Chapter 1 Introduction to the MySQL PHP API

PHP is a server-side, HTML-embedded scripting language that may be used to create dynamic Web pages. It is available for most operating systems and Web servers, and can access most common databases, including MySQL. PHP may be run as a separate program or compiled as a module for use with a Web server.

PHP provides three different MySQL API extensions:

- Chapter 3, *MySQL Improved Extension*: Stands for “MySQL, Improved”; this extension is available as of PHP 5.0.0. It is intended for use with MySQL 4.1.1 and later. This extension fully supports the authentication protocol used in MySQL 5.0, as well as the Prepared Statements and Multiple Statements APIs. In addition, this extension provides an advanced, object-oriented programming interface.
- Chapter 4, *MySQL Functions (PDO_MYSQL)*: Not its own API, but instead it's a MySQL driver for the PHP database abstraction layer PDO (PHP Data Objects). The PDO MySQL driver sits in the layer below PDO itself, and provides MySQL-specific functionality. This extension is available as of PHP 5.1.0.
- Chapter 5, *Original MySQL API*: Available for PHP versions 4 and 5, this extension is intended for use with MySQL versions prior to MySQL 4.1. This extension does not support the improved authentication protocol used in MySQL 4.1, nor does it support prepared statements or multiple statements. To use this extension with MySQL 4.1, you will likely configure the MySQL server to set the `old_passwords` system variable to 1 (see Client does not support authentication protocol).

  > **Warning**
  >
  > This extension was removed from PHP 5.5.0. All users must migrate to either `mysqli` or `PDO_MySQL`. For further information, see Section 2.3, “Choosing an API”.

  > **Note**
  >
  > This documentation, and other publications, sometimes uses the term `Connector/PHP`. This term refers to the full set of MySQL related functionality in PHP, which includes the three APIs that are described in the preceding discussion, along with the mysqlnd core library and all of its plugins.

The PHP distribution and documentation are available from the PHP Web site.

Chapter 2 Overview of the MySQL PHP drivers

Table of Contents

2.1 Introduction

There are three PHP APIs for accessing the MySQL database. This guide explains the terminology used to describe each API, information about choosing which API to use, and also information to help choose which MySQL library to use with the API.

2.2 Terminology overview

This section provides an introduction to the options available to you when developing a PHP application that needs to interact with a MySQL database.

What is an API?

An Application Programming Interface, or API, defines the classes, methods, functions and variables that your application will need to call in order to carry out its desired task. In the case of PHP applications that need to communicate with databases the necessary APIs are usually exposed via PHP extensions.

APIs can be procedural or object-oriented. With a procedural API you call functions to carry out tasks, with the object-oriented API you instantiate classes and then call methods on the resulting objects. Of the two the latter is usually the preferred interface, as it is more modern and leads to better organized code.

When writing PHP applications that need to connect to the MySQL server there are several API options available. This document discusses what is available and how to select the best solution for your application.

What is a Connector?

In the MySQL documentation, the term *connector* refers to a piece of software that allows your application to connect to the MySQL database server. MySQL provides connectors for a variety of languages, including PHP.

If your PHP application needs to communicate with a database server you will need to write PHP code to perform such activities as connecting to the database server, querying the database and other database-related functions. Software is required to provide the API that your PHP application will use, and also handle the communication between your application and the database server, possibly using other

intermediate libraries where necessary. This software is known generically as a connector, as it allows your application to *connect* to a database server.

What is a Driver?

A driver is a piece of software designed to communicate with a specific type of database server. The driver may also call a library, such as the MySQL Client Library or the MySQL Native Driver. These libraries implement the low-level protocol used to communicate with the MySQL database server.

By way of an example, the PHP Data Objects (PDO) [16] database abstraction layer may use one of several database-specific drivers. One of the drivers it has available is the PDO MYSQL driver, which allows it to interface with the MySQL server.

Sometimes people use the terms connector and driver interchangeably, this can be confusing. In the MySQL-related documentation the term “driver” is reserved for software that provides the database-specific part of a connector package.

What is an Extension?

In the PHP documentation you will come across another term - *extension*. The PHP code consists of a core, with optional extensions to the core functionality. PHP's MySQL-related extensions, such as the `mysqli` extension, and the `mysql` extension, are implemented using the PHP extension framework.

An extension typically exposes an API to the PHP programmer, to allow its facilities to be used programmatically. However, some extensions which use the PHP extension framework do not expose an API to the PHP programmer.

The PDO MySQL driver extension, for example, does not expose an API to the PHP programmer, but provides an interface to the PDO layer above it.

The terms API and extension should not be taken to mean the same thing, as an extension may not necessarily expose an API to the programmer.

2.3 Choosing an API

PHP offers three different APIs to connect to MySQL. Below we show the APIs provided by the mysql, mysqli, and PDO extensions. Each code snippet creates a connection to a MySQL server running on "example.com" using the username "user" and the password "password". And a query is run to greet the user.

Example 2.1 Comparing the three MySQL APIs

```
<?php
// mysqli
$mysqli = new mysqli("example.com", "user", "password", "database");
$result = $mysqli->query("SELECT 'Hello, dear MySQL user!' AS _message FROM DUAL");
$row = $result->fetch_assoc();
echo htmlentities($row['_message']);

// PDO
$pdo = new PDO('mysql:host=example.com;dbname=database', 'user', 'password');
$statement = $pdo->query("SELECT 'Hello, dear MySQL user!' AS _message FROM DUAL");
$row = $statement->fetch(PDO::FETCH_ASSOC);
```

```
echo htmlentities($row['_message']);

// mysql
$c = mysql_connect("example.com", "user", "password");
mysql_select_db("database");
$result = mysql_query("SELECT 'Hello, dear MySQL user!' AS _message FROM DUAL");
$row = mysql_fetch_assoc($result);
echo htmlentities($row['_message']);
?>
```

Recommended API

It is recommended to use either the mysqli or PDO_MySQL extensions. It is not recommended to use the old mysql extension for new development, as it has been deprecated as of PHP 5.5.0 and will be removed in the future. A detailed feature comparison matrix is provided below. The overall performance of all three extensions is considered to be about the same. Although the performance of the extension contributes only a fraction of the total run time of a PHP web request. Often, the impact is as low as 0.1%.

Feature comparison

	ext/mysqli	**PDO_MySQL**	**ext/mysql**
PHP version introduced	5.0	5.1	2.0
Included with PHP 5.x	Yes	Yes	Yes
Development status	Active	Active	Maintenance only
Lifecycle	Active	Active	Deprecated
Recommended for new projects	Yes	Yes	No
OOP Interface	Yes	Yes	No
Procedural Interface	Yes	No	Yes
API supports non-blocking, asynchronous queries with mysqlnd	Yes	No	No
Persistent Connections	Yes	Yes	Yes
API supports Charsets	Yes	Yes	Yes
API supports server-side Prepared Statements	Yes	Yes	No
API supports client-side Prepared Statements	No	Yes	No
API supports Stored Procedures	Yes	Yes	No
API supports Multiple Statements	Yes	Most	No
API supports Transactions	Yes	Yes	No
Transactions can be controlled with SQL	Yes	Yes	Yes
Supports all MySQL 5.1+ functionality	Yes	Most	No

2.4 Choosing a library

The mysqli, PDO_MySQL and mysql PHP extensions are lightweight wrappers on top of a C client library. The extensions can either use the mysqlnd library or the `libmysqlclient` library. Choosing a library is a compile time decision.

