

---

# **Doctrine DBAL Documentation**

***Release 2.1.0***

**Roman Borschel, Guilherme Blanco, Benjamin Eberlei, Jonathan**

June 17, 2015



<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Architecture</b>	<b>5</b>
2.1	Drivers . . . . .	5
2.2	Platforms . . . . .	5
2.3	Logging . . . . .	6
2.4	Schema . . . . .	6
2.5	Types . . . . .	6
<b>3</b>	<b>Configuration</b>	<b>7</b>
3.1	Getting a Connection . . . . .	7
<b>4</b>	<b>Data Retrieval And Manipulation</b>	<b>15</b>
4.1	Data Retrieval . . . . .	15
4.2	Binding Types . . . . .	17
4.3	API . . . . .	19
<b>5</b>	<b>SQL Query Builder</b>	<b>23</b>
5.1	Security: Safely preventing SQL Injection . . . . .	23
5.2	Building a Query . . . . .	24
5.3	Building Expressions . . . . .	27
5.4	Binding Parameters to Placeholders . . . . .	28
<b>6</b>	<b>Transactions</b>	<b>29</b>
6.1	Transaction Nesting . . . . .	29
6.2	Auto-commit mode . . . . .	31
<b>7</b>	<b>Platforms</b>	<b>33</b>
7.1	MySQL . . . . .	33
7.2	Oracle . . . . .	33
7.3	Microsoft SQL Server . . . . .	33
7.4	PostgreSQL . . . . .	34
7.5	SAP Sybase SQL Anywhere . . . . .	34
7.6	SQLite . . . . .	34
7.7	Drizzle . . . . .	34
<b>8</b>	<b>Types</b>	<b>35</b>
8.1	Reference . . . . .	35
8.2	Mapping Matrix . . . . .	40

8.3	Detection of Database Types . . . . .	44
8.4	Custom Mapping Types . . . . .	44
<b>9</b>	<b>Schema-Manager</b>	<b>47</b>
9.1	listDatabases() . . . . .	47
9.2	listSequences() . . . . .	47
9.3	listTableColumns() . . . . .	48
9.4	listTableDetails() . . . . .	48
9.5	listTableForeignKeys() . . . . .	48
9.6	listTableIndexes() . . . . .	48
9.7	listTables() . . . . .	49
9.8	listViews() . . . . .	49
9.9	createSchema() . . . . .	49
<b>10</b>	<b>Schema-Representation</b>	<b>51</b>
10.1	Schema Assets . . . . .	52
<b>11</b>	<b>Events</b>	<b>55</b>
11.1	PostConnect Event . . . . .	55
<b>12</b>	<b>Security</b>	<b>57</b>
12.1	SQL Injection: Safe and Unsafe APIs for User Input . . . . .	57
12.2	User input in your queries . . . . .	58
12.3	Non-ASCII compatible Charsets in MySQL . . . . .	59
<b>13</b>	<b>Sharding</b>	<b>61</b>
13.1	ID Generation . . . . .	62
13.2	Transactions . . . . .	63
13.3	Foreign Keys . . . . .	64
13.4	Complex Queries . . . . .	64
13.5	ShardManager Interface . . . . .	64
13.6	Schema Operations: SchemaSynchronizer Interface . . . . .	65
13.7	SQL Azure Federations . . . . .	65
13.8	Generic SQL Sharding Support . . . . .	67
<b>14</b>	<b>SQLAzure Sharding Tutorial</b>	<b>69</b>
14.1	Install Doctrine . . . . .	69
14.2	Setup Connection . . . . .	69
14.3	Create Database . . . . .	70
14.4	Create Schema . . . . .	70
14.5	View Federation Members . . . . .	71
14.6	Insert Data . . . . .	71
14.7	Split Federation . . . . .	74
14.8	Inserting Data after Split . . . . .	74
14.9	Querying data with filtering off . . . . .	75
14.10	Querying data with filtering on . . . . .	75
<b>15</b>	<b>Supporting Other Databases</b>	<b>77</b>
15.1	Implementation Steps in Detail . . . . .	77
<b>16</b>	<b>Portability</b>	<b>79</b>
16.1	Connection Wrapper . . . . .	79
16.2	Database Platform . . . . .	80
16.3	Keyword Lists . . . . .	80

<b>17 Caching</b>	<b>81</b>
<b>18 Known Vendor Issues</b>	<b>83</b>
18.1 PostgreSQL . . . . .	83
18.2 MySQL . . . . .	84
18.3 Sqlite . . . . .	84
18.4 IBM DB2 . . . . .	84
18.5 Oracle . . . . .	84
18.6 Microsoft SQL Server . . . . .	85
<b>19 Indices and tables</b>	<b>87</b>



Contents:





---

## Introduction

---

The Doctrine database abstraction & access layer (DBAL) offers a lightweight and thin runtime layer around a PDO-like API and a lot of additional, horizontal features like database schema introspection and manipulation through an OO API.

The fact that the Doctrine DBAL abstracts the concrete PDO API away through the use of interfaces that closely resemble the existing PDO API makes it possible to implement custom drivers that may use existing native or self-made APIs. For example, the DBAL ships with a driver for Oracle databases that uses the oci8 extension under the hood.

The following database vendors are currently supported:

- MySQL
- Oracle
- Microsoft SQL Server
- PostgreSQL
- SAP Sybase SQL Anywhere
- SQLite
- Drizzle

The Doctrine 2 database layer can be used independently of the object-relational mapper. In order to use the DBAL all you need is the `Doctrine\Common` and `Doctrine\DBAL` namespaces. Once you have the Common and DBAL namespaces you must setup a class loader to be able to autoload the classes:

```
<?php
use Doctrine\Common\ClassLoader;

require '/path/to/doctrine/lib/Doctrine/Common/ClassLoader.php';

$classLoader = new ClassLoader('Doctrine', '/path/to/doctrine');
$classLoader->register();
```

Now you are able to load classes that are in the `/path/to/doctrine` directory like `/path/to/doctrine/Doctrine/DBAL/DriverManager.php` which we will use later in this documentation to configure our first Doctrine DBAL connection.



---

## Architecture

---

As already said, the DBAL is a thin layer on top of PDO. PDO itself is mainly defined in terms of 2 classes: PDO and PDOStatement. The equivalent classes in the DBAL are Doctrine\DBAL\Connection and Doctrine\DBAL\Statement. A Doctrine\DBAL\Connection wraps a Doctrine\DBAL\Driver\Connection and a Doctrine\DBAL\Statement wraps a Doctrine\DBAL\Driver\Statement.

Doctrine\DBAL\Driver\Connection and Doctrine\DBAL\Driver\Statement are just interfaces. These interfaces are implemented by concrete drivers. For all PDO based drivers, PDO and PDOStatement are the implementations of these interfaces. Thus, for PDO-based drivers, a Doctrine\DBAL\Connection wraps a PDO instance and a Doctrine\DBAL\Statement wraps a PDOStatement instance. Even more, a Doctrine\DBAL\Connection *is a* Doctrine\DBAL\Driver\Connection and a Doctrine\DBAL\Statement *is a* Doctrine\DBAL\Driver\Statement.

What does a Doctrine\DBAL\Connection or a Doctrine\DBAL\Statement add to the underlying driver implementations? The enhancements include SQL logging, events and control over the transaction isolation level in a portable manner, among others.

A DBAL driver is defined to the outside in terms of 3 interfaces: Doctrine\DBAL\Driver, Doctrine\DBAL\Driver\Connection and Doctrine\DBAL\Driver\Statement. The latter two resemble (a subset of) the corresponding PDO API.

A concrete driver implementation must provide implementation classes for these 3 interfaces.

The DBAL is separated into several different packages that perfectly separate responsibilities of the different RDBMS layers.

### 2.1 Drivers

The drivers abstract a PHP specific database API by enforcing two interfaces:

- \Doctrine\DBAL\Driver\Driver
- \Doctrine\DBAL\Driver\Statement

The above two interfaces require exactly the same methods as PDO.

### 2.2 Platforms

The platforms abstract the generation of queries and which database features a platform supports. The \Doctrine\DBAL\Platforms\AbstractPlatform defines the common denominator of what a database

platform has to publish to the userland, to be fully supportable by Doctrine. This includes the SchemaTool, Transaction Isolation and many other features. The Database platform for MySQL for example can be used by all 3 MySQL extensions, PDO, Mysqli and ext/mysql.

## 2.3 Logging

The logging holds the interface and some implementations for debugging of Doctrine SQL query execution during a request.

## 2.4 Schema

The schema offers an API for each database platform to execute DDL statements against your platform or retrieve metadata about it. It also holds the Schema Abstraction Layer which is used by the different Schema Management facilities of Doctrine DBAL and ORM.

## 2.5 Types

The types offer an abstraction layer for the converting and generation of types between Databases and PHP. Doctrine comes bundled with some common types but offers the ability for developers to define custom types or extend existing ones easily.

---

## Configuration

---

### 3.1 Getting a Connection

You can get a DBAL Connection through the `Doctrine\DBAL\DriverManager` class.

```
<?php
$config = new \Doctrine\DBAL\Configuration();
//..
$connectionParams = array(
    'dbname' => 'mydb',
    'user' => 'user',
    'password' => 'secret',
    'host' => 'localhost',
    'driver' => 'pdo_mysql',
);
$conn = \Doctrine\DBAL\DriverManager::getConnection($connectionParams, $config);
```

Or, using the simpler URL form:

```
<?php
$config = new \Doctrine\DBAL\Configuration();
//..
$connectionParams = array(
    'url' => 'mysql://user:secret@localhost/mydb',
);
$conn = \Doctrine\DBAL\DriverManager::getConnection($connectionParams, $config);
```

The `DriverManager` returns an instance of `Doctrine\DBAL\Connection` which is a wrapper around the underlying driver connection (which is often a `PDO` instance).

The following sections describe the available connection parameters in detail.

#### 3.1.1 Connecting using a URL

The easiest way to specify commonly used connection parameters is using a database URL. The scheme is used to specify a driver, the user and password in the URL encode user and password for the connection, followed by the host and port parts (the “authority”). The path after the authority part represents the name of the database, sans the leading slash. Any query parameters are used as additional connection parameters.

The scheme names representing the drivers are either the regular driver names (see below) with any underscores in their name replaced with a hyphen (to make them legal in URL scheme names), or one of the following simplified driver names that serve as aliases:

- `db2`: alias for `ibm_db2`
- `mssql`: alias for `pdo_sqlsrv`
- `mysql/mysql2`: alias for `pdo_mysql`
- `pgsql/postgres/postgresql`: alias for `pdo_pgsql`
- `sqlite/sqlite3`: alias for `pdo_sqlite`

For example, to connect to a “foo” MySQL DB using the `pdo_mysql` driver on localhost port 4486 with the charset set to UTF-8, you would use the following URL:

```
mysql://localhost:4486/foo?charset=UTF-8
```

This is identical to the following connection string using the full driver name:

```
pdo-mysql://localhost:4486/foo?charset=UTF-8
```

If you wanted to use the `drizzle_pdo__mysql` driver instead:

```
drizzle-pdo-mysql://localhost:4486/foo?charset=UTF-8
```

In the two last example above, mind the dashes instead of the underscores in the URL schemes.

For connecting to an SQLite database, the authority portion of the URL is obviously irrelevant and thus can be omitted. The path part of the URL is, like for all other drivers, stripped of its leading slash, resulting in a relative file name for the database:

```
sqlite:///somedb.sqlite
```

This would access `somedb.sqlite` in the current working directory and is identical to the following:

```
sqlite://ignored:ignored@ignored:1234/somedb.sqlite
```

To specify an absolute file path, e.g. `/usr/local/var/db.sqlite`, simply use that as the database name, which results in two leading slashes for the path part of the URL, and four slashes in total after the URL scheme name and its following colon:

```
sqlite:///usr/local/var/db.sqlite
```

Which is, again, identical to supplying ignored user/pass/authority:

```
sqlite://notused:inthis@case//usr/local/var/db.sqlite
```

To connect to an in-memory SQLite instance, use `:memory:` as the database name:

```
sqlite:///memory:
```

---

**Note:** Any information extracted from the URL overwrites existing values for the parameter in question, but the rest of the information is merged together. You could, for example, have a URL without the `charset` setting in the query string, and then add a `charset` connection parameter next to `url`, to provide a default value in case the URL doesn’t contain a charset value.

---

### 3.1.2 Driver

The driver specifies the actual implementations of the DBAL interfaces to use. It can be configured in one of three ways:

- `driver`: The built-in driver implementation to use. The following drivers are currently available:

- `pdo_mysql`: A MySQL driver that uses the `pdo_mysql` PDO extension.
- `drizzle_pdo_mysql`: A Drizzle driver that uses `pdo_mysql` PDO extension.
- `mysqli`: A MySQL driver that uses the `mysqli` extension.
- `pdo_sqlite`: An SQLite driver that uses the `pdo_sqlite` PDO extension.
- `pdo_pgsql`: A PostgreSQL driver that uses the `pdo_pgsql` PDO extension.
- `pdo_oci`: An Oracle driver that uses the `pdo_oci` PDO extension. **Note that this driver caused problems in our tests. Prefer the `oci8` driver if possible.**
- `pdo_sqlsrv`: A Microsoft SQL Server driver that uses `pdo_sqlsrv` PDO **Note that this driver caused problems in our tests. Prefer the `sqlsrv` driver if possible.**
- `sqlsrv`: A Microsoft SQL Server driver that uses the `sqlsrv` PHP extension.
- `oci8`: An Oracle driver that uses the `oci8` PHP extension.
- `sqlanywhere`: A SAP Sybase SQL Anywhere driver that uses the `sqlanywhere` PHP extension.
- `driverClass`: Specifies a custom driver implementation if no 'driver' is specified. This allows the use of custom drivers that are not part of the Doctrine DBAL itself.
- `pdo`: Specifies an existing PDO instance to use.

