create log/development.log
create log/test.log
```

2. Change the current directory to the newly created newapp directory:

```
C:\>cd newapp
```

3. Optional: If you are using a Rails version before the Rails 2.0, you must register the IBM_DB adapter to the list of connection adapters in the Rails framework. You can register the IBM_DB adapter to the list of connection adapters in the Rails framework by manually adding `ibm_db` to the list of connection adapters in <RubyHome>\gems\1.8\gems\activerecord-1.15.6\lib\active_record.rb at approximately line 77:

```
RAILS_CONNECTION_ADAPTERS = %w( mysql postgresql sqlite firebird
sqlserver db2 oracle sybase openbase frontbase ibm_db )
```

4. To configure the connections for the Rails applications, edit the `database.yml` file. A sample development section entry for the `database.yml` file is listed in the following example:

```
development:
adapter: ibm_db
username: db2inst1
password: secret
database: devdb # Database name
#schema: db2inst1
#host: localhost #Host on which the database resides
#port: 50000 #port to which the remote Dataserver is listening
```

5. Create a model and a scaffold by issuing the **rails** command:

```
C:\>rails generate scaffold Tool name:string model_num:integer
exists app/models/
exists app/controllers/
[….]
create db/migrate
create db/migrate/20080716103959_create_tools.rb
```

6. Create the *tools* table in the *devdb* database by issuing the **rake db:migrate** command:

```
C:\ >rake db:migrate
(in C:/ruby trials/newapp)
== 20080716111617 CreateTools: migrating
======================================
-- create_table(:tools)
-> 0.5320s
== 20080716111617 CreateTools: migrated (0.5320s)
```

   The Rails application can now access the *tools* table.

7. To test the application, issue the **rails console** command:

```
C:\ruby trials\newapp>rails console
Loading development environment (Rails )
>> tool = Tool.new
=> #<Tool id: nil, name: nil, model_num: nil, created_at: nil,
updated_at: nil>
>> tool.name = 'chistel'
=> "chistel"
>> tool.model_num = '007'
=> "007"
>> tool.save
=> true
>> Tool.find :all
=> [#<Tool id: 100, name: "chistel", model_num: 7, created_at:
```

```
"2008-07-16 11:29:35", updated_at: "2008-07-16 11:29:35">]
>>
```

Configuring Rails application connections to IBM data servers

You configure database connections for a Rails application by specifying connection details in the `database.yml` file.

Procedure

Edit the database configuration details in *rails_application_path*`\config\database.yml`, and specify the listed connection attributes:

```
# The IBM_DB Adapter requires the native Ruby driver (ibm_db)
    # for IBM data servers (ibm_db.so).
    # +config+ the hash passed as an initializer argument content:
    # == mandatory parameters
    #   adapter:     'ibm_db'        // IBM_DB Adapter name
    #   username:    'db2user'       // data server (database) user
    #   password:    'secret'        // data server (database) password
    #   database:    'DEVDB'         // remote database name (or catalog entry alias)
    # == optional (highly recommended for data server auditing and monitoring purposes)
    #   schema:      'rails123'      // name space qualifier
    #   account:     'tester'        // OS account (client workstation)
    #   app_user:    'test11'        // authenticated application user
    #   application: 'rtests'        // application name
    #   workstation: 'plato'         // client workstation name
    # == remote TCP/IP connection (required when no local database catalog entry available)
    #   host:        'Socrates'      // fully qualified hostname or IP address
    #   port:        '50000'         // data server TCP/IP port number
    #
    # When schema is not specified, the username value is used instead.
```

Note: Changes to connection information in this file are applied when the Rails environment is initialized during server startup. Any changes that you make after initialization do not affect the connections that are created.

Schema, account, app_user, application and workstation are not supported for IBM Informix.

IBM Ruby driver and trusted contexts

The IBM_DB Ruby driver supports trusted contexts by using connection string keywords.

Trusted contexts provide a way of building much faster and more secure three-tier applications. The user's identity is always preserved for auditing and security purposes. When you require secure connections, trusted contexts improve performance because you do not have to get new connections.

Examples

The example establishes a trusted connection and switches the user on the same connection.