The mysqlnd library is part of the PHP distribution since 5.3.0. It offers features like lazy connections and query caching, features that are not available with libmysqlclient, so using the built-in mysqlnd library is highly recommended. See the mysqlnd documentation for additional details, and a listing of features and functionality that it offers.

Example 2.2 Configure commands for using mysqlnd or libmysqlclient

```
// Recommended, compiles with mysqlnd
$ ./configure --with-mysqli=mysqlnd --with-pdo-mysql=mysqlnd --with-mysql=mysqlnd

// Not recommended, compiles with libmysqlclient
$ ./configure --with-mysqli=/path/to/mysql_config --with-pdo-mysql=/path/to/mysql_config --with-mysql=/path/to
```

Library feature comparison

It is recommended to use the mysqlnd library instead of the MySQL Client Server library (libmysqlclient). Both libraries are supported and constantly being improved.

	MySQL native driver (mysqlnd)	**MySQL client server library (`libmysqlclient`)**
Part of the PHP distribution	Yes	No
PHP version introduced	5.3.0	N/A
License	PHP License 3.01	Dual-License
Development status	Active	Active
Lifecycle	No end announced	No end announced
PHP 5.4 compile default (for all MySQL extensions)	Yes	No
PHP 5.3 compile default (for all MySQL extensions)	No	Yes
Compression protocol support	Yes (5.3.1+)	Yes
SSL support	Yes (5.3.3+)	Yes
Named pipe support	Yes (5.3.4+)	Yes
Non-blocking, asynchronous queries	Yes	No
Performance statistics	Yes	No
LOAD LOCAL INFILE respects the open_basedir directive	Yes	No
Uses PHP's native memory management system (e.g., follows PHP memory limits)	Yes	No

	MySQL native driver (mysqlnd)	MySQL client server library (libmysqlclient)
Return numeric column as double (COM_QUERY)	Yes	No
Return numeric column as string (COM_QUERY)	Yes	Yes
Plugin API	Yes	Limited
Read/Write splitting for MySQL Replication	Yes, with plugin	No
Load Balancing	Yes, with plugin	No
Fail over	Yes, with plugin	No
Lazy connections	Yes, with plugin	No
Query caching	Yes, with plugin	No
Transparent query manipulations (E.g., auto-EXPLAIN or monitoring)	Yes, with plugin	No

2.5 Concepts

These concepts are specific to the MySQL drivers for PHP.

2.5.1 Buffered and Unbuffered queries

Queries are using the buffered mode by default. This means that query results are immediately transferred from the MySQL Server to PHP and is then kept in the memory of the PHP process. This allows additional operations like counting the number of rows, and moving (seeking) the current result pointer. It also allows issuing further queries on the same connection while working on the result set. The downside of the buffered mode is that larger result sets might require quite a lot memory. The memory will be kept occupied till all references to the result set are unset or the result set was explicitly freed, which will automatically happen during request end the latest. The terminology "store result" is also used for buffered mode, as the whole result set is stored at once.

> **Note**
>
> When using libmysqlclient as library PHP's memory limit won't count the memory used for result sets unless the data is fetched into PHP variables. With mysqlnd the memory accounted for will include the full result set.

Unbuffered MySQL queries execute the query and then return a resource while the data is still waiting on the MySQL server for being fetched. This uses less memory on the PHP-side, but can increase the load on the server. Unless the full result set was fetched from the server no further queries can be sent over the same connection. Unbuffered queries can also be referred to as "use result".

Following these characteristics buffered queries should be used in cases where you expect only a limited result set or need to know the amount of returned rows before reading all rows. Unbuffered mode should be used when you expect larger results.

Because buffered queries are the default, the examples below will demonstrate how to execute unbuffered queries with each API.

Example 2.3 Unbuffered query example: mysqli

```
<?php
$mysqli  = new mysqli("localhost", "my_user", "my_password", "world");
$uresult = $mysqli->query("SELECT Name FROM City", MYSQLI_USE_RESULT);

if ($uresult) {
   while ($row = $uresult->fetch_assoc()) {
       echo $row['Name'] . PHP_EOL;
   }
}
$uresult->close();
?>
```

Example 2.4 Unbuffered query example: pdo_mysql

```
<?php
$pdo = new PDO("mysql:host=localhost;dbname=world", 'my_user', 'my_pass');
$pdo->setAttribute(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, false);

$uresult = $pdo->query("SELECT Name FROM City");
if ($uresult) {
   while ($row = $uresult->fetch(PDO::FETCH_ASSOC)) {
       echo $row['Name'] . PHP_EOL;
   }
}
?>
```

Example 2.5 Unbuffered query example: mysql

```
<?php
$conn = mysql_connect("localhost", "my_user", "my_pass");
$db   = mysql_select_db("world");

$uresult = mysql_unbuffered_query("SELECT Name FROM City");
if ($uresult) {
   while ($row = mysql_fetch_assoc($uresult)) {
       echo $row['Name'] . PHP_EOL;
   }
}
?>
```

2.5.2 Character sets

Ideally a proper character set will be set at the server level, and doing this is described within the Character Set Configuration section of the MySQL Server manual. Alternatively, each MySQL API offers a method to set the character set at runtime.

The character set and character escaping

The character set should be understood and defined, as it has an affect on every action, and includes security implications. For example, the escaping mechanism (e.g., `mysqli_real_escape_string` for mysqli, `mysql_real_escape_string` for mysql, and `PDO::quote` for PDO_MySQL) will adhere to this setting. It is important to realize that these functions will not use the character set that is defined with a query, so for example the following will not have an effect on them:

Example 2.6 Problems with setting the character set with SQL

```
<?php

$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

// Will not affect $mysqli->real_escape_string();
$mysqli->query("SET NAMES utf8");

// Will not affect $mysqli->real_escape_string();
$mysqli->query("SET CHARACTER SET utf8");

// But, this will affect $mysqli->real_escape_string();
$mysqli->set_charset('utf8');

?>
```

Below are examples that demonstrate how to properly alter the character set at runtime using each API.