### 3.1.3 Wrapper Class

By default a `Doctrine\DBAL\Connection` is wrapped around a driver `Connection`. The `wrapperClass` option allows to specify a custom wrapper implementation to use, however, a custom wrapper class must be a subclass of `Doctrine\DBAL\Connection`.

### 3.1.4 Connection Details

The connection details identify the database to connect to as well as the credentials to use. The connection details can differ depending on the used driver. The following sections describe the options recognized by each built-in driver.

---

**Note:** When using an existing PDO instance through the `pdo` option, specifying connection details is obviously not necessary.

---

#### `pdo_sqlite`

- `user` (string): Username to use when connecting to the database.
- `password` (string): Password to use when connecting to the database.
- `path` (string): The filesystem path to the database file. Mutually exclusive with `memory`. `path` takes precedence.
- `memory` (boolean): True if the SQLite database should be in-memory (non-persistent). Mutually exclusive with `path`. `path` takes precedence.

## pdo\_mysql

- `user` (string): Username to use when connecting to the database.
- `password` (string): Password to use when connecting to the database.
- `host` (string): Hostname of the database to connect to.
- `port` (integer): Port of the database to connect to.
- `dbname` (string): Name of the database/schema to connect to.
- `unix_socket` (string): Name of the socket used to connect to the database.
- `charset` (string): The charset used when connecting to the database.

## drizzle\_pdo\_mysql

**Requires** drizzle plugin `mysql_protocol` or `mysql_unix_socket_protocol` to be enabled. On Ubuntu this can be done by editing `/etc/drizzle/conf.d/mysql-protocol.cnf` or `/etc/drizzle/conf.d/mysql-unix-socket-protocol.cnf` and restart drizzled daemon.

- `user` (string): Username to use when connecting to the database. Only needed if authentication is configured for drizzled.
- `password` (string): Password to use when connecting to the database. Only needed if authentication is configured for drizzled.
- `host` (string): Hostname of the database to connect to.
- `port` (integer): Port of the database to connect to.
- `dbname` (string): Name of the database/schema to connect to.
- `unix_socket` (string): Name of the socket used to connect to the database.

## mysqli

- `user` (string): Username to use when connecting to the database.
- `password` (string): Password to use when connecting to the database.
- `host` (string): Hostname of the database to connect to.
- `port` (integer): Port of the database to connect to.
- `dbname` (string): Name of the database/schema to connect to.
- `unix_socket` (string): Name of the socket used to connect to the database.
- `charset` (string): The charset used when connecting to the database.
- `driverOptions` Any supported flags for mysqli found on <http://www.php.net/manual/en/mysqli.real-connect.php>

## pdo\_pgsql

- `user` (string): Username to use when connecting to the database.
- `password` (string): Password to use when connecting to the database.
- `host` (string): Hostname of the database to connect to.



- `port` (integer): Port of the database to connect to.
- `dbname` (string): Name of the database/schema to connect to.
- `charset` (string): The charset used when connecting to the database.
- `sslmode` (string): Determines whether or with what priority a SSL TCP/IP connection will be negotiated with the server. See the list of available modes: <http://www.postgresql.org/docs/9.1/static/libpq-connect.html#LIBPQ-CONNECT-SSLMODE>

PostgreSQL behaves differently with regard to booleans when you use `PDO::ATTR_EMULATE_PREPARES` or not. To switch from using `'true'` and `'false'` as strings you can change to integers by using: `$conn->getDatabasePlatform()->setUseBooleanTrueFalseStrings($flag)`.

### `pdo_oci` / `oci8`

- `user` (string): Username to use when connecting to the database.
- `password` (string): Password to use when connecting to the database.
- `host` (string): Hostname of the database to connect to.
- `port` (integer): Port of the database to connect to.
- `dbname` (string): Name of the database/schema to connect to.
- `servicename` (string): Optional name by which clients can connect to the database instance. Will be used as Oracle's SID connection parameter if given and defaults to Doctrine's `dbname` connection parameter value.
- `service` (boolean): Whether to use Oracle's `SERVICE_NAME` connection parameter in favour of SID when connecting. The value for this will be read from Doctrine's `servicename` if given, `dbname` otherwise.
- `pooled` (boolean): Whether to enable database resident connection pooling.
- `charset` (string): The charset used when connecting to the database.
- `instancename` (string): Optional parameter, complete whether to add the `INSTANCE_NAME` parameter in the connection. It is generally used to connect to an Oracle RAC server to select the name of a particular instance.

### `pdo_sqlsrv` / `sqlsrv`

- `user` (string): Username to use when connecting to the database.
- `password` (string): Password to use when connecting to the database.
- `host` (string): Hostname of the database to connect to.
- `port` (integer): Port of the database to connect to.
- `dbname` (string): Name of the database/schema to connect to.

### `sqlanywhere`

- `user` (string): Username to use when connecting to the database.
- `password` (string): Password to use when connecting to the database.
- `server` (string): Name of a running database server to connect to.
- `host` (string): Hostname of the database to connect to.

- `port` (integer): Port of the database to connect to.
- `dbname` (string): Name of the database/schema to connect to.
- `persistent` (boolean): Whether to establish a persistent connection.

Depending on the used underlying platform version, you can specify any other connection parameter that is supported by the particular platform version via the `driverOptions` option. You can find a list of supported connection parameters for each platform version here:

- [SQL Anywhere 10.0.1](#)
- [SQL Anywhere 11.0.0](#)
- [SQL Anywhere 11.0.1](#)
- [SQL Anywhere 12.0.0](#)
- [SQL Anywhere 12.0.1](#)
- [SAP Sybase SQL Anywhere 16.0](#)

### 3.1.5 Automatic platform version detection

Doctrine ships with different database platform implementations for some vendors to support version specific features, dialect and behaviour. As of Doctrine DBAL 2.5 the appropriate platform implementation for the underlying database server version can be detected at runtime automatically for nearly all drivers. Before 2.5 you had to configure Doctrine to use a certain platform implementation explicitly with the `platform` connection parameter (see section below). Otherwise Doctrine always used a default platform implementation. For example if your application was backed by a SQL Server 2012 database, Doctrine would still use the SQL Server 2008 platform implementation as it is the default, unless you told Doctrine explicitly to use the SQL Server 2012 implementation.

The following drivers support automatic database platform detection out of the box without any extra configuration required:

- `pdo_mysql`
- `mysqli`
- `pdo_pgsql`
- `pdo_sqlsrv`
- `sqlsrv`

Some drivers cannot provide the version of the underlying database server without having to query for it explicitly. For performance reasons (to save one extra query on every connect), Doctrine does not enable automatic database platform version detection for the following drivers:

- `sqlanywhere`

If you still want to tell Doctrine which database server version you are using in order to choose the appropriate platform implementation, you can pass the `serverVersion` option with a vendor specific version string that matches the database server version you are using. You can also pass this option if you want to disable automatic database platform detection for a driver that natively supports it and choose the platform version implementation explicitly.

### 3.1.6 Custom Platform

Each built-in driver uses a default implementation of `Doctrine\DBAL\Platforms\AbstractPlatform`. If you wish to use a customized or custom implementation, you can pass a precreated instance in the `platform` option.

### 3.1.7 Custom Driver Options

The `driverOptions` option allows to pass arbitrary options through to the driver. This is equivalent to the fourth argument of the [PDO constructor](#).



---

## Data Retrieval And Manipulation

---

Doctrine DBAL follows the PDO API very closely. If you have worked with PDO before you will get to know Doctrine DBAL very quickly. On top of the API provided by PDO there are tons of convenience functions in Doctrine DBAL.

### 4.1 Data Retrieval

Using a database implies retrieval of data. It is the primary use-case of a database. For this purpose each database vendor exposes a Client API that can be integrated into programming languages. PHP has a generic abstraction layer for this kind of API called PDO (PHP Data Objects). However because of disagreements between the PHP community there are often native extensions for each database vendor that are much more maintained (OCI8 for example).

Doctrine DBAL API builds on top of PDO and integrates native extensions by wrapping them into the PDO API as well. If you already have an open connection through the `Doctrine\DBAL\DriverManager::getConnection()` method you can start using this API for data retrieval easily.

Start writing an SQL query and pass it to the `query()` method of your connection:

```
<?php
use Doctrine\DBAL\DriverManager;

$conn = DriverManager::getConnection($params, $config);

$sql = "SELECT * FROM articles";
$stmt = $conn->query($sql); // Simple, but has several drawbacks
```

The query method executes the SQL and returns a database statement object. A database statement object can be iterated to retrieve all the rows that matched the query until there are no more rows:

```
<?php

while ($row = $stmt->fetch()) {
    echo $row['headline'];
}
```

The query method is the most simple one for fetching data, but it also has several drawbacks:

- There is no way to add dynamic parameters to the SQL query without modifying `$sql` itself. This can easily lead to a category of security holes called **SQL injection**, where a third party can modify the SQL executed and even execute their own queries through clever exploiting of the security hole.

- **Quoting** dynamic parameters for an SQL query is tedious work and requires lots of use of the `Doctrine\DBAL\Connection#quote()` method, which makes the original SQL query hard to read/understand.
- Databases optimize SQL queries before they are executed. Using the query method you will trigger the optimization process over and over again, although it could re-use this information easily using a technique called **prepared statements**.

This three arguments and some more technical details hopefully convinced you to investigate prepared statements for accessing your database.

### 4.1.1 Dynamic Parameters and Prepared Statements

Consider the previous query, now parameterized to fetch only a single article by id. Using **ext/mysql** (still the primary choice of MySQL access for many developers) you had to escape every value passed into the query using `mysql_real_escape_string()` to avoid SQL injection:

```
<?php
$sql = "SELECT * FROM articles WHERE id = '" . mysql_real_escape_string($id, $link) . "'";
$rs = mysql_query($sql);
```

If you start adding more and more parameters to a query (for example in UPDATE or INSERT statements) this approach might lead to complex to maintain SQL queries. The reason is simple, the actual SQL query is not clearly separated from the input parameters. Prepared statements separate these two concepts by requiring the developer to add **placeholders** to the SQL query (prepare) which are then replaced by their actual values in a second step (execute).

```
<?php
// $conn instanceof Doctrine\DBAL\Connection

$sql = "SELECT * FROM articles WHERE id = ?";
$stmt = $conn->prepare($sql);
$stmt->bindValue(1, $id);
$stmt->execute();
```

Placeholders in prepared statements are either simple positional question marks (?) or named labels starting with a double-colon (:name1). You cannot mix the positional and the named approach. The approach using question marks is called positional, because the values are bound in order from left to right to any question mark found in the previously prepared SQL query. That is why you specify the position of the variable to bind into the `bindValue()` method:

```
<?php
// $conn instanceof Doctrine\DBAL\Connection

$sql = "SELECT * FROM articles WHERE id = ? AND status = ?";
$stmt = $conn->prepare($sql);
$stmt->bindValue(1, $id);
$stmt->bindValue(2, $status);
$stmt->execute();
```

Named parameters have the advantage that their labels can be re-used and only need to be bound once:

```
<?php
// $conn instanceof Doctrine\DBAL\Connection

$sql = "SELECT * FROM users WHERE name = :name OR username = :name";
$stmt = $conn->prepare($sql);
$stmt->bindValue("name", $name);
$stmt->execute();
```

The following section describes the API of Doctrine DBAL with regard to prepared statements.

**Note:** Support for positional and named prepared statements varies between the different database extensions. PDO implements its own client side parser so that both approaches are feasible for all PDO drivers. OCI8/Oracle only supports named parameters, but Doctrine implements a client side parser to allow positional parameters also.

## 4.1.2 Using Prepared Statements

There are three low-level methods on `Doctrine\DBAL\Connection` that allow you to use prepared statements:

- `prepare($sql)` - Create a prepared statement of the type `Doctrine\DBAL\Statement`. Using this method is preferred if you want to re-use the statement to execute several queries with the same SQL statement only with different parameters.
- `executeQuery($sql, $params, $types)` - Create a prepared statement for the passed SQL query, bind the given params with their binding types and execute the query. This method returns the executed prepared statement for iteration and is useful for SELECT statements.
- `executeUpdate($sql, $params, $types)` - Create a prepared statement for the passed SQL query, bind the given params with their binding types and execute the query. This method returns the number of affected rows by the executed query and is useful for UPDATE, DELETE and INSERT statements.

A simple usage of `prepare` was shown in the previous section, however it is useful to dig into the features of a `Doctrine\DBAL\Statement` a little bit more. There are essentially two different types of methods available on a statement. Methods for binding parameters and types and methods to retrieve data from a statement.

- `bindValue($pos, $value, $type)` - Bind a given value to the positional or named parameter in the prepared statement.
- `bindParam($pos, &$param, $type)` - Bind a given reference to the positional or named parameter in the prepared statement.