```
def trusted_connection(database,hostname,port,auth_user,auth_pass,tc_user,tc_pass)
  dsn = "DATABASE=#{database};HOSTNAME=#{hostname};PORT=#{port};PROTOCOL=TCPIP;UID=#{auth_user};PWD=#{auth_pass};"
  conn_options = {IBM_DB::SQL_ATTR_USE_TRUSTED_CONTEXT => IBM_DB::SQL_TRUE}
  tc_options = {IBM_DB::SQL_ATTR_TRUSTED_CONTEXT_USERID => tc_user, IBM_DB::SQL_ATTR_TRUSTED_CONTEXT_PASSWORD => tc_pass}
  tc_conn = IBM_DB.connect dsn, '', '', conn_options
  if tc_conn
    puts "Trusted connection established successfully."
    val = IBM_DB.get_option tc_conn, IBM_DB::SQL_ATTR_USE_TRUSTED_CONTEXT, 1
    if val
      userBefore = IBM_DB.get_option tc_conn, IBM_DB::SQL_ATTR_TRUSTED_CONTEXT_USERID, 1
      #do some work as user 1
      #....
      #....
      #switch the user
      result = IBM_DB.set_option tc_conn, tc_options, 1
      userAfter = IBM_DB.get_option tc_conn, IBM_DB::SQL_ATTR_TRUSTED_CONTEXT_USERID, 1
      if userBefore != userAfter
        puts "User has been switched."
        #do some work as user 2
        #....
        #....
      end
    end
    IBM_DB.close tc_conn
  else
    puts "Attempt to connect failed due to: #{IBM_DB.conn_errormsg}"
  end
end
```

IBM_DB Rails adapter dependencies and consequences

The IBM_DB adapter (ibm_db_adapter.rb) has a direct dependency on the IBM_DB driver, which uses IBM Data Server Driver for ODBC and CLI to connect to IBM data servers. The IBM Call Level Interface (CLI) is a callable SQL interface to IBM data servers, which is Open Database Connectivity (ODBC) compliant.

This dependency has several ramifications for the IBM_DB adapter and driver.

- Installation of IBM Data Server Driver for ODBC and CLI, which meets the IBM_DB requirement, is required.

  IBM Data Server Driver for ODBC and CLI is included with a full Db2 database install, or you can obtain it separately

  Note: The IBM Data Server Driver for ODBC and CLI is included in the listed client packages:

  – IBM Data Server Client
  – IBM Data Server Runtime Client
  – IBM Data Server Driver Package
- Driver behavior can be modified outside of a Rails application with use of CLI and IBM data server driver configuration keywords.

  The CLI keywords that are set in the `db2cli.ini` file and IBM data server driver configuration keywords in the IBM data server driver configuration file (`db2dsdriver.cfg`) affect Rails applications in a same way as CLI applications. For example, CLI keywords can be used to set the current schema or alter transactional elements such as turning off autocommit behavior.
- Any diagnostic gathering requires CLI driver tracing.

  Because all requests through the IBM_DB driver are implemented through IBM Data Server Driver for ODBC and CLI, the CLI trace facility can identify problems for applications that use the IBM_DB adapter and driver.

  A CLI trace captures all of the API calls made by an application to the IBM Data Server Driver for ODBC and CLI (including all input parameters), and it captures all of the values returned from the driver to the application. It is an interface trace that captures how an application interacts with the IBM Data Server Driver for ODBC and CLI and offers information about the inner workings of the driver.

The IBM_DB Ruby driver and Rails adapter are not supported on JRuby

The IBM_DB adapter is not supported on JRuby.

The IBM_DB adapter is not supported on JRuby because (as stated in the JRuby Wiki, "Getting Started"): "Many Gems will work fine in JRuby, however some Gems build native C libraries as part of their install process. These Gems will not work in JRuby unless the Gem has also provided a Java™ equivalent to the native library." For more information, see JRuby GitHub wiki.

The IBM_DB adapter relies on the IBM_DB Ruby driver (C extension) and the IBM Data Server Driver for ODBC and CLI to access databases on IBM data servers. Alternatively, you can either use the regular C implementation of Ruby, or use JDBC_adapter to access databases.

Heap size considerations with Db2 on Rails

Rails applications on Db2 require the **applheapsz** database configuration parameter to be set to values above 1024.

You must set this parameter for each database for which you will be running Db2 on Rails applications. Use the `db2 update db cfg` command to update the **applheapsz** parameter:

```
db2 update db cfg for database_name using APPLHEAPSZ 1024
```

To activate this parameter, you must restart your Db2 instance.

Index

F

H

I

J

L

M

N

P

R

S

T