Contents

Code Samples

Example Application (PDO_SQLSRV Driver)

Example Application (SQLSRV Driver)

Microsoft Drivers for PHP for SQL Server

10/1/2018 • 2 minutes to read • <u>Edit Online</u>

Download PHP Driver

The Microsoft Drivers for PHP for SQL Server enable integration with SQL Server for PHP applications. The drivers are PHP extensions that allow the reading and writing of SQL Server data from within PHP scripts. The drivers provide interfaces for accessing data in Azure SQL Databases and in all editions of SQL Server 2005 and later (including Express Editions). The drivers make 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 Microsoft Drivers for PHP for SQL Server

Download

- To download drivers for PHP for SQL

Samples

- Code Samples for the Microsoft Drivers for PHP for SQL Server
- Getting Started with PHP on Windows
- Getting Started with PHP on macOS
- Getting Started with PHP on Ubuntu
- Getting Started with PHP on Red Hat Enterprise Linux (RHEL)
- Getting Started with PHP on SUSE Linux Enterprise Server (SLES)

Getting Started with the Microsoft Drivers for PHP for SQL Server

10/1/2018 • 2 minutes to read • Edit Online

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 the Microsoft Drivers for PHP for SQL Server

SQLSRV Driver API Reference

Step 1: Configure environment for PHP development

10/1/2018 • 2 minutes to read • Edit Online

⊕ Download PHP Driver

- Identify which version of the PHP driver you will use, based on your environment, as noted here: System Requirements for the Microsoft Drivers for PHP for SQL Server
- 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 Microsoft Drivers for PHP for SQL Server
- Configure IIS to host PHP applications, as noted here: Configuring IIS for the Microsoft Drivers for PHP for SQL Server

Linux and Mac

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

Step 2: Create a SQL database for PHP development

10/1/2018 • 2 minutes to read • <ins>Edit Online</ins>

<ins>Download PHP Driver</ins>

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 Samples on GitHub

Step 3: Proof of concept connecting to SQL using PHP

10/1/2018 • 2 minutes to read • Edit Online

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 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

10/1/2018 • 2 minutes to read • Edit Online

⊕Download PHP Driver

The demo program is designed so that a transient error (that is any error code with the prefix '08' as listed in this appendix) 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 sqlsrv_query() function can be used to retrieve a result set from a query against SQL Database. This function essentially accepts any query and connection object and returns a result set, which can be iterated over with the use of sqlsrv_fetch_array().

```php
<?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 = null;
    $arrayOfTransientErrors = array('08001', '08002', '08003', '08004', '08007', '08S01');
    for ($cc = 1; $cc <= $maxCountTriesConnectAndQuery; $cc++) {
        // [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 Name FROM Production.ProductCategory";
            $stmt = sqlsrv_query($conn, $tsql);
            if ($stmt === false) {
                echo "Error in query execution";
                echo "<br>";
                die(print_r(sqlsrv_errors(), true));
            }
            while($row = sqlsrv_fetch_array($stmt, SQLSRV_FETCH_ASSOC)) {
                echo $row['Name'] . "<br/>" ;
            }
            sqlsrv_free_stmt($stmt);
            sqlsrv_close( $conn);
            break;
        } else {
            // [A.4] Check whether the error code is on the whitelist of transients.
            $isTransientError = false;
            $errorCode = '';
            if (($errors = sqlsrv_errors()) != null) {
                foreach ($errors as $error) {
                    $errorCode = $error['code'];
                    $isTransientError = in_array($errorCode, $arrayOfTransientErrors);
                    if ($isTransientError) {
                        break;
                    }
                }
```

```php
                    }
                }
                if (!$isTransientError) {
                    // it is a static persistent error...
                    echo("Persistent error suffered with error code = $errorCode. Program will terminate.");
                    echo "<br>";
                    // [A.5] Either the connection attempt or the query command attempt suffered a persistent
error condition.
                    // Break the loop, let the hopeless program end.
                    exit(0);
                }
                // [A.6] It is a transient error from an attempt to issue a query command.
                // So let this method reloop and try again. However, we recommend that the new query
                // attempt should start at the beginning and establish a new connection.
                if ($cc >= $maxCountTriesConnectAndQuery) {
                    echo "Transient errors suffered in too many retries - $cc. Program will terminate.";
                    echo "<br>";
                    exit(0);
                }
                echo("Transient error encountered with error code = $errorCode. Program might retry by
itself.");
                echo "<br>";
                echo "$cc attempts so far. Might retry.";
                echo "<br>";
                // A very simple retry strategy, a brief pause before looping.
                sleep(1*$secondsBetweenRetries);
            }
            // [A.3] All has gone well, so let the program end.
        }
    ?>
```

Overview of the Microsoft Drivers for PHP for SQL Server

10/1/2018 • 2 minutes to read • Edit Online

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 with the SQLSRV driver and an object-oriented interface with the PDO_SQLSRV driver for accessing data in all versions of SQL Server, including Express, beginning with SQL Server 2005. Support for versions 3.1 and later of the drivers 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 Microsoft Drivers for PHP for SQL Server.

In This Section

TOPIC	DESCRIPTION
To download drivers for PHP for SQL Server	Links to download Microsoft Drivers for PHP for SQL Server.
Release Notes for the Microsoft Drivers for PHP for SQL Server	Lists the features that were added for versions 4.0, 3.2, 3.1, 3.0, and 2.0.
Support Resources for the Microsoft Drivers for PHP for SQL Server	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 Microsoft Drivers for PHP for SQL Server

Programming Guide for the Microsoft Drivers for PHP for SQL Server

Example Application (SQLSRV Driver)

System Requirements for the Microsoft Drivers for PHP for SQL Server

10/19/2018 • 7 minutes to read • Edit Online

⊕Download PHP Driver

This document lists the components that must be installed on your system to access data in a SQL Server or Azure SQL Database using the Microsoft Drivers for PHP for SQL Server.

Versions 3.1 and later of the Microsoft PHP Drivers for SQL Server are officially supported. For full details on support lifecycles and requirements including earlier versions of the PHP drivers, see the support matrix.

PHP

For information about how to download and install the latest stable PHP binaries, see the PHP web site. The Microsoft Drivers for PHP for SQL Server require the following versions of PHP:

PHP FOR SQL SERVER DRIVER VERSION → ↓ PHP VERSION	5.3 AND 5.2	4.3	4.0	3.2	3.1
7.2	7.2.1+ on Windows 7.2.0+ on other platforms				
7.1	7.1.0+	7.1.0+			
7.0	7.0.0+	7.0.0+	7.0.0+		
5.6				5.6.4+	
5.5				5.5.16+	5.5.16+
5.4				5.4.32	5.4.32

- A version of the driver file must be in your PHP extension directory. See Driver Versions for information about the different driver files. To download the drivers, see Download the Microsoft Drivers for PHP for SQL Server. For information on configuring the driver for the PHP, see Loading the 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 IIS, see the tutorial on PHP's web site.

  The Microsoft Drivers for PHP for SQL Server has been tested using IIS 10 with FastCGI.

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

- Version 5.3 of the Microsoft Drivers for PHP for SQL Server will be the last to support PHP 7.0.

ODBC Driver

The correct version of the Microsoft ODBC Driver for SQL Server is required on the computer on which PHP is running. You can download all supported versions of the driver for supported platforms on this page.

If you are downloading the Windows version of the driver on a 64-bit version of Windows, the ODBC 64-bit installer installs both 32-bit and 64-bit ODBC drivers. If you use a 32-bit version of Windows, use the ODBC x86 installer. On non-Windows platforms, only 64-bit versions of the driver are available.

PHP FOR SQL SERVER DRIVER VERSION → ↓ ODBC DRIVER VERSION	5.3	5.2	4.3	4.0	3.2	3.1
ODBC Driver 17+	Y	Y				
ODBC Driver 13.1	Y	Y	Y	Y		
ODBC Driver 13				Y		
ODBC Driver 11	Y	Y	Y	Y	Y	Y

If you are using the SQLSRV driver, sqlsrv_client_info returns information about which version of SQL Server 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.

SQL Server

Azure SQL Databases are supported. For information, see Connecting to Microsoft Azure SQL Database.

PHP FOR SQL SERVER DRIVER VERSION → ↓ SQL SERVER VERSION	5.3	5.2	4.3	4.0	3.2	3.1
Azure SQL Database	Y	Y	Y			
Azure SQL Managed Instance	Y	Y	Y			
Azure SQL Data Warehouse	Y	Y	Y			
SQL Server 2017	Y	Y	Y			

PHP FOR SQL SERVER DRIVER VERSION → ↓ SQL SERVER VERSION	5.3	5.2	4.3	4.0	3.2	3.1
SQL Server 2016	Y	Y	Y	Y		
SQL Server 2014	Y	Y	Y	Y	Y	Y
SQL Server 2012	Y	Y	Y	Y	Y	Y
SQL Server 2008 R2	Y	Y	Y	Y	Y	Y
SQL Server 2008				Y	Y	Y

Operating Systems

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

PHP FOR SQL SERVER DRIVER VERSION → ↓ OPERATING SYSTEM	5.3	5.2	4.3	4.0	3.2	3.1
Windows Server 2016	Y	Y	Y			
Windows Server 2012 R2	Y	Y	Y	Y	Y	Y
Windows Server 2012	Y	Y	Y	Y	Y	Y
Windows Server 2008 R2 SP1				Y	Y	Y
Windows Server 2008 SP2				Y	Y	Y
Windows 10	Y	Y	Y	Y		
Windows 8.1	Y	Y	Y	Y	Y	Y
Windows 8			Y	Y	Y	Y

PHP FOR SQL SERVER DRIVER VERSION → ↓ OPERATING SYSTEM	5.3	5.2	4.3	4.0	3.2	3.1
Windows 7 SP1				Y	Y	Y
Windows Vista SP2				Y	Y	Y
Ubuntu 18.04 (64-bit)	Y					
Ubuntu 17.10 (64-bit)	Y	Y				
Ubuntu 16.04 (64-bit)	Y	Y	Y	Y		
Ubuntu 15.10 (64-bit)			Y			
Ubuntu 15.04 (64-bit)				Y		
Debian 9 (64-bit)	Y	Y				
Debian 8 (64-bit)	Y	Y	Y			
Red Hat Enterprise Linux 7 (64-bit)	Y	Y	Y	Y		
Suse Enterprise Linux 12 (64-bit)	Y	Y				
macOS High Sierra (64-bit)	Y					
macOS Sierra (64-bit)	Y	Y	Y			
macOS El Capitan (64-bit)	Y	Y	Y			

Driver Versions

This section lists the driver files that are included with each version of the Microsoft Drivers for PHP for SQL Server. Each installation package contains SQLSRV and PDO_SQLSRV driver files in threaded and non-threaded

variants. On Windows, they are also available in 32-bit and 64-bit variants. To configure the driver for use with the PHP runtime, follow the installation instructions in Loading the Microsoft Drivers for PHP for SQL Server.

On supported versions of Linux and macOS, the appropriate drivers can be installed using PHP's PECL package system, following the Linux and macOS installation instructions. Alternatively, you can download prebuilt binaries for your platform from the Microsoft Drivers for PHP for SQL Server Github project page -- the tables below list the files found in the prebuilt binary packages.

Microsoft Drivers 5.3 for PHP for SQL Server:

On Windows, the following versions of the driver are included:

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
32-bit php_sqlsrv_7_nts.dll 32-bit php_pdo_sqlsrv_7_nts.dll	7.0	no	32-bit php7.dll
32-bit php_sqlsrv_7_ts.dll 32-bit php_pdo_sqlsrv_7_ts.dll	7.0	yes	32-bit php7ts.dll
64-bit php_sqlsrv_7_nts.dll 64-bit php_pdo_sqlsrv_7_nts.dll	7.0	no	64-bit php7.dll
64-bit php_sqlsrv_7_ts.dll 64-bit php_pdo_sqlsrv_7_ts.dll	7.0	yes	64-bit php7ts.dll
32-bit php_sqlsrv_71_nts.dll 32-bit php_pdo_sqlsrv_71_nts.dll	7.1	no	32-bit php7.dll
32-bit php_sqlsrv_71_ts.dll 32-bit php_pdo_sqlsrv_71_ts.dll	7.1	yes	32-bit php7ts.dll
64-bit php_sqlsrv_71_nts.dll 64-bit php_pdo_sqlsrv_71_nts.dll	7.1	no	64-bit php7.dll
64-bit php_sqlsrv_71_ts.dll 64-bit php_pdo_sqlsrv_71_ts.dll	7.1	yes	64-bit php7ts.dll
32-bit php_sqlsrv_72_nts.dll 32-bit php_pdo_sqlsrv_72_nts.dll	7.2	no	32-bit php7.dll
32-bit php_sqlsrv_72_ts.dll 32-bit php_pdo_sqlsrv_72_ts.dll	7.2	yes	32-bit php7ts.dll
64-bit php_sqlsrv_72_nts.dll 64-bit php_pdo_sqlsrv_72_nts.dll	7.2	no	64-bit php7.dll

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
64-bit php_sqlsrv_72_ts.dll 64-bit php_pdo_sqlsrv_72_ts.dll	7.2	yes	64-bit php7ts.dll

On Linux, the following versions of the driver are included:

DRIVER FILE	PHP VERSION	THREAD SAFE?
php_sqlsrv_7_nts.so php_pdo_sqlsrv_7_nts.so	7.0	no
php_sqlsrv_7_ts.so php_pdo_sqlsrv_7_ts.so	7.0	yes
php_sqlsrv_71_nts.so php_pdo_sqlsrv_71_nts.so	7.1	no
php_sqlsrv_71_ts.so php_pdo_sqlsrv_71_ts.so	7.1	yes
php_sqlsrv_72_nts.so php_pdo_sqlsrv_72_nts.so	7.2	no
php_sqlsrv_72_ts.so php_pdo_sqlsrv_72_ts.so	7.2	yes

Microsoft Drivers 5.2 for PHP for SQL Server:

On Windows, the following versions of the driver are included:

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
32-bit php_sqlsrv_7_nts.dll 32-bit php_pdo_sqlsrv_7_nts.dll	7.0	no	32-bit php7.dll
32-bit php_sqlsrv_7_ts.dll 32-bit php_pdo_sqlsrv_7_ts.dll	7.0	yes	32-bit php7ts.dll
64-bit php_sqlsrv_7_nts.dll 64-bit php_pdo_sqlsrv_7_nts.dll	7.0	no	64-bit php7.dll
64-bit php_sqlsrv_7_ts.dll 64-bit php_pdo_sqlsrv_7_ts.dll	7.0	yes	64-bit php7ts.dll
32-bit php_sqlsrv_71_nts.dll 32-bit php_pdo_sqlsrv_71_nts.dll	7.1	no	32-bit php7.dll

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
32-bit php_sqlsrv_71_ts.dll 32-bit php_pdo_sqlsrv_71_ts.dll	7.1	yes	32-bit php7ts.dll
64-bit php_sqlsrv_71_nts.dll 64-bit php_pdo_sqlsrv_71_nts.dll	7.1	no	64-bit php7.dll
64-bit php_sqlsrv_71_ts.dll 64-bit php_pdo_sqlsrv_71_ts.dll	7.1	yes	64-bit php7ts.dll
32-bit php_sqlsrv_72_nts.dll 32-bit php_pdo_sqlsrv_72_nts.dll	7.2	no	32-bit php7.dll
32-bit php_sqlsrv_72_ts.dll 32-bit php_pdo_sqlsrv_72_ts.dll	7.2	yes	32-bit php7ts.dll
64-bit php_sqlsrv_72_nts.dll 64-bit php_pdo_sqlsrv_72_nts.dll	7.2	no	64-bit php7.dll
64-bit php_sqlsrv_72_ts.dll 64-bit php_pdo_sqlsrv_72_ts.dll	7.2	yes	64-bit php7ts.dll

On Linux, the following versions of the driver are included:

DRIVER FILE	PHP VERSION	THREAD SAFE?
php_sqlsrv_7_nts.so php_pdo_sqlsrv_7_nts.so	7.0	no
php_sqlsrv_7_ts.so php_pdo_sqlsrv_7_ts.so	7.0	yes
php_sqlsrv_71_nts.so php_pdo_sqlsrv_71_nts.so	7.1	no
php_sqlsrv_71_ts.so php_pdo_sqlsrv_71_ts.so	7.1	yes
php_sqlsrv_72_nts.so php_pdo_sqlsrv_72_nts.so	7.2	no
php_sqlsrv_72_ts.so php_pdo_sqlsrv_72_ts.so	7.2	yes

Microsoft Drivers 4.3 for PHP for SQL Server:

On Windows, the following versions of the driver are included:

DRIVER FILE	PHP VERSION	THREAD SAFE?	USE WITH PHP .DLL
32-bit php_sqlsrv_7_nts.dll 32-bit php_pdo_sqlsrv_7_nts.dll	7.0	no	32-bit php7.dll
32-bit php_sqlsrv_7_ts.dll 32-bit php_pdo_sqlsrv_7_ts.dll	7.0	yes	32-bit php7ts.dll
64-bit php_sqlsrv_7_nts.dll 64-bit php_pdo_sqlsrv_7_nts.dll	7.0	no	64-bit php7.dll
64-bit php_sqlsrv_7_ts.dll 64-bit php_pdo_sqlsrv_7_ts.dll	7.0	yes	64-bit php7ts.dll
32-bit php_sqlsrv_71_nts.dll 32-bit php_pdo_sqlsrv_71_nts.dll	7.1	no	32-bit php7.dll
32-bit php_sqlsrv_71_ts.dll 32-bit php_pdo_sqlsrv_71_ts.dll	7.1	yes	32-bit php7ts.dll
64-bit php_sqlsrv_71_nts.dll 64-bit php_pdo_sqlsrv_71_nts.dll	7.1	no	64-bit php7.dll
64-bit php_sqlsrv_71_ts.dll 64-bit php_pdo_sqlsrv_71_ts.dll	7.1	yes	64-bit php7ts.dll

On Linux, the following versions of the driver are included:

DRIVER FILE	PHP VERSION	THREAD SAFE?
php_sqlsrv_7_nts.so php_pdo_sqlsrv_7_nts.so	7.0	no
php_sqlsrv_7_ts.so php_pdo_sqlsrv_7_ts.so	7.0	yes
php_sqlsrv_71_nts.so php_pdo_sqlsrv_71_nts.so	7.1	no
php_sqlsrv_71_ts.so php_pdo_sqlsrv_71_ts.so	7.1	yes

Microsoft Drivers 4.0 for PHP for SQL Server:

On Windows, the following versions of the driver are included:

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
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 Linux, the following versions of the driver are included:

DRIVER FILE	PHP VERSION	THREAD SAFE?
php_sqlsrv_7_nts.so php_pdo_sqlsrv_7_nts.so	7.0	no
php_sqlsrv_7_ts.so php_pdo_sqlsrv_7_ts.so	7.0	yes

Microsoft Drivers 3.2 for PHP for SQL Server:

On Windows, the following versions of the driver are included:

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:

On Windows, the following versions of the driver are included:

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

See Also

Getting Started with the Microsoft Drivers for PHP for SQL Server

Programming Guide for the Microsoft Drivers for PHP for SQL Server

SQLSRV Driver API Reference

PDO_SQLSRV Driver API Reference

Download the Microsoft Drivers for PHP for SQL Server

10/1/2018 • 2 minutes to read • Edit Online

⊕Download PHP Driver

Windows

The following versions of the drivers for PHP on Windows are available for download:

- Microsoft Drivers 5.3 for PHP for SQL Server
- Microsoft Drivers 5.2 for PHP for SQL Server
- Microsoft Drivers 4.3 for PHP for SQL Server
- Microsoft Drivers 4.0, 3.2, 3.1, and 3.0 for PHP for SQL Server

Linux and macOS

The drivers for PHP are easily downloaded and installed using PECL on Linux and macOS. See the Linux and macOS installation tutorial for details. If you need to install the drivers for PHP on Linux and macOS manually, the following versions are available for download:

- Microsoft Drivers 5.3 for PHP for SQL Server
- Microsoft Drivers 5.2 for PHP for SQL Server
- Microsoft Drivers 4.3 for PHP for SQL Server

See also

Getting Started with the Microsoft Drivers for PHP for SQL Server

System Requirements for the Microsoft Drivers for PHP for SQL Server

Programming Guide for the Microsoft Drivers for PHP for SQL Server

SQLSRV Driver API Reference

PDO_SQLSRV Driver API Reference

Loading the Microsoft Drivers for PHP for SQL Server

10/1/2018 • 3 minutes to read • Edit Online

⊕Download PHP Driver

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

You can download the prebuilt drivers for your platform from the Microsoft Drivers for PHP for SQL Server Github project page. Each installation package contains SQLSRV and PDO_SQLSRV driver files in threaded and non-threaded variants. On Windows, they are also available in 32-bit and 64-bit variants. See System Requirements for the Microsoft Drivers for PHP for SQL Server for a list of the driver files that are contained in each package. The driver file must match the PHP version, architecture, and threadedness of your PHP environment.

On Linux and macOS, the drivers can alternatively be installed using PECL, as found in the installation tutorial.

Moving the Driver File into Your Extension Directory

The driver file must be located in a directory where the PHP runtime can find it. It is easiest to put the driver file in your default PHP extension directory - to find the default directory, run `php -i | sls extension_dir` on Windows or `php -i | grep extension_dir` on Linux/macOS. If you are not using the default extension directory, specify a directory in the PHP configuration file (php.ini), using the **extension_dir** option. For example, on Windows, if you have put the driver file in your `c:\php\ext` directory, add the following line to php.ini:

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

Loading the Driver at PHP Startup

To load the SQLSRV driver 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, changing the filename as appropriate:

   On Windows:

   ```
   extension=php_sqlsrv_72_ts.dll
   ```

   On Linux, if you have downloaded the prebuilt binaries for your distribution:

   ```
   extension=php_sqlsrv_72_nts.so
   ```

   If you have compiled the SQLSRV binary from source or with PECL, it will instead be named sqlsrv.so:

   ```
   extension=sqlsrv.so
   ```

2. To enable 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.

   On Windows, the prebuilt PHP binaries come with PDO built-in, so there is no need to modify php.ini to load it. If, however, you have compiled PHP from source and specified a separate PDO extension to be built, it will be named `php_pdo.dll`, and you must copy it to your extension directory and add the following line to php.ini:

   ```
   extension=php_pdo.dll
   ```

   On Linux, if you have installed PHP using your system's package manager, PDO is probably installed as a dynamically loaded extension named pdo.so. The PDO extension must be loaded before the PDO_SQLSRV extension, or loading will fail. Extensions are usually loaded using individual .ini files, and these files are read after php.ini. Therefore, if pdo.so is loaded through its own .ini file, a separate file loading the PDO_SQLSRV driver after PDO is required.

   To find out which directory the extension-specific .ini files are located, run `php --ini` and note the directory listed under `Scan for additional .ini files in:`. Find the file that loads pdo.so -- it is likely prefixed by a number, such as 10-pdo.ini. The numerical prefix indicates the loading order of the .ini files, while files that do not have a numerical prefix are loaded alphabetically. Create a file to load the PDO_SQLSRV driver file called either 30-pdo_sqlsrv.ini (any number larger than the one that prefixes pdo.ini works) or pdo_sqlsrv.ini (if pdo.ini is not prefixed by a number), and add the following line to it, changing the filename as appropriate:

   ```
   extension=php_pdo_sqlsrv_72_nts.so
   ```

   As with SQLSRV, if you have compiled the PDO_SQLSRV binary from source or with PECL, it will instead be named pdo_sqlsrv.so:

   ```
   extension=pdo_sqlsrv.so
   ```

   Copy this file to the directory that contains the other .ini files.

   If you have compiled PHP from source with built-in PDO support, you do not require a separate .ini file, and you can add the appropriate line above to php.ini.

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.

See Also

Getting Started with the Microsoft Drivers for PHP for SQL Server

System Requirements for the Microsoft Drivers for PHP for SQL Server

Programming Guide for the Microsoft Drivers for PHP for SQL Server

SQLSRV Driver API Reference

Configuring IIS for the Microsoft Drivers for PHP for SQL Server

10/1/2018 • 2 minutes to read • Edit Online

⊕ 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 reuses 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 Microsoft Drivers for PHP for SQL Server

Programming Guide for the Microsoft Drivers for PHP for SQL Server

SQLSRV Driver API Reference

Constants (Microsoft Drivers for PHP for SQL Server)

Linux and macOS Installation Tutorial for the Microsoft Drivers for PHP for SQL Server

10/1/2018 • 7 minutes to read • Edit Online

The following instructions assume a clean environment and show how to install PHP 7.x, the Microsoft ODBC driver, Apache, and the Microsoft Drivers for PHP for SQL Server on Ubuntu 16.04, 17.10 and 18.04, RedHat 7, Debian 8 and 9, Suse 12, and macOS 10.11, 10.12 and 10.13. These instructions advise installing the drivers using PECL, but you can also download the prebuilt binaries from the Microsoft Drivers for PHP for SQL Server Github project page and install them following the instructions in Loading the Microsoft Drivers for PHP for SQL Server. For an explanation of extension loading and why we do not add the extensions to php.ini, see the section on loading the drivers.

These instructions install PHP 7.2 by default -- see the notes at the beginning of each section to install PHP 7.0 or 7.1.

Contents of this page:

- Installing the drivers on Ubuntu 16.04, 17.10, and 18.04
- Installing the drivers on Red Hat 7
- Installing the drivers on Debian 8 and 9
- Installing the drivers on Suse 12
- Installing the drivers on macOS El Capitan, Sierra and High Sierra

Installing the drivers on Ubuntu 16.04, 17.10 and 18.04

> **NOTE**
>
> To install PHP 7.0 or 7.1, replace 7.2 with 7.0 or 7.1 in the following commands. For Ubuntu 18.04, the step to add the ondrej repository is not required unless PHP 7.0 or 7.1 is needed. However, installing PHP 7.0 or 7.1 in Ubuntu 18.04 may not work as packages from the ondrej repository come with dependencies that may conflict with a base Ubuntu 18.04 install.

Step 1. Install PHP

```
sudo su
add-apt-repository ppa:ondrej/php -y
apt-get update
apt-get install php7.2 php7.2-dev php7.2-xml -y --allow-unauthenticated
```

Step 2. Install prerequisites

Install the ODBC driver for Ubuntu by following the instructions on the Linux and macOS installation page.

Step 3. Install the PHP drivers for Microsoft SQL Server

```
sudo pecl install sqlsrv
sudo pecl install pdo_sqlsrv
sudo su
echo extension=pdo_sqlsrv.so >> `php --ini | grep "Scan for additional .ini files" | sed -e "s|.*:\s*||"`/30-
pdo_sqlsrv.ini
echo extension=sqlsrv.so >> `php --ini | grep "Scan for additional .ini files" | sed -e "s|.*:\s*||"`/20-
sqlsrv.ini
exit
```

Step 4. Install Apache and configure driver loading

```
sudo su
apt-get install libapache2-mod-php7.2 apache2
a2dismod mpm_event
a2enmod mpm_prefork
a2enmod php7.2
echo "extension=pdo_sqlsrv.so" >> /etc/php/7.2/apache2/conf.d/30-pdo_sqlsrv.ini
echo "extension=sqlsrv.so" >> /etc/php/7.2/apache2/conf.d/20-sqlsrv.ini
exit
```

Step 5. Restart Apache and test the sample script

```
sudo service apache2 restart
```

To test your installation, see Testing your installation at the end of this document.

Installing the drivers on Red Hat 7

> **NOTE**
>
> To install PHP 7.0 or 7.1, replace remi-php72 with remi-php70 or remi-php71 respectively in the following commands.

Step 1. Install PHP

```
sudo su
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
wget http://rpms.remirepo.net/enterprise/remi-release-7.rpm
rpm -Uvh remi-release-7.rpm epel-release-latest-7.noarch.rpm
subscription-manager repos --enable=rhel-7-server-optional-rpms
yum-config-manager --enable remi-php72
yum update
yum install php php-pdo php-xml php-pear php-devel re2c gcc-c++ gcc
```

Step 2. Install prerequisites

Install the ODBC driver for Red Hat 7 by following the instructions on the Linux and macOS installation page.

Compiling the PHP drivers with PECL with PHP 7.2 requires a more recent GCC than the default:

```
sudo yum-config-manager --enable rhel-server-rhscl-7-rpms
sudo yum install devtoolset-7
scl enable devtoolset-7 bash
```

Step 3. Install the PHP drivers for Microsoft SQL Server

```
sudo pecl install sqlsrv
sudo pecl install pdo_sqlsrv
sudo su
echo extension=pdo_sqlsrv.so >> `php --ini | grep "Scan for additional .ini files" | sed -e "s|.*:\s*||"`/30-
pdo_sqlsrv.ini
echo extension=sqlsrv.so >> `php --ini | grep "Scan for additional .ini files" | sed -e "s|.*:\s*||"`/20-
sqlsrv.ini
exit
```

An issue in PECL may prevent correct installation of the latest version of the drivers even if you have upgraded GCC. To install, download the packages and compile manually (similar steps for pdo_sqlsrv):

```
pecl download sqlsrv
tar xvzf sqlsrv-5.3.0.tgz
cd sqlsrv-5.3.0/
phpize
./configure --with-php-config=/usr/bin/php-config
make
sudo make install
```

You can alternatively download the prebuilt binaries from the Github project page, or install from the Remi repo:

```
sudo yum install php-sqlsrv php-pdo_sqlsrv
```

Step 4. Install Apache

```
sudo yum install httpd
```

SELinux is installed by default and runs in Enforcing mode. To allow Apache to connect to databases through SELinux, run the following command:

```
sudo setsebool -P httpd_can_network_connect_db 1
```

Step 5. Restart Apache and test the sample script

```
sudo apachectl restart
```

To test your installation, see Testing your installation at the end of this document.

Installing the drivers on Debian 8 and 9

> **NOTE**
>
> To install PHP 7.0 or 7.1, replace 7.2 in the following commands with 7.0 or 7.1.

Step 1. Install PHP

```
sudo su
apt-get install curl apt-transport-https
wget -O /etc/apt/trusted.gpg.d/php.gpg https://packages.sury.org/php/apt.gpg
echo "deb https://packages.sury.org/php/ $(lsb_release -sc) main" > /etc/apt/sources.list.d/php.list
apt-get update
apt-get install -y php7.2 php7.2-dev php7.2-xml
```

Step 2. Install prerequisites

Install the ODBC driver for Debian by following the instructions on the Linux and macOS installation page.

You may also need to generate the correct locale to get PHP output to display correctly in a browser. For example, for the en_US UTF-8 locale, run the following commands:

```
sudo su
sed -i 's/# en_US.UTF-8 UTF-8/en_US.UTF-8 UTF-8/g' /etc/locale.gen
locale-gen
```

Step 3. Install the PHP drivers for Microsoft SQL Server

```
sudo pecl install sqlsrv
sudo pecl install pdo_sqlsrv
sudo su
echo extension=pdo_sqlsrv.so >> `php --ini | grep "Scan for additional .ini files" | sed -e "s|.*:\s*||"`/30-
pdo_sqlsrv.ini
echo extension=sqlsrv.so >> `php --ini | grep "Scan for additional .ini files" | sed -e "s|.*:\s*||"`/20-
sqlsrv.ini
exit
```

Step 4. Install Apache and configure driver loading

```
sudo su
apt-get install libapache2-mod-php7.2 apache2
a2dismod mpm_event
a2enmod mpm_prefork
a2enmod php7.2
echo "extension=pdo_sqlsrv.so" >> /etc/php/7.2/apache2/conf.d/30-pdo_sqlsrv.ini
echo "extension=sqlsrv.so" >> /etc/php/7.2/apache2/conf.d/20-sqlsrv.ini
```

Step 5. Restart Apache and test the sample script

```
sudo service apache2 restart
```

To test your installation, see Testing your installation at the end of this document.

Installing the drivers on Suse 12

> **NOTE**
>
> To install PHP 7.0, skip the command below adding the repository - 7.0 is the default PHP on suse 12. To install PHP 7.1, replace the repository URL below with the following URL:
>
> ```
> http://download.opensuse.org/repositories/devel:/languages:/php:/php71/SLE_12/devel:languages:php:php71.repo
> ```

Step 1. Install PHP

```
sudo su
zypper -n ar -f http://download.opensuse.org/repositories/devel:languages:php/SLE_12/devel:languages:php.repo
zypper --gpg-auto-import-keys refresh
zypper -n install php7 php7-pear php7-devel
```

Step 2. Install prerequisites

Install the ODBC driver for Suse 12 by following the instructions on the Linux and macOS installation page.

Step 3. Install the PHP drivers for Microsoft SQL Server

```
sudo pecl install sqlsrv
sudo pecl install pdo_sqlsrv
sudo su
echo extension=pdo_sqlsrv.so >> `php --ini | grep "Scan for additional .ini files" | sed -e
"s|.*:\s*||"`/pdo_sqlsrv.ini
echo extension=sqlsrv.so >> `php --ini | grep "Scan for additional .ini files" | sed -e
"s|.*:\s*||"`/sqlsrv.ini
exit
```

Step 4. Install Apache and configure driver loading

```
sudo su
zypper install apache2 apache2-mod_php7
a2enmod php7
echo "extension=sqlsrv.so" >> /etc/php7/apache2/php.ini
echo "extension=pdo_sqlsrv.so" >> /etc/php7/apache2/php.ini
exit
```

Step 5. Restart Apache and test the sample script

```
sudo systemctl restart apache2
```

To test your installation, see Testing your installation at the end of this document.

Installing the drivers on macOS El Capitan, Sierra and High Sierra

If you do not already have it, install brew as follows:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

> **NOTE**
>
> To install PHP 7.0 or 7.1, replace php@7.2 with php@7.0 or php@7.1 respectively in the following commands.

Step 1. Install PHP

```
brew tap
brew tap homebrew/core
brew install php@7.2
```

PHP should now be in your path -- run `php -v` to verify that you are running the correct version of PHP. If PHP is not in your path or it is not the correct version, run the following:

```
brew link --force --overwrite php@7.2
```

Step 2. Install prerequisites

Install the ODBC driver for macOS by following the instructions on the Linux and macOS installation page.

In addition, you may need to install the GNU make tools:

```
brew install autoconf automake libtool
```

Step 3. Install the PHP drivers for Microsoft SQL Server

```
sudo pecl install sqlsrv
sudo pecl install pdo_sqlsrv
```

Step 4. Install Apache and configure driver loading

```
brew install apache2
```

To find the Apache configuration file for your Apache installation, run

```
apachectl -V | grep SERVER_CONFIG_FILE
```

and substitute the path for `httpd.conf` in the following commands:

```
echo "LoadModule php7_module /usr/local/opt/php@7.2/lib/httpd/modules/libphp7.so" >>
/usr/local/etc/httpd/httpd.conf
(echo "<FilesMatch .php$>"; echo "SetHandler application/x-httpd-php"; echo "</FilesMatch>";) >>
/usr/local/etc/httpd/httpd.conf
```

Step 5. Restart Apache and test the sample script

```
sudo apachectl restart
```

To test your installation, see Testing your installation at the end of this document.

Testing Your Installation

To test this sample script, create a file called testsql.php in your system's document root. This is `/var/www/html/` on Ubuntu, Debian, and Redhat, `/srv/www/htdocs` on SUSE, or `/usr/local/var/www` on macOS. Copy the following script to it, replacing the server, database, username, and password as appropriate.

```php
<?php
$serverName = "yourServername";
$connectionOptions = array(
    "database" => "yourDatabase",
    "uid" => "yourUsername",
    "pwd" => "yourPassword"
);

// Establishes the connection
$conn = sqlsrv_connect($serverName, $connectionOptions);
if ($conn === false) {
    die(formatErrors(sqlsrv_errors()));
}

// Select Query
$tsql = "SELECT @@Version AS SQL_VERSION";

// Executes the query
$stmt = sqlsrv_query($conn, $tsql);

// Error handling
if ($stmt === false) {
    die(formatErrors(sqlsrv_errors()));
}
?>

<h1> Results : </h1>

<?php
while ($row = sqlsrv_fetch_array($stmt, SQLSRV_FETCH_ASSOC)) {
    echo $row['SQL_VERSION'] . PHP_EOL;
}

sqlsrv_free_stmt($stmt);
sqlsrv_close($conn);

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/>";
    }
}
?>
```

Point your browser to http://localhost/testsql.php (http://localhost:8080/testsql.php on macOS). You should now be able to connect to your SQL Server/Azure SQL database.

See Also

Getting Started with the Microsoft Drivers for PHP for SQL Server

Loading the Microsoft Drivers for PHP for SQL Server

System Requirements for the Microsoft Drivers for PHP for SQL Server

Download PHP Driver

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

What's New in Version 5.3

- Support for Microsoft ODBC Driver 17.2 on all platforms
- Support for macOS High Sierra (requires ODBC Driver 17 and above)
- Support for Azure Key Vault for Always Encrypted for basic CRUD functionalities such that Always Encrypted feature is available to all supported Windows, Linux or macOS platforms Using Always Encrypted with the PHP Drivers for SQL Server
- Support Ubuntu 18.04 LTS (requires ODBC Driver 17.2)
- Support for Connection Resiliency in Linux or macOS as well (requires ODBC Driver 17.2)

What's New in Version 5.2

- Support for PHP 7.2.1 and up on Windows, and 7.2.0 and up on other platforms
- Support for Microsoft ODBC Driver 17
  - Version 17 is now the default on all platforms
- Support for Ubuntu 17.10, Debian 9, and Suse Enterprise Linux 12
- Dropped support for Ubuntu 15.10
- Support for Always Encrypted with CRUD functionalities on Windows. For more information, see Using Always Encrypted with the PHP Drivers for SQL Server
  - Support for Windows Certificate Store
  - Always Encrypted is only supported with Microsoft ODBC Driver 17 and above
- Support for non-UTF8 locales on Linux and macOS
  - Non-UTF8 locales on Linux and macOS are only supported with Microsoft ODBC Driver 17 and above
- Support for Azure SQL Data Warehouse
- Support for Azure SQL Managed Instance (Extended Private Preview)

What's New in Version 4.3

- Support for PHP 7.1
- Support for macOS Sierra and macOS El Capitan
- Support for Ubuntu 15.10, and Debian 8
- Dropped support for Ubuntu 15.04
- Support for Always On Availability groups via Transparent Network IP Resolution. For more information, see Connection Options.
- Added support for sql_variant data type with limitation.
- Idle Connection Resiliency support in Windows. For more information, see Connection Options.
- Connection pooling support for Linux and macOS. For more information, see Connection Pooling.
- Support for Azure Active Directory Authentication with ActiveDirectoryPassword and SqlPassword. For more

information, see Connection Options.

What's New in Version 4.0

- Support for PHP 7.0
- Full 64-bit support
- Support for Ubuntu 15.04, Ubuntu 16.04, and RedHat 7

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 (11.x). For more information, see 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 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. For more information, see PDO::prepare.

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 Microsoft Drivers for PHP for SQL Server

Support Resources for the Microsoft Drivers for PHP for SQL Server

10/1/2018 • 2 minutes to read • Edit Online

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 Documentation.

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, you can contact Microsoft Support, or ask on the Github project page.

See Also

Overview of the Microsoft Drivers for PHP for SQL Server

About Code Examples in the Documentation

10/1/2018 • 2 minutes to read • Edit Online

Download PHP Driver

Remarks about the code examples

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 2008 or later 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 and install the AdventureWorks database, see the AdventureWorks page in the SQL Server Samples Github repository.

- 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 meant 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 format output nicely, 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 Microsoft Drivers for PHP for SQL Server

Microsoft PHP Drivers for SQL Server Support Matrix

10/1/2018 • 3 minutes to read • Edit Online

⬇ Download PHP Driver

This page contains the support matrix and support lifecycle policy for the Microsoft PHP Drivers for SQL Server.

Microsoft PHP Drivers Support Lifecycle Matrix and Policy

The Microsoft Support Lifecycle (MSL) policy provides transparent, predictable information regarding the support lifecycle of Microsoft products. PHP Drivers versions 3.x, 4.x, and 5.x have five years of Mainstream support from the driver release date. Mainstream support is defined on the Microsoft support lifecycle website.

Extended and custom support options are not available for the Microsoft PHP Drivers.

The following Microsoft PHP Drivers are supported, until the indicated End of Support date.

DRIVER NAME	DRIVER PACKAGE VERSION	END OF MAINSTREAM SUPPORT
Microsoft PHP Drivers 5.3 for SQL Server	5.3	July 20, 2023
Microsoft PHP Drivers 5.2 for SQL Server	5.2	February 9, 2023
Microsoft PHP Drivers 4.3 for SQL Server	4.3	July 6, 2022
Microsoft PHP Drivers 4.0 for SQL Server	4.0	July 11, 2021
Microsoft PHP Drivers 3.2 for SQL Server	3.2	March 9, 2020
Microsoft PHP Drivers 3.1 for SQL Server	3.1	December 12, 2019

The following Microsoft PHP Drivers are no longer supported.

DRIVER NAME	DRIVER PACKAGE VERSION	END OF MAINSTREAM SUPPORT
Microsoft PHP Drivers 3.0 for SQL Server	3.0	March 6, 2017
Microsoft PHP Drivers 2.0 for SQL Server	2.0	August 10, 2015
Microsoft PHP Drivers 1.0 for SQL Server	1.0	April 28, 2014

SQL Server Version Certified Compatibility

The following matrix lists SQL Server versions that have been tested and certified as compatible with the corresponding driver version. We strive to maintain backward compatibility with previous driver versions, but only the latest supported driver is tested and certified with new SQL Server versions as SQL Server is released.

PHP FOR SQL SERVER DRIVER VERSION → ↓ SQL SERVER VERSION	5.3 AND 5.2	4.3	4.0	3.2	3.1	3.0	2.0
Azure SQL Managed Instance (Extended Private Preview)	Y	Y					
Azure SQL Data Warehouse	Y	Y					
SQL Server 2017	Y	Y					
SQL Server 2016	Y	Y	Y				
SQL Server 2014	Y	Y	Y	Y	Y		
SQL Server 2012	Y	Y	Y	Y	Y	Y	
SQL Server 2008 R2	Y	Y	Y	Y	Y	Y	Y
SQL Server 2008			Y	Y	Y	Y	Y

PHP Version Support

The following versions of PHP are supported with the listed version of the Microsoft PHP Drivers:

PHP FOR SQL SERVER DRIVER VERSION → ↓ PHP VERSION	5.3 AND 5.2	4.3	4.0	3.2	3.1	3.0	2.0
7.2	7.2.1+ on Windows 7.2.0+ on other platforms						
7.1	7.1.0+	7.1.0+					

PHP FOR SQL SERVER DRIVER VERSION → ↓ PHP VERSION	5.3 AND 5.2	4.3	4.0	3.2	3.1	3.0	2.0
7.0	7.0.0+	7.0.0+	7.0.0+				
5.6				5.6.4+			
5.5				5.5.16+	5.5.16+		
5.4				5.4.32	5.4.32	5.4.32	
5.3						5.3.0	5.3.0
5.2							5.2.4 5.2.13

Supported Operating Systems

The following Windows operating system versions are supported with the listed version of the Microsoft PHP Drivers:

PHP FOR SQL SERVER DRIVER VERSION → ↓ OPERATING SYSTEM	5.3 AND 5.2	4.3	4.0	3.2	3.1	3.0	2.0
Windows Server 2016	Y	Y					
Windows Server 2012 R2	Y	Y	Y	Y	Y		
Windows Server 2012	Y	Y	Y	Y	Y		
Windows Server 2008 R2 SP1			Y	Y	Y	Y	
Windows Server 2008 R2							Y
Windows Server 2008 SP2			Y	Y	Y	Y	
Windows Server 2008							Y

PHP FOR SQL SERVER DRIVER VERSION → ↓ OPERATING SYSTEM	5.3 AND 5.2	4.3	4.0	3.2	3.1	3.0	2.0
Windows Server 2003 SP1							Y
Windows 10	Y	Y	Y				
Windows 8.1	Y	Y	Y	Y	Y		
Windows 8		Y	Y	Y	Y		
Windows 7 SP1			Y	Y	Y	Y	
Windows Vista SP2			Y	Y	Y	Y	Y
Windows XP SP3							Y

The following Linux and Mac operating system versions (64-bit only) are supported with the listed version of the Microsoft PHP Drivers:

PHP FOR SQL SERVER DRIVER VERSION → ↓ OPERATING SYSTEM	5.3	5.2	4.3	4.0	3.2	3.1	3.0	2.0
Ubuntu 18.04 (64-bit)	Y							
Ubuntu 17.10 (64-bit)	Y	Y						
Ubuntu 16.04 (64-bit)	Y	Y	Y	Y				
Ubuntu 15.10 (64-bit)			Y					

PHP FOR SQL SERVER DRIVER VERSION → ↓ OPERATING SYSTEM	5.3	5.2	4.3	4.0	3.2	3.1	3.0	2.0
Ubuntu 15.04 (64-bit)				Y				
Debian 9 (64-bit)	Y	Y						
Debian 8 (64-bit)	Y	Y	Y					
Red Hat Enterprise Linux 7 (64-bit)	Y	Y	Y	Y				
Suse Enterprise Linux 12 (64-bit)	Y	Y						
macOS High Sierra (64-bit)	Y							
macOS Sierra (64-bit)	Y	Y	Y					
macOS El Capitan (64-bit)	Y	Y	Y					

See Also

Release Notes

Support Resources

System Requirements

Programming Guide for the Microsoft Drivers for PHP for SQL Server

10/1/2018 • 2 minutes to read • Edit Online

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.
Using Always Encrypted with the Microsoft Drivers for PHP for SQL Server	Provides information about using Always Encrypted feature with the PHP drivers.
Security Considerations for the Microsoft Drivers for PHP for SQL Server	Describes security considerations for developing, deploying, and running applications.

See Also

Overview of the Microsoft Drivers for PHP for SQL Server

Getting Started with the Microsoft Drivers for PHP for SQL Server

SQLSRV Driver API Reference

Constants (Microsoft Drivers for PHP for SQL Server)

Example Application (SQLSRV Driver)

Connecting to the Server

10/1/2018 • 2 minutes to read • Edit Online

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 Using Azure Active Directory Authentication	Describes how to set the authentication mode and connect using Azure Active Directory identities.
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.
Support for LocalDB	Describes Microsoft Drivers for PHP for SQL Server support for the LocalDB feature, which was added in SQL Server 2012 (11.x).
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 (11.x).
Connecting to Microsoft Azure SQL Database	Discusses how to connect to an Azure SQL Database.
Connection Resiliency	Discusses the connection resiliency feature that reestablishes broken connections.

See Also

Programming Guide for the Microsoft Drivers for PHP for SQL Server

How to: Connect Using Windows Authentication

10/1/2018 • 2 minutes to read • Edit Online

⊕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 note 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
<?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
<?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 the Microsoft Drivers for PHP for SQL Server

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

10/1/2018 • 3 minutes to read • Edit Online

⊕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
<?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
<?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 the Microsoft Drivers for PHP for SQL Server

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)

Connect Using Azure Active Directory Authentication

10/1/2018 • 3 minutes to read • Edit Online

⊕Download PHP Driver

Azure Active Directory (Azure AD) is a central user ID management technology that operates as an alternative to SQL Server authentication. Azure AD allows connections to Microsoft Azure SQL Database and SQL Data Warehouse with federated identities in Azure AD using a username and password, Windows Integrated Authentication, or an Azure AD access token; the PHP drivers for SQL Server offer partial support for these features.

To use Azure AD, use the **Authentication** keyword. The values that **Authentication** can take on are explained in the following table.

KEYWORD	VALUES	DESCRIPTION
Authentication	Not set (default)	Authentication mode determined by other keywords. For more information, see Connection Options.
	`SqlPassword`	Directly authenticate to a SQL Server instance (which may be an Azure instance) using a username and password. The username and password must be passed into the connection string using the **UID** and **PWD** keywords.
	`ActiveDirectoryPassword`	Authenticate with an Azure Active Directory identity using a username and password. The username and password must be passed into the connection string using the **UID** and **PWD** keywords.

The **Authentication** keyword affects the connection security settings. If it is set in the connection string, then by default the **Encrypt** keyword is set to true, so the client will request encryption. Moreover, the server certificate will be validated irrespective of the encryption setting unless **TrustServerCertificate** is set to true. This is distinguished from the old, and less secure, login method, in which the server certificate is not validated unless encryption is specifically requested in the connection string.

Before using Azure AD with the PHP drivers for SQL Server on Windows, ensure that you have installed the Microsoft Online Services Sign-In Assistant (not required for Linux and MacOS).

Limitations

On Windows, the underlying ODBC driver supports one more value for the **Authentication** keyword, **ActiveDirectoryIntegrated**, but the PHP drivers do not support this value on any platform and hence also do not support Azure AD token-based authentication.

Example

The following example shows how to connect using **SqlPassword** and **ActiveDirectoryPassword**.

```php
<?php
// First connect to a local SQL Server instance by setting Authentication to SqlPassword
$serverName = "myserver.mydomain";

$connectionInfo = array( "UID"=>$myusername, "PWD"=>$mypassword, "Authentication"=>'SqlPassword' );

$conn = sqlsrv_connect( $serverName, $connectionInfo );
if( $conn === false )
{
    echo "Could not connect with Authentication=SqlPassword.\n";
    print_r( sqlsrv_errors() );
}
else
{
    echo "Connected successfully with Authentication=SqlPassword.\n";
    sqlsrv_close( $conn );
}

// Now connect to an Azure SQL database by setting Authentication to ActiveDirectoryPassword
$azureServer = "myazureserver.database.windows.net";
$azureDatabase = "myazuredatabase";
$azureUsername = "myuid";
$azurePassword = "mypassword";
$connectionInfo = array( "Database"=>$azureDatabase, "UID"=>$azureUsername, "PWD"=>$azurePassword,
                         "Authentication"=>'ActiveDirectoryPassword' );

$conn = sqlsrv_connect( $azureServer, $connectionInfo );
if( $conn === false )
{
    echo "Could not connect with Authentication=ActiveDirectoryPassword.\n";
    print_r( sqlsrv_errors() );
}
else
{
    echo "Connected successfully with Authentication=ActiveDirectoryPassword.\n";
    sqlsrv_close( $conn );
}

?>
```

The following example does the same as above with the PDO_SQLSRV driver.

```php
<?php
// First connect to a local SQL Server instance by setting Authentication to SqlPassword
$serverName = "myserver.mydomain";

$connectionInfo = "Database = $databaseName; Authentication = SqlPassword;";

try
{
    $conn = new PDO( "sqlsrv:server = $serverName ; $connectionInfo", $myusername, $mypassword );
    echo "Connected successfully with Authentication=SqlPassword.\n";
    $conn = null;
}
catch( PDOException $e )
{
    echo "Could not connect with Authentication=SqlPassword.\n";
    print_r( $e->getMessage() );
    echo "\n";
}

// Now connect to an Azure SQL database by setting Authentication to ActiveDirectoryPassword
$azureServer = "myazureserver.database.windows.net";
$azureDatabase = "myazuredatabase";
$azureUsername = "myuid";
$azurePassword = "mypassword";
$connectionInfo = "Database = $azureDatabase; Authentication = ActiveDirectoryPassword;";

try
{
    $conn = new PDO( "sqlsrv:server = $azureServer ; $connectionInfo", $azureUsername, $azurePassword );
    echo "Connected successfully with Authentication=ActiveDirectoryPassword.\n";
    $conn = null;
}
catch( PDOException $e )
{
    echo "Could not connect with Authentication=ActiveDirectoryPassword.\n";
    print_r( $e->getMessage() );
    echo "\n";
}

?>
```

See Also

Using Azure Active Directory with the ODBC Driver

How to: Connect on a Specified Port

10/1/2018 • 2 minutes to read • Edit Online

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 the Microsoft Drivers for PHP for SQL Server

Getting Started with the Microsoft Drivers for PHP for SQL Server

SQLSRV Driver API Reference

PDO_SQLSRV Driver Reference

Connection Pooling (Microsoft Drivers for PHP for SQL Server)

10/1/2018 • 2 minutes to read • Edit Online

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 in Windows. In Linux and macOS, connections are pooled only if connection pooling is enabled for ODBC (see Enabling/Disabling connection pooling). When connection pooling is enabled and 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.

Enabling/Disabling connection pooling

Windows

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 only creates a new connection if an equivalent connection does not exist in the connection pool.

For information about other connection attributes, see Connection Options.

Linux and macOS

The *ConnectionPooling* attribute cannot be used to enable/disable connection pooling.

Connection pooling can be enabled/disabled by editing the odbcinst.ini configuration file. The driver should be reloaded for the changes to take effect.

Setting `Pooling` to `Yes` and a positive `CPTimeout` value in the odbcinst.ini file enables connection pooling.

```
[ODBC]
Pooling=Yes

[ODBC Driver 13 for SQL Server]
CPTimeout=<int value>
```

Minimally, the odbcinst.ini file should look something like this example:

```
[ODBC]
Pooling=Yes

[ODBC Driver 13 for SQL Server]
Description=Microsoft ODBC Driver 13 for SQL Server
Driver=/opt/microsoft/msodbcsql/lib64/libmsodbcsql-13.1.so.3.0
UsageCount=1
CPTimeout=120
```

Setting `Pooling` to `No` in the odbcinst.ini file forces the driver to create a new connection.

```
[ODBC]
Pooling=No
```

Remarks

- In Linux or macOS, all connections will be pooled if pooling is enabled in the odbcinst.ini file. This means the ConnectionPooling connection option has no effect. To disable pooling, set Pooling=No in the odbcinst.ini file and reload the drivers.
  - unixODBC <= 2.3.4 (Linux and macOS) might not return proper diagnostic information, such as error messages, warnings and informative messages
  - for this reason, SQLSRV and PDO_SQLSRV drivers might not be able to properly fetch long data (such as xml, binary) as strings. Long data can be fetched as streams as a workaround. See the example below for SQLSRV.

```php
<?php
$connectionInfo = array("Database"=>"test", "UID"=>"username", "PWD"=>"password");

$conn1 = sqlsrv_connect("servername", $connectionInfo);

$longSample = str_repeat("a", 8500);
$xml1 =
'<ParentXMLTag>
  <ChildTag01>'.$longSample.'</ChildTag01>
</ParentXMLTag>';

// Create table and insert xml string into it
sqlsrv_query($conn1, "CREATE TABLE xml_table (field xml)");
sqlsrv_query($conn1, "INSERT into xml_table values ('$xml1')");

// retrieve the inserted xml
$column1 = getColumn($conn1);

// return the connection to the pool
sqlsrv_close($conn1);

// This connection is from the pool
$conn2 = sqlsrv_connect("servername", $connectionInfo);
$column2 = getColumn($conn2);

sqlsrv_query($conn2, "DROP TABLE xml_table");
sqlsrv_close($conn2);

function getColumn($conn)
{
    $tsql = "SELECT * from xml_table";
    $stmt = sqlsrv_query($conn, $tsql);
    sqlsrv_fetch($stmt);
    // This might fail in Linux and macOS
    // $column = sqlsrv_get_field($stmt, 0, SQLSRV_PHPTYPE_STRING(SQLSRV_ENC_CHAR));
    // The workaround is to fetch it as a stream
    $column = sqlsrv_get_field($stmt, 0, SQLSRV_PHPTYPE_STREAM(SQLSRV_ENC_CHAR));
    sqlsrv_free_stmt($stmt);
    return ($column);
}
?>
```

See Also

How to: Connect Using Windows Authentication

How to: Connect Using SQL Server Authentication

⊕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
<?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
<?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

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).

Table of Connection Options

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 Support for High Availability, Disaster Recovery.	ReadWrite
AttachDBFileName	String	Specifies which database file the server should attach.	No value set.
Authentication	One of the following strings: 'SqlPassword' 'ActiveDirectoryPassword'	Specifies the authentication mode.	Not 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

KEY	VALUE	DESCRIPTION	DEFAULT
ColumnEncryption	**Enabled** or **Disabled**	Specifies whether the Always Encrypted feature is enabled or not.	Disabled
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**).[1]	**true** (1)
ConnectRetryCount	Integer between 0 and 255 (inclusive)	The maximum number of attempts to reestablish a broken connection before giving up. By default, a single attempt is made to reestablish a connection when broken. A value of 0 means that no reconnection will be attempted.	1
ConnectRetryInterval	Integer between 1 and 60 (inclusive)	The time, in seconds, between attempts to reestablish a connection. The application will attempt to reconnect immediately upon detecting a broken connection, and will then wait ConnectRetryInterval seconds before trying again. This keyword is ignored if ConnectRetryCount is equal to 0.	1
Database	String	Specifies the name of the database in use for the connection being established[2].	The default database for the login being used.
Driver	String	Specifies the Microsoft ODBC driver used to communicate with SQL Server. Possible values are: ODBC Driver 17 for SQL Server ODBC Driver 13 for SQL Server ODBC Driver 11 for SQL Server (Windows only).	When the Driver keyword is not specified, the Microsoft Drivers for PHP for SQL Server attempt to find supported Microsoft ODBC driver(s) in the system, starting with the latest version of ODBC and so on.
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**)[3].	**false** (0)

KEY	VALUE	DESCRIPTION	DEFAULT
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 Support for High Availability, Disaster Recovery.	No value set.
KeyStoreAuthentication	**KeyVaultPassword** **KeyVaultClientSecret**	Authentication method for accessing Azure Key Vault. Controls what kind of credentials are used with KeyStorePrincipalId and KeyStoreSecret. For more information, see Using Azure Key Vault.	Not set.
KeyStorePrincipalId	String	Identifier for the account seeking to access Azure Key Vault. If KeyStoreAuthentication is **KeyVaultPassword**, this must be an Azure Active Directory username. If KeyStoreAuthentication is **KeyVaultClientSecret**, this must be an application client ID.	Not set.
KeyStoreSecret	String	Credential secret for the account seeking to access Azure Key Vault. If KeyStoreAuthentication is **KeyVaultPassword**, this must be an Azure Active Directory password. If KeyStoreAuthentication is **KeyVaultClientSecret**, this must be an application client secret.	Not set.
LoginTimeout	Integer (SQLSRV driver) String (PDO_SQLSRV driver)	Specifies the number of seconds to wait before failing the connection attempt.	No timeout.

KEY	VALUE	DESCRIPTION	DEFAULT
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)
MultiSubnetFailover	String	Always specify **multiSubnetFailover=yes** when connecting to the availability group listener of a SQL Server 2012 (11.x) availability group or a SQL Server 2012 (11.x) 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 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[4].	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)

KEY	VALUE	DESCRIPTION	DEFAULT
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**
Scrollable	String	"buffered" indicates that you want a client-side (buffered) cursor, which allows you to cache an entire result set in memory. For more information, see Cursor Types (SQLSRV Driver).	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 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 is 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 Support for LocalDB.
TraceFile	String	Specifies the path for the file used for trace data.	No value set.

KEY	VALUE	DESCRIPTION	DEFAULT
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)
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
TransparentNetworkIPResolution	**Enabled** or **Disabled**	Affects the connection sequence when the first resolved IP of the hostname does not respond and there are multiple IPs associated with the hostname. It interacts with MultiSubnetFailover to provide different connection sequences. For more information, see Transparent Network IP Resolution or Using Transparent Network IP Resolution.	Enabled

KEY	VALUE	DESCRIPTION	DEFAULT
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[4].	No value set.
WSID	String	Specifies the name of the computer for tracing.	No value set.

1. The `ConnectionPooling` attribute cannot be used to enable/disable connection pooling in Linux and Mac. See Connection Pooling (Microsoft Drivers for PHP for SQL Server).

2. All queries executed on the established connection are made to the database that is specified by the *Database* attribute. However, if the user has the appropriate permissions, data in other databases can be accessed by using a fully qualified name. 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.

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

4. 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

Support for LocalDB

10/1/2018 • 2 minutes to read • Edit Online

⊕Download PHP Driver

LocalDB is a lightweight version of SQL Server which has been available since SQL Server 2012 (11.x). 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 (11.x) Express LocalDB.

In brief, 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);
```

For instructions on installing LocalDB, see the LocalDB documentation. If you use sqlcmd.exe to modify data in your LocalDB instance, you will need the sqlcmd utility.

See Also

Connecting to the Server

Support for High Availability, Disaster Recovery

10/1/2018 • 6 minutes to read • Edit Online

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 (11.x). 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 significantly reduces 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 an 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:

- If the secondary replica location is not configured to accept connections.

- 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**.

Transparent Network IP Resolution (TNIR)

Transparent Network IP Resolution (TNIR) is a revision of the existing MultiSubnetFailover feature. It affects the connection sequence of the driver when the first resolved IP of the hostname does not respond and there are multiple IPs associated with the hostname. Together with MultiSubnetFailover they provide the following four connection sequences:

- TNIR Enabled & MultiSubnetFailover Disabled: One IP is attempted, followed by all IPs in parallel
- TNIR Enabled & MultiSubnetFailover Enabled: All IPs are attempted in parallel
- TNIR Disabled & MultiSubnetFailover Disabled: All IPs are attempted one after another
- TNIR Disabled & MultiSubnetFailover Enabled: All IPs are attempted in parallel

TNIR is enabled by default, and MultiSubnetFailover is Disabled by default.

This is an example of enabling both TNIR and MultiSubnetFailover using the PDO_SQLSRV driver:

```php
<?php
$serverName = "yourservername";
$username = "yourusername";
$password = "yourpassword";
$connectionString = "sqlsrv:Server=$serverName; TransparentNetworkIPResolution=Enabled;
MultiSubnetFailover=yes";
try {
    $conn = new PDO($connectionString, $username, $password, array(PDO::ATTR_ERRMODE =>
PDO::ERRMODE_EXCEPTION));
    // your code
    // more of your code
    // when done, close the connection
    unset($conn);
} catch(PDOException $e) {
    print_r($e->errorInfo);
}
?>
```

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

The keyword **ApplicationIntent** can be specified in your connection string. The assignable values are **ReadWrite** or **ReadOnly**. The default is **ReadWrite**.

When **ApplicationIntent=ReadOnly**, the client requests a read workload when connecting. The server enforces the intent at connection time, and during a **USE** database statement.

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

Targets of ReadOnly

When a connection chooses **ReadOnly**, the connection is assigned to any of the following special configurations that might exist for the database:

- Always On

  - A database can allow or disallow read workloads on the targeted Always On database. This choice is controlled by using the **ALLOW_CONNECTIONS** clause of the **PRIMARY_ROLE** and **SECONDARY_ROLE** Transact-SQL statements.

- Geo-Replication

- Read Scale-Out

If none of those special targets are available, the regular database is read from.

The **ApplicationIntent** keyword enables *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, all of the following apply:

- You must connect to an Always On Availability Group availability group listener.

- The **ApplicationIntent** connection string keyword must be set to **ReadOnly**.

- The Availability Group must be configured by the database administrator to enable read-only routing.

Multiple connections each using read-only routing might 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. You can ensure that all read-only requests connect to the same read-only replica. Ensure this sameness by *not* passing 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. The longer wait is because read-only routing first connects to the primary, and then looks for the best available readable secondary. Due to these multiple staps, you should increase your login timeout to at least 30 seconds.

See Also

Connecting to the Server

Connecting to Microsoft Azure SQL Database

10/1/2018 • 2 minutes to read • Edit Online

Download PHP Driver

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

Idle Connection Resiliency

10/19/2018 • 3 minutes to read • **Edit Online**

⊕ Download PHP Driver

Connection resiliency is the principle that a broken idle connection can be reestablished, within certain constraints. If a connection to Microsoft SQL Server fails, connection resiliency allows the client to automatically attempt to reestablish the connection. Connection resiliency is a property of the data source; only SQL Server 2014 and later and Azure SQL Database support connection resiliency.

Connection resiliency is implemented with two connection keywords that can be added to connection strings: **ConnectRetryCount** and **ConnectRetryInterval**.

KEYWORD	VALUES	DEFAULT	DESCRIPTION
ConnectRetryCount	Integer between 0 and 255 (inclusive)	1	The maximum number of attempts to reestablish a broken connection before giving up. By default, a single attempt is made to reestablish a connection when broken. A value of 0 means that no reconnection will be attempted.
ConnectRetryInterval	Integer between 1 and 60 (inclusive)	1	The time, in seconds, between attempts to reestablish a connection. The application will attempt to reconnect immediately upon detecting a broken connection, and will then wait **ConnectRetryInterval** seconds before trying again. This keyword is ignored if **ConnectRetryCount** is equal to 0.

If the product of **ConnectRetryCount** multiplied by **ConnectRetryInterval** is larger than **LoginTimeout**, then the client will cease attempting to connect once **LoginTimeout** is reached; otherwise, it will continue to try to reconnect until **ConnectRetryCount** is reached.

Remarks

Connection resiliency applies when the connection is idle. Failures that occur while executing a transaction, for example, will not trigger reconnection attempts – they will fail as would otherwise be expected. The following situations, known as non-recoverable session states, will not trigger reconnection attempts:

- Temporary tables
- Global and local cursors
- Transaction context and session level transaction locks
- Application locks
- EXECUTE AS/REVERT security context
- OLE automation handles

- Prepared XML handles
- Trace flags

Example

The following code connects to a database and executes a query. The connection is interrupted by killing the session and a new query is attempted using the broken connection. This example uses the AdventureWorks sample database.

In this example, we specify a buffered cursor before breaking the connection. If we do not specify a buffered cursor, the connection would not be reestablished because there would be an active server-side cursor and thus the connection would not be idle when broken. However, in that case we could call sqlsrv_free_stmt() before breaking the connection to vacate the cursor, and the connection would be successfully reestablished.

```php
<?php
// This function breaks the connection by determining its session ID and killing it.
// A separate connection is used to break the main connection because a session
// cannot kill itself. The sleep() function ensures enough time has passed for KILL
// to finish ending the session.
function BreakConnection( $conn, $conn_break )
{
    $stmt1 = sqlsrv_query( $conn, "SELECT @@SPID" );
    if ( sqlsrv_fetch( $stmt1 ) )
    {
        $spid=sqlsrv_get_field( $stmt1, 0 );
    }

    $stmt2 = sqlsrv_prepare( $conn_break, "KILL ".$spid );
    sqlsrv_execute( $stmt2 );
    sleep(1);
}

// Connect to the local server using Windows authentication and specify
// AdventureWorks as the database in use. Specify values for
// ConnectRetryCount and ConnectRetryInterval as well.
$databaseName = 'AdventureWorks2014';
$serverName = '(local)';
$connectionInfo = array( "Database"=>$databaseName, "ConnectRetryCount"=>10, "ConnectRetryInterval"=>10 );

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

// A separate connection that will be used to break the main connection $conn
$conn_break = sqlsrv_connect( $serverName, array( "Database"=>$databaseName) );

// Create a statement to retrieve the contents of a table
$stmt1 = sqlsrv_query( $conn, "SELECT * FROM HumanResources.Employee",
                       array(), array( "Scrollable"=>"buffered" ) );
if( $stmt1 === false )
{
    echo "Error in statement 1.\n";
    die( print_r( sqlsrv_errors(), true ));
}
else
{
    echo "Statement 1 successful.\n";
    $rowcount = sqlsrv_num_rows( $stmt1 );
    echo $rowcount." rows in result set.\n";
}

// Now break the connection $conn
```

```php
// Now break the connection $conn
BreakConnection( $conn, $conn_break );

// Create another statement. The connection will be reestablished.
$stmt2 = sqlsrv_query( $conn, "SELECT * FROM HumanResources.Department",
                       array(), array( "Scrollable"=>"buffered" ) );
if( $stmt2 === false )
{
    echo "Error in statement 2.\n";
    die( print_r( sqlsrv_errors(), true ));
}
else
{
    echo "Statement 2 successful.\n";
    $rowcount = sqlsrv_num_rows( $stmt2 );
    echo $rowcount." rows in result set.\n";
}

sqlsrv_close( $conn );
sqlsrv_close( $conn_break );
?>
```

Expected output:

```
Statement 1 successful.
290 rows in result set.
Statement 2 successful.
16 rows in result set.
```

See Also

[Connection Resiliency in the Windows ODBC Driver](#)

Comparing Execution Functions

10/1/2018 • 2 minutes to read • <u>Edit Online</u>

Download PHP Driver

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

SQLSRV Execution 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.

PDO_SQLSRV Execution Functions

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 the Microsoft Drivers for PHP for SQL Server

Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver

10/1/2018 • 2 minutes to read • Edit Online

Download PHP Driver

This topic discusses the use of the PDO::SQLSRV_ATTR_DIRECT_QUERY attribute to specify direct statement execution instead of the default, which is prepared statement execution. Using a prepared statement can result in better performance if the statement is executed more than once using parameter binding.

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 optimizes execution of the repeated query. In this 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, then 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
<?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 the Microsoft Drivers for PHP for SQL Server

Retrieving Data

⊕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 the Microsoft Drivers for PHP for SQL Server

Retrieving Data

Retrieving Data as a Stream Using the SQLSRV Driver

10/1/2018 • 2 minutes to read • Edit Online

⊕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

10/1/2018 • 2 minutes to read • Edit Online

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

10/1/2018 • 2 minutes to read • Edit Online

⊕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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⬇ 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
<?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

10/1/2018 • 2 minutes to read • Edit Online

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

10/1/2018 • 2 minutes to read • Edit Online

⊕Download PHP Driver

This topic describes how to use the SQLSRV driver to specify parameter direction when you call a stored procedure. 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 specify the input parameter, explicitly specify the output parameter, and explicitly specify 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

10/1/2018 • 3 minutes to read • Edit Online

⊕Download PHP Driver

This topic demonstrates how to call a stored procedure in which one parameter has been defined as an output parameter. 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 1

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
<?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";
```

```php
        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);
?>
```

Example 2

This code sample shows how to bind a large bigint value as an output parameter.

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

// Assume the stored procedure spTestProcedure exists, which retrieves a bigint value of some large number
// e.g. 9223372036854
$bigintOut = 0;
$outSql = "{CALL spTestProcedure (?)}";
$stmt = sqlsrv_prepare($conn, $outSql, array(array(&$bigintOut, SQLSRV_PARAM_OUT, null,
SQLSRV_SQLTYPE_BIGINT)));
sqlsrv_execute($stmt);
echo "$bigintOut\n";    // Expect 9223372036854

sqlsrv_free_stmt($stmt);
sqlsrv_close($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

10/1/2018 • 3 minutes to read • Edit Online

⬇ 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. 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 1

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
<?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));
```

```php
}

/* 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);
?>
```

> **NOTE**
>
> When binding an input/output parameter to a bigint type, if the value may end up outside the range of an integer, you will need to specify its SQL field type as SQLSRV_SQLTYPE_BIGINT. Otherwise, it may result in a "value out of range" exception.

Example 2

This code sample shows how to bind a large bigint value as an input/output parameter.

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

// Assume the stored procedure spTestProcedure exists, which retrieves a bigint value of some large number
// e.g. 9223372036854
$bigintOut = 0;
$outSql = "{CALL spTestProcedure (?)}";
$stmt = sqlsrv_prepare($conn, $outSql, array(array(&$bigintOut, SQLSRV_PARAM_INOUT, null,
SQLSRV_SQLTYPE_BIGINT)));
sqlsrv_execute($stmt);
echo "$bigintOut\n";    // Expect 9223372036854

sqlsrv_free_stmt($stmt);
sqlsrv_close($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

10/1/2018 • 2 minutes to read • Edit Online

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)

10/1/2018 • 6 minutes to read • Edit Online

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
<?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
<?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)

10/1/2018 • 2 minutes to read • Edit Online

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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕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
<?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
<?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
<?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
<?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)

10/1/2018 • 2 minutes to read • Edit Online

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 the Microsoft Drivers for PHP for SQL Server

⊕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 which correspond to parameter placeholders in the Transact-SQL string. The parameter values in the array must be in the same order as the placeholders meant to represent them.

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
<?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
<?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 the Microsoft Drivers for PHP for SQL Server

About Code Examples in the Documentation

sqlsrv_rows_affected

How to: Send Data as a Stream

10/1/2018 • 3 minutes to read • Edit Online

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 (8 kB) 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: Sending Stream Data at Execution

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: Sending Stream Data Using sqlsrv_send_stream_data

The next example is the same as the preceding example, 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 (8 kB) 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: Sending an Image as a Stream

```php
<?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

10/1/2018 • 3 minutes to read • Edit Online

⬇ 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 is 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
<?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)
{
```

```php
{
    /* 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 checking any call to a **sqlsrv** function for errors and handling them accordingly.

See Also

Updating Data (Microsoft Drivers for PHP for SQL Server)

Transactions (Database Engine)

About Code Examples in the Documentation

Converting Data Types

10/1/2018 • 2 minutes to read • Edit Online

⊕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 are 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.
How to: Send and Retrieve ASCII Data in Linux and macOS	Demonstrates how to use Microsoft Drivers for PHP for SQL Server's support for ASCII data in Linux or macOS. Support for ASCII characters in non-Windows environments was added in version 5.2 of the Microsoft Drivers for PHP for SQL Server.

See Also

Programming Guide for the Microsoft Drivers for PHP for SQL Server

SQLSRV Driver API Reference

Constants (Microsoft Drivers for PHP for SQL Server)

Example Application (SQLSRV Driver)

Default SQL Server Data Types

10/1/2018 • 2 minutes to read • Edit Online

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

10/1/2018 • 2 minutes to read • Edit Online

⬇ 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[4]	Datetime	Not applicable
datetime[4]	Datetime	Not applicable
datetime2[4]	Datetime	Not applicable
datetimeoffset[4]	Datetime	Not applicable
decimal	String	8-bit character[1]
float	Float	8-bit character[1]
geography	STREAM	Binary[3]
geometry	STREAM	Binary[3]
image[5]	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[6]	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[7]	String	8-bit character[1]
text[8]	Stream[2]	8-bit character[1]
time[4]	Datetime	Not applicable
timestamp	String	8-bit character[1]
tinyint	Integer	8-bit character[1]
UDT	Stream[2]	Binary[3]
uniqueidentifier	String[9]	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]
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 is 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 is a binary string.

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

4. 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.

5. This is a legacy type that maps to the varbinary(max) type.

6. This is a legacy type that maps to the nvarchar(max) type.

7. sql_variant is not supported in bidirectional or output parameters.

8. This is a legacy type that maps to the varchar(max) type.

9. 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}

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 following table 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

10/1/2018 • 3 minutes to read • Edit Online

⊕ 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 are 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 subarray 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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕ 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 is 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

10/1/2018 • 4 minutes to read • Edit Online

⊕ 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.

You can pass UTF-8 or SQLSRV_ENC_CHAR to **CharacterSet**, but 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 **nvarchar(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
<?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](#)

[Working with ASCII Data in non-Windows](#)

[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)](#)

How to: Send and Retrieve ASCII Data in Linux and macOS

10/1/2018 • 3 minutes to read • Edit Online

⬇ Download PHP Driver

This article assumes the ASCII (non-UTF-8) locales have been generated or installed in your Linux or macOS systems.

To send or retrieve ASCII character sets to the server:

1. If the desired locale is not the default in your system environment, make sure you invoke `setlocale(LC_ALL, $locale)` before making the first connection. The PHP setlocale() function changes the locale only for the current script, and if invoked after making the first connection, it may be ignored.

2. When using the SQLSRV driver, you may specify `'CharacterSet' => SQLSRV_ENC_CHAR` as a connection option, but this step is optional because it is the default encoding.

3. When using the PDO_SQLSRV driver, there are two ways. First, when making the connection, set `PDO::SQLSRV_ATTR_ENCODING` to `PDO::SQLSRV_ENCODING_SYSTEM` (for an example of setting a connection option, see PDO::__construct). Alternatively, after successfully connected, add this line `$conn->setAttribute(PDO::SQLSRV_ATTR_ENCODING, PDO::SQLSRV_ENCODING_SYSTEM);`

When you specify the encoding of a connection resource (in SQLSRV) or connection object (PDO_SQLSRV), the driver assumes that the other connection option strings use that same encoding. The server name and query strings are also assumed to use the same character set.

The default encoding for PDO_SQLSRV driver is UTF-8 (PDO::SQLSRV_ENCODING_UTF8), unlike the SQLSRV driver. For more information about these constants, see Constants (Microsoft Drivers for PHP for SQL Server).

Example

The following examples demonstrate how to send and retrieve ASCII data using the PHP Drivers for SQL Server by specifying a particular locale before making the connection. The locales in various Linux platforms may be named differently from the same locales in macOS. For example, the US ISO-8859-1 (Latin 1) locale is `en_US.ISO-8859-1` in Linux while in macOS the name is `en_US.ISO8859-1` .

The examples assume that SQL Server is installed on a server. All output is written to the browser when the examples are run from the browser.

```php
<?php

// SQLSRV Example
//
// Setting locale for the script is only necessary if Latin 1 is not the default
// in the environment
$locale = strtoupper(PHP_OS) === 'LINUX' ? 'en_US.ISO-8859-1' : 'en_US.ISO8859-1';
setlocale(LC_ALL, $locale);

$serverName = 'MyServer';
$database = 'Test';
$connectionInfo = array('Database'=>'Test', 'UID'=>$uid, 'PWD'=>$pwd);
$conn = sqlsrv_connect($serverName, $connectionInfo);
```

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

// Set up the Transact-SQL query to create a test table
//
$stmt = sqlsrv_query($conn, "CREATE TABLE [Table1] ([c1_int] int, [c2_varchar] varchar(512))");

// Insert data using a parameter array
//
$tsql = "INSERT INTO [Table1] (c1_int, c2_varchar) VALUES (?, ?)";

// Execute the query, $value being some ASCII string
//
$stmt = sqlsrv_query($conn, $tsql, array(1, array($value, SQLSRV_PARAM_IN,
SQLSRV_PHPTYPE_STRING(SQLSRV_ENC_CHAR))));

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

// Retrieve the newly inserted data
//
$stmt = sqlsrv_query($conn, "SELECT * FROM Table1");
$outValue = null;
if ($stmt === false) {
    echo "Error in statement execution.<br>";
    die(print_r(sqlsrv_errors(), true));
}

// Retrieve and display the data
//
if (sqlsrv_fetch($stmt)) {
    $outValue = sqlsrv_get_field($stmt, 1, SQLSRV_PHPTYPE_STRING(SQLSRV_ENC_CHAR));
    echo "Value: " . $outValue . "<br>";
    if ($value !== $outValue) {
        echo "Data retrieved, \'$outValue\', is unexpected!<br>";
    }
}
else {
    echo "Error in fetching data.<br>";
    die(print_r(sqlsrv_errors(), true));
}

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

```php
<?php

// PDO_SQLSRV Example:
//
// Setting locale for the script is only necessary if Latin 1 is not the default
// in the environment
$locale = strtoupper(PHP_OS) === 'LINUX' ? 'en_US.ISO-8859-1' : 'en_US.ISO8859-1';
setlocale(LC_ALL, $locale);

$serverName = 'MyServer';
$database = 'Test';

try {
    $conn = new PDO("sqlsrv:Server=$serverName;Database=$database;", $uid, $pwd);
    $conn->setAttribute(PDO::SQLSRV_ATTR_ENCODING, PDO::SQLSRV_ENCODING_SYSTEM);

    // Set up the Transact-SQL query to create a test table
    //
    $stmt = $conn->query("CREATE TABLE [Table1] ([c1_int] int, [c2_varchar] varchar(512))");

    // Insert data using parameters, $value being some ASCII string
    //
    $stmt = $conn->prepare("INSERT INTO [Table1] (c1_int, c2_varchar) VALUES (:var1, :var2)");
    $stmt->bindValue(1, 1);
    $stmt->bindParam(2, $value);
    $stmt->execute();

    // Retrieve and display the data
    //
    $stmt = $conn->query("SELECT * FROM [Table1]");
    $outValue = null;
    if ($row = $stmt->fetch()) {
        $outValue = $row[1];
        echo "Value: " . $outValue . "<br>";
        if ($value !== $outValue) {
            echo "Data retrieved, \'$outValue\', is unexpected!<br>";
        }
    }
} catch (PDOException $e) {
    echo $e->getMessage() . "<br>";
} finally {
    // Free statement and connection resources
    //
    unset($stmt);
    unset($conn);
}

?>
```

See Also

Retrieving Data

Working with UTF-8 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

10/1/2018 • 2 minutes to read • Edit Online

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 the Microsoft Drivers for PHP for SQL Server

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

10/1/2018 • 2 minutes to read • Edit Online

⊕ 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 returns **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 resets 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 turns 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
<?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 is 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 the Microsoft Drivers for PHP for SQL Server

SQLSRV Driver API Reference

How to: Handle Errors and Warnings Using the SQLSRV Driver

10/1/2018 • 4 minutes to read • Edit Online

⊕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 returns **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.

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
<?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
exit the script. */
```

```php
    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);
```

```php
$emp_hrs = array (7 /4, 5 /5, 9 /6, 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);
```

```php
        $warnings = sqlsrv_errors(SQLSRV_ERR_WARNINGS);
        if(!is_null($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

10/1/2018 • 3 minutes to read • Edit Online

⊕ 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 are logged.
1	Specifies that errors are logged.
2	Specifies that warnings are logged.
4	Specifies that notices are logged.

Logging information is 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 on logging, 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 on logging 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 on logging 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 turns 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 turns 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 are 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 are logged.
SQLSRV_LOG_SEVERITY_ERROR (1)	Specifies that errors are logged. This is the default.

VALUE (INTEGER EQUIVALENT IN PARENTHESES)	DESCRIPTION
SQLSRV_LOG_SEVERITY_WARNING (2)	Specifies that warnings are logged.
SQLSRV_LOG_SEVERITY_NOTICE (4)	Specifies that notices are 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 on logging. You must turn on logging 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 enables logging of errors and warnings:

```
sqlsrv.LogSeverity = 3
```

See Also

Programming Guide for the Microsoft Drivers for PHP for SQL Server

Constants (Microsoft Drivers for PHP for SQL Server)

sqlsrv_configure

sqlsrv_get_config

SQLSRV Driver API Reference

Using Always Encrypted with the PHP Drivers for SQL Server

10/1/2018 • 16 minutes to read • Edit Online

⊕ Download PHP Driver

Applicable to

- Microsoft Drivers 5.2 for PHP for SQL Server

Introduction

This article provides information on how to develop PHP applications using Always Encrypted (Database Engine) and the PHP Drivers for SQL Server.

Always Encrypted allows client applications to encrypt sensitive data and never reveal the data or the encryption keys to SQL Server or Azure SQL Database. An Always Encrypted enabled driver, such as the ODBC Driver for SQL Server, transparently encrypts and decrypts sensitive data in the client application. The driver automatically determines which query parameters correspond to sensitive database columns (protected using Always Encrypted), and encrypts the values of those parameters before passing the data to SQL Server or Azure SQL Database. Similarly, the driver transparently decrypts data retrieved from encrypted database columns in query results. For more information, see Always Encrypted (Database Engine). The PHP Drivers for SQL Server utilize the ODBC Driver for SQL Server to encrypt sensitive data.

Prerequisites

- Configure Always Encrypted in your database. This configuration involves provisioning Always Encrypted keys and setting up encryption for selected database columns. If you do not already have a database with Always Encrypted configured, follow the directions in Getting Started with Always Encrypted. In particular, your database should contain the metadata definitions for a Column Master Key (CMK), a Column Encryption Key (CEK), and a table containing one or more columns encrypted using that CEK.
- Make sure ODBC Driver for SQL Server version 17 or higher is installed on your development machine. For details, see ODBC Driver for SQL Server.

Enabling Always Encrypted in a PHP Application

The easiest way to enable the encryption of parameters targeting the encrypted columns and the decryption of query results is by setting the value of the `ColumnEncryption` connection string keyword to `Enabled`. The following are examples of enabling Always Encrypted in the SQLSRV and PDO_SQLSRV drivers:

SQLSRV:

```
$connectionInfo = array("Database"=>$databaseName, "UID"=>$uid, "PWD"=>$pwd, "ColumnEncryption"=>"Enabled");
$conn = sqlsrv_connect($server, $connectionInfo);
```

PDO_SQLSRV:

```
$connectionInfo = "Database = $databaseName; ColumnEncryption = Enabled;";
$conn = new PDO("sqlsrv:server = $server; $connectionInfo", $uid, $pwd);
```

Enabling Always Encrypted is not sufficient for encryption or decryption to succeed; you also need to make sure that:

- The application has the VIEW ANY COLUMN MASTER KEY DEFINITION and VIEW ANY COLUMN ENCRYPTION KEY DEFINITION database permissions, required to access the metadata about Always Encrypted keys in the database. For details, see Database Permission.
- The application can access the CMK that protects the CEKs for the queried encrypted columns. This requirement is dependent on the key store provider that stores the CMK. For more information, see Working with Column Master Key Stores.

Retrieving and Modifying Data in Encrypted Columns

Once you enable Always Encrypted on a connection, you can use standard SQLSRV APIs (see SQLSRV Driver API Reference) or PDO_SQLSRV APIs (see PDO_SQLSRV Driver API Reference) to retrieve or modify data in encrypted database columns. Assuming your application has the required database permissions and can access the column master key, the driver encrypts any query parameters that target encrypted columns and decrypt data retrieved from encrypted columns, behaving transparently to the application as if the columns were not encrypted.

If Always Encrypted is not enabled, queries with parameters that target encrypted columns fail. Data can still be retrieved from encrypted columns, as long as the query has no parameters targeting encrypted columns. However, the driver does not attempt any decryption and the application receives the binary encrypted data (as byte arrays).

The following table summarizes the behavior of queries, depending on whether Always Encrypted is enabled or not:

QUERY CHARACTERISTIC	ALWAYS ENCRYPTED IS ENABLED AND APPLICATION CAN ACCESS THE KEYS AND KEY METADATA	ALWAYS ENCRYPTED IS ENABLED AND APPLICATION CANNOT ACCESS THE KEYS OR KEY METADATA	ALWAYS ENCRYPTED IS DISABLED
Parameters targeting encrypted columns.	Parameter values are transparently encrypted.	Error	Error
Retrieving data from encrypted columns, without parameters targeting encrypted columns.	Results from encrypted columns are transparently decrypted. The application receives plaintext column values.	Error	Results from encrypted columns are not decrypted. The application receives encrypted values as byte arrays.

The following examples illustrate retrieving and modifying data in encrypted columns. The examples assume a table with the following schema. The SSN and BirthDate columns are encrypted.

```
CREATE TABLE [dbo].[Patients](
 [PatientId] [int] IDENTITY(1,1),
 [SSN] [char](11) COLLATE Latin1_General_BIN2
 ENCRYPTED WITH (ENCRYPTION_TYPE = DETERMINISTIC,
 ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256',
 COLUMN_ENCRYPTION_KEY = CEK1) NOT NULL,
 [FirstName] [nvarchar](50) NULL,
 [LastName] [nvarchar](50) NULL,
 [BirthDate] [date]
 ENCRYPTED WITH (ENCRYPTION_TYPE = RANDOMIZED,
 ALGORITHM = 'AEAD_AES_256_CBC_HMAC_SHA_256',
 COLUMN_ENCRYPTION_KEY = CEK1) NOT NULL
 PRIMARY KEY CLUSTERED ([PatientId] ASC) ON [PRIMARY])
 GO
```

Data Insertion Example

The following examples demonstrate how to use the SQLSRV and PDO_SQLSRV drivers to insert a row into the Patient table. Note the following points:

- There is nothing specific to encryption in the sample code. The driver automatically detects and encrypts the values of the SSN and BirthDate parameters, which target encrypted columns. This mechanism makes encryption transparent to the application.

- The values inserted into database columns, including the encrypted columns, are passed as bound parameters. While using parameters is optional when sending values to non-encrypted columns (although it is highly recommended because it helps prevent SQL injection), it is required for values targeting encrypted columns. If the values inserted in the SSN or BirthDate columns were passed as literals embedded in the query statement, the query would fail because the driver does not attempt to encrypt or otherwise process literals in queries. As a result, the server would reject them as incompatible with the encrypted columns.

- When inserting values using bind parameters, a SQL type that is identical to the data type of the target column or whose conversion to the data type of the target column is supported must be passed to the database. This requirement is because Always Encrypted supports few type conversions (for details, see Always Encrypted (Database Engine)). The two PHP drivers, SQLSRV and PDO_SQLSRV, each has a mechanism to help the user determine the SQL type of the value. Therefore, the user does not have to provide the SQL type explicitly.
  - For the SQLSRV driver, the user has two options:
  - Rely on the PHP driver to determine and set the right SQL type. In this case, the user must use `sqlsrv_prepare` and `sqlsrv_execute` to execute a parameterized query.
  - Set the SQL type explicitly.
  - For the PDO_SQLSRV driver, the user does not have the option to explicitly set the SQL type of a parameter. The PDO_SQLSRV driver automatically helps the user determine the SQL type when binding a parameter.

- For the drivers to determine the SQL type, some limitations apply:
  - SQLSRV Driver:
  - If the user wants the driver to determine the SQL types for the encrypted columns, the user must use `sqlsrv_prepare` and `sqlsrv_execute`.
  - If `sqlsrv_query` is preferred, the user is responsible for specifying the SQL types for all parameters. The specified SQL type must include the string length for string types, and the scale and precision for decimal types.
  - PDO_SQLSRV Driver:
  - The statement attribute `PDO::SQLSRV_ATTR_DIRECT_QUERY` is not supported in a parameterized query.
  - The statement attribute `PDO::ATTR_EMULATE_PREPARES` is not supported in a parameterized query.

SQLSRV driver and sqlsrv_prepare:

```
    // insertion into encrypted columns must use a parameterized query
    $query = "INSERT INTO [dbo].[Patients] ([SSN], [FirstName], [LastName], [BirthDate]) VALUES (?, ?, ?, ?)";
    $ssn = "795-73-9838";
    $firstName = "Catherine";
    $lastName = "Abel;
    $birthDate = "1996-10-19";
    $params = array($ssn, $firstName, $lastName, $birthDate);
    // during sqlsrv_prepare, the driver determines the SQL types for each parameter and pass them to SQL server
    $stmt = sqlsrv_prepare($conn, $query, $params);
    sqlsrv_execute($stmt);
```

SQLSRV driver and sqlsrv_query:

```
    // insertion into encrypted columns must use a parameterized query
    $query = "INSERT INTO [dbo].[Patients] ([SSN], [FirstName], [LastName], [BirthDate]) VALUES (?, ?, ?, ?)";
    $ssn = "795-73-9838";
    $firstName = "Catherine";
    $lastName = "Abel";
    $birthDate = "1996-10-19";
    // need to provide the SQL types for ALL parameters
    // note the SQL types (including the string length) have to be the same at the column definition
    $params = array(array(&$ssn, null, null, SQLSRV_SQLTYPE_CHAR(11)),
                    array(&$firstName, null, null, SQLSRV_SQLTYPE_NVARCHAR(50)),
                    array(&$lastName, null, null, SQLSRV_SQLTYPE_NVARCHAR(50)),
                    array(&$birthDate, null, null, SQLSRV_SQLTYPE_DATE));
    sqlsrv_query($conn, $query, $params);
```

PDO_SQLSRV driver and PDO::prepare:

```
    // insertion into encrypted columns must use a parameterized query
    $query = "INSERT INTO [dbo].[Patients] ([SSN], [FirstName], [LastName], [BirthDate]) VALUES (?, ?, ?, ?)";
    $ssn = "795-73-9838";
    $firstName = "Catherine";
    $lastName = "Able";
    $birthDate = "1996-10-19";
    // during PDO::prepare, the driver determines the SQL types for each parameter and pass them to SQL server
    $stmt = $conn->prepare($query);
    $stmt->bindParam(1, $ssn);
    $stmt->bindParam(2, $firstName);
    $stmt->bindParam(3, $lastName);
    $stmt->bindParam(4, $birthDate);
    $stmt->execute();
```

Plaintext Data Retrieval Example

The following examples demonstrate filtering data based on encrypted values, and retrieving plaintext data from encrypted columns using the SQLSRV and PDO_SQLSRV drivers. Note the following points:

- The value used in the WHERE clause to filter on the SSN column needs to be passed using bind parameter, so that the driver can transparently encrypt it before sending it to the server.

- When executing a query with bound parameters, the PHP drivers automatically determines the SQL type for the user unless the user explicitly specifies the SQL type when using the SQLSRV driver.

- All values printed by the program are in plaintext, since the driver transparently decrypts the data retrieved from the SSN and BirthDate columns.

Note: Queries can perform equality comparisons on encrypted columns only if the encryption is deterministic. For more information, see Selecting Deterministic or Randomized encryption.

SQLSRV:

```
    // since SSN is an encrypted column, need to pass the value in the WHERE clause through bind parameter
    $query = "SELECT [SSN], [FirstName], [LastName], [BirthDate] FROM [dbo].[Patients] WHERE [SSN] = ?";
    $ssn = "795-73-9838";
    $stmt = sqlsrv_prepare($conn, $query, array(&$ssn));
    // during sqlsrv_execute, the driver encrypts the ssn value and passes it to the database
    sqlsrv_execute($stmt);
    // fetch like usual
    $row = sqlsrv_fetch_array($stmt);
```

PDO_SQLSRV:

```
    // since SSN is an encrypted column, need to pass the value in the WHERE clause through bind parameter
    $query = "SELET [SSN], [FirstName], [LastName], [BirthDate] FROM [dbo].[Patients] WHERE [SSN] = ?";
    $ssn = "795-73-9838";
    $stmt = $conn->prepare($query);
    $stmt->bindParam(1, $ssn);
    // during PDOStatement::execute, the driver encrypts the ssn value and passes it to the database
    $stmt->execute();
    // fetch like usual
    $row = $stmt->fetch();
```

Ciphertext Data Retrieval Example

If Always Encrypted is not enabled, a query can still retrieve data from encrypted columns, as long as the query has no parameters targeting encrypted columns.

The following examples illustrate retrieving binary encrypted data from encrypted columns using the SQLSRV and PDO_SQLSRV drivers. Note the following points:

- As Always Encrypted is not enabled in the connection string, the query returns encrypted values of SSN and BirthDate as byte arrays (the program converts the values to strings).
- A query retrieving data from encrypted columns with Always Encrypted disabled can have parameters, as long as none of the parameters target an encrypted column. The following query filters by LastName, which is not encrypted in the database. If the query filters by SSN or BirthDate, the query would fail.

SQLSRV:

```
    $query = "SELET [SSN], [FirstName], [LastName], [BirthDate] FROM [dbo].[Patients] WHERE [LastName] = ?";
    $lastName = "Abel";
    $stmt = sqlsrv_prepare($conn, $query, array(&$lastName));
    sqlsrv_execute($stmt);
    $row = sqlsrv_fetch_array($stmt);
```

PDO_SQLSRV:

```
    $query = "SELET [SSN], [FirstName], [LastName], [BirthDate] FROM [dbo].[Patients] WHERE [LastName] = ?";
    $lastName = "Abel";
    $stmt = $conn->prepare($query);
    $stmt->bindParam(1, $lastName);
    $stmt->execute();
    $row = $stmt->fetch();
```

Avoiding Common Problems when Querying Encrypted Columns

This section describes common categories of errors when querying encrypted columns from PHP applications and a few guidelines on how to avoid them.

Unsupported data type conversion errors

Always Encrypted supports few conversions for encrypted data types. See Always Encrypted (Database Engine)

for the detailed list of supported type conversions. Do the following to avoid data type conversion errors:

- When using the SQLSRV driver with `sqlsrv_prepare` and `sqlsrv_execute` the SQL type, along with the column size and the number of decimal digits of the parameter is automatically determined.
- When using the PDO_SQLSRV driver to execute a query, the SQL type with the column size and the number of decimal digits of the parameter is also automatically determined
- When using the SQLSRV driver with `sqlsrv_query` to execute a query:
  - The SQL type of the parameter is either exactly the same as the type of the targeted column, or the conversion from the SQL type to the type of the column is supported.
  - The precision and scale of the parameters targeting columns of the `decimal` and `numeric` SQL Server data types is the same as the precision and scale configure for the target column.
  - The precision of parameters targeting columns of `datetime2`, `datetimeoffset`, or `time` SQL Server data types is not greater than the precision for the target column, in queries that modify the target column.
- Do not use PDO_SQLSRV statement attributes `PDO::SQLSRV_ATTR_DIRECT_QUERY` or `PDO::ATTR_EMULATE_PREPARES` in a parameterized query

Errors due to passing plaintext instead of encrypted values

Any value that targets an encrypted column needs to be encrypted before being sent to the server. An attempt to insert, modify, or filter by a plaintext value on an encrypted column results in an error. To prevent such errors, make sure that:

- Always Encrypted is enabled (in the connection string, set the `ColumnEncryption` keyword to `Enabled`).
- You use bind parameter to send data targeting encrypted columns. The following example shows a query that incorrectly filters by a literal/constant on an encrypted column (SSN):

```
$query = "SELET [SSN], [FirstName], [LastName], [BirthDate] FROM [dbo].[Patients] WHERE SSN='795-73-9838'";
```

Controlling Performance Impact of Always Encrypted

Because Always Encrypted is a client-side encryption technology, most of the performance overhead is observed on the client side, not in the database. Apart from the cost of encryption and decryption operations, the other sources of performance overhead on the client side are:

- Additional round-trips to the database to retrieve metadata for query parameters.
- Calls to a column master key store to access a column master key.

Round-trips to Retrieve Metadata for Query Parameters

If Always Encrypted is enabled for a connection, the ODBC Driver will, by default, call sys.sp_describe_parameter_encryption for each parameterized query, passing the query statement (without any parameter values) to SQL Server. This stored procedure analyzes the query statement to find out if any parameters need to be encrypted, and if so, returns the encryption-related information for each parameter to allow the driver to encrypt them.

Since the PHP drivers allow the user to bind a parameter in a prepared statement without providing the SQL type, when binding a parameter in an Always Encrypted enabled connection, the PHP Drivers call SQLDescribeParam on the parameter to get the SQL type, column size, and decimal digits. The metadata is then used to call SQLBindParameter. These extra `SQLDescribeParam` calls do not require extra round-trips to the database as the ODBC Driver has already stored the information on the client side when `sys.sp_describe_parameter_encryption` was called.

The preceding behaviors ensure a high level of transparency to the client application (and the application developer) does not need to be aware of which queries access encrypted columns, as long as the values targeting encrypted columns are passed to the driver in parameters.

Unlike the ODBC Driver for SQL Server, enabling Always Encrypted at the statement/query-level is not yet supported in the PHP drivers.

Column Encryption Key Caching

To reduce the number of calls to a column master key store to decrypt column encryption keys (CEK), the driver caches the plaintext CEKs in memory. After receiving the encrypted CEK (ECEK) from database metadata, the ODBC driver first tries to find the plaintext CEK corresponding to the encrypted key value in the cache. The driver calls the key store containing the CMK only if it cannot find the corresponding plaintext CEK in the cache.

Note: In the ODBC Driver for SQL Server, the entries in the cache are evicted after a two-hour timeout. This behavior means that for a given ECEK, the driver contacts the key store only once during the lifetime of the application or every two hours, whichever is less.

Working with Column Master Key Stores

To encrypt or decrypt data, the driver needs to obtain a CEK that is configured for the target column. CEKs are stored in encrypted form (ECEKs) in the database metadata. Each CEK has a corresponding CMK that was used to encrypt it. The database metadata does not store the CMK itself; it only contains the name of the key store and information that the key store can use to locate the CMK.

To obtain the plaintext value of an ECEK, the driver first obtains the metadata about both the CEK and its corresponding CMK, and then it uses this information to contact the key store containing the CMK and requests it to decrypt the ECEK. The driver communicates with a key store using a key store provider.

For Microsoft Driver 5.3.0 for PHP for SQL Server, only Windows Certificate Store Provider and Azure Key Vault are supported. The other Keystore Provider supported by the ODBC Driver (Custom Keystore Provider) is not yet supported.

Using the Windows Certificate Store Provider

The ODBC Driver for SQL Server on Windows includes a built-in column master key store provider for the Windows Certificate Store, named `MSSQL_CERTIFICATE_STORE`. (This provider is not available on macOS or Linux.) With this provider, the CMK is stored locally on the client machine and no additional configuration by the application is necessary to use it with the driver. However, the application must have access to the certificate and its private key in the store. For more information, see Create and Store Column Master Keys (Always Encrypted).

Using Azure Key Vault

Azure Key Vault offers a way to store encryption keys, passwords, and other secrets using Azure and can be used to store keys for Always Encrypted. The ODBC Driver for SQL Server (version 17 and higher) includes a built-in master key store provider for Azure Key Vault. The following connection options handle Azure Key Vault configuration: `KeyStoreAuthentication`, `KeyStorePrincipalId`, and `KeyStoreSecret`.

- `KeyStoreAuthentication` can take one of two possible string values: `KeyVaultPassword` and `KeyVaultClientSecret`. These values control what kind of authentication credentials are used with the other two keywords.
- `KeyStorePrincipalId` takes a string representing an identifier for the account seeking to access the Azure Key Vault.
  - If `KeyStoreAuthentication` is set to `KeyVaultPassword`, then `KeyStorePrincipalId` must be the name of an Azure ActiveDirectory user.
  - If `KeyStoreAuthentication` is set to `KeyVaultClientSecret`, then `KeyStorePrincipalId` must be an application client ID.
- `KeyStoreSecret` takes a string representing a credential secret.
  - If `KeyStoreAuthentication` is set to `KeyVaultPassword`, then `KeyStoreSecret` must be the user's password.
  - If `KeyStoreAuthentication` is set to `KeyVaultClientSecret`, then `KeyStoreSecret` must be the application

secret associated with the application client ID.

All three options must be present in the connection string to use Azure Key Vault. In addition, `ColumnEncryption` must be set to `Enabled`. If `ColumnEncryption` is set to `Disabled` but the Azure Key Vault options are present, the script will proceed without errors but no encryption will be performed.

The following examples show how to connect to SQL Server using Azure Key Vault.

SQLSRV:

Using an Azure Active Directory account:

```
$connectionInfo = array("Database"=>$databaseName, "UID"=>$uid, "PWD"=>$pwd, "ColumnEncryption"=>"Enabled",
"KeyStoreAuthentication"=>"KeyVaultPassword", "KeyStorePrincipalId"=>$AADUsername,
"KeyStoreAuthentication"=>$AADPassword);
$conn = sqlsrv_connect($server, $connectionInfo);
```

Using an Azure application client ID and secret:

```
$connectionInfo = array("Database"=>$databaseName, "UID"=>$uid, "PWD"=>$pwd, "ColumnEncryption"=>"Enabled",
"KeyStoreAuthentication"=>"KeyVaultClientSecret", "KeyStorePrincipalId"=>$applicationClientID,
"KeyStoreAuthentication"=>$applicationClientSecret);
$conn = sqlsrv_connect($server, $connectionInfo);
```

PDO_SQLSRV: Using an Azure Active Directory account:

```
$connectionInfo = "Database = $databaseName; ColumnEncryption = Enabled; KeyStoreAuthentication =
KeyVaultPassword; KeyStorePrincipalId = $AADUsername; KeyStoreAuthentication = $AADPassword;";
$conn = new PDO("sqlsrv:server = $server; $connectionInfo", $uid, $pwd);
```

Using an Azure application client ID and secret:

```
$connectionInfo = "Database = $databaseName; ColumnEncryption = Enabled; KeyStoreAuthentication =
KeyVaultClientSecret; KeyStorePrincipalId = $applicationClientID; KeyStoreAuthentication =
$applicationClientSecret;";
$conn = new PDO("sqlsrv:server = $server; $connectionInfo", $uid, $pwd);
```

Limitations of the PHP drivers when using Always Encrypted

SQLSRV and PDO_SQLSRV:

- Linux/macOS do not support Windows Certificate Store Provider
- Forcing parameter encryption
- Enabling Always Encrypted at the statement level
- When using the Always Encrypted feature and non-UTF8 locales on Linux and macOS (such as "en_US.ISO-8859-1"), inserting null data or an empty string into an encrypted char(n) column may not work unless Code Page 1252 has been installed on your system

SQLSRV only:

- Using `sqlsrv_query` for binding parameter without specifying the SQL type
- Using `sqlsrv_prepare` for binding parameters in a batch of SQL statements

PDO_SQLSRV only:

- `PDO::SQLSRV_ATTR_DIRECT_QUERY` statement attribute specified in a parameterized query

- `PDO::ATTR_EMULATE_PREPARE` statement attribute specified in a parameterized query
- binding parameters in a batch of SQL statements

The PHP drivers also inherit the limitations imposed by the ODBC Driver for SQL Server and the database. See Limitations of the ODBC driver when using Always Encrypted and Always Encrypted Feature Details.

See Also

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

Constants (Microsoft Drivers for PHP for SQL Server)

10/1/2018 • 6 minutes to read • Edit Online

⬇ 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 are represented as ints, while SQL floats and reals are 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 CONSTANT	DESCRIPTION
SQLSRV_FETCH_ASSOC	**sqlsrv_fetch_array** returns the next row of data as an associative array.
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 CONSTANT (INTEGER EQUIVALENT IN PARENTHESES)	DESCRIPTION
SQLSRV_NULLABLE_NO (1)	The column is not nullable.
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.

SQLSRV CONSTANT	DESCRIPTION
"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 CONSTANT	SQL SERVER DATA TYPE
SQLSRV_SQLTYPE_MONEY	money
SQLSRV_SQLTYPE_NCHAR	nchar[5]
SQLSRV_SQLTYPE_NCHAR($charCount)	nchar
SQLSRV_SQLTYPE_NUMERIC	numeric[5]
SQLSRV_SQLTYPE_NUMERIC($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 CONSTANT	SQL SERVER DATA TYPE
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

⬇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 Microsoft Drivers for PHP for SQL Server

Constants (Microsoft Drivers for PHP for SQL Server)

Programming Guide for the Microsoft Drivers for PHP for SQL Server

Getting Started with the Microsoft Drivers for PHP for SQL Server

sqlsrv_begin_transaction

10/1/2018 • 2 minutes to read • Edit Online

⊕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 Microsoft Drivers for PHP for SQL Server

sqlsrv_cancel

10/1/2018 • 2 minutes to read • Edit Online

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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕ 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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕ 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
<?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

10/1/2018 • 3 minutes to read • Edit Online

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
<?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 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 Microsoft Drivers for PHP for SQL Server

sqlsrv_configure

10/1/2018 • 2 minutes to read • Edit Online

⊕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 the table below for a 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 and negative numbers are not allowed.	10240 KB
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 the Microsoft Drivers for PHP for SQL Server

sqlsrv_connect

10/1/2018 • 2 minutes to read • Edit Online

⬇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 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 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

10/1/2018 • 3 minutes to read • Edit Online

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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕ 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
<?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

10/1/2018 • 2 minutes to read • Edit Online

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
<?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

10/1/2018 • 4 minutes to read • Edit Online

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
<?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 the Microsoft Drivers for PHP for SQL Server

sqlsrv_fetch_object

10/1/2018 • 7 minutes to read • Edit Online

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
<?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
<?php
/* Define the Product class. */
class Product
{
     /* Constructor */
     public function Product($ID)
     {
         $this->objID = $ID;
     }
```

```php
        }
        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

10/1/2018 • 3 minutes to read • Edit Online

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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕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
<?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

10/1/2018 • 2 minutes to read • Edit Online

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

10/1/2018 • 2 minutes to read • Edit Online

⬇ 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
<?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

10/1/2018 • 2 minutes to read • Edit Online

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
<?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

10/1/2018 • 5 minutes to read • Edit Online

⊕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
<?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));
}
```

```php
/* 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);
```

```php
$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
<?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 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

10/1/2018 • 2 minutes to read • Edit Online

Retrieves the number of fields in an active result set. This function 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
<?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

10/1/2018 • 2 minutes to read • Edit Online

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
<?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
<?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

10/1/2018 • 6 minutes to read • Edit Online

⬇ 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 is 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 following table 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 waits 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
<?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
<?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));
}
```

```php
/* 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);
?>
```

> **NOTE**
>
> It is recommended to use strings as inputs when binding values to a decimal or numeric column to ensure precision and accuracy as PHP has limited precision for floating point numbers. The same applies to bigint columns, especially when the values are outside the range of an integer.

Example

This code sample shows how to bind a decimal value as an input parameter.

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

// Assume TestTable exists with a decimal field
$input = "9223372036854.80000";
$params = array($input);
$stmt = sqlsrv_prepare($conn, "INSERT INTO TestTable (DecimalCol) VALUES (?)", $params);
sqlsrv_execute($stmt);

sqlsrv_free_stmt($stmt);
sqlsrv_close($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

10/1/2018 • 5 minutes to read • Edit Online

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 following table:

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 waits 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
<?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 following example 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
<?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);
?>
```

> **NOTE**
>
> It is recommended to use strings as inputs when binding values to a decimal or numeric column to ensure precision and accuracy as PHP has limited precision for floating point numbers. The same applies to bigint columns, especially when the values are outside the range of an integer.

Example

This code sample shows how to bind a decimal value as an input parameter.

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

// Assume TestTable exists with a decimal field
$input = "9223372036854.80000";
$params = array($input);
$stmt = sqlsrv_query($conn, "INSERT INTO TestTable (DecimalCol) VALUES (?)", $params);

sqlsrv_free_stmt($stmt);
sqlsrv_close($conn);

?>
```

Example

This code sample shows how to create a table of sql_variant types and fetch the inserted data.

```php
<?php
$server = 'serverName';
$dbName = 'databaseName';
$uid = 'yourUserName';
$pwd = 'yourPassword';

$options = array("Database"=>$dbName, "UID"=>$uid, "PWD"=>$pwd);
$conn = sqlsrv_connect($server, $options);
if($conn === false) {
    die(print_r(sqlsrv_errors(), true));
}

$tableName = 'testTable';
$query = "CREATE TABLE $tableName ([c1_int] sql_variant, [c2_varchar] sql_variant)";

$stmt = sqlsrv_query($conn, $query);
if($stmt === false) {
    die(print_r(sqlsrv_errors(), true));
}
sqlsrv_free_stmt($stmt);

$query = "INSERT INTO [$tableName] (c1_int, c2_varchar) VALUES (1, 'test_data')";
$stmt = sqlsrv_query($conn, $query);
if($stmt === false) {
    die(print_r(sqlsrv_errors(), true));
}
sqlsrv_free_stmt($stmt);

$query = "SELECT * FROM $tableName";
$stmt = sqlsrv_query($conn, $query);

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

$col1 = sqlsrv_get_field($stmt, 0);
echo "First field:  $col1 \n";

$col2 = sqlsrv_get_field($stmt, 1);
echo "Second field:  $col2 \n";

sqlsrv_free_stmt($stmt);
sqlsrv_close($conn);

?>
```

The expected output would be:

```
First field:  1
Second field:  test_data
```

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

10/1/2018 • 3 minutes to read • Edit Online

⬇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 Microsoft Drivers for PHP for SQL Server

sqlsrv_rows_affected

10/1/2018 • 2 minutes to read • Edit Online

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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕ 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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕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 Microsoft Drivers for PHP for SQL Server

Constants (Microsoft Drivers for PHP for SQL Server)

Programming Guide for the Microsoft Drivers for PHP for SQL Server

Getting Started with the Microsoft Drivers for PHP for SQL Server

PDO Class

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 Microsoft Drivers for PHP for SQL Server

Constants (Microsoft Drivers for PHP for SQL Server)

Programming Guide for the Microsoft Drivers for PHP for SQL Server

Getting Started with the Microsoft Drivers for PHP for SQL Server

PDO

PDO::__construct

⬇ 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 does 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 displays 01000 instead of 00000.

If PDO::__construct fails for any reason, an exception is 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
<?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
<?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

10/1/2018 • 2 minutes to read • Edit Online

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 ends 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 returns 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
<?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

⊕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

⊕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 returns warnings on some successful operations. For example, on a successful connection, PDO::errorCode returns "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 an error is generated on the statement object, PDO::errorCode does 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

10/1/2018 • 2 minutes to read • Edit Online

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, then the driver-specific fields are 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
<?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

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

For more information, see PDO::setAttribute.

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
<?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

10/1/2018 • 3 minutes to read • Edit Online

⊕ 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_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_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 10,240 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_ENCOD ING	Microsoft Drivers for PHP for SQL Server	PDO::SQLSRV_ENCODING_U TF8 PDO::SQLSRV_ENCODING_S YSTEM	Specifies the character set encoding used by the driver to communicate with the server. The default is PDO::SQLSRV_ENCODING_U TF8.
PDO::SQLSRV_ATTR_FETCHE S_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_NU MERIC_TYPE is on, the return value is a string. When the returned PDO type in bind column is PDO_PARAM_INT, the return value from an integer column is an int even if SQLSRV_ATTR_FETCHES_NU MERIC_TYPE is off.
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 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 returns 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
<?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

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
<?php
print_r(PDO::getAvailableDrivers());
?>
```

See Also

PDO Class

PDO

PDO::lastInsertId

⊕ 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. If a sequence name is provided, `lastInsertId` returns the most recently inserted sequence number for the provided sequence name (for more information about sequence numbers, see here).

Syntax

```
string PDO::lastInsertId ([ $name = NULL ] );
```

Parameters

$*name*: An optional string that lets you specify a sequence name.

Return Value

If no sequence name is provided, a string of the identifier for the row most recently added. If a sequence name is provided, a string of the identifier for the sequence most recently added. If the method call fails, empty string is returned.

Remarks

Support for PDO was added in version 2.0 of the Microsoft Drivers for PHP for SQL Server.
Between version 2.0 and 4.3, the optional parameter is a table name, and the return value is the ID of the row most recently added to the provided table. Starting with 5.0, the optional parameter is regarded as a sequence name, and the return value is the sequence most recently added for the provided sequence name. If a table name is provided for versions after 4.3, `lastInsertId` returns an empty string. Sequences are supported only in SQL Server 2012 and above.

Example

```php
<?php
$server = "myserver";
$databaseName = "mydatabase";
$uid = "myusername";
$pwd = "mypasword";

try {
    $conn = new PDO("sqlsrv:Server=$server;Database=$databaseName", $uid, $pwd);

    // One sequence, two tables
    $tableName1 = 'seqtable1';
    $tableName2 = 'seqtable2';
    $sequenceName = 'sequence1';

    $stmt = $conn->query("IF OBJECT_ID('$sequenceName', 'SO') IS NOT NULL DROP SEQUENCE $sequenceName");
    $sql = "CREATE TABLE $tableName1 (seqnum INTEGER NOT NULL PRIMARY KEY, SomeNumber INT)";
    $stmt = $conn->query($sql);
    $sql = "CREATE TABLE $tableName2 (ID INT IDENTITY(1,2), SomeValue char(10))";
    $stmt = $conn->query($sql);
```

```php
    $sql = "CREATE SEQUENCE $sequenceName AS INTEGER START WITH 1 INCREMENT BY 1 MINVALUE 1 MAXVALUE 100
CYCLE";
    $stmt = $conn->query($sql);

    $ret = $conn->exec("INSERT INTO $tableName1 VALUES( NEXT VALUE FOR $sequenceName, 20)");
    $ret = $conn->exec("INSERT INTO $tableName1 VALUES( NEXT VALUE FOR $sequenceName, 40)");
    $ret = $conn->exec("INSERT INTO $tableName1 VALUES( NEXT VALUE FOR $sequenceName, 60)");
    $ret = $conn->exec("INSERT INTO $tableName2 VALUES( '20' )");

    // return the last sequence number if sequence name is provided
    $lastSeq1 = $conn->lastInsertId($sequenceName);

    // defaults to $tableName2 -- because it returns the last inserted id value
    $lastRow = $conn->lastInsertId();

    // providing a table name in lastInsertId should return an empty string
    $lastSeq2 = $conn->lastInsertId($tableName2);

    echo "Last sequence number = $lastSeq1\n";
    echo "Last inserted ID     = $lastRow\n";
    echo "Last inserted ID when a table name is supplied = $lastSeq2\n";

    // One table, two sequences
    $tableName = 'seqtable';
    $sequence1 = 'sequence1';
    $sequence2 = 'sequenceNeg1';
    $stmt = $conn->query("IF OBJECT_ID('$sequence1', 'SO') IS NOT NULL DROP SEQUENCE $sequence1");
    $stmt = $conn->query("IF OBJECT_ID('$sequence2', 'SO') IS NOT NULL DROP SEQUENCE $sequence2");
    $sql = "CREATE TABLE $tableName (ID INT IDENTITY(1,1), SeqNumInc INTEGER NOT NULL PRIMARY KEY, SomeNumber
INT)";
    $stmt = $conn->query($sql);
    $sql = "CREATE SEQUENCE $sequence1 AS INTEGER START WITH 1 INCREMENT BY 1 MINVALUE 1 MAXVALUE 100";
    $stmt = $conn->query($sql);

    $sql = "CREATE SEQUENCE $sequence2 AS INTEGER START WITH 200 INCREMENT BY -1 MINVALUE 101 MAXVALUE 200";
    $stmt = $conn->query($sql);
    $ret = $conn->exec("INSERT INTO $tableName VALUES( NEXT VALUE FOR $sequence1, 20 )");
    $ret = $conn->exec("INSERT INTO $tableName VALUES( NEXT VALUE FOR $sequence2, 180 )");
    $ret = $conn->exec("INSERT INTO $tableName VALUES( NEXT VALUE FOR $sequence1, 40 )");
    $ret = $conn->exec("INSERT INTO $tableName VALUES( NEXT VALUE FOR $sequence2, 160 )");
    $ret = $conn->exec("INSERT INTO $tableName VALUES( NEXT VALUE FOR $sequence1, 60 )");
    $ret = $conn->exec("INSERT INTO $tableName VALUES( NEXT VALUE FOR $sequence2, 140 )");

    // return the last sequence number of 'sequence1'
    $lastSeq1 = $conn->lastInsertId($sequence1);

    // return the last sequence number of 'sequenceNeg1'
    $lastSeq2 = $conn->lastInsertId($sequence2);

    // providing a table name in lastInsertId should return an empty string
    $lastSeq3 = $conn->lastInsertId($tableName);

    echo "Last sequence number of sequence1    = $lastSeq1\n";
    echo "Last sequence number of sequenceNeg1 = $lastSeq2\n";
    echo "Last sequence number when a table name is supplied = $lastSeq3\n";

    $stmt = $conn->query("DROP TABLE $tableName1");
    $stmt = $conn->query("DROP TABLE $tableName2");
    $stmt = $conn->query("DROP SEQUENCE $sequenceName");
    $stmt = $conn->query("DROP TABLE $tableName");
    $stmt = $conn->query("DROP SEQUENCE $sequence1");
    $stmt = $conn->query("DROP SEQUENCE $sequence2");

    unset($stmt);
    unset($conn);
} catch (Exception $e) {
    echo "Exception $e\n";
}
```

```
?>
```

The expected output is:

```
Last sequence number = 3
Last inserted ID     = 1
Last inserted ID when a table name is supplied =

Last sequence number of sequence1    = 3
Last sequence number of sequenceNeg1 = 198
Last sequence number when a table name is supplied =
```

See Also

PDO Class

PDO

PDO::prepare

⊕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.
PDO::ATTR_EMULATE_PREPARES	By default, this attribute is false, which can be changed by this `PDO::ATTR_EMULATE_PREPARES => true`. See Emulate Prepare for details and example.

KEY	DESCRIPTION
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).
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
<?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
<?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);
?>
```

Example

This example shows how to use the PDO::prepare method with `PDO::ATTR_EMULATE_PREPARES` set to true.

```php
<?php
$serverName = "yourservername";
$username = "yourusername";
$password = "yourpassword";
$database = "tempdb";
$conn = new PDO("sqlsrv:server = $serverName; Database = $database", $username, $password);

$pdo_options = array();
$pdo_options[PDO::ATTR_EMULATE_PREPARES] = true;
$pdo_options[PDO::SQLSRV_ATTR_ENCODING] = PDO::SQLSRV_ENCODING_UTF8;

$stmt = $conn->prepare("CREATE TABLE TEST([id] [int] IDENTITY(1,1) NOT NULL,
                                          [name] nvarchar(max))",
                                          $pdo_options);
$stmt->execute();

$prefix = '가각';
$name = '가각ácasa';
$name2 = '가각sample2';

$stmt = $conn->prepare("INSERT INTO TEST(name) VALUES(:p0)", $pdo_options);
$stmt->execute(['p0' => $name]);
unset($stmt);

$stmt = $conn->prepare("SELECT * FROM TEST WHERE NAME LIKE :p0", $pdo_options);
$stmt->execute(['p0' => "$prefix%"]);
foreach ($stmt as $row) {
    echo "\n" . 'FOUND: ' . $row['name'];
}

unset($stmt);
unset($conn);
?>
```

The PDO_SQLSRV driver internally replaces all the placeholders with the parameters that are bound by PDOStatement::bindParam(). Therefore, a SQL query string with no placeholders is sent to the server. Consider this example,

```
$statement = $PDO->prepare("INSERT into Customers (CustomerName, ContactName) VALUES (:cus_name,
:con_name)");
$statement->bindParam(:cus_name, "Cardinal");
$statement->bindParam(:con_name, "Tom B. Erichsen");
$statement->execute();
```

With `PDO::ATTR_EMULATE_PREPARES` set to false (the default case), the data sent to the database is:

```
"INSERT into Customers (CustomerName, ContactName) VALUES (:cus_name, :con_name)"
Information on :cus_name parameter
Information on :con_name parameter
```

The server will execute the query using its parameterized query feature for binding parameters. On the other hand, with `PDO::ATTR_EMULATE_PREPARES` set to true, the query sent to the server is essentially:

```
"INSERT into Customers (CustomerName, ContactName) VALUES ('Cardinal', 'Tom B. Erichsen')"
```

Setting `PDO::ATTR_EMULATE_PREPARES` to true can bypass some restrictions in SQL Server. For example, SQL Server does not support named or positional parameters in some Transact-SQL clauses. Moreover, SQL Server has a limit of binding 2100 parameters.

> **NOTE**
>
> With emulate prepares set to true, the security of parameterized queries is not in effect. Therefore, your application should ensure that the data that is bound to the parameter(s) does not contain malicious Transact-SQL code.

Encoding

If user wishes to bind parameters with different encodings (for instance, UTF-8 or binary), user should clearly specify the encoding in the PHP script.

The PDO_SQLSRV driver first checks the encoding specified in `PDO::bindParam()` (for example, `$statement->bindParam(:cus_name, "Cardinal", PDO::PARAM_STR, 10, PDO::SQLSRV_ENCODING_UTF8)`).

If not found, the driver checks if any encoding is set in `PDO::prepare()` or `PDOStatement::setAttribute()`. Otherwise, the driver will use the encoding specified in `PDO::__construct()` or `PDO::setAttribute()`.

Limitations

As you can see, binding is done internally by the driver. A valid query is sent to the server for execution without any parameter. Compared to the regular case, some limitations result when the parameterized query feature is not in use.

- It does not work for parameters that are bound as `PDO::PARAM_INPUT_OUTPUT`.
  - When the user specifies `PDO::PARAM_INPUT_OUTPUT` in `PDO::bindParam()`, a PDO exception is thrown.
- It does not work for parameters that are bound as output parameters.
  - When the user creates a prepared statement with placeholders that are meant for output parameters (that is, having an equal sign immediately after a placeholder, like `SELECT ? = COUNT(*) FROM Table1`), a PDO exception is thrown.
  - When a prepared statement invokes a stored procedure with a placeholder as the argument for an output parameter, no exception is thrown because the driver cannot detect the output parameter. However, the variable that the user provides for the output parameter will remain unchanged.
- Duplicated placeholders for a binary encoded parameter will not work

See Also

[PDO Class](#)

[PDO](#)

PDO::query

10/1/2018 • 2 minutes to read • Edit Online

⊕ 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. For more information, see Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver.

PDO::SQLSRV_ATTR_QUERY_TIMEOUT also affects the behavior of PDO::exec; for more information, see PDO::setAttribute.

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
<?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;
?>
```

Example

This code sample shows how to create a table of sql_variant types and fetch the inserted data.

```php
<?php
$server = 'serverName';
$dbName = 'databaseName';
$uid = 'yourUserName';
$pwd = 'yourPassword';

$conn = new PDO("sqlsrv:server=$server; database = $dbName", $uid, $pwd);
$conn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );

try {
    $tableName = 'testTable';
    $query = "CREATE TABLE $tableName ([c1_int] sql_variant, [c2_varchar] sql_variant)";

    $stmt = $conn->query($query);
    unset($stmt);

    $query = "INSERT INTO [$tableName] (c1_int, c2_varchar) VALUES (1, 'test_data')";
    $stmt = $conn->query($query);
    unset($stmt);

    $query = "SELECT * FROM $tableName";
    $stmt = $conn->query($query);

    $result = $stmt->fetch(PDO::FETCH_ASSOC);
    print_r($result);

    unset($stmt);
    unset($conn);
} catch (Exception $e) {
    echo $e->getMessage();
}
?>
```

The expected output would be:

```
Array
(
    [c1_int] => 1
    [c2_varchar] => test_data
)
```

See Also

PDO Class

PDO

PDO::quote

⊕ 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
<?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

⊕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

10/1/2018 • 2 minutes to read • Edit Online

⊕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 reports 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))` . For more information, see the PDO documentation.
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.	Sets the size of the buffer that holds the result set when using a client-side cursor. The default is 10240 KB, if not specified in the php.ini file. Zero and negative numbers are not allowed. 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, the return value is a string even when SQLSRV_ATTR_FETCHES_NUMERIC_TYPE is on. When the returned PDO type in bind column is PDO_PARAM_INT, the return value from an integer column is an 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 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 is 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
<?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

PDO

PDOStatement Class

10/1/2018 • 2 minutes to read • Edit Online

⊕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

PDOStatement::bindColumn

10/1/2018 • 2 minutes to read • Edit Online

⬇ 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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕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, use a parameter name (:name). For a prepared statement using the question mark syntax, it is 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*. For more information about encoding constants, see 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
<?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
<?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;
?>
```

> **NOTE**
>
> When binding an output parameter to a bigint type, if the value may end up outside the range of an integer, using PDO::PARAM_INT with PDO::SQLSRV_PARAM_OUT_DEFAULT_SIZE may result in a "value out of range" exception. Therefore, use the default PDO::PARAM_STR instead and provide the size of the resulting string, which is at most 21. It is the maximum number of digits, including the negative sign, of any bigint value.

Example

This code sample shows how to use an input/output parameter.

```php
<?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
?>
```

> **NOTE**
>
> It is recommended to use strings as inputs when binding values to a decimal or numeric column to ensure precision and accuracy as PHP has limited precision for floating point numbers. The same applies to bigint columns, especially when the values are outside the range of an integer.

Example

This code sample shows how to bind a decimal value as an input parameter.

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

// Assume TestTable exists with a decimal field
$input = 9223372036854.80000;
$stmt = $conn->prepare("INSERT INTO TestTable (DecimalCol) VALUES (?)");
// by default it is PDO::PARAM_STR, rounding of a large input value may
// occur if PDO::PARAM_INT is specified
$stmt->bindParam(1, $input, PDO::PARAM_STR);
$stmt->execute();
```

See Also

PDOStatement Class

PDO

PDOStatement::bindValue

⊕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, use a parameter name (:name). For a prepared statement using the question mark syntax, it is 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
<?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";
}
?>
```

> **NOTE**
>
> It is recommended to use strings as inputs when binding values to a decimal or numeric column to ensure precision and accuracy as PHP has limited precision for floating point numbers. The same applies to bigint columns, especially when the values are outside the range of an integer.

Example

This code sample shows how to bind a decimal value as an input parameter.

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

// Assume TestTable exists with a decimal field
$input = 9223372036854.80000;
$stmt = $conn->prepare("INSERT INTO TestTable (DecimalCol) VALUES (?)");
// by default it is PDO::PARAM_STR, rounding of a large input value may
// occur if PDO::PARAM_INT is specified
$stmt->bindValue(1, $input, PDO::PARAM_STR);
$stmt->execute();
```

See Also

PDOStatement Class

PDO

PDOStatement::closeCursor

10/1/2018 • 2 minutes to read • Edit Online

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
<?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

PDOStatement::columnCount

10/1/2018 • 2 minutes to read • Edit Online

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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕ 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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕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

10/1/2018 • 2 minutes to read • Edit Online

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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⊕ 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
<?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";
}
?>
```

> **NOTE**
>
> It is recommended to use strings as inputs when binding values to a decimal or numeric column to ensure precision and accuracy as PHP has limited precision for floating point numbers. The same applies to bigint columns, especially when the values are outside the range of an integer.

See Also

PDOStatement Class

PDO

PDOStatement::fetch

10/1/2018 • 3 minutes to read • Edit Online

⬇ 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

10/1/2018 • 2 minutes to read • Edit Online

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
<?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

10/1/2018 • 2 minutes to read • Edit Online

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
<?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

10/1/2018 • 2 minutes to read • <u>Edit Online</u>

<u>Download PHP Driver</u>

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
<?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

<u>PDOStatement Class</u>

<u>PDO</u>

PDOStatement::getAttribute

10/1/2018 • 2 minutes to read • Edit Online

⊕ 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 (for more information, see Direct Statement Execution and Prepared Statement Execution in the PDO_SQLSRV Driver), PDO::ATTR_CURSOR and PDO::SQLSRV_ATTR_CURSOR_SCROLL_TYPE (for more information, see Cursor Types (PDO_SQLSRV Driver)).

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

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
<?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

10/1/2018 • 2 minutes to read • <u>Edit Online</u>

⊕<u>Download PHP Driver</u>

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
<?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

<u>PDOStatement Class</u>

<u>PDO</u>

PDOStatement::rowCount

10/1/2018 • 2 minutes to read • Edit Online

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
<?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

10/1/2018 • 2 minutes to read • Edit Online

⬇ 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 10,240 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_FETCHES_NUMERIC_TYPE	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, the return value is a string even when SQLSRV_ATTR_FETCHES_NUMERIC_TYPE is on. When the returned PDO type in bind column is PDO_PARAM_INT, the return value from an integer column is an int even if SQLSRV_ATTR_FETCHES_NUMERIC_TYPE is off.
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
<?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

10/1/2018 • 2 minutes to read • <u>Edit Online</u>

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
<?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 the Microsoft Drivers for PHP for SQL Server

10/1/2018 • 2 minutes to read • Edit Online

⊕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 Overview of SQL Server Security.

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. Therefore, 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 the Microsoft Drivers for PHP for SQL Server

Code Samples for the Microsoft Drivers for PHP for SQL Server

10/1/2018 • 2 minutes to read • Edit Online

⊕Download PHP Driver

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

Example Application (PDO_SQLSRV Driver)

⊕ 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 Microsoft Drivers for PHP for SQL Server

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 Microsoft Drivers for PHP for SQL Server.
  - 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
```

```
<!--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;
```

```php
/* 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();
```

```php
$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)
```

```php
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'>
```

```php
<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'/>
```

```php
<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
<?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)

10/1/2018 • 9 minutes to read • Edit Online

⊕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 Microsoft Drivers for PHP for SQL Server.
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 Microsoft Drivers for PHP for SQL Server.
- 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. */
```

```php
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':
```

```
        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);
```

```php
$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
<?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