If you are finished with binding parameters you have to call `execute()` on the statement, which will trigger a query to the database. After the query is finished you can access the results of this query using the fetch API of a statement:

- `fetch($fetchStyle)` - Retrieves the next row from the statement or false if there are none. Moves the pointer forward one row, so that consecutive calls will always return the next row.
- `fetchColumn($column)` - Retrieves only one column of the next row specified by column index. Moves the pointer forward one row, so that consecutive calls will always return the next row.
- `fetchAll($fetchStyle)` - Retrieves all rows from the statement.

The fetch API of a prepared statement obviously works only for SELECT queries.

If you find it tedious to write all the prepared statement code you can alternatively use the `Doctrine\DBAL\Connection#executeQuery()` and `Doctrine\DBAL\Connection#executeUpdate()` methods. See the API section below on details how to use them.

Additionally there are lots of convenience methods for data-retrieval and manipulation on the Connection, which are all described in the API section below.

## 4.2 Binding Types

Doctrine DBAL extends PDOs handling of binding types in prepared statement considerably. Besides the well known `\PDO::PARAM_*` constants you can make use of two very powerful additional features.

## 4.2.1 DoctrineDBALTypes Conversion

If you don't specify an integer (through a `PDO::PARAM*` constant) to any of the parameter binding methods but a string, Doctrine DBAL will ask the type abstraction layer to convert the passed value from its PHP to a database representation. This way you can pass `\DateTime` instances to a prepared statement and have Doctrine convert them to the appropriate vendors database format:

```
<?php
$date = new \DateTime("2011-03-05 14:00:21");
$stmt = $conn->prepare("SELECT * FROM articles WHERE publish_date > ?");
$stmt->bindValue(1, $date, "datetime");
$stmt->execute();
```

If you take a look at `Doctrine\DBAL\Types\DateTimeType` you will see that parts of the conversion is delegated to a method on the current database platform, which means this code works independent of the database you are using.

---

**Note:** Be aware this type conversion only works with `Statement#bindValue()`, `Connection#executeQuery()` and `Connection#executeUpdate()`. It is not supported to pass a doctrine type name to `Statement#bindParam()`, because this would not work with binding by reference.

---

## 4.2.2 List of Parameters Conversion

---

**Note:** This is a Doctrine 2.1 feature.

---

One rather annoying bit of missing functionality in SQL is the support for lists of parameters. You cannot bind an array of values into a single prepared statement parameter. Consider the following very common SQL statement:

```
SELECT * FROM articles WHERE id IN (?)
```

Since you are using an `IN` expression you would really like to use it in the following way (and I guess everybody has tried to do this once in his life, before realizing it doesn't work):

```
<?php
$stmt = $conn->prepare('SELECT * FROM articles WHERE id IN (?)');
// THIS WILL NOT WORK:
$stmt->bindValue(1, array(1, 2, 3, 4, 5, 6));
$stmt->execute();
```

Implementing a generic way to handle this kind of query is tedious work. This is why most developers fallback to inserting the parameters directly into the query, which can open SQL injection possibilities if not handled carefully.

Doctrine DBAL implements a very powerful parsing process that will make this kind of prepared statement possible natively in the binding type system. The parsing necessarily comes with a performance overhead, but only if you really use a list of parameters. There are two special binding types that describe a list of integers or strings:

- `\Doctrine\DBAL\Connection::PARAM_INT_ARRAY`
- `\Doctrine\DBAL\Connection::PARAM_STR_ARRAY`

Using one of this constants as a type you can activate the `SQLParser` inside Doctrine that rewrites the SQL and flattens the specified values into the set of parameters. Consider our previous example:

```
<?php
$stmt = $conn->executeQuery('SELECT * FROM articles WHERE id IN (?)',
    array(array(1, 2, 3, 4, 5, 6)),
```



```
array(\Doctrine\DBAL\Connection::PARAM_INT_ARRAY)
);
```

The SQL statement passed to `Connection#executeQuery` is not the one actually passed to the database. It is internally rewritten to look like the following explicit code that could be specified as well:

```
<?php
// Same SQL WITHOUT usage of Doctrine\DBAL\Connection::PARAM_INT_ARRAY
$stmt = $conn->executeQuery('SELECT * FROM articles WHERE id IN (?, ?, ?, ?, ?, ?)',
    array(1, 2, 3, 4, 5, 6),
    array(\PDO::PARAM_INT, \PDO::PARAM_INT, \PDO::PARAM_INT, \PDO::PARAM_INT, \PDO::PARAM_INT, \PDO::PARAM_INT));
```

This is much more complicated and is ugly to write generically.

**Note:** The parameter list support only works with `Doctrine\DBAL\Connection::executeQuery()` and `Doctrine\DBAL\Connection::executeUpdate()`, NOT with the binding methods of a prepared statement.

## 4.3 API

The DBAL contains several methods for executing queries against your configured database for data retrieval and manipulation. Below we'll introduce these methods and provide some examples for each of them.

### 4.3.1 prepare()

Prepare a given SQL statement and return the `\Doctrine\DBAL\Driver\Statement` instance:

```
<?php
$stmt = $conn->prepare('SELECT * FROM user');
$stmt->execute();
$users = $stmt->fetchAll();

/*
array(
    0 => array(
        'username' => 'jwage',
        'password' => 'changeme'
    )
)
*/
```

### 4.3.2 executeUpdate()

Executes a prepared statement with the given SQL and parameters and returns the affected rows count:

```
<?php
$count = $conn->executeUpdate('UPDATE user SET username = ? WHERE id = ?', array('jwage', 1));
echo $count; // 1
```

The `$types` variable contains the PDO or Doctrine Type constants to perform necessary type conversions between actual input parameters and expected database values. See the Types section for more information.

### 4.3.3 executeQuery()

Creates a prepared statement for the given SQL and passes the parameters to the execute method, then returning the statement:

```
<?php
$stmt = $conn->executeQuery('SELECT * FROM user WHERE username = ?', array('jwage'));
$user = $stmt->fetch();

/*
array(
    0 => 'jwage',
    1 => 'changeme'
)
*/
```

The `$types` variable contains the PDO or Doctrine Type constants to perform necessary type conversions between actual input parameters and expected database values. See the Types section for more information.

### 4.3.4 fetchAll()

Execute the query and fetch all results into an array:

```
<?php
$users = $conn->fetchAll('SELECT * FROM user');

/*
array(
    0 => array(
        'username' => 'jwage',
        'password' => 'changeme'
    )
)
*/
```

### 4.3.5 fetchArray()

Numeric index retrieval of first result row of the given query:

```
<?php
$user = $conn->fetchArray('SELECT * FROM user WHERE username = ?', array('jwage'));

/*
array(
    0 => 'jwage',
    1 => 'changeme'
)
*/
```

### 4.3.6 fetchColumn()

Retrieve only the given column of the first result row.

```
<?php
$username = $conn->fetchColumn('SELECT username FROM user WHERE id = ?', array(1), 0);
echo $username; // jwage
```

### 4.3.7 fetchAssoc()

Retrieve assoc row of the first result row.

```
<?php
$user = $conn->fetchAssoc('SELECT * FROM user WHERE username = ?', array('jwage'));
/*
array(
    'username' => 'jwage',
    'password' => 'changeme'
)
*/
```

There are also convenience methods for data manipulation queries:

### 4.3.8 delete()

Delete all rows of a table matching the given identifier, where keys are column names.

```
<?php
$conn->delete('user', array('id' => 1));
// DELETE FROM user WHERE id = ? (1)
```

### 4.3.9 insert()

Insert a row into the given table name using the key value pairs of data.

```
<?php
$conn->insert('user', array('username' => 'jwage'));
// INSERT INTO user (username) VALUES (?) (jwage)
```

### 4.3.10 update()

Update all rows for the matching key value identifiers with the given data.

```
<?php
$conn->update('user', array('username' => 'jwage'), array('id' => 1));
// UPDATE user (username) VALUES (?) WHERE id = ? (jwage, 1)
```

By default the Doctrine DBAL does no escaping. Escaping is a very tricky business to do automatically, therefore there is none by default. The ORM internally escapes all your values, because it has lots of metadata available about the current context. When you use the Doctrine DBAL as standalone, you have to take care of this yourself. The following methods help you with it:

### 4.3.11 quote()

Quote a value:

```
<?php
$quoted = $conn->quote('value');
$quoted = $conn->quote('1234', \PDO::PARAM_INT);
```

### 4.3.12 quoteIdentifier()

Quote an identifier according to the platform details.

```
<?php
$quoted = $conn->quoteIdentifier('id');
```

---

## SQL Query Builder

---

Doctrine 2.1 ships with a powerful query builder for the SQL language. This QueryBuilder object has methods to add parts to an SQL statement. If you built the complete state you can execute it using the connection it was generated from. The API is roughly the same as that of the DQL Query Builder.

You can access the QueryBuilder by calling `Doctrine\DBAL\Connection#createQueryBuilder()`:

```
<?php
$conn = DriverManager::getConnection(array(/*...*/));
$queryBuilder = $conn->createQueryBuilder();
```

### 5.1 Security: Safely preventing SQL Injection

It is important to understand how the query builder works in terms of preventing SQL injection. Because SQL allows expressions in almost every clause and position the Doctrine QueryBuilder can only prevent SQL injections for calls to the methods `setFirstResult()` and `setMaxResults()`.

All other methods cannot distinguish between user- and developer input and are therefore subject to the possibility of SQL injection.

To safely work with the QueryBuilder you should **NEVER** pass user input to any of the methods of the QueryBuilder and use the placeholder `?` or `:name` syntax in combination with `$queryBuilder->setParameter($placeholder, $value)` instead:

```
<?php
$queryBuilder
    ->select('id', 'name')
    ->from('users')
    ->where('email = ?')
    ->setParameter(0, $userInputEmail)
;
```

---

**Note:** Due to an API design error the numerical parameters in the QueryBuilder API start with the needle 0, not with 1 as in the PDO API. This is very unfortunate, but we have found no BC way to fix this.

---

## 5.2 Building a Query

The `\Doctrine\DBAL\Query\QueryBuilder` supports building SELECT, INSERT, UPDATE and DELETE queries. Which sort of query you are building depends on the methods you are using.

For SELECT queries you start with invoking the `select()` method

```
<?php

$queryBuilder
    ->select('id', 'name')
    ->from('users');
```

For INSERT, UPDATE and DELETE queries you can pass the table name into the `insert($tableName)`, `update($tableName)` and `delete($tableName)`:

```
<?php

$queryBuilder
    ->insert('users')
;

$queryBuilder
    ->update('users')
;

$queryBuilder
    ->delete('users')
;
```

You can convert a query builder to its SQL string representation by calling `$queryBuilder->getSQL()` or casting the object to string.

### 5.2.1 WHERE-Clause

The SELECT, UPDATE and DELETE types of queries allow where clauses with the following API:

```
<?php

$queryBuilder
    ->select('id', 'name')
    ->from('users')
    ->where('email = ?')
;
```

Calling `where()` overwrites the previous clause and you can prevent this by combining expressions with `andWhere()` and `orWhere()` methods. You can alternatively use expressions to generate the where clause.

### 5.2.2 Table alias

The `from()` method takes an optional second parameter with which a table alias can be specified.

```
<?php

$queryBuilder
    ->select('u.id', 'u.name')
```

```

->from('users', 'u')
->where('u.email = ?')
;

```

### 5.2.3 GROUP BY and HAVING Clause

The SELECT statement can be specified with GROUP BY and HAVING clauses. Using `having()` works exactly like using `where()` and there are corresponding `andHaving()` and `orHaving()` methods to combine predicates. For the GROUP BY you can use the methods `groupBy()` which replaces previous expressions or `addGroupBy()` which adds to them:

```

<?php
$queryBuilder
    ->select('DATE(last_login) as date', 'COUNT(id) AS users')
    ->from('users')
    ->groupBy('DATE(last_login)')
    ->having('users > 10')
;

```

### 5.2.4 Join Clauses

For SELECT clauses you can generate different types of joins: INNER, LEFT and RIGHT. The RIGHT join is not portable across all platforms (Sqlite for example does not support it).

A join always belongs to one part of the from clause. This is why you have to specify the alias of the FROM part the join belongs to as the first argument.

As a second and third argument you can then specify the name and alias of the join-table and the fourth argument contains the ON clause.

```

<?php
$queryBuilder
    ->select('u.id', 'u.name', 'p.number')
    ->from('users', 'u')
    ->innerJoin('u', 'phonenumbers', 'p', 'u.id = p.user_id')
;

```

The method signature for `join()`, `innerJoin()`, `leftJoin()` and `rightJoin()` is the same. `join()` is a shorthand syntax for `innerJoin()`.

### 5.2.5 Order-By Clause

The `orderBy($sort, $order = null)` method adds an expression to the ORDER BY clause. Be aware that the optional `$order` parameter is not safe for user input and accepts SQL expressions.

```

<?php
$queryBuilder
    ->select('id', 'name')
    ->from('users')
    ->orderBy('username', 'ASC')
    ->addOrderBy('last_login', 'ASC NULLS FIRST')
;

```

Use the `addOrderBy` method to add instead of replace the `orderBy` clause.