Example 2.7 Setting the character set example: mysqli

```
<?php

$mysqli = new mysqli("localhost", "my_user", "my_password", "world");

if (!$mysqli->set_charset('utf8')) {
    printf("Error loading character set utf8: %s\n", $mysqli->error);
} else {
    printf("Current character set: %s\n", $mysqli->character_set_name());
}

print_r( $mysqli->get_charset() );

?>
```

Example 2.8 Setting the character set example: pdo_mysql

Note: This only works as of PHP 5.3.6.

```
<?php
$pdo = new PDO("mysql:host=localhost;dbname=world;charset=utf8", 'my_user', 'my_pass');
?>
```

Example 2.9 Setting the character set example: mysql

```
<?php

$conn = mysql_connect("localhost", "my_user", "my_pass");
$db   = mysql_select_db("world");

if (!mysql_set_charset('utf8', $conn)) {
    echo "Error: Unable to set the character set.\n";
    exit;
}

echo 'Your current character set is: ' .  mysql_client_encoding($conn);

?>
```

The mysqli extension allows you to access the functionality provided by MySQL 4.1 and above. More information about the MySQL Database server can be found at http://www.mysql.com/

An overview of software available for using MySQL from PHP can be found at Section 3.2, “Overview”

Documentation for MySQL can be found at http://dev.mysql.com/doc/.

Parts of this documentation included from MySQL manual with permissions of Oracle Corporation.

3.1 Examples

All examples in the mysqli documentation use the world database. The world database can be found at http://downloads.mysql.com/docs/world.sql.gz

3.2 Overview

This section provides an introduction to the options available to you when developing a PHP application that needs to interact with a MySQL database.

What is an API?

An Application Programming Interface, or API, defines the classes, methods, functions and variables that your application will need to call in order to carry out its desired task. In the case of PHP applications that need to communicate with databases the necessary APIs are usually exposed via PHP extensions.

APIs can be procedural or object-oriented. With a procedural API you call functions to carry out tasks, with the object-oriented API you instantiate classes and then call methods on the resulting objects. Of the two the latter is usually the preferred interface, as it is more modern and leads to better organized code.

When writing PHP applications that need to connect to the MySQL server there are several API options available. This document discusses what is available and how to select the best solution for your application.

What is a Connector?

In the MySQL documentation, the term *connector* refers to a piece of software that allows your application to connect to the MySQL database server. MySQL provides connectors for a variety of languages, including PHP.

If your PHP application needs to communicate with a database server you will need to write PHP code to perform such activities as connecting to the database server, querying the database and other database-related functions. Software is required to provide the API that your PHP application will use, and also handle the communication between your application and the database server, possibly using other intermediate libraries where necessary. This software is known generically as a connector, as it allows your application to *connect* to a database server.

What is a Driver?

A driver is a piece of software designed to communicate with a specific type of database server. The driver may also call a library, such as the MySQL Client Library or the MySQL Native Driver. These libraries implement the low-level protocol used to communicate with the MySQL database server.

By way of an example, the PHP Data Objects (PDO) [16] database abstraction layer may use one of several database-specific drivers. One of the drivers it has available is the PDO MYSQL driver, which allows it to interface with the MySQL server.

Sometimes people use the terms connector and driver interchangeably, this can be confusing. In the MySQL-related documentation the term “driver” is reserved for software that provides the database-specific part of a connector package.

What is an Extension?

In the PHP documentation you will come across another term - *extension*. The PHP code consists of a core, with optional extensions to the core functionality. PHP's MySQL-related extensions, such as the `mysqli` extension, and the `mysql` extension, are implemented using the PHP extension framework.

An extension typically exposes an API to the PHP programmer, to allow its facilities to be used programmatically. However, some extensions which use the PHP extension framework do not expose an API to the PHP programmer.

The PDO MySQL driver extension, for example, does not expose an API to the PHP programmer, but provides an interface to the PDO layer above it.

The terms API and extension should not be taken to mean the same thing, as an extension may not necessarily expose an API to the programmer.

What are the main PHP API offerings for using MySQL?

There are three main API options when considering connecting to a MySQL database server:

- PHP's MySQL Extension
- PHP's mysqli Extension
- PHP Data Objects (PDO)

Each has its own advantages and disadvantages. The following discussion aims to give a brief introduction to the key aspects of each API.

What is PHP's MySQL Extension?

This is the original extension designed to allow you to develop PHP applications that interact with a MySQL database. The `mysql` extension provides a procedural interface and is intended for use only with MySQL versions older than 4.1.3. This extension can be used with versions of MySQL 4.1.3 or newer, but not all of the latest MySQL server features will be available.

> **Note**
>
> If you are using MySQL versions 4.1.3 or later it is *strongly* recommended that you use the `mysqli` extension instead.

The `mysql` extension source code is located in the PHP extension directory `ext/mysql`.

For further information on the `mysql` extension, see Chapter 5, *Original MySQL API*.

What is PHP's mysqli Extension?

The `mysqli` extension, or as it is sometimes known, the MySQL *improved* extension, was developed to take advantage of new features found in MySQL systems versions 4.1.3 and newer. The `mysqli` extension is included with PHP versions 5 and later.

The `mysqli` extension has a number of benefits, the key enhancements over the `mysql` extension being:

- Object-oriented interface
- Support for Prepared Statements
- Support for Multiple Statements
- Support for Transactions
- Enhanced debugging capabilities
- Embedded server support

> **Note**
>
> If you are using MySQL versions 4.1.3 or later it is *strongly* recommended that you use this extension.

As well as the object-oriented interface the extension also provides a procedural interface.

The `mysqli` extension is built using the PHP extension framework, its source code is located in the directory `ext/mysqli`.

For further information on the `mysqli` extension, see Chapter 3, *MySQL Improved Extension*.

What is PDO?

PHP Data Objects, or PDO, is a database abstraction layer specifically for PHP applications. PDO provides a consistent API for your PHP application regardless of the type of database server your application will connect to. In theory, if you are using the PDO API, you could switch the database server you used, from say Firebird to MySQL, and only need to make minor changes to your PHP code.

Other examples of database abstraction layers include JDBC for Java applications and DBI for Perl.

While PDO has its advantages, such as a clean, simple, portable API, its main disadvantage is that it doesn't allow you to use all of the advanced features that are available in the latest versions of MySQL server. For example, PDO does not allow you to use MySQL's support for Multiple Statements.

PDO is implemented using the PHP extension framework, its source code is located in the directory `ext/pdo`.

For further information on PDO, see the http://www.php.net/book.pdo.

What is the PDO MYSQL driver?

The PDO MYSQL driver is not an API as such, at least from the PHP programmer's perspective. In fact the PDO MYSQL driver sits in the layer below PDO itself and provides MySQL-specific functionality. The programmer still calls the PDO API, but PDO uses the PDO MYSQL driver to carry out communication with the MySQL server.

The PDO MYSQL driver is one of several available PDO drivers. Other PDO drivers available include those for the Firebird and PostgreSQL database servers.

The PDO MYSQL driver is implemented using the PHP extension framework. Its source code is located in the directory `ext/pdo_mysql`. It does not expose an API to the PHP programmer.

For further information on the PDO MYSQL driver, see Chapter 4, *MySQL Functions (PDO_MYSQL)*.

What is PHP's MySQL Native Driver?

In order to communicate with the MySQL database server the `mysql` extension, `mysqli` and the PDO MYSQL driver each use a low-level library that implements the required protocol. In the past, the only available library was the MySQL Client Library, otherwise known as `libmysqlclient`.

However, the interface presented by `libmysqlclient` was not optimized for communication with PHP applications, as `libmysqlclient` was originally designed with C applications in mind. For this reason the MySQL Native Driver, `mysqlnd`, was developed as an alternative to `libmysqlclient` for PHP applications.

The `mysql` extension, the `mysqli` extension and the PDO MySQL driver can each be individually configured to use either `libmysqlclient` or `mysqlnd`. As `mysqlnd` is designed specifically to be utilised in the PHP system it has numerous memory and speed enhancements over `libmysqlclient`. You are strongly encouraged to take advantage of these improvements.

> **Note**
>
> The MySQL Native Driver can only be used with MySQL server versions 4.1.3 and later.

The MySQL Native Driver is implemented using the PHP extension framework. The source code is located in `ext/mysqlnd`. It does not expose an API to the PHP programmer.

Comparison of Features

The following table compares the functionality of the three main methods of connecting to MySQL from PHP:

Table 3.1 Comparison of MySQL API options for PHP

	PHP's mysqli Extension	PDO (Using PDO MySQL Driver and MySQL Native Driver)	PHP's MySQL Extension
PHP version introduced	5.0	5.0	Prior to 3.0
Included with PHP 5.x	yes	yes	Yes
MySQL development status	Active development	Active development as of PHP 5.3	Maintenance only
Recommended by MySQL for new projects	Yes - preferred option	Yes	No
API supports Charsets	Yes	Yes	No
API supports server-side Prepared Statements	Yes	Yes	No
API supports client-side Prepared Statements	No	Yes	No
API supports Stored Procedures	Yes	Yes	No
API supports Multiple Statements	Yes	Most	No
Supports all MySQL 4.1+ functionality	Yes	Most	No

3.3 Quick start guide

This quick start guide will help with choosing and gaining familiarity with the PHP MySQL API.

This quick start gives an overview on the mysqli extension. Code examples are provided for all major aspects of the API. Database concepts are explained to the degree needed for presenting concepts specific to MySQL.

Required: A familiarity with the PHP programming language, the SQL language, and basic knowledge of the MySQL server.

3.3.1 Dual procedural and object-oriented interface

The mysqli extension features a dual interface. It supports the procedural and object-oriented programming paradigm.

Users migrating from the old mysql extension may prefer the procedural interface. The procedural interface is similar to that of the old mysql extension. In many cases, the function names differ only by prefix. Some mysqli functions take a connection handle as their first argument, whereas matching functions in the old mysql interface take it as an optional last argument.

Example 3.1 Easy migration from the old mysql extension

```
<?php
$mysqli = mysqli_connect("example.com", "user", "password", "database");
$res = mysqli_query($mysqli, "SELECT 'Please, do not use ' AS _msg FROM DUAL");
$row = mysqli_fetch_assoc($res);
echo $row['_msg'];

$mysql = mysql_connect("example.com", "user", "password");
mysql_select_db("test");
$res = mysql_query("SELECT 'the mysql extension for new developments.' AS _msg FROM DUAL", $mysql);
$row = mysql_fetch_assoc($res);
echo $row['_msg'];
?>
```

The above example will output:

```
Please, do not use the mysql extension for new developments.
```

The object-oriented interface

In addition to the classical procedural interface, users can choose to use the object-oriented interface. The documentation is organized using the object-oriented interface. The object-oriented interface shows functions grouped by their purpose, making it easier to get started. The reference section gives examples for both syntax variants.

There are no significant performance differences between the two interfaces. Users can base their choice on personal preference.

Example 3.2 Object-oriented and procedural interface

```
<?php
$mysqli = mysqli_connect("example.com", "user", "password", "database");
if (mysqli_connect_errno($mysqli)) {
    echo "Failed to connect to MySQL: " . mysqli_connect_error();
}

$res = mysqli_query($mysqli, "SELECT 'A world full of ' AS _msg FROM DUAL");
$row = mysqli_fetch_assoc($res);
echo $row['_msg'];

$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: " . $mysqli->connect_error;
}

$res = $mysqli->query("SELECT 'choices to please everybody.' AS _msg FROM DUAL");
$row = $res->fetch_assoc();
echo $row['_msg'];
?>
```

The above example will output:

```
A world full of choices to please everybody.
```

The object oriented interface is used for the quickstart because the reference section is organized that way.

Mixing styles

It is possible to switch between styles at any time. Mixing both styles is not recommended for code clarity and coding style reasons.

Example 3.3 Bad coding style

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: " . $mysqli->connect_error;
}

$res = mysqli_query($mysqli, "SELECT 'Possible but bad style.' AS _msg FROM DUAL");
if (!$res) {
    echo "Failed to run query: (" . $mysqli->errno . ") " . $mysqli->error;
}

if ($row = $res->fetch_assoc()) {
    echo $row['_msg'];
}
?>
```

The above example will output:

```
Possible but bad style.
```

See also

`mysqli::__construct`
`mysqli::query`
`mysqli_result::fetch_assoc`
$mysqli::connect_errno
$mysqli::connect_error
$mysqli::errno
$mysqli::error
The MySQLi Extension Function Summary

3.3.2 Connections

The MySQL server supports the use of different transport layers for connections. Connections use TCP/IP, Unix domain sockets or Windows named pipes.

The hostname `localhost` has a special meaning. It is bound to the use of Unix domain sockets. It is not possible to open a TCP/IP connection using the hostname `localhost` you must use `127.0.0.1` instead.

Example 3.4 Special meaning of localhost

```
<?php
$mysqli = new mysqli("localhost", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}
echo $mysqli->host_info . "\n";

$mysqli = new mysqli("127.0.0.1", "user", "password", "database", 3306);
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

echo $mysqli->host_info . "\n";
?>
```

The above example will output:

```
Localhost via UNIX socket
127.0.0.1 via TCP/IP
```

Connection parameter defaults

Depending on the connection function used, assorted parameters can be omitted. If a parameter is not provided, then the extension attempts to use the default values that are set in the PHP configuration file.

Example 3.5 Setting defaults

```
mysqli.default_host=192.168.2.27
mysqli.default_user=root
mysqli.default_pw=""
mysqli.default_port=3306
mysqli.default_socket=/tmp/mysql.sock
```

The resulting parameter values are then passed to the client library that is used by the extension. If the client library detects empty or unset parameters, then it may default to the library built-in values.

Built-in connection library defaults

If the host value is unset or empty, then the client library will default to a Unix socket connection on `localhost`. If socket is unset or empty, and a Unix socket connection is requested, then a connection to the default socket on `/tmp/mysql.sock` is attempted.

On Windows systems, the host name `.` is interpreted by the client library as an attempt to open a Windows named pipe based connection. In this case the socket parameter is interpreted as the pipe name. If not given or empty, then the socket (pipe name) defaults to `\\.\pipe\MySQL`.

If neither a Unix domain socket based not a Windows named pipe based connection is to be established and the port parameter value is unset, the library will default to port `3306`.

The mysqlnd library and the MySQL Client Library (libmysqlclient) implement the same logic for determining defaults.

Connection options

Connection options are available to, for example, set init commands which are executed upon connect, or for requesting use of a certain charset. Connection options must be set before a network connection is established.

For setting a connection option, the connect operation has to be performed in three steps: creating a connection handle with `mysqli_init`, setting the requested options using `mysqli_options`, and establishing the network connection with `mysqli_real_connect`.

Connection pooling

The mysqli extension supports persistent database connections, which are a special kind of pooled connections. By default, every database connection opened by a script is either explicitly closed by the user during runtime or released automatically at the end of the script. A persistent connection is not. Instead it is put into a pool for later reuse, if a connection to the same server using the same username, password, socket, port and default database is opened. Reuse saves connection overhead.

Every PHP process is using its own mysqli connection pool. Depending on the web server deployment model, a PHP process may serve one or multiple requests. Therefore, a pooled connection may be used by one or more scripts subsequently.

Persistent connection

If a unused persistent connection for a given combination of host, username, password, socket, port and default database can not be found in the connection pool, then mysqli opens a new connection. The use of persistent connections can be enabled and disabled using the PHP directive mysqli.allow_persistent.

The total number of connections opened by a script can be limited with mysqli.max_links. The maximum number of persistent connections per PHP process can be restricted with mysqli.max_persistent. Please note, that the web server may spawn many PHP processes.

A common complain about persistent connections is that their state is not reset before reuse. For example, open and unfinished transactions are not automatically rolled back. But also, authorization changes which happened in the time between putting the connection into the pool and reusing it are not reflected. This may be seen as an unwanted side-effect. On the contrary, the name `persistent` may be understood as a promise that the state is persisted.

The mysqli extension supports both interpretations of a persistent connection: state persisted, and state reset before reuse. The default is reset. Before a persistent connection is reused, the mysqli extension implicitly calls `mysqli_change_user` to reset the state. The persistent connection appears to the user as if it was just opened. No artifacts from previous usages are visible.

The `mysqli_change_user` function is an expensive operation. For best performance, users may want to recompile the extension with the compile flag `MYSQLI_NO_CHANGE_USER_ON_PCONNECT` being set.

It is left to the user to choose between safe behavior and best performance. Both are valid optimization goals. For ease of use, the safe behavior has been made the default at the expense of maximum performance.

See also

`mysqli::__construct`
`mysqli::init`
`mysqli::options`
`mysqli::real_connect`
`mysqli::change_user`
$mysqli::host_info
MySQLi Configuration Options
Persistent Database Connections

3.3.3 Executing statements

Statements can be executed with the `mysqli_query`, `mysqli_real_query` and `mysqli_multi_query` functions. The `mysqli_query` function is the most common, and combines the executing statement with a buffered fetch of its result set, if any, in one call. Calling `mysqli_query` is identical to calling `mysqli_real_query` followed by `mysqli_store_result`.

Example 3.6 Connecting to MySQL

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT)") ||
    !$mysqli->query("INSERT INTO test(id) VALUES (1)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

Buffered result sets

After statement execution results can be retrieved at once to be buffered by the client or by read row by row. Client-side result set buffering allows the server to free resources associated with the statement results as early as possible. Generally speaking, clients are slow consuming result sets. Therefore, it is recommended to use buffered result sets. `mysqli_query` combines statement execution and result set buffering.

PHP applications can navigate freely through buffered results. Navigation is fast because the result sets are held in client memory. Please, keep in mind that it is often easier to scale by client than it is to scale the server.

Example 3.7 Navigation through buffered results

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT)") ||
    !$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$res = $mysqli->query("SELECT id FROM test ORDER BY id ASC");

echo "Reverse order...\n";
for ($row_no = $res->num_rows - 1; $row_no >= 0; $row_no--) {
    $res->data_seek($row_no);
    $row = $res->fetch_assoc();
    echo " id = " . $row['id'] . "\n";
}

echo "Result set order...\n";
$res->data_seek(0);
while ($row = $res->fetch_assoc()) {
    echo " id = " . $row['id'] . "\n";
}
?>
```

The above example will output:

```
Reverse order...
 id = 3
 id = 2
 id = 1
Result set order...
 id = 1
 id = 2
 id = 3
```

Unbuffered result sets

If client memory is a short resource and freeing server resources as early as possible to keep server load low is not needed, unbuffered results can be used. Scrolling through unbuffered results is not possible before all rows have been read.

Example 3.8 Navigation through unbuffered results

```
<?php
$mysqli->real_query("SELECT id FROM test ORDER BY id ASC");
$res = $mysqli->use_result();

echo "Result set order...\n";
while ($row = $res->fetch_assoc()) {
    echo " id = " . $row['id'] . "\n";
}
?>
```

Result set values data types

The `mysqli_query`, `mysqli_real_query` and `mysqli_multi_query` functions are used to execute non-prepared statements. At the level of the MySQL Client Server Protocol, the command `COM_QUERY` and the text protocol are used for statement execution. With the text protocol, the MySQL server converts all data of a result sets into strings before sending. This conversion is done regardless of the SQL result set column data type. The mysql client libraries receive all column values as strings. No further client-side casting is done to convert columns back to their native types. Instead, all values are provided as PHP strings.

Example 3.9 Text protocol returns strings by default

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$res = $mysqli->query("SELECT id, label FROM test WHERE id = 1");
$row = $res->fetch_assoc();

printf("id = %s (%s)\n", $row['id'], gettype($row['id']));
printf("label = %s (%s)\n", $row['label'], gettype($row['label']));
?>
```

The above example will output:

```
id = 1 (string)
label = a (string)
```

It is possible to convert integer and float columns back to PHP numbers by setting the MYSQLI_OPT_INT_AND_FLOAT_NATIVE connection option, if using the mysqlnd library. If set, the mysqlnd library will check the result set meta data column types and convert numeric SQL columns to PHP numbers, if the PHP data type value range allows for it. This way, for example, SQL INT columns are returned as integers.

Example 3.10 Native data types with mysqlnd and connection option

```
<?php
$mysqli = mysqli_init();
$mysqli->options(MYSQLI_OPT_INT_AND_FLOAT_NATIVE, 1);
$mysqli->real_connect("example.com", "user", "password", "database");

if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$res = $mysqli->query("SELECT id, label FROM test WHERE id = 1");
$row = $res->fetch_assoc();

printf("id = %s (%s)\n", $row['id'], gettype($row['id']));
printf("label = %s (%s)\n", $row['label'], gettype($row['label']));
?>
```

The above example will output:

```
id = 1 (integer)
label = a (string)
```

See also

mysqli::__construct
mysqli::init
mysqli::options
mysqli::real_connect
mysqli::query
mysqli::multi_query
mysqli::use_result
mysqli::store_result
mysqli_result::free

3.3.4 Prepared Statements

The MySQL database supports prepared statements. A prepared statement or a parameterized statement is used to execute the same statement repeatedly with high efficiency.

Basic workflow

The prepared statement execution consists of two stages: prepare and execute. At the prepare stage a statement template is sent to the database server. The server performs a syntax check and initializes server internal resources for later use.

The MySQL server supports using anonymous, positional placeholder with ?.

Example 3.11 First stage: prepare

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

/* Non-prepared statement */
if (!$mysqli->query("DROP TABLE IF EXISTS test") || !$mysqli->query("CREATE TABLE test(id INT)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

/* Prepared statement, stage 1: prepare */
if (!($stmt = $mysqli->prepare("INSERT INTO test(id) VALUES (?)"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

Prepare is followed by execute. During execute the client binds parameter values and sends them to the server. The server creates a statement from the statement template and the bound values to execute it using the previously created internal resources.

Example 3.12 Second stage: bind and execute

```
<?php
/* Prepared statement, stage 2: bind and execute */
$id = 1;
if (!$stmt->bind_param("i", $id)) {
    echo "Binding parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}
?>
```

Repeated execution

A prepared statement can be executed repeatedly. Upon every execution the current value of the bound variable is evaluated and sent to the server. The statement is not parsed again. The statement template is not transferred to the server again.

Example 3.13 INSERT prepared once, executed multiple times

```
```

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

/* Non-prepared statement */
if (!$mysqli->query("DROP TABLE IF EXISTS test") || !$mysqli->query("CREATE TABLE test(id INT)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

/* Prepared statement, stage 1: prepare */
if (!($stmt = $mysqli->prepare("INSERT INTO test(id) VALUES (?)"))) {
     echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

/* Prepared statement, stage 2: bind and execute */
$id = 1;
if (!$stmt->bind_param("i", $id)) {
    echo "Binding parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

/* Prepared statement: repeated execution, only data transferred from client to server */
for ($id = 2; $id < 5; $id++) {
    if (!$stmt->execute()) {
        echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
    }
}

/* explicit close recommended */
$stmt->close();

/* Non-prepared statement */
$res = $mysqli->query("SELECT id FROM test");
var_dump($res->fetch_all());
?>
```

The above example will output:

```
array(4) {
  [0]=>
  array(1) {
    [0]=>
    string(1) "1"
  }
  [1]=>
  array(1) {
    [0]=>
    string(1) "2"
  }
  [2]=>
  array(1) {
    [0]=>
    string(1) "3"
  }
  [3]=>
  array(1) {
    [0]=>
    string(1) "4"
```

```
  }
}
```

Every prepared statement occupies server resources. Statements should be closed explicitly immediately after use. If not done explicitly, the statement will be closed when the statement handle is freed by PHP.

Using a prepared statement is not always the most efficient way of executing a statement. A prepared statement executed only once causes more client-server round-trips than a non-prepared statement. This is why the `SELECT` is not run as a prepared statement above.

Also, consider the use of the MySQL multi-INSERT SQL syntax for INSERTs. For the example, multi-INSERT requires less round-trips between the server and client than the prepared statement shown above.

Example 3.14 Less round trips using multi-INSERT SQL

```
<?php
if (!$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3), (4)")) {
    echo "Multi-INSERT failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

Result set values data types

The MySQL Client Server Protocol defines a different data transfer protocol for prepared statements and non-prepared statements. Prepared statements are using the so called binary protocol. The MySQL server sends result set data "as is" in binary format. Results are not serialized into strings before sending. The client libraries do not receive strings only. Instead, they will receive binary data and try to convert the values into appropriate PHP data types. For example, results from an SQL `INT` column will be provided as PHP integer variables.

Example 3.15 Native datatypes

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$stmt = $mysqli->prepare("SELECT id, label FROM test WHERE id = 1");
$stmt->execute();
$res = $stmt->get_result();
$row = $res->fetch_assoc();

printf("id = %s (%s)\n", $row['id'], gettype($row['id']));
printf("label = %s (%s)\n", $row['label'], gettype($row['label']));
?>
```

The above example will output:

```
id = 1 (integer)
label = a (string)
```

This behavior differs from non-prepared statements. By default, non-prepared statements return all results as strings. This default can be changed using a connection option. If the connection option is used, there are no differences.

Fetching results using bound variables

Results from prepared statements can either be retrieved by binding output variables, or by requesting a `mysqli_result` object.

Output variables must be bound after statement execution. One variable must be bound for every column of the statements result set.

Example 3.16 Output variable binding

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($stmt = $mysqli->prepare("SELECT id, label FROM test"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$out_id    = NULL;
$out_label = NULL;
if (!$stmt->bind_result($out_id, $out_label)) {
    echo "Binding output parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}

while ($stmt->fetch()) {
    printf("id = %s (%s), label = %s (%s)\n", $out_id, gettype($out_id), $out_label, gettype($out_label));
}
?>
```

The above example will output:

```
id = 1 (integer), label = a (string)
```

Prepared statements return unbuffered result sets by default. The results of the statement are not implicitly fetched and transferred from the server to the client for client-side buffering. The result set takes server resources until all results have been fetched by the client. Thus it is recommended to consume results timely. If a client fails to fetch all results or the client closes the statement before having fetched all data, the data has to be fetched implicitly by `mysqli`.

It is also possible to buffer the results of a prepared statement using `mysqli_stmt_store_result`.

Fetching results using mysqli_result interface

Instead of using bound results, results can also be retrieved through the mysqli_result interface. `mysqli_stmt_get_result` returns a buffered result set.

Example 3.17 Using mysqli_result to fetch results

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($stmt = $mysqli->prepare("SELECT id, label FROM test ORDER BY id ASC"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$stmt->execute()) {
     echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!($res = $stmt->get_result())) {
    echo "Getting result set failed: (" . $stmt->errno . ") " . $stmt->error;
}

var_dump($res->fetch_all());
?>
```

The above example will output:

```
array(1) {
  [0]=>
  array(2) {
    [0]=>
    int(1)
    [1]=>
    string(1) "a"
  }
}
```

Using the `mysqli_result interface` offers the additional benefit of flexible client-side result set navigation.

Example 3.18 Buffered result set for flexible read out

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT, label CHAR(1))") ||
    !$mysqli->query("INSERT INTO test(id, label) VALUES (1, 'a'), (2, 'b'), (3, 'c')")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($stmt = $mysqli->prepare("SELECT id, label FROM test"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$stmt->execute()) {
     echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

if (!($res = $stmt->get_result())) {
    echo "Getting result set failed: (" . $stmt->errno . ") " . $stmt->error;
}

for ($row_no = ($res->num_rows - 1); $row_no >= 0; $row_no--) {
    $res->data_seek($row_no);
    var_dump($res->fetch_assoc());
}
$res->close();
?>
```

The above example will output:

```
array(2) {
  ["id"]=>
  int(3)
  ["label"]=>
  string(1) "c"
}
array(2) {
  ["id"]=>
  int(2)
  ["label"]=>
  string(1) "b"
}
array(2) {
  ["id"]=>
  int(1)
  ["label"]=>
  string(1) "a"
}
```

Escaping and SQL injection

Bound variables are sent to the server separately from the query and thus cannot interfere with it. The server uses these values directly at the point of execution, after the statement template is parsed. Bound

parameters do not need to be escaped as they are never substituted into the query string directly. A hint must be provided to the server for the type of bound variable, to create an appropriate conversion. See the `mysqli_stmt_bind_param` function for more information.

Such a separation sometimes considered as the only security feature to prevent SQL injection, but the same degree of security can be achieved with non-prepared statements, if all the values are formatted correctly. It should be noted that correct formatting is not the same as escaping and involves more logic than simple escaping. Thus, prepared statements are simply a more convenient and less error-prone approach to this element of database security.

Client-side prepared statement emulation

The API does not include emulation for client-side prepared statement emulation.

Quick prepared - non-prepared statement comparison

The table below compares server-side prepared and non-prepared statements.

Table 3.2 Comparison of prepared and non-prepared statements

	Prepared Statement	**Non-prepared statement**
Client-server round trips, SELECT, single execution	2	1
Statement string transferred from client to server	1	1
Client-server round trips, SELECT, repeated (n) execution	1 + n	n
Statement string transferred from client to server	1 template, n times bound parameter, if any	n times together with parameter, if any
Input parameter binding API	Yes, automatic input escaping	No, manual input escaping
Output variable binding API	Yes	No
Supports use of mysqli_result API	Yes, use `mysqli_stmt_get_result`	Yes
Buffered result sets	Yes, use `mysqli_stmt_get_result` or binding with `mysqli_stmt_store_result`	Yes, default of `mysqli_query`
Unbuffered result sets	Yes, use output binding API	Yes, use `mysqli_real_query` with `mysqli_use_result`
MySQL Client Server protocol data transfer flavor	Binary protocol	Text protocol
Result set values SQL data types	Preserved when fetching	Converted to string or preserved when fetching
Supports all SQL statements	Recent MySQL versions support most but not all	Yes

See also

`mysqli::__construct`
`mysqli::query`
`mysqli::prepare`
`mysqli_stmt::prepare`

mysqli_stmt::execute
mysqli_stmt::bind_param
mysqli_stmt::bind_result

3.3.5 Stored Procedures

The MySQL database supports stored procedures. A stored procedure is a subroutine stored in the database catalog. Applications can call and execute the stored procedure. The CALL SQL statement is used to execute a stored procedure.

Parameter

Stored procedures can have IN, INOUT and OUT parameters, depending on the MySQL version. The mysqli interface has no special notion for the different kinds of parameters.

IN parameter

Input parameters are provided with the CALL statement. Please, make sure values are escaped correctly.

Example 3.19 Calling a stored procedure

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") || !$mysqli->query("CREATE TABLE test(id INT)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$mysqli->query("DROP PROCEDURE IF EXISTS p") ||
    !$mysqli->query("CREATE PROCEDURE p(IN id_val INT) BEGIN INSERT INTO test(id) VALUES(id_val); END;")) {
    echo "Stored procedure creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$mysqli->query("CALL p(1)")) {
    echo "CALL failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($res = $mysqli->query("SELECT id FROM test"))) {
    echo "SELECT failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

var_dump($res->fetch_assoc());
?>
```

The above example will output:

```
array(1) {
  ["id"]=>
  string(1) "1"
}
```

INOUT/OUT parameter

The values of `INOUT`/`OUT` parameters are accessed using session variables.

Example 3.20 Using session variables

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP PROCEDURE IF EXISTS p") ||
    !$mysqli->query('CREATE PROCEDURE p(OUT msg VARCHAR(50)) BEGIN SELECT "Hi!" INTO msg; END;')) {
    echo "Stored procedure creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}


if (!$mysqli->query("SET @msg = ''") || !$mysqli->query("CALL p(@msg)")) {
    echo "CALL failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($res = $mysqli->query("SELECT @msg as _p_out"))) {
    echo "Fetch failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$row = $res->fetch_assoc();
echo $row['_p_out'];
?>
```

The above example will output:

```
Hi!
```

Application and framework developers may be able to provide a more convenient API using a mix of session variables and databased catalog inspection. However, please note the possible performance impact of a custom solution based on catalog inspection.

Handling result sets

Stored procedures can return result sets. Result sets returned from a stored procedure cannot be fetched correctly using `mysqli_query`. The `mysqli_query` function combines statement execution and fetching the first result set into a buffered result set, if any. However, there are additional stored procedure result sets hidden from the user which cause `mysqli_query` to fail returning the user expected result sets.

Result sets returned from a stored procedure are fetched using `mysqli_real_query` or `mysqli_multi_query`. Both functions allow fetching any number of result sets returned by a statement, such as `CALL`. Failing to fetch all result sets returned by a stored procedure causes an error.

Example 3.21 Fetching results from stored procedures

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
```

```
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT)") ||
    !$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$mysqli->query("DROP PROCEDURE IF EXISTS p") ||
    !$mysqli->query('CREATE PROCEDURE p() READS SQL DATA BEGIN SELECT id FROM test; SELECT id + 1 FROM tes
    echo "Stored procedure creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$mysqli->multi_query("CALL p()")) {
    echo "CALL failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

do {
    if ($res = $mysqli->store_result()) {
        printf("---\n");
        var_dump($res->fetch_all());
        $res->free();
    } else {
        if ($mysqli->errno) {
            echo "Store failed: (" . $mysqli->errno . ") " . $mysqli->error;
        }
    }
} while ($mysqli->more_results() && $mysqli->next_result());
?>
```

The above example will output:

```
---
array(3) {
  [0]=>
  array(1) {
    [0]=>
    string(1) "1"
  }
  [1]=>
  array(1) {
    [0]=>
    string(1) "2"
  }
  [2]=>
  array(1) {
    [0]=>
    string(1) "3"
  }
}
---
array(3) {
  [0]=>
  array(1) {
    [0]=>
    string(1) "2"
  }
  [1]=>
  array(1) {
    [0]=>
    string(1) "3"
  }
  [2]=>
```

```
  array(1) {
    [0]=>
    string(1) "4"
  }
}
```

Use of prepared statements

No special handling is required when using the prepared statement interface for fetching results from the same stored procedure as above. The prepared statement and non-prepared statement interfaces are similar. Please note, that not every MYSQL server version may support preparing the `CALL` SQL statement.

Example 3.22 Stored Procedures and Prepared Statements

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") ||
    !$mysqli->query("CREATE TABLE test(id INT)") ||
    !$mysqli->query("INSERT INTO test(id) VALUES (1), (2), (3)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$mysqli->query("DROP PROCEDURE IF EXISTS p") ||
    !$mysqli->query('CREATE PROCEDURE p() READS SQL DATA BEGIN SELECT id FROM test; SELECT id + 1 FROM test; E
    echo "Stored procedure creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!($stmt = $mysqli->prepare("CALL p()"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

do {
    if ($res = $stmt->get_result()) {
        printf("---\n");
        var_dump(mysqli_fetch_all($res));
        mysqli_free_result($res);
    } else {
        if ($stmt->errno) {
            echo "Store failed: (" . $stmt->errno . ") " . $stmt->error;
        }
    }
} while ($stmt->more_results() && $stmt->next_result());
?>
```

Of course, use of the bind API for fetching is supported as well.

Example 3.23 Stored Procedures and Prepared Statements using bind API

```
<?php
```

```
if (!($stmt = $mysqli->prepare("CALL p()"))) {
    echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

if (!$stmt->execute()) {
    echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}

do {

    $id_out = NULL;
    if (!$stmt->bind_result($id_out)) {
        echo "Bind failed: (" . $stmt->errno . ") " . $stmt->error;
    }

    while ($stmt->fetch()) {
        echo "id = $id_out\n";
    }
} while ($stmt->more_results() && $stmt->next_result());
?>
```

See also

`mysqli::query`
`mysqli::multi_query`
`mysqli_result::next-result`
`mysqli_result::more-results`

3.3.6 Multiple Statements

MySQL optionally allows having multiple statements in one statement string. Sending multiple statements at once reduces client-server round trips but requires special handling.

Multiple statements or multi queries must be executed with `mysqli_multi_query`. The individual statements of the statement string are separated by semicolon. Then, all result sets returned by the executed statements must be fetched.

The MySQL server allows having statements that do return result sets and statements that do not return result sets in one multiple statement.

Example 3.24 Multiple Statements

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

if (!$mysqli->query("DROP TABLE IF EXISTS test") || !$mysqli->query("CREATE TABLE test(id INT)")) {
    echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}

$sql = "SELECT COUNT(*) AS _num FROM test; ";
$sql.= "INSERT INTO test(id) VALUES (1); ";
$sql.= "SELECT COUNT(*) AS _num FROM test; ";

if (!$mysqli->multi_query($sql)) {
    echo "Multi query failed: (" . $mysqli->errno . ") " . $mysqli->error;
```

```
}

do {
    if ($res = $mysqli->store_result()) {
        var_dump($res->fetch_all(MYSQLI_ASSOC));
        $res->free();
    }
} while ($mysqli->more_results() && $mysqli->next_result());
?>
```

The above example will output:

```
array(1) {
  [0]=>
  array(1) {
    ["_num"]=>
    string(1) "0"
  }
}
array(1) {
  [0]=>
  array(1) {
    ["_num"]=>
    string(1) "1"
  }
}
```

Security considerations

The API functions `mysqli_query` and `mysqli_real_query` do not set a connection flag necessary for activating multi queries in the server. An extra API call is used for multiple statements to reduce the likeliness of accidental SQL injection attacks. An attacker may try to add statements such as `; DROP DATABASE mysql` or `; SELECT SLEEP(999)`. If the attacker succeeds in adding SQL to the statement string but `mysqli_multi_query` is not used, the server will not execute the second, injected and malicious SQL statement.

Example 3.25 SQL Injection

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
$res    = $mysqli->query("SELECT 1; DROP TABLE mysql.user");
if (!$res) {
    echo "Error executing query: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

The above example will output:

```
Error executing query: (1064) You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server version for the right syntax
to use near 'DROP TABLE mysql.user' at line 1
```

Prepared statements

Use of the multiple statement with prepared statements is not supported.

See also

mysqli::query
mysqli::multi_query
mysqli_result::next-result
mysqli_result::more-results

3.3.7 API support for transactions

The MySQL server supports transactions depending on the storage engine used. Since MySQL 5.5, the default storage engine is InnoDB. InnoDB has full ACID transaction support.

Transactions can either be controlled using SQL or API calls. It is recommended to use API calls for enabling and disabling the auto commit mode and for committing and rolling back transactions.

Example 3.26 Setting auto commit mode with SQL and through the API

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

/* Recommended: using API to control transactional settings */
$mysqli->autocommit(false);

/* Won't be monitored and recognized by the replication and the load balancing plugin */
if (!$mysqli->query('SET AUTOCOMMIT = 0')) {
    echo "Query failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
?>
```

Optional feature packages, such as the replication and load balancing plugin, can easily monitor API calls. The replication plugin offers transaction aware load balancing, if transactions are controlled with API calls. Transaction aware load balancing is not available if SQL statements are used for setting auto commit mode, committing or rolling back a transaction.

Example 3.27 Commit and rollback

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
$mysqli->autocommit(false);

$mysqli->query("INSERT INTO test(id) VALUES (1)");
$mysqli->rollback();

$mysqli->query("INSERT INTO test(id) VALUES (2)");
$mysqli->commit();
?>
```

Please note, that the MySQL server cannot roll back all statements. Some statements cause an implicit commit.

See also

`mysqli::autocommit`
`mysqli_result::commit`
`mysqli_result::rollback`

3.3.8 Metadata

A MySQL result set contains metadata. The metadata describes the columns found in the result set. All metadata sent by MySQL is accessible through the `mysqli` interface. The extension performs no or negligible changes to the information it receives. Differences between MySQL server versions are not aligned.

Meta data is access through the `mysqli_result` interface.

Example 3.28 Accessing result set meta data

```
<?php
$mysqli = new mysqli("example.com", "user", "password", "database");
if ($mysqli->connect_errno) {
    echo "Failed to connect to MySQL: (" . $mysqli->connect_errno . ") " . $mysqli->connect_error;
}

$res = $mysqli->query("SELECT 1 AS _one, 'Hello' AS _two FROM DUAL");
var_dump($res->fetch_fields());
?>
```

The above example will output:

```
array(2) {
  [0]=>
  object(stdClass)#3 (13) {
    ["name"]=>
    string(4) "_one"
    ["orgname"]=>
    string(0) ""
    ["table"]=>
    string(0) ""
    ["orgtable"]=>
    string(0) ""
    ["def"]=>
    string(0) ""
    ["db"]=>
    string(0) ""
    ["catalog"]=>
    string(3) "def"
    ["max_length"]=>
    int(1)
    ["length"]=>
    int(1)
    ["charsetnr"]=>
    int(63)
    ["flags"]=>
```

```
    int(32897)
    ["type"]=>
    int(8)
    ["decimals"]=>
    int(0)
  }
  [1]=>
  object(stdClass)#4 (13) {
    ["name"]=>
    string(4) "_two"
    ["orgname"]=>
    string(0) ""
    ["table"]=>
    string(0) ""
    ["orgtable"]=>
    string(0) ""
    ["def"]=>
    string(0) ""
    ["db"]=>
    string(0) ""
    ["catalog"]=>
    string(3) "def"
    ["max_length"]=>
    int(5)
    ["length"]=>
    int(5)
    ["charsetnr"]=>
    int(8)
    ["flags"]=>
    int(1)
    ["type"]=>
    int(253)
    ["decimals"]=>
    int(31)
  }
}
```

Prepared statements

Meta data of result sets created using prepared statements are accessed the same way. A suitable `mysqli_result` handle is returned by `mysqli_stmt_result_metadata`.

Example 3.29 Prepared statements metadata

```
<?php
$stmt = $mysqli->prepare("SELECT 1 AS _one, 'Hello' AS _two FROM DUAL");
$stmt->execute();
$res = $stmt->result_metadata();
var_dump($res->fetch_fields());
?>
```

See also

`mysqli::query`
`mysqli_result::fetch_fields`

3.4 Installing/Configuring