Table of Contents

Microsoft PHP Driver for SQL Server
- Getting Started
  - Step 1: Configure development environment for PHP development
  - Step 2: Create a SQL database for PHP development
  - Step 3: Proof of concept connecting to SQL using PHP
  - Step 4: Connect resiliently to SQL with PHP
- Overview
  - Download
  - System Requirements
  - Loading the driver
  - Configuring IIS
  - Release Notes
  - Support Resources
  - About Code Samples
- Programming Guide
  - Connecting to the Server
  - Comparing Execution Functions
  - Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver
  - Retrieving Data
  - Updating Data
  - Converting Data Types
  - Handling Errors and Warnings
  - Logging Activity
  - Constants (Microsoft Drivers for PHP for SQL Server)
  - SQLSRV Driver API Reference
  - PDO_SQLSRV Driver Reference
- Security Considerations
- Code Samples

Example Application (PDO_SQLSRV Driver)
Example Application (SQLSRV Driver)

Microsoft PHP Driver for SQL Server

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The SQL Server Driver for PHP is designed to enable reliable, scalable integration with SQL Server for PHP applications. The SQL Server Driver for PHP is a PHP extension that allows the reading and writing of SQL Server data from within PHP scripts. It provides interfaces for accessing data in all Editions of SQL Server 2005 and later (including Express Editions) as well as Azure SQL Database, and makes use of PHP features, including PHP streams to read and write large objects.

Getting Started

- Step 1: Configure development environment for PHP development
- Step 2: Create a database for PHP development
- Step 3: Proof of concept connecting to SQL using PHP
- Step 4: Connect resiliently to SQL with PHP

Documentation

- Getting Started
- Overview
- Programming Guide
- Security Considerations

Community

- Support Resources for the PHP SQL Driver

Download

- Download Microsoft PHP Driver for SQL Server

Samples

- Code Samples for PHP SQL Driver

Getting Started with the PHP SQL Driver

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

This section provides information about the system requirements for using the Microsoft Drivers for PHP for SQL Server, and for loading the driver into the PHP process space.

Getting Started

- Step 1: Configure development environment for PHP development
- Step 2: Create a database for PHP development
- Step 3: Proof of concept connecting to SQL using PHP
- Step 4: Connect resiliently to SQL with PHP

See Also

Example Application (SQLSRV Driver)
Programming Guide for PHP SQL Driver SQLSRV Driver API Reference

Step 1: Configure development environment for PHP development

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

- Identify which version of the PHP driver you will use, based on your environment, as noted here: System Requirements for the PHP SQL Driver
- Download and install applicable PHP Driver here: Download Microsoft PHP Driver
- Download and install applicable ODBC Driver here: Download ODBC Driver for SQL Server
- Configure the PHP driver and web server for your specific operating system:

Windows

- Configure loading the PHP driver, as noted here: Loading the PHP SQL Driver
- Configure IIS to host PHP applications, as noted here: Configuring IIS for PHP SQL Driver

Linux

- Configure loading the PHP driver and configure your web server to host PHP applications, as noted here: PHP Linux Drivers for SQL Server - Installation Tutorial

Step 2: Create a SQL database for PHP development

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The samples in this section only work with the AdventureWorks schema, on either Microsoft SQL Server or Azure SQL Database.

Azure SQL Database

Create a SQL database in minutes using the Azure portal

Microsoft SQL Server

Microsoft SQL Server database product samples, on CodePlex

Step 3: Proof of concept connecting to SQL using PHP

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

Step 1: Connect

This **OpenConnection** function is called near the top in all of the functions that follow.

```
function OpenConnection()
{
    try
    {
        $serverName = "tcp:myserver.database.windows.net,1433";
        $connectionOptions = array("Database"=>"AdventureWorks",
            "Uid"=>"MyUser", "PWD"=>"MyPassword");
        $conn = sqlsrv_connect($serverName, $connectionOptions);
        if($conn == false)
            die(FormatErrors(sqlsrv_errors()));
    }
    catch(Exception $e)
    {
        echo("Error!");
    }
}
```

Step 2: Execute query

The sqlsrv_query() function can be used to retrieve a result set from a query against SQL Database. This function essentially accepts any query and the connection object and returns a result set which can be iterated over with the use of sqlsrv_fetch_array().

```
    function ReadData()
    {
        try
        {
            $conn = OpenConnection();
            $tsql = "SELECT [CompanyName] FROM SalesLT.Customer";
            $getProducts = sqlsrv_query($conn, $tsql);
            if ($getProducts == FALSE)
                die(FormatErrors(sqlsrv_errors()));
            $productCount = 0;
            while($row = sqlsrv_fetch_array($getProducts, SQLSRV_FETCH_ASSOC))
            {
                echo($row['CompanyName']);
                echo("<br/>");
                $productCount++;
            }
            sqlsrv_free_stmt($getProducts);
            sqlsrv_close($conn);
        }
        catch(Exception $e)
        {
            echo("Error!");
        }
    }
```

Step 3: Insert a row

In this example you will see how to execute an INSERT statement safely, pass parameters which protect your application from SQL injection vulnerability, and retrieve the auto-generated Primary Key value.

```
    function InsertData()
    {
        try
        {
            $conn = OpenConnection();

            $tsql = "INSERT SalesLT.Product (Name, ProductNumber, StandardCost, ListPrice, SellStartDate)
OUTPUT            INSERTED.ProductID VALUES ('SQL Server 1', 'SQL Server 2', 0, 0, getdate())";
            //Insert query
            $insertReview = sqlsrv_query($conn, $tsql);
            if($insertReview == FALSE)
                die(FormatErrors( sqlsrv_errors()));
            echo "Product Key inserted is :";
            while($row = sqlsrv_fetch_array($insertReview, SQLSRV_FETCH_ASSOC))
            {
                echo($row['ProductID']);
            }
            sqlsrv_free_stmt($insertReview);
            sqlsrv_close($conn);
        }
        catch(Exception $e)
        {
            echo("Error!");
        }
    }
```

Step 4: Rollback a transaction

This code example demonstrates the use of transactions in which you:

-Begin a transaction

-Insert a row of data, Update another row of data

-Commit your transaction if the insert and update were successful and rollback the transaction if one of them was not

```
function Transactions()
{
    try
    {
        $conn = OpenConnection();

        if (sqlsrv_begin_transaction($conn) == FALSE)
            die(FormatErrors(sqlsrv_errors()));

        $tsql1 = "INSERT INTO SalesLT.SalesOrderDetail (SalesOrderID,OrderQty,ProductID,UnitPrice)
        VALUES (71774, 22, 709, 33)";
        $stmt1 = sqlsrv_query($conn, $tsql1);

        /* Set up and execute the second query. */
        $tsql2 = "UPDATE SalesLT.SalesOrderDetail SET OrderQty = (OrderQty + 1) WHERE ProductID = 709";
        $stmt2 = sqlsrv_query( $conn, $tsql2);

        /* If both queries were successful, commit the transaction. */
        /* Otherwise, rollback the transaction. */
        if($stmt1 && $stmt2)
        {
               sqlsrv_commit($conn);
               echo("Transaction was commited");
        }
        else
        {
            sqlsrv_rollback($conn);
            echo "Transaction was rolled back.\n";
        }
        /* Free statement and connection resources. */
        sqlsrv_free_stmt( $stmt1);
        sqlsrv_free_stmt( $stmt2);
    }
    catch(Exception $e)
    {
        echo("Error!");
    }
}
```

Additional Examples

Example Application (SQLSRV Driver)
Example Application (PDO_SQLSRV Driver)

Step 4: Connect resiliently to SQL with PHP

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

The demo program is designed so that a transient error during an attempt to connect leads to a retry. But a transient error during query command causes the program to discard the connection and create a new connection, before retrying the query command. We neither recommend nor disrecommend this design choice. The demo program illustrates some of the design flexibility that is available to you.

The length of this code sample is due mostly to the catch exception logic.

The Main method is in Program.cs. The callstack runs as follows:

- Main calls ConnectAndQuery.
- ConnectAndQuery calls EstablishConnection.
- EstablishConnection calls IssueQueryCommand.

The sqlsrv_query() function can be used to retrieve a result set from a query against SQL Database. This function essentially accepts any query and the connection object and returns a result set which can be iterated over with the use of sqlsrv_fetch_array().

```
<?php
    // Variables to tune the retry logic.
    $connectionTimeoutSeconds = 30;  // Default of 15 seconds is too short over the Internet, sometimes.
    $maxCountTriesConnectAndQuery = 3;  // You can adjust the various retry count values.
    $secondsBetweenRetries = 4;  // Simple retry strategy.
    $errNo = 0;
    $serverName = "tcp:yourdatabase.database.windows.net,1433";
    $connectionOptions = array("Database"=>"AdventureWorks",
       "Uid"=>"yourusername", "PWD"=>"yourpassword", "LoginTimeout" => $connectionTimeoutSeconds);
    $conn;
    $errorArr = array();
    for ($cc = 1; $cc <= $maxCountTriesConnectAndQuery; $cc++)
    {
        $errorArr = array();
        $ctr = 0;
        // [A.2] Connect, which proceeds to issue a query command.
        $conn = sqlsrv_connect($serverName, $connectionOptions);
        if( $conn == true)
        {
            echo "Connection was established";
            echo "<br>";

            $tsql = "SELECT [CompanyName] FROM SalesLT.Customer";
            $getProducts = sqlsrv_query($conn, $tsql);
            if ($getProducts == FALSE)
                die(FormatErrors(sqlsrv_errors()));
            $productCount = 0;
            $ctr = 0;
            while($row = sqlsrv_fetch_array($getProducts, SQLSRV_FETCH_ASSOC))
            {
                $ctr++;
                echo($row['CompanyName']);
                echo("<br/>");
                $productCount++;
                if($ctr>10)
                    break;
                }
                sqlsrv_free_stmt($getProducts);
                break;
            }
            // Adds any the error codes from the SQL Exception to an array.
            else {
                if( ($errors = sqlsrv_errors() ) != null) {
                    foreach( $errors as $error ) {
                        $errorArr[$ctr] = $error['code'];
                        $ctr = $ctr + 1;
                    }
                }
                $isTransientError = TRUE;
                // [A.4] Check whether sqlExc.Number is on the whitelist of transients.
                $isTransientError = IsTransientStatic($errorArr);
                if ($isTransientError == TRUE)  // Is a static persistent error...
                {
                    echo("Persistent error suffered, SqlException.Number==". $errorArr[0].". Program Will 
terminate.");
                    echo "<br>";
                    // [A.5] Either the connection attempt or the query command attempt suffered a persistent 
SqlException.
                    // Break the loop, let the hopeless program end.
                    exit(0);
                }
                // [A.6] The SqlException identified a transient error from an attempt to issue a query 
command.
                // So let this method reloop and try again. However, we recommend that the new query
                // attempt should start at the beginning and establish a new connection.
                if ($cc >= $maxCountTriesConnectAndQuery)
                {
                    echo "Transient errors suffered in too many retries - " . $cc ." Program will terminate.";
                    echo "<br>";
                    exit(0);
                }
                echo("Transient error encountered.  SqlException.Number==". $errorArr[0]. " . Program might 
retry by itself.");
                echo "<br>";
                echo $cc . " Attempts so far. Might retry.";
                echo "<br>";
                // A very simple retry strategy, a brief pause before looping. This could be changed to 
exponential if you want.
                sleep(1*$secondsBetweenRetries);
            }
            // [A.3] All has gone well, so let the program end.
        }
        function IsTransientStatic($errorArr) {
            $arrayOfTransientErrorNumbers = array(4060, 10928, 10929, 40197, 40501, 40613);
            $result = array_intersect($arrayOfTransientErrorNumber, $errorArr);
            $count = count($result);
            if($count >= 0) //change to > 0 later.
                return TRUE;
        }
    ?>
```

Overview of the PHP SQL Driver

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The Microsoft Drivers for PHP for SQL Server is a PHP extension that provides data access to SQL Server 2005 and later versions including Azure SQL Database. The extension provides a procedural interface (the SQLSRV driver) and an object-oriented interface (the PDO_SQLSRV driver) for accessing data in all versions (including Express) beginning with SQL Server 2005 (support for PHP for SQL Server versions 3.1 and later begins with SQL Server 2008). The Microsoft Drivers for PHP for SQL Server API includes support for Windows Authentication, transactions, parameter binding, streaming, metadata access, and error handling.

To use the Microsoft Drivers for PHP for SQL Server you must have the correct version of SQL Server Native Client or Microsoft ODBC Driver installed on the same computer PHP is running. For more information, see System Requirements for the PHP SQL Driver.

In This Section

TOPIC	DESCRIPTION
Download Microsoft PHP Driver for SQL Server	Links to download drivers and source code for Microsoft PHP driver for SQL Server.
Release Notes for the PHP SQL Driver	Lists the features that were added for versions 4.0, 3.2, 3.1, 3.0, and 2.0.
Support Resources for the PHP SQL Driver	Provides links to resources that can be helpful when you are developing applications that use the Microsoft Drivers for PHP for SQL Server.
About Code Examples in the Documentation	Provides information that might be helpful when you run the code examples in this documentation.

Reference

SQLSRV Driver API Reference
PDO_SQLSRV Driver Reference
Constants (Microsoft Drivers for PHP for SQL Server)

See Also

Getting Started with the PHP SQL Driver Programming Guide for PHP SQL Driver Example Application (SQLSRV Driver)

Download Microsoft PHP Driver for SQL Server

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The Microsoft Drivers for PHP for SQL Server allow PHP developers to access SQL Server databases. The drivers rely on the Microsoft SQL Server ODBC Driver to handle the low-level communication with SQL Server.

Download

Download Microsoft PHP Driver

Source code

Microsoft has published the source code to the driver on Github. You can download the source code and provide feedback in the form of bug reports and feature requests.

Microsoft PHP driver on Github

System Requirements for the PHP SQL Driver

3/14/2017 • 5 min to read • Edit on GitHub

Download PHP Driver

To access data in a SQL Server or Azure SQL Database using the Microsoft Drivers for PHP for SQL Server, you must have the following components installed on your computer:

- PHP is required. For information about how to download and install the latest stable binaries, see http://php.net. The Microsoft Drivers for PHP for SQL Server require the following versions:

MICROSOFT DRIVERS FOR PHP FOR SQL SERVER VERSION	SUPPORTED PHP VERSIONS
4.0	PHP 7.0
3.2	PHP 5.6.4+ or PHP 5.5.16+ or PHP 5.4.32
3.1	PHP 5.5.16+ or PHP 5.4.32
3.0	PHP 5.4.32 or PHP 5.3.0
2.0	PHP 5.3.0 or PHP 5.2.4 or PHP 5.2.13

- A version of the driver file must be in your PHP extension directory. See Driver Versions later in this topic for information about the different driver files. See Loading the PHP SQL Driver for information on configuring the driver for the PHP runtime. To download the drivers, see Microsoft Drivers for PHP for SQL Server.

- A Web server is required. Your Web server must be configured to run PHP. For information about hosting PHP applications with Internet Information Services (IIS) 6.0, see Using FastCGI to Host PHP Applications on IIS 6.0. For information about hosting PHP applications with IIS 7.0, see Using FastCGI to Host PHP Applications on IIS 7.0.

  The Microsoft Drivers for PHP for SQL Server has been tested using IIS 6 and IIS 7 with FastCGI.

  > **NOTE**
  > Microsoft provides support only for IIS.

- The correct version of the Microsoft ODBC Driver for SQL Server or SQL Server Native Client is required on the computer where PHP is running. Note that if you are using a 64-bit operating system, the x86 version will be installed with the x64 installer (do not use the x86 version of the installer unless you are using a 32-bit operating system).

MICROSOFT DRIVERS FOR PHP FOR SQL SERVER VERSION	VERSION OF MICROSOFT ODBC DRIVER FOR SQL SERVER OR SQL SERVER NATIVE CLIENT
4.0	Microsoft ODBC Driver 11 for SQL Server or Microsoft ODBC Driver 13 for SQL Server. To download the x64 package for 64-bit operating systems, or x86 package for 32-bit operating systems, see the Microsoft ODBC Driver 11 for SQL Server page or Microsoft ODBC Driver 13 for SQL Server page
3.2 or 3.1	Microsoft ODBC Driver 11 for SQL Server. To download the x64 package for 64-bit operating systems, or x86 package for 32-bit operating systems, see the Microsoft ODBC Driver 11 for SQL Server page
3.0	Microsoft SQL Server 2012 Native Client. You can download Microsoft SQL Server 2012 Native Client from the SQL Server 2012 feature pack page
2.0	Microsoft SQL Server 2008 R2 Native Client: Download the X86 package for 32-bit operating systems Download the X64 package for 64-bit operating systems

If you are using the SQLSRV driver, sqlsrv_client_info will return information about which version of SQL Server Native Client or Microsoft ODBC Driver for SQL Server is being used by the Microsoft Drivers for PHP for SQL Server. If you are using the PDO_SQLSRV driver, you can use PDO::getAttribute to discover the version.

Database Versions

- Azure SQL Databases are supported. For information see Connecting to Microsoft Azure SQL Database.
- Microsoft Drivers for PHP for SQL Server version 3.1 and later support SQL Server 2008 and later
- Microsoft Drivers for PHP for SQL Server version 2.0 and 3.0 support SQL Server 2005 and later

Driver Versions

This section lists the drivers that are included with each version of the Microsoft Drivers for PHP for SQL Server.

Follow the installation instructions in Loading the PHP SQL Driver to configure the driver for use with the PHP runtime.

Microsoft Drivers 4.0 for PHP for SQL Server:

On Windows, for 4.0 the following versions of the driver are installed:

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
php_sqlsrv_7_nts_x86.dll php_pdo_sqlsrv_7_nts_x86.dll	7.0	no	32-bit php7.dll

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
php_sqlsrv_7_ts_x86.dll php_pdo_sqlsrv_7_ts_x86.dll	7.0	yes	32-bit php7ts.dll
php_sqlsrv_7_nts_x64.dll php_pdo_sqlsrv_7_nts_x64.dll	7.0	no	64-bit php7.dll
php_sqlsrv_7_ts_x64.dll php_pdo_sqlsrv_7_ts_x64.dll	7.0	yes	64-bit php7ts.dll

On the supported versions of Linux, the appropriate version of sqlsrv and/or pdo_sqlsrv can be installed using PHP's PECL package system.

Microsoft Drivers 3.2 for PHP for SQL Server installs the following versions of the driver:

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
php_sqlsrv_54_nts.dll php_pdo_sqlsrv_54_nts.dll	5.4	no	php5.dll
php_sqlsrv_54_ts.dll php_pdo_sqlsrv_54_ts.dll	5.4	yes	php5ts.dll
php_sqlsrv_55_nts.dll php_pdo_sqlsrv_55_nts.dll	5.5	no	php5.dll
php_sqlsrv_55_ts.dll php_pdo_sqlsrv_55_ts.dll	5.5	yes	php5ts.dll
php_sqlsrv_56_nts.dll php_pdo_sqlsrv_56_nts.dll	5.6	no	php5.dll
php_sqlsrv_56_ts.dll php_pdo_sqlsrv_56_ts.dll	5.6	yes	php5ts.dll

Microsoft Drivers 3.1 for PHP for SQL Server installs the following versions of the driver:

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
php_sqlsrv_54_nts.dll php_pdo_sqlsrv_54_nts.dll	5.4	no	php5.dll
php_sqlsrv_54_ts.dll php_pdo_sqlsrv_54_ts.dll	5.4	yes	php5ts.dll

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
php_sqlsrv_55_nts.dll php_pdo_sqlsrv_55_nts.dll	5.5	no	php5.dll
php_sqlsrv_55_ts.dll php_pdo_sqlsrv_55_ts.dll	5.5	yes	php5ts.dll

Microsoft Drivers 3.0 for PHP for SQL Server installs the following versions of the driver:

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
php_sqlsrv_53_nts.dll php_pdo_sqlsrv_53_nts.dll	5.3	no	php5.dll
php_sqlsrv_53_ts.dll php_pdo_sqlsrv_53_ts.dll	5.3	yes	php5ts.dll
php_sqlsrv_54_nts.dll php_pdo_sqlsrv_54_nts.dll	5.4	no	php5.dll
php_sqlsrv_54_ts.dll php_pdo_sqlsrv_54_ts.dll	5.4	yes	php5ts.dll

Microsoft Drivers 2.0 for PHP for SQL Server installs the following versions of the driver:

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
php_sqlsrv_53_nts_vc6.dll php_pdo_sqlsrv_53_nts_vc6.dll	5.3	no	php5.dll
php_sqlsrv_53_nts_vc9.dll php_pdo_sqlsrv_53_nts_vc9.dll	5.3	no	php5.dll
php_sqlsrv_53_ts_vc6.dll php_pdo_sqlsrv_53_ts_vc6.dll	5.3	yes	php5ts.dll
php_sqlsrv_53_ts_vc9.dll php_pdo_sqlsrv_53_ts_vc9.dll	5.3	yes	php5ts.dll

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
php_sqlsrv_52_nts_vc6.dll php_pdo_sqlsrv_52_nts_vc6.dll	5.2	no	php5.dll
php_sqlsrv_52_ts_vc6.dll php_pdo_sqlsrv_52_ts_vc6.dll	5.2	yes	php5ts.dll

If the name of the driver file contains "vc9", it should be used with a PHP version compiled with Visual C++ 9.0.

Operating Systems

Supported operating systems for the versions of the driver are as follows:

- 4.0 (for Windows):
  - Windows Server 2008 SP2
  - Windows Server 2008 R2 SP1
  - Windows Server 2012
  - Windows Server 2012 R2
  - Windows Vista SP2
  - Windows 7 SP1
  - Windows 8
  - Windows 8.1
  - Windows 10
- 4.0 (for Linux):
  - Ubuntu 15.04 (64-bit)
  - Ubuntu 16.04 (64-bit)
  - Red Hat Enterprise Linux 7 (64-bit)
- 3.2 and 3.1 :
  - Windows Server 2008 R2 SP1
  - Windows Vista SP2
  - Windows Server 2008 SP2
  - Windows 7 SP1
  - Windows Server 2012
  - Windows Server 2012 R2
  - Windows 8
  - Windows 8.1
- 3.0 :
  - Windows Server 2008 R2 SP1
  - Windows Vista SP2
  - Windows Server 2008 SP2
  - Windows 7 SP1
- 2.0:
  - Windows Server 2003 Service Pack 1

- Windows XP Service Pack 3
- Windows Vista Service Pack 1 or later
- Windows Server 2008
- Windows Server 2008 R2
- Windows 7

See Also

Getting Started with the PHP SQL Driver Programming Guide for PHP SQL Driver SQLSRV Driver API Reference

Loading the PHP SQL Driver

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

This topic provides instructions for loading the Microsoft Drivers for PHP for SQL Server into the PHP process space.

There are two options for loading a driver. The driver can be loaded when PHP is started or it can be loaded at PHP script runtime.

Moving the Driver File into Your Extension Directory

Regardless of which procedure you use, the first step will be to put the driver file in a directory where the PHP runtime can find it. So, put the driver file in your PHP extension directory. See System Requirements for the PHP SQL Driver for a list of the driver files that are installed with the Microsoft Drivers for PHP for SQL Server.

If necessary, specify the directory location of the driver file in the PHP configuration file (php.ini), using the **extension_dir** option. For example, if you will put the driver file in your c:\php\ext directory, use this option:

```
extension_dir = "c:\PHP\ext"
```

Loading the Driver at PHP Startup

To load the Microsoft Drivers for PHP for SQL Server when PHP is started, first move a driver file into your extension directory. Then, follow these steps:

1. To enable the **SQLSRV** driver, modify **php.ini** by adding the following line to the extension section, or modifying the line that is already there:

   On Windows (this example uses the version 4.0 thread safe driver for PHP 7):

   ```
   extension=php_sqlsrv_7_ts.dll
   ```

   On Linux (this example uses the version as installed by PECL):

   ```
   extension=sqlsrv.so
   ```

   To enable the **PDO_SQLSRV** driver, modify **php.ini** by adding the following line to the extension section, or modifying the line that is already there:

   On Windows (this example uses the version 4.0 thread safe driver for PHP 7):

   ```
   extension=php_pdo_sqlsrv_7_ts.dll
   ```

   On Linux (this example uses the version as installed by PECL):

   ```
   extension=pdo_sqlsrv.so
   ```

2. If you want to use the **PDO_SQLSRV** driver, the PHP Data Objects (PDO) extension must be available, either as a built-in extension, or as a dynamically-loaded extension. If you need to load the PDO_SQLSRV driver dynamically, the php_pdo.dll (or pdo.so on Linux) must be present in the extension directory and the following line needs to be in the php.ini:

   On Windows:

   ```
   extension=php_pdo.dll
   ```

   On Linux:

   ```
   extension=pdo.so
   ```

3. Restart the Web server.

NOTE

To determine whether the driver has been successfully loaded, run a script that calls phpinfo().

For more information about **php.ini** directives, see Description of core php.ini directives.

Loading the Driver at PHP Script Runtime

To load the Microsoft Drivers for PHP for SQL Server at script runtime, first move a driver file into your extension directory. Then include the following line at the start of the PHP script that matches the filename of the driver:

```
dl('php_pdo_sqlsrv_7_ts.dll');
```

For more information about PHP functions related to dynamically loading extensions, see dl and extension_loaded.

See Also

Getting Started with the PHP SQL Driver System Requirements for the PHP SQL Driver Programming Guide for PHP SQL Driver SQLSRV Driver API Reference

Configuring IIS for PHP SQL Driver

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

This topic provides links to resources on the Internet Information Services (IIS) Web site that are relevant to configuring IIS to host PHP applications. The resources listed here are specific to using FastCGI with IIS. FastCGI is a standard protocol that allows an application framework's Common Gateway Interface (CGI) executables to interface with the Web server. FastCGI differs from the standard CGI protocol in that FastCGI re-uses CGI processes for multiple requests.

Tutorials

The following links are for tutorials about setting up FastCGI for PHP and hosting PHP applications on IIS 6.0 and IIS 7.0:

- FastCGI with PHP
- Using FastCGI to Host PHP Applications on IIS 7.0
- Using FastCGI to Host PHP Applications on IIS 6.0
- Configuring FastCGI Extension for IIS 6.0

Video Presentations

The following links are for video presentations about setting up FastCGI for PHP and using IIS 7.0 features to host PHP applications:

- Setting up FastCGI for PHP
- Partying with PHP on Microsoft Internet Information Services 7

Support Resources

The following forums provide community support for FastCGI on IIS:

- FastCGI Handler
- IIS 7 - FastCGI Module

See Also

Getting Started with the PHP SQL Driver Programming Guide for PHP SQL Driver SQLSRV Driver API Reference Constants (Microsoft Drivers for PHP for SQL Server)

Release Notes for the PHP SQL Driver

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

This topic discusses what was added in the each version of the Microsoft Drivers for PHP for SQL Server.

What's New in Version 4.0

Support for PHP 7.0

Full 64-bit support

What's New in Version 3.2

Support for PHP 5.6

Includes latest updates for prior PHP versions 5.5 and 5.4

Requires Microsoft ODBC Driver 11 for SQL Server

What's New in Version 3.1

Support for PHP 5.5

Requires Microsoft ODBC Driver 11 for SQL Server. Previous versions required SQL Native Client.

What's New in Version 3.0

Support for PHP 5.4. PHP 5.2 is not supported in version 3 of the Microsoft Drivers for PHP for SQL Server.

AttachDBFileName connection option is added. For more information, see Connection Options.

Support for LocalDB, which was added in SQL Server 2012 . For more information, see PHP Driver for SQL Server Support for LocalDB.

AttachDBFileName connection option is added. For more information, see Connection Options.

Support for the high-availability, disaster recovery features. For more information, see PHP Driver for SQL Server Support for High Availability, Disaster Recovery.

Support for client-side cursors (caching a result set in-memory). For more information, see Cursor Types (SQLSRV Driver) and Cursor Types (PDO_SQLSRV Driver).

The PDO::ATTR_EMULATE_PREPARES attribute has been added. See PDO::prepare for more information.

What's New in Version 2.0

In version 2.0, support for the PDO_SQLSRV driver was added. For more information, see PDO_SQLSRV Driver Reference.

See Also

Overview of the PHP SQL Driver

Support Resources for the PHP SQL Driver

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

This topic lists resources that may be helpful when you are developing applications that use the Microsoft Drivers for PHP for SQL Server.

Microsoft Drivers for PHP for SQL Server Support Resources

For the latest documentation, see the Microsoft PHP Driver for SQL Server for the Microsoft Drivers for PHP for SQL Server.

For peer-to-peer support, visit the Microsoft Drivers for PHP for SQL Server Forum.

To provide feedback, ask questions, or learn what the development team is considering, visit the Microsoft Drivers for PHP for SQL Server Team Blog.

Microsoft PHP driver for SQL Server source code on Github

SQL Server/Transact-SQL Support Resources

SQL Server and Transact-SQL documentation can be found at SQL Server Books Online.

For peer-to-peer support, visit the MSDN SQL Server Forums.

Internet Information Services (IIS) Support Resources

For the latest IIS news, visit IIS Home.

For peer-to-peer support, visit the IIS Forums.

PHP Support Resources

PHP for Windows Documentation

For the latest information about PHP, visit http:/www.php.net/.

For PHP documentation, visit http://www.php.net/docs.php.

Microsoft Customer Support

For support questions related to the Microsoft Drivers for PHP for SQL Server, contact Microsoft Customer Service and Support. For more information, see How and when to contact Microsoft Customer Service and Support.

See Also

Overview of the PHP SQL Driver

About Code Examples in the Documentation

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

There are several points to note when you execute the code examples in the Microsoft Drivers for PHP for SQL Server documentation:

- Nearly all the examples assume that SQL Server 2005 or later (SQL Server 2008 or later if using version 3.1) and the AdventureWorks database are installed on the local computer.

  For information about how to download free editions and trial versions of SQL Server, see SQL Server.

  For information about how to download the AdventureWorks database, see Microsoft SQL Server Samples and Community Projects.

  For information about how to install the AdventureWorks database, see Walkthrough: Installing the AdventureWorks Database.

- Nearly all the code examples in this documentation are intended to be run from the command line, which enables automated testing of all the code examples. For information about how to run PHP from the command line, see Using PHP from the command line.

- Although examples are written to be run from the command line, each example can be run by invoking it from a browser without making any changes to the script. To achieve nice output formatting, replace each "\n" with "<\/br>" in each example before invoking it from a browser.

- For the purpose of keeping each example narrowly focused, correct error handling is not done in all examples. It is recommended that any call to a **sqlsrv** function or PDO method be checked for errors and handled according to the needs of the application.

  An easy way to obtain error information when an error is encountered is to exit the script with the following line of code:

  ```
  die( print_r( sqlsrv_errors(), true));
  ```

  Or, if you are using PDO,

  ```
  print_r ($stmt->errorInfo());
  die();
  ```

  For more information about handling errors and warnings, see Handling Errors and Warnings.

See Also

Overview of the PHP SQL Driver

Programming Guide for PHP SQL Driver

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

This section contains topics that help you develop applications with the Microsoft Drivers for PHP for SQL Server.

In This Section

TOPIC	DESCRIPTION
Connecting to the Server	Describes the options and procedures for connecting to SQL Server.
Comparing Execution Functions	Compares the functions that are used to execute a query by examining different use cases for each. Specifically, this document compares executing a single query with executing a prepared query multiple times.
Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver	Discusses how to use the PDO::SQLSRV_ATTR_DIRECT_QUERY attribute to specify direct statement execution instead of the default, which is prepared statement execution.
Retrieving Data	Examines the various ways that data can be retrieved.
Updating Data	Addresses how to update data in a database by examining common use cases.
Converting Data Types	Addresses how to specify data types and provide details on default data types.
Handling Errors and Warnings	Addresses how to handle errors and warnings.
Logging Activity	Provides information about logging errors and warnings.
Security Considerations for PHP SQL Driver	Describes security considerations for developing, deploying, and running applications.

See Also

Overview of the PHP SQL Driver Getting Started with the PHP SQL Driver SQLSRV Driver API Reference
Constants (Microsoft Drivers for PHP for SQL Server)
Example Application (SQLSRV Driver)

Connecting to the Server

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The topics in this section describe the options and procedures for connecting to SQL Server with the Microsoft Drivers for PHP for SQL Server.

The Microsoft Drivers for PHP for SQL Server can connect to SQL Server by using Windows Authentication or by using SQL Server Authentication. By default, the Microsoft Drivers for PHP for SQL Server try to connect to the server by using Windows Authentication.

In This Section

TOPIC	DESCRIPTION
How to: Connect Using Windows Authentication	Describes how to establish a connection by using Windows Authentication.
How to: Connect Using SQL Server Authentication	Describes how to establish a connection by using SQL Server Authentication.
How to: Connect on a Specified Port	Describes how to connect to the server on a specific port.
Connection Pooling	Provides information about connection pooling in the driver.
How to: Disable Multiple Active Resultsets (MARS)	Describes how to disable the MARS feature when making a connection.
Connection Options	Lists the options that are permitted in the associative array that contains connection attributes.
PHP Driver for SQL Server Support for LocalDB	Describes Microsoft Drivers for PHP for SQL Server support for the LocalDB feature, which was added in SQL Server 2012 .
PHP Driver for SQL Server Support for High Availability, Disaster Recovery	Discusses how your application can be configured to take advantage of the high-availability, disaster recovery features added in SQL Server 2012 .
Connecting to Microsoft Azure SQL Database	Discusses how to connect to an Azure SQL Database.

See Also

Programming Guide for PHP SQL Driver Example Application (SQLSRV Driver)

How to: Connect Using Windows Authentication

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

By default, the Microsoft Drivers for PHP for SQL Server use Windows Authentication to connect to SQL Server. It is important to notice that in most scenarios, this means that the Web server's process identity or thread identity (if the Web server is using impersonation) is used to connect to the server, not an end-user's identity.

The following points must be considered when you use Windows Authentication to connect to SQL Server:

- The credentials under which the Web server's process (or thread) is running must map to a valid SQL Server login in order to establish a connection.
- If SQL Server and the Web server are on different computers, SQL Server must be configured to enable remote connections.

NOTE

Connection attributes such as *Database* and *ConnectionPooling* can be set when you establish a connection. For a complete list of supported connection attributes, see Connection Options.

Windows Authentication should be used to connect to SQL Server whenever possible for the following reasons:

- No credentials are passed over the network during authentication; user names and passwords are not embedded in the database connection string. This means that malicious users or attackers cannot obtain the credentials by monitoring the network or by viewing connection strings inside configuration files.
- Users are subject to centralized account management; security policies such as password expiration periods, minimum password lengths, and account lockout after multiple invalid logon requests are enforced.

If Windows Authentication is not a practical option, see How to: Connect Using SQL Server Authentication.

Example

Using the SQLSRV driver of the Microsoft Drivers for PHP for SQL Server, the following example uses the Windows Authentication to connect to a local instance of SQL Server. After the connection has been established, the server is queried for the login of the user who is accessing the database.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the browser when the example is run from the browser.

```
<?php
/* Specify the server and connection string attributes. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");

/* Connect using Windows Authentication. */
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Unable to connect.</br>";
     die( print_r( sqlsrv_errors(), true));
}

/* Query SQL Server for the login of the user accessing the
database. */
$tsql = "SELECT CONVERT(varchar(32), SUSER_SNAME())";
$stmt = sqlsrv_query( $conn, $tsql);
if( $stmt === false )
{
     echo "Error in executing query.</br>";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve and display the results of the query. */
$row = sqlsrv_fetch_array($stmt);
echo "User login: ".$row[0]."</br>";

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

Example

The following example uses the PDO_SQLSRV driver to accomplish the same task as the previous sample.

```
<?php
try {
   $conn = new PDO( "sqlsrv:Server=(local);Database=AdventureWorks", NULL, NULL);
   $conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
}

catch( PDOException $e ) {
   die( "Error connecting to SQL Server" );
}

echo "Connected to SQL Server\n";

$query = 'select * from Person.ContactType';
$stmt = $conn->query( $query );
while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
   print_r( $row );
}
?>
```

See Also

How to: Connect Using SQL Server Authentication
Programming Guide for PHP SQL Driver About Code Examples in the Documentation
How to: Create a SQL Server Login
How to: Create a Database User

Managing Users, Roles, and Logins

User-Schema Separation

Grant Object Permissions (Transact-SQL)

How to: Connect Using SQL Server Authentication

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

The Microsoft Drivers for PHP for SQL Server supports SQL Server Authentication when you connect to SQL Server.

SQL Server Authentication should be used only when Windows Authentication is not possible. For information about connecting with Windows Authentication, see How to: Connect Using Windows Authentication.

The following points must be considered when you use SQL Server Authentication to connect to SQL Server:

- SQL Server Mixed Mode Authentication must be enabled on the server.
- The user ID and password (*UID* and *PWD* connection attributes in the SQLSRV driver) must be set when you try to establish a connection. The user ID and password must map to a valid SQL Server user and password.

NOTE

Passwords that contain a closing curly brace (}) must be escaped with a second closing curly brace. For example, if the SQL Server password is "pass}word", the value of the *PWD* connection attribute must be set to "pass}}word".

The following precautions should be taken when you use SQL Server Authentication to connect to SQL Server:

- Protect (encrypt) the credentials passed over the network from the Web server to the database. Credentials are encrypted by default beginning in SQL Server 2005. For added security, set the Encrypt connection attribute to "on" in order to encrypt all data sent to the server.

NOTE

Setting the Encrypt connection attribute to "on" can cause slower performance because data encryption can be computationally intensive.

- Do not include values for the connection attributes *UID* and *PWD* in plain text in PHP scripts. These values should be stored in an application-specific directory with the appropriate restricted permissions.
- Avoid use of the *sa* account. Map the application to a database user who has the desired privileges and use a strong password.

NOTE

Connection attributes besides user ID and password can be set when you establish a connection. For a complete list of supported connection attributes, see Connection Options.

Example

The following example uses the SQLSRV driver with SQL Server Authentication to connect to a local instance of SQL Server. The values for the required *UID* and *PWD* connection attributes are taken from application-specific text files, *uid.txt* and *pwd.txt*, in the *C:\AppData* directory. After the connection has been established, the server is queried to verify the user login.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the browser when the example is run from the browser.

```
<?php
/* Specify the server and connection string attributes. */
$serverName = "(local)";

/* Get UID and PWD from application-specific files.  */
$uid = file_get_contents("C:\AppData\uid.txt");
$pwd = file_get_contents("C:\AppData\pwd.txt");
$connectionInfo = array( "UID"=>$uid,
                         "PWD"=>$pwd,
                         "Database"=>"AdventureWorks");

/* Connect using SQL Server Authentication. */
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Unable to connect.</br>";
     die( print_r( sqlsrv_errors(), true));
}

/* Query SQL Server for the login of the user accessing the
database. */
$tsql = "SELECT CONVERT(varchar(32), SUSER_SNAME())";
$stmt = sqlsrv_query( $conn, $tsql);
if( $stmt === false )
{
     echo "Error in executing query.</br>";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve and display the results of the query. */
$row = sqlsrv_fetch_array($stmt);
echo "User login: ".$row[0]."</br>";

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

Example

This sample uses the PDO_SQLSRV driver to demonstrate how to connect with SQL Server Authentication.

```
<?php
   $serverName = "(local)";
   $database = "AdventureWorks";

   // Get UID and PWD from application-specific files.
   $uid = file_get_contents("C:\AppData\uid.txt");
   $pwd = file_get_contents("C:\AppData\pwd.txt");

   try {
      $conn = new PDO( "sqlsrv:server=$serverName;Database = $database", $uid, $pwd);
      $conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
   }

   catch( PDOException $e ) {
      die( "Error connecting to SQL Server" );
   }

   echo "Connected to SQL Server\n";

   $query = 'select * from Person.ContactType';
   $stmt = $conn->query( $query );
   while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
      print_r( $row );
   }

   // Free statement and connection resources.
   $stmt = null;
   $conn = null;
?>
```

See Also

How to: Connect Using SQL Server Authentication
Programming Guide for PHP SQL Driver About Code Examples in the Documentation
SUSER_SNAME (Transact-SQL)
How to: Create a SQL Server Login
How to: Create a Database User
Managing Users, Roles, and Logins
User-Schema Separation
Grant Object Permissions (Transact-SQL)

How to: Connect on a Specified Port

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

This topic describes how to connect to SQL Server on a specified port with the Microsoft Drivers for PHP for SQL Server.

To connect on a specified port

1. Verify the port on which the server is configured to accept connections. For information about configuring a server to accept connections on a specified port, see How to: Configure a Server to Listen on a Specific TCP Port (SQL Server Configuration Manager).

2. Add the desired port to the *$serverName* parameter of the sqlsrv_connect function. Separate the server name and the port with a comma. For example, the following lines of code use the SQLSRV driver to demonstrate how to connect to a server named *myServer* on port 1521:

   ```
   $serverName = "myServer, 1521";
   sqlsrv_connect( $serverName );
   ```

   The following lines of code use the PDO_SQLSRV driver to demonstrate how to connect to a server named *myServer* on port 1521:

   ```
   $serverName = "(local), 1521";
   $database = "AdventureWorks";
   $conn = new PDO( "sqlsrv:server=$serverName;Database=$database", "", "");
   ```

See Also

Connecting to the Server
Programming Guide for PHP SQL Driver Getting Started with the PHP SQL Driver SQLSRV Driver API Reference
PDO_SQLSRV Driver Reference

Connection Pooling (Microsoft Drivers for PHP for SQL Server)

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The following are important points to note about connection pooling in the Microsoft Drivers for PHP for SQL Server:

- The Microsoft Drivers for PHP for SQL Server uses ODBC connection pooling.
- By default, connection pooling is enabled. When you connect to a server, the driver attempts to use a pooled connection before it creates a new one. If an equivalent connection is not found in the pool, a new connection is created and added to the pool. The driver determines whether connections are equivalent based on a comparison of connection strings.
- When a connection from the pool is used, the connection state is reset.
- Closing the connection returns the connection to the pool.

For more information about connection pooling, see Driver Manager Connection Pooling.

Connection Attributes

You can force the driver to create a new connection (instead of looking for an equivalent connection in the connection pool) by setting the value of the *ConnectionPooling* attribute in the connection string to **false** (or 0).

If the *ConnectionPooling* attribute is omitted from the connection string or if it is set to **true** (or 1), the driver will only create a new connection if an equivalent connection does not exist in the connection pool.

For information about other connection attributes, see Connection Options.

See Also

How to: Connect Using Windows Authentication
How to: Connect Using SQL Server Authentication

How to: Disable Multiple Active Resultsets (MARS)

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

If you need to connect to a SQL Server data source that does not enable Multiple Active Result Sets (MARS), you can use the MultipleActiveResultSets connection option to disable or enable MARS.

Procedure

To disable MARS support

- Use the following connection option:

  ```
  'MultipleActiveResultSets'=>false
  ```

  If your application attempts to execute a query on a connection that has an open active result set, the second query attempt will return the following error information:

  The connection cannot process this operation because there is a statement with pending results. To make the connection available for other queries either fetch all results, cancel or free the statement. For more information about the MultipleActiveResultSets connection option, see Connection Options.

Example

The following example shows how to disable MARS support, using the SQLSRV driver of the Microsoft Drivers for PHP for SQL Server.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "MyServer";
$connectionInfo = array( "Database"=>"AdventureWorks", 'MultipleActiveResultSets'=> false);
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
   echo "Could not connect.\n";
   die( print_r( sqlsrv_errors(), true));
}

sqlsrv_close( $conn);
?>
```

Example

The following example shows how to disable MARS support, using the PDO_SQLSRV driver of the Microsoft Drivers for PHP for SQL Server.

```
<?php
// Connect to the local server using Windows Authentication and AdventureWorks database
$serverName = "(local)";
$database = "AdventureWorks";

try {
   $conn = new PDO(" sqlsrv:server=$serverName ; Database=$database ; MultipleActiveResultSets=false ", NULL,
NULL);
   $conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
}

catch( PDOException $e ) {
   die( "Error connecting to SQL Server" );
}

$conn = null;
?>
```

See Also

Connecting to the Server

Connection Options

3/14/2017 • 5 min to read • Edit on GitHub

Download PHP Driver

This topic lists the options that are permitted in the associative array (when using sqlsrv_connect in the SQLSRV driver) or the keywords that are permitted in the data source name (dsn) (when using PDO::__construct in the PDO_SQLSRV driver).

KEY	VALUE	DESCRIPTION	DEFAULT
APP	String	Specifies the application name used in tracing.	No value set.
ApplicationIntent	String	Declares the application workload type when connecting to a server. Possible values are ReadOnly and ReadWrite. For more information about Microsoft Drivers for PHP for SQL Server support for AlwaysOn Availability Groups , see PHP Driver for SQL Server Support for High Availability, Disaster Recovery.	ReadWrite
AttachDBFileName	String	Specifies which database file the server should attach.	No value set.
CharacterSet (not supported in the PDO_SQLSRV driver)	String	Specifies the character set used to send data to the server. Possible values are SQLSRV_ENC_CHAR and UTF-8. For more information, see How to: Send and Retrieve UTF-8 Data Using Built-In UTF-8 Support.	SQLSRV_ENC_CHAR
ConnectionPooling	1 or **true** for connection pooling on. 0 or **false** for connection pooling off.	Specifies whether the connection is assigned from a connection pool (1 or **true**) or not (0 or **false**).	**true** (1)
Database	String	Specifies the name of the database in use for the connection being established[1].	The default database for the login being used.

KEY	VALUE	DESCRIPTION	DEFAULT
Encrypt	1 or **true** for encryption on. 0 or **false** for encryption off.	Specifies whether the communication with SQL Server is encrypted (1 or **true**) or unencrypted (0 or **false**)[2].	**false** (0)
Failover_Partner	String	Specifies the server and instance of the database's mirror (if enabled and configured) to use when the primary server is unavailable. There are restrictions to using Failover_Partner with MultiSubnetFailover. For more information, see PHP Driver for SQL Server Support for High Availability, Disaster Recovery.	No value set.
LoginTimeout	Integer (SQLSRV driver String (PDO_SQLSRV driver	Specifies the number of seconds to wait before failing the connection attempt.	No timeout.
MultipleActiveResultSets	1 or **true** to use multiple active result sets. 0 or **false** to disable multiple active result sets.	Disables or explicitly enables support for multiple active Result sets (MARS). For more information, see How to: Disable Multiple Active Resultsets (MARS).	true (1)

KEY	VALUE	DESCRIPTION	DEFAULT
MultiSubnetFailover	String	Always specify **multiSubnetFailover=yes** when connecting to the availability group listener of a SQL Server 2012 availability group or a SQL Server 2012 Failover Cluster Instance. **multiSubnetFailover=yes** configures Microsoft Drivers for PHP for SQL Server to provide faster detection of and connection to the (currently) active server. Possible values are Yes and No. For more information about Microsoft Drivers for PHP for SQL Server support for AlwaysOn Availability Groups , see PHP Driver for SQL Server Support for High Availability, Disaster Recovery.	No
PWD (not supported in the PDO_SQLSRV driver)	String	Specifies the password associated with the User ID to be used when connecting with SQL Server Authentication[3].	No value set.
QuotedId	1 or **true** to use SQL-92 rules. 0 or **false** to use legacy rules.	Specifies whether to use SQL-92 rules for quoted identifiers (1 or **true**) or to use legacy Transact-SQL rules (0 or **false**).	**true** (1)
ReturnDatesAsStrings (not supported in the PDO_SQLSRV driver)	1 or **true** to return date and time types as strings. 0 or **false** to return date and time types as PHP **DateTime** types.	Retrieves date and time types (datetime, date, time, datetime2, and datetimeoffset) as strings or as PHP types. When using the PDO_SQLSRV driver, dates are returned as strings. The PDO_SQLSRV driver has no **datetime** type. For more information, see How to: Retrieve Date and Time Type as Strings Using the SQLSRV Driver.	**false**

KEY	VALUE	DESCRIPTION	DEFAULT
Scrollable	String	"buffered" indicates that you want a client-side (buffered) cursor, which allows you to cache an entire result set in memory. See Cursor Types (SQLSRV Driver) for more information.	Forward-only cursor
Server (not supported in the SQLSRV driver)	String	The SQL Server instance to connect to. You can also specify a virtual network name, to connect to an AlwaysOn availability group. For more information about Microsoft Drivers for PHP for SQL Server support for AlwaysOn Availability Groups , see PHP Driver for SQL Server Support for High Availability, Disaster Recovery.	Server is a required keyword (although it does not have to be the first keyword in the connection string). If a server name is not passed to the keyword, an attempt will be made to connect to the local instance. The value passed to Server can be the name of a SQL Server instance, or the IP address of the instance. You can optionally specify a port number (for example, `sqlsrv:server=(local),1033`). Beginning in version 3.0 of the Microsoft Drivers for PHP for SQL Server you can also specify a LocalDB instance with `server=(localdb)\instancename` . For more information, see PHP Driver for SQL Server Support for LocalDB.
TraceFile	String	Specifies the path for the file used for trace data.	No value set.
TraceOn	1 or **true** to enable tracing. 0 or **false** to disable tracing.	Specifies whether ODBC tracing is enabled (1 or **true**) or disabled (0 or **false**) for the connection being established.	**false** (0)

KEY	VALUE	DESCRIPTION	DEFAULT
TransactionIsolation	The SQLSRV driver uses the following values: SQLSRV_TXN_READ_UNCOMMITTED SQLSRV_TXN_READ_COMMITTED SQLSRV_TXN_REPEATABLE_READ SQLSRV_TXN_SNAPSHOT SQLSRV_TXN_SERIALIZABLE The PDO_SQLSRV driver uses the following values: PDO::SQLSRV_TXN_READ_UNCOMMITTED PDO::SQLSRV_TXN_READ_COMMITTED PDO::SQLSRV_TXN_REPEATABLE_READ PDO::SQLSRV_TXN_SNAPSHOT PDO::SQLSRV_TXN_SERIALIZABLE	Specifies the transaction isolation level. For more information about transaction isolation, see SET TRANSACTION ISOLATION LEVEL in the SQL Server documentation.	SQLSRV_TXN_READ_COMMITTED or PDO::SQLSRV_TXN_READ_COMMITTED
TrustServerCertificate	1 or **true** to trust certificate. 0 or **false** to not trust certificate.	Specifies whether the client should trust (1 or **true**) or reject (0 or **false**) a self-signed server certificate.	**false** (0)
UID (not supported in the PDO_SQLSRV driver)	String	Specifies the User ID to be used when connecting with SQL Server Authentication[3].	No value set.
WSID	String	Specifies the name of the computer for tracing.	No value set.

1. All queries executed on the established connection will be made to the database that is specified by the *Database* attribute. However, data in other databases can be accessed by using a fully-qualified name if the user has the appropriate permissions. For example, if the *master* database is set with the *Database* connection attribute, it is still possible to execute a Transact-SQL query that accesses the *AdventureWorks.HumanResources.Employee* table by using the fully-qualified name.

2. Enabling *Encryption* can impact the performance of some applications due to the computational overhead required to encrypt data.

3. The *UID* and *PWD* attributes must both be set when connecting with SQL Server Authentication.

Many of the supported keys are ODBC connection string attributes. For information about ODBC connection strings, see Using Connection String Keywords with SQL Native Client.

See Also

Connecting to the Server

PHP Driver for SQL Server Support for LocalDB

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Beginning in SQL Server 2012 , a lightweight version of SQL Server , called LocalDB, will be available. This topic discusses how to connect to a database in a LocalDB instance.

Remarks

For more information about LocalDB, including how to install LocalDB and configure your LocalDB instance, see the SQL Server Books Online topic on SQL Server 2012 Express LocalDB.

To summarize, LocalDB allows you to:

- Use **sqllocaldb.exe i** to discover the name of the default instance.
- Use the **AttachDBFilename** connection string keyword to specify which database file the server should attach. When using **AttachDBFilename**, if you do not specify the name of the database with the **Database** connection string keyword, the database will be removed from the LocalDB instance when the application closes.
- Specify a LocalDB instance in your connection string. For example, here is a sample SQLSRV connection string:

```
$conn = sqlsrv_connect( '(localdb)\\v11.0',
    array( 'Database'=>'myData'));

$conn = sqlsrv_connect( '(localdb)\\v11.0',
    array('AttachDBFileName'=>'c:\\myData.MDF','Database'=>'myData'));

$conn = sqlsrv_connect( '(localdb)\\v11.0',
    array('AttachDBFileName'=>'c:\\myData.MDF'));
```

  Next is a sample PDO_SQLSRV connection string:

```
$conn = new PDO( 'sqlsrv:server=(localdb)\\v11.0;'
    . 'Database=myData', NULL, NULL);

$conn = new PDO( 'sqlsrv:server=(localdb)\\v11.0;'
    . 'AttachDBFileName=c:\\myData.MDF;Database=myData ',
    NULL, NULL);

$conn = new PDO( 'sqlsrv:server=(localdb)\\v11.0;'
    . 'AttachDBFileName=c:\\myData.MDF', NULL, NULL);
```

If necessary, you can create a LocalDB instance with sqllocaldb.exe. You can also use sqlcmd.exe to add and modify databases in a LocalDB instance. For example, `sqlcmd -S (localdb)\v11.0`. (When running in IIS, you need to run under the correct account to get the same results as when you run at the command line; see Using LocalDB with Full IIS, Part 2: Instance Ownership for more information.)

The following are example connection strings using the SQLSRV driver that connect to a database in a LocalDB named instance called myInstance:

```
$conn = sqlsrv_connect( '(localdb)\\myInstance',
    array( 'Database'=>'myData'));
```

The following are example connection strings using the PDO_SQLSRV driver that connect to a database in a LocalDB named instance called myInstance:

```
$conn = new PDO( 'sqlsrv:server=(localdb)\\myInstance;'
    . 'database=myData', NULL, NULL);
```

You can download LocalDB from the SQL Server 2012 feature pack page, or from the SQL Server 2012 Express edition. If you will use sqlcmd.exe to modify data in your LocalDB instance, you will need sqlcmd from SQL Server 2012 , which you can get from the Command Line Utilities download in the SQL Server 2012 Feature Pack page.

See Also

Connecting to the Server

PHP Driver for SQL Server Support for High Availability, Disaster Recovery

3/14/2017 • 5 min to read • Edit on GitHub

Download PHP Driver

This topic discusses Microsoft Drivers for PHP for SQL Server support (added in version 3.0) for high-availability, disaster recovery -- AlwaysOn Availability Groups . AlwaysOn Availability Groups support was added in SQL Server 2012 . For more information about AlwaysOn Availability Groups , see SQL Server Books Online.

In version 3.0 of the Microsoft Drivers for PHP for SQL Server, you can specify the availability group listener of a (high-availability, disaster-recovery) availability group (AG) in the connection property. If a Microsoft Drivers for PHP for SQL Server application is connected to an AlwaysOn database that fails over, the original connection is broken and the application must open a new connection to continue work after the failover.

If you are not connecting to an availability group listener, and if multiple IP addresses are associated with a hostname, the Microsoft Drivers for PHP for SQL Server will iterate sequentially through all IP addresses associated with DNS entry. This can be time consuming if the first IP address returned by DNS server is not bound to any network interface card (NIC). When connecting to an availability group listener, the Microsoft Drivers for PHP for SQL Server attempts to establish connections to all IP addresses in parallel and if a connection attempt succeeds, the driver will discard any pending connection attempts.

NOTE

Increasing connection timeout and implementing connection retry logic will increase the probability that an application will connect to an availability group. Also, because a connection can fail because of an availability group failover, you should implement connection retry logic, retrying a failed connection until it reconnects.

Connecting With MultiSubnetFailover

The **MultiSubnetFailover** connection property indicates that the application is being deployed in an availability group or Failover Cluster Instance and that the Microsoft Drivers for PHP for SQL Server will try to connect to the database on the primary SQL Server instance by trying to connect to all the IP addresses. When **MultiSubnetFailover=true** is specified for a connection, the client retries TCP connection attempts faster than the operating system's default TCP retransmit intervals. This enables faster reconnection after failover of either an AlwaysOn Availability Group or an AlwaysOn Failover Cluster Instance, and is applicable to both single- and multi-subnet Availability Groups and Failover Cluster Instances.

Always specify **MultiSubnetFailover=True** when connecting to a SQL Server 2012 availability group listener or SQL Server 2012 Failover Cluster Instance. **MultiSubnetFailover** enables faster failover for all Availability Groups and failover cluster instance in SQL Server 2012 and will significantly reduce failover time for single and multi-subnet AlwaysOn topologies. During a multi-subnet failover, the client will attempt connections in parallel. During a subnet failover, the Microsoft Drivers for PHP for SQL Server will aggressively retry the TCP connection.

For more information about connection string keywords in Microsoft Drivers for PHP for SQL Server, see Connection Options.

Specifying **MultiSubnetFailover=true** when connecting to something other than a availability group listener or Failover Cluster Instance may result in a negative performance impact, and is not supported.

Use the following guidelines to connect to a server in an availability group:

- Use the **MultiSubnetFailover** connection property when connecting to a single subnet or multi-subnet; it will improve performance for both.
- To connect to an availability group, specify the availability group listener of the availability group as the server in your connection string.
- Connecting to a SQL Server instance configured with more than 64 IP addresses will cause a connection failure.
- Behavior of an application that uses the **MultiSubnetFailover** connection property is not affected based on the type of authentication: SQL Server Authentication, Kerberos Authentication, or Windows Authentication.
- Increase the value of **loginTimeout** to accommodate for failover time and reduce application connection retry attempts.
- Distributed transactions are not supported.

If read-only routing is not in effect, connecting to a secondary replica location in an availability group will fail in the following situations:

1. If the secondary replica location is not configured to accept connections.
2. If an application uses **ApplicationIntent=ReadWrite** (discussed below) and the secondary replica location is configured for read-only access.

A connection will fail if a primary replica is configured to reject read-only workloads and the connection string contains **ApplicationIntent=ReadOnly**.

Upgrading to Use Multi-Subnet Clusters from Database Mirroring

A connection error will occur if the **MultiSubnetFailover** and **Failover_Partner** connection keywords are present in the connection string. An error will also occur if **MultiSubnetFailover** is used and the SQL Server returns a failover partner response indicating it is part of a database mirroring pair.

If you upgrade a Microsoft Drivers for PHP for SQL Server application that currently uses database mirroring to a multi-subnet scenario, you should remove the **Failover_Partner** connection property and replace it with **MultiSubnetFailover** set to **Yes** and replace the server name in the connection string with an availability group listener. If a connection string uses **Failover_Partner** and **MultiSubnetFailover=true**, the driver will generate an error. However, if a connection string uses **Failover_Partner** and **MultiSubnetFailover=false** (or **ApplicationIntent=ReadWrite**), the application will use database mirroring.

The driver will return an error if database mirroring is used on the primary database in the AG, and if **MultiSubnetFailover=true** is used in the connection string that connects to a primary database instead of to an availability group listener.

Specifying Application Intent

When **ApplicationIntent=ReadOnly**, the client requests a read workload when connecting to an AlwaysOn enabled database. The server will enforce the intent at connection time and during a USE database statement but only to an Always On enabled database.

The **ApplicationIntent** keyword does not work with legacy, read-only databases.

A database can allow or disallow read workloads on the targeted AlwaysOn database. (This is done with the **ALLOW_CONNECTIONS** clause of the **PRIMARY_ROLE** and **SECONDARY_ROLE**Transact-SQL statements.)

The **ApplicationIntent** keyword is used to enable read-only routing.

Read-Only Routing

Read-only routing is a feature that can ensure the availability of a read only replica of a database. To enable read-only routing:

1. You must connect to an Always On Availability Group availability group listener.
2. The **ApplicationIntent** connection string keyword must be set to **ReadOnly**.
3. The Availability Group must be configured by the database administrator to enable read-only routing.

It is possible that multiple connections using read-only routing will not all connect to the same read-only replica. Changes in database synchronization or changes in the server's routing configuration can result in client connections to different read-only replicas. To ensure that all read-only requests connect to the same read-only replica, do not pass an availability group listener to the **Server** connection string keyword. Instead, specify the name of the read-only instance.

Read-only routing may take longer than connecting to the primary because read only routing first connects to the primary and then looks for the best available readable secondary. Because of this, you should increase your login timeout.

See Also

Connecting to the Server

Connecting to Microsoft Azure SQL Database

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

For more information on connecting to Azure SQL Databases, see How to Access Azure SQL Database from PHP.

Comparing Execution Functions

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The Microsoft Drivers for PHP for SQL Server provides several options for executing functions.

If you are using the SQLSRV driver, use sqlsrv_query to execute a single query and sqlsrv_prepare with sqlsrv_execute to execute a prepared statement multiple times with different parameter values for each execution.

If you are using the PDO_SQLSRV driver, you can execute a query with one of the following:

- PDO::exec
- PDO::query
- PDO::prepare and PDOStatement::execute.

See Also

SQLSRV Driver API Reference
PDO_SQLSRV Driver Reference
Programming Guide for PHP SQL Driver

Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

This topic discusses how you can use the PDO::SQLSRV_ATTR_DIRECT_QUERY attribute to specify direct statement execution instead of the default, which is prepared statement execution. When the driver prepares a statement, it can result in better performance if the statement will be executed more than once using bound parameters.

Remarks

If you want to send a Transact-SQL statement directly to the server without statement preparation by the driver, you can set the PDO::SQLSRV_ATTR_DIRECT_QUERY attribute with PDO::setAttribute (or as a driver option passed to PDO::__construct) or when you call PDO::prepare. By default, the value of PDO::SQLSRV_ATTR_DIRECT_QUERY is False (use prepared statement execution).

If you use PDO::query, you might want direct execution. Before calling PDO::query, call PDO::setAttribute and set PDO::SQLSRV_ATTR_DIRECT_QUERY to True. Each call to PDO::query can be executed with a different setting for PDO::SQLSRV_ATTR_DIRECT_QUERY.

If you use PDO::prepare and PDOStatement::execute to execute a query multiple times using bound parameters, prepared statement execution will optimize execution of the repeated query. In that situation, call PDO::prepare with PDO::SQLSRV_ATTR_DIRECT_QUERY set to False in the driver options array parameter. When necessary, you can execute prepared statements with PDO::SQLSRV_ATTR_DIRECT_QUERY set to False.

After you call PDO::prepare, the value of PDO::SQLSRV_ATTR_DIRECT_QUERY cannot change when executing the prepared query.

If a query requires the context that was set in a previous query, you should execute your queries with PDO::SQLSRV_ATTR_DIRECT_QUERY set to True. For example, if you use temporary tables in your queries, PDO::SQLSRV_ATTR_DIRECT_QUERY must be set to True.

The following sample shows that when context from a previous statement is required, you need to set PDO::SQLSRV_ATTR_DIRECT_QUERY to True. This sample uses temporary tables, which are only available to subsequent statements in your program when queries are executed directly.

```
<?php
   $conn = new PDO('sqlsrv:Server=(local)', '', '');
   $conn->setAttribute(constant('PDO::SQLSRV_ATTR_DIRECT_QUERY'), true);

   $stmt1 = $conn->query("DROP TABLE #php_test_table");

   $stmt2 = $conn->query("CREATE TABLE #php_test_table ([c1_int] int, [c2_int] int)");

   $v1 = 1;
   $v2 = 2;

   $stmt3 = $conn->prepare("INSERT INTO #php_test_table (c1_int, c2_int) VALUES (:var1, :var2)");

   if ($stmt3) {
      $stmt3->bindValue(1, $v1);
      $stmt3->bindValue(2, $v2);

      if ($stmt3->execute())
         echo "Execution succeeded\n";
      else
         echo "Execution failed\n";
   }
   else
      var_dump($conn->errorInfo());

   $stmt4 = $conn->query("DROP TABLE #php_test_table");
?>
```

See Also

Programming Guide for PHP SQL Driver

Retrieving Data

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

This topic and the topics in this section discuss how to retrieve data.

SQLSRV Driver

The SQLSRV driver of the Microsoft Drivers for PHP for SQL Server provides the following options for retrieving data from a result set:

- sqlsrv_fetch_array
- sqlsrv_fetch_object
- sqlsrv_fetch/sqlsrv_get_field

NOTE

When you use any of the functions mentioned above, avoid null comparisons as the criterion for exiting loops. Because **sqlsrv** functions return false when an error occurs, the following code could result in an infinite loop upon an error in sqlsrv_fetch_array:

```
/*``This code could result in an infinite loop. It is recommended that
you do NOT use null comparisons as the criterion for exiting loops,
as is done here. */
do{
$result = sqlsrv_fetch_array($stmt);
} while( !is_null($result));
```

If your query retrieves more than one result set, you can move to the next result set with sqlsrv_next_result.

Beginning in version 1.1 of the Microsoft Drivers for PHP for SQL Server, you can use sqlsrv_has_rows to see if a result set has rows.

PDO_SQLSRV Driver

The PDO_SQLSRV driver of the Microsoft Drivers for PHP for SQL Server provides the following options for retrieving data from a result set:

- PDOStatement::fetch
- PDOStatement::fetchAll
- PDOStatement::fetchColumn
- PDOStatement::fetchObject

If your query retrieves more than one result set, you can move to the next result set with PDOStatement::nextRowset.

You can see how many rows are in a result set if you specify a scrollable cursor, and then call PDOStatement::rowCount.

PDO::prepare lets you specify a cursor type. Then, with PDOStatement::fetch you can select a row. See PDO::prepare for a sample and more information.

In This Section

TOPIC	DESCRIPTION
Retrieving Data as a Stream	Provides an overview of how to stream data from the server, and provides links to specific use cases.
Using Directional Parameters	Describes how to use directional parameters when calling a stored procedure.
Specifying a Cursor Type and Selecting Rows	Demonstrates how to create a result set with rows that you can access in any order when using the SQLSRV driver.
How to: Retrieve Date and Time Type as Strings Using the SQLSRV Driver	Describes how to retrieve date and time types as strings.

Related Sections

How to: Specify PHP Data Types

See Also

Programming Guide for PHP SQL Driver Retrieving Data

Retrieving Data as a Stream Using the SQLSRV Driver

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Retrieving data as a stream is only available in the SQLSRV driver of the Microsoft Drivers for PHP for SQL Server, and is not available in the PDO_SQLSRV driver.

The Microsoft Drivers for PHP for SQL Server takes advantage of streams for retrieving large amounts of data. The topics in this section provide details about how to retrieve data as a stream.

The following steps summarize how to retrieve data as a stream:

1. Prepare and execute a Transact-SQL query with sqlsrv_query or the combination of sqlsrv_prepare/sqlsrv_execute.

2. Use sqlsrv_fetch to move to the next row in the result set.

3. Use sqlsrv_get_field to retrieve a field from the row. Specify that the data is to be retrieved as a stream by using **SQLSRV_PHPTYPE_STREAM()** as the third parameter in the function call. This table lists the constants used to specify encodings and their descriptions:

SQLSRV CONSTANT	DESCRIPTION
SQLSRV_ENC_BINARY	Data is returned as a raw byte stream from the server without performing encoding or translation.
SQLSRV_ENC_CHAR	Data is returned in 8-bit characters as specified in the code page of the Windows locale set on the system. Any multi-byte characters or characters that do not map into this code page are substituted with a single byte question mark (?) character.

NOTE

Some data types are returned as streams by default. For more information, see Default PHP Data Types.

In This Section

TOPIC	DESCRIPTION
Data Types with Stream Support Using the SQLSRV Driver	Lists the SQL Server data types that can be retrieved as streams.
How to: Retrieve Character Data as a Stream Using the SQLSRV Driver	Demonstrates how to retrieve character data as a stream.
How to: Retrieve Binary Data as a Stream Using the SQLSRV Driver	Demonstrates how to retrieve binary data as a stream.

See Also

Retrieving Data
Constants (Microsoft Drivers for PHP for SQL Server)

Data Types with Stream Support Using the SQLSRV Driver

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Retrieving data as a stream is only available in the SQLSRV driver of the Microsoft Drivers for PHP for SQL Server, and is not available in the PDO_SQLSRV driver.

The following SQL Server data types can be retrieved as streams with the SQLSRV driver:

- binary
- char
- image
- nchar
- ntext
- nvarchar
- text
- UDT
- varbinary
- varchar
- XML

See Also

Retrieving Data as a Stream Using the SQLSRV Driver
Default PHP Data Types
How to: Specify PHP Data Types

How to: Retrieve Character Data as a Stream Using the SQLSRV Driver

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

Retrieving data as a stream is only available in the SQLSRV driver of the Microsoft Drivers for PHP for SQL Server, and is not available in the PDO_SQLSRV driver.

The SQLSRV driver takes advantage of PHP streams for retrieving large amounts of data from the server. The example in this topic demonstrates how to retrieve character data as a stream.

Example

The following example retrieves a row from the *Production.ProductReview* table of the AdventureWorks database. The *Comments* field of the returned row is retrieved as a stream and displayed by using the PHP fpassthru function.

Retrieving data as a stream is accomplished by using sqlsrv_fetch and sqlsrv_get_field with the return type specified as a character stream. The return type is specified by using the constant **SQLSRV_PHPTYPE_STREAM**. For information about **sqlsrv** constants, see Constants (Microsoft Drivers for PHP for SQL Server).

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/*Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up the Transact-SQL query. */
$tsql = "SELECT ReviewerName,
               CONVERT(varchar(32), ReviewDate, 107) AS [ReviewDate],
               Rating,
               Comments
         FROM Production.ProductReview
         WHERE ProductReviewID = ? ";

/* Set the parameter value. */
$productReviewID = 1;
$params = array( $productReviewID);

/* Execute the query. */
$stmt = sqlsrv_query($conn, $tsql, $params);
if( $stmt === false )
{
     echo "Error in statement execution.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve and display the data. The first three fields are retrieved
as strings and the fourth as a stream with character encoding. */
if(sqlsrv_fetch( $stmt ) === false )
{
     echo "Error in retrieving row.\n";
     die( print_r( sqlsrv_errors(), true));
}

echo "Name: ".sqlsrv_get_field( $stmt, 0 )."\n";
echo "Date: ".sqlsrv_get_field( $stmt, 1 )."\n";
echo "Rating: ".sqlsrv_get_field( $stmt, 2 )."\n";
echo "Comments: ";
$comments = sqlsrv_get_field( $stmt, 3,
                             SQLSRV_PHPTYPE_STREAM(SQLSRV_ENC_CHAR));
fpassthru($comments);

/* Free the statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

Because no PHP return type is specified for the first three fields, each field is returned according to its default PHP type. For information about default PHP data types, see Default PHP Data Types. For information about how to specify PHP return types, see How to: Specify PHP Data Types.

See Also

Retrieving Data
Retrieving Data as a Stream Using the SQLSRV Driver
About Code Examples in the Documentation

How to: Retrieve Binary Data as a Stream Using the SQLSRV Driver

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

Retrieving data as a stream is only available in the SQLSRV driver of the Microsoft Drivers for PHP for SQL Server, and is not available in the PDO_SQLSRV driver.

The Microsoft Drivers for PHP for SQL Server takes advantage of PHP streams for retrieving large amounts of binary data from the server. This topic demonstrates how to retrieve binary data as a stream.

Using the streams to retrieve binary data, such as images, avoids using large amounts of script memory by retrieving chunks of data instead of loading the whole object into script memory.

Example

The following example retrieves binary data, an image in this case, from the *Production.ProductPhoto* table of the AdventureWorks database. The image is retrieved as a stream and displayed in the browser.

Retrieving image data as a stream is accomplished by using sqlsrv_fetch and sqlsrv_get_field with the return type specified as a binary stream. The return type is specified by using the constant **SQLSRV_PHPTYPE_STREAM**. For information about **sqlsrv** constants, see Constants (Microsoft Drivers for PHP for SQL Server).

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the browser when the example is run from the browser.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up the Transact-SQL query. */
$tsql = "SELECT LargePhoto
         FROM Production.ProductPhoto
         WHERE ProductPhotoID = ?";

/* Set the parameter values and put them in an array. */
$productPhotoID = 70;
$params = array( $productPhotoID);

/* Execute the query. */
$stmt = sqlsrv_query($conn, $tsql, $params);
if( $stmt === false )
{
     echo "Error in statement execution.</br>";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve and display the data.
The return data is retrieved as a binary stream. */
if ( sqlsrv_fetch( $stmt ) )
{
   $image = sqlsrv_get_field( $stmt, 0,
                      SQLSRV_PHPTYPE_STREAM(SQLSRV_ENC_BINARY));
   header("Content-Type: image/jpg");
   fpassthru($image);
}
else
{
     echo "Error in retrieving data.</br>";
     die(print_r( sqlsrv_errors(), true));
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

Specifying the return type in the example demonstrates how to specify the PHP return type as a binary stream. Technically, it is not required in the example because the *LargePhoto* field has SQL Server type varbinary(max) and is therefore returned as a binary stream by default. For information about default PHP data types, see Default PHP Data Types. For information about how to specify PHP return types, see How to: Specify PHP Data Types.

See Also

Retrieving Data
Retrieving Data as a Stream Using the SQLSRV Driver
About Code Examples in the Documentation

Using Directional Parameters

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

When using the PDO_SQLSRV driver, you can use PDOStatement::bindParam to specify input and output parameters.

The topics in this section describe how to use directional parameters when calling stored procedures using the SQLSRV driver.

In This Section

TOPIC	DESCRIPTION
How to: Specify Parameter Direction Using the SQLSRV Driver	Demonstrates how to specify parameter direction when calling a stored procedure.
How to: Retrieve Output Parameters Using the SQLSRV Driver	Demonstrates how to call a stored procedure with an output parameter and how to retrieve its value.
How to: Retrieve Input and Output Parameters Using the SQLSRV Driver	Demonstrates how to call a stored procedure with an input/output parameter and how to retrieve its value.

See Also

Retrieving Data
Updating Data (Microsoft Drivers for PHP for SQL Server)

How to: Specify Parameter Direction Using the SQLSRV Driver

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

This topic describes how to use the SQLSRV driver to specify parameter direction when you call a stored procedure. Note that the parameter direction is specified when you construct a parameter array (step 3) that is passed to sqlsrv_query or sqlsrv_prepare.

To specify parameter direction

1. Define a Transact-SQL query that calls a stored procedure. Use question marks (?) instead of the parameters to be passed to the stored procedure. For example, this string calls a stored procedure (UpdateVacationHours) that accepts two parameters:

```
$tsql = "{call UpdateVacationHours(?, ?)}";
```

> **NOTE**
> Calling stored procedures using canonical syntax is the recommended practice. For more information about canonical syntax, see Calling a Stored Procedure.

2. Initialize or update PHP variables that correspond to the placeholders in the Transact-SQL query. For example, the following code initializes the two parameters for the UpdateVacationHours stored procedure:

```
$employeeId = 101;
$usedVacationHours = 8;
```

> **NOTE**
> Variables that are initialized or updated to **null**, **DateTime**, or stream types cannot be used as output parameters.

3. Use your PHP variables from step 2 to create or update an array of parameter values that correspond, in order, to the parameter placeholders in the Transact-SQL string. Specify the direction for each parameter in the array. The direction of each parameter is determined in one of two ways: by default (for input parameters) or by using **SQLSRV_PARAM_*** constants (for output and bidirectional parameters). For example, the following code specifies the *$employeeId* parameter as an input parameter and the *$usedVacationHours* parameter as a bidirectional parameter:

```
$params = array(
                 array($employeeId, SQLSRV_PARAM_IN),
                 array($usedVacationHours, SQLSRV_PARAM_INOUT)
                );
```

To understand the syntax for specifying parameter direction in general, suppose that *$var1*, *$var2*, and *$var3* correspond to input, output, and bidirectional parameters, respectively. You can specify the parameter direction in either of the following ways:

- Implicitly specify the input parameter, explicitly specify the output parameter, and explicitly specify a bidirectional parameter:

  ```
  array(
          array($var1),
          array($var2, SQLSRV_PARAM_OUT),
          array($var3, SQLSRV_PARAM_INOUT)
          );
  ```

- Explicitly specificy the input parameter, explicitly specify the output parameter, and explicitly specificy a bidirectional parameter:

  ```
  array(
          array($var1, SQLSRV_PARAM_IN),
          array($var2, SQLSRV_PARAM_OUT),
          array($var3, SQLSRV_PARAM_INOUT)
          );
  ```

4. Execute the query with sqlsrv_query or with sqlsrv_prepare and sqlsrv_execute. For example, the following code uses the connection *$conn* to execute the query *$tsql* with parameter values specified in *$params*:

   ```
   sqlsrv_query($conn, $tsql, $params);
   ```

See Also

How to: Retrieve Output Parameters Using the SQLSRV Driver
How to: Retrieve Input and Output Parameters Using the SQLSRV Driver

How to: Retrieve Output Parameters Using the SQLSRV Driver

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

This topic demonstrates how to call a stored procedure in which one parameter has been defined as an output parameter. Note that when retrieving an output or input/output parameter, all results returned by the stored procedure must be consumed before the returned parameter value is accessible.

NOTE

Variables that are initialized or updated to **null**, **DateTime**, or stream types cannot be used as output parameters.

Data truncation can occur when stream types such as SQLSRV_SQLTYPE_VARCHAR('max') are used as output parameters. Stream types are not supported as output parameters. For non-stream types, data truncation can occur if the length of the output parameter is not specified or if the specified length is not sufficiently large for the output parameter.

Example

The following example calls a stored procedure that returns the year-to-date sales by a specified employee. The PHP variable *$lastName* is an input parameter and *$salesYTD* is an output parameter.

NOTE

Initializing *$salesYTD* to 0.0 sets the returned PHPTYPE to **float**. To ensure data type integrity, output parameters should be initialized before calling the stored procedure, or the desired PHPTYPE should be specified. For information about specifying the PHPTYPE, see How to: Specify PHP Data Types.

Because only one result is returned by the stored procedure, *$salesYTD* contains the returned value of the output parameter immediately after the stored procedure is executed.

NOTE

Calling stored procedures using canonical syntax is the recommended practice. For more information about canonical syntax, see Calling a Stored Procedure.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Drop the stored procedure if it already exists. */
$tsql_dropSP = "IF OBJECT_ID('GetEmployeeSalesYTD', 'P') IS NOT NULL
                DROP PROCEDURE GetEmployeeSalesYTD";
$stmt1 = sqlsrv_query( $conn, $tsql_dropSP);
if( $stmt1 === false )
{
     echo "Error in executing statement 1.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Create the stored procedure. */
$tsql_createSP = " CREATE PROCEDURE GetEmployeeSalesYTD
                   @SalesPerson nvarchar(50),
                   @SalesYTD money OUTPUT
                   AS
                   SELECT @SalesYTD = SalesYTD
                   FROM Sales.SalesPerson AS sp
                   JOIN HumanResources.vEmployee AS e
                   ON e.EmployeeID = sp.SalesPersonID
                   WHERE LastName = @SalesPerson";
$stmt2 = sqlsrv_query( $conn, $tsql_createSP);
if( $stmt2 === false )
{
     echo "Error in executing statement 2.\n";
     die( print_r( sqlsrv_errors(), true));
}

/*--------- The next few steps call the stored procedure. ---------*/

/* Define the Transact-SQL query. Use question marks (?) in place of
 the parameters to be passed to the stored procedure */
$tsql_callSP = "{call GetEmployeeSalesYTD( ?, ? )}";

/* Define the parameter array. By default, the first parameter is an
INPUT parameter. The second parameter is specified as an OUTPUT
parameter. Initializing $salesYTD to 0.0 sets the returned PHPTYPE to
float. To ensure data type integrity, output parameters should be
initialized before calling the stored procedure, or the desired
PHPTYPE should be specified in the $params array.*/
$lastName = "Blythe";
$salesYTD = 0.0;
$params = array(
                 array($lastName, SQLSRV_PARAM_IN),
                 array($salesYTD, SQLSRV_PARAM_OUT)
               );

/* Execute the query. */
$stmt3 = sqlsrv_query( $conn, $tsql_callSP, $params);
if( $stmt3 === false )
{
     echo "Error in executing statement 3.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Display the value of the output parameter $salesYTD. */
echo "YTD sales for ".$lastName." are ". $salesYTD. ".";

/*Free the statement and connection resources. */
sqlsrv_free_stmt( $stmt1);
sqlsrv_free_stmt( $stmt2);
sqlsrv_free_stmt( $stmt3);
sqlsrv_close( $conn);
?>
```

See Also

How to: Specify Parameter Direction Using the SQLSRV Driver
How to: Retrieve Input and Output Parameters Using the SQLSRV Driver
Retrieving Data

How to: Retrieve Input and Output Parameters Using the SQLSRV Driver

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

This topic demonstrates how to use the SQLSRV driver to call a stored procedure in which one parameter has been defined as an input/output parameter, and how to retrieve the results. Note that when retrieving an output or input/output parameter, all results returned by the stored procedure must be consumed before the returned parameter value is accessible.

NOTE

Variables that are initialized or updated to **null**, **DateTime**, or stream types cannot be used as output parameters.

Example

The following example calls a stored procedure that subtracts used vacation hours from the available vacation hours of a specified employee. The variable that represents used vacation hours, *$vacationHrs*, is passed to the stored procedure as an input parameter. After updating the available vacation hours, the stored procedure uses the same parameter to return the number of remaining vacation hours.

NOTE

Initializing *$vacationHrs* to 4 sets the returned PHPTYPE to integer. To ensure data type integrity, input/output parameters should be initialized before calling the stored procedure, or the desired PHPTYPE should be specified. For information about specifying the PHPTYPE, see How to: Specify PHP Data Types.

Because the stored procedure returns two results, sqlsrv_next_result must be called after the stored procedure has been executed to make the value of the output parameter available. After calling **sqlsrv_next_result**, *$vacationHrs* contains the value of the output parameter returned by the stored procedure.

NOTE

Calling stored procedures using canonical syntax is the recommended practice. For more information about canonical syntax, see Calling a Stored Procedure.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Drop the stored procedure if it already exists. */
$tsql_dropSP = "IF OBJECT_ID('SubtractVacationHours', 'P') IS NOT NULL
                DROP PROCEDURE SubtractVacationHours";
$stmt1 = sqlsrv_query( $conn, $tsql_dropSP);
if( $stmt1 === false )
{
     echo "Error in executing statement 1.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Create the stored procedure. */
$tsql_createSP = "CREATE PROCEDURE SubtractVacationHours
                        @EmployeeID int,
                        @VacationHrs smallint OUTPUT
                  AS
                  UPDATE HumanResources.Employee
                  SET VacationHours = VacationHours - @VacationHrs
                  WHERE EmployeeID = @EmployeeID;
                  SET @VacationHrs = (SELECT VacationHours
                                      FROM HumanResources.Employee
                                      WHERE EmployeeID = @EmployeeID)";

$stmt2 = sqlsrv_query( $conn, $tsql_createSP);
if( $stmt2 === false )
{
     echo "Error in executing statement 2.\n";
     die( print_r( sqlsrv_errors(), true));
}

/*--------- The next few steps call the stored procedure. ---------*/

/* Define the Transact-SQL query. Use question marks (?) in place of
the parameters to be passed to the stored procedure */
$tsql_callSP = "{call SubtractVacationHours( ?, ?)}";

/* Define the parameter array. By default, the first parameter is an
INPUT parameter. The second parameter is specified as an INOUT
parameter. Initializing $vacationHrs to 8 sets the returned PHPTYPE to
integer. To ensure data type integrity, output parameters should be
initialized before calling the stored procedure, or the desired
PHPTYPE should be specified in the $params array.*/

$employeeId = 4;
$vacationHrs = 8;
$params = array(
                 array($employeeId, SQLSRV_PARAM_IN),
                 array($vacationHrs, SQLSRV_PARAM_INOUT)
               );

/* Execute the query. */
$stmt3 = sqlsrv_query( $conn, $tsql_callSP, $params);
if( $stmt3 === false )
{
     echo "Error in executing statement 3.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Display the value of the output parameter $vacationHrs. */
sqlsrv_next_result($stmt3);
echo "Remaining vacation hours: ".$vacationHrs;

/*Free the statement and connection resources. */
sqlsrv_free_stmt( $stmt1);
sqlsrv_free_stmt( $stmt2);
sqlsrv_free_stmt( $stmt3);
sqlsrv_close( $conn);
?>
```

See Also

How to: Specify Parameter Direction Using the SQLSRV Driver
How to: Retrieve Output Parameters Using the SQLSRV Driver
Retrieving Data

Specifying a Cursor Type and Selecting Rows

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

You can create a result set with rows that you can access in any order, depending on the cursor type. This section discusses client-side and server-side cursors:

- Cursor Types (SQLSRV Driver)
- Cursor Types (PDO_SQLSRV Driver)

See Also

Retrieving Data

Cursor Types (SQLSRV Driver)

3/14/2017 • 6 min to read • Edit on GitHub

Download PHP Driver

The SQLSRV driver lets you create a result set with rows that you can access in any order, depending on the cursor type. This topic will discuss client-side (buffered) and server-side (unbuffered) cursors.

Cursor Types

When you create a result set with sqlsrv_query or with sqlsrv_prepare, you can specify the type of cursor. By default, a forward-only cursor is used, which lets you move one row at a time starting at the first row of the result set until you reach the end of the result set.

You can create a result set with a scrollable cursor, which allows you to access any row in the result set, in any order. The following table lists the values that can be passed to the **Scrollable** option in sqlsrv_query or sqlsrv_prepare.

OPTION	DESCRIPTION
SQLSRV_CURSOR_FORWARD	Lets you move one row at a time starting at the first row of the result set until you reach the end of the result set. This is the default cursor type. sqlsrv_num_rows returns an error for result sets created with this cursor type. **forward** is the abbreviated form of SQLSRV_CURSOR_FORWARD.
SQLSRV_CURSOR_STATIC	Lets you access rows in any order but will not reflect changes in the database. **static** is the abbreviated form of SQLSRV_CURSOR_STATIC.
SQLSRV_CURSOR_DYNAMIC	Lets you access rows in any order and will reflect changes in the database. sqlsrv_num_rows returns an error for result sets created with this cursor type. **dynamic** is the abbreviated form of SQLSRV_CURSOR_DYNAMIC.
SQLSRV_CURSOR_KEYSET	Lets you access rows in any order. However, a keyset cursor does not update the row count if a row is deleted from the table (a deleted row is returned with no values). **keyset** is the abbreviated form of SQLSRV_CURSOR_KEYSET.

OPTION	DESCRIPTION
SQLSRV_CURSOR_CLIENT_BUFFERED	Lets you access rows in any order. Creates a client-side cursor query. **buffered** is the abbreviated form of SQLSRV_CURSOR_CLIENT_BUFFERED.

If a query generates multiple result sets, the **Scrollable** option applies to all result sets.

Selecting Rows in a Result Set

After you create a result set, you can use sqlsrv_fetch, sqlsrv_fetch_array, or sqlsrv_fetch_object to specify a row.

The following table describes the values you can specify in the *row* parameter.

PARAMETER	DESCRIPTION
SQLSRV_SCROLL_NEXT	Specifies the next row. This is the default value, if you do not specify the *row* parameter for a scrollable result set.
SQLSRV_SCROLL_PRIOR	Specifies the row before the current row.
SQLSRV_SCROLL_FIRST	Specifies the first row in the result set.
SQLSRV_SCROLL_LAST	Specifies the last row in the result set.
SQLSRV_SCROLL_ABSOLUTE	Specifies the row specified with the *offset* parameter.
SQLSRV_SCROLL_RELATIVE	Specifies the row specified with the *offset* parameter from the current row.

Server-Side Cursors and the SQLSRV Driver

The following example shows the effect of the various cursors. On line 33 of the example, you see the first of three query statements that specify different cursors. Two of the query statements are commented. Each time you run the program, use a different cursor type to see the effect of the database update on line 47.

```
<?php
$server = "server_name";
$conn = sqlsrv_connect( $server, array( 'Database' => 'test' ));
if ( $conn === false ) {
   die( print_r( sqlsrv_errors(), true ));
}

$stmt = sqlsrv_query( $conn, "DROP TABLE dbo.ScrollTest" );
if ( $stmt !== false ) {
   sqlsrv_free_stmt( $stmt );
}

$stmt = sqlsrv_query( $conn, "CREATE TABLE ScrollTest (id int, value char(10))" );
if ( $stmt === false ) {
   die( print_r( sqlsrv_errors(), true ));
}

$stmt = sqlsrv_query( $conn, "INSERT INTO ScrollTest (id, value) VALUES(?,?)", array( 1, "Row 1" ));
if ( $stmt === false ) {
   die( print_r( sqlsrv_errors(), true ));
}

$stmt = sqlsrv_query( $conn, "INSERT INTO ScrollTest (id, value) VALUES(?,?)", array( 2, "Row 2" ));
if ( $stmt === false ) {
   die( print_r( sqlsrv_errors(), true ));
}

$stmt = sqlsrv_query( $conn, "INSERT INTO ScrollTest (id, value) VALUES(?,?)", array( 3, "Row 3" ));
if ( $stmt === false ) {
   die( print_r( sqlsrv_errors(), true ));
}

$stmt = sqlsrv_query( $conn, "SELECT * FROM ScrollTest", array(), array( "Scrollable" => 'keyset' ));
// $stmt = sqlsrv_query( $conn, "SELECT * FROM ScrollTest", array(), array( "Scrollable" => 'dynamic' ));
// $stmt = sqlsrv_query( $conn, "SELECT * FROM ScrollTest", array(), array( "Scrollable" => 'static' ));

$rows = sqlsrv_has_rows( $stmt );
if ( $rows != true ) {
   die( "Should have rows" );
}

$result = sqlsrv_fetch( $stmt, SQLSRV_SCROLL_LAST );
$field1 = sqlsrv_get_field( $stmt, 0 );
$field2 = sqlsrv_get_field( $stmt, 1 );
echo "\n$field1 $field2\n";

$stmt2 = sqlsrv_query( $conn, "delete from ScrollTest where id = 3" );
// or
// $stmt2 = sqlsrv_query( $conn, "UPDATE ScrollTest SET id = 4 WHERE id = 3" );
if ( $stmt2 !== false ) {
   sqlsrv_free_stmt( $stmt2 );
}

$result = sqlsrv_fetch( $stmt, SQLSRV_SCROLL_LAST );
$field1 = sqlsrv_get_field( $stmt, 0 );
$field2 = sqlsrv_get_field( $stmt, 1 );
echo "\n$field1 $field2\n";

sqlsrv_free_stmt( $stmt );
sqlsrv_close( $conn );
?>
```

Client-Side Cursors and the SQLSRV Driver

Client-side cursors are a feature added in version 3.0 of the Microsoft Drivers for PHP for SQL Server that allows you to cache an entire result set in memory. Row count is available after the query is executed when using a client-side cursor.

Client-side cursors should be used for small- to medium-sized result sets. Use server-side cursors for large result sets.

A query will return false if the buffer is not large enough to hold the entire result set. You can increase the buffer size up to the PHP memory limit.

Using the SQLSRV driver, you can configure the size of the buffer that holds the result set with the ClientBufferMaxKBSize setting for sqlsrv_configure. sqlsrv_get_config returns the value of ClientBufferMaxKBSize. You can also set the maximum buffer size in the php.ini file with sqlsrv.ClientBufferMaxKBSize (for example, sqlsrv.ClientBufferMaxKBSize = 1024).

The following sample shows:

- Row count is always available with a client-side cursor.
- Use of client-side cursors and batch statements.

```php
<?php
$serverName = "(local)";
$connectionInfo = array("Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);

if ( $conn === false ) {
   echo "Could not connect.\n";
   die( print_r( sqlsrv_errors(), true));
}

$tsql = "select * from HumanResources.Department";

// Execute the query with client-side cursor.
$stmt = sqlsrv_query($conn, $tsql, array(), array("Scrollable"=>"buffered"));
if (! $stmt) {
   echo "Error in statement execution.\n";
   die( print_r( sqlsrv_errors(), true));
}

// row count is always available with a client-side cursor
$row_count = sqlsrv_num_rows( $stmt );
echo "\nRow count = $row_count\n";

// Move to a specific row in the result set.
$row = sqlsrv_fetch($stmt, SQLSRV_SCROLL_FIRST);
$EmployeeID = sqlsrv_get_field( $stmt, 0);
echo "Employee ID = $EmployeeID \n";

// Client-side cursor and batch statements
$tsql = "select top 2 * from HumanResources.Employee;Select top 3 * from HumanResources.EmployeeAddress";

$stmt = sqlsrv_query($conn, $tsql, array(), array("Scrollable"=>"buffered"));
if (! $stmt) {
   echo "Error in statement execution.\n";
   die( print_r( sqlsrv_errors(), true));
}

$row_count = sqlsrv_num_rows( $stmt );
echo "\nRow count for first result set = $row_count\n";

$row = sqlsrv_fetch($stmt, SQLSRV_SCROLL_FIRST);
$EmployeeID = sqlsrv_get_field( $stmt, 0);
echo "Employee ID = $EmployeeID \n";

sqlsrv_next_result($stmt);

$row_count = sqlsrv_num_rows( $stmt );
echo "\nRow count for second result set = $row_count\n";

$row = sqlsrv_fetch($stmt, SQLSRV_SCROLL_LAST);
$EmployeeID = sqlsrv_get_field( $stmt, 0);
echo "Employee ID = $EmployeeID \n";
?>
```

The following sample shows a client-side cursor using sqlsrv_prepare.

```
<?php
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);

if ( $conn === false ) {
   echo "Could not connect.\n";
   die( print_r( sqlsrv_errors(), true));
}

$tsql = "select * from HumanResources.Employee";
$stmt = sqlsrv_prepare( $conn, $tsql, array(), array("Scrollable"=>SQLSRV_CURSOR_CLIENT_BUFFERED));

if (! $stmt ) {
   echo "Statement could not be prepared.\n";
   die( print_r( sqlsrv_errors(), true));
}

sqlsrv_execute( $stmt);

$row_count = sqlsrv_num_rows( $stmt );
if ($row_count)
   echo "\nRow count = $row_count\n";

$row = sqlsrv_fetch($stmt, SQLSRV_SCROLL_FIRST);
if ($row ) {
   $EmployeeID = sqlsrv_get_field( $stmt, 0);
   echo "Employee ID = $EmployeeID \n";
}
?>
```

See Also

Specifying a Cursor Type and Selecting Rows

Cursor Types (PDO_SQLSRV Driver)

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

The PDO_SQLSRV driver lets you create scrollable result sets with one of several cursors.

For information on how to specify a cursor using the PDO_SQLSRV driver, and for code samples, see PDO::prepare.

PDO_SQLSRV and Server-Side Cursors

Prior to version 3.0 of the Microsoft Drivers for PHP for SQL Server, the PDO_SQLSRV driver allowed you to create a result set with a server-side forward-only or static cursor. Beginning in version 3.0 of the Microsoft Drivers for PHP for SQL Server, keyset and dynamic cursors are also available.

You can indicate the type of server-side cursor by using PDO::prepare or PDOStatement::setAttribute to select either cursor type:

- PDO::ATTR_CURSOR => PDO::CURSOR_FWDONLY
- PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL

You can request a keyset or dynamic cursor by specifying PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL and then pass the appropriate value to PDO::SQLSRV_ATTR_CURSOR_SCROLL_TYPE. Possible values that you can pass to PDO::SQLSRV_ATTR_CURSOR_SCROLL_TYPE are:

- PDO::SQLSRV_CURSOR_BUFFERED
- PDO::SQLSRV_CURSOR_DYNAMIC
- PDO::SQLSRV_CURSOR_KEYSET_DRIVEN
- PDO::SQLSRV_CURSOR_STATIC

PDO_SQLSRV and Client-Side Cursors

Client-side cursors were added in version 3.0 of the Microsoft Drivers for PHP for SQL Server that allows you to cache an entire result set in memory. One advantage is that row count is available after a query is executed.

Client-side cursors should be used for small- to medium-sized result sets. Large result sets should use server-side cursors.

A query will return false if the buffer is not large enough to hold an entire result set when using a client-side cursor. You can increase the buffer size up to the PHP memory limit.

You can configure the size of the buffer that holds the result set with the PDO::SQLSRV_ATTR_CLIENT_BUFFER_MAX_KB_SIZE attribute of PDO::setAttribute or PDOStatement::setAttribute. You can also set the maximum buffer size in the php.ini file with pdo_sqlsrv.client_buffer_max_kb_size (for example, pdo_sqlsrv.client_buffer_max_kb_size = 1024).

You indicate that you want a client-side cursor by using PDO::prepare or PDOStatement::setAttribute and select the PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL cursor type. You then specify PDO::SQLSRV_ATTR_CURSOR_SCROLL_TYPE => PDO::SQLSRV_CURSOR_BUFFERED.

```
<?php
$database = "AdventureWorks";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$query = "select * from Person.ContactType";
$stmt = $conn->prepare( $query, array(PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL,
PDO::SQLSRV_ATTR_CURSOR_SCROLL_TYPE => PDO::SQLSRV_CURSOR_BUFFERED));
$stmt->execute();
print $stmt->rowCount();

echo "\n";

while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
   print "$row[Name]\n";
}
echo "\n..\n";

$row = $stmt->fetch( PDO::FETCH_BOTH, PDO::FETCH_ORI_FIRST );
print_r($row);

$row = $stmt->fetch( PDO::FETCH_ASSOC, PDO::FETCH_ORI_REL, 1 );
print "$row[Name]\n";

$row = $stmt->fetch( PDO::FETCH_NUM, PDO::FETCH_ORI_NEXT );
print "$row[1]\n";

$row = $stmt->fetch( PDO::FETCH_NUM, PDO::FETCH_ORI_PRIOR );
print "$row[1]..\n";

$row = $stmt->fetch( PDO::FETCH_NUM, PDO::FETCH_ORI_ABS, 0 );
print_r($row);

$row = $stmt->fetch( PDO::FETCH_NUM, PDO::FETCH_ORI_LAST );
print_r($row);
?>
```

See Also

Specifying a Cursor Type and Selecting Rows

How to: Retrieve Date and Time Type as Strings Using the SQLSRV Driver

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

This feature was added in version 1.1 of the Microsoft Drivers for PHP for SQL Server and is only valid when using the SQLSRV driver for the Microsoft Drivers for PHP for SQL Server. It is an error to use the ReturnDatesAsStrings connection option with the PDO_SQLSRV driver.

You can retrieve date and time types (**datetime**, **date**, **time**, **datetime2**, and **datetimeoffset**) as strings by specifying an option in the connection string.

To retrieve date and time types as strings

- Use the following connection option:

  ```
  'ReturnDatesAsStrings'=>true
  ```

  The default is **false**, which means that **datetime**, **Date**, **Time**, **DateTime2**, and **DateTimeOffset** types will be returned as PHP **Datetime** types.

Example

The following example shows the syntax specifying to retrieve date and time types as strings.

```
<?php
$serverName = "MyServer";
$connectionInfo = array( "Database"=>"AdventureWorks", 'ReturnDatesAsStrings '=> true);
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
   echo "Could not connect.\n";
   die( print_r( sqlsrv_errors(), true));
}

sqlsrv_close( $conn);
?>
```

Example

The following example shows that you can retrieve dates as strings by specifying UTF-8 when you retrieve the string, even when the connection was made with `"ReturnDatesAsStrings" => false`.

```
<?php
$serverName = "MyServer";
$connectionInfo = array( "Database"=>"AdventureWorks", "ReturnDatesAsStrings" => false);
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false ) {
   echo "Could not connect.\n";
   die( print_r( sqlsrv_errors(), true));
}

$tsql = "SELECT VersionDate FROM AWBuildVersion";

$stmt = sqlsrv_query( $conn, $tsql);

if ( $stmt === false ) {
   echo "Error in statement preparation/execution.\n";
   die( print_r( sqlsrv_errors(), true));
}

sqlsrv_fetch( $stmt );

// retrieve date as string
$date = sqlsrv_get_field( $stmt, 0, SQLSRV_PHPTYPE_STRING("UTF-8"));

if( $date === false ) {
   die( print_r( sqlsrv_errors(), true ));
}

echo $date;

sqlsrv_close( $conn);
?>
```

Example

The following example shows how to retrieve dates as strings by specifying UTF-8 and `"ReturnDatesAsStrings" => true` in the connection string.

```
<?php
$serverName = "MyServer";
$connectionInfo = array( "Database"=>"AdventureWorks", 'ReturnDatesAsStrings'=> true, "CharacterSet" => 'utf-
8' );
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false ) {
   echo "Could not connect.\n";
   die( print_r( sqlsrv_errors(), true));
}

$tsql = "SELECT VersionDate FROM AWBuildVersion";

$stmt = sqlsrv_query( $conn, $tsql);

if ( $stmt === false ) {
   echo "Error in statement preparation/execution.\n";
   die( print_r( sqlsrv_errors(), true));
}

sqlsrv_fetch( $stmt );

// retrieve date as string
$date = sqlsrv_get_field( $stmt, 0 );

if ( $date === false ) {
   die( print_r( sqlsrv_errors(), true ));
}

echo $date;
sqlsrv_close( $conn);
?>
```

Example

The following example shows how to retrieve the date as a PHP type. `'ReturnDatesAsStrings'=> false` is on by default.

```
<?php
$serverName = "MyServer";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false ) {
   echo "Could not connect.\n";
   die( print_r( sqlsrv_errors(), true));
}

$tsql = "SELECT VersionDate FROM AWBuildVersion";

$stmt = sqlsrv_query( $conn, $tsql);

if ( $stmt === false ) {
   echo "Error in statement preparation/execution.\n";
   die( print_r( sqlsrv_errors(), true));
}

sqlsrv_fetch( $stmt );

// retrieve date as string
$date = sqlsrv_get_field( $stmt, 0 );

if ( $date === false ) {
   die( print_r( sqlsrv_errors(), true ));
}

$date_string = date_format( $date, 'jS, F Y' );
echo "Date = $date_string\n";

sqlsrv_close( $conn);
?>
```

See Also

Retrieving Data

Updating Data (Microsoft Drivers for PHP for SQL Server)

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The topics in this section address how to update data in a database by examining common use cases.

The steps for using the Microsoft Drivers for PHP for SQL Server to update data in a database can be summarized as follows:

1. Define a Transact-SQL query that performs an insert, update, or delete operation.
2. Update parameter values for parameterized queries.
3. Execute the Transact-SQL queries with the updated parameter values (if applicable). See Comparing Execution Functions for more information about executing a query.

In This Section

TOPIC	DESCRIPTION
How to: Perform Parameterized Queries	Describes how to perform parameterized queries.
How to: Send Data as a Stream	Describes how to stream data to the server.
How to: Perform Transactions	Describes how to use **sqlsrv** functions to perform transactions.

See Also

Example Application (SQLSRV Driver)
Programming Guide for PHP SQL Driver

How to: Perform Parameterized Queries

3/14/2017 • 4 min to read • Edit on GitHub

Download PHP Driver

This topic summarizes and demonstrates how to use the Microsoft Drivers for PHP for SQL Server to perform a parameterized query.

The steps for performing a parameterized query can be summarized into four steps:

1. Put question marks (?) as parameter placeholders in the Transact-SQL string that is the query to be executed.

2. Initialize or update PHP variables that correspond to the placeholders in the Transact-SQL query.

3. Use PHP variables from step 2 to create or update an array of parameter values that correspond in order to parameter placeholders in the Transact-SQL string.

4. Execute the query:

   - If you are using the SQLSRV driver, use sqlsrv_query or sqlsrv_prepare/sqlsrv_execute.

   - If you are using the PDO_SQLSRV driver, execute the query with PDO::prepare and PDOStatement::execute. The topics for PDO::prepare and PDOStatement::execute have code examples.

The rest of this topic discusses parameterized queries using the SQLSRV driver.

NOTE

Parameters are implicitly bound by using **sqlsrv_prepare**. This means that if a parameterized query is prepared using **sqlsrv_prepare** and values in the parameter array are updated, the updated values will be used upon the next execution of the query. See the second example in this topic for more detail.

Example

The following example updates the quantity for a specified product ID in the *Production.ProductInventory* table of the AdventureWorks database. The quantity and product ID are parameters in the UPDATE query.

The example then queries the database to verify that the quantity has been correctly updated. The product ID is a parameter in the SELECT query.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Define the Transact-SQL query.
Use question marks as parameter placeholders. */
$tsql1 = "UPDATE Production.ProductInventory
          SET Quantity = ?
          WHERE ProductID = ?";

/* Initialize $qty and $productId */
$qty = 10; $productId = 709;

/* Execute the statement with the specified parameter values. */
$stmt1 = sqlsrv_query( $conn, $tsql1, array($qty, $productId));
if( $stmt1 === false )
{
     echo "Statement 1 could not be executed.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Free statement resources. */
sqlsrv_free_stmt( $stmt1);

/* Now verify the updated quantity.
Use a question mark as parameter placeholder. */
$tsql2 = "SELECT Quantity
          FROM Production.ProductInventory
          WHERE ProductID = ?";

/* Execute the statement with the specified parameter value.
Display the returned data if no errors occur. */
$stmt2 = sqlsrv_query( $conn, $tsql2, array($productId));
if( $stmt2 === false )
{
     echo "Statement 2 could not be executed.\n";
     die( print_r(sqlsrv_errors(), true));
}
else
{
     $qty = sqlsrv_fetch_array( $stmt2);
     echo "There are $qty[0] of product $productId in inventory.\n";
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt2);
sqlsrv_close( $conn);
?>
```

The previous example uses the **sqlsrv_query** function to execute queries. This function is good for executing one-time queries since it does both statement preparation and execution. The combination of **sqlsrv_prepare/sqlsrv_execute** is best for re-execution of a query with different parameter values. To see an example of re-execution of a query with different parameter values, see the next example.

Example

The following example demonstrates the implicit binding of variables when you use the **sqlsrv_prepare** function. The example inserts several sales orders into the *Sales.SalesOrderDetail* table. The *$params* array is bound to the statement (*$stmt*) when **sqlsrv_prepare** is called. Before each execution of a query that inserts a new sales order into the table, the *$params* array is updated with new values corresponding to sales order details. The subsequent query execution uses the new parameter values.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

$tsql = "INSERT INTO Sales.SalesOrderDetail (SalesOrderID,
                                             OrderQty,
                                             ProductID,
                                             SpecialOfferID,
                                             UnitPrice)
         VALUES (?, ?, ?, ?, ?)";

/* Each sub array here will be a parameter array for a query.
The values in each sub array are, in order, SalesOrderID, OrderQty,
 ProductID, SpecialOfferID, UnitPrice. */
$parameters = array( array(43659, 8, 711, 1, 20.19),
                     array(43660, 6, 762, 1, 419.46),
                     array(43661, 4, 741, 1, 818.70)
                    );

/* Initialize parameter values. */
$orderId = 0;
$qty = 0;
$prodId = 0;
$specialOfferId = 0;
$price = 0.0;

/* Prepare the statement. $params is implicitly bound to $stmt. */
$stmt = sqlsrv_prepare( $conn, $tsql, array( &$orderId,
                                             &$qty,
                                             &$prodId,
                                             &$specialOfferId,
                                             &$price));
if( $stmt === false )
{
     echo "Statement could not be prepared.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Execute a statement for each set of params in $parameters.
Because $params is bound to $stmt, as the values are changed, the
new values are used in the subsequent execution. */
foreach( $parameters as $params)
{
     list($orderId, $qty, $prodId, $specialOfferId, $price) = $params;
     if( sqlsrv_execute($stmt) === false )
     {
          echo "Statement could not be executed.\n";
          die( print_r( sqlsrv_errors(), true));
     }
     else
     {
          /* Verify that the row was successfully inserted. */
          echo "Rows affected: ".sqlsrv_rows_affected( $stmt )."\n";
     }
}
/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

See Also

Converting Data Types
Security Considerations for PHP SQL Driver About Code Examples in the Documentation
sqlsrv_rows_affected

How to: Send Data as a Stream

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

The Microsoft Drivers for PHP for SQL Server takes advantage of PHP streams for sending large objects to the server. The examples in this topic demonstrate how to send data as a stream. The first example uses the SQLSRV driver to demonstrate the default behavior, which is to send all stream data at the time of query execution. The second example uses the SQLSRV driver to demonstrate how to send up to eight kilobytes (8K) of stream data at a time to the server.

The third example shows how to send stream data to the server using the PDO_SQLSRV driver.

Example

The following example inserts a row into the *Production.ProductReview* table of the AdventureWorks database. The customer comments (*$comments*) are opened as a stream with the PHP fopen function and then streamed to the server upon execution of the query.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console.

```php
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up the Transact-SQL query. */
$tsql = "INSERT INTO Production.ProductReview (ProductID,
                                               ReviewerName,
                                               ReviewDate,
                                               EmailAddress,
                                               Rating,
                                               Comments)
         VALUES (?, ?, ?, ?, ?, ?)";

/* Set the parameter values and put them in an array.
Note that $comments is opened as a stream. */
$productID = '709';
$name = 'Customer Name';
$date = date("Y-m-d");
$email = 'customer@name.com';
$rating = 3;
$comments = fopen( "data://text/plain,[ Insert lengthy comment here.]",
                  "r");
$params = array($productID, $name, $date, $email, $rating, $comments);

/* Execute the query. All stream data is sent upon execution.*/
$stmt = sqlsrv_query($conn, $tsql, $params);
if( $stmt === false )
{
     echo "Error in statement execution.\n";
     die( print_r( sqlsrv_errors(), true));
}
else
{
     echo "The query was successfully executed.";
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

Example

The next example is the same as the example above, but the default behavior of sending all stream data at execution is turned off. The example uses sqlsrv_send_stream_data to send stream data to the server. Up to eight kilobytes (8K) of data is sent with each call to **sqlsrv_send_stream_data**. The script counts the number of calls made by **sqlsrv_send_stream_data** and displays the count to the console.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console.

```php
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up the Transact-SQL query. */
$tsql = "INSERT INTO Production.ProductReview (ProductID,
                                               ReviewerName,
                                               ReviewDate,
                                               EmailAddress,
                                               Rating,
                                               Comments)
         VALUES (?, ?, ?, ?, ?, ?)";

/* Set the parameter values and put them in an array.
Note that $comments is opened as a stream. */
$productID = '709';
$name = 'Customer Name';
$date = date("Y-m-d");
$email = 'customer@name.com';
$rating = 3;
$comments = fopen( "data://text/plain,[ Insert lengthy comment here.]",
                  "r");
$params = array($productID, $name, $date, $email, $rating, $comments);

/* Turn off the default behavior of sending all stream data at
execution. */
$options = array("SendStreamParamsAtExec" => 0);

/* Execute the query. */
$stmt = sqlsrv_query($conn, $tsql, $params, $options);
if( $stmt === false )
{
     echo "Error in statement execution.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Send up to 8K of parameter data to the server with each call to
sqlsrv_send_stream_data. Count the calls. */
$i = 1;
while( sqlsrv_send_stream_data( $stmt))
{
     echo "$i call(s) made.\n";
     $i++;
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

Although the examples in this topic send character data to the server, data in any format can be sent as a stream. For example, you can also use the techniques that are demonstrated in this topic to send images in binary format as streams.

Example

```
<?php
   $server = "(local)";
   $database = "Test";
   $conn = new PDO( "sqlsrv:server=$server;Database = $database", "", "");

   $binary_source = fopen( "data://text/plain,", "r");

   $stmt = $conn->prepare("insert into binaries (imagedata) values (?)");
   $stmt->bindParam(1, $binary_source, PDO::PARAM_LOB);

   $conn->beginTransaction();
   $stmt->execute();
   $conn->commit();
?>
```

See Also

Updating Data (Microsoft Drivers for PHP for SQL Server)
Retrieving Data as a Stream Using the SQLSRV Driver
About Code Examples in the Documentation

How to: Perform Transactions

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

The SQLSRV driver of the Microsoft Drivers for PHP for SQL Server provides three functions for performing transactions:

- sqlsrv_begin_transaction
- sqlsrv_commit
- sqlsrv_rollback

The PDO_SQLSRV driver provides three methods for performing transactions:

- PDO::beginTransaction
- PDO::commit
- PDO::rollback

See PDO::beginTransaction for an example.

The remainder of this topic explains and demonstrates how to use the SQLSRV driver to perform transactions.

Remarks

The steps to execute a transaction can be summarized as follows:

1. Begin the transaction with **sqlsrv_begin_transaction**.
2. Check the success or failure of each query that is part of the transaction.
3. If appropriate, commit the transaction with **sqlsrv_commit**. Otherwise, roll back the transaction with **sqlsrv_rollback**. After calling **sqlsrv_commit** or **sqlsrv_rollback**, the driver is returned to auto-commit mode.

   By default, the Microsoft Drivers for PHP for SQL Server is in auto-commit mode. This means that all queries are automatically committed upon success unless they have been designated as part of an explicit transaction by using **sqlsrv_begin_transaction**.

   If an explicit transaction is not committed with **sqlsrv_commit**, it will be rolled back upon closing of the connection or termination of the script.

   Do not use embedded Transact-SQL to perform transactions. For example, do not execute a statement with "BEGIN TRANSACTION" as the Transact-SQL query to begin a transaction. The expected transactional behavior cannot be guaranteed when you use embedded Transact-SQL to perform transactions.

   The **sqlsrv** functions listed earlier should be used to perform transactions.

Example

Description

The following example executes several queries as part of a transaction. If all the queries are successful, the transaction is committed. If any one of the queries fails, the transaction is rolled back.

The example tries to delete a sales order from the *Sales.SalesOrderDetail* table and adjust product inventory levels in the *Product.ProductInventory* table for each product in the sales order. These queries are included in a transaction because all queries must be successful for the database to accurately reflect the state of orders and product availability.

The first query in the example retrieves product IDs and quantities for a specified sales order ID. This query is not part of the transaction. However, the script ends if this query fails because the product IDs and quantities are required to complete queries that are part of the subsequent transaction.

The ensuing queries (deletion of the sales order and updating of the product inventory quantities) are part of the transaction.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

Code

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Begin transaction. */
if( sqlsrv_begin_transaction($conn) === false )
{
     echo "Could not begin transaction.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set the Order ID.  */
$orderId = 43667;

/* Execute operations that are part of the transaction. Commit on
success, roll back on failure. */
if (perform_trans_ops($conn, $orderId))
{
     //If commit fails, roll back the transaction.
     if(sqlsrv_commit($conn))
     {
         echo "Transaction committed.\n";
     }
     else
     {
         echo "Commit failed - rolling back.\n";
         sqlsrv_rollback($conn);
     }
}
else
{
     "Error in transaction operation - rolling back.\n";
     sqlsrv_rollback($conn);
}

/*Free connection resources*/
sqlsrv_close( $conn);

/*----------------  FUNCTION: perform_trans_ops  -----------------*/
function perform_trans_ops($conn, $orderId)
{
    /* Define query to update inventory based on sales order info. */
    $tsql1 = "UPDATE Production.ProductInventory
              SET Quantity = Quantity + s.OrderQty
              FROM Production.ProductInventory p
              JOIN Sales.SalesOrderDetail s
              ON s.ProductID = p.ProductID
              WHERE s.SalesOrderID = ?";

    /* Define the parameters array. */
    $params = array($orderId);

    /* Execute the UPDATE statement. Return false on failure. */
    if( sqlsrv_query( $conn, $tsql1, $params) === false ) return false;

    /* Delete the sales order. Return false on failure */
    $tsql2 = "DELETE FROM Sales.SalesOrderDetail
              WHERE SalesOrderID = ?";
    if(sqlsrv_query( $conn, $tsql2, $params) === false ) return false;

    /* Return true because all operations were successful. */
    return true;
}
?>
```

Comments

For the purpose of focusing on transaction behavior, some recommended error handling is not included in the previous example. For a production application, we recommend that any call to a **sqlsrv** function be checked for errors and handled accordingly.

See Also

Updating Data (Microsoft Drivers for PHP for SQL Server)
Transactions (Database Engine)
About Code Examples in the Documentation

Converting Data Types

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The Microsoft Drivers for PHP for SQL Server allows you to specify data types when you send data to or retrieve data from SQL Server . Specifying data types is optional. If data types are not specified, default types will be used. The topics in this section describe how to specify data types and provide details about default data types.

In This Section

TOPIC	DESCRIPTION
Default SQL Server Data Types	Provides information about the default SQL Server data types when sending data to the server.
Default PHP Data Types	Provides information about the default PHP data types when retrieving data from the server.
How to: Specify SQL Server Data Types	Demonstrates how to specify SQL Server data types when sending data to the server.
How to: Specify PHP Data Types	Demonstrates how to specify PHP data types when retrieving data from the server.
How to: Send and Retrieve UTF-8 Data Using Built-In UTF-8 Support	Demonstrates how to use Microsoft Drivers for PHP for SQL Server's built-in support for UTF-8 data. Support for UTF-8 characters was added in version 1.1 of the Microsoft Drivers for PHP for SQL Server.

See Also

Programming Guide for PHP SQL Driver SQLSRV Driver API Reference
Constants (Microsoft Drivers for PHP for SQL Server)
Example Application (SQLSRV Driver)

Default SQL Server Data Types

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

When sending data to the server, the Microsoft Drivers for PHP for SQL Server converts data from its PHP data type to a SQL Server data type if no SQL Server data type has been specified by the user. The table that follows lists the PHP data type (the data type being sent to the server) and the default SQL Server data type (the data type to which the data is converted). For details about how to specify data types when sending data to the server, see How to: Specify SQL Server Data Types When Using the SQLSRV Driver.

PHP DATA TYPE	DEFAULT SQL SERVER TYPE IN THE SQLSRV DRIVER	DEFAULT SQL SERVER TYPE IN THE PDO_SQLSRV DRIVER
NULL	varchar(1)	not supported
Boolean	bit	bit
Integer	int	int
Float	float(24)	not supported
String (length less than 8000 bytes)	varchar()	varchar()
String (length greater than 8000 bytes)	varchar(max)	varchar(max)
Resource	Not supported.	Not supported.
Stream (encoding: not binary)	varchar(max)	varchar(max)
Stream (encoding: binary)	varbinary	varbinary
Array	Not supported.	Not supported.
Object	Not supported.	Not supported.
DateTime (1)	datetime	Not supported.

See Also

Constants (Microsoft Drivers for PHP for SQL Server)
Converting Data Types
sqlsrv_field_metadata
PHP Types
Data Types (Transact-SQL)

Default PHP Data Types

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

When retrieving data from the server, the Microsoft Drivers for PHP for SQL Server converts data to a default PHP data type if no PHP data type has been specified by the user.

When data is returned using the PDO_SQLSRV driver, the data type will either be int or string.

The remainder of this topic discusses default data types using the SQLSRV driver.

The following table lists the SQL Server data type (the data type being retrieved from the server), the default PHP data type (the data type to which data is converted), and the default encoding for streams and strings. For details about how to specify data types when retrieving data from the server, see How to: Specify PHP Data Types.

SQL SERVER TYPE	DEFAULT PHP TYPE	DEFAULT ENCODING
bigint	String	8-bit character[1]
binary	Stream[2]	Binary[3]
bit	Integer	8-bit character[1]
char	String	8-bit character[1]
date[8]	Datetime	Not applicable
datetime[8]	Datetime	Not applicable
datetime2[8]	Datetime	Not applicable
datetimeoffset[8]	Datetime	Not applicable
decimal	String	8-bit character[1]
float	Float	8-bit character[1]
geography	STREAM	Binary[3]
geometry	STREAM	Binary[3]
image[4]	Stream[2]	Binary[3]
int	Integer	8-bit character[1]
money	String	8-bit character[1]
nchar	String	8-bit character[1]

SQL SERVER TYPE	DEFAULT PHP TYPE	DEFAULT ENCODING
numeric	String	8-bit character[1]
nvarchar	String	8-bit character[1]
nvarchar(MAX)	Stream[2]	8-bit character[1]
ntext[5]	Stream[2]	8-bit character[1]
real	Float	8-bit character[1]
smalldatetime	Datetime	8-bit character[1]
smallint	Integer	8-bit character[1]
smallmoney	String	8-bit character[1]
sql_variant	String	8-bit character[1]
text[6]	Stream[2]	8-bit character[1]
time[8]	Datetime	Not applicable
timestamp	String	8-bit character[1]
tinyint	Integer	8-bit character[1]
UDT	Stream[2]	Binary[3]
uniqueidentifier	String[7]	8-bit character[1]
varbinary	Stream[2]	Binary[3]
varbinary(MAX)	Stream[2]	Binary[3]
varchar	String	8-bit character[1]
varchar(MAX)	Stream[2]	8-bit character[1]
variant	Not supported	Not supported
xml	Stream[2]	8-bit character[1]

1. Data is returned in 8-bit characters as specified in the code page of the Windows locale set on the system. Any multi-byte characters or characters that do not map into this code page are substituted with a single byte question mark (?) character.

2. If sqlsrv_fetch_array or sqlsrv_fetch_object is used to retrieve data that has a default PHP type of Stream, the data will be returned as a string with the same encoding as the stream. For example, if a SQL Server binary type is retrieved by using **sqlsrv_fetch_array**, the default return type will be a binary string.

3. Data is returned as a raw byte stream from the server without performing encoding or translation.

4. This is a legacy type that maps to the varbinary(max) type.
5. This is a legacy type that maps to the nvarchar(max) type.
6. This is a legacy type that maps to the varchar(max) type.
7. UNIQUEIDENTIFIERs are GUIDs represented by the following regular expression:

   [0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-f]{4}-[0-9a-fA-f]{4}-[0-9a-fA-F]{12}

8. Date and time types can be retrieved as strings. For more information, see How to: Retrieve Date and Time Type as Strings Using the SQLSRV Driver.

Other New SQL Server 2008 Data Types and Features

Data types that are new in SQL Server 2008 and that exist outside of columns (such as table-valued parameters) are not supported in the Microsoft Drivers for PHP for SQL Server. The table below summarizes the PHP support for new SQL Server 2008 features.

FEATURE	PHP SUPPORT
Table-valued parameter	No
Sparse columns	Partial
Null-bit compression	Yes
Large CLR user-defined types (UDTs)	Yes
Service principal name	No
MERGE	Yes
FILESTREAM	Partial

Partial type support means that you cannot programmatically query for the type of the column.

See Also

Constants (Microsoft Drivers for PHP for SQL Server)
Converting Data Types
PHP Types
Data Types (Transact-SQL)
sqlsrv_field_metadata

How to: Specify SQL Server Data Types When Using the SQLSRV Driver

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

This topic demonstrates how to use the SQLSRV driver to specify the SQL Server data type for data that is sent to the server. This topic does not apply when using the PDO_SQLSRV driver.

To specify the SQL Server data type, you must use the optional *$params* array when you prepare or execute a query that inserts or updates data. For details about the structure and syntax of the *$params* array, see sqlsrv_query or sqlsrv_prepare.

The following steps summarize how to specify the SQL Server data type when sending data to the server:

NOTE

If no SQL Server data type is specified, default types will be used. For information about default SQL Server data types, see Default SQL Server Data Types.

1. Define a Transact-SQL query that inserts or updates data. Use question marks (?) as placeholders for parameter values in the query.

2. Initialize or update PHP variables that correspond to the placeholders in the Transact-SQL query.

3. Construct the *$params* array to be used when preparing or executing the query. Note that each element of the *$params* array must also be an array when you specify the SQL Server data type.

4. Specify the desired SQL Server data type by using the appropriate **SQLSRV_SQLTYPE_\*** constant as the fourth parameter in each sub-array of the *$params* array. For a complete list of the **SQLSRV_SQLTYPE_*** constants, see the SQLTYPEs section of Constants (Microsoft Drivers for PHP for SQL Server). For example, in the code below, *$changeDate*, *$rate*, and *$payFrequency* are specified respectively as the SQL Server types **datetime**, **money**, and **tinyint** in the *$params* array. Because no SQL Server type is specified for *$employeeId* and it is initialized to an integer, the default SQL Server type **integer** is used.

   ```
   $employeeId = 5;
   $changeDate = "2005-06-07";
   $rate = 30;
   $payFrequency = 2;
   $params = array(
               array($employeeId, null),
               array($changeDate, null, null, SQLSRV_SQLTYPE_DATETIME),
               array($rate, null, null, SQLSRV_SQLTYPE_MONEY),
               array($payFrequency, null, null, SQLSRV_SQLTYPE_TINYINT)
             );
   ```

Example

The following example inserts data into the *HumanResources.EmployeePayHistory* table of the Adventureworks database. SQL Server types are specified for the *$changeDate*, *$rate*, and *$payFrequency* parameters. The default SQL Server type is used for the *$employeeId* parameter. To verify that the data was inserted successfully, the same data is retrieved and displayed.

This example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Define the query. */
$tsql1 = "INSERT INTO HumanResources.EmployeePayHistory (EmployeeID,
                                                        RateChangeDate,
                                                        Rate,
                                                        PayFrequency)
           VALUES (?, ?, ?, ?)";

/* Construct the parameter array. */
$employeeId = 5;
$changeDate = "2005-06-07";
$rate = 30;
$payFrequency = 2;
$params1 = array(
               array($employeeId, null),
               array($changeDate, null, null, SQLSRV_SQLTYPE_DATETIME),
               array($rate, null, null, SQLSRV_SQLTYPE_MONEY),
               array($payFrequency, null, null, SQLSRV_SQLTYPE_TINYINT)
           );

/* Execute the INSERT query. */
$stmt1 = sqlsrv_query($conn, $tsql1, $params1);
if( $stmt1 === false )
{
     echo "Error in execution of INSERT.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve the newly inserted data. */
/* Define the query. */
$tsql2 = "SELECT EmployeeID, RateChangeDate, Rate, PayFrequency
          FROM HumanResources.EmployeePayHistory
          WHERE EmployeeID = ? AND RateChangeDate = ?";

/* Construct the parameter array. */
$params2 = array($employeeId, $changeDate);

/*Execute the SELECT query. */
$stmt2 = sqlsrv_query($conn, $tsql2, $params2);
if( $stmt2 === false )
{
     echo "Error in execution of SELECT.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve and display the results. */
$row = sqlsrv_fetch_array( $stmt2 );
if( $row === false )
{
     echo "Error in fetching data.\n";
     die( print_r( sqlsrv_errors(), true));
}
echo "EmployeeID: ".$row['EmployeeID']."\n";
echo "Change Date: ".date_format($row['RateChangeDate'], "Y-m-d")."\n";
echo "Rate: ".$row['Rate']."\n";
echo "PayFrequency: ".$row['PayFrequency']."\n";

/* Free statement and connection resources. */
sqlsrv_free_stmt($stmt1);
sqlsrv_free_stmt($stmt2);
sqlsrv_close($conn);
?>
```

See Also

Retrieving Data
About Code Examples in the Documentation
How to: Specify PHP Data Types
Converting Data Types
How to: Send and Retrieve UTF-8 Data Using Built-In UTF-8 Support

How to: Specify PHP Data Types

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

When using the PDO_SQLSRV driver, you can specify the PHP data type when retrieving data from the server with PDOStatement::bindColumn and PDOStatement::bindParam. See PDOStatement::bindColumn and PDOStatement::bindParam for more information.

The following steps summarize how to specify PHP data types when retrieving data from the server using the SQLSRV driver:

1. Set up and execute a Transact-SQL query with sqlsrv_query or the combination of sqlsrv_prepare/sqlsrv_execute.

2. Make a row of data available for reading with sqlsrv_fetch.

3. Retrieve field data from a returned row using sqlsrv_get_field with the desired PHP data type specified as the optional third parameter. If the optional third parameter is not specified, data will be returned according to the default PHP types. For information about the default PHP return types, see Default PHP Data Types.

   For information about the constants used to specify the PHP data type, see the PHPTYPEs section of Constants (Microsoft Drivers for PHP for SQL Server).

Example

The following example retrieves rows from the *Production.ProductReview* table of the AdventureWorks database. In each returned row the *ReviewDate* field is retrieved as a string and the *Comments* field is retrieved as a stream. The stream data is displayed by using the PHP fpassthru function.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```php
<?php
/*Connect to the local server using Windows Authentication and specify
the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up the Transact-SQL query. */
$tsql = "SELECT ReviewerName,
                ReviewDate,
                Rating,
                Comments
         FROM Production.ProductReview
         WHERE ProductID = ?
         ORDER BY ReviewDate DESC";

/* Set the parameter value. */
$productID = 709;
$params = array( $productID);

/* Execute the query. */
$stmt = sqlsrv_query($conn, $tsql, $params);
if( $stmt === false )
{
     echo "Error in statement execution.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve and display the data. The first and third fields are
retrieved according to their default types, strings. The second field
is retrieved as a string with 8-bit character encoding. The fourth
field is retrieved as a stream with 8-bit character encoding.*/
while ( sqlsrv_fetch( $stmt))
{
   echo "Name: ".sqlsrv_get_field( $stmt, 0 )."\n";
   echo "Date: ".sqlsrv_get_field( $stmt, 1,
                       SQLSRV_PHPTYPE_STRING( SQLSRV_ENC_CHAR))."\n";
   echo "Rating: ".sqlsrv_get_field( $stmt, 2 )."\n";
   echo "Comments: ";
   $comments = sqlsrv_get_field( $stmt, 3,
                            SQLSRV_PHPTYPE_STREAM(SQLSRV_ENC_CHAR));
   fpassthru( $comments);
   echo "\n";
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

In the example, retrieving the second field (*ReviewDate*) as a string preserves millisecond accuracy of the SQL Server DATETIME data type. By default, the SQL Server DATETIME data type is retrieved as a PHP DateTime object in which the millisecond accuracy is lost.

Retrieving the fourth field (*Comments*) as a stream is for demonstration purposes. By default, the SQL Server data type nvarchar(3850) is retrieved as a string, which is acceptable for most situations.

NOTE

The sqlsrv_field_metadata function provides a way to obtain field information, including type information, before executing a query.

See Also

Retrieving Data
About Code Examples in the Documentation
How to: Retrieve Output Parameters Using the SQLSRV Driver
How to: Retrieve Input and Output Parameters Using the SQLSRV Driver

How to: Send and Retrieve UTF-8 Data Using Built-In UTF-8 Support

3/14/2017 • 4 min to read • Edit on GitHub

Download PHP Driver

If you are using the PDO_SQLSRV driver, you can specify the encoding with the PDO::SQLSRV_ATTR_ENCODING attribute. For more information, see Constants (Microsoft Drivers for PHP for SQL Server).

The remainder of this topic discusses encoding with the SQLSRV driver.

To send or retrieve UTF-8 encoded data to the server:

1. Make sure that the source or destination column is of type **nchar** or **nvarchar**.

2. Specify the PHP type as `SQLSRV_PHPTYPE_STRING('UTF-8')` in the parameters array. Or, specify `"CharacterSet" => "UTF-8"` as a connection option.

   When you specify a character set as part of the connection options, the driver assumes that the other connection option strings use that same character set. The server name and query strings are also assumed to use the same character set.

Note that you can pass UTF-8 or SQLSRV_ENC_CHAR to **CharacterSet** (you cannot pass SQLSRV_ENC_BINARY). The default encoding is SQLSRV_ENC_CHAR.

Example

The following example demonstrates how to send and retrieve UTF-8 encoded data by specifying the UTF-8 character set when making the connection. The example updates the Comments column of the Production.ProductReview table for a specified review ID. The example also retrieves the newly updated data and displays it. Note that the Comments column is of type **nvarcahr(3850).** Also note that before data is sent to the server it is converted to UTF-8 encoding using the PHP **utf8_encode** function. This is done for demonstration purposes only. In a real application scenario you would begin with UTF-8 encoded data.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the browser when the example is run from the browser.

```
<?php

// Connect to the local server using Windows Authentication and
// specify the AdventureWorks database as the database in use.
//
$serverName = "MyServer";
$connectionInfo = array( "Database"=>"AdventureWorks", "CharacterSet" => "UTF-8");
$conn = sqlsrv_connect( $serverName, $connectionInfo);

if ( $conn === false ) {
   echo "Could not connect.<br>";
   die( print_r( sqlsrv_errors(), true));
}

// Set up the Transact-SQL query.
//
$tsql1 = "UPDATE Production.ProductReview
          SET Comments = ?
          WHERE ProductReviewID = ?";

// Set the parameter values and put them in an array. Note that
// $comments is converted to UTF-8 encoding with the PHP function
// utf8_encode to simulate an application that uses UTF-8 encoded data.
//
$reviewID = 3;
$comments = utf8_encode("testing 1, 2, 3, 4.  Testing.");
$params1 = array(
                  array( $comments, null ),
                  array( $reviewID, null )
                );

// Execute the query.
//
$stmt1 = sqlsrv_query($conn, $tsql1, $params1);

if ( $stmt1 === false ) {
   echo "Error in statement execution.<br>";
   die( print_r( sqlsrv_errors(), true));
}
else {
   echo "The update was successfully executed.<br>";
}

// Retrieve the newly updated data.
//
$tsql2 = "SELECT Comments
          FROM Production.ProductReview
          WHERE ProductReviewID = ?";

// Set up the parameter array.
//
$params2 = array($reviewID);

// Execute the query.
//
$stmt2 = sqlsrv_query($conn, $tsql2, $params2);
if ( $stmt2 === false ) {
   echo "Error in statement execution.<br>";
   die( print_r( sqlsrv_errors(), true));
}

// Retrieve and display the data.
//
if ( sqlsrv_fetch($stmt2) ) {
   echo "Comments: ";
   $data = sqlsrv_get_field( $stmt2, 0 );
   echo $data."<br>";
}
else {
   echo "Error in fetching data.<br>";
   die( print_r( sqlsrv_errors(), true));
}

// Free statement and connection resources.
//
sqlsrv_free_stmt( $stmt1 );
sqlsrv_free_stmt( $stmt2 );
sqlsrv_close( $conn);
?>
```

For information about storing Unicode data, see Working with Unicode Data.

Example

The following example is similar to the first sample but instead of specifying the UTF-8 character set on the connection, this sample shows how to specify the UTF-8 character set on the column.

```php
<?php

// Connect to the local server using Windows Authentication and
// specify the AdventureWorks database as the database in use.
//
$serverName = "MyServer";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);

if ( $conn === false ) {
   echo "Could not connect.<br>";
   die( print_r( sqlsrv_errors(), true));
}

// Set up the Transact-SQL query.
//
$tsql1 = "UPDATE Production.ProductReview
          SET Comments = ?
          WHERE ProductReviewID = ?";

// Set the parameter values and put them in an array. Note that
// $comments is converted to UTF-8 encoding with the PHP function
// utf8_encode to simulate an application that uses UTF-8 encoded data.
//
$reviewID = 3;
$comments = utf8_encode("testing");
$params1 = array(
                  array($comments,
                        SQLSRV_PARAM_IN,
                        SQLSRV_PHPTYPE_STRING('UTF-8')
                  ),
                  array($reviewID)
                );

// Execute the query.
//
$stmt1 = sqlsrv_query($conn, $tsql1, $params1);

if ( $stmt1 === false ) {
   echo "Error in statement execution.<br>";
   die( print_r( sqlsrv_errors(), true));
}
else {
   echo "The update was successfully executed.<br>";
}

// Retrieve the newly updated data.
//
$tsql2 = "SELECT Comments
          FROM Production.ProductReview
          WHERE ProductReviewID = ?";

// Set up the parameter array.
//
$params2 = array($reviewID);

// Execute the query.
//
$stmt2 = sqlsrv_query($conn, $tsql2, $params2);
if ( $stmt2 === false ) {
   echo "Error in statement execution.<br>";
   die( print_r( sqlsrv_errors(), true));
}

// Retrieve and display the data.
//
```

```
if ( sqlsrv_fetch($stmt2) ) {
   echo "Comments: ";
   $data = sqlsrv_get_field($stmt2,
                            0,
                            SQLSRV_PHPTYPE_STRING('UTF-8')
                           );
   echo $data."<br>";
}
else {
   echo "Error in fetching data.<br>";
   die( print_r( sqlsrv_errors(), true));
}

// Free statement and connection resources.
//
sqlsrv_free_stmt( $stmt1 );
sqlsrv_free_stmt( $stmt2 );
sqlsrv_close( $conn);
?>
```

See Also

Retrieving Data
Updating Data (Microsoft Drivers for PHP for SQL Server)
SQLSRV Driver API Reference
Constants (Microsoft Drivers for PHP for SQL Server)
Example Application (SQLSRV Driver)

Handling Errors and Warnings

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

If you are using the PDO_SQLSRV driver, you can find more information about errors and error handling on the PDO website.

Topics in this section provide information about handling errors and warnings with the SQLSRV driver of the Microsoft Drivers for PHP for SQL Server.

In This Section

TOPIC	DESCRIPTION
How to: Configure Error and Warning Handling Using the SQLSRV Driver	Demonstrates how to change configuration settings for handling errors and warnings.
How to: Handle Errors and Warnings Using the SQLSRV Driver	Demonstrates how to handle errors and warnings separately.

Reference

sqlsrv_errors

sqlsrv_configure

sqlsrv_get_config

See Also

Programming Guide for PHP SQL Driver

How to: Configure Error and Warning Handling Using the SQLSRV Driver

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

This topic describes how to configure the SQLSRV driver to handle errors and warnings.

By default, the SQLSRV driver treats warnings as errors; a call to a **sqlsrv** function that generates an error or a warning will return **false**. To disable this behavior, use the sqlsrv_configure function. When the following line of code is included at the beginning of a script, a **sqlsrv** function that generates only warnings (no errors) will not return **false**:

```
sqlsrv_configure("WarningsReturnAsErrors", 0);
```

The following line of code will reset the default behavior (warnings are treated as errors):

```
sqlsrv_configure("WarningsReturnAsErrors", 1);
```

NOTE

Warnings that correspond to SQLSTATE values 01000, 01001, 01003, and 01S02 are never treated as errors. Regardless of the configuration, a **sqlsrv** function that generates only warnings that correspond to one of these states will not return **false**.

The value for **WarningsReturnAsErrors** can also be set in the php.ini file. For example, this entry in the `[sqlsrv]` section of the php.ini file will turn off the default behavior.

```
sqlsrv.WarningsReturnAsErrors = 0
```

For information about retrieving error and warning information, see sqlsrv_errors and How to: Handle Errors and Warnings.

Example

The following code example demonstrates how to disable the default error-handling behavior. The example uses the Transact-SQL PRINT command to generate a warning. For more information about the PRINT command, see PRINT (Transact-SQL).

The example first demonstrates the default error-handling behavior by executing a query that generates a warning. This warning is treated as an error. After changing the error-handling configuration, the same query is executed. The warning is not treated as an error.

The example assumes that SQL Server is installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication. */
$serverName = "(local)";
$conn = sqlsrv_connect( $serverName );
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* The Transact-SQL PRINT statement can be used to return
informational or warning messages*/
$tsql = "PRINT 'The PRINT statement can be used ";
$tsql .= "to return user-defined warnings.'";

/* Execute the query and print any errors. */
$stmt1 = sqlsrv_query( $conn, $tsql);
if($stmt1 === false)
{
     echo "By default, warnings are treated as errors:\n";
     /* Dump errors in the error collection. */
     print_r(sqlsrv_errors(SQLSRV_ERR_ERRORS));
}

/* Disable warnings as errors behavior. */
sqlsrv_configure("WarningsReturnAsErrors", 0);

/* Execute the same query and print any errors. */
$stmt2 = sqlsrv_query( $conn, $tsql);
if($stmt2 === false)
{
     /* Dump errors in the error collection. */
     /* Since the warning generated by the query will not be treated as
        an error, this block of code will not be executed. */
     print_r(sqlsrv_errors(SQLSRV_ERR_ERRORS));
}
else
{
     echo "After calling ";
     echo "sqlsrv_configure('WarningsReturnAsErrors', 0), ";
     echo "warnings are not treated as errors.";
}

/*Close the connection. */
sqlsrv_close($conn);
?>
```

See Also

Logging Activity
Programming Guide for PHP SQL Driver SQLSRV Driver API Reference

How to: Handle Errors and Warnings Using the SQLSRV Driver

3/14/2017 • 4 min to read • Edit on GitHub

Download PHP Driver

By default, the SQLSRV driver treats warnings as errors; a call to a **sqlsrv** function that generates an error or a warning will return **false**. This topic demonstrates how to turn off this default behavior and how to handle warnings separately from errors.

NOTE

There are some exceptions to the default behavior of treating warnings as errors. Warnings that correspond to the SQLSTATE values 01000, 01001, 01003, and 01S02 are never treated as errors.

Example

The following code example uses two user-defined functions, **DisplayErrors** and **DisplayWarnings**, to handle errors and warnings. The example demonstrates how to handle warnings and errors separately by doing the following:

1. Turns off the default behavior of treating warnings as errors.
2. Creates a stored procedure that updates an employee's vacation hours and returns the remaining vacation hours as an output parameter. When an employee's available vacation hours are less than zero, the stored procedure prints a warning.
3. Updates vacation hours for several employees by calling the stored procedure for each employee, and displays the messages that correspond to any warnings and errors that occur.
4. Displays the remaining vacation hours for each employee.

Note that in the first call to a **sqlsrv** function (sqlsrv_configure), warnings are treated as errors. Because warnings are added to the error collection, you do not have to check for warnings separately from errors. In subsequent calls to **sqlsrv** functions, however, warnings will not be treated as errors, so you must check explicitly for warnings and for errors.

Also note that the example code checks for errors after each call to a **sqlsrv** function. This is the recommended practice.

This example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line. When the example is run against a new installation of the AdventureWorks database, it produces three warnings and two errors. The first two warnings are standard warnings that are issued when you connect to a database. The third warning occurs because an employee's available vacation hours are updated to a value less than zero. The errors occur because an employee's available vacation hours are updated to a value less than -40 hours, which is a violation of a constraint on the table.

```
<?php
/* Turn off the default behavior of treating errors as warnings.
Note: Turning off the default behavior is done here for demonstration
purposes only. If setting the configuration fails, display errors and
purposes only. If setting the configuration fails, display errors and
exit the script. */
if( sqlsrv_configure("WarningsReturnAsErrors", 0) === false)
{
     DisplayErrors();
     die;
}

/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array("Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);

/* If the connection fails, display errors and exit the script. */
if( $conn === false )
{
     DisplayErrors();
     die;
}
/* Display any warnings. */
DisplayWarnings();

/* Drop the stored procedure if it already exists. */
$tsql1 = "IF OBJECT_ID('SubtractVacationHours', 'P') IS NOT NULL
                DROP PROCEDURE SubtractVacationHours";
$stmt1 = sqlsrv_query($conn, $tsql1);

/* If the query fails, display errors and exit the script. */
if( $stmt1 === false)
{
     DisplayErrors();
     die;
}
/* Display any warnings. */
DisplayWarnings();

/* Free the statement resources. */
sqlsrv_free_stmt( $stmt1 );

/* Create the stored procedure. */
$tsql2 = "CREATE PROCEDURE SubtractVacationHours
                  @EmployeeID int,
                  @VacationHours smallint OUTPUT
              AS
                  UPDATE HumanResources.Employee
                  SET VacationHours = VacationHours - @VacationHours
                  WHERE EmployeeID = @EmployeeID;
                  SET @VacationHours = (SELECT VacationHours
                                       FROM HumanResources.Employee
                                       WHERE EmployeeID = @EmployeeID);
              IF @VacationHours < 0
              BEGIN
                PRINT 'WARNING: Vacation hours are now less than zero.'
              END;";
$stmt2 = sqlsrv_query( $conn, $tsql2 );

/* If the query fails, display errors and exit the script. */
if( $stmt2 === false)
{
     DisplayErrors();
     die;
}
/* Display any warnings. */
DisplayWarnings();

/* Free the statement resources. */
sqlsrv_free_stmt( $stmt2 );
```

```
/* Set up the array that maps employee ID to used vacation hours. */
$emp_hrs = array (7=>4, 8=>5, 9=>8, 11=>50);

/* Initialize variables that will be used as parameters. */
$employeeId = 0;
$vacationHrs = 0;

/* Set up the parameter array. */
$params = array(
                 array(&$employeeId, SQLSRV_PARAM_IN),
                 array(&$vacationHrs, SQLSRV_PARAM_INOUT)
                );

/* Define and prepare the query to substract used vacation hours. */
$tsql3 = "{call SubtractVacationHours(?, ?)}";
$stmt3 = sqlsrv_prepare($conn, $tsql3, $params);

/* If the statement preparation fails, display errors and exit the script. */
if( $stmt3 === false)
{
     DisplayErrors();
     die;
}
/* Display any warnings. */
DisplayWarnings();

/* Loop through the employee=>vacation hours array. Update parameter
 values before statement execution. */
foreach(array_keys($emp_hrs) as $employeeId)
{
     $vacationHrs = $emp_hrs[$employeeId];
     /* Execute the query.  If it fails, display the errors. */
     if( sqlsrv_execute($stmt3) === false)
     {
          DisplayErrors();
          die;
     }
     /* Display any warnings. */
     DisplayWarnings();

     /*Move to the next result returned by the stored procedure. */
     if( sqlsrv_next_result($stmt3) === false)
     {
          DisplayErrors();
          die;
     }
     /* Display any warnings. */
     DisplayWarnings();

     /* Display updated vacation hours. */
     echo "EmployeeID $employeeId has $vacationHrs ";
     echo "remaining vacation hours.\n";
}

/* Free the statement and connection resources. */
sqlsrv_free_stmt( $stmt3 );
sqlsrv_close( $conn );

/* ------------- Error Handling Functions --------------*/
function DisplayErrors()
{
     $errors = sqlsrv_errors(SQLSRV_ERR_ERRORS);
     foreach( $errors as $error )
     {
          echo "Error: ".$error['message']."\n";
     }
}

function DisplayWarnings()
```

```
{
     $warnings = sqlsrv_errors(SQLSRV_ERR_WARNINGS);
     if(!is_null($warnings))
     {
          foreach( $warnings as $warning )
          {
               echo "Warning: ".$warning['message']."\n";
          }
     }
}
?>
```

See Also

How to: Configure Error and Warning Handling Using the SQLSRV Driver
SQLSRV Driver API Reference

Logging Activity

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

By default, errors and warnings that are generated by the Microsoft Drivers for PHP for SQL Server are not logged. This topic discusses how to configure logging activity.

Logging Activity Using the PDO_SQLSRV Driver

The only configuration that is available for the PDO_SQLSRV driver is the pdo_sqlsrv.log_severity entry in the php.ini file.

Add the following at the end of your php.ini file:

```
[pdo_sqlsrv]
pdo_sqlsrv.log_severity = <number>
```

log_severity can be one of the following values:

VALUE	DESCRIPTION
0	Logging is disabled (is the default if nothing is defined).
-1	Specifies that errors, warnings, and notices will be logged.
1	Specifies that errors will be logged.
2	Specifies that warnings will be logged.
4	Specifies that notices will be logged.

Logging information will be added to the phperrors.log file.

PHP reads the configuration file on initialization and stores the data in a cache; it also provides an API to update these settings and use right away, and is written to the configuration file. This API enables application scripts to change the settings even after PHP initialization.

Logging Activity Using the SQLSRV Driver

To turn logging on, you can use the sqlsrv_configure function or you can alter the php.ini file. You can log activity on initializations, connections, statements, or error functions. You can also specify whether to log errors, warnings, notices, or all three.

NOTE

You can configure the location of the log file in the php.ini file.

Turning Logging On

You can turn logging on by using the sqlsrv_configure function to specify a value for the **LogSubsystems** setting.

For example, the following line of code configures the driver to log activity on connections:

```
sqlsrv_configure("LogSubsystems", SQLSRV_LOG_SYSTEM_CONN);
```

The following table describes the constants that can be used as the value for the **LogSubsystems** setting:

VALUE (INTEGER EQUIVALENT IN PARENTHESES)	DESCRIPTION
SQLSRV_LOG_SYSTEM_ALL (-1)	Turns on logging of all subsystems.
SQLSRV_LOG_SYSTEM_OFF (0)	Turns logging off. This is the default.
SQLSRV_LOG_SYSTEM_INIT (1)	Turns on logging of initialization activity.
SQLSRV_LOG_SYSTEM_CONN (2)	Turns on logging of connection activity.
SQLSRV_LOG_SYSTEM_STMT (4)	Turns on logging of statement activity.
SQLSRV_LOG_SYSTEM_UTIL (8)	Turns on logging of error functions activity (such as handle_error and handle_warning).

You can set more than one value at a time for the **LogSubsystems** setting by using the logical OR operator (|). For example, the following line of code turns on logging of activity on both connections and statements:

```
sqlsrv_configure("LogSubsystems", SQLSRV_LOG_SYSTEM_CONN | SQLSRV_LOG_SYSTEM_STMT);
```

You can also turn logging on by specifying an integer value for the **LogSubsystems** setting in the php.ini file. For example, adding the following line to the `[sqlsrv]` section of the php.ini file will turn on logging of connection activity:

```
sqlsrv.LogSubsystems = 2
```

By adding integer values together you can specify more than one option at a time. For example, adding the following line to the `[sqlsrv]` section of the php.ini file will turn on logging of connection and statement activity:

```
sqlsrv.LogSubsystems = 6
```

Logging Errors, Warnings, and Notices

After turning logging on, you must specify what to log. You can log one or more of the following: errors, warnings, and notices. For example, the following line of code specifies that only warnings will be logged:

```
sqlsrv_configure("LogSeverity", SQLSRV_LOG_SEVERITY_WARNING);
```

NOTE
The default setting for **LogSeverity** is **SQLSRV_LOG_SEVERITY_ERROR**. If logging is turned on and no setting for **LogSeverity** is specified, only errors are logged.

The following table describes the constants that can be used as the value for the **LogSeverity** setting:

VALUE (INTEGER EQUIVALENT IN PARENTHESES)	DESCRIPTION
SQLSRV_LOG_SEVERITY_ALL (-1)	Specifies that errors, warnings, and notices will be logged.
SQLSRV_LOG_SEVERITY_ERROR (1)	Specifies that errors will be logged. This is the default.

VALUE (INTEGER EQUIVALENT IN PARENTHESES)	DESCRIPTION
SQLSRV_LOG_SEVERITY_WARNING (2)	Specifies that warnings will be logged.
SQLSRV_LOG_SEVERITY_NOTICE (4)	Specifies that notices will be logged.

You can set more than one value at a time for the **LogSeverity** setting by using the logical OR operator (|). For example, the following line of code specifies that errors and warnings should be logged:

```
sqlsrv_configure("LogSeverity", SQLSRV_LOG_SEVERITY_ERROR | SQLSRV_LOG_SEVERITY_WARNING);
```

NOTE
Specifying a value for the **LogSeverity** setting does not turn logging on. You must turn logging on by specifying a value for the **LogSubsystems** setting, then specify the severity of what is logged by setting a value for **LogSeverity**.

You can also specify a setting for the **LogSeverity** setting by using integer values in the php.ini file. For example, adding the following line to the `[sqlsrv]` section of the php.ini file enables logging of warnings only:

```
sqlsrv.LogSeverity = 2
```

By adding integer values together, you can specify more than one option at a time. For example, adding the following line to the `[sqlsrv]` section of the php.ini file will enable logging of errors and warnings:

```
sqlsrv.LogSeverity = 3
```

See Also

Programming Guide for PHP SQL Driver Constants (Microsoft Drivers for PHP for SQL Server)
sqlsrv_configure
sqlsrv_get_config
SQLSRV Driver API Reference

Constants (Microsoft Drivers for PHP for SQL Server)

3/14/2017 • 6 min to read • Edit on GitHub

Download PHP Driver

This topic discusses the constants that are defined by the Microsoft Drivers for PHP for SQL Server.

PDO_SQLSRV Driver Constants

The constants listed on the PDO website are valid in the Microsoft Drivers for PHP for SQL Server.

The following describe the Microsoft specific constants in the PDO_SQLSRV driver.

Transaction Isolation Level Constants

The **TransactionIsolation** key, which is used with PDO::__construct, accepts one of the following constants:

- PDO::SQLSRV_TXN_READ_UNCOMMITTED
- PDO::SQLSRV_TXN_READ_COMMITTED
- PDO::SQLSRV_TXN_REPEATABLE_READ
- PDO::SQLSRV_TXN_SNAPSHOT
- PDO::SQLSRV_TXN_SERIALIZABLE

For more information about the **TransactionIsolation** key, see Connection Options.

Encoding Constants

The PDO::SQLSRV_ATTR_ENCODING attribute can be passed to PDOStatement::setAttribute, PDO::setAttribute, PDO::prepare, PDOStatement::bindColumn, and PDOStatement::bindParam.

The available values to pass to PDO::SQLSRV_ATTR_ENCODING are

PDO_SQLSRV DRIVER CONSTANT	DESCRIPTION
PDO::SQLSRV_ENCODING_BINARY	Data is a raw byte stream from the server without performing encoding or translation. Not valid for PDO::setAttribute.
PDO::SQLSRV_ENCODING_SYSTEM	Data is 8-bit characters as specified in the code page of the Windows locale that is set on the system. Any multi-byte characters or characters that do not map into this code page are substituted with a single byte question mark (?) character.
PDO::SQLSRV_ENCODING_UTF8	Data is in the UTF-8 encoding. This is the default encoding.

PDO_SQLSRV DRIVER CONSTANT	DESCRIPTION
PDO::SQLSRV_ENCODING_DEFAULT	Uses PDO::SQLSRV_ENCODING_SYSTEM if specified during connection. Use the connection's encoding if specified in a prepare statement.

Query Timeout

The PDO::SQLSRV_ATTR_QUERY_TIMEOUT attribute is any non-negative integer representing the timeout period, in seconds. Zero (0) is the default and means no timeout.

You can specify the PDO::SQLSRV_ATTR_QUERY_TIMEOUT attribute with PDOStatement::setAttribute, PDO::setAttribute, and PDO::prepare.

Direct or Prepared Execution

You can select direct query execution or prepared statement execution with the PDO::SQLSRV_ATTR_DIRECT_QUERY attribute. PDO::SQLSRV_ATTR_DIRECT_QUERY can be set with PDO::prepare or PDO::setAttribute. For more information about PDO::SQLSRV_ATTR_DIRECT_QUERY, see Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver.

Handling Numeric Fetches

The PDO::SQLSRV_ATTR_FETCHES_NUMERIC_TYPE attribute can be used to handle numeric fetches from columns with numeric SQL types (bit, integer, smallint, tinyint, float, and real). When PDO::SQLSRV_ATTR_FETCHES_NUMERIC_TYPE is set to true, the results from an integer column will be represented as ints, while SQL floats and reals will be represented as floats. This attribute can be set with PDOStatement::setAttribute.

SQLSRV Driver Constants

The following sections list the constants used by the SQLSRV driver.

ERR Constants

The following table lists the constants that are used to specify if sqlsrv_errors returns errors, warnings, or both.

VALUE	DESCRIPTION
SQLSRV_ERR_ALL	Errors and warnings generated on the last **sqlsrv** function call are returned. This is the default value.
SQLSRV_ERR_ERRORS	Errors generated on the last **sqlsrv** function call are returned.
SQLSRV_ERR_WARNINGS	Warnings generated on the last **sqlsrv** function call are returned.

FETCH Constants

The following table lists the constants that are used to specify the type of array returned by sqlsrv_fetch_array.

SQLSRV CONSTANT	DESCRIPTION
SQLSRV_FETCH_ASSOC	**sqlsrv_fetch_array** returns the next row of data as an associative array.

SQLSRV CONSTANT	DESCRIPTION
SQLSRV_FETCH_BOTH	**sqlsrv_fetch_array** returns the next row of data as an array with both numeric and associative keys. This is the default value.
SQLSRV_FETCH_NUMERIC	**sqlsrv_fetch_array** returns the next row of data as a numerically indexed array.

Logging Constants

This section lists the constants that are used to change the logging settings with sqlsrv_configure. For more information about logging activity, see Logging Activity.

The following table lists the constants that can be used as the value for the **LogSubsystems** setting:

SQLSRV CONSTANT (INTEGER EQUIVALENT IN PARENTHESES)	DESCRIPTION
SQLSRV_LOG_SYSTEM_ALL (-1)	Turns on logging of all subsystems.
SQLSRV_LOG_SYSTEM_CONN (2)	Turns on logging of connection activity.
SQLSRV_LOG_SYSTEM_INIT (1)	Turns on logging of initialization activity.
SQLSRV_LOG_SYSTEM_OFF (0)	Turns logging off.
SQLSRV_LOG_SYSTEM_STMT (4)	Turns on logging of statement activity.
SQLSRV_LOG_SYSTEM_UTIL (8)	Turns on logging of error functions activity (such as **handle_error** and **handle_warning**).

The following table lists the constants that can be used as the value for the **LogSeverity** setting:

SQLSRV CONSTANT (INTEGER EQUIVALENT IN PARENTHESES)	DESCRIPTION
SQLSRV_LOG_SEVERITY_ALL (-1)	Specifies that errors, warnings, and notices will be logged.
SQLSRV_LOG_SEVERITY_ERROR (1)	Specifies that errors will be logged.
SQLSRV_LOG_SEVERITY_NOTICE (4)	Specifies that notices will be logged.
SQLSRV_LOG_SEVERITY_WARNING (2)	Specifies that warnings will be logged.

Nullable Constants

The following table lists the constants that you can use to determine whether or not a column is nullable or if this information is not available. You can compare the value of the **Nullable** key that is returned by sqlsrv_field_metadata to determine the column's nullable status.

SQLSRV CONSTANT (INTEGER EQUIVALENT IN PARENTHESES)	DESCRIPTION
SQLSRV_NULLABLE_YES (0)	The column is nullable.
SQLSRV_NULLABLE_NO (1)	The column is not nullable.

SQLSRV CONSTANT (INTEGER EQUIVALENT IN PARENTHESES)	DESCRIPTION
SQLSRV_NULLABLE_UNKNOWN (2)	It is not known if the column is nullable.

PARAM Constants

The following list contains the constants for specifying parameter direction when you call sqlsrv_query or sqlsrv_prepare.

SQLSRV CONSTANT	DESCRIPTION
SQLSRV_PARAM_IN	Indicates an input parameter.
SQLSRV_PARAM_INOUT	Indicates a bidirectional parameter.
SQLSRV_PARAM_OUT	Indicates an output parameter.

PHPTYPE Constants

The following table lists the constants that are used to describe PHP data types. For information about PHP data types, see PHP Types.

SQLSRV CONSTANT	PHP DATA TYPE
SQLSRV_PHPTYPE_INT	Integer
SQLSRV_PHPTYPE_DATETIME	Datetime
SQLSRV_PHPTYPE_FLOAT	Float
SQLSRV_PHPTYPE_STREAM($encoding[1])	Stream
SQLSRV_PHPTYPE_STRING($encoding[1])	String

1. **SQLSRV_PHPTYPE_STREAM** and **SQLSRV_PHPTYPE_STRING** accept a parameter that specifies the stream encoding. The following table contains the SQLSRV constants that are acceptable parameters, and a description of the corresponding encoding.

SQLSRV CONSTANT	DESCRIPTION
SQLSRV_ENC_BINARY	Data is returned as a raw byte stream from the server without performing encoding or translation.
SQLSRV_ENC_CHAR	Data is returned in 8-bit characters as specified in the code page of the Windows locale that is set on the system. Any multi-byte characters or characters that do not map into this code page are substituted with a single byte question mark (?) character. This is the default encoding.
"UTF-8"	Data is returned in the UTF-8 encoding. This constant was added in version 1.1 of the Microsoft Drivers for PHP for SQL Server. For more information about UTF-8 support, see How to: Send and Retrieve UTF-8 Data Using Built-In UTF-8 Support.

NOTE

When you use **SQLSRV_PHPTYPE_STREAM** or **SQLSRV_PHPTYPE_STRING**, the encoding must be specified. If no parameter is supplied, an error will be returned.

For more information about these constants, see How to: Specify PHP Data Types, How to: Retrieve Character Data as a Stream Using the SQLSRV Driver.

SQLTYPE Constants

The following table lists the constants that are used to describe SQL Server data types. Some constants are function-like and may take parameters that correspond to precision, scale, and/or length. When binding parameters, the function-like constants should be used. For type comparisons, the standard (non function-like) constants are required. For information about SQL Server data types, see Data Types (Transact-SQL). For information about precision, scale, and length, see Precision, Scale, and Length (Transact-SQL).

SQLSRV CONSTANT	SQL SERVER DATA TYPE
SQLSRV_SQLTYPE_BIGINT	bigint
SQLSRV_SQLTYPE_BINARY	binary
SQLSRV_SQLTYPE_BIT	bit
SQLSRV_SQLTYPE_CHAR	char[5]
SQLSRV_SQLTYPE_CHAR($charCount)	char
SQLSRV_SQLTYPE_DATE	date[4]
SQLSRV_SQLTYPE_DATETIME	datetime
SQLSRV_SQLTYPE_DATETIME2	datetime2[4]
SQLSRV_SQLTYPE_DATETIMEOFFSET	datetimeoffset[4]
SQLSRV_SQLTYPE_DECIMAL	decimal[5]
SQLSRV_SQLTYPE_DECIMAL($precision, $scale)	decimal
SQLSRV_SQLTYPE_FLOAT	float
SQLSRV_SQLTYPE_IMAGE	image[1]
SQLSRV_SQLTYPE_INT	int
SQLSRV_SQLTYPE_MONEY	money
SQLSRV_SQLTYPE_NCHAR	nchar[5]
SQLSRV_SQLTYPE_NCHAR($charCount)	nchar
SQLSRV_SQLTYPE_NUMERIC	numeric[5]

SQLSRV CONSTANT	SQL SERVER DATA TYPE
SQLSRV_SQLTYPE_NUMERIC($precision, $scale)	numeric
SQLSRV_SQLTYPE_NVARCHAR	nvarchar[5]
SQLSRV_SQLTYPE_NVARCHAR($charCount)	nvarchar
SQLSRV_SQLTYPE_NVARCHAR('max')	nvarchar(MAX)
SQLSRV_SQLTYPE_NTEXT	ntext[2]
SQLSRV_SQLTYPE_REAL	real
SQLSRV_SQLTYPE_SMALLDATETIME	smalldatetime
SQLSRV_SQLTYPE_SMALLINT	smallint
SQLSRV_SQLTYPE_SMALLMONEY	smallmoney
SQLSRV_SQLTYPE_TEXT	text[3]
SQLSRV_SQLTYPE_TIME	time[4]
SQLSRV_SQLTYPE_TIMESTAMP	timestamp
SQLSRV_SQLTYPE_TINYINT	tinyint
SQLSRV_SQLTYPE_UNIQUEIDENTIFIER	uniqueidentifier
SQLSRV_SQLTYPE_UDT	UDT
SQLSRV_SQLTYPE_VARBINARY	varbinary[5]
SQLSRV_SQLTYPE_VARBINARY($byteCount)	varbinary
SQLSRV_SQLTYPE_VARBINARY('max')	varbinary(MAX)
SQLSRV_SQLTYPE_VARCHAR	varchar[5]
SQLSRV_SQLTYPE_VARCHAR($charCount)	varchar
SQLSRV_SQLTYPE_VARCHAR('max')	varchar(MAX)
SQLSRV_SQLTYPE_XML	xml

1. This is a legacy type that maps to the varbinary(max) type.
2. This is a legacy type that maps to the newer nvarchar type.
3. This is a legacy type that maps to the newer varchar type.
4. Support for this type was added in version 1.1 of the Microsoft Drivers for PHP for SQL Server.

5. These constants should be used in type comparison operations and don't replace the function-like constants with similar syntax. For binding parameters, you should use the function-like constants.

The following table lists the SQLTYPE constants that accept parameters and the range of values allowed for the parameter.

SQLTYPE	PARAMETER	ALLOWABLE RANGE FOR PARAMETER
SQLSRV_SQLTYPE_CHAR, SQLSRV_SQLTYPE_VARCHAR	charCount	1 - 8000
SQLSRV_SQLTYPE_NCHAR, SQLSRV_SQLTYPE_NVARCHAR	charCount	1 - 4000
SQLSRV_SQLTYPE_BINARY, SQLSRV_SQLTYPE_VARBINARY	byteCount	1 - 8000
SQLSRV_SQLTYPE_DECIMAL, SQLSRV_SQLTYPE_NUMERIC	precision	1 - 38
SQLSRV_SQLTYPE_DECIMAL, SQLSRV_SQLTYPE_NUMERIC	scale	1 - precision

Transaction Isolation Level Constants

The **TransactionIsolation** key, which is used with sqlsrv_connect, accepts one of the following constants:

- SQLSRV_TXN_READ_UNCOMMITTED
- SQLSRV_TXN_READ_COMMITTED
- SQLSRV_TXN_REPEATABLE_READ
- SQLSRV_TXN_SNAPSHOT
- SQLSRV_TXN_SERIALIZABLE

Cursor and Scrolling Constants

The following constants specify the kind of cursor that you can use in a result set:

- SQLSRV_CURSOR_FORWARD
- SQLSRV_CURSOR_STATIC
- SQLSRV_CURSOR_DYNAMIC
- SQLSRV_CURSOR_KEYSET
- SQLSRV_CURSOR_CLIENT_BUFFERED

The following constants specify which row to select in the result set:

- SQLSRV_SCROLL_NEXT
- SQLSRV_SCROLL_PRIOR
- SQLSRV_SCROLL_FIRST

- SQLSRV_SCROLL_LAST
- SQLSRV_SCROLL_ABSOLUTE
- SQLSRV_SCROLL_RELATIVE

For information on using these constants, see Specifying a Cursor Type and Selecting Rows.

See Also

SQLSRV Driver API Reference

SQLSRV Driver API Reference

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The API name for the SQLSRV driver in the Microsoft Drivers for PHP for SQL Server is **sqlsrv**. All **sqlsrv** functions begin with **sqlsrv_** and are followed by a verb or a noun. Those followed by a verb perform some action and those followed by a noun return some form of metadata.

In This Section

The SQLSRV driver contains the following functions:

FUNCTION	DESCRIPTION
sqlsrv_begin_transaction	Begins a transaction.
sqlsrv_cancel	Cancels a statement; discards any pending results for the statement.
sqlsrv_client_info	Provides information about the client.
sqlsrv_close	Closes a connection. Frees all resources associated with the connection.
sqlsrv_commit	Commits a transaction.
sqlsrv_configure	Changes error handling and logging configurations.
sqlsrv_connect	Creates and opens a connection.
sqlsrv_errors	Returns error and/or warning information about the last operation.
sqlsrv_execute	Executes a prepared statement.
sqlsrv_fetch	Makes the next row of data available for reading.
sqlsrv_fetch_array	Retrieves the next row of data as a numerically indexed array, an associative array, or both.
sqlsrv_fetch_object	Retrieves the next row of data as an object.
sqlsrv_field_metadata	Returns field metadata.
sqlsrv_free_stmt	Closes a statement. Frees all resources associated with the statement.
sqlsrv_get_config	Returns the value of the specified configuration setting.

FUNCTION	DESCRIPTION
sqlsrv_get_field	Retrieves a field in the current row by index. The PHP return type can be specified.
sqlsrv_has_rows	Detects if a result set has one or more rows.
sqlsrv_next_result	Makes the next result available for processing.
sqlsrv_num_rows	Reports the number of rows in a result set.
sqlsrv_num_fields	Retrieves the number of fields in an active result set.
sqlsrv_prepare	Prepares a Transact-SQL query without executing it. Implicitly binds parameters.
sqlsrv_query	Prepares and executes a Transact-SQL query.
sqlsrv_rollback	Rolls back a transaction.
sqlsrv_rows_affected	Returns the number of modified rows.
sqlsrv_send_stream_data	Sends up to eight kilobytes (8 KB) of data to the server with each call to the function.
sqlsrv_server_info	Provides information about the server.

Reference

PHP Manual

See Also

Overview of the PHP SQL Driver Constants (Microsoft Drivers for PHP for SQL Server)
Programming Guide for PHP SQL Driver Getting Started with the PHP SQL Driver

sqlsrv_begin_transaction

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

Begins a transaction on a specified connection. The current transaction includes all statements on the specified connection that were executed after the call to **sqlsrv_begin_transaction** and before any calls to sqlsrv_rollback or sqlsrv_commit.

NOTE

The Microsoft Drivers for PHP for SQL Server is in auto-commit mode by default. This means that all queries are automatically committed upon success unless they have been designated as part of an explicit transaction by using **sqlsrv_begin_transaction**.

NOTE

If **sqlsrv_begin_transaction** is called after a transaction has already been initiated on the connection but not completed by calling either **sqlsrv_commit** or **sqlsrv_rollback**, the call returns **false** and an *Already in Transaction* error is added to the error collection.

Syntax

```
sqlsrv_begin_transaction( resource $conn)
```

Parameters

$conn: The connection with which the transaction is associated.

Return Value

A Boolean value: **true** if the transaction was successfully begun. Otherwise, **false**.

Example

The example below executes two queries as part of a transaction. If both queries are successful, the transaction is committed. If either (or both) of the queries fail, the transaction is rolled back.

The first query in the example inserts a new sales order into the *Sales.SalesOrderDetail* table of the AdventureWorks database. The order is for five units of the product that has product ID 709. The second query reduces the inventory quantity of product ID 709 by five units. These queries are included in a transaction because both queries must be successful for the database to accurately reflect the state of orders and product availability.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```php
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true ));
}

/* Initiate transaction. */
/* Exit script if transaction cannot be initiated. */
if ( sqlsrv_begin_transaction( $conn ) === false )
{
     echo "Could not begin transaction.\n";
     die( print_r( sqlsrv_errors(), true ));
}

/* Initialize parameter values. */
$orderId = 43659; $qty = 5; $productId = 709;
$offerId = 1; $price = 5.70;

/* Set up and execute the first query. */
$tsql1 = "INSERT INTO Sales.SalesOrderDetail
                     (SalesOrderID,
                      OrderQty,
                      ProductID,
                      SpecialOfferID,
                      UnitPrice)
          VALUES (?, ?, ?, ?, ?)";
$params1 = array( $orderId, $qty, $productId, $offerId, $price);
$stmt1 = sqlsrv_query( $conn, $tsql1, $params1 );

/* Set up and execute the second query. */
$tsql2 = "UPDATE Production.ProductInventory
          SET Quantity = (Quantity - ?)
          WHERE ProductID = ?";
$params2 = array($qty, $productId);
$stmt2 = sqlsrv_query( $conn, $tsql2, $params2 );

/* If both queries were successful, commit the transaction. */
/* Otherwise, rollback the transaction. */
if( $stmt1 && $stmt2 )
{
     sqlsrv_commit( $conn );
     echo "Transaction was committed.\n";
}
else
{
     sqlsrv_rollback( $conn );
     echo "Transaction was rolled back.\n";
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt1);
sqlsrv_free_stmt( $stmt2);
sqlsrv_close( $conn);
?>
```

For the purpose of focusing on transaction behavior, some recommended error handling is not included in the example above. For a production application it is recommended that any call to a **sqlsrv** function be checked for errors and handled accordingly.

NOTE

Do not use embedded Transact-SQL to perform transactions. For example, do not execute a statement with "BEGIN TRANSACTION" as the Transact-SQL query to begin a transaction. The expected transactional behavior cannot be guaranteed when using embedded Transact-SQL to perform transactions.

See Also

SQLSRV Driver API Reference
How to: Perform Transactions
Overview of the PHP SQL Driver

sqlsrv_cancel

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Cancels a statement. This means that any pending results for the statement are discarded. After this function is called, the statement can be re-executed if it was prepared with sqlsrv_prepare. Calling this function is not necessary if all the results associated with the statement have been consumed.

Syntax

```
sqlsrv_cancel( resource $stmt)
```

Parameters

$stmt: The statement to be canceled.

Return Value

A Boolean value: **true** if the operation was successful. Otherwise, **false**.

Example

The following example targets the AdventureWorks database to execute a query, then consumes and counts results until the variable *$salesTotal* reaches a specified amount. The remaining query results are then discarded. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Prepare and execute the query. */
$tsql = "SELECT OrderQty, UnitPrice FROM Sales.SalesOrderDetail";
$stmt = sqlsrv_prepare( $conn, $tsql);
if( $stmt === false )
{
     echo "Error in statement preparation.\n";
     die( print_r( sqlsrv_errors(), true));
}
if( sqlsrv_execute( $stmt ) === false)
{
     echo "Error in statement execution.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Initialize tracking variables. */
$salesTotal = 0;
$count = 0;

/* Count and display the number of sales that produce revenue
of $100,000. */
while( ($row = sqlsrv_fetch_array( $stmt)) && $salesTotal <=100000)
{
     $qty = $row[0];
     $price = $row[1];
     $salesTotal += ( $price * $qty);
     $count++;
}
echo "$count sales accounted for the first $$salesTotal in revenue.\n";

/* Cancel the pending results. The statement can be reused. */
sqlsrv_cancel( $stmt);
?>
```

Comments

A statement that is prepared and executed using the combination of sqlsrv_prepare and sqlsrv_execute can be re-executed with **sqlsrv_execute** after calling **sqlsrv_cancel**. A statement that is executed with sqlsrv_query cannot be re-executed after calling **sqlsrv_cancel**.

See Also

SQLSRV Driver API Reference
Connecting to the Server
Retrieving Data
About Code Examples in the Documentation
sqlsrv_free_stmt

sqlsrv_client_info

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Returns information about the connection and client stack.

Syntax

```
sqlsrv_client_info( resource $conn)
```

Parameters

$conn: The connection resource by which the client is connected.

Return Value

An associative array with keys described in the table below, or **false** if the connection resource is null.

For PHP for SQL Server versions 3.2 and 3.1:

KEY	DESCRIPTION
DriverDllName	MSODBCSQL11.DLL (ODBC Driver 11 for SQL Server)
DriverODBCVer	ODBC version (xx.yy)
DriverVer	ODBC Driver 11 for SQL Server DLL version: xx.yy.zzzz (Microsoft Drivers for PHP for SQL Server version 3.2 or 3.1)
ExtensionVer	php_sqlsrv.dll version: 3.2.xxxx.x (for Microsoft Drivers for PHP for SQL Server version 3.2) 3.1.xxxx.x (for Microsoft Drivers for PHP for SQL Server version 3.1)

For PHP for SQL Server versions 3.0 and 2.0:

KEY	DESCRIPTION
DriverDllName	SQLNCLI10.DLL (Microsoft Drivers for PHP for SQL Server version 2.0)
DriverODBCVer	ODBC version (xx.yy)

KEY	DESCRIPTION
DriverVer	SQL Server Native Client DLL version: 10.50.xxx (Microsoft Drivers for PHP for SQL Server version 2.0)
ExtensionVer	php_sqlsrv.dll version: 2.0.xxxx.x (Microsoft Drivers for PHP for SQL Server version 2.0)

Example

The following example writes client information to the console when the example is run from the command line. The example assumes that SQL Server is installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/*Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$conn = sqlsrv_connect( $serverName);

if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

if( $client_info = sqlsrv_client_info( $conn))
{
       foreach( $client_info as $key => $value)
      {
              echo $key.": ".$value."\n";
      }
}
else
{
       echo "Client info error.\n";
}

/* Close connection resources. */
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference
About Code Examples in the Documentation

sqlsrv_close

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Closes the specified connection and releases associated resources.

Syntax

```
sqlsrv_close( resource $conn )
```

Parameters

$conn: The connection to be closed.

Return Value

The Boolean value **true** unless the function is called with an invalid parameter. If the function is called with an invalid parameter, **false** is returned.

NOTE

Null is a valid parameter for this function. This allows the function to be called multiple times in a script. For example, if you close a connection in an error condition and close it again at the end of the script, the second call to **sqlsrv_close** will return **true** because the first call to **sqlsrv_close** (in the error condition) sets the connection resource to **null**.

Example

The following example closes a connection. The example assumes that SQL Server is installed on the local computer. All output is writing to the console when the example is run from the command line.

```
<?php
/*Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$conn = sqlsrv_connect( $serverName);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

//-----------------------------------------------
// Perform operations with connection here.
//-----------------------------------------------

/* Close the connection. */
sqlsrv_close( $conn);
echo "Connection closed.\n";
?>
```

See Also

SQLSRV Driver API Reference

About Code Examples in the Documentation

sqlsrv_commit

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

Commits the current transaction on the specified connection and returns the connection to the auto-commit mode. The current transaction includes all statements on the specified connection that were executed after the call to sqlsrv_begin_transaction and before any calls to sqlsrv_rollback or **sqlsrv_commit**.

NOTE

The Microsoft Drivers for PHP for SQL Server is in auto-commit mode by default. This means that all queries are automatically committed upon success unless they have been designated as part of an explicit transaction by using **sqlsrv_begin_transaction**.

NOTE

If **sqlsrv_commit** is called on a connection that is not in an active transaction and that was initiated with **sqlsrv_begin_transaction**, the call returns **false** and a *Not in Transaction* error is added to the error collection.

Syntax

```
sqlsrv_commit( resource $conn )
```

Parameters

$conn: The connection on which the transaction is active.

Return Value

A Boolean value: **true** if the transaction was successfully committed. Otherwise, **false**.

Example

The example below executes two queries as part of a transaction. If both queries are successful, the transaction is committed. If either (or both) of the queries fail, the transaction is rolled back.

The first query in the example inserts a new sales order into the *Sales.SalesOrderDetail* table of the AdventureWorks database. The order is for five units of the product that has product ID 709. The second query reduces the inventory quantity of product ID 709 by five units. These queries are included in a transaction because both queries must be successful for the database to accurately reflect the state of orders and product availability.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true ));
}

/* Initiate transaction. */
/* Exit script if transaction cannot be initiated. */
if (sqlsrv_begin_transaction( $conn) === false)
{
     echo "Could not begin transaction.\n";
     die( print_r( sqlsrv_errors(), true ));
}

/* Initialize parameter values. */
$orderId = 43659; $qty = 5; $productId = 709;
$offerId = 1; $price = 5.70;

/* Set up and execute the first query. */
$tsql1 = "INSERT INTO Sales.SalesOrderDetail
                     (SalesOrderID,
                      OrderQty,
                      ProductID,
                      SpecialOfferID,
                      UnitPrice)
          VALUES (?, ?, ?, ?, ?)";
$params1 = array( $orderId, $qty, $productId, $offerId, $price);
$stmt1 = sqlsrv_query( $conn, $tsql1, $params1 );

/* Set up and execute the second query. */
$tsql2 = "UPDATE Production.ProductInventory
          SET Quantity = (Quantity - ?)
          WHERE ProductID = ?";
$params2 = array($qty, $productId);
$stmt2 = sqlsrv_query( $conn, $tsql2, $params2 );

/* If both queries were successful, commit the transaction. */
/* Otherwise, rollback the transaction. */
if( $stmt1 && $stmt2 )
{
     sqlsrv_commit( $conn );
     echo "Transaction was committed.\n";
}
else
{
     sqlsrv_rollback( $conn );
     echo "Transaction was rolled back.\n";
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt1);
sqlsrv_free_stmt( $stmt2);
sqlsrv_close( $conn);
?>
```

For the purpose of focusing on transaction behavior, some recommended error handling is not included in the example above. For a production application it is recommended that any call to a **sqlsrv** function be checked for errors and handled accordingly.

NOTE

Do not use embedded Transact-SQL to perform transactions. For example, do not execute a statement with "BEGIN TRANSACTION" as the Transact-SQL query to begin a transaction. The expected transactional behavior cannot be guaranteed when using embedded Transact-SQL to perform transactions.

See Also

SQLSRV Driver API Reference
How to: Perform Transactions
Overview of the PHP SQL Driver

sqlsrv_configure

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Changes the settings for error handling and logging options.

Syntax

```
sqlsrv_configure( string $setting, mixed $value )
```

Parameters

$setting: The name of the setting to be configured. See table below for list of settings.

$value: The value to be applied to the setting specified in the *$setting* parameter. The possible values for this parameter depend on which setting is specified. The following table lists the possible combinations:

SETTING	POSSIBLE VALUES FOR $VALUE PARAMETER (INTEGER EQUIVALENT IN PARENTHESES)	DEFAULT VALUE
ClientBufferMaxKBSize[1]	A non negative number up to the PHP memory limit. Zero (0) means no limit to the buffer size.	10240
LogSeverity[2]	SQLSRV_LOG_SEVERITY_ALL (-1) SQLSRV_LOG_SEVERITY_ERROR (1) SQLSRV_LOG_SEVERITY_NOTICE (4) SQLSRV_LOG_SEVERITY_WARNING (2)	SQLSRV_LOG_SEVERITY_ERROR (1)
LogSubsystems[2]	SQLSRV_LOG_SYSTEM_ALL (-1) SQLSRV_LOG_SYSTEM_CONN (2) SQLSRV_LOG_SYSTEM_INIT (1) SQLSRV_LOG_SYSTEM_OFF (0) SQLSRV_LOG_SYSTEM_STMT (4) SQLSRV_LOG_SYSTEM_UTIL (8)	SQLSRV_LOG_SYSTEM_OFF (0)
WarningsReturnAsErrors[3]	**true** (1) or **false** (0)	**true** (1)

Return Value

If **sqlsrv_configure** is called with an unsupported setting or value, the function returns **false**. Otherwise, the function returns **true**.

Remarks

(1) For more information about client-side queries, see Cursor Types (SQLSRV Driver).

(2) For more information about logging activity, see Logging Activity.

(3) For more information about configuring error and warning handling, see How to: Configure Error and Warning Handling Using the SQLSRV Driver.

See Also

SQLSRV Driver API Reference
Programming Guide for PHP SQL Driver

sqlsrv_connect

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

Creates a connection resource and opens a connection. By default, the connection is attempted using Windows Authentication.

Syntax

```
sqlsrv_connect( string $serverName [, array $connectionInfo])
```

Parameters

$serverName: A string specifying the name of the server to which a connection is being established. An instance name (for example, "myServer\instanceName") or port number (for example, "myServer, 1521") can be included as part of this string. For a complete description of the options available for this parameter, see the Server keyword in the ODBC Driver Connection String Keywords section of Using Connection String Keywords with SQL Native Client.

Beginning in version 3.0 of the Microsoft Drivers for PHP for SQL Server, you can also specify a LocalDB instance with `"(localdb)\instancename"`. For more information, see PHP Driver for SQL Server Support for LocalDB.

Also beginning in version 3.0 of the Microsoft Drivers for PHP for SQL Server, you can specify a virtual network name, to connect to an AlwaysOn availability group. For more information about Microsoft Drivers for PHP for SQL Server support for AlwaysOn Availability Groups , see PHP Driver for SQL Server Support for High Availability, Disaster Recovery.

$connectionInfo [OPTIONAL]: An associative **array** that contains connection attributes (for example, **array**("Database" => "AdventureWorks")). See Connection Options for a list of the supported keys for the array.

Return Value

A PHP connection resource. If a connection cannot be successfully created and opened, **false** is returned.

Remarks

If values for the *UID* and *PWD* keys are not specified in the optional *$connectionInfo* parameter, the connection will be attempted using Windows Authentication. For more information about connecting to the server, see How to: Connect Using Windows Authentication and How to: Connect Using SQL Server Authentication.

Example

The following example creates and opens a connection using Windows Authentication. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```php
<?php
/*
Connect to the local server using Windows Authentication and specify
the AdventureWorks database as the database in use. To connect using
SQL Server Authentication, set values for the "UID" and "PWD"
 attributes in the $connectionInfo parameter. For example:
$connectionInfo = array("UID" => $uid, "PWD" => $pwd, "Database"=>"AdventureWorks");
*/
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);

if( $conn )
{
     echo "Connection established.\n";
}
else
{
     echo "Connection could not be established.\n";
     die( print_r( sqlsrv_errors(), true));
}

//-----------------------------------------------
// Perform operations with connection.
//-----------------------------------------------

/* Close the connection. */
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference
Connecting to the Server
About Code Examples in the Documentation

sqlsrv_errors

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

Returns extended error and/or warning information about the last **sqlsrv** operation performed.

The **sqlsrv_errors** function can return error and/or warning information by calling it with one of the parameter values specified in the Parameters section below.

By default, warnings generated on a call to any **sqlsrv** function are treated as errors; if a warning occurs on a call to a **sqlsrv** function, the function returns false. However, warnings that correspond to SQLSTATE values 01000, 01001, 01003, and 01S02 are never treated as errors.

The following line of code turns off the behavior mentioned above; a warning generated by a call to a **sqlsrv** function does not cause the function to return false:

```
sqlsrv_configure("WarningsReturnAsErrors", 0);
```

The following line of code reinstates the default behavior; warnings (with exceptions, noted above) are treated as errors:

```
sqlsrv_configure("WarningsReturnAsErrors", 1);
```

Regardless of the setting, warnings can only be retrieved by calling **sqlsrv_errors** with either the **SQLSRV_ERR_ALL** or **SQLSRV_ERR_WARNINGS** parameter value (see Parameters section below for details).

Syntax

```

sqlsrv_errors( [int $errorsAndOrWarnings] )
```

Parameters

$errorsAndOrWarnings[OPTIONAL]: A predefined constant. This parameter can take one of the values listed in the following table:

VALUE	DESCRIPTION
SQLSRV_ERR_ALL	Errors and warnings generated on the last **sqlsrv** function call are returned.
SQLSRV_ERR_ERRORS	Errors generated on the last **sqlsrv** function call are returned.
SQLSRV_ERR_WARNINGS	Warnings generated on the last **sqlsrv** function call are returned.

If no parameter value is supplied, both errors and warnings generated by the last **sqlsrv** function call are returned.

Return Value

An **array** of arrays, or **null**. Each **array** in the returned **array** contains three key-value pairs. The following table lists each key and its description:

KEY	DESCRIPTION
SQLSTATE	For errors that originate from the ODBC driver, the SQLSTATE returned by ODBC. For information about SQLSTATE values for ODBC, see ODBC Error Codes. For errors that originate from the Microsoft Drivers for PHP for SQL Server, a SQLSTATE of IMSSP. For warnings that originate from the Microsoft Drivers for PHP for SQL Server, a SQLSTATE of 01SSP.
code	For errors that originate from SQL Server, the native SQL Server error code. For errors that originate from the ODBC driver, the error code returned by ODBC. For errors that originate from the Microsoft Drivers for PHP for SQL Server, the Microsoft Drivers for PHP for SQL Server error code. For more information, see Handling Errors and Warnings.
message	A description of the error.

The array values can also be accessed with numeric keys 0, 1, and 2. If no errors or warnings occur, **null** is returned.

Example

The following example displays errors that occur during a failed statement execution. (The statement fails because **InvalidColumName** is not a valid column name in the specified table.) The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up a query to select an invalid column name. */
$tsql = "SELECT InvalidColumnName FROM Sales.SalesOrderDetail";

/* Attempt execution. */
/* Execution will fail because of the invalid column name. */
$stmt = sqlsrv_query( $conn, $tsql);
if( $stmt === false )
{
      if( ($errors = sqlsrv_errors() ) != null)
      {
         foreach( $errors as $error)
         {
            echo "SQLSTATE: ".$error[ 'SQLSTATE']."\n";
            echo "code: ".$error[ 'code']."\n";
            echo "message: ".$error[ 'message']."\n";
         }
      }
}

/* Free connection resources */
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference
About Code Examples in the Documentation

sqlsrv_execute

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Executes a previously prepared statement. See sqlsrv_prepare for information on preparing a statement for execution.

NOTE

This function is ideal for executing a prepared statement multiple times with different parameter values.

Syntax

```
sqlsrv_execute( resource $stmt)
```

Parameters

$stmt: A resource specifying the statement to be executed. For more information about how to create a statement resource, see sqlsrv_prepare.

Return Value

A Boolean value: **true** if the statement was successfully executed. Otherwise, **false**.

Example

The following example executes a statement that updates a field in the *Sales.SalesOrderDetail* table in the AdventureWorks database. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/*Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false)
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up the Transact-SQL query. */
$tsql = "UPDATE Sales.SalesOrderDetail
         SET OrderQty = ( ?)
         WHERE SalesOrderDetailID = ( ?)";

/* Set up the parameters array. Parameters correspond, in order, to
question marks in $tsql. */
$params = array( 5, 10);

/* Create the statement. */
$stmt = sqlsrv_prepare( $conn, $tsql, $params);
if( $stmt )
{
     echo "Statement prepared.\n";
}
else
{
     echo "Error in preparing statement.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Execute the statement. Display any errors that occur. */
if( sqlsrv_execute( $stmt))
{
      echo "Statement executed.\n";
}
else
{
     echo "Error in executing statement.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Free the statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference
About Code Examples in the Documentation
sqlsrv_query

sqlsrv_fetch

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

Makes the next row of a result set available for reading. Use sqlsrv_get_field to read fields of the row.

Syntax

```
sqlsrv_fetch( resource $stmt[, row[, ]offset])
```

Parameters

$stmt: A statement resource corresponding to an executed statement.

> **NOTE**
>
> A statement must be executed before results can be retrieved. For information on executing a statement, see sqlsrv_query and sqlsrv_execute.

row [OPTIONAL]: One of the following values, specifying the row to access in a result set that uses a scrollable cursor:

- SQLSRV_SCROLL_NEXT
- SQLSRV_SCROLL_PRIOR
- SQLSRV_SCROLL_FIRST
- SQLSRV_SCROLL_LAST
- SQLSRV_SCROLL_ABSOLUTE
- SQLSRV_SCROLL_RELATIVE

For more information on these values, see Specifying a Cursor Type and Selecting Rows.

offset [OPTIONAL]: Used with SQLSRV_SCROLL_ABSOLUTE and SQLSRV_SCROLL_RELATIVE to specify the row to retrieve. The first record in the result set is 0.

Return Value

If the next row of the result set was successfully retrieved, **true** is returned. If there are no more results in the result set, **null** is returned. If an error occurred, **false** is returned.

Example

The following example uses **sqlsrv_fetch** to retrieve a row of data containing a product review and the name of the reviewer. To retrieve data from the result set, sqlsrv_get_field is used. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/*Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up and execute the query. Note that both ReviewerName and
Comments are of SQL Server type nvarchar. */
$tsql = "SELECT ReviewerName, Comments
         FROM Production.ProductReview
         WHERE ProductReviewID=1";
$stmt = sqlsrv_query( $conn, $tsql);
if( $stmt === false )
{
     echo "Error in statement preparation/execution.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Make the first row of the result set available for reading. */
if( sqlsrv_fetch( $stmt ) === false)
{
     echo "Error in retrieving row.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Note: Fields must be accessed in order.
Get the first field of the row. Note that no return type is
specified. Data will be returned as a string, the default for
a field of type nvarchar.*/
$name = sqlsrv_get_field( $stmt, 0);
echo "$name: ";

/*Get the second field of the row as a stream.
Because the default return type for a nvarchar field is a
string, the return type must be specified as a stream. */
$stream = sqlsrv_get_field( $stmt, 1,
                            SQLSRV_PHPTYPE_STREAM( SQLSRV_ENC_CHAR));
while( !feof( $stream ))
{
    $str = fread( $stream, 10000);
    echo $str;
}

/* Free the statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

See Also

Retrieving Data
SQLSRV Driver API Reference
About Code Examples in the Documentation

sqlsrv_fetch_array

3/14/2017 • 4 min to read • Edit on GitHub

Download PHP Driver

Retrieves the next row of data as a numerically indexed array, associative array, or both.

Syntax

```
sqlsrv_fetch_array( resource $stmt[, int $fetchType [, row[, ]offset]])
```

Parameters

$stmt: A statement resource corresponding to an executed statement.

$fetchType [OPTIONAL]: A predefined constant. This parameter can take on one of the values listed in the following table:

VALUE	DESCRIPTION
SQLSRV_FETCH_NUMERIC	The next row of data is returned as a numeric array.
SQLSRV_FETCH_ASSOC	The next row of data is returned as an associative array. The array keys are the column names in the result set.
SQLSRV_FETCH_BOTH	The next row of data is returned as both a numeric array and an associative array. This is the default value.

row [OPTIONAL]: Added in version 1.1. One of the following values, specifying the row to access in a result set that uses a scrollable cursor. (When *row* is specified, *fetchtype* must be explicitly specified, even if you specify the default value.)

- SQLSRV_SCROLL_NEXT
- SQLSRV_SCROLL_PRIOR
- SQLSRV_SCROLL_FIRST
- SQLSRV_SCROLL_LAST
- SQLSRV_SCROLL_ABSOLUTE
- SQLSRV_SCROLL_RELATIVE

For more information about these values, see Specifying a Cursor Type and Selecting Rows. Scrollable cursor support was added in version 1.1 of the Microsoft Drivers for PHP for SQL Server.

offset [OPTIONAL]: Used with SQLSRV_SCROLL_ABSOLUTE and SQLSRV_SCROLL_RELATIVE to specify the row to retrieve. The first record in the result set is 0.

Return Value

If a row of data is retrieved, an **array** is returned. If there are no more rows to retrieve, **null** is returned. If an error occurs, **false** is returned.

Based on the value of the *$fetchType* parameter, the returned **array** can be a numerically indexed **array**, an associative **array**, or both. By default, an **array** with both numeric and associative keys is returned. The data type of a value in the returned array will be the default PHP data type. For information about default PHP data types, see Default PHP Data Types.

Remarks

If a column with no name is returned, the associative key for the array element will be an empty string (""). For example, consider this Transact-SQL statement that inserts a value into a database table and retrieves the server-generated primary key:

```
INSERT INTO Production.ProductPhoto (LargePhoto) VALUES (?);
SELECT SCOPE_IDENTITY()
```

If the result set returned by the `SELECT SCOPE_IDENTITY()` portion of this statement is retrieved as an associative array, the key for the returned value will be an empty string ("") because the returned column has no name. To avoid this, you can retrieve the result as a numeric array, or you can specify a name for the returned column in the Transact-SQL statement. The following is one way to specify a column name in Transact-SQL:

```
SELECT SCOPE_IDENTITY() AS PictureID
```

If a result set contains multiple columns without names, the value of the last unnamed column will be assigned to the empty string ("") key.

Example

The following example retrieves each row of a result set as an associative **array**. The example assumes that the SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up and execute the query. */
$tsql = "SELECT FirstName, LastName
         FROM Person.Contact
         WHERE LastName='Alan'";
$stmt = sqlsrv_query( $conn, $tsql);
if( $stmt === false)
{
     echo "Error in query preparation/execution.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve each row as an associative array and display the results.*/
while( $row = sqlsrv_fetch_array( $stmt, SQLSRV_FETCH_ASSOC))
{
      echo $row['LastName'].", ".$row['FirstName']."\n";
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

Example

The following example retrieves each row of a result set as a numerically indexed array.

The example retrieves product information from the *Purchasing.PurchaseOrderDetail* table of the AdventureWorks database for products that have a specified date and a stocked quantity (*StockQty*) less than a specified value.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```php
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Define the query. */
$tsql = "SELECT ProductID,
                UnitPrice,
                StockedQty
         FROM Purchasing.PurchaseOrderDetail
         WHERE StockedQty < 3
         AND DueDate='2002-01-29'";

/* Execute the query. */
$stmt = sqlsrv_query( $conn, $tsql);
if ( $stmt )
{
     echo "Statement executed.\n";
}
else
{
     echo "Error in statement execution.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Iterate through the result set printing a row of data upon each
iteration.*/
while( $row = sqlsrv_fetch_array( $stmt, SQLSRV_FETCH_NUMERIC))
{
     echo "ProdID: ".$row[0]."\n";
     echo "UnitPrice: ".$row[1]."\n";
     echo "StockedQty: ".$row[2]."\n";
     echo "-----------------\n";
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

The **sqlsrv_fetch_array** function always returns data according to the Default PHP Data Types. For information about how to specify the PHP data type, see How to: Specify PHP Data Types.

If a field with no name is retrieved, the associative key for the array element will be an empty string (""). For more information, see sqlsrv_fetch_array.

See Also

SQLSRV Driver API Reference
Retrieving Data
About Code Examples in the Documentation
Programming Guide for PHP SQL Driver

sqlsrv_fetch_object

3/14/2017 • 7 min to read • Edit on GitHub

Download PHP Driver

Retrieves the next row of data as a PHP object.

Syntax

```
sqlsrv_fetch_object( resource $stmt [, string $className [, array $ctorParams[, row[, ]offset]]])
```

Parameters

$stmt: A statement resource corresponding to an executed statement.

$className [OPTIONAL]: A string specifying the name of the class to instantiate. If a value for the *$className* parameter is not specified, an instance of the PHP **stdClass** is instantiated.

$ctorParams [OPTIONAL]: An array that contains values passed to the constructor of the class specified with the *$className* parameter. If the constructor of the specified class accepts parameter values, the *$ctorParams* parameter must be used when calling **sqlsrv_fetch_object**.

row [OPTIONAL]: One of the following values, specifying the row to access in a result set that uses a scrollable cursor. (If *row* is specified, *$className* and *$ctorParams* must be explicitly specified, even if you must specify null for *$className* and *$ctorParams*.)

- SQLSRV_SCROLL_NEXT
- SQLSRV_SCROLL_PRIOR
- SQLSRV_SCROLL_FIRST
- SQLSRV_SCROLL_LAST
- SQLSRV_SCROLL_ABSOLUTE
- SQLSRV_SCROLL_RELATIVE

For more information about these values, see Specifying a Cursor Type and Selecting Rows.

offset [OPTIONAL]: Used with SQLSRV_SCROLL_ABSOLUTE and SQLSRV_SCROLL_RELATIVE to specify the row to retrieve. The first record in the result set is 0.

Return Value

A PHP object with properties that correspond to result set field names. Property values are populated with the corresponding result set field values. If the class specified with the optional *$className* parameter does not exist or if there is no active result set associated with the specified statement, **false** is returned. If there are no more rows to retrieve, **null** is returned.

The data type of a value in the returned object will be the default PHP data type. For information on default PHP data types, see Default PHP Data Types.

Remarks

If a class name is specified with the optional *$className* parameter, an object of this class type is instantiated. If the class has properties whose names match the result set field names, the corresponding result set values are applied to the properties. If a result set field name does not match a class property, a property with the result set field name is added to the object and the result set value is applied to the property.

The following rules apply when specifying a class with the *$className* parameter:

- Matching is case-sensitive. For example, the property name CustomerId does not match the field name CustomerID. In this case, a CustomerID property would be added to the object and the value of the CustomerID field would be given to the CustomerID property.
- Matching occurs regardless of access modifiers. For example, if the specified class has a private property whose name matches a result set field name, the value from the result set field is applied to the property.
- Class property data types are ignored. If the "CustomerID" field in the result set is a string but the "CustomerID" property of the class is an integer, the string value from the result set is written to the "CustomerID" property.
- If the specified class does not exist, the function returns **false** and adds an error to the error collection. For information about retrieving error information, see sqlsrv_errors.

If a field with no name is returned, **sqlsrv_fetch_object** will discard the field value and issue a warning. For example, consider this Transact-SQL statement that inserts a value into a database table and retrieves the server-generated primary key:

```
INSERT INTO Production.ProductPhoto (LargePhoto) VALUES (?);
SELECT SCOPE_IDENTITY()
```

If the results returned by this query are retrieved with **sqlsrv_fetch_object**, the value returned by `SELECT SCOPE_IDENTITY()` will be discarded and a warning will be issued. To avoid this, you can specify a name for the returned field in the Transact-SQL statement. The following is one way to specify a column name in Transact-SQL:

`SELECT SCOPE_IDENTITY() AS PictureID`

Example

The following example retrieves each row of a result set as a PHP object. The example assumes that the SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up and execute the query. */
$tsql = "SELECT FirstName, LastName
         FROM Person.Contact
         WHERE LastName='Alan'";
$stmt = sqlsrv_query( $conn, $tsql);
if( $stmt === false )
{
     echo "Error in query preparation/execution.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve each row as a PHP object and display the results.*/
while( $obj = sqlsrv_fetch_object( $stmt))
{
      echo $obj->LastName.", ".$obj->FirstName."\n";
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

Example

The following example retrieves each row of a result set as an instance of the *Product* class defined in the script. The example retrieves product information from the *Purchasing.PurchaseOrderDetail* and *Production.Product* tables of the AdventureWorks database for products that have a specified due date (*DueDate*), and a stocked quantity (*StockQty*) less than a specified value. The example highlights some of the rules that apply when specifying a class in a call to **sqlsrv_fetch_object**:

- The *$product* variable is an instance of the *Product* class, because "Product" was specified with the *$className* parameter and the *Product* class exists.
- The *Name* property is added to the *$product* instance because the existing *name* property does not match.
- The *Color* property is added to the *$product* instance because there is no matching property.
- The private property *UnitPrice* is populated with the value of the *UnitPrice* field.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Define the Product class. */
class Product
{
     /* Constructor */
     public function Product($ID)
     {
          $this->objID = $ID;
     }
     }
     public $objID;
     public $name;
     public $StockedQty;
     public $SafetyStockLevel;
     private $UnitPrice;
     function getPrice()
     {
          return $this->UnitPrice;
     }
}

/* Connect to the local server using Windows Authentication, and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Define the query. */
$tsql = "SELECT Name,
                SafetyStockLevel,
                StockedQty,
                UnitPrice,
                Color
         FROM Purchasing.PurchaseOrderDetail AS pdo
         JOIN Production.Product AS p
         ON pdo.ProductID = p.ProductID
         WHERE pdo.StockedQty < ?
         AND pdo.DueDate= ?";

/* Set the parameter values. */
$params = array(3, '2002-01-29');

/* Execute the query. */
$stmt = sqlsrv_query( $conn, $tsql, $params);
if ( $stmt )
{
     echo "Statement executed.\n";
}
else
{
     echo "Error in statement execution.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Iterate through the result set, printing a row of data upon each
 iteration. Note the following:
     1) $product is an instance of the Product class.
     2) The $ctorParams parameter is required in the call to
        sqlsrv_fetch_object, because the Product class constructor is
        explicity defined and requires parameter values.
     3) The "Name" property is added to the $product instance because
        the existing "name" property does not match.
     4) The "Color" property is added to the $product instance
        because there is no matching property.
     5) The private property "UnitPrice" is populated with the value
        of the "UnitPrice" field.*/
$i=0; //Used as the $objID in the Product class constructor.
while( $product = sqlsrv_fetch_object( $stmt, "Product", array($i)))
{
     echo "Object ID: ".$product->objID."\n";
     echo "Product Name: ".$product->Name."\n";
     echo "Stocked Qty: ".$product->StockedQty."\n";
     echo "Safety Stock Level: ".$product->SafetyStockLevel."\n";
     echo "Product Color: ".$product->Color."\n";
```

```
      echo "Product Color: ".$product->Color."\n";
      echo "Unit Price: ".$product->getPrice()."\n";
      echo "-----------------\n";
      $i++;
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

The **sqlsrv_fetch_object** function always returns data according to the Default PHP Data Types. For information about how to specify the PHP data type, see How to: Specify PHP Data Types.

If a field with no name is returned, **sqlsrv_fetch_object** will discard the field value and issue a warning. For example, consider this Transact-SQL statement that inserts a value into a database table and retrieves the server-generated primary key:

```
INSERT INTO Production.ProductPhoto (LargePhoto) VALUES (?);
SELECT SCOPE_IDENTITY()
```

If the results returned by this query are retrieved with **sqlsrv_fetch_object**, the value returned by `SELECT SCOPE_IDENTITY()` will be discarded and a warning will be issued. To avoid this, you can specify a name for the returned field in the Transact-SQL statement. The following is one way to specify a column name in Transact-SQL:

`SELECT SCOPE_IDENTITY() AS PictureID`

See Also

Retrieving Data
About Code Examples in the Documentation
SQLSRV Driver API Reference

sqlsrv_field_metadata

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

Retrieves metadata for the fields of a prepared statement. For information about preparing a statement, see sqlsrv_query or sqlsrv_prepare. Note that **sqlsrv_field_metadata** can be called on any prepared statement, pre- or post-execution.

Syntax

```
sqlsrv_field_metadata( resource $stmt)
```

Parameters

$stmt: A statement resource for which field metadata is sought.

Return Value

An **array** of arrays or **false**. The array consists of one array for each field in the result set. Each sub-array has keys as described in the table below. If there is an error in retrieving field metadata, **false** is returned.

KEY	DESCRIPTION
Name	Name of the column to which the field corresponds.
Type	Numeric value that corresponds to a SQL type.
Size	Number of characters for fields of character type (char(n), varchar(n), nchar(n), nvarchar(n), XML). Number of bytes for fields of binary type (binary(n), varbinary(n), UDT). **NULL** for other SQL Server data types.
Precision	The precision for types of variable precision (real, numeric, decimal, datetime2, datetimeoffset, and time). **NULL** for other SQL Server data types.
Scale	The scale for types of variable scale (numeric, decimal, datetime2, datetimeoffset, and time). **NULL** for other SQL Server data types.
Nullable	An enumerated value indicating whether the column is nullable (**SQLSRV_NULLABLE_YES**), the column is not nullable (**SQLSRV_NULLABLE_NO**), or it is not known if the column is nullable (**SQLSRV_NULLABLE_UNKNOWN**).

The following table gives more information on the keys for each sub-array (see the SQL Server documentation for more information on these types):

SQL SERVER 2008 DATA TYPE	TYPE	MIN/MAX PRECISION	MIN/MAX SCALE	SIZE
bigint	SQL_BIGINT (-5)			8
binary	SQL_BINARY (-2)			0 < *n* < 8000 [1]
bit	SQL_BIT (-7)			
char	SQL_CHAR (1)			0 < *n* < 8000 [1]
date	SQL_TYPE_DATE (91)	10/10	0/0	
datetime	SQL_TYPE_TIMESTAMP (93)	23/23	3/3	
datetime2	SQL_TYPE_TIMESTAMP (93)	19/27	0/7	
datetimeoffset	SQL_SS_TIMESTAMPOFFSET (-155)	26/34	0/7	
decimal	SQL_DECIMAL (3)	1/38	0/precision value	
float	SQL_FLOAT (6)	4/8		
image	SQL_LONGVARBINARY (-4)			2 GB
int	SQL_INTEGER (4)			
money	SQL_DECIMAL (3)	19/19	4/4	
nchar	SQL_WCHAR (-8)			0 < *n* < 4000 [1]
ntext	SQL_WLONGVARCHAR (-10)			1 GB
numeric	SQL_NUMERIC (2)	1/38	0/precision value	
nvarchar	SQL_WVARCHAR (-9)			0 < *n* < 4000 [1]
real	SQL_REAL (7)	4/4		
smalldatetime	SQL_TYPE_TIMESTAMP (93)	16/16	0/0	
smallint	SQL_SMALLINT (5)			2 bytes
Smallmoney	SQL_DECIMAL (3)	10/10	4/4	
text	SQL_LONGVARCHAR (-1)			2 GB

SQL SERVER 2008 DATA TYPE	TYPE	MIN/MAX PRECISION	MIN/MAX SCALE	SIZE
time	SQL_SS_TIME2 (-154)	8/16	0/7	
timestamp	SQL_BINARY (-2)			8 bytes
tinyint	SQL_TINYINT (-6)			1 byte
udt	SQL_SS_UDT (-151)			variable
uniqueidentifier	SQL_GUID (-11)			16
varbinary	SQL_VARBINARY (-3)			$0 < n < 8000$ [1]
varchar	SQL_VARCHAR (12)			$0 < n < 8000$ [1]
xml	SQL_SS_XML (-152)			0

(1) Zero (0) indicates that the maximum size is allowed.

The Nullable key can either be yes or no.

Example

The following example creates a statement resource, then retrieves and displays the field metadata. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Prepare the statement. */
$tsql = "SELECT ReviewerName, Comments FROM Production.ProductReview";
$stmt = sqlsrv_prepare( $conn, $tsql);

/* Get and display field metadata. */
foreach( sqlsrv_field_metadata( $stmt) as $fieldMetadata)
{
      foreach( $fieldMetadata as $name => $value)
      {
           echo "$name: $value\n";
      }
      echo "\n";
}

/* Note: sqlsrv_field_metadata can be called on any statement
resource, pre- or post-execution. */

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference
Constants (Microsoft Drivers for PHP for SQL Server)
About Code Examples in the Documentation

sqlsrv_free_stmt

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Frees all resources associated with the specified statement. The statement cannot be used again after this function has been called.

Syntax

```
sqlsrv_free_stmt( resource $stmt)
```

Parameters

$stmt: The statement to be closed.

Return Value

The Boolean value **true** unless the function is called with an invalid parameter. If the function is called with an invalid parameter, **false** is returned.

NOTE

Null is a valid parameter for this function. This allows the function to be called multiple times in a script. For example, if you free a statement in an error condition and free it again at the end of the script, the second call to **sqlsrv_free_stmt** will return **true** because the first call to **sqlsrv_free_stmt** (in the error condition) sets the statement resource to **null**.

Example

The following example creates a statement resource, executes a simple query, and calls **sqlsrv_free_stmt** to free all resources associated with the statement. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

$stmt = sqlsrv_query( $conn, "SELECT * FROM Person.Contact");
if( $stmt )
{
     echo "Statement executed.\n";
}
else
{
     echo "Query could not be executed.\n";
     die( print_r( sqlsrv_errors(), true));
}

/*-------------------------------
     Process query results here.
-------------------------------*/

/* Free the statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference

About Code Examples in the Documentation

sqlsrv_cancel

sqlsrv_get_config

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Returns the current value of the specified configuration setting.

Syntax

```
sqlsrv_get_config( string $setting )
```

Parameters

$setting: The configuration setting for which the value is returned. For a list of configurable settings, see sqlsrv_configure.

Return Value

The value of the setting specified by the *$setting* parameter. If an invalid setting is specified, **false** is returned and an error is added to the error collection.

Remarks

If **false** is returned by **sqlsrv_get_config**, you must call sqlsrv_errors to determine if an error occurred or if **false** is the value of the setting specified by the *$setting* parameter.

See Also

SQLSRV Driver API Reference
sqlsrv_configure
sqlsrv_errors

sqlsrv_get_field

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

Retrieves data from the specified field of the current row. Field data must be accessed in order. For example, data from the first field cannot be accessed after data from the second field has been accessed.

Syntax

```
sqlsrv_get_field( resource $stmt, int $fieldIndex [, int $getAsType])
```

Parameters

$stmt: A statement resource corresponding to an executed statement.

$fieldIndex: The index of the field to be retrieved. Indexes begin at zero.

$getAsType [OPTIONAL]: A **SQLSRV** constant (**SQLSRV_PHPTYPE_***) that determines the PHP data type for the returned data. For information about supported data types, see Constants (Microsoft Drivers for PHP for SQL Server). If no return type is specified, a default PHP type will be returned. For information about default PHP types, see Default PHP Data Types. For information about specifying PHP data types, see How to: Specify PHP Data Types.

Return Value

The field data. You can specify the PHP data type of the returned data by using the *$getAsType* parameter. If no return data type is specified, the default PHP data type will be returned. For information about default PHP types, see Default PHP Data Types. For information about specifying PHP data types, see How to: Specify PHP Data Types.

Remarks

The combination of **sqlsrv_fetch** and **sqlsrv_get_field** provides forward-only access to data.

The combination of **sqlsrv_fetch**/**sqlsrv_get_field** loads only one field of a result set row into script memory and allows PHP return type specification. (For information about how to specify the PHP return type, see How to: Specify PHP Data Types.) This combination of functions also allows data to be retrieved as a stream. (For information about retrieving data as a stream, see Retrieving Data as a Stream Using the SQLSRV Driver.)

Example

The following example retrieves a row of data that contains a product review and the name of the reviewer. To retrieve data from the result set, **sqlsrv_get_field** is used. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/*Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up and execute the query. Note that both ReviewerName and
Comments are of the SQL Server nvarchar type. */
$tsql = "SELECT ReviewerName, Comments
         FROM Production.ProductReview
         WHERE ProductReviewID=1";
$stmt = sqlsrv_query( $conn, $tsql);
if( $stmt === false )
{
     echo "Error in statement preparation/execution.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Make the first row of the result set available for reading. */
if( sqlsrv_fetch( $stmt ) === false )
{
     echo "Error in retrieving row.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Note: Fields must be accessed in order.
Get the first field of the row. Note that no return type is
specified. Data will be returned as a string, the default for
a field of type nvarchar.*/
$name = sqlsrv_get_field( $stmt, 0);
echo "$name: ";

/*Get the second field of the row as a stream.
Because the default return type for a nvarchar field is a
string, the return type must be specified as a stream. */
$stream = sqlsrv_get_field( $stmt, 1,
                            SQLSRV_PHPTYPE_STREAM( SQLSRV_ENC_CHAR));
while( !feof( $stream))
{
    $str = fread( $stream, 10000);
    echo $str;
}

/* Free the statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference
Retrieving Data
About Code Examples in the Documentation

sqlsrv_has_rows

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Indicates if the result set has one or more rows.

Syntax

```
sqlsrv_has_rows( resource $stmt )
```

Parameters

$stmt: The executed statement.

Return Value

If there are rows in the result set, the return value will be **true**. If there are no rows, or if the function call fails, the return value will be **false**.

Example

```
<?php
   $server = "server_name";
   $conn = sqlsrv_connect( $server, array( 'Database' => 'Northwind' ) );

   $stmt = sqlsrv_query( $conn, "select * from orders where CustomerID = 'VINET'" , array());

   if ($stmt !== NULL) {
      $rows = sqlsrv_has_rows( $stmt );

      if ($rows === true)
         echo "\nthere are rows\n";
      else
         echo "\nno rows\n";
   }
?>
```

See Also

SQLSRV Driver API Reference

sqlsrv_next_result

3/14/2017 • 5 min to read • Edit on GitHub

Download PHP Driver

Makes the next result (result set, row count, or output parameter) of the specified statement active.

NOTE

The first (or only) result returned by a batch query or stored procedure is active without a call to **sqlsrv_next_result**.

Syntax

```
sqlsrv_next_result( resource $stmt )
```

Parameters

$stmt: The executed statement on which the next result is made active.

Return Value

If the next result was successfully made active, the Boolean value **true** is returned. If an error occurred in making the next result active, **false** is returned. If no more results are available, **null** is returned.

Example

The following example creates and executes a stored procedure that inserts a product review into the *Production.ProductReview* table, and then selects all reviews for the specified product. After execution of the stored procedure, the first result (the number of rows affected by the INSERT query in the stored procedure) is consumed without calling **sqlsrv_next_result**. The next result (the rows returned by the SELECT query in the stored procedure) is made available by calling **sqlsrv_next_result** and consumed using sqlsrv_fetch_array.

NOTE

Calling stored procedures using canonical syntax is the recommended practice. For more information about canonical syntax, see Calling a Stored Procedure.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}
```

```
/* Drop the stored procedure if it already exists. */
$tsql_dropSP = "IF OBJECT_ID('InsertProductReview', 'P') IS NOT NULL
                DROP PROCEDURE InsertProductReview";
$stmt1 = sqlsrv_query( $conn, $tsql_dropSP);
if( $stmt1 === false )
{
     echo "Error in executing statement 1.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Create the stored procedure. */
$tsql_createSP = " CREATE PROCEDURE InsertProductReview
                                    @ProductID int,
                                    @ReviewerName nvarchar(50),
                                    @ReviewDate datetime,
                                    @EmailAddress nvarchar(50),
                                    @Rating int,
                                    @Comments nvarchar(3850)
                   AS
                       BEGIN
                             INSERT INTO Production.ProductReview
                                         (ProductID,
                                          ReviewerName,
                                          ReviewDate,
                                          EmailAddress,
                                          Rating,
                                          Comments)
                                    VALUES
                                         (@ProductID,
                                          @ReviewerName,
                                          @ReviewDate,
                                          @EmailAddress,
                                          @Rating,
                                          @Comments);
                             SELECT * FROM Production.ProductReview
                                WHERE ProductID = @ProductID;
                       END";
$stmt2 = sqlsrv_query( $conn, $tsql_createSP);

if( $stmt2 === false)
{
     echo "Error in executing statement 2.\n";
     die( print_r( sqlsrv_errors(), true));
}
/*-------- The next few steps call the stored procedure. --------*/

/* Define the Transact-SQL query. Use question marks (?) in place of the
parameters to be passed to the stored procedure */
$tsql_callSP = "{call InsertProductReview(?, ?, ?, ?, ?, ?)}";

/* Define the parameter array. */
$productID = 709;
$reviewerName = "Customer Name";
$reviewDate = "2008-02-12";
$emailAddress = "customer@email.com";
$rating = 3;
$comments = "[Insert comments here.]";
$params = array(
                 $productID,
                 $reviewerName,
                 $reviewDate,
                 $emailAddress,
                 $rating,
                 $comments
               );

/* Execute the query. */
$stmt3 = sqlsrv_query( $conn, $tsql_callSP, $params);
```

```
if( $stmt3 === false)
{
     echo "Error in executing statement 3.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Consume the first result (rows affected by INSERT query in the
stored procedure) without calling sqlsrv_next_result. */
echo "Rows affectd: ".sqlsrv_rows_affected($stmt3)."-----\n";

/* Move to the next result and display results. */
$next_result = sqlsrv_next_result($stmt3);
if( $next_result )
{
     echo "\nReview information for product ID ".$productID.".---\n";
     while( $row = sqlsrv_fetch_array( $stmt3, SQLSRV_FETCH_ASSOC))
     {
          echo "ReviewerName: ".$row['ReviewerName']."\n";
          echo "ReviewDate: ".date_format($row['ReviewDate'],
                                       "M j, Y")."\n";
          echo "EmailAddress: ".$row['EmailAddress']."\n";
          echo "Rating: ".$row['Rating']."\n\n";
     }
}
elseif( is_null($next_result))
{
     echo "No more results.\n";
}
else
{
     echo "Error in moving to next result.\n";
     die(print_r(sqlsrv_errors(), true));
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt1 );
sqlsrv_free_stmt( $stmt2 );
sqlsrv_free_stmt( $stmt3 );
sqlsrv_close( $conn );
?>
```

When executing a stored procedure that has output parameters, it is recommended that all other results are consumed before accessing the values of output parameters. For more information see How to: Specify Parameter Direction Using the SQLSRV Driver.

Example

The following example executes a batch query that retrieves product review information for a specified product ID, inserts a review for the product, then again retrieves the product review information for the specified product ID. The newly inserted product review will be included in the final result set of the batch query. The example uses sqlsrv_next_result to move from one result of the batch query to the next.

NOTE

The first (or only) result returned by a batch query or stored procedure is active without a call to **sqlsrv_next_result**.

The example uses the *Purchasing.ProductReview* table of the AdventureWorks database, and assumes that this database is installed on the server. All output is written to the console when the example is run from the command line.

```
<?php
```

```
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Define the batch query. */
$tsql = "--Query 1
         SELECT ProductID, ReviewerName, Rating
         FROM Production.ProductReview
         WHERE ProductID=?;

         --Query 2
         INSERT INTO Production.ProductReview (ProductID,
                                               ReviewerName,
                                               ReviewDate,
                                               EmailAddress,
                                               Rating)
         VALUES (?, ?, ?, ?, ?);

         --Query 3
         SELECT ProductID, ReviewerName, Rating
         FROM Production.ProductReview
         WHERE ProductID=?;";

/* Assign parameter values and execute the query. */
$params = array(798,
                798,
                'CustomerName',
                '2008-4-15',
                'test@customer.com',
                3,
                798 );
$stmt = sqlsrv_query($conn, $tsql, $params);
if( $stmt === false )
{
     echo "Error in statement execution.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve and display the first result. */
echo "Query 1 result:\n";
while( $row = sqlsrv_fetch_array( $stmt, SQLSRV_FETCH_NUMERIC ))
{
     print_r($row);
}

/* Move to the next result of the batch query. */
sqlsrv_next_result($stmt);

/* Display the result of the second query. */
echo "Query 2 result:\n";
echo "Rows Affected: ".sqlsrv_rows_affected($stmt)."\n";

/* Move to the next result of the batch query. */
sqlsrv_next_result($stmt);

/* Retrieve and display the third result. */
echo "Query 3 result:\n";
while( $row = sqlsrv_fetch_array( $stmt, SQLSRV_FETCH_NUMERIC ))
{
     print_r($row);
}
```

```
/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt );
sqlsrv_close( $conn );
?>
```

See Also

SQLSRV Driver API Reference
About Code Examples in the Documentation
Retrieving Data
Updating Data (Microsoft Drivers for PHP for SQL Server)
Example Application (SQLSRV Driver)

sqlsrv_num_fields

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Retrieves the number of fields in an active result set. Note that **sqlsrv_num_fields** can be called on any prepared statement, before or after execution.

Syntax

```
sqlsrv_num_fields( resource $stmt)
```

Parameters

$stmt: The statement on which the targeted result set is active.

Return Value

An integer value that represents the number of fields in the active result set. If an error occurs, the Boolean value **false** is returned.

Example

The following example executes a query to retrieve all fields for the top three rows in the *HumanResources.Department* table of the Adventureworks database. The **sqlsrv_num_fields** function determines the number of fields in the result set. This allows data to be displayed by iterating through the fields in each returned row.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Define and execute the query. */
$tsql = "SELECT TOP (3) * FROM HumanResources.Department";
$stmt = sqlsrv_query($conn, $tsql);
if( $stmt === false)
{
     echo "Error in executing query.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve the number of fields. */
$numFields = sqlsrv_num_fields( $stmt );

/* Iterate through each row of the result set. */
while( sqlsrv_fetch( $stmt ))
{
     /* Iterate through the fields of each row. */
     for($i = 0; $i < $numFields; $i++)
     {
          echo sqlsrv_get_field($stmt, $i,
                   SQLSRV_PHPTYPE_STRING(SQLSRV_ENC_CHAR))." ";
     }
     echo "\n";
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt );
sqlsrv_close( $conn );
?>
```

See Also

SQLSRV Driver API Reference
sqlsrv_field_metadata
About Code Examples in the Documentation

sqlsrv_num_rows

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Reports the number of rows in a result set.

Syntax

```
sqlsrv_num_rows( resource $stmt )
```

Parameters

$stmt: The result set for which to count the rows.

Return Value

false if there was an error calculating the number of rows. Otherwise, returns the number of rows in the result set.

Remarks

sqlsrv_num_rows requires a client-side, static, or keyset cursor, and will return **false** if you use a forward cursor or a dynamic cursor. (A forward cursor is the default.) For more information about cursors, see sqlsrv_query and Cursor Types (SQLSRV Driver).

Example

```
<?php
   $server = "server_name";
   $conn = sqlsrv_connect( $server, array( 'Database' => 'Northwind' ) );

   $stmt = sqlsrv_query( $conn, "select * from orders where CustomerID = 'VINET'" , array(), array(
"Scrollable" => SQLSRV_CURSOR_KEYSET ));

   $row_count = sqlsrv_num_rows( $stmt );

   if ($row_count === false)
      echo "\nerror\n";
   else if ($row_count >=0)
      echo "\n$row_count\n";
?>
```

The following sample shows that when there is more than one result set (a batch query), the number of rows is only available when you use a client-side cursor.

```
<?php
$serverName = "(local)";
$connectionInfo = array("Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);

$tsql = "select * from HumanResources.Department";

// Client-side cursor and batch statements
$tsql = "select top 2 * from HumanResources.Employee;Select top 3 * from HumanResources.EmployeeAddress";

// works
$stmt = sqlsrv_query($conn, $tsql, array(), array("Scrollable"=>"buffered"));

// fails
// $stmt = sqlsrv_query($conn, $tsql);
// $stmt = sqlsrv_query($conn, $tsql, array(), array("Scrollable"=>"forward"));
// $stmt = sqlsrv_query($conn, $tsql, array(), array("Scrollable"=>"static"));
// $stmt = sqlsrv_query($conn, $tsql, array(), array("Scrollable"=>"keyset"));
// $stmt = sqlsrv_query($conn, $tsql, array(), array("Scrollable"=>"dynamic"));

$row_count = sqlsrv_num_rows( $stmt );
echo "\nRow count for first result set = $row_count\n";

sqlsrv_next_result($stmt);

$row_count = sqlsrv_num_rows( $stmt );
echo "\nRow count for second result set = $row_count\n";
?>
```

See Also

SQLSRV Driver API Reference

sqlsrv_prepare

3/14/2017 • 5 min to read • Edit on GitHub

Download PHP Driver

Creates a statement resource associated with the specified connection. This function is useful for execution of multiple queries.

Syntax

```
sqlsrv_prepare( resource $conn, string $tsql [, array $params [, array $options]])
```

Parameters

$conn: The connection resource associated with the created statement.

$tsql: The Transact-SQL expression that corresponds to the created statement.

$params [OPTIONAL]: An **array** of values that correspond to parameters in a parameterized query. Each element of the array can be one of the following:

- A literal value.
- A reference to a PHP variable.
- An **array** with the following structure:

```
array(&$value [, $direction [, $phpType [, $sqlType]]])
```

NOTE

Variables passed as query parameters should be passed by reference instead of by value. For example, pass `&$myVariable` instead of `$myVariable`. A PHP warning will be raised when a query with by-value parameters is executed.

The following table describes these array elements:

ELEMENT	DESCRIPTION
&$value	A literal value or a reference to a PHP variable.
$direction[OPTIONAL]	One of the following **SQLSRV_PARAM_*** constants used to indicate the parameter direction: **SQLSRV_PARAM_IN**, **SQLSRV_PARAM_OUT**, **SQLSRV_PARAM_INOUT**. The default value is **SQLSRV_PARAM_IN**. For more information about PHP constants, see Constants (Microsoft Drivers for PHP for SQL Server).

ELEMENT	DESCRIPTION
$phpType[OPTIONAL]	A **SQLSRV_PHPTYPE_*** constant that specifies PHP data type of the returned value.
$sqlType[OPTIONAL]	A **SQLSRV_SQLTYPE_*** constant that specifies the SQL Server data type of the input value.

$options [OPTIONAL]: An associative array that sets query properties. The table below lists the supported keys and corresponding values:

KEY	SUPPORTED VALUES	DESCRIPTION
QueryTimeout	A positive integer value.	Sets the query timeout in seconds. By default, the driver will wait indefinitely for results.
SendStreamParamsAtExec	**true** or **false** The default value is **true**.	Configures the driver to send all stream data at execution (**true**), or to send stream data in chunks (**false**). By default, the value is set to **true**. For more information, see sqlsrv_send_stream_data.
Scrollable	SQLSRV_CURSOR_FORWARD SQLSRV_CURSOR_STATIC SQLSRV_CURSOR_DYNAMIC SQLSRV_CURSOR_KEYSET SQLSRV_CURSOR_CLIENT_BUFFERED	For more information about these values, see Specifying a Cursor Type and Selecting Rows.

Return Value

A statement resource. If the statement resource cannot be created, **false** is returned.

Remarks

When you prepare a statement that uses variables as parameters, the variables are bound to the statement. That means that if you update the values of the variables, the next time you execute the statement it will run with updated parameter values.

The combination of **sqlsrv_prepare** and **sqlsrv_execute** separates statement preparation and statement execution in to two function calls and can be used to execute parameterized queries. This function is ideal to execute a statement multiple times with different parameter values for each execution.

For alternative strategies for writing and reading large amounts of information, see Batches of SQL Statements and BULK INSERT.

For more information, see How to: Retrieve Output Parameters Using the SQLSRV Driver.

Example

The following example prepares and executes a statement. The statement, when executed (see sqlsrv_execute), updates a field in the *Sales.SalesOrderDetail* table of the AdventureWorks database. The example assumes that

SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up Transact-SQL query. */
$tsql = "UPDATE Sales.SalesOrderDetail
         SET OrderQty = ?
         WHERE SalesOrderDetailID = ?";

/* Assign parameter values. */
$param1 = 5;
$param2 = 10;
$params = array( &$param1, &$param2);

/* Prepare the statement. */
if( $stmt = sqlsrv_prepare( $conn, $tsql, $params))
{
      echo "Statement prepared.\n";
}
else
{
      echo "Statement could not be prepared.\n";
      die( print_r( sqlsrv_errors(), true));
}

/* Execute the statement. */
if( sqlsrv_execute( $stmt))
{
      echo "Statement executed.\n";
}
else
{
      echo "Statement could not be executed.\n";
      die( print_r( sqlsrv_errors(), true));
}

/* Free the statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

Example

The following example demonstrates how to prepare a statement and then re-execute it with different parameter values. The example updates the *OrderQty* column of the *Sales.SalesOrderDetail* table in the AdventureWorks database. After the updates have occurred, the database is queried to verify that the updates were successful. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
```

```
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Define the parameterized query. */
$tsql = "UPDATE Sales.SalesOrderDetail
         SET OrderQty = ?
         WHERE SalesOrderDetailID = ?";

/* Initialize parameters and prepare the statement. Variables $qty
and $id are bound to the statement, $stmt1. */
$qty = 0; $id = 0;
$stmt1 = sqlsrv_prepare( $conn, $tsql, array( &$qty, &$id));
if( $stmt1 )
{
     echo "Statement 1 prepared.\n";
}
else
{
     echo "Error in statement preparation.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up the SalesOrderDetailID and OrderQty information. This array
maps the order ID to order quantity in key=>value pairs. */
$orders = array( 1=>10, 2=>20, 3=>30);

/* Execute the statement for each order. */
foreach( $orders as $id => $qty)
{
     // Because $id and $qty are bound to $stmt1, their updated
     // values are used with each execution of the statement.
     if( sqlsrv_execute( $stmt1) === false )
     {
          echo "Error in statement execution.\n";
          die( print_r( sqlsrv_errors(), true));
     }
}
echo "Orders updated.\n";

/* Free $stmt1 resources.  This allows $id and $qty to be bound to a different statement.*/
sqlsrv_free_stmt( $stmt1);

/* Now verify that the results were successfully written by selecting
the newly inserted rows. */
$tsql = "SELECT OrderQty
         FROM Sales.SalesOrderDetail
         WHERE SalesOrderDetailID = ?";

/* Prepare the statement. Variable $id is bound to $stmt2. */
$stmt2 = sqlsrv_prepare( $conn, $tsql, array( &$id));
if( $stmt2 )
{
     echo "Statement 2 prepared.\n";
}
else
{
     echo "Error in statement preparation.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Execute the statement for each order. */
foreach( array_keys($orders) as $id)
{
     /* Because $id is bound to $stmt2, its updated value
        is used with each execution of the statement. */
     if( sqlsrv_execute( $stmt2))
     {
          sqlsrv_fetch( $stmt2);
          $quantity = sqlsrv_get_field( $stmt2, 0);
          echo "Order $id is for $quantity units.\n";
     }
     else
     {
          echo "Error in statement execution.\n";
          die( print_r( sqlsrv_errors(), true));
     }
}

/* Free $stmt2 and connection resources. */
sqlsrv_free_stmt( $stmt2);
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference
How to: Perform Parameterized Queries
About Code Examples in the Documentation
How to: Send Data as a Stream
Using Directional Parameters
Retrieving Data
Updating Data (Microsoft Drivers for PHP for SQL Server)

sqlsrv_query

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

Prepares and executes a statement.

Syntax

```
sqlsrv_query( resource $conn, string $tsql [, array $params [, array $options]])
```

Parameters

$conn: The connection resource associated with the prepared statement.

$tsql: The Transact-SQL expression that corresponds to the prepared statement.

$params [OPTIONAL]: An **array** of values that correspond to parameters in a parameterized query. Each element of the array can be one of the following:

- A literal value.
- A PHP variable.
- An **array** with the following structure:

  ```
  array($value [, $direction [, $phpType [, $sqlType]]])
  ```

  The description for each element of the array is in the table below:

ELEMENT	DESCRIPTION
$value	A literal value, a PHP variable, or a PHP by-reference variable.
$direction[OPTIONAL]	One of the following **SQLSRV_PARAM_*** constants used to indicate the parameter direction: **SQLSRV_PARAM_IN**, **SQLSRV_PARAM_OUT**, **SQLSRV_PARAM_INOUT**. The default value is **SQLSRV_PARAM_IN**. For more information about PHP constants, see Constants (Microsoft Drivers for PHP for SQL Server).
$phpType[OPTIONAL]	A **SQLSRV_PHPTYPE_*** constant that specifies PHP data type of the returned value. For more information about PHP constants, see Constants (Microsoft Drivers for PHP for SQL Server).

ELEMENT	DESCRIPTION
$sqlType[OPTIONAL]	A **SQLSRV_SQLTYPE_*** constant that specifies the SQL Server data type of the input value. For more information about PHP constants, see Constants (Microsoft Drivers for PHP for SQL Server).

$options [OPTIONAL]: An associative array that sets query properties. The supported keys are as follows:

KEY	SUPPORTED VALUES	DESCRIPTION
QueryTimeout	A positive integer value.	Sets the query timeout in seconds. By default, the driver will wait indefinitely for results.
SendStreamParamsAtExec	**true** or **false** The default value is **true**.	Configures the driver to send all stream data at execution (**true**), or to send stream data in chunks (**false**). By default, the value is set to **true**. For more information, see sqlsrv_send_stream_data.
Scrollable	SQLSRV_CURSOR_FORWARD SQLSRV_CURSOR_STATIC SQLSRV_CURSOR_DYNAMIC SQLSRV_CURSOR_KEYSET SQLSRV_CURSOR_CLIENT_BUFFERED	For more information about these values, see Specifying a Cursor Type and Selecting Rows.

Return Value

A statement resource. If the statement cannot be created and/or executed, **false** is returned.

Remarks

The **sqlsrv_query** function is well-suited for one-time queries and should be the default choice to execute queries unless special circumstances apply. This function provides a streamlined method to execute a query with a minimum amount of code. The **sqlsrv_query** function does both statement preparation and statement execution, and can be used to execute parameterized queries.

For more information, see How to: Retrieve Output Parameters Using the SQLSRV Driver.

Example

In the following example, a single row is inserted into the *Sales.SalesOrderDetail* table of the AdventureWorks database. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

NOTE

Although the following example uses an INSERT statement to demonstrate the use of **sqlsrv_query** for a one-time statement execution, the concept applies to any Transact-SQL statement.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up the parameterized query. */
$tsql = "INSERT INTO Sales.SalesOrderDetail
        (SalesOrderID,
         OrderQty,
         ProductID,
         SpecialOfferID,
         UnitPrice,
         UnitPriceDiscount)
        VALUES
        (?, ?, ?, ?, ?, ?)";

/* Set parameter values. */
$params = array(75123, 5, 741, 1, 818.70, 0.00);

/* Prepare and execute the query. */
$stmt = sqlsrv_query( $conn, $tsql, $params);
if( $stmt )
{
     echo "Row successfully inserted.\n";
}
else
{
     echo "Row insertion failed.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

Example

The example below updates a field in the *Sales.SalesOrderDetail* table of the AdventureWorks database. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array("Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up the parameterized query. */
$tsql = "UPDATE Sales.SalesOrderDetail
         SET OrderQty = ( ?)
         WHERE SalesOrderDetailID = ( ?)";

/* Assign literal parameter values. */
$params = array( 5, 10);

/* Execute the query. */
if( sqlsrv_query( $conn, $tsql, $params))
{
      echo "Statement executed.\n";
}
else
{
      echo "Error in statement execution.\n";
      die( print_r( sqlsrv_errors(), true));
}

/* Free connection resources. */
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference
How to: Perform Parameterized Queries
About Code Examples in the Documentation
How to: Send Data as a Stream
Using Directional Parameters

sqlsrv_rollback

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

Rolls back the current transaction on the specified connection and returns the connection to the auto-commit mode. The current transaction includes all statements on the specified connection that were executed after the call to sqlsrv_begin_transaction and before any calls to **sqlsrv_rollback** or sqlsrv_commit.

NOTE

The Microsoft Drivers for PHP for SQL Server is in auto-commit mode by default. This means that all queries are automatically committed upon success unless they have been designated as part of an explicit transaction by using **sqlsrv_begin_transaction**.

NOTE

If **sqlsrv_rollback** is called on a connection that is not in an active transaction that was initiated with **sqlsrv_begin_transaction**, the call returns **false** and a *Not in Transaction* error is added to the error collection.

Syntax

```
sqlsrv_rollback( resource $conn)
```

Parameters

$conn: The connection on which the transaction is active.

Return Value

A Boolean value: **true** if the transaction was successfully rolled back. Otherwise, **false**.

Example

The following example executes two queries as part of a transaction. If both queries are successful, the transaction is committed. If either (or both) of the queries fail, the transaction is rolled back.

The first query in the example inserts a new sales order into the *Sales.SalesOrderDetail* table of the AdventureWorks database. The order is for five units of the product that has product ID 709. The second query reduces the inventory quantity of product ID 709 by five units. These queries are included in a transaction because both queries must be successful for the database to accurately reflect the state of orders and product availability.

The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```php
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true ));
}

/* Initiate transaction. */
/* Exit script if transaction cannot be initiated. */
if ( sqlsrv_begin_transaction( $conn) === false )
{
     echo "Could not begin transaction.\n";
     die( print_r( sqlsrv_errors(), true ));
}

/* Initialize parameter values. */
$orderId = 43659; $qty = 5; $productId = 709;
$offerId = 1; $price = 5.70;

/* Set up and execute the first query. */
$tsql1 = "INSERT INTO Sales.SalesOrderDetail
                     (SalesOrderID,
                      OrderQty,
                      ProductID,
                      SpecialOfferID,
                      UnitPrice)
          VALUES (?, ?, ?, ?, ?)";
$params1 = array( $orderId, $qty, $productId, $offerId, $price);
$stmt1 = sqlsrv_query( $conn, $tsql1, $params1 );

/* Set up and executee the second query. */
$tsql2 = "UPDATE Production.ProductInventory
          SET Quantity = (Quantity - ?)
          WHERE ProductID = ?";
$params2 = array($qty, $productId);
$stmt2 = sqlsrv_query( $conn, $tsql2, $params2 );

/* If both queries were successful, commit the transaction. */
/* Otherwise, rollback the transaction. */
if( $stmt1 && $stmt2 )
{
     sqlsrv_commit( $conn );
     echo "Transaction was committed.\n";
}
else
{
     sqlsrv_rollback( $conn );
     echo "Transaction was rolled back.\n";
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt1);
sqlsrv_free_stmt( $stmt2);
sqlsrv_close( $conn);
?>
```

For the purpose of focusing on transaction behavior, some recommended error handling is not included in the preceding example. For a production application, it is recommended that any call to a **sqlsrv** function be checked for errors and handled accordingly.

NOTE

Do not use embedded Transact-SQL to perform transactions. For example, do not execute a statement with "BEGIN TRANSACTION" as the Transact-SQL query to begin a transaction. The expected transactional behavior cannot be guaranteed when using embedded Transact-SQL to perform transactions.

See Also

SQLSRV Driver API Reference
How to: Perform Transactions
Overview of the PHP SQL Driver

sqlsrv_rows_affected

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Returns the number of rows modified by the last statement executed. This function does not return the number of rows returned by a SELECT statement.

Syntax

```
sqlsrv_rows_affected( resource $stmt)
```

Parameters

$stmt: A statement resource corresponding to an executed statement.

Return Value

An integer indicating the number of rows modified by the last executed statement. If no rows were modified, zero (0) is returned. If no information about the number of modified rows is available, negative one (-1) is returned. If an error occurred in retrieving the number of modified rows, **false** is returned.

Example

The following example displays the number of rows modified by an UPDATE statement. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Set up Transact-SQL query. */
$tsql = "UPDATE Sales.SalesOrderDetail
         SET SpecialOfferID = ?
         WHERE ProductID = ?";

/* Set parameter values. */
$params = array(2, 709);

/* Execute the statement. */
$stmt = sqlsrv_query( $conn, $tsql, $params);

/* Get the number of rows affected and display appropriate message.*/
$rows_affected = sqlsrv_rows_affected( $stmt);
if( $rows_affected === false)
{
     echo "Error in calling sqlsrv_rows_affected.\n";
     die( print_r( sqlsrv_errors(), true));
}
elseif( $rows_affected == -1)
{
      echo "No information available.\n";
}
else
{
      echo $rows_affected." rows were updated.\n";
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference
About Code Examples in the Documentation
Updating Data (Microsoft Drivers for PHP for SQL Server)

sqlsrv_send_stream_data

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Sends data from parameter streams to the server. Up to eight kilobytes (8K) of data is sent with each call to **sqlsrv_send_stream_data**.

NOTE

By default, all stream data is sent to the server when a query is executed. If this default behavior is not changed, you do not have to use **sqlsrv_send_stream_data** to send stream data to the server. For information about changing the default behavior, see the Parameters section of sqlsrv_query or sqlsrv_prepare.

Syntax

```
sqlsrv_send_stream_data( resource $stmt)
```

Parameters

$stmt: A statement resource corresponding to an executed statement.

Return Value

Boolean : **true** if there is more data to be sent. Otherwise, **false**.

Example

The following example opens a product review as a stream and sends it to the server. The default behavior of sending the all stream data at the time of execution is disabled. The example assumes that SQL Server and the AdventureWorks database are installed on the local computer. All output is written to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication and
specify the AdventureWorks database as the database in use. */
$serverName = "(local)";
$connectionInfo = array( "Database"=>"AdventureWorks");
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Define the query. */
$tsql = "UPDATE Production.ProductReview
         SET Comments = ( ?)
         WHERE ProductReviewID = 3";

/* Open parameter data as a stream and put it in the $params array. */
$comment = fopen( "data://text/plain,[ Insert lengthy comment.]", "r");
$params = array( &$comment);

/* Prepare the statement. Use the $options array to turn off the
default behavior, which is to send all stream data at the time of query
execution. */
$options = array("SendStreamParamsAtExec"=>0);
$stmt = sqlsrv_prepare( $conn, $tsql, $params, $options);

/* Execute the statement. */
sqlsrv_execute( $stmt);

/* Send up to 8K of parameter data to the server with each call to
sqlsrv_send_stream_data. Count the calls. */
$i = 1;
while( sqlsrv_send_stream_data( $stmt))
{
      echo "$i call(s) made.\n";
      $i++;
}

/* Free statement and connection resources. */
sqlsrv_free_stmt( $stmt);
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference
Updating Data (Microsoft Drivers for PHP for SQL Server)
About Code Examples in the Documentation

sqlsrv_server_info

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Returns information about the server. A connection must be established before calling this function.

Syntax

```
sqlsrv_server_info( resource $conn)
```

Parameters

$conn: The connection resource by which the client and server are connected.

Return Value

An associative array with the following keys:

KEY	DESCRIPTION
CurrentDatabase	The database currently being targeted.
SQLServerVersion	The version of SQL Server.
SQLServerName	The name of the server.

Example

The following example writes server information to the console when the example is run from the command line.

```
<?php
/* Connect to the local server using Windows Authentication. */
$serverName = "(local)";
$conn = sqlsrv_connect( $serverName);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

$server_info = sqlsrv_server_info( $conn);
if( $server_info )
{
      foreach( $server_info as $key => $value)
      {
             echo $key.": ".$value."\n";
      }
}
else
{
      echo "Error in retrieving server info.\n";
      die( print_r( sqlsrv_errors(), true));
}

/* Free connection resources. */
sqlsrv_close( $conn);
?>
```

See Also

SQLSRV Driver API Reference
About Code Examples in the Documentation

PDO_SQLSRV Driver Reference

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Two objects support PDO:

- PDO Class
- PDOStatement Class

For more information, see PDO.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

See Also

Overview of the PHP SQL Driver Constants (Microsoft Drivers for PHP for SQL Server)
Programming Guide for PHP SQL Driver Getting Started with the PHP SQL Driver

PDO Class

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The PDO class contains methods that allow your PHP application to connect to an SQL Server instance.

Syntax

```
PDO {}
```

Remarks

The PDO class contains the following methods:

PDO::__construct

PDO::beginTransaction

PDO::commit

PDO::errorCode

PDO::errorInfo

PDO::exec

PDO::getAttribute

PDO::getAvailableDrivers

PDO::lastInsertId

PDO::prepare

PDO::query

PDO::quote

PDO::rollback

PDO::setAttribute

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

See Also

PDO_SQLSRV Driver Reference
Overview of the PHP SQL Driver
Constants (Microsoft Drivers for PHP for SQL Server)
Programming Guide for PHP SQL Driver
Getting Started with the PHP SQL Driver
PDO

PDO::__construct

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Creates a connection to a SQL Server database.

Syntax

```
PDO::__construct($dsn [,$username [,$password [,$driver_options ]]] )
```

Parameters

$dsn: A string that contains the prefix name (always `sqlsrv`), a colon, and the Server keyword. For example `"sqlsrv:server=(local)"`. You can optionally specify other connection keywords. See Connection Options for a description of the Server keyword and the other connection keywords. The entire *$dsn* is in quotation marks, so each connection keyword should not be individually quoted.

$username: Optional. A string that contains the user's name. To connect using SQL Server Authentication, specify the login ID. To connect using Windows Authentication, specify `""`.

$password: Optional. A string that contains the user's password. To connect using SQL Server Authentication, specify the password. To connect using Windows Authentication, specify `""`.

$driver_options: Optional. You can specify PDO Driver Manager attributes, and Microsoft Drivers for PHP for SQL Server specific driver attributes -- PDO::SQLSRV_ATTR_ENCODING, PDO::SQLSRV_ATTR_DIRECT_QUERY. An invalid attribute will not generate an exception. Invalid attributes generate exceptions when specified with PDO::setAttribute.

Return Value

Returns a PDO object. If failure, returns a PDOException object.

Exceptions

PDOException

Remarks

You can close a connection object by setting the instance to null.

After a connection, PDO::errorCode will display 01000 instead of 00000.

If PDO::__construct fails for any reason, an exception will be thrown, even if PDO::ATTR_ERRMODE is set to PDO::ERRMODE_SILENT.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

This example shows how to connect to a server using Windows Authentication, and specify a database.

```
<?php
   $c = new PDO( "sqlsrv:Server=(local) ; Database = AdventureWorks ", "", "",
array(PDO::SQLSRV_ATTR_DIRECT_QUERY => true));

   $query = 'SELECT * FROM Person.ContactType';
   $stmt = $c->query( $query );
   while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ) {
      print_r( $row );
   }
   $c = null;
?>
```

Example

This example shows how to connect to a server, specifying the database later.

```
<?php
   $c = new PDO( "sqlsrv:server=(local)");

   $c->exec( "USE AdventureWorks");
   $query = 'SELECT * FROM Person.ContactType';
   $stmt = $c->query( $query );
   while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
      print_r( $row );
   }
   $c = null;
?>
```

See Also

PDO Class
PDO

PDO::beginTransaction

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Turns off auto commit mode and begins a transaction.

Syntax

```
bool PDO::beginTransaction();
```

Return Value

true if the method call succeeded, false otherwise.

Remarks

The transaction begun with PDO::beginTransaction will end when PDO::commit or PDO::rollback is called.

PDO::beginTransaction is not affected by (and does not affect) the value of PDO::ATTR_AUTOCOMMIT.

You are not allowed to call PDO::beginTransaction before the previous PDO::beginTransaction is ended with PDO::rollback or PDO::commit.

The connection will return to auto commit mode if this method fails.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

The following example uses a database called Test and a table called Table1. It starts a transaction and then issues commands to add two rows and then delete one row. The commands are sent to the database and the transaction is explicitly ended with `PDO::commit`.

```
<?php
   $conn = new PDO( "sqlsrv:server=(local); Database = Test", "", "");
   $conn->beginTransaction();
   $ret = $conn->exec("insert into Table1(col1, col2) values('a', 'b') ");
   $ret = $conn->exec("insert into Table1(col1, col2) values('a', 'c') ");
   $ret = $conn->exec("delete from Table1 where col1 = 'a' and col2 = 'b'");
   $conn->commit();
   // $conn->rollback();
   echo $ret;
?>
```

See Also

PDO Class
PDO

PDO::commit

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Sends commands to the database that were issued after calling PDO::beginTransaction and returns the connection to auto commit mode.

Syntax

```
bool PDO::commit();
```

Return Value

true if the method call succeeded, false otherwise.

Remarks

PDO::commit is not affected by (and does not affect) the value of PDO::ATTR_AUTOCOMMIT.

See PDO::beginTransaction for an example that uses PDO::commit.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

See Also

PDO Class
PDO

PDO::errorCode

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

PDO::errorCode retrieves the SQLSTATE of the most recent operation on the database handle.

Syntax

```
mixed PDO::errorCode();
```

Return Value

PDO::errorCode returns a five-char SQLSTATE as a string or NULL if there was no operation on the database handle.

Remarks

PDO::errorCode in the PDO_SQLSRV driver will return warnings on some successful operations. For example, on a successful connection, PDO::errorCode will return "01000" indicating SQL_SUCCESS_WITH_INFO.

PDO::errorCode only retrieves error codes for operations performed directly on the database connection. If you create a PDOStatement instance through PDO::prepare or PDO::query and generate an error on the statement object, PDO::errorCode will not retrieve that error. You must call PDOStatement::errorCode to return the error code for an operation performed on a particular statement object.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

In this example, the name of the column is misspelled (`Cityx` instead of `City`), causing an error, which is then reported.

```
<?php
$conn = new PDO( "sqlsrv:server=(local) ; Database = AdventureWorks ", "", "");
$query = "SELECT * FROM Person.Address where Cityx = 'Essen'";

$conn->query($query);
print $conn->errorCode();
?>
```

See Also

PDO Class
PDO

PDO::errorInfo

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Retrieves extended error information of the most recent operation on the database handle.

Syntax

```
array PDO::errorInfo();
```

Return Value

An array of error information about the most recent operation on the database handle. The array consists of the following fields:

- The SQLSTATE error code.
- The driver-specific error code.
- The driver-specific error message.

If there is no error, or if the SQLSTATE is not set, the driver-specific fields will be NULL.

Remarks

PDO::errorInfo only retrieves error information for operations performed directly on the database. Use PDOStatement::errorInfo when a PDOStatement instance is created using PDO::prepare or PDO::query.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

In this example, the name of the column is misspelled (`Cityx` instead of `City`), causing an error, which is then reported.

```
<?php
$conn = new PDO( "sqlsrv:server=(local) ; Database = AdventureWorks ", "");
$query = "SELECT * FROM Person.Address where Cityx = 'Essen'";

$conn->query($query);
print $conn->errorCode();
echo "\n";
print_r ($conn->errorInfo());
?>
```

See Also

PDO Class
PDO

PDO::exec

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Prepares and executes an SQL statement in a single function call, returning the number of rows affected by the statement.

Syntax

```
int PDO::exec ($statement)
```

Parameters

$statement: A string containing the SQL statement to execute.

Return Value

An integer reporting the number of rows affected.

Remarks

If *$statement* contains multiple SQL statements, the count of affected rows is reported for the last statement only.

PDO::exec does not return results for a SELECT statement.

The following attributes affect the behavior of PDO::exec:

- PDO::ATTR_DEFAULT_FETCH_MODE
- PDO::SQLSRV_ATTR_ENCODING
- PDO::SQLSRV_ATTR_QUERY_TIMEOUT

See PDO::setAttribute for more information.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

This example deletes rows in Table1 that have 'xxxyy' in col1. The example then reports how many rows were deleted.

```
<?php
   $c = new PDO( "sqlsrv:server=(local)");

   $c->exec("use Test");
   $ret = $c->exec("delete from Table1 where col1 = 'xxxyy'");
   echo $ret;
?>
```

See Also

PDO Class

PDO

PDO::getAttribute

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

Retrieves the value of a predefined PDO or driver attribute.

Syntax

```
mixed PDO::getAttribute ( $attribute )
```

Parameters

$attribute: One of the supported attributes. See the Remarks section for the list of supported attributes.

Return Value

On success, returns the value of a connection option, predefined PDO attribute, or custom driver attribute. On failure, returns null.

Remarks

The following table contains the list of supported attributes.

ATTRIBUTE	PROCESSED BY	SUPPORTED VALUES	DESCRIPTION
PDO::ATTR_CASE	PDO	PDO::CASE_LOWER PDO::CASE_NATURAL PDO::CASE_UPPER	Specifies whether the column names should be in a specific case. PDO::CASE_LOWER forces lower case column names, PDO::CASE_NATURAL leaves the column name as returned by the database, and PDO::CASE_UPPER forces column names to upper case. The default is PDO::CASE_NATURAL. This attribute can also be set using PDO::setAttribute.

ATTRIBUTE	PROCESSED BY	SUPPORTED VALUES	DESCRIPTION
PDO::ATTR_CLIENT_VERSION	Microsoft Drivers for PHP for SQL Server	Array of strings	Describes the versions of the driver and related libraries. Returns an array with the following elements: ODBC version (*MajorVer.MinorVer*), SQL Server Native Client DLL name and version, Microsoft Drivers for PHP for SQL Server version (*MajorVer.MinorVer.BuildNumber.Revision*)
PDO::ATTR_DEFAULT_FETCH_MODE	PDO	See the PDO documentation.	See the PDO documentation.
PDO::ATTR_DRIVER_NAME	PDO	String	Always returns "sqlsrv".
PDO::ATTR_DRIVER_VERSION	Microsoft Drivers for PHP for SQL Server	String	Indicates the Microsoft Drivers for PHP for SQL Server version (*MajorVer.MinorVer.BuildNumber.Revision*)
PDO::ATTR_ERRMODE	PDO	PDO::ERRMODE_SILENT PDO::ERRMODE_WARNING PDO::ERRMODE_EXCEPTION	Specifies how failures should be handled by the driver. PDO::ERRMODE_SILENT (the default) sets the error codes and information. PDO::ERRMODE_WARNING raises an E_WARNING. PDO::ERRMODE_EXCEPTION raises an exception. This attribute can also be set using PDO::setAttribute.
PDO::ATTR_ORACLE_NULLS	PDO	See the PDO documentation.	See the PDO documentation.
PDO::ATTR_SERVER_INFO	Microsoft Drivers for PHP for SQL Server	Array of 3 elements	Returns the current database, SQL Server version, and SQL Server instance.
PDO::ATTR_SERVER_VERSION	Microsoft Drivers for PHP for SQL Server	String	Indicates the SQL Server version (*Major.Minor.BuildNumber*)
PDO::ATTR_STATEMENT_CLASS	PDO	See PDO documentation	See PDO documentation. (returns PDOStatement)
PDO::ATTR_STRINGIFY_FETCHES	PDO	See PDO documentation	See the PDO documentation.

ATTRIBUTE	PROCESSED BY	SUPPORTED VALUES	DESCRIPTION
PDO::SQLSRV_ATTR_CLIENT_BUFFER_MAX_KB_SIZE	Microsoft Drivers for PHP for SQL Server	1 to the PHP memory limit.	Configures the size of the buffer that holds the result set for a client-side cursor. The default is 10240 KB (10 MB). For more information about client-side cursors, see Cursor Types (SQLSRV Driver).
PDO::SQLSRV_ATTR_DIRECT_QUERY	Microsoft Drivers for PHP for SQL Server	true false	Specifies direct or prepared query execution. For more information, see Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver.
PDO::SQLSRV_ATTR_ENCODING	Microsoft Drivers for PHP for SQL Server	One of the following: PDO::SQLSRV_ENCODING_UTF8 PDO::SQLSRV_ENCODING_SYSTEM	Specifies the character set encoding used by the driver to communicate with the server. The default is PDO::SQLSRV_ENCODING_UTF8.

PDO processes some of the predefined attributes while it requires the driver to handle others. All custom attributes and connection options are handled by the driver, an unsupported attribute or connection option will return null.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

This example shows the value of the PDO::ATTR_ERRMODE attribute, before and after changing its value.

```
<?php
$database = "AdventureWorks";
$conn = new PDO( "sqlsrv:server=(local) ; Database = $database", "", "");

$attributes1 = array( "ERRMODE" );
foreach ( $attributes1 as $val ) {
     echo "PDO::ATTR_$val: ";
     var_dump ($conn->getAttribute( constant( "PDO::ATTR_$val" ) ));
}

$conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );

$attributes1 = array( "ERRMODE" );
foreach ( $attributes1 as $val ) {
     echo "PDO::ATTR_$val: ";
     var_dump ($conn->getAttribute( constant( "PDO::ATTR_$val" ) ));
}

// An example using PDO::ATTR_CLIENT_VERSION
print_r($conn->getAttribute( PDO::ATTR_CLIENT_VERSION ));
?>
```

See Also

PDO Class

PDO

PDO::getAvailableDrivers

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Returns an array of the PDO drivers in your PHP installation.

Syntax

```
array PDO::getAvailableDrivers ();
```

Return Value

An array with the list of PDO drivers.

Remarks

The name of the PDO driver is used in PDO::__construct, to create a PDO instance.

PDO::getAvailableDrivers is not required to be implemented by PHP drivers. For more information about this method, see the PHP documentation.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
print_r(PDO::getAvailableDrivers());
?>
```

See Also

PDO Class
PDO

PDO::lastInsertId

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Returns the identifier for the row most recently inserted into a table in the database. The table must have an IDENTITY NOT NULL column.

Syntax

```
string PDO::lastInsertId ([ $name ] );
```

Parameters

$*name*: An optional string that lets you specify the table.

Return Value

A string of the identifier for the row most recently added. An empty string if the method call fails.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
   $database = "test";
   $server = "(local)";
   $conn = new PDO( "sqlsrv:server=$server; Database = $database", "", "");

   $conn->exec("use Test");

   $ret = $conn->exec("INSERT INTO Table1 VALUES( '19' )");
   $ret = $conn->exec("INSERT INTO ScrollTest VALUES( 1, '19' )");

   $lastRow = $conn->lastInsertId('Table1');
   echo $lastRow . "\n";

   // defaults to ScrollTest
   $lastRow = $conn->lastInsertId();
   echo $lastRow . "\n";
?>
```

See Also

PDO Class
PDO

PDO::prepare

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

Prepares a statement for execution.

Syntax

```
PDOStatement PDO::prepare ( $statement [, array(key_pair)] )
```

Parameters

$*statement*: A string containing the SQL statement.

key_pair: An array containing an attribute name and value. See the Remarks section for more information.

Return Value

Returns a PDOStatement object on success. On failure, returns a PDOException object, or false depending on the value of PDO::ATTR_ERRMODE.

Remarks

The Microsoft Drivers for PHP for SQL Server does not evaluate prepared statements until execution.

The following table lists the possible *key_pair* values.

KEY	DESCRIPTION
PDO::ATTR_CURSOR	Specifies cursor behavior. The default is PDO::CURSOR_FWDONLY. PDO::CURSOR_SCROLL is a static cursor. For example, `array( PDO::ATTR_CURSOR => PDO::CURSOR_FWDONLY )`. If you use PDO::CURSOR_SCROLL, you can use PDO::SQLSRV_ATTR_CURSOR_SCROLL_TYPE, which is described below. See Cursor Types (PDO_SQLSRV Driver) for more information about result sets and cursors in the PDO_SQLSRV driver.

KEY	DESCRIPTION
PDO::ATTR_EMULATE_PREPARES	The purpose of PDO::ATTR_EMULATE_PREPARES is described in the PHP manual. SQL Server does not support named or positional parameters in some Transact-SQL clauses. If your PHP application must use parameters in a Transact-SQL clause that will generate an error on the server, you can set the PDO::ATTR_EMULATE_PREPARES attribute to true. For example: `PDO::ATTR_EMULATE_PREPARES => true` By default, this attribute is set to false. **Note:** The security of parameterized queries is not in effect when you use `PDO::ATTR_EMULATE_PREPARES => true`. Your application should ensure that the data that is bound to the parameter(s) does not contain malicious Transact-SQL code.
PDO::SQLSRV_ATTR_ENCODING	PDO::SQLSRV_ENCODING_UTF8 (default) PDO::SQLSRV_ENCODING_SYSTEM PDO::SQLSRV_ENCODING_BINARY
PDO::SQLSRV_ATTR_DIRECT_QUERY	When True, specifies direct query execution. False means prepared statement execution. For more information about PDO::SQLSRV_ATTR_DIRECT_QUERY, see Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver.
PDO::SQLSRV_ATTR_QUERY_TIMEOUT	For more information, see PDO::setAttribute.

When you use PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL, you can use PDO::SQLSRV_ATTR_CURSOR_SCROLL_TYPE. For example,

```
array(PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL, PDO::SQLSRV_ATTR_CURSOR_SCROLL_TYPE =>
PDO::SQLSRV_CURSOR_DYNAMIC));
```

The following table shows the possible values for PDO::SQLSRV_ATTR_CURSOR_SCROLL_TYPE.

VALUE	DESCRIPTION
PDO::SQLSRV_CURSOR_BUFFERED	Creates a client-side (buffered) static cursor. For more information about client-side cursors, see Cursor Types (PDO_SQLSRV Driver).
PDO::SQLSRV_CURSOR_DYNAMIC	Creates a server-side (unbuffered) dynamic cursor, which lets you access rows in any order and will reflect changes in the database.
PDO::SQLSRV_CURSOR_KEYSET_DRIVEN	Creates a server-side keyset cursor. A keyset cursor does not update the row count if a row is deleted from the table (a deleted row is returned with no values).

VALUE	DESCRIPTION
PDO::SQLSRV_CURSOR_STATIC	Creates a server-side static cursor, which lets you access rows in any order but will not reflect changes in the database. PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL implies PDO::SQLSRV_ATTR_CURSOR_SCROLL_TYPE => PDO::SQLSRV_CURSOR_STATIC.

You can close a PDOStatement object by setting it to null.

Example

This example shows how to use the PDO::prepare method with parameter markers and a forward-only cursor.

```
<?php
$database = "Test";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$col1 = 'a';
$col2 = 'b';

$query = "insert into Table1(col1, col2) values(?, ?)";
$stmt = $conn->prepare( $query, array( PDO::ATTR_CURSOR => PDO::CURSOR_FWDONLY,
PDO::SQLSRV_ATTR_QUERY_TIMEOUT => 1  ) );
$stmt->execute( array( $col1, $col2 ) );
print $stmt->rowCount();
echo "\n";

$query = "insert into Table1(col1, col2) values(:col1, :col2)";
$stmt = $conn->prepare( $query, array( PDO::ATTR_CURSOR => PDO::CURSOR_FWDONLY,
PDO::SQLSRV_ATTR_QUERY_TIMEOUT => 1  ) );
$stmt->execute( array( ':col1' => $col1, ':col2' => $col2 ) );
print $stmt->rowCount();

$stmt = null
?>
```

Example

This example shows how to use the PDO::prepare method with a client-side cursor. For a sample showing a server-side cursor, see Cursor Types (PDO_SQLSRV Driver).

```
<?php
$database = "AdventureWorks";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$query = "select * from Person.ContactType";
$stmt = $conn->prepare( $query, array(PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL));
$stmt->execute();

echo "\n";

while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
   print "$row[Name]\n";
}
echo "\n..\n";

$row = $stmt->fetch( PDO::FETCH_BOTH, PDO::FETCH_ORI_FIRST );
print_r($row);

$row = $stmt->fetch( PDO::FETCH_ASSOC, PDO::FETCH_ORI_REL, 1 );
print "$row[Name]\n";

$row = $stmt->fetch( PDO::FETCH_NUM, PDO::FETCH_ORI_NEXT );
print "$row[1]\n";

$row = $stmt->fetch( PDO::FETCH_NUM, PDO::FETCH_ORI_PRIOR );
print "$row[1]..\n";

$row = $stmt->fetch( PDO::FETCH_NUM, PDO::FETCH_ORI_ABS, 0 );
print_r($row);

$row = $stmt->fetch( PDO::FETCH_NUM, PDO::FETCH_ORI_LAST );
print_r($row);
?>
```

See Also

PDO Class
PDO

PDO::query

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

Executes an SQL query and returns a result set as a PDOStatement object.

Syntax

```
PDOStatement PDO::query ($statement[, $fetch_style);
```

Parameters

$statement: The SQL statement you want to execute.

$fetch_style: The optional instructions on how to perform the query. See the Remarks section for more details. $*fetch_style* in PDO::query can be overridden with $*fetch_style* in PDO::fetch.

Return Value

If the call succeeds, PDO::query returns a PDOStatement object. If the call fails, PDO::query throws a PDOException object or returns false, depending on the setting of PDO::ATTR_ERRMODE.

Exceptions

PDOException.

Remarks

A query executed with PDO::query can execute either a prepared statement or directly, depending on the setting of PDO::SQLSRV_ATTR_DIRECT_QUERY; see Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver for more information.

PDO::SQLSRV_ATTR_QUERY_TIMEOUT also affects the behavior of PDO::exec; see PDO::setAttribute for more information.

You can specify the following options for $*fetch_style*.

STYLE	DESCRIPTION
PDO::FETCH_COLUMN, *num*	Queries for data in the specified column. The first column in the table is column 0.
PDO::FETCH_CLASS, '*classname*', array(*arglist*)	Creates an instance of a class and assigns column names to properties in the class. If the class constructor takes one or more parameters, you can also pass an *arglist*.
PDO::FETCH_CLASS, '*classname*'	Assigns column names to properties in an existing class.

Call PDOStatement::closeCursor to release database resources associated with the PDOStatement object before calling PDO::query again.

You can close a PDOStatement object by setting it to null.

If all the data in a result set is not fetched, the next PDO::query call will not fail.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

This example shows several queries.

```
<?php
$database = "AdventureWorks";
$conn = new PDO( "sqlsrv:server=(local) ; Database = $database", "", "");
$conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
$conn->setAttribute( PDO::SQLSRV_ATTR_QUERY_TIMEOUT, 1 );

$query = 'select * from Person.ContactType';

// simple query
$stmt = $conn->query( $query );
while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
   print_r( $row['Name'] ."\n" );
}

echo "\n........ query for a column ............\n";

// query for one column
$stmt = $conn->query( $query, PDO::FETCH_COLUMN, 1 );
while ( $row = $stmt->fetch() ){
   echo "$row\n";
}

echo "\n........ query with a new class ............\n";
$query = 'select * from HumanResources.Department order by GroupName';
// query with a class
class cc {
   function __construct( $arg ) {
      echo "$arg";
   }

   function __toString() {
      return $this->DepartmentID . "; " . $this->Name . "; " . $this->GroupName;
   }
}

$stmt = $conn->query( $query, PDO::FETCH_CLASS, 'cc', array( "arg1 " ));

while ( $row = $stmt->fetch() ){
   echo "$row\n";
}

echo "\n........ query into an existing class ............\n";
$c_obj = new cc( '' );
$stmt = $conn->query( $query, PDO::FETCH_INTO, $c_obj );
while ( $stmt->fetch() ){
   echo "$c_obj\n";
}

$stmt = null;
?>
```

See Also

PDO Class

PDO::quote

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Processes a string for use in a query by placing quotes around the input string as required by the underlying SQL Server database. PDO::quote will escape special characters within the input string using a quoting style appropriate to SQL Server.

Syntax

```
string PDO::quote( $string[, $parameter_type ] )
```

Parameters

$*string*: The string to quote.

$*parameter_type*: An optional (integer) symbol indicating the data type. The default is PDO::PARAM_STR.

Return Value

A quoted string that can be passed to an SQL statement, or false if failure.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
$database = "test";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$param = 'a \' g';
$param2 = $conn->quote( $param );

$query = "INSERT INTO Table1 VALUES( ?, '1' )";
$stmt = $conn->prepare( $query );
$stmt->execute(array($param));

$query = "INSERT INTO Table1 VALUES( ?, ? )";
$stmt = $conn->prepare( $query );
$stmt->execute(array($param, $param2));
?>
```

See Also

PDO Class
PDO

PDO::rollback

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Discards database commands that were issued after calling PDO::beginTransaction and returns the connection to auto commit mode.

Syntax

```
bool PDO::rollBack ();
```

Return Value

true if the method call succeeded, false otherwise.

Remarks

PDO::rollback is not affected by (and does not affect) the value of PDO::ATTR_AUTOCOMMIT.

See PDO::beginTransaction for an example that uses PDO::rollback.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

See Also

PDO Class
PDO

PDO::setAttribute

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

Sets a predefined PDO attribute or a custom driver attribute.

Syntax

```
bool PDO::setAttribute ( $attribute, $value );
```

Parameters

$attribute: The attribute to set. See the Remarks section for a list of supported attributes.

$value: The value (type mixed).

Return Value

Returns true on success, otherwise false.

Remarks

ATTRIBUTE	PROCESSED BY	SUPPORTED VALUES	DESCRIPTION
PDO::ATTR_CASE	PDO	PDO::CASE_LOWER PDO::CASE_NATURAL PDO::CASE_UPPER	Specifies the case of column names. PDO::CASE_LOWER causes lower case column names. PDO::CASE_NATURAL (the default) displays column names as returned by the database. PDO::CASE_UPPER causes column names to upper case. This attribute can be set using PDO::setAttribute.
PDO::ATTR_DEFAULT_FETCH_MODE	PDO	See the PDO documentation.	See the PDO documentation.

ATTRIBUTE	PROCESSED BY	SUPPORTED VALUES	DESCRIPTION
PDO::ATTR_ERRMODE	PDO	PDO::ERRMODE_SILENT PDO::ERRMODE_WARNING PDO::ERRMODE_EXCEPTION	Specifies how the driver will report failures. PDO::ERRMODE_SILENT (the default) sets the error codes and information. PDO::ERRMODE_WARNING raises E_WARNING. PDO::ERRMODE_EXCEPTION causes an exception to be thrown. This attribute can be set using PDO::setAttribute.
PDO::ATTR_ORACLE_NULLS	PDO	See the PDO documentation.	Specifies how nulls should be returned. PDO::NULL_NATURAL does no conversion. PDO::NULL_EMPTY_STRING converts an empty string to null. PDO::NULL_TO_STRING converts null to an empty string.
PDO::ATTR_STATEMENT_CLASS	PDO	See the PDO documentation.	Sets the user-supplied statement class derived from PDOStatement. Requires `array(string classname, array(mixed constructor_args))`. See the PDO documentation for more information.
PDO::ATTR_STRINGIFY_FETCHES	PDO	true or false	Converts numeric values to strings when retrieving data.

ATTRIBUTE	PROCESSED BY	SUPPORTED VALUES	DESCRIPTION
PDO::SQLSRV_ATTR_CLIENT_BUFFER_MAX_KB_SIZE	Microsoft Drivers for PHP for SQL Server	1 to the PHP memory limit.	Configures the size of the buffer that holds the result set. The default is 10240 KB (10 MB). For more information about queries that create a client-side cursor, see Cursor Types (PDO_SQLSRV Driver).
PDO::SQLSRV_ATTR_DIRECT_QUERY	Microsoft Drivers for PHP for SQL Server	true false	Specifies direct or prepared query execution. For more information, see Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver.
PDO::SQLSRV_ATTR_ENCODING	Microsoft Drivers for PHP for SQL Server	PDO::SQLSRV_ENCODING_UTF8 PDO::SQLSRV_ENCODING_SYSTEM.	Sets the character set encoding used by the driver to communicate with the server. PDO::SQLSRV_ENCODING_BINARY is not supported. The default is PDO::SQLSRV_ENCODING_UTF8.
PDO::SQLSRV_ATTR_FETCHES_NUMERIC_TYPE	Microsoft Drivers for PHP for SQL Server	true or false	Handles numeric fetches from columns with numeric SQL types (bit, integer, smallint, tinyint, float, or real). When connection option flag ATTR_STRINGIFY_FETCHES is on, even when SQLSRV_ATTR_FETCHES_NUMERIC_TYPE is on, the return value will still be string. When the returned PDO type in bind column is PDO_PARAM_INT, the return value from an integer column will be int even if SQLSRV_ATTR_FETCHES_NUMERIC_TYPE is off.

ATTRIBUTE	PROCESSED BY	SUPPORTED VALUES	DESCRIPTION
PDO::SQLSRV_ATTR_QUERY_TIMEOUT	Microsoft Drivers for PHP for SQL Server	integer	Sets the query timeout in seconds. The default is 0, which means the driver will wait indefinitely for results. Negative numbers are not allowed.
PDO::SQLSVR_CLIENT_BUFFER_MAX_SIZE	Microsoft Drivers for PHP for SQL Server	integer	Sets the size of the query buffer. The default is 0, which indicates unlimited buffer size. Negative numbers are not allowed. For more information about queries that create a client-side cursor, see Cursor Types (PDO_SQLSRV Driver).

PDO processes some of the predefined attributes and requires the driver to process others. All custom attributes and connection options are processed by the driver. An unsupported attribute, connection option, or unsupported value will be reported according to the setting of PDO::ATTR_ERRMODE.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

This sample shows how to set the PDO::ATTR_ERRMODE attribute.

```
<?php
   $database = "AdventureWorks";
   $conn = new PDO( "sqlsrv:server=(local) ; Database = $database", "", "");

   $attributes1 = array( "ERRMODE" );
   foreach ( $attributes1 as $val ) {
      echo "PDO::ATTR_$val: ";
      var_dump ($conn->getAttribute( constant( "PDO::ATTR_$val" ) ));
   }

   $conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );

   $attributes1 = array( "ERRMODE" );
   foreach ( $attributes1 as $val ) {
      echo "PDO::ATTR_$val: ";
      var_dump ($conn->getAttribute( constant( "PDO::ATTR_$val" ) ));
   }
?>
```

See Also

PDO Class

PDOStatement Class

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

The PDOStatement class represents a statement and the results of the statement.

Syntax

```
PDOStatement {}
```

Remarks

The PDOStatement class contains the following methods:

- PDOStatement::bindColumn
- PDOStatement::bindParam
- PDOStatement::bindValue
- PDOStatement::closeCursor
- PDOStatement::columnCount
- PDOStatement::debugDumpParams
- PDOStatement::errorCode
- PDOStatement::errorInfo
- PDOStatement::execute
- PDOStatement::fetch
- PDOStatement::fetchAll
- PDOStatement::fetchColumn
- PDOStatement::fetchObject
- PDOStatement::getAttribute
- PDOStatement::getColumnMeta
- PDOStatement::nextRowset
- PDOStatement::rowCount
- PDOStatement::setAttribute
- PDOStatement::setFetchMode

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

See Also

PDO_SQLSRV Driver Reference
Overview of the PHP SQL Driver Constants (Microsoft Drivers for PHP for SQL Server)
Programming Guide for PHP SQL Driver Getting Started with the PHP SQL Driver PDO

PDOStatement::bindColumn

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Binds a variable to a column in a result set.

Syntax

```
bool PDOStatement::bindColumn($column, &$param[, $type[, $maxLen[, $driverdata ]]] );
```

Parameters

$*column*: The (mixed) number of the column (1-based index) or name of the column in the result set.

&$*param*: The (mixed) name of the PHP variable to which the column will be bound.

$*type*: The optional data type of the parameter, represented by a PDO::PARAM_* constant.

$*maxLen*: Optional integer, not used by the Microsoft Drivers for PHP for SQL Server.

$*driverdata*: Optional mixed parameter(s) for the driver. For example, you could specify PDO::SQLSRV_ENCODING_UTF8 to bind the column to a variable as a string encoded in UTF-8.

Return Value

TRUE if success, otherwise FALSE.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

This example shows how a variable can be bound to a column in a result set.

```
<?php
$database = "AdventureWorks";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$query = "SELECT Title, FirstName, EmailAddress FROM Person.Contact where LastName = 'Estes'";
$stmt = $conn->prepare($query);
$stmt->execute();

$stmt->bindColumn('EmailAddress', $email);
while ( $row = $stmt->fetch( PDO::FETCH_BOUND ) ){
   echo "$email\n";
}
?>
```

See Also

PDOStatement Class

PDO

PDOStatement::bindParam

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

Binds a parameter to a named or question mark placeholder in the SQL statement.

Syntax

```
bool PDOStatement::bindParam( $parameter, &$variable [,$data_type[, $length[, $driver_options]]] );
```

Parameters

$*parameter*: A (mixed) parameter identifier. For a statement using named placeholders, a parameter name (:name). For a prepared statement using the question mark syntax, this will be the 1-based index of the parameter.

&$*variable*: The (mixed) name of the PHP variable to bind to the SQL statement parameter.

$*data_type*: An optional (integer) PDO::PARAM_* constant. Default is PDO::PARAM_STR.

$*length*: An optional (integer) length of the data type. You can specify PDO::SQLSRV_PARAM_OUT_DEFAULT_SIZE to indicate the default size when using PDO::PARAM_INT or PDO::PARAM_BOOL in $*data_type*.

$*driver_options*: The optional (mixed) driver-specific options. For example, you could specify PDO::SQLSRV_ENCODING_UTF8 to bind the column to a variable as a string encoded in UTF-8.

Return Value

TRUE on success, otherwise FALSE.

Remarks

When binding null data to server columns of type varbinary, binary, or varbinary(max) you should specify binary encoding (PDO::SQLSRV_ENCODING_BINARY) using the $*driver_options*. See Constants for more information about encoding constants.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

This code sample shows that after $contact is bound to the parameter, changing the value does change the value passed in the query.

```
<?php
$database = "AdventureWorks";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$contact = "Sales Agent";
$stmt = $conn->prepare("select * from Person.ContactType where name = ?");
$stmt->bindParam(1, $contact);
$contact = "Owner";
$stmt->execute();

while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
   print "$row[Name]\n\n";
}

$stmt = null;
$contact = "Sales Agent";
$stmt = $conn->prepare("select * from Person.ContactType where name = :contact");
$stmt->bindParam(':contact', $contact);
$contact = "Owner";
$stmt->execute();

while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
   print "$row[Name]\n\n";
}
?>
```

Example

This code sample shows how to access an output parameter.

```
<?php
$database = "Test";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$input1 = 'bb';

$stmt = $conn->prepare("select ? = count(* ) from Sys.tables");
$stmt->bindParam( 1, $input1, PDO::PARAM_STR, 10 );
$stmt->execute();
echo $input1;
?>
```

Example

This code sample shows how to use an input/output parameter.

```
<?php
   $database = "AdventureWorks";
   $server = "(local)";
   $dbh = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

   $dbh->query("IF OBJECT_ID('dbo.sp_ReverseString', 'P') IS NOT NULL DROP PROCEDURE dbo.sp_ReverseString");
   $dbh->query("CREATE PROCEDURE dbo.sp_ReverseString @String as VARCHAR(2048) OUTPUT as SELECT @String = 
REVERSE(@String)");
   $stmt = $dbh->prepare("EXEC dbo.sp_ReverseString ?");
   $string = "123456789";
   $stmt->bindParam(1, $string, PDO::PARAM_STR | PDO::PARAM_INPUT_OUTPUT, 2048);
   $stmt->execute();
   print $string;   // Expect 987654321
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::bindValue

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Binds a value to a named or question mark placeholder in the SQL statement.

Syntax

```
bool PDOStatement::bindValue( $parameter, $value [,$data_type] );
```

Parameters

$*parameter*: A (mixed) parameter identifier. For a statement using named placeholders, a parameter name (:name). For a prepared statement using the question mark syntax, this will be the 1-based index of the parameter.

$*value*: The (mixed) value to bind to the parameter.

$*data_type*: The optional (integer) data type represented by a PDO::PARAM_* constant. The default is PDO::PARAM_STR.

Return Value

TRUE on success, otherwise FALSE.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

This example shows that after the value of $contact is bound, changing the value does not change the value passed in the query.

```
<?php
$database = "AdventureWorks";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$contact = "Sales Agent";
$stmt = $conn->prepare("select * from Person.ContactType where name = ?");
$stmt->bindValue(1, $contact);
$contact = "Owner";
$stmt->execute();

while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
   print "$row[Name]\n\n";
}

$stmt = null;
$contact = "Sales Agent";
$stmt = $conn->prepare("select * from Person.ContactType where name = :contact");
$stmt->bindValue(':contact', $contact);
$contact = "Owner";
$stmt->execute();

while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
   print "$row[Name]\n\n";
}
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::closeCursor

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Closes the cursor, enabling the statement to be executed again.

Syntax

```
bool PDOStatement::closeCursor();
```

Return Value

true on success, otherwise false.

Remarks

closeCursor has an effect when the MultipleActiveResultSets connection option is set to false. For more information about the MultipleActiveResultSets connection option, see How to: Disable Multiple Active Resultsets (MARS).

Instead of calling closeCursor, you can also just set the statement handle to null.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
$database = "AdventureWorks";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "", array('MultipleActiveResultSets' =>
false ) );

$stmt = $conn->prepare('SELECT * FROM Person.ContactType');

$stmt2 = $conn->prepare('SELECT * FROM HumanResources.Department');

$stmt->execute();

$result = $stmt->fetch();
print_r($result);

$stmt->closeCursor();

$stmt2->execute();
$result = $stmt2->fetch();
print_r($result);
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::columnCount

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Returns the number of columns in a result set.

Syntax

```
int PDOStatement::columnCount ();
```

Return Value

Returns the number of columns in a result set. Returns zero if the result set is empty.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
$database = "AdventureWorks";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$query = "select * from Person.ContactType";
$stmt = $conn->prepare( $query );
print $stmt->columnCount();   // 0

echo "\n";
$stmt->execute();
print $stmt->columnCount();

echo "\n";
$stmt = $conn->query("select * from HumanResources.Department");
print $stmt->columnCount();
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::debugDumpParams

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Displays a prepared statement.

Syntax

```
bool PDOStatement::debugDumpParams();
```

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
$database = "AdventureWorks";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$param = "Owner";

$stmt = $conn->prepare("select * from Person.ContactType where name = :param");
$stmt->execute(array($param));
$stmt->debugDumpParams();

echo "\n\n";

$stmt = $conn->prepare("select * from Person.ContactType where name = ?");
$stmt->execute(array($param));
$stmt->debugDumpParams();
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::errorCode

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Retrieves the SQLSTATE of the most recent operation on the database statement object.

Syntax

```
string PDOStatement::errorCode();
```

Return Value

Returns a five-char SQLSTATE as a string, or NULL if there was no operation on the statement handle.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

This example shows a SQL query that has an error. The error code is then displayed.

```
<?php
$conn = new PDO( "sqlsrv:server=(local) ; Database = AdventureWorks", "", "");
$stmt = $conn->prepare('SELECT * FROM Person.Addressx');

$stmt->execute();
print $stmt->errorCode();
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::errorInfo

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Retrieves extended error information of the most recent operation on the statement handle.

Syntax

```
array PDOStatement::errorInfo();
```

Return Value

An array of error information about the most recent operation on the statement handle. The array consists of the following fields:

- The SQLSTATE error code
- The driver-specific error code
- The driver-specific error message

If there is no error, or if the SQLSTATE is not set, the driver-specific fields will be NULL.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

In this example, the SQL statement has an error, which is then reported.

```
<?php
$conn = new PDO( "sqlsrv:server=(local) ; Database = AdventureWorks", "", "");
$stmt = $conn->prepare('SELECT * FROM Person.Addressx');

$stmt->execute();
print_r ($stmt->errorInfo());
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::execute

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Executes a statement.

Syntax

```
bool PDOStatement::execute ([ $input ] );
```

Parameters

$input: (Optional) An associative array containing the values for parameter markers.

Return Value

true on success, false otherwise.

Remarks

Statements executed with PDOStatement::execute must first be prepared with PDO::prepare. See Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver for information on how to specify direct or prepared statement execution.

All values of the input parameters array are treated as PDO::PARAM_STR values.

If the prepared statement includes parameter markers, you must either call PDOStatement::bindParam to bind the PHP variables to the parameter markers or pass an array of input-only parameter values.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
$database = "AdventureWorks";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$query = "select * from Person.ContactType";
$stmt = $conn->prepare( $query );
$stmt->execute();

while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
   print "$row[Name]\n";
}

echo "\n";
$param = "Owner";
$query = "select * from Person.ContactType where name = ?";
$stmt = $conn->prepare( $query );
$stmt->execute(array($param));

while ( $row = $stmt->fetch( PDO::FETCH_ASSOC ) ){
   print "$row[Name]\n";
}
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::fetch

3/14/2017 • 3 min to read • Edit on GitHub

Download PHP Driver

Retrieves a row from a result set.

Syntax

```
mixed PDOStatement::fetch ([ $fetch_style[, $cursor_orientation[, $cursor_offset]]] );
```

Parameters

$*fetch_style*: An optional (integer) symbol specifying the format of the row data. See the Remarks section for the list of possible values for $*fetch_style*. Default is PDO::FETCH_BOTH. $*fetch_style* in the fetch method will override the $*fetch_style* specified in the PDO::query method.

$*cursor_orientation*: An optional (integer) symbol indicating the row to retrieve when the prepare statement specifies `PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL`. See the Remarks section for the list of possible values for $*cursor_orientation*. See PDO::prepare for a sample using a scrollable cursor.

$*cursor_offset*: An optional (integer) symbol specifying the row to fetch when $*cursor_orientation* is either PDO::FETCH_ORI_ABS or PDO::FETCH_ORI_REL and PDO::ATTR_CURSOR is PDO::CURSOR_SCROLL.

Return Value

A mixed value that returns a row or false.

Remarks

The cursor is automatically advanced when fetch is called. The following table contains the list of possible $*fetch_style* values.

$FETCH_STYLE	**DESCRIPTION**
PDO::FETCH_ASSOC	Specifies an array indexed by column name.
PDO::FETCH_BOTH	Specifies an array indexed by column name and 0-based order. This is the default.
PDO::FETCH_BOUND	Returns true and assigns the values as specified by PDOStatement::bindColumn.
PDO::FETCH_CLASS	Creates an instance and maps columns to named properties. Call PDOStatement::setFetchMode before calling fetch.
PDO::FETCH_INTO	Refreshes an instance of the requested class. Call PDOStatement::setFetchMode before calling fetch.

$*FETCH_STYLE*	**DESCRIPTION**
PDO::FETCH_LAZY	Creates variable names during access and creates an unnamed object.
PDO::FETCH_NUM	Specifies an array indexed by zero-based column order.
PDO::FETCH_OBJ	Specifies an unnamed object with property names that map to column names.

If the cursor is at the end of the result set (the last row has been retrieved and the cursor has advanced past the result set boundary) and if the cursor is forward-only (PDO::ATTR_CURSOR = PDO::CURSOR_FWDONLY), subsequent fetch calls will fail.

If the cursor is scrollable (PDO::ATTR_CURSOR = PDO::CURSOR_SCROLL), fetch will move the cursor within the result set boundary. The following table contains the list of possible $*cursor_orientation* values.

$*CURSOR_ORIENTATION*	**DESCRIPTION**
PDO::FETCH_ORI_NEXT	Retrieves the next row. This is the default.
PDO::FETCH_ORI_PRIOR	Retrieves the previous row.
PDO::FETCH_ORI_FIRST	Retrieves the first row.
PDO::FETCH_ORI_LAST	Retrieves the last row.
PDO::FETCH_ORI_ABS, *num*	Retrieves the row requested in $*cursor_offset* by row number.
PDO::FETCH_ORI_REL, *num*	Retrieves the row requested in $*cursor_offset* by relative position from the current position.

If the value specified for $*cursor_offset* or $*cursor_orientation* results in a position outside result set boundary, fetch will fail.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```php
<?php
   $server = "(local)";
   $database = "AdventureWorks";
   $conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

   print( "\n---------- PDO::FETCH_CLASS -------------\n" );
   $stmt = $conn->query( "select * from HumanResources.Department order by GroupName" );

   class cc {
      function __construct( $arg ) {
         echo "$arg";
      }

      function __toString() {
         return $this->DepartmentID . "; " . $this->Name . "; " . $this->GroupName;
      }
   }

   $stmt->setFetchMode(PDO::FETCH_CLASS, 'cc', array( "arg1 " ));
   while ( $row = $stmt->fetch(PDO::FETCH_CLASS)) {
      print($row . "\n");
   }

   print( "\n---------- PDO::FETCH_INTO -------------\n" );
   $stmt = $conn->query( "select * from HumanResources.Department order by GroupName" );
   $c_obj = new cc( '' );

   $stmt->setFetchMode(PDO::FETCH_INTO, $c_obj);
   while ( $row = $stmt->fetch(PDO::FETCH_INTO)) {
      echo "$c_obj\n";
   }

   print( "\n---------- PDO::FETCH_ASSOC -------------\n" );
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $result = $stmt->fetch( PDO::FETCH_ASSOC );
   print_r( $result );

   print( "\n---------- PDO::FETCH_NUM -------------\n" );
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $result = $stmt->fetch( PDO::FETCH_NUM );
   print_r ($result );

   print( "\n---------- PDO::FETCH_BOTH -------------\n" );
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $result = $stmt->fetch( PDO::FETCH_BOTH );
   print_r( $result );

   print( "\n---------- PDO::FETCH_LAZY -------------\n" );
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $result = $stmt->fetch( PDO::FETCH_LAZY );
   print_r( $result );

   print( "\n---------- PDO::FETCH_OBJ -------------\n" );
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $result = $stmt->fetch( PDO::FETCH_OBJ );
   print $result->Name;
   print( "\n \n" );

   print( "\n---------- PDO::FETCH_BOUND -------------\n" );
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $stmt->bindColumn('Name', $name);
   $result = $stmt->fetch( PDO::FETCH_BOUND );
   print $name;
   print( "\n \n" );
?>
```

See Also

PDOStatement Class

PDO

PDOStatement::fetchAll

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Returns the rows in a result set in an array.

Syntax

```
array PDOStatement::fetchAll([ $fetch_style[, $column_index ][, ctor_args]] );
```

Parameters

$*fetch_style*: An (integer) symbol specifying the format of the row data. See PDOStatement::fetch for a list of values. PDO::FETCH_COLUMN is also allowed. PDO::FETCH_BOTH is the default.

$*column_index*: An integer value representing the column to return if $*fetch_style* is PDO::FETCH_COLUMN. 0 is the default.

$*ctor_args*: An array of the parameters for a class constructor, when $*fetch_style* is PDO::FETCH_CLASS or PDO::FETCH_OBJ.

Return Value

An array of the remaining rows in the result set, or false if the method call fails.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
   $server = "(local)";
   $database = "AdventureWorks";
   $conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

   print "-----------\n";
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $result = $stmt->fetchall(PDO::FETCH_BOTH);
   print_r( $result );
   print "\n-----------\n";

   print "-----------\n";
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $result = $stmt->fetchall(PDO::FETCH_NUM);
   print_r( $result );
   print "\n-----------\n";

   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $result = $stmt->fetchall(PDO::FETCH_COLUMN, 1);
   print_r( $result );
   print "\n-----------\n";

   class cc {
      function __construct( $arg ) {
         echo "$arg\n";
      }

      function __toString() {
         echo "To string\n";
      }
   };

   $stmt = $conn->query( 'SELECT TOP(2) * FROM Person.ContactType' );
   $all = $stmt->fetchAll( PDO::FETCH_CLASS, 'cc', array( 'Hi!' ));
   var_dump( $all );
?>
```

See Also

PDOStatement Class

PDO

PDOStatement::fetchColumn

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Returns one column in a row.

Syntax

```
string PDOStatement::fetchColumn ([ $column_number ] );
```

Parameters

$*column_number*: An optional integer indicating the zero-based column number. The default is 0 (the first column in the row).

Return Value

One column or false if there are no more rows.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
   $server = "(local)";
   $database = "AdventureWorks";
   $conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   while ( $result = $stmt->fetchColumn(1)) {
      print($result . "\n");
   }
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::fetchObject

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Retrieves the next row as an object.

Syntax

```
mixed PDOStatement::fetchObject([ $class_name[,$ctor_args ]] )
```

Parameters

$*class_name*: An optional string specifying the name of the class to create. The default is stdClass.

$*ctor_args*: An optional array with arguments to a custom class constructor.

Return Value

On success, returns an object with an instance of the class. Properties map to columns. Returns false on failure.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
   $server = "(local)";
   $database = "AdventureWorks";
   $conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $result = $stmt->fetchObject();
   print $result->Name;
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::getAttribute

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Retrieves the value of a predefined PDOStatement attribute or custom driver attribute.

Syntax

```
mixed PDOStatement::getAttribute( $attribute );
```

Parameters

$*attribute*: An integer, one of the PDO::ATTR_* or PDO::SQLSRV_ATTR_* constants. Supported attributes are the attributes you can set with PDOStatement::setAttribute, PDO::SQLSRV_ATTR_DIRECT_QUERY (see Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver for more information about PDO::SQLSRV_ATTR_DIRECT_QUERY), and PDO::ATTR_CURSOR.

Return Value

On success, returns a (mixed) value for a predefined PDO attribute or custom driver attribute. Returns null on failure.

Remarks

See PDOStatement::setAttribute for a sample.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

See Also

PDOStatement Class
PDO

PDOStatement::getColumnMeta

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Retrieves metadata for a column.

Syntax

```
array PDOStatement::getColumnMeta ( $column );
```

Parameters

$conn: (Integer) The zero-based number of the column whose metadata you want to retrieve.

Return Value

An associative array (key and value) containing the metadata for the column. See the Remarks section for a description of the fields in the array.

Remarks

The following table describes the fields in the array returned by getColumnMeta.

NAME	VALUES
native_type	Specifies the PHP type for column. Always string.
driver:decl_type	Specifies the SQL type used to represent the column value in the database. If the column in the result set is the result of a function, this value is not returned by PDOStatement::getColumnMeta.
flags	Specifies the flags set for this column. Always 0.
name	Specifies the name of the column in the database.
table	Specifies the name of the table that contains the column in the database. Always blank.
len	Specifies the column length.
precision	Specifies the numeric precision of this column.
pdo_type	Specifies the type of this column as represented by the PDO::PARAM_* constants. Always PDO::PARAM_STR (2).

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
$database = "AdventureWorks";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$stmt = $conn->query("select * from Person.ContactType");
$metadata = $stmt->getColumnMeta(2);
var_dump($metadata);

print $metadata['sqlsrv:decl_type'] . "\n";
print $metadata['native_type'] . "\n";
print $metadata['name'];
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::nextRowset

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Moves the cursor to the next result set.

Syntax

```
bool PDOStatement::nextRowset();
```

Return Value

true on success, false otherwise.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
   $server = "(local)";
   $database = "AdventureWorks";
   $conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

   $query1 = "select * from Person.Address where City = 'Bothell';";
   $query2 = "select * from Person.ContactType;";

   $stmt = $conn->query( $query1 . $query2 );

   $rowset1 = $stmt->fetchAll();
   $stmt->nextRowset();
   $rowset2 = $stmt->fetchAll();
   var_dump( $rowset1 );
   var_dump( $rowset2 );
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::rowCount

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Returns the number of rows added, deleted, or changed by the last statement.

Syntax

```
int PDOStatement::rowCount ();
```

Return Value

The number of rows added, deleted, or changed.

Remarks

If the last SQL statement executed by the associated PDOStatement was a SELECT statement, a PDO::CURSOR_FWDONLY cursor returns -1. A PDO::CURSOR_SCROLLABLE cursor returns the number of rows in the result set.

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

This example shows two uses of rowCount. The first use returns the number of rows that were added to the table. The second use shows that rowCount can return the number of rows in a result set when you specify a scrollable cursor.

```
<?php
$database = "Test";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$col1 = 'a';
$col2 = 'b';

$query = "insert into Table2(col1, col2) values(?, ?)";
$stmt = $conn->prepare( $query );
$stmt->execute( array( $col1, $col2 ) );
print $stmt->rowCount();

echo "\n\n";

$con = null;
$database = "AdventureWorks";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

$query = "select * from Person.ContactType";
$stmt = $conn->prepare( $query, array(PDO::ATTR_CURSOR => PDO::CURSOR_SCROLL));
$stmt->execute();
print $stmt->rowCount();
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::setAttribute

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Sets an attribute value, either a predefined PDO attribute or a custom driver attribute.

Syntax

```
bool PDOStatement::setAttribute ($attribute, $value );
```

Parameters

$*attribute*: An integer, one of the PDO::ATTR_* or PDO::SQLSRV_ATTR_* constants. See the Remarks section for the list of available attributes.

$*value*: The (mixed) value to be set for the specified $*attribute*.

Return Value

TRUE on success, FALSE otherwise.

Remarks

The following table contains the list of available attributes:

ATTRIBUTE	VALUES	DESCRIPTION
PDO::SQLSRV_ATTR_CLIENT_BUFFER_MAX_KB_SIZE	1 to the PHP memory limit.	Configures the size of the buffer that holds the result set for a client-side cursor. The default is 10240 KB (10 MB). For more information about client-side cursors, see Cursor Types (PDO_SQLSRV Driver).
PDO::SQLSRV_ATTR_ENCODING	Integer PDO::SQLSRV_ENCODING_UTF8 (Default) PDO::SQLSRV_ENCODING_SYSTEM PDO::SQLSRV_ENCODING_BINARY	Sets the character set encoding to be used by the driver to communicate with the server.

ATTRIBUTE	VALUES	DESCRIPTION
PDO::SQLSRV_ATTR_QUERY_TIMEOUT	Integer	Sets the query timeout in seconds. By default, the driver will wait indefinitely for results. Negative numbers are not allowed. 0 means no timeout.

Example

```
<?php
$database = "AdventureWorks";
$server = "(local)";
$conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "",
array('MultipleActiveResultSets'=>false )  );

$stmt = $conn->prepare('SELECT * FROM Person.ContactType');

echo $stmt->getAttribute( constant( "PDO::ATTR_CURSOR" ) );

echo "\n";

$stmt->setAttribute(PDO::SQLSRV_ATTR_QUERY_TIMEOUT, 2);
echo $stmt->getAttribute( constant( "PDO::SQLSRV_ATTR_QUERY_TIMEOUT" ) );
?>
```

See Also

PDOStatement Class
PDO

PDOStatement::setFetchMode

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

Specifies the fetch mode for the PDOStatement handle.

Syntax

```
bool PDOStatement::setFetchMode( $mode );
```

Parameters

$*mode*: Any parameter(s) that are valid to pass to PDOStatement::fetch.

Return Value

true on success, false otherwise.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.

Example

```
<?php
   $server = "(local)";
   $database = "AdventureWorks";
   $conn = new PDO( "sqlsrv:server=$server ; Database = $database", "", "");

   $stmt1 = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   while ( $row = $stmt1->fetch()) {
      print($row['Name'] . "\n");
   }
   print( "\n---------- PDO::FETCH_ASSOC -------------\n" );
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $stmt->setFetchMode(PDO::FETCH_ASSOC);
   $result = $stmt->fetch();
   print_r( $result );

   print( "\n---------- PDO::FETCH_NUM -------------\n" );
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $stmt->setFetchMode(PDO::FETCH_NUM);
   $result = $stmt->fetch();
   print_r ($result );

   print( "\n---------- PDO::FETCH_BOTH -------------\n" );
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $stmt->setFetchMode(PDO::FETCH_BOTH);
   $result = $stmt->fetch();
   print_r( $result );

   print( "\n---------- PDO::FETCH_LAZY -------------\n" );
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $stmt->setFetchMode(PDO::FETCH_LAZY);
   $result = $stmt->fetch();
   print_r( $result );

   print( "\n---------- PDO::FETCH_OBJ -------------\n" );
   $stmt = $conn->query( "select * from Person.ContactType where ContactTypeID < 5 " );
   $stmt->setFetchMode(PDO::FETCH_OBJ);
   $result = $stmt->fetch();
   print $result->Name;
   print( "\n \n" );
?>
```

See Also

PDOStatement Class
PDO

Security Considerations for PHP SQL Driver

3/14/2017 • 2 min to read • Edit on GitHub

Download PHP Driver

This topic describes security considerations that are specific to developing, deploying, and running applications that use the Microsoft Drivers for PHP for SQL Server. For more detailed information about SQL Server security, see Data security and compliance.

Connect Using Windows Authentication

Windows Authentication should be used to connect to SQL Server whenever possible for the following reasons:

- **No credentials are passed over the network during authentication.** User names and passwords are not embedded in the database connection string. This means that malicious users or attackers cannot obtain the credentials by monitoring the network or by viewing connection strings inside configuration files.
- **Users are subject to centralized account management.** Security policies such as password expiration periods, minimum password lengths, and account lockout after multiple invalid logon requests are enforced.

For information about how to connect to a server with Windows Authentication, see How to: Connect using Windows Authentication.

When you connect using Windows Authentication, it is recommended that you configure your environment so that SQL Server can use the Kerberos authentication protocol. For more information, see How to make sure that you are using Kerberos authentication when you create a remote connection to an instance of SQL Server 2005 or Kerberos Authentication and SQL Server.

Use Encrypted Connections when Transferring Sensitive Data

Encrypted connections should be used whenever sensitive data is being sent to or retrieved from SQL Server. For information about how to enable encrypted connections, see How to Enable Encrypted Connections to the Database Engine (SQL Server Configuration Manager). To establish a secure connection with the Microsoft Drivers for PHP for SQL Server, use the Encrypt connection attribute when connecting to the server. For more information about connection attributes, see Connection Options.

Use Parameterized Queries

Use parameterized queries to reduce the risk of SQL injection attacks. For examples of executing parameterized queries, see How to: Perform Parameterized Queries.

For more information about SQL injection attacks and related security considerations, see SQL Injection.

Do Not Accept Server or Connection String Information from End Users

Write applications so that end users cannot submit server or connection string information to the application. Maintaining strict control over server and connection string information reduces the surface area for malicious activity.

Turn WarningsAsErrors On During Application Development

Develop applications with the **WarningsAsErrors** setting set to **true** so that warnings issued by the driver will be treated as errors. This will allow you to address warnings before you deploy your application. For more information, see Handling Errors and Warnings.

Secure Logs for Deployed Application

For deployed applications, make sure that logs are written to a secure location or that logging is turned off. This helps protect against the possibility of end-users accessing information that has been written to the log files. For more information, see Logging Activity.

See Also

Programming Guide for PHP SQL Driver

Code Samples for PHP SQL Driver

3/14/2017 • 1 min to read • Edit on GitHub

Download PHP Driver

- Example Application (SQLSRV Driver)
- Example Application (PDO_SQLSRV Driver)

Example Application (PDO_SQLSRV Driver)

3/14/2017 • 7 min to read • Edit on GitHub

Download PHP Driver

The AdventureWorks Product Reviews example application is a Web application that uses the PDO_SQLSRV driver of the Microsoft Drivers for PHP for SQL Server. The application lets a user search for products by entering a keyword, see reviews for a selected product, write a review for a selected product, and upload an image for a selected product.

Running the Example Application

1. Install the Microsoft Drivers for PHP for SQL Server. For detailed information, see Getting Started with the PHP SQL Driver
2. Copy the code listed later in this document into two files: adventureworks_demo.php and photo.php.
3. Put the adventureworks_demo.php and photo.php files in the root directory of your Web server.
4. Run the application by starting http://localhost/adventureworks_demo.php from your browser.

Requirements

To run the AdventureWorks Product Reviews example application, the following must be true for your computer:

- Your system meets the requirements for the Microsoft Drivers for PHP for SQL Server. For detailed information, see System Requirements for the PHP SQL Driver.
  - The adventureworks_demo.php and photo.php files are in the root directory of your Web server. The files must contain the code listed later in this document.
- SQL Server 2005 or SQL Server 2008, with the AdventureWorks2008 database attached, is installed on the local computer.
- A Web browser is installed.

Demonstrates

The AdventureWorks Product Reviews example application demonstrates the following:

- How to open a connection to SQL Server by using Windows Authentication.
- How to prepare and execute a parameterized query.
- How to retrieve data.
- How to check for errors.

Example

The AdventureWorks Product Reviews example application returns product information from the database for products whose names contain a string entered by the user. From the list of returned products, the user can see reviews, see an image, upload an image, and write a review for a selected product.

Put the following code in a file named adventureworks_demo_pdo.php:

```
<!--=============
This file is part of a Microsoft SQL Server Shared Source Application.
Copyright (C) Microsoft Corporation.  All rights reserved.

THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY
```

```
KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
PARTICULAR PURPOSE.
============= *-->

<!--Note: The presentation formatting of this example application -->
<!-- is intentionally simple to emphasize the SQL Server -->
<!-- data access code.-->
<html>
<head>
<title>AdventureWorks Product Reviews</title>
</head>
<body>
<h1 align='center'>AdventureWorks Product Reviews</h1>
<h5 align='center'>This application is a demonstration of the
                   object oriented API (PDO_SQLSRV driver) for the
                   Microsoft Drivers for PHP for SQL Server.</h5><br/>
<?php
$serverName = "(local)\sqlexpress";

/* Connect using Windows Authentication. */
try
{
$conn = new PDO( "sqlsrv:server=$serverName ; Database=AdventureWorks", "", "");
$conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
}
catch(Exception $e)
{
die( print_r( $e->getMessage() ) );
}

if(isset($_REQUEST['action']))
{
switch( $_REQUEST['action'] )
{
/* Get AdventureWorks products by querying against the product name.*/
case 'getproducts':
try
{
$params = array($_POST['query']);
$tsql = "SELECT ProductID, Name, Color, Size, ListPrice
 FROM Production.Product
 WHERE Name LIKE '%' + ? + '%' AND ListPrice > 0.0";

$getProducts = $conn->prepare($tsql);
$getProducts->execute($params);
$products = $getProducts->fetchAll(PDO::FETCH_ASSOC);
$productCount = count($products);
if($productCount > 0)
{
BeginProductsTable($productCount);
foreach( $products as $row )
{
PopulateProductsTable( $row );
}
EndProductsTable();
}
else
{
DisplayNoProdutsMsg();
}
}
catch(Exception $e)
{
die( print_r( $e->getMessage() ) );
}
GetSearchTerms( !null );
break;
```

```
/* Get reviews for a specified productID. */
case 'getreview':
GetPicture( $_GET['productid'] );
GetReviews( $conn, $_GET['productid'] );
break;

/* Write a review for a specified productID. */
case 'writereview':
DisplayWriteReviewForm( $_POST['productid'] );
break;

/* Submit a review to the database. */
case 'submitreview':
try
{
$tsql = "INSERT INTO Production.ProductReview (ProductID,
   ReviewerName,
   ReviewDate,
   EmailAddress,
   Rating,
   Comments)
        VALUES (?,?,?,?,?,?)";
$params = array(&$_POST['productid'],
&$_POST['name'],
date("Y-m-d"),
&$_POST['email'],
&$_POST['rating'],
&$_POST['comments']);
$insertReview = $conn->prepare($tsql);
$insertReview->execute($params);
}
catch(Exception $e)
{
die( print_r( $e->getMessage() ) );
}
GetSearchTerms( true );
GetReviews( $conn, $_POST['productid'] );
break;

/* Display form for uploading a picture.*/
case 'displayuploadpictureform':
try
{
$tsql = "SELECT Name FROM Production.Product WHERE ProductID = ?";
$getName = $conn->prepare($tsql);
$getName->execute(array($_GET['productid']));
$name = $getName->fetchColumn(0);
}
catch(Exception $e)
{
die( print_r( $e->getMessage() ) );
}
DisplayUploadPictureForm( $_GET['productid'], $name );
break;

/* Upload a new picture for the selected product. */
case 'uploadpicture':
try
{
$tsql = "INSERT INTO Production.ProductPhoto (LargePhoto)
 VALUES (?)";
$uploadPic = $conn->prepare($tsql);
$fileStream = fopen($_FILES['file']['tmp_name'], "r");
$uploadPic->bindParam(1,
  $fileStream,
  PDO::PARAM_LOB,
  0,
  PDO::SQLSRV_ENCODING_BINARY);
$uploadPic->execute();
$uploadPic->execute();

/* Get the first field - the identity from INSERT -
   so we can associate it with the product ID. */
$photoID = $conn->lastInsertId();
$tsql = "UPDATE Production.ProductProductPhoto
 SET ProductPhotoID = ?
 WHERE ProductID = ?";
$associateIds = $conn->prepare($tsql);
$associateIds->execute(array($photoID, $_POST['productid']));
}
catch(Exception $e)
{
die(print_r($e->getMessage()));
}

GetPicture( $_POST['productid']);
DisplayWriteReviewButton( $_POST['productid'] );
GetSearchTerms (!null);
break;
}//End Switch
}
else
{
    GetSearchTerms( !null );
}

function GetPicture( $productID )
{
    echo "<table align='center'><tr align='center'><td>";
    echo "<img src='photo_pdo.php?productId=".$productID."'
      height='150' width='150'/></td></tr>";
    echo "<tr align='center'><td><a href='?action=displayuploadpictureform&productid=".$productID."'>Upload
new picture.</a></td></tr>";
    echo "</td></tr></table></br>";
}

function GetReviews( $conn, $productID )
{
try
{
$tsql = "SELECT ReviewerName,
CONVERT(varchar(32),
ReviewDate, 107) AS [ReviewDate],
Rating,
Comments
 FROM Production.ProductReview
 WHERE ProductID = ?
 ORDER BY ReviewDate DESC";
$getReviews = $conn->prepare( $tsql);
$getReviews->execute(array($productID));
$reviews = $getReviews->fetchAll(PDO::FETCH_NUM);
$reviewCount = count($reviews);
if($reviewCount > 0 )
{
foreach($reviews as $row)
{
$name = $row[0];
$date = $row[1];
$rating = $row[2];
$comments = $row[3];
DisplayReview( $productID, $name, $date, $rating, $comments );
}
}
else
{
DisplayNoReviewsMsg();
}
}
catch(Exception $e)
catch(Exception $e)
{
die(print_r($e->getMessage()));
}
    DisplayWriteReviewButton( $productID );
GetSearchTerms(!null);
}

/*** Presentation and Utility Functions ***/

function BeginProductsTable($rowCount)
{
    /* Display the beginning of the search results table. */
$headings = array("Product ID", "Product Name", "Color", "Size", "Price");
echo "<table align='center' cellpadding='5'>";
echo "<tr bgcolor='silver'>$rowCount Results</tr><tr>";
foreach ( $headings as $heading )
{
echo "<td>$heading</td>";
}
echo "</tr>";
}

function DisplayNoProdutsMsg()
{
    echo "<h4 align='center'>No products found.</h4>";
}

function DisplayNoReviewsMsg()
{
    echo "<h4 align='center'>There are no reviews for this product.</h4>";
}

function DisplayReview( $productID, $name, $date, $rating, $comments)
{
    /* Display a product review. */
    echo "<table style='WORD-BREAK:BREAK-ALL' width='50%' align='center' border='1' cellpadding='5'>";
    echo "<tr>
            <td>ProductID</td>
            <td>Reviewer</td>
            <td>Date</td>
            <td>Rating</td>
          </tr>";
      echo "<tr>
              <td>$productID</td>
              <td>$name</td>
              <td>$date</td>
              <td>$rating</td>
            </tr>
            <tr>
              <td width='50%' colspan='4'>$comments</td></tr></table><br/><br/>";
}

function DisplayUploadPictureForm( $productID, $name )
{
    echo "<h3 align='center'>Upload Picture</h3>";
    echo "<h4 align='center'>$name</h4>";
    echo "<form align='center' action='adventureworks_demo_pdo.php'
enctype='multipart/form-data' method='POST'>
<input type='hidden' name='action' value='uploadpicture'/>
<input type='hidden' name='productid' value='$productID'/>
<table align='center'>
 <tr>
   <td align='center'>
 <input id='fileName' type='file' name='file'/>
   </td>
 </tr>
 <tr>
   <td align='center'>
```

```
<input type='submit' name='submit' value='Upload Picture'/>
   </td>
 </tr>
</table>
  </form>";
}

function DisplayWriteReviewButton( $productID )
{
    echo "<table align='center'><form action='adventureworks_demo_pdo.php'
             enctype='multipart/form-data' method='POST'>
          <input type='hidden' name='action' value='writereview'/>
          <input type='hidden' name='productid' value='$productID'/>
          <input type='submit' name='submit' value='Write a Review'/>
          </p></td></tr></form></table>";
}

function DisplayWriteReviewForm( $productID )
{
    /* Display the form for entering a product review. */
    echo "<h5 align='center'>Name, E-mail, and Rating are required fields.</h5>";
    echo "<table align='center'>
<form action='adventureworks_demo_pdo.php'
  enctype='multipart/form-data' method='POST'>
<input type='hidden' name='action' value='submitreview'/>
<input type='hidden' name='productid' value='$productID'/>
<tr>
<td colspan='5'>Name: <input type='text' name='name' size='50'/></td>
</tr>
<tr>
<td colspan='5'>E-mail: <input type='text' name='email' size='50'/></td>
</tr>
<tr>
<td>Rating: 1<input type='radio' name='rating' value='1'/></td>
<td>2<input type='radio' name='rating' value='2'/></td>
<td>3<input type='radio' name='rating' value='3'/></td>
<td>4<input type='radio' name='rating' value='4'/></td>
<td>5<input type='radio' name='rating' value='5'/></td>
</tr>
<tr>
<td colspan='5'>
<textarea rows='20' cols ='50' name='comments'>[Write comments here.]</textarea>
</td>
</tr>
<tr>
<td colspan='5'>
<p align='center'><input type='submit' name='submit' value='Submit Review'/>
</td>
</tr>
</form>
          </table>";
}

function EndProductsTable()
{
    echo "</table><br/>";
}

function GetSearchTerms( $success )
{
    /* Get and submit terms for searching the database. */
    if (is_null( $success ))
    {
echo "<h4 align='center'>Review successfully submitted.</h4>";}
echo "<h4 align='center'>Enter search terms to find products.</h4>";
echo "<table align='center'>
<form action='adventureworks_demo_pdo.php'
  enctype='multipart/form-data' method='POST'>
<input type='hidden' name='action' value='getproducts'/>
```

```
<tr>
   <td><input type='text' name='query' size='40'/></td>
</tr>
<tr align='center'>
   <td><input type='submit' name='submit' value='Search'/></td>
</tr>
</form>
  </table>";
}

function PopulateProductsTable( $values )
{
    /* Populate Products table with search results. */
    $productID = $values['ProductID'];
    echo "<tr>";
    foreach ( $values as $key => $value )
    {
        if ( 0 == strcasecmp( "Name", $key ) )
        {
            echo "<td><a href='?action=getreview&productid=$productID'>$value</a></td>";
        }
        elseif( !is_null( $value ) )
        {
            if ( 0 == strcasecmp( "ListPrice", $key ) )
            {
                /* Format with two digits of precision. */
                $formattedPrice = sprintf("%.2f", $value);
                echo "<td>$$formattedPrice</td>";
            }
            else
            {
                echo "<td>$value</td>";
            }
        }
        else
        {
            echo "<td>N/A</td>";
        }
    }
    echo "<td>
            <form action='adventureworks_demo_pdo.php' enctype='multipart/form-data' method='POST'>
            <input type='hidden' name='action' value='writereview'/>
            <input type='hidden' name='productid' value='$productID'/>
            <input type='submit' name='submit' value='Write a Review'/>
            </td></tr>
            </form></td></tr>";
}
?>
</body>
</html>
```

Example

The photo.php script returns a product photo for a specified **ProductID**. This script is called from the adventureworks_demo.php script.

Put the following code in a file named photo_pdo.php:

```
<?php
/*
=============
This file is part of a Microsoft SQL Server Shared Source Application.
Copyright (C) Microsoft Corporation.  All rights reserved.

THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY
KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
PARTICULAR PURPOSE.
=============
*/
$serverName = "(local)\sqlexpress";

/* Connect using Windows Authentication. */
try
{
$conn = new PDO( "sqlsrv:server=$serverName ; Database=AdventureWorks", "", "");
$conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
}
catch(Exception $e)
{
die( print_r( $e->getMessage() ) );
}

/* Get the product picture for a given product ID. */
try
{
$tsql = "SELECT LargePhoto
 FROM Production.ProductPhoto AS p
 JOIN Production.ProductProductPhoto AS q
 ON p.ProductPhotoID = q.ProductPhotoID
 WHERE ProductID = ?";
$stmt = $conn->prepare($tsql);
$stmt->execute(array(&$_GET['productId']));
$stmt->bindColumn(1, $image, PDO::PARAM_LOB, 0, PDO::SQLSRV_ENCODING_BINARY);
$stmt->fetch(PDO::FETCH_BOUND);
echo $image;
}
catch(Exception $e)
{
die( print_r( $e->getMessage() ) );
}
?>
```

See Also

Connecting to the Server
Comparing Execution Functions
Retrieving Data
Updating Data (Microsoft Drivers for PHP for SQL Server)
SQLSRV Driver API Reference

Example Application (SQLSRV Driver)

3/14/2017 • 9 min to read • Edit on GitHub

Download PHP Driver

The AdventureWorks Product Reviews example application is a Web application that uses the SQLSRV driver of Microsoft Drivers for PHP for SQL Server. The application lets a user search for products by entering a keyword, see reviews for a selected product, write a review for a selected product, and upload an image for a selected product.

Running the Example Application

1. Install the Microsoft Drivers for PHP for SQL Server. For detailed information, see Getting Started with the PHP SQL Driver.
2. Copy the code listed later in this document into two files: adventureworks_demo.php and photo.php.
3. Put the adventureworks_demo.php and photo.php files in the root directory of your Web server.
4. Run the application by starting http://localhost/adventureworks_demo.php from your browser.

Requirements

To run the AdventureWorks Product Reviews example application, the following must be true for your computer:

- Your system meets the requirements for the Microsoft Drivers for PHP for SQL Server. For detailed information, see System Requirements for the PHP SQL Driver.
- The adventureworks_demo.php and photo.php files are in the root directory of your Web server. The files must contain the code listed later in this document.
- SQL Server 2005 or SQL Server 2008, with the AdventureWorks2008 database attached, is installed on the local computer.
- A Web browser is installed.

Demonstrates

The AdventureWorks Product Reviews example application demonstrates the following:

- How to open a connection to SQL Server by using Windows Authentication.
- How to execute a parameterized query with sqlsrv_query.
- How to prepare and execute a parameterized query by using the combination of sqlsrv_prepare and sqlsrv_execute.
- How to retrieve data by using sqlsrv_fetch_array.
- How to retrieve data by using the combination of sqlsrv_fetch and sqlsrv_get_field.
- How to retrieve data as a stream.
- How to send data as a stream.
- How to check for errors.

Example

The AdventureWorks Product Reviews example application returns product information from the database for products whose names contain a string entered by the user. From the list of returned products, the user can see reviews, see an image, upload an image, and write a review for a selected product.

Put the following code in a file named adventureworks_demo.php:

```
<!--=============
This file is part of a Microsoft SQL Server Shared Source Application.
Copyright (C) Microsoft Corporation.  All rights reserved.

THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY
KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
PARTICULAR PURPOSE.
============= *-->

<!--Note: The presentation formatting of the example application -->
<!-- is intentionally simple to emphasize the SQL Server -->
<!-- data access code.-->
<html>
<head>
<title>AdventureWorks Product Reviews</title>
</head>
<body>
<h1 align='center'>AdventureWorks Product Reviews</h1>
<h5 align='center'>This application is a demonstration of the
                   procedural API (SQLSRV driver) of the Microsoft
                   Drivers for PHP for SQL Server.</h5><br/>
<?php
$serverName = "(local)\sqlexpress";
$connectionOptions = array("Database"=>"AdventureWorks");

/* Connect using Windows Authentication. */
$conn = sqlsrv_connect( $serverName, $connectionOptions);
if( $conn === false )
die( FormatErrors( sqlsrv_errors() ) );

if(isset($_REQUEST['action']))
{
switch( $_REQUEST['action'] )
{
/* Get AdventureWorks products by querying
   against the product name.*/
case 'getproducts':
$params = array(&$_POST['query']);
$tsql = "SELECT ProductID, Name, Color, Size, ListPrice
FROM Production.Product
WHERE Name LIKE '%' + ? + '%' AND ListPrice > 0.0";
/*Execute the query with a scrollable cursor so
  we can determine the number of rows returned.*/
$cursorType = array("Scrollable" => SQLSRV_CURSOR_KEYSET);
$getProducts = sqlsrv_query($conn, $tsql, $params, $cursorType);
if ( $getProducts === false)
die( FormatErrors( sqlsrv_errors() ) );

if(sqlsrv_has_rows($getProducts))
{
$rowCount = sqlsrv_num_rows($getProducts);
BeginProductsTable($rowCount);
while( $row = sqlsrv_fetch_array( $getProducts, SQLSRV_FETCH_ASSOC))
{
PopulateProductsTable( $row );
}
EndProductsTable();
}
else
{
DisplayNoProdutsMsg();
}
GetSearchTerms( !null );

/* Free the statement and connection resources. */
```

```
sqlsrv_free_stmt( $getProducts );
sqlsrv_close( $conn );
break;

/* Get reviews for a specified productID. */
case 'getreview':
GetPicture( $_GET['productid'] );
GetReviews( $conn, $_GET['productid'] );
sqlsrv_close( $conn );
break;

/* Write a review for a specified productID. */
case 'writereview':
DisplayWriteReviewForm( $_POST['productid'] );
break;

/* Submit a review to the database. */
case 'submitreview':
/*Prepend the review so it can be opened as a stream.*/
$comments = "data://text/plain,".$_POST['comments'];
$stream = fopen( $comments, "r" );
$tsql = "INSERT INTO Production.ProductReview (ProductID,
   ReviewerName,
   ReviewDate,
   EmailAddress,
   Rating,
   Comments)
 VALUES (?,?,?,?,?,?)";
$params = array(&$_POST['productid'],
&$_POST['name'],
date("Y-m-d"),
&$_POST['email'],
&$_POST['rating'],
&$stream);

/* Prepare and execute the statement. */
$insertReview = sqlsrv_prepare($conn, $tsql, $params);
if( $insertReview === false )
die( FormatErrors( sqlsrv_errors() ) );
/* By default, all stream data is sent at the time of
query execution. */
if( sqlsrv_execute($insertReview) === false )
die( FormatErrors( sqlsrv_errors() ) );
sqlsrv_free_stmt( $insertReview );
GetSearchTerms( true );

/* Display a list of reviews, including the latest addition. */
GetReviews( $conn, $_POST['productid'] );
sqlsrv_close( $conn );
break;

        /* Display a picture of the selected product.*/
        case 'displaypicture':
            $tsql = "SELECT Name
                     FROM Production.Product
                     WHERE ProductID = ?";
            $getName = sqlsrv_query($conn, $tsql,
                                    array(&$_GET['productid']));
            if( $getName === false )
die( FormatErrors( sqlsrv_errors() ) );
            if ( sqlsrv_fetch( $getName ) === false )
die( FormatErrors( sqlsrv_errors() ) );
            $name = sqlsrv_get_field( $getName, 0);
            DisplayUploadPictureForm( $_GET['productid'], $name );
            sqlsrv_close( $conn );
            break;

        /* Upload a new picture for the selected product. */
        case 'uploadpicture':
            $tsql = "INSERT INTO Production.ProductPhoto (LargePhoto)
                     VALUES (?); SELECT SCOPE_IDENTITY() AS PhotoID";
            $fileStream = fopen($_FILES['file']['tmp_name'], "r");
            $uploadPic = sqlsrv_prepare($conn, $tsql, array(
                       array(&$fileStream,
                             SQLSRV_PARAM_IN,
                             SQLSRV_PHPTYPE_STREAM(SQLSRV_ENC_BINARY),
                             SQLSRV_SQLTYPE_VARBINARY('max'))));
            if( $uploadPic === false )
die( FormatErrors( sqlsrv_errors() ) );
            if( sqlsrv_execute($uploadPic) === false )
die( FormatErrors( sqlsrv_errors() ) );

/*Skip the open result set (row affected). */
$next_result = sqlsrv_next_result($uploadPic);
if( $next_result === false )
die( FormatErrors( sqlsrv_errors() ) );

/* Fetch the next result set. */
if( sqlsrv_fetch($uploadPic) === false)
die( FormatErrors( sqlsrv_errors() ) );

/* Get the first field - the identity from INSERT. */
$photoID = sqlsrv_get_field($uploadPic, 0);

/* Associate the new photoID with the productID. */
$tsql = "UPDATE Production.ProductProductPhoto
 SET ProductPhotoID = ?
 WHERE ProductID = ?";

$reslt = sqlsrv_query($conn, $tsql, array(&$photoID, &$_POST['productid']));
if($reslt === false )
die( FormatErrors( sqlsrv_errors() ) );

GetPicture( $_POST['productid']);
DisplayWriteReviewButton( $_POST['productid'] );
GetSearchTerms (!null);
sqlsrv_close( $conn );
break;
}//End Switch
}
else
{
    GetSearchTerms( !null );
}

function GetPicture( $productID )
{
    echo "<table align='center'><tr align='center'><td>";
    echo "<img src='photo.php?productId=".$productID."'
      height='150' width='150'/></td></tr>";
    echo "<tr align='center'><td><a href='?action=displaypicture&
          productid=".$productID."'>Upload new picture.</a></td></tr>";
    echo "</td></tr></table></br>";
}

function GetReviews( $conn, $productID )
{
    $tsql = "SELECT ReviewerName,
             CONVERT(varchar(32), ReviewDate, 107) AS [ReviewDate],
 Rating,
 Comments
             FROM Production.ProductReview
             WHERE ProductID = ?
             ORDER BY ReviewDate DESC";
/*Execute the query with a scrollable cursor so
  we can determine the number of rows returned.*/
$cursorType = array("Scrollable" => SQLSRV_CURSOR_KEYSET);
$getReviews = sqlsrv_query( $conn, $tsql, array(&$productID), $cursorType);
```

```
$getReviews = sqlsrv_query( $conn, $tsql, array(&$productID), $cursorType);
if( $getReviews === false )
die( FormatErrors( sqlsrv_errors() ) );
if(sqlsrv_has_rows($getReviews))
{
$rowCount = sqlsrv_num_rows($getReviews);
echo "<table width='50%' align='center' border='1px'>";
echo "<tr bgcolor='silver'><td>$rowCount Reviews</td></tr></table>";
while ( sqlsrv_fetch( $getReviews ) )
{
$name = sqlsrv_get_field( $getReviews, 0 );
$date = sqlsrv_get_field( $getReviews, 1 );
$rating = sqlsrv_get_field( $getReviews, 2 );
/* Open comments as a stream. */
$comments = sqlsrv_get_field( $getReviews, 3,
SQLSRV_PHPTYPE_STREAM(SQLSRV_ENC_CHAR));
DisplayReview($productID,
  $name,
              $date,
              $rating,
              $comments );
}
}
    else
    {
DisplayNoReviewsMsg();
}
    DisplayWriteReviewButton( $productID );
    sqlsrv_free_stmt( $getReviews );
}

/*** Presentation and Utility Functions ***/

function BeginProductsTable($rowCount)
{
    /* Display the beginning of the search results table. */
$headings = array("Product ID", "Product Name",
"Color", "Size", "Price");
    echo "<table align='center' cellpadding='5'>";
    echo "<tr bgcolor='silver'>$rowCount Results</tr><tr>";
    foreach ( $headings as $heading )
    {
        echo "<td>$heading</td>";
    }
    echo "</tr>";
}

function DisplayNoProdutsMsg()
{
    echo "<h4 align='center'>No products found.</h4>";
}

function DisplayNoReviewsMsg()
{
    echo "<h4 align='center'>There are no reviews for this product.</h4>";
}

function DisplayReview( $productID, $name, $date, $rating, $comments)
{
    /* Display a product review. */
    echo "<table style='WORD-BREAK:BREAK-ALL' width='50%'
                 align='center' border='1' cellpadding='5'>";
    echo "<tr>
            <td>ProductID</td>
            <td>Reviewer</td>
            <td>Date</td>
            <td>Rating</td>
          </tr>";
      echo "<tr>
```

```
                <td>$productID</td>
                <td>$name</td>
                <td>$date</td>
                <td>$rating</td>
              </tr>
              <tr>
                <td width='50%' colspan='4'>";
                   fpassthru( $comments );
       echo "</td></tr></table><br/><br/>";
}

function DisplayUploadPictureForm( $productID, $name )
{
    echo "<h3 align='center'>Upload Picture</h3>";
    echo "<h4 align='center'>$name</h4>";
    echo "<form align='center' action='adventureworks_demo.php'
                    enctype='multipart/form-data' method='POST'>
<input type='hidden' name='action' value='uploadpicture'/>
<input type='hidden' name='productid' value='$productID'/>
<table align='center'>
         <tr>
           <td align='center'>
             <input id='fileName' type='file' name='file'/>
           </td>
         </tr>
         <tr>
           <td align='center'>
            <input type='submit' name='submit' value='Upload Picture'/>
           </td>
         </tr>
</table>
</form>";
}

function DisplayWriteReviewButton( $productID )
{
    echo "<table align='center'><form action='adventureworks_demo.php'
                 enctype='multipart/form-data' method='POST'>
          <input type='hidden' name='action' value='writereview'/>
          <input type='hidden' name='productid' value='$productID'/>
          <input type='submit' name='submit' value='Write a Review'/>
          </p></td></tr></form></table>";
}

function DisplayWriteReviewForm( $productID )
{
    /* Display the form for entering a product review. */
    echo "<h5 align='center'>Name, E-mail, and Rating are required fields.</h5>";
    echo "<table align='center'>
<form action='adventureworks_demo.php'
                enctype='multipart/form-data' method='POST'>
<input type='hidden' name='action' value='submitreview'/>
<input type='hidden' name='productid' value='$productID'/>
<tr>
<td colspan='5'>Name: <input type='text' name='name' size='50'/></td>
</tr>
<tr>
<td colspan='5'>E-mail: <input type='text' name='email' size='50'/></td>
</tr>
<tr>
<td>Rating: 1<input type='radio' name='rating' value='1'/></td>
<td>2<input type='radio' name='rating' value='2'/></td>
<td>3<input type='radio' name='rating' value='3'/></td>
<td>4<input type='radio' name='rating' value='4'/></td>
<td>5<input type='radio' name='rating' value='5'/></td>
</tr>
<tr>
<td colspan='5'>
<textarea rows='20' cols ='50' name='comments'>[Write comments here.]</textarea>
```

```
</td>
</tr>
<tr>
<td colspan='5'>
                 <p align='center'><input type='submit' name='submit' value='Submit Review'/>
</td>
</tr>
</form>
          </table>";
}

function EndProductsTable()
{
    echo "</table><br/>";
}

function GetSearchTerms( $success )
{
    /* Get and submit terms for searching the database. */
    if (is_null( $success ))
    {
echo "<h4 align='center'>Review successfully submitted.</h4>";}
echo "<h4 align='center'>Enter search terms to find products.</h4>";
echo "<table align='center'>
            <form action='adventureworks_demo.php'
                  enctype='multipart/form-data' method='POST'>
            <input type='hidden' name='action' value='getproducts'/>
            <tr>
               <td><input type='text' name='query' size='40'/></td>
            </tr>
            <tr align='center'>
               <td><input type='submit' name='submit' value='Search'/></td>
            </tr>
            </form>
            </table>";
}

function PopulateProductsTable( $values )
{
    /* Populate Products table with search results. */
    $productID = $values['ProductID'];
    echo "<tr>";
    foreach ( $values as $key => $value )
    {
        if ( 0 == strcasecmp( "Name", $key ) )
        {
            echo "<td><a href='?action=getreview&productid=$productID'>$value</a></td>";
        }
        elseif( !is_null( $value ) )
        {
            if ( 0 == strcasecmp( "ListPrice", $key ) )
            {
                /* Format with two digits of precision. */
                $formattedPrice = sprintf("%.2f", $value);
                echo "<td>$$formattedPrice</td>";
            }
            else
            {
                echo "<td>$value</td>";
            }
        }
        else
        {
            echo "<td>N/A</td>";
        }
    }
    echo "<td>
            <form action='adventureworks_demo.php'
                  enctype='multipart/form-data' method='POST'>
            <input type='hidden' name='action' value='writereview'/>
            <input type='hidden' name='productid' value='$productID'/>
            <input type='submit' name='submit' value='Write a Review'/>
            </td></tr>
            </form></td></tr>";
}

function FormatErrors( $errors )
{
    /* Display errors. */
    echo "Error information: <br/>";

    foreach ( $errors as $error )
    {
        echo "SQLSTATE: ".$error['SQLSTATE']."<br/>";
        echo "Code: ".$error['code']."<br/>";
        echo "Message: ".$error['message']."<br/>";
    }
}
?>
</body>
</html>
```

Example

The photo.php script returns a product photo for a specified **ProductID**. This script is called from the adventureworks_demo.php script.

Put the following code in a file named photo.php:

```
<?php
/*=============
This file is part of a Microsoft SQL Server Shared Source Application.
Copyright (C) Microsoft Corporation.  All rights reserved.

THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY
KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
PARTICULAR PURPOSE.
============= */

$serverName = "(local)\sqlexpress";
$connectionInfo = array( "Database"=>"AdventureWorks");

/* Connect using Windows Authentication. */
$conn = sqlsrv_connect( $serverName, $connectionInfo);
if( $conn === false )
{
     echo "Could not connect.\n";
     die( print_r( sqlsrv_errors(), true));
}

/* Get the product picture for a given product ID. */
$tsql = "SELECT LargePhoto
         FROM Production.ProductPhoto AS p
         JOIN Production.ProductProductPhoto AS q
         ON p.ProductPhotoID = q.ProductPhotoID
         WHERE ProductID = ?";

$params = array(&$_REQUEST['productId']);

/* Execute the query. */
$stmt = sqlsrv_query($conn, $tsql, $params);
if( $stmt === false )
{
     echo "Error in statement execution.</br>";
     die( print_r( sqlsrv_errors(), true));
}

/* Retrieve the image as a binary stream. */
$getAsType = SQLSRV_PHPTYPE_STREAM(SQLSRV_ENC_BINARY);
if ( sqlsrv_fetch( $stmt ) )
{
   $image = sqlsrv_get_field( $stmt, 0, $getAsType);
   fpassthru($image);
}
else
{
     echo "Error in retrieving data.</br>";
     die(print_r( sqlsrv_errors(), true));
}

/* Free the statement and connectin resources. */
sqlsrv_free_stmt( $stmt );
sqlsrv_close( $conn );
?>
```

See Also

Connecting to the Server
Comparing Execution Functions
Retrieving Data
Updating Data (Microsoft Drivers for PHP for SQL Server)
SQLSRV Driver API Reference