## 5.2.6 Limit Clause

Only a few database vendors have the `LIMIT` clause as known from MySQL, but we support this functionality for all vendors using workarounds. To use this functionality you have to call the methods `setFirstResult($offset)` to set the offset and `setMaxResults($limit)` to set the limit of results returned.

```
<?php
$queryBuilder
    ->select('id', 'name')
    ->from('users')
    ->setFirstResult(10)
    ->setMaxResults(20);
```

## 5.2.7 VALUES Clause

For the `INSERT` clause setting the values for columns to insert can be done with the `values()` method on the query builder:

```
<?php

$queryBuilder
    ->insert('users')
    ->values(
        array(
            'name' => '?',
            'password' => '?'
        )
    )
    ->setParameter(0, $username)
    ->setParameter(1, $password)
;
// INSERT INTO users (name, password) VALUES (?, ?)
```

Each subsequent call to `values()` overwrites any previous set values. Setting single values instead of all at once is also possible with the `setValue()` method:

```
<?php

$queryBuilder
    ->insert('users')
    ->setValue('name', '?')
    ->setValue('password', '?')
    ->setParameter(0, $username)
    ->setParameter(1, $password)
;
// INSERT INTO users (name, password) VALUES (?, ?)
```

Of course you can also use both methods in combination:

```
<?php

$queryBuilder
    ->insert('users')
    ->values(
        array(
            'name' => '?'
        )
    )
```



```

->setParameter(0, $username)
;
// INSERT INTO users (name) VALUES (?)

if ($password) {
    $queryBuilder
        ->setValue('password', '?')
        ->setParameter(1, $password)
    ;
    // INSERT INTO users (name, password) VALUES (?, ?)
}

```

Not setting any values at all will result in an empty insert statement:

```

<?php

$queryBuilder
    ->insert('users')
;
// INSERT INTO users () VALUES ()

```

## 5.2.8 Set Clause

For the UPDATE clause setting columns to new values is necessary and can be done with the `set()` method on the query builder. Be aware that the second argument allows expressions and is not safe for user-input:

```

<?php

$queryBuilder
    ->update('users', 'u')
    ->set('u.logins', 'u.logins + 1')
    ->set('u.last_login', '?')
    ->setParameter(0, $userInputLastLogin)
;

```

## 5.3 Building Expressions

For more complex WHERE, HAVING or other clauses you can use expressions for building these query parts. You can invoke the expression API, by calling `$queryBuilder->expr()` and then invoking the helper method on it.

Most notably you can use expressions to build nested And-/Or statements:

```

<?php

$queryBuilder
    ->select('id', 'name')
    ->from('users')
    ->where(
        $queryBuilder->expr()->andX(
            $queryBuilder->expr()->eq('username', '?'),
            $queryBuilder->expr()->eq('email', '?')
        )
    );

```

The `andX()` and `orX()` methods accept an arbitrary amount of arguments and can be nested in each other.

There is a bunch of methods to create comparisons and other SQL snippets on the Expression object that you can see on the API documentation.

## 5.4 Binding Parameters to Placeholders

It is often not necessary to know about the exact placeholder names during the building of a query. You can use two helper methods to bind a value to a placeholder and directly use that placeholder in your query as a return value:

```
<?php

$queryBuilder
    ->select('id', 'name')
    ->from('users')
    ->where('email = ' . $queryBuilder->createNamedParameter($userInputEmail))
;
// SELECT id, name FROM users WHERE email = :dcValue1

$queryBuilder
    ->select('id', 'name')
    ->from('users')
    ->where('email = ' . $queryBuilder->createPositionalParameter($userInputEmail))
;
// SELECT id, name FROM users WHERE email = ?
```

---

## Transactions

---

A `Doctrine\DBAL\Connection` provides a PDO-like API for transaction management, with the methods `Connection#beginTransaction()`, `Connection#commit()` and `Connection#rollback()`.

Transaction demarcation with the Doctrine DBAL looks as follows:

```
<?php
$conn->beginTransaction();
try{
    // do stuff
    $conn->commit();
} catch(Exception $e) {
    $conn->rollback();
    throw $e;
}
```

Alternatively, the control abstraction `Connection#transactional($func)` can be used to make the code more concise and to make sure you never forget to rollback the transaction in the case of an exception. The following code snippet is functionally equivalent to the previous one:

```
<?php
$conn->transactional(function($conn) {
    // do stuff
});
```

The `Doctrine\DBAL\Connection` also has methods to control the transaction isolation level as supported by the underlying database. `Connection#setTransactionIsolation($level)` and `Connection#getTransactionIsolation()` can be used for that purpose. The possible isolation levels are represented by the following constants:

```
<?php
Connection::TRANSACTION_READ_UNCOMMITTED
Connection::TRANSACTION_READ_COMMITTED
Connection::TRANSACTION_REPEATABLE_READ
Connection::TRANSACTION_SERIALIZABLE
```

The default transaction isolation level of a `Doctrine\DBAL\Connection` is chosen by the underlying platform but it is always at least `READ_COMMITTED`.

### 6.1 Transaction Nesting

A `Doctrine\DBAL\Connection` also adds support for nesting transactions, or rather propagating transaction control up the call stack. For that purpose, the `Connection` class keeps an internal counter that represents the

nesting level and is increased/decreased as `beginTransaction()`, `commit()` and

`rollback()` are invoked. `beginTransaction()` increases the

**nesting level whilst** `commit()` and `rollback()` decrease the nesting level. The nesting level starts at 0. Whenever the nesting level transitions from 0 to 1, `beginTransaction()` is invoked on the underlying driver connection and whenever the nesting level transitions from 1 to 0, `commit()` or `rollback()` is invoked on the underlying driver, depending on whether the transition was caused by `Connection#commit()` or `Connection#rollback()`.

What this means is that transaction control is basically passed to code higher up in the call stack and the inner transaction block is ignored, with one important exception that is described further below. Do not confuse this with “real” nested transactions or savepoints. These are not supported by Doctrine. There is always only a single, real database transaction.

To visualize what this means in practice, consider the following example:

```
<?php
// $conn instanceof Doctrine\DBAL\Connection
$conn->beginTransaction(); // 0 => 1, "real" transaction started
try {
    ...

    // nested transaction block, this might be in some other API/library code that is
    // unaware of the outer transaction.
    $conn->beginTransaction(); // 1 => 2
    try {
        ...

        $conn->commit(); // 2 => 1
    } catch (Exception $e) {
        $conn->rollback(); // 2 => 1, transaction marked for rollback only
        throw $e;
    }

    ...

    $conn->commit(); // 1 => 0, "real" transaction committed
} catch (Exception $e) {
    $conn->rollback(); // 1 => 0, "real" transaction rollback
    throw $e;
}
```

However, a **rollback in a nested transaction block will always mark the current transaction so that the only possible outcome of the transaction is to be rolled back**. That means in the above example, the rollback in the inner transaction block marks the whole transaction for rollback only. Even if the nested transaction block would not rethrow the exception, the transaction is marked for rollback only and the commit of the outer transaction would trigger an exception, leading to the final rollback. This also means that you can not successfully commit some changes in an outer transaction if an inner transaction block fails and issues a rollback, even if this would be the desired behavior (i.e. because the nested operation is “optional” for the purpose of the outer transaction block). To achieve that, you need to restructure your application logic so as to avoid nesting transaction blocks. If this is not possible because the nested transaction blocks are in a third-party API you’re out of luck.

All that is guaranteed to the inner transaction is that it still happens atomically, all or nothing, the transaction just gets a wider scope and the control is handed to the outer scope.

**Note:** The transaction nesting described here is a debated feature that has its critics. Form your own opinion. We recommend avoiding nesting transaction blocks when possible, and most of the time, it is possible. Transaction control

should mostly be left to a service layer and not be handled in data access objects or similar.

**Warning:** Directly invoking `PDO#beginTransaction()`, `PDO#commit()` or `PDO#rollback()` or the corresponding methods on the particular `Doctrine\DBAL\Driver\Connection` instance in use bypasses the transparent transaction nesting that is provided by `Doctrine\DBAL\Connection` and can therefore corrupt the nesting level, causing errors with broken transaction boundaries that may be hard to debug.

## 6.2 Auto-commit mode

A `Doctrine\DBAL\Connection` supports setting the auto-commit mode to control whether queries should be automatically wrapped into a transaction or directly be committed to the database. By default a connection runs in auto-commit mode which means that it is non-transactional unless you start a transaction explicitly via `beginTransaction()`. To have a connection automatically open up a new transaction on `connect()` and after `commit()` or `rollback()`, you can disable auto-commit mode with `setAutoCommit(false)`.

```
<?php
// define connection parameters $params and initialize driver $driver

$conn = new \Doctrine\DBAL\Connection($params, $driver);

$conn->setAutoCommit(false); // disables auto-commit
$conn->connect(); // connects and immediately starts a new transaction

try {
    // do stuff
    $conn->commit(); // commits transaction and immediately starts a new one
} catch (\Exception $e) {
    $conn->rollback(); // rolls back transaction and immediately starts a new one
}

// still transactional
```

**Note:** Changing auto-commit mode during an active transaction, implicitly commits active transactions for that particular connection.

```
<?php
// define connection parameters $params and initialize driver $driver

$conn = new \Doctrine\DBAL\Connection($params, $driver);

// we are in auto-commit mode
$conn->beginTransaction();

// disable auto-commit, commits currently active transaction
$conn->setAutoCommit(false); // also causes a new transaction to be started

// no-op as auto-commit is already disabled
$conn->setAutoCommit(false);

// enable auto-commit again, commits currently active transaction
$conn->setAutoCommit(true); // does not start a new transaction automatically
```

Committing or rolling back an active transaction will of course only open up a new transaction automatically if the particular action causes the transaction context of a connection to terminate. That means committing or rolling back

nested transactions are not affected by this behaviour.

```
<?php
// we are not in auto-commit mode, transaction is active

try {
    // do stuff

    $conn->beginTransaction(); // start inner transaction, nesting level 2

    try {
        // do stuff
        $conn->commit(); // commits inner transaction, does not start a new one
    } catch (\Exception $e) {
        $conn->rollback(); // rolls back inner transaction, does not start a new one
    }

    // do stuff

    $conn->commit(); // commits outer transaction, and immediately starts a new one
} catch (\Exception $e) {
    $conn->rollback(); // rolls back outer transaction, and immediately starts a new one
}
```

To initialize a Doctrine\DBAL\Connection with auto-commit disabled, you can also use the Doctrine\DBAL\Configuration container to modify the default auto-commit mode via Doctrine\DBAL\Configuration::setAutoCommit(false) and pass it to a Doctrine\DBAL\Connection when instantiating.

---

## Platforms

---

Platforms abstract query generation and the subtle differences of the supported database vendors. In most cases you don't need to interact with the `Doctrine\DBAL\Platforms` package a lot, but there might be certain cases when you are programming database independent where you want to access the platform to generate queries for you.

The platform can be accessed from any `Doctrine\DBAL\Connection` instance by calling the `getDatabasePlatform()` method.

```
<?php
$platform = $conn->getDatabasePlatform();
```

Each database driver has a platform associated with it by default. Several drivers also share the same platform, for example `PDO_OCI` and `OCI8` share the `OraclePlatform`.

Doctrine provides abstraction for different versions of platforms if necessary to represent their specific features and dialects. For example has Microsoft added support for sequences in their 2012 version. Therefore Doctrine offers a separate platform class for this extending the previous 2008 version. The 2008 version adds support for additional data types which in turn don't exist in the previous 2005 version and so on. A list of available platform classes that can be used for each vendor can be found as follows:

### 7.1 MySQL

- `MySQLPlatform` for version 5.0 and above.
- `MySQL57Platform` for version 5.7 and above.

### 7.2 Oracle

- `OraclePlatform` for all versions.

### 7.3 Microsoft SQL Server

- `SQLServerPlatform` for version 2000 and above.
- `SQLServer2005Platform` for version 2005 and above.
- `SQLServer2008Platform` for version 2008 and above.
- `SQLServer2012Platform` for version 2012 and above.

## 7.4 PostgreSQL

- `PostgreSqlPlatform` for all versions.
- `PostgreSQL91Platform` for version 9.1 and above.
- `PostgreSQL92Platform` for version 9.2 and above.

## 7.5 SAP Sybase SQL Anywhere

- `SQLAnywherePlatform` for version 10 and above.
- `SQLAnywhere11Platform` for version 11 and above.
- `SQLAnywhere12Platform` for version 12 and above.
- `SQLAnywhere16Platform` for version 16 and above.

## 7.6 SQLite

- `SqlitePlatform` for all versions.

## 7.7 Drizzle

- `DrizzlePlatform` for all versions.

It is highly encouraged to use the platform class that matches your database vendor and version best. Otherwise it is not guaranteed that the compatibility in terms of SQL dialect and feature support between Doctrine DBAL and the database server will always be given.

If you want to overwrite parts of your platform you can do so when creating a connection. There is a `platform` option you can pass an instance of the platform you want the connection to use:

```
<?php
$myPlatform = new MyPlatform();
$options = array(
    'driver' => 'pdo_sqlite',
    'path' => 'database.sqlite',
    'platform' => $myPlatform
);
$conn = DriverManager::getConnection($options);
```

This way you can optimize your schema or generated SQL code with features that might not be portable for instance, however are required for your special needs. This can include using triggers or views to simulate features or adding behaviour to existing SQL functions.

Platforms are also responsible to know which database type translates to which PHP Type. This is a very tricky issue across all the different database vendors, for example MySQL BIGINT and Oracle NUMBER should be handled as integer. Doctrine 2 offers a powerful way to abstract the database to php and back conversion, which is described in the next section.



---

## Types

---

Besides abstraction of SQL one needs a translation between database and PHP data-types to implement database independent applications. Doctrine 2 has a type translation system baked in that supports the conversion from and to PHP values from any database platform, as well as platform independent SQL generation for any Doctrine Type.

Using the ORM you generally don't need to know about the Type system. This is unless you want to make use of database vendor specific database types not included in Doctrine 2.

Types are flyweights. This means there is only ever one instance of a type and it is not allowed to contain any state. Creation of type instances is abstracted through a static get method `Doctrine\DBAL\Types\Type::getType()`.

Types are abstracted across all the supported database vendors.

### 8.1 Reference

The following chapter gives an overview of all available Doctrine 2 types with short explanations on their context and usage. The type names listed here equal those that can be passed to the `Doctrine\DBAL\Types\Type::getType()` factory method in order to retrieve the desired type instance.

```
<?php
// Returns instance of \Doctrine\DBAL\Types\IntegerType
$type = \Doctrine\DBAL\Types\Type::getType('integer');
```

#### 8.1.1 Numeric types

Types that map numeric data such as integers, fixed and floating point numbers.

##### Integer types

Types that map numeric data without fractions.

##### **smallint**

Maps and converts 2-byte integer values. Unsigned integer values have a range of **0** to **65535** while signed integer values have a range of **32768** to **32767**. If you know the integer data you want to store always fits into one of these ranges you should consider using this type. Values retrieved from the database are always converted to PHP's `integer` type or `null` if no data is present.

---

**Note:** Not all of the database vendors support unsigned integers, so such an assumption might not be propagated to the database.

---

### integer

Maps and converts 4-byte integer values. Unsigned integer values have a range of **0** to **4294967295** while signed integer values have a range of **2147483648** to **2147483647**. If you know the integer data you want to store always fits into one of these ranges you should consider using this type. Values retrieved from the database are always converted to PHP's `integer` type or `null` if no data is present.

---

**Note:** Not all of the database vendors support unsigned integers, so such an assumption might not be propagated to the database.

---

### bigint

Maps and converts 8-byte integer values. Unsigned integer values have a range of **0** to **18446744073709551615** while signed integer values have a range of **9223372036854775808** to **9223372036854775807**. If you know the integer data you want to store always fits into one of these ranges you should consider using this type. Values retrieved from the database are always converted to PHP's `string` type or `null` if no data is present.

---

**Note:** For compatibility reasons this type is not converted to an integer as PHP can only represent big integer values as real integers on systems with a 64-bit architecture and would fall back to approximated float values otherwise which could lead to false assumptions in applications.

Not all of the database vendors support unsigned integers, so such an assumption might not be propagated to the database.

---

## Decimal types

Types that map numeric data with fractions.

### decimal

Maps and converts numeric data with fixed-point precision. If you need an exact precision for numbers with fractions, you should consider using this type. Values retrieved from the database are always converted to PHP's `string` type or `null` if no data is present.

---

**Note:** For compatibility reasons this type is not converted to a double as PHP can only preserve the precision to a certain degree. Otherwise it approximates precision which can lead to false assumptions in applications.

---

### float

Maps and converts numeric data with floating-point precision. If you only need an approximate precision for numbers with fractions, you should consider using this type. Values retrieved from the database are always converted to PHP's `float/double` type or `null` if no data is present.

## 8.1.2 String types

Types that map string data such as character and binary text.

### Character string types

Types that map string data of letters, numbers, and other symbols.

#### string

Maps and converts string data with a maximum length. If you know that the data to be stored always fits into the specified length, you should consider using this type. Values retrieved from the database are always converted to PHP's `string` type or `null` if no data is present.

---

**Note:** Database vendors have different limits for the maximum length of a varying string. Doctrine internally maps the `string` type to the vendor's `text` type if the maximum allowed length is exceeded. This can lead to type inconsistencies when reverse engineering the type from the database.

---

#### text

Maps and converts string data without a maximum length. If you don't know the maximum length of the data to be stored, you should consider using this type. Values retrieved from the database are always converted to PHP's `string` type or `null` if no data is present.

#### guid

Maps and converts a "Globally Unique Identifier". If you want to store a GUID, you should consider using this type, as some database vendors have a native data type for this kind of data which offers the most efficient way to store it. For vendors that do not support this type natively, this type is mapped to the `string` type internally. Values retrieved from the database are always converted to PHP's `string` type or `null` if no data is present.

### Binary string types

Types that map binary string data including images and other types of information that are not interpreted by the database. If you know that the data to be stored always is in binary format, you should consider using one of these types in favour of character string types, as it offers the most efficient way to store it.

#### binary

Maps and converts binary string data with a maximum length. If you know that the data to be stored always fits into the specified length, you should consider using this type. Values retrieved from the database are always converted to PHP's `resource` type or `null` if no data is present.

---

**Note:** Database vendors have different limits for the maximum length of a varying binary string. Doctrine internally maps the `binary` type to the vendor's `blob` type if the maximum allowed length is exceeded. This can lead to type inconsistencies when reverse engineering the type from the database.

---

## blob

Maps and converts binary string data without a maximum length. If you don't know the maximum length of the data to be stored, you should consider using this type. Values retrieved from the database are always converted to PHP's `resource` type or `null` if no data is present.

### 8.1.3 Bit types

Types that map bit data such as boolean values.

## boolean

Maps and converts boolean data. If you know that the data to be stored always is a `boolean` (`true` or `false`), you should consider using this type. Values retrieved from the database are always converted to PHP's `boolean` type or `null` if no data is present.

---

**Note:** As most of the database vendors do not have a native boolean type, this type silently falls back to the smallest possible integer or bit data type if necessary to ensure the least possible data storage requirements are met.

---

### 8.1.4 Date and time types

Types that map date, time and timezone related values such as date only, date and time, date, time and timezone or time only.

## date

Maps and converts date data without time and timezone information. If you know that the data to be stored always only needs to be a date without time and timezone information, you should consider using this type. Values retrieved from the database are always converted to PHP's `\DateTime` object or `null` if no data is present.

## datetime

Maps and converts date and time data without timezone information. If you know that the data to be stored always only needs to be a date with time but without timezone information, you should consider using this type. Values retrieved from the database are always converted to PHP's `\DateTime` object or `null` if no data is present.

**Warning:** Before 2.5 this type always required a specific format, defined in `$platform->getDateTimeFormatString()`, which could cause quite some troubles on platforms that had various microtime precision formats. Starting with 2.5 whenever the parsing of a date fails with the predefined platform format, the `date_create()` function will be used to parse the date. This could cause some troubles when your date format is weird and not parsed correctly by `date_create()`, however since databases are rather strict on dates there should be no problem.

## datetimetz

Maps and converts date with time and timezone information data. If you know that the data to be stored always contains date, time and timezone information, you should consider using this type. Values retrieved from the database are always converted to PHP's `\DateTime` object or `null` if no data is present.

## time

Maps and converts time data without date and timezone information. If you know that the data to be stored only needs to be a time without date, time and timezone information, you should consider using this type. Values retrieved from the database are always converted to PHP's `\DateTime` object or `null` if no data is present.

**Note:** See the Known Vendor Issue section for details about the different handling of microseconds and timezones across all the different vendors.

**Warning:** All date types assume that you are exclusively using the default timezone set by `date_default_timezone_set()` or by the `php.ini` configuration `date.timezone`. If you need specific timezone handling you have to handle this in your domain, converting all the values back and forth from UTC.

## 8.1.5 Array types

Types that map array data in different variations such as simple arrays, real arrays or JSON format arrays.

### array

Maps and converts array data based on PHP serialization. If you need to store an exact representation of your array data, you should consider using this type as it uses serialization to represent an exact copy of your array as string in the database. Values retrieved from the database are always converted to PHP's `array` type using deserialization or `null` if no data is present.

**Note:** This type will always be mapped to the database vendor's `text` type internally as there is no way of storing a PHP array representation natively in the database. Furthermore this type requires a SQL column comment hint so that it can be reverse engineered from the database. Doctrine cannot map back this type properly on vendors not supporting column comments and will fall back to `text` type instead.

### simple\_array

Maps and converts array data based on PHP comma delimited imploding and exploding. If you know that the data to be stored always is a scalar value based one-dimensional array, you should consider using this type as it uses simple PHP imploding and exploding techniques to serialize and deserialize your data. Values retrieved from the database are always converted to PHP's `array` type using comma delimited `explode()` or `null` if no data is present.

**Note:** This type will always be mapped to the database vendor's `text` type internally as there is no way of storing a PHP array representation natively in the database. Furthermore this type requires a SQL column comment hint so that it can be reverse engineered from the database. Doctrine cannot map back this type properly on vendors not supporting column comments and will fall back to `text` type instead.

**Warning:** You should never rely on a specific PHP type like `boolean`, `integer`, `float` or `null` when retrieving values from the database as the `explode()` deserialization technique used by this type converts every single array item to `string`. This basically means that every array item other than `string` will lose its type awareness.

## json\_array

Maps and converts array data based on PHP's JSON encoding functions. If you know that the data to be stored always is in a valid UTF-8 encoded JSON format string, you should consider using this type. Values retrieved from the database are always converted to PHP's `array` type using PHP's `json_decode()` function.

**Note:** Some vendors have a native JSON type and Doctrine will use it if possible and otherwise silently fall back to the vendor's `text` type to ensure the most efficient storage requirements. If the vendor does not have a native JSON type, this type requires a SQL column comment hint so that it can be reverse engineered from the database. Doctrine cannot map back this type properly on vendors not supporting column comments and will fall back to `text` type instead.

## 8.1.6 Object types

Types that map to objects such as POPOs.

### object

Maps and converts object data based on PHP serialization. If you need to store an exact representation of your object data, you should consider using this type as it uses serialization to represent an exact copy of your object as string in the database. Values retrieved from the database are always converted to PHP's `object` type using deserialization or `null` if no data is present.

**Note:** This type will always be mapped to the database vendor's `text` type internally as there is no way of storing a PHP object representation natively in the database. Furthermore this type requires a SQL column comment hint so that it can be reverse engineered from the database. Doctrine cannot map back this type properly on vendors not supporting column comments and will fall back to `text` type instead.

**Warning:** Because the build-in `text` type of PostgreSQL does not support NULL bytes, the `object` type will cause deserialization errors on PostgreSQL. A workaround is to `serialize()/unserialize()` and `base64_encode()/base64_decode()` PHP objects and store them into a `text` field manually.

## 8.2 Mapping Matrix

The following table shows an overview of Doctrine's type abstraction. The matrix contains the mapping information for how a specific Doctrine type is mapped to the database and back to PHP. Please also notice the mapping specific footnotes for additional information.

Doctrine	PHP	Database vendor		
		Name	Version	Type
<b>smallint</b>	integer	<b>MySQL</b>	<i>all</i>	SMALLINT UNSIGNED <sup>1</sup> AUTO_INCREMENT <sup>2</sup>
		<b>Drizzle</b>	<i>all</i>	INT UNSIGNED <sup>10</sup> AUTO_INCREMENT <sup>11</sup>
		<b>PostgreSQL</b>	<i>all</i>	SMALLINT
		<b>Oracle</b>	<i>all</i>	NUMBER (5)
		<b>SQL Server</b>	<i>all</i>	SMALLINT IDENTITY <sup>11</sup>
		<b>SQL Anywhere</b>	<i>all</i>	UNSIGNED <sup>10</sup> SMALLINT IDENTITY <sup>11</sup>

Continued on next page

<sup>1</sup> Used if **unsigned** attribute is set to `true` in the column definition (default `false`).

<sup>2</sup> Used if **autoincrement** attribute is set to `true` in the column definition (default `false`).

Table 8.1 – continued from previous page

Doctrine	PHP	Database vendor		
		Name	Version	Type
		SQLite	<i>all</i>	INTEGER <sup>3</sup>
integer	integer	MySQL	<i>all</i>	INT UNSIGNED <sup>10</sup> AUTO_INCREMENT <sup>11</sup>
		Drizzle		
		PostgreSQL	<i>all</i>	INT <sup>4</sup>
				SERIAL <sup>11</sup>
		Oracle	<i>all</i>	NUMBER (10)
		SQL Server	<i>all</i>	INT IDENTITY <sup>11</sup>
		SQL Anywhere	<i>all</i>	UNSIGNED <sup>10</sup> INT IDENTITY <sup>11</sup>
		SQLite	<i>all</i>	INTEGER <sup>16</sup>
bigint	string <sup>5</sup>	MySQL	<i>all</i>	BIGINT UNSIGNED <sup>10</sup> AUTO_INCREMENT <sup>11</sup>
		Drizzle		
		PostgreSQL	<i>all</i>	BIGINT <sup>12</sup>
				BIGSERIAL <sup>11</sup>
		Oracle	<i>all</i>	NUMBER (20)
		SQL Server	<i>all</i>	BIGINT IDENTITY <sup>11</sup>
		SQL Anywhere	<i>all</i>	UNSIGNED <sup>10</sup> BIGINT IDENTITY <sup>11</sup>
		SQLite	<i>all</i>	INTEGER <sup>16</sup>
decimal <sup>6</sup>	string <sup>7</sup>	MySQL	<i>all</i>	NUMERIC (p, s)
		PostgreSQL		
		Oracle		
		SQL Server		
		SQL Anywhere		
		SQLite		
		Drizzle		
float	float	MySQL	<i>all</i>	DOUBLE PRECISION
		PostgreSQL		
		Oracle		
		SQL Server		
		SQL Anywhere		
		SQLite		
		Drizzle		
string <sup>8 9</sup>	string	MySQL	<i>all</i>	VARCHAR (n) <sup>10</sup>
		PostgreSQL		CHAR (n) <sup>11</sup>
		SQL Anywhere		
		SQLite	<i>all</i>	CHAR (n) <sup>11</sup>
		Drizzle		CHAR (n) <sup>11</sup>
		Oracle	<i>all</i>	VARCHAR2 (n) <sup>3</sup>
				CHAR (n) <sup>4</sup>
		SQL Server	<i>all</i>	NVARCHAR (n) <sup>3</sup>
				CHAR (n) <sup>4</sup>
				NCHAR (n) <sup>4</sup>
		MySQL	<i>all</i>	TINYTEXT <sup>12</sup>
				TEXT <sup>13</sup>
				MEDIUMTEXT <sup>14</sup>

Continued on next page

**text**                      **string**

<sup>3</sup> Cannot be safely reverse engineered to the same Doctrine type as the vendor does not have a native distinct data type for this mapping. Using this type with this vendor can therefore have implications on schema comparison (*online* vs *offline* schema) and PHP type safety (data conversion from database to PHP value) because it silently falls back to its appropriate Doctrine type.

<sup>4</sup> Chosen if the column definition has the **autoincrement** attribute set to *false* (default).

<sup>12</sup> Chosen if the column length is less or equal to  $2^8 - 1 = 255$ .

<sup>13</sup> Chosen if the column length is less or equal to  $2^{16} - 1 = 65535$ .

<sup>14</sup> Chosen if the column length is less or equal to  $2^{24} - 1 = 16777215$ .

Table 8.1 – continued from previous page

Doctrine	PHP	Database vendor		
		Name	Version	Type
				LONGTEXT <sup>15</sup>
		PostgreSQL	all	TEXT
		SQL Anywhere		
		Drizzle		
		Oracle	all	CLOB
		SQLite		
		SQL Server	all	VARCHAR (MAX)
**guid**	string	MySQL	all	VARCHAR (255) <sup>16</sup>
		Oracle		
		SQLite		
		Drizzle	all	UNIQUEIDENTIFIER
		SQL Server		
		SQL Anywhere	all	UUID
binary <sup>2 17</sup>	resource	MySQL	all	VARBINARY (n) <sup>3</sup>
		SQL Server		BINARY (n) <sup>4</sup>
		SQL Anywhere		
		Drizzle	all	VARBINARY (n)
		Oracle	all	RAW (n)
		PostgreSQL	all	BYTEA <sup>16</sup>
		SQLite	all	BLOB <sup>16</sup>
blob	resource	MySQL	all	TINYBLOB <sup>17</sup>
				BLOB <sup>18</sup>
				MEDIUMBLOB <sup>19</sup>
				LOB <sup>20</sup>
		Oracle	all	BLOB
		SQLite		
		Drizzle		
		SQL Server	all	VARBINARY (MAX)
boolean	boolean	SQL Anywhere	all	LONG BINARY
		PostgreSQL	all	BYTEA
		MySQL	all	TINYINT (1)
		PostgreSQL	all	BOOLEAN
		SQLite		
		Drizzle	all	BIT
		SQL Server		
date	\DateTime	SQL Anywhere	all	NUMBER (1)
		Oracle	all	
		MySQL	all	DATE
		PostgreSQL		
		Oracle		
		SQL Anywhere		
		SQLite		
		Drizzle		
		SQL Server	>= 2008	DATETIME <sup>16</sup>
			< 2008	

Continued on next page

<sup>15</sup> Chosen if the column length is less or equal to  $2^{32} - 1 = 4294967295$  or empty.



Table 8.1 – continued from previous page

Doctrine	PHP	Database vendor		
		Name	Version	Type
datetime	\DateTime	MySQL	all	DATETIME <sup>18</sup>
		Drizzle		TIMESTAMP <sup>19</sup>
		SQL Server	all	DATETIME
		SQL Anywhere		
		SQLite		
		PostgreSQL	all	TIMESTAMP (0) WITHOUT TIME ZONE
		Oracle	all	TIMESTAMP (0)
datetimetz	\DateTime	MySQL	all	DATETIME <sup>20 16</sup>
		Drizzle		
		SQLite		
		SQL Server	< 2008	DATETIMEOFFSET (6)
			>= 2008	
		PostgreSQL	all	TIMESTAMP (0) WITH TIME ZONE
		Oracle		
		SQL Anywhere	< 12	DATETIME <sup>15 16</sup>
			>= 12	TIMESTAMP WITH TIME ZONE
time	\DateTime	MySQL	all	TIME
		SQL Anywhere		
		SQLite		
		Drizzle		
		PostgreSQL	all	TIME (0) WITHOUT TIME ZONE
		Oracle	all	DATE <sup>16</sup>
		SQL Server	< 2008	DATETIME <sup>16</sup>
			>= 2008	TIME (0)
array <sup>1</sup>	array	MySQL	all	TINYTEXT <sup>17</sup>
simple array <sup>1</sup>				TEXT <sup>18</sup>
				MEDIUMTEXT <sup>19</sup>
				LONGTEXT <sup>20</sup>
		PostgreSQL	all	TEXT
		SQL Anywhere		
Drizzle				
Oracle		all	CLOB	
SQLite				
SQL Server	all	VARCHAR (MAX)		
json_array	array	MySQL <sup>1</sup>	all	TINYTEXT <sup>17</sup>
				TEXT <sup>18</sup>
				MEDIUMTEXT <sup>19</sup>
				LONGTEXT <sup>20</sup>
		PostgreSQL	>= 9.2	JSON
			< 9.2	
		SQL Anywhere	all	TEXT <sup>1</sup>
		Drizzle		
		Oracle	all	CLOB <sup>1</sup>
SQLite				
SQL Server	all	VARCHAR (MAX) <sup>1</sup>		
				TINYTEXT <sup>17</sup>
MySQLall				Continued on next page

MySQL

all

Continued on next page

<sup>18</sup> Chosen if the column definition not contains the **version** option inside the **platformOptions** attribute array or is set to **false** which marks it as a non-locking information column.

<sup>19</sup> Chosen if the column definition contains the **version** option inside the **platformOptions** attribute array and is set to **true** which marks it as a locking information column.

Table 8.1 – continued from previous page

Doctrine	PHP	Database vendor		
		Name	Version	Type
				TEXT <sup>18</sup>
				MEDIUMTEXT <sup>19</sup>
				LONGTEXT <sup>20</sup>
		PostgreSQL	<i>all</i>	TEXT
		SQL Anywhere		
		Drizzle		
		Oracle	<i>all</i>	CLOB
		SQLite		
		SQL Server	<i>all</i>	VARCHAR (MAX)

## 8.3 Detection of Database Types

When calling table inspection methods on your connections `SchemaManager` instance the retrieved database column types are translated into Doctrine mapping types. Translation is necessary to allow database abstraction and metadata comparisons for example for Migrations or the ORM SchemaTool.

Each database platform has a default mapping of database types to Doctrine types. You can inspect this mapping for platform of your choice looking at the `AbstractPlatform::initializeDoctrineTypeMappings()` implementation.

If you want to change how Doctrine maps a database type to a `Doctrine\DBAL\Types\Type` instance you can use the `AbstractPlatform::registerDoctrineTypeMapping($dbType, $doctrineType)` method to add new database types or overwrite existing ones.

**Note:** You can only map a database type to exactly one Doctrine type. Database vendors that allow to define custom types like PostgreSQL can help to overcome this issue.

## 8.4 Custom Mapping Types

Just redefining how database types are mapped to all the existing Doctrine types is not at all that useful. You can define your own Doctrine Mapping Types by extending `Doctrine\DBAL\Types\Type`. You are required to implement 4 different methods to get this working.

See this example of how to implement a Money object in PostgreSQL. For this we create the type in PostgreSQL as:

```
CREATE DOMAIN MyMoney AS DECIMAL(18,3);
```

Now we implement our `Doctrine\DBAL\Types\Type` instance:

```
<?php
namespace My\Project\Types;

use Doctrine\DBAL\Types\Type;
use Doctrine\DBAL\Platforms\AbstractPlatform;

/**
 * My custom datatype.
 */
class MoneyType extends Type
{
```

```

const MONEY = 'money'; // modify to match your type name

public function getSqlDeclaration(array $fieldDeclaration, AbstractPlatform $platform)
{
    return 'MyMoney';
}

public function convertToPHPValue($value, AbstractPlatform $platform)
{
    return new Money($value);
}

public function convertToDatabaseValue($value, AbstractPlatform $platform)
{
    return $value->toDecimal();
}

public function getName()
{
    return self::MONEY;
}
}

```

The job of Doctrine-DBAL is to transform your type into SQL declaration. You can modify the SQL declaration Doctrine will produce. At first, you must to enable this feature by overriding the `canRequireSQLConversion` method:

```

<?php
public function canRequireSQLConversion()
{
    return true;
}

```

Then you override the methods `convertToPhpValueSQL` and `convertToDatabaseValueSQL` :

```

<?php
public function convertToPHPValueSQL($sqlExpr, $platform)
{
    return 'MyMoneyFunction(\''.$sqlExpr.'\') ';
}

public function convertToDatabaseValueSQL($sqlExpr, AbstractPlatform $platform)
{
    return 'MyFunction('.$sqlExpr.')';
}

```

Now we have to register this type with the Doctrine Type system and hook it into the database platform:

```

<?php
Type::addType('money', 'My\Project\Types\MoneyType');
$conn->getDatabasePlatform()->registerDoctrineTypeMapping('MyMoney', 'money');

```

This would allow to use a money type in the ORM for example and have Doctrine automatically convert it back and forth to the database.



---

## Schema-Manager

---

A Schema Manager instance helps you with the abstraction of the generation of SQL assets such as Tables, Sequences, Foreign Keys and Indexes.

To retrieve the `SchemaManager` for your connection you can use the `getSchemaManager()` method:

```
<?php
$sm = $conn->getSchemaManager();
```

Now with the `SchemaManager` instance in `$sm` you can use the available methods to learn about your database schema:

**Note:** Parameters containing identifiers passed to the `SchemaManager` methods are *NOT* quoted automatically! Identifier quoting is really difficult to do manually in a consistent way across different databases. You have to manually quote the identifiers when you accept data from user- or other sources not under your control.

---

### 9.1 listDatabases()

Retrieve an array of databases on the configured connection:

```
<?php
$databases = $sm->listDatabases();
```

### 9.2 listSequences()

Retrieve an array of `Doctrine\DBAL\Schema\Sequence` instances that exist for a database:

```
<?php
$sequences = $sm->listSequences();
```

Or if you want to manually specify a database name:

```
<?php
$sequences = $sm->listSequences('dbname');
```

Now you can loop over the array inspecting each sequence object:

```
<?php
foreach ($sequences as $sequence) {
    echo $sequence->getName() . "\n";
}
```

## 9.3 listTableColumns()

Retrieve an array of Doctrine\DBAL\Schema\Column instances that exist for the given table:

```
<?php
$columns = $sm->listTableColumns('user');
```

Now you can loop over the array inspecting each column object:

```
<?php
foreach ($columns as $column) {
    echo $column->getName() . ': ' . $column->getType() . "\n";
}
```

## 9.4 listTableDetails()

Retrieve a single Doctrine\DBAL\Schema\Table instance that encapsulates all the details of the given table:

```
<?php
$table = $sm->listTableDetails('user');
```

Now you can call methods on the table to manipulate the in memory schema for that table. For example we can add a new column:

```
<?php
$table->addColumn('email_address', 'string');
```

## 9.5 listTableForeignKeys()

Retrieve an array of Doctrine\DBAL\Schema\ForeignKeyConstraint instances that exist for the given table:

```
<?php
$foreignKeys = $sm->listTableForeignKeys('user');
```

Now you can loop over the array inspecting each foreign key object:

```
<?php
foreach ($foreignKeys as $foreignKey) {
    echo $foreignKey->getName() . ': ' . $foreignKey->getLocalTableName() . "\n";
}
```

## 9.6 listTableIndexes()

Retrieve an array of Doctrine\DBAL\Schema/Index instances that exist for the given table:

```
<?php
$indexes = $sm->listTableIndexes('user');
```

Now you can loop over the array inspecting each index object:

```
<?php
foreach ($indexes as $index) {
    echo $index->getName() . ': ' . ($index->isUnique() ? 'unique' : 'not unique') . "\n";
}
```

## 9.7 listTables()

Retrieve an array of Doctrine\DBAL\Schema\Table instances that exist in the connections database:

```
<?php
$tables = $sm->listTables();
```

Each Doctrine\DBAL\Schema\Table instance is populated with information provided by all the above methods. So it encapsulates an array of Doctrine\DBAL\Schema\Column instances that can be retrieved with the `getColumns()` method:

```
<?php
foreach ($tables as $table) {
    echo $table->getName() . " columns:\n\n";
    foreach ($table->getColumns() as $column) {
        echo ' - ' . $column->getName() . "\n";
    }
}
```

## 9.8 listViews()

Retrieve an array of Doctrine\DBAL\Schema\View instances that exist in the connections database:

```
<?php
$views = $sm->listViews();
```

Now you can loop over the array inspecting each view object:

```
<?php
foreach ($views as $view) {
    echo $view->getName() . ': ' . $view->getSql() . "\n";
}
```

## 9.9 createSchema()

For a complete representation of the current database you can use the `createSchema()` method which returns an instance of Doctrine\DBAL\Schema\Schema, which you can use in conjunction with the SchemaTool or Schema Comparator.

```
<?php
$fromSchema = $sm->createSchema();
```

Now we can clone the `$fromSchema` to `$toSchema` and drop a table:

```
<?php
$toSchema = clone $fromSchema;
$toSchema->dropTable('user');
```

Now we can compare the two schema instances in order to calculate the differences between them and return the SQL required to make the changes on the database:

```
<?php
$sql = $fromSchema->getMigrateToSql($toSchema, $conn->getDatabasePlatform());
```

The `$sql` array should give you a SQL query to drop the user table:

```
<?php
print_r($sql);

/*
array(
    0 => 'DROP TABLE user'
)
*/
```



---

## Schema-Representation

---

Doctrine has a very powerful abstraction of database schemas. It offers an object-oriented representation of a database schema with support for all the details of Tables, Sequences, Indexes and Foreign Keys. These Schema instances generate a representation that is equal for all the supported platforms. Internally this functionality is used by the ORM Schema Tool to offer you create, drop and update database schema methods from your Doctrine ORM Metadata model. Up to very specific functionality of your database system this allows you to generate SQL code that makes your Domain model work.

You will be pleased to hear, that Schema representation is completely decoupled from the Doctrine ORM though, that is you can also use it in any other project to implement database migrations or for SQL schema generation for any metadata model that your application has. You can easily generate a Schema, as a simple example shows:

```
<?php
$хсhema = new \Doctrine\DBAL\Schema\Schema();
$myTable = $schema->createTable("my_table");
$myTable->addColumn("id", "integer", array("unsigned" => true));
$myTable->addColumn("username", "string", array("length" => 32));
$myTable->setPrimaryKey(array("id"));
$myTable->addUniqueIndex(array("username"));
$хсhema->createSequence("my_table_seq");

$myForeign = $schema->createTable("my_foreign");
$myForeign->addColumn("id", "integer");
$myForeign->addColumn("user_id", "integer");
$myForeign->addForeignKeyConstraint($myTable, array("user_id"), array("id"), array("onUpdate" => "CASCADE"));

$queries = $schema->toSql($myPlatform); // get queries to create this schema.
$dropSchema = $schema->toDropSql($myPlatform); // get queries to safely delete this schema.
```

Now if you want to compare this schema with another schema, you can use the Comparator class to get instances of SchemaDiff, TableDiff and ColumnDiff, as well as information about other foreign key, sequence and index changes.

```
<?php
$comparator = new \Doctrine\DBAL\Schema\Comparator();
$хсhemaDiff = $comparator->compare($fromSchema, $toSchema);

$queries = $хсhemaDiff->toSql($myPlatform); // queries to get from one to another schema.
$saveQueries = $хсhemaDiff->toSaveSql($myPlatform);
```

The Save Diff mode is a specific mode that prevents the deletion of tables and sequences that might occur when making a diff of your schema. This is often necessary when your target schema is not complete but only describes a subset of your application.

All methods that generate SQL queries for you make much effort to get the order of generation correct, so that no problems will ever occur with missing links of foreign keys.

## 10.1 Schema Assets

A schema asset is considered any abstract atomic unit in a database such as schemas, tables, indexes, but also sequences, columns and even identifiers. The following chapter gives an overview of all available Doctrine 2 schema assets with short explanations on their context and usage. All schema assets reside in the `Doctrine\DBAL\Schema` namespace.

---

**Note:** This chapter is far from being completely documented.

---

### 10.1.1 Column

Represents a table column in the database schema. A column consists of a name, a type, portable options, commonly supported options and vendors specific options.

#### Portable options

The following options are considered to be fully portable across all database platforms:

- **notnull** (boolean): Whether the column is nullable or not. Defaults to `true`.
- **default** (integer|string): The default value of the column if no value was specified. Defaults to `null`.
- **autoincrement** (boolean): Whether this column should use an autoincremented value if no value was specified. Only applies to Doctrine's `smallint`, `integer` and `bigint` types. Defaults to `false`.
- **length** (integer): The maximum length of the column. Only applies to Doctrine's `string` and `binary` types. Defaults to `null` and is evaluated to 255 in the platform.
- **fixed** (boolean): Whether a `string` or `binary` Doctrine type column has a fixed length. Defaults to `false`.
- **precision** (integer): The precision of a Doctrine `decimal` or `float` type column that determines the overall maximum number of digits to be stored (including scale). Defaults to 10.
- **scale** (integer): The exact number of decimal digits to be stored in a Doctrine `decimal` or `float` type column. Defaults to 0.
- **customSchemaOptions** (array): Additional options for the column that are supported by all vendors:
  - **unique** (boolean): Whether to automatically add a unique constraint for the column. Defaults to `false`.

#### Common options

The following options are not completely portable but are supported by most of the vendors:

- **unsigned** (boolean): Whether a `smallint`, `integer` or `bigint` Doctrine type column should allow unsigned values only. Supported by MySQL, SQL Anywhere and Drizzle. Defaults to `false`.
- **comment** (integer|string): The column comment. Supported by MySQL, PostgreSQL, Oracle, SQL Server, SQL Anywhere and Drizzle. Defaults to `null`.

## Vendor specific options

The following options are completely vendor specific and absolutely not portable:

- **columnDefinition**: The custom column declaration SQL snippet to use instead of the generated SQL by Doctrine. Defaults to `null`. This can be useful to add vendor specific declaration information that is not evaluated by Doctrine (such as the `ZEROFILL` attribute on MySQL).
- **customSchemaOptions** (array): Additional options for the column that are supported by some vendors but not portable:
  - **charset** (string): The character set to use for the column. Currently only supported on MySQL and Drizzle.
  - **collate** (string): The collation to use for the column. Currently only supported on SQL Server.
  - **check** (string): The check constraint clause to add to the column. Defaults to `null`.



---

## Events

---

Both `Doctrine\DBAL\DriverManager` and `Doctrine\DBAL\Connection` accept an instance of `Doctrine\Common\EventManager`. The `EventManager` has a couple of events inside the DBAL layer that are triggered for the user to listen to.

### 11.1 PostConnect Event

`Doctrine\DBAL\Events::postConnect` is triggered right after the connection to the database is established. It allows to specify any relevant connection specific options and gives access to the `Doctrine\DBAL\Connection` instance that is responsible for the connection management via an instance of `Doctrine\DBAL\Event\ConnectionEventArgs` event arguments instance.

Doctrine ships with one implementation for the “PostConnect” event:

- `Doctrine\DBAL\Event\Listeners\OracleSessionInit` allows to specify any number of Oracle Session related environment variables that are set right after the connection is established.

You can register events by subscribing them to the `EventManager` instance passed to the Connection factory:

```
<?php
$evm = new EventManager();
$evm->addEventSubscriber(new OracleSessionInit(array(
    'NLS_TIME_FORMAT' => 'HH24:MI:SS',
)));

$conn = DriverManager::getConnection($connectionParams, null, $evm);
```



---

## Security

---

Allowing users of your website to communicate with a database can possibly have security implications that you should be aware of. Databases allow very powerful commands that not every user of your website should be able to execute. Additionally the data in your database probably contains information that should not be visible to everyone with access to the website.

The most dangerous security problem with regard to databases is the possibility of SQL injections. An SQL injection security hole allows an attacker to execute new or modify existing SQL statements to access information that he is not allowed to access.

Neither Doctrine DBAL nor ORM can prevent such attacks if you are careless as a developer. This section explains to you the problems of SQL injection and how to prevent them.

### 12.1 SQL Injection: Safe and Unsafe APIs for User Input

A database library naturally falls touches the class of SQL injection security vulnerabilities. You should read the following information carefully to understand how Doctrine can and cannot help you to prevent SQL injection.

In general you should assume that APIs in Doctrine are not safe for user input. There are however some exceptions.

The following APIs are designed to be **SAFE** from SQL injections:

- For `Doctrine\DBAL\Connection#insert($table, $values, $types)`, `Doctrine\DBAL\Connection#update($table, $values, $where, $types)` and `Doctrine\DBAL\Connection#delete($table, $where, $types)` only the array values of `$values` and `$where`. The table name and keys of `$values` and `$where` are NOT escaped.
- `Doctrine\DBAL\Query\QueryBuilder#setFirstResult($offset)`
- `Doctrine\DBAL\Query\QueryBuilder#setMaxResults($limit)`
- `Doctrine\DBAL\Platforms\AbstractPlatform#modifyLimitQuery($sql, $limit, $offset)` for the `$limit` and `$offset` parameters.

Consider **ALL** other APIs to be not safe for user-input:

- Query methods on the Connection
- The QueryBuilder API
- The Platforms and SchemaManager APIs to generate and execute DML/DDDL SQL statements

To escape user input in those scenarios use the `Connection#quote()` method.

## 12.2 User input in your queries

A database application necessarily requires user-input to be passed to your queries. There are wrong and right ways to do this and it is very important to be very strict about this:

### 12.2.1 Wrong: String Concatenation

You should never ever build your queries dynamically and concatenate user-input into your SQL or DQL query. For Example:

```
<?php
// Very wrong!
$sql = "SELECT * FROM users WHERE name = '" . $_GET['username'] . "'";
```

An attacker could inject any value into the GET variable “username” to modify the query to his needs.

Although DQL is a wrapper around SQL that can prevent you from some security implications, the previous example is also a threat to DQL queries.

```
<?php
// DQL is not safe against arbitrary user-input as well:
$dql = "SELECT u FROM User u WHERE u.username = '" . $_GET['username'] . "'";
```

In this scenario an attacker could still pass a username set to `' OR 1 = 1` and create a valid DQL query. Although DQL will make use of quoting functions when literals are used in a DQL statement, allowing the attacker to modify the DQL statement with valid literals cannot be detected by the DQL parser, it is your responsibility.

### 12.2.2 Right: Prepared Statements

You should always use prepared statements to execute your queries. Prepared statements is a two-step procedure, separating SQL query from the parameters. They are supported (and encouraged) for both DBAL SQL queries and for ORM DQL queries.

Instead of using string concatenation to insert user-input into your SQL/DQL statements you just specify either placeholders instead and then explain to the database driver which variable should be bound to which placeholder. Each database vendor supports different placeholder styles:

- All PDO Drivers support positional (using question marks) and named placeholders (:param1, :foo, :bar).
- OCI8 only supports named parameters, but Doctrine DBAL has a thin layer around OCI8 and also allows positional placeholders.
- Doctrine ORM DQL allows both named and positional parameters. The positional parameters however are not just question marks, but suffixed with a number (?1, ?2, ?3, ...).

Following are examples of using prepared statements with SQL and DQL:

```
<?php
// SQL Prepared Statements: Positional
$sql = "SELECT * FROM users WHERE username = ?";
$stmt = $connection->prepare($sql);
$stmt->bindValue(1, $_GET['username']);
$stmt->execute();

// SQL Prepared Statements: Named
$sql = "SELECT * FROM users WHERE username = :user";
$stmt = $connection->prepare($sql);
```



```

$stmt->bindValue("user", $_GET['username']);
$stmt->execute();

// DQL Prepared Statements: Positional
$sql = "SELECT u FROM User u WHERE u.username = ?1";
$query = $em->createQuery($sql);
$query->setParameter(1, $_GET['username']);
$data = $query->getResult();

// DQL Prepared Statements: Named
$sql = "SELECT u FROM User u WHERE u.username = :name";
$query = $em->createQuery($sql);
$query->setParameter("name", $_GET['username']);
$data = $query->getResult();

```

You can see this is a bit more tedious to write, but this is the only way to write secure queries. If you are using just the DBAL there are also helper methods which simplify the usage quite a lot:

```

<?php
// bind parameters and execute query at once.
$sql = "SELECT * FROM users WHERE username = ?";
$stmt = $connection->executeQuery($sql, array($_GET['username']));

```

There is also `executeUpdate` which does not return a statement but the number of affected rows.

Besides binding parameters you can also pass the type of the variable. This allows Doctrine or the underlying vendor to not only escape but also cast the value to the correct type. See the docs on querying and DQL in the respective chapters for more information.

### 12.2.3 Right: Quoting/Escaping values

Although previously we said string concatenation is wrong, there is a way to do it correctly using the `Connection#quote` method:

```

<?php
// Parameter quoting
$sql = "SELECT * FROM users WHERE name = " . $connection->quote($_GET['username'], \PDO::PARAM_STR);

```

This method is only available for SQL, not for DQL. For DQL it is always encouraged to use prepared statements not only for security, but also for caching reasons.

## 12.3 Non-ASCII compatible Charsets in MySQL

Up until PHP 5.3.6 PDO has a security problem when using non ascii compatible charsets. Even if specifying the charset using “SET NAMES”, emulated prepared statements and `PDO#quote` could not reliably escape values, opening up to potential SQL injections. If you are running PHP 5.3.6 you can solve this issue by passing the driver option “charset” to Doctrine PDO MySQL driver. Using SET NAMES does not suffice!



---

## Sharding

---

---

**Note:** The sharding extension is currently in transition from a separate Project into DBAL. Class names may differ.

---

Starting with 2.3 Doctrine DBAL contains some functionality to simplify the development of horizontally sharded applications. In this first release it contains a `ShardManager` interface. This interface allows to programatically select a shard to send queries to. At the moment there are no functionalities yet to dynamically pick a shard based on ID, query or database row yet. That means the sharding extension is primarily suited for:

- multi-tenant applications or
- applications with completely separated datasets (example: weather data).

Both kind of application will work with both DBAL and ORM.

---

**Note:** Horizontal sharding is an evasive architecture that will affect your application code and using this extension to Doctrine will not make it work “magically”.

---

You have to understand and integrate the following drawbacks:

- Pre-generation of IDs that are unique across all shards required.
- No transaction support across shards.
- No foreign key support across shards (meaning no “real” relations).
- Very complex (or impossible) to query aggregates across shards.
- Denormalization: Composite keys required where normalized non-sharded db schemas don’t need them.
- Schema Operations have to be done on all shards.

The primary questions in a sharding architecture are:

- Where is my data located?
- Where should I save this new data to find it later?

To answer these questions you generally have to craft a function that will tell you for a given ID, on which shard the data for this ID is located. To simplify this approach you will generally just pick a table which is the root of a set of related data and decide for the IDs of this table. All the related data that belong to this table are saved on the same shard.

Take for example a multi-user blog application with the following tables:

- Blog [id, name]
- Post [id, blog\_id, subject, body, author\_id]

- Comment [id, post\_id, comment, author\_id]
- User [id, username]

A sensible sharding architecture will split the application by blog. That means all the data for a particular blog will be on a single shard and scaling is done by putting the amount of blogs on many different database servers.

Now users can post and comment on different blogs that reside on different shards. This makes the database schema above slightly tricky, because both *author\_id* columns cannot have foreign keys to *User (id)*. Instead the User table is located in an entirely different “dimension” of the application in terms of the sharding architecture.

To simplify working with this kind of multi-dimensional database schema, you can replace the *author\_ids* with something more “meaningful”, for example the e-mail address of the users if that is always known. The “user” table can then be separated from the database schema above and put on a second horizontally scaled sharding architecture.

As you can see, even with just the four tables above, sharding actually becomes quite complex to think about.

The rest of this section discusses Doctrine sharding functionality in technical detail.

## 13.1 ID Generation

To solve the issue of unique ID-generation across all shards are several approaches you should evaluate:

### 13.1.1 Use GUID/UUIDs

The most simple ID-generation mechanism for sharding are universally unique identifiers. These are 16-byte (128-bit) numbers that are guaranteed to be unique across different servers. You can [read up on UUIDs on Wikipedia](#).

The drawback of UUIDs is the segmentation they cause on indexes. Because UUIDs are not sequentially generated, they can have negative impact on index access performance. Additionally they are much bigger than numerical primary keys (which are normally 4-bytes in length).

At the moment Doctrine DBAL drivers MySQL and SQL Server support the generation of UUID/GUIDs. You can use the following bit of code to generate them across platforms:

```
<?php
use Doctrine\DBAL\DriverManager;

$conn = DriverManager::getConnection(/**...*/);
$guid = $conn->fetchColumn('SELECT ' . $conn->getDatabasePlatform()->getGuidExpression());

$conn->insert("my_table", array("id" => $guid, "foo" => "bar"));
```

In your application you should hide this details in Id-Generation services:

```
<?php
namespace MyApplication;

class IdGenerationService
{
    private $conn;

    public function generateCustomerId()
    {
        return $this->conn->fetchColumn('SELECT ' .
            $this->conn->getDatabasePlatform()->getGuidExpression()
        );
    }
}
```

```
}
}
```

A good starting point to read up on GUIDs (vs numerical ids) is this blog post [Coding Horror: Primary Keys: IDs vs GUIDs](#).

### 13.1.2 Table Generator

In some scenarios there is no way around a numerical, automatically incrementing id. The way Auto incrementing IDs are implemented in MySQL and SQL Server however is completely unsuitable for sharding. Remember in a sharding architecture you have to know where the row for a specific ID is located and IDs have to be globally unique across all servers. Auto-Increment Primary Keys are missing both properties.

To get around this issue you can use the so-called “table-generator” strategy. In this case you define a single database that is responsible for the generation of auto-incremented ids. You create a table on this database and through the use of locking create new sequential ids.

There are three important drawbacks to this strategy:

- Single point of failure
- Bottleneck when application is write-heavy
- A second independent database connection is needed to guarantee transaction safety.

If you can live with this drawbacks then you can use table-generation with the following code in Doctrine:

```
<?php
use Doctrine\DBAL\DriverManager;
use Doctrine\DBAL\Id\TableGenerator;

$conn = DriverManager::getConnection(/**...**); // connection 1

// creating the TableGenerator automatically opens a second connection.
$tableGenerator = new TableGenerator($conn, "sequences_tbl_name");

$id1 = $tableGenerator->nextValue("sequence_name1");
$id2 = $tableGenerator->nextValue("sequence_name2");
```

The table generator obviously needs a table to work. The schema of this table is described in the `TableGenerator` class-docblock. Alternatively you can use the `Doctrine\DBAL\Id\TableGeneratorSchemaVisitor` and apply it to your `Doctrine\DBAL\Schema\Schema` instance. It will automatically add the required sequence table.

### 13.1.3 Natural Identifiers

Sometimes you are lucky and your application data-model comes with a natural id. This is mostly the case for applications who get their IDs generated somewhere else (exogeneous ID-generation) or that work with temporal data. In that case you can just define the natural primary key and shard your application based on this data.

## 13.2 Transactions

Transactions in sharding can only work for data that is located on a single shard. If you need transactions in your sharding architecture then you have to make sure that the data updated during a transaction is located on a single shard.

## 13.3 Foreign Keys

Since you cannot create foreign keys between remote database servers, in a sharding architecture you should put the data on a shard that belongs to each other. But even if you can isolate most of the rows on a single shard there may exist relations between tables that exist on different shards. In this case your application should be aware of the potential inconsistencies and handle them graciously.

## 13.4 Complex Queries

GROUP BY, DISTINCT and ORDER BY are clauses that cannot be easily used in a sharding architecture. If you have to execute these queries against multiple shards then you cannot just append the different results to each other.

You have to be aware of this problem and design your queries accordingly or shard the data in a way that you never have to query multiple shards to calculate a result.

## 13.5 ShardManager Interface

The central API of the sharding extension is the `ShardManager` interface. It contains two different groups of functions with regard to sharding.

First, it contains the Shard Selection API. You can pick a shard based on a so-called “distribution-value” or reset the connection to the “global” shard, a necessary database that often contains heavily cached, sharding independent data such as meta tables or the “user/tenant” table.

```
<?php
use Doctrine\DBAL\DriverManager;
use Doctrine\Shards\DBAL\SQLAzure\SQLAzureShardManager;

$conn = DriverManager::getConnection(array(
    'sharding' => array(
        'federationName' => 'my_database',
        'distributionKey' => 'customer_id',
    )
));
$shardManager = new SQLAzureShardManager($conn);

$currentCustomerId = 1234;
$shardManager->selectShard($currentCustomerId);
// all queries after this call hit the shard
// where customer with id 1234 is on.

$shardManager->selectGlobal();
// the global database is selected.
```

To access the currently selected distribution value use the following API method:

```
<?php
$value = $shardManager->getCurrentDistributionValue();
```

The shard manager will prevent you switching shards when a transaction is open. This is especially important when using sharding with the ORM. Because the ORM uses a single transaction during the flush-operation this means that you can only ever use one `EntityManager` with data from a single shard.

The second API is the “fan-out” query API. This allows you to execute queries against ALL shards. The order of the results of this operation is undefined, that means your query has to return the data in a way that works for the application, or you have to sort the data in the application.

```
<?php
$sql = "SELECT * FROM customers";
$rows = $shardManager->queryAll($sql, $params);
```

## 13.6 Schema Operations: SchemaSynchronizer Interface

Schema Operations in a sharding architecture are tricky. You have to perform them on all databases instances (shards) at the same time. Also Doctrine has problems with this in particular as you cannot generate an SQL file with changes on any development machine anymore and apply this on production. The required changes depend on the amount of shards.

To allow the Doctrine Schema API operations on a sharding architecture we performed a refactored from code inside ORM `Doctrine\ORM\Tools\SchemaTool` class and extracted the code for operations on Schema instances into a new `Doctrine\Shards\DBAL\SchemaSynchronizer` interface.

Every sharding implementation can implement this interface and allow schema operations to take part on multiple shards.

## 13.7 SQL Azure Federations

Doctrine Shards ships with a custom implementation for Microsoft SQL Azure. The Azure platform provides a native sharding functionality. In SQL Azure the sharding functionality is called Federations. This functionality applies the following restrictions (in line with the ones listed above):

- IDENTITY columns are not allowed on sharded tables (federated tables)
- Each table may only have exactly one clustered index and this index has to have the distribution key/sharding-id as one column.
- Every unique index (or primary key) has to contain the distribution-key/sharding-id.

Especially the requirements 2 and 3 prevent normalized database schemas. You have to put the distribution key on every sharded table, which can affect your application code quite a bit. This may lead to the creation of composite keys where you normally wouldn't need them.

The benefit of SQL Azure Federations is that they implement all the shard-picking logic on the server. You only have to make use of the `USE FEDERATION` statement. You don't have to maintain a list of all the shards inside your application and more importantly, resizing shards is done transparently on the server.

Features of SQL Azure are:

- Central server to log into federations architecture. No need to know all connection details of all shards.
- Database level operation to split shards, taking away the tediousness of this operation for application developers.
- A global tablespace that can contain global data to all shards.
- One or many different federations (this library only supports working with one)
- Sharded or non-sharded tables inside federations
- Allows filtering SELECT queries on the database based on the selected sharding key value. This allows to implement sharded Multi-Tenant Apps very easily.

To setup an SQL Azure ShardManager use the following code:

```
<?php
use Doctrine\DBAL\DriverManager;
use Doctrine\Shards\DBAL\SQLAzure\SQLAzureShardManager;

$conn = DriverManager::getConnection(array(
    'dbname' => 'my_database',
    'host' => 'tcp:dbname.windows.net',
    'user' => 'user@dbname',
    'password' => 'XXX',
    'sharding' => array(
        'federationName' => 'my_federation',
        'distributionKey' => 'customer_id',
        'distributionType' => 'integer',
    )
));
$shardManager = new SQLAzureShardManager($conn);
```

Currently you are limited to one federation in your application.

You can inspect all the currently known shards on SQL Azure using the `ShardManager#getShards()` function:

```
<?php
foreach ($shardManager->getShards() as $shard) {
    echo $shard['id'] . " " . $shard['rangeLow'] . " - " . $shard['rangeHigh'];
}
```

### 13.7.1 Schema Operations

Schema Operations on SQL Azure Federations are possible with the `SQLAzureSchemaSynchronizer`. You can instantiate this from your code:

```
<?php
use Doctrine\Shards\DBAL\SQLAzure\SQLAzureSchemaSynchronizer;

$synchronizer = new SQLAzureSchemaSynchronizer($conn, $shardManager);
```

You can use the API such as `createSchema($schema)` then and it will be distributed across all shards. The assumptions are:

- Using `SchemaSynchronizer#createSchema()` assumes the database is empty. The federation is created during this operation.
- Using `SchemaSynchronizer#updateSchema()` assumes the database and the federation exists. All shards of the federation are iterated and update is applied to all shards consecutively.

For a schema with tables in the global or federated sub-schema you have to use the Schema API to mark tables:

```
<?php
use Doctrine\DBAL\Schema\Schema;

$schema = new Schema();

// no options set, this table will be on the federation root
$users = $schema->createTable('Users');
//...

// marked as sharded, but no distribution column given:
```



```
// non-federated table inside the federation
$products = $schema->createTable('Products');
$products->addOption('azure.federated', true);
//...

// shared + distribution column:
// federated table
$customers = $schema->createTable('Customers');
$customers->addColumn('CustomerID', 'integer');
//...
$customers->addOption('azure.federated', true);
$customers->addOption('azure.federatedOnColumnName', 'CustomerID');
```

### 13.7.2 SQLAzure Filtering

SQL Azure comes with a powerful filtering feature, that allows you to automatically implement a multi-tenant application for a formerly single-tenant application. The restriction to make this work is that your application does not work with IDENTITY columns.

Normally when you select a shard using `ShardManager#selectShard()` any query executed against this shard will return data from ALL the tenants located on this shard. With the “FILTERING=ON” flag on the USE FEDERATION query however SQL Azure can automatically filter all SELECT queries with the chosen distribution value. Additionally you can automatically set the currently selected distribution value in every INSERT statement using a function for this value as the DEFAULT part of the column. If you are using GUIDs for every row then UPDATE and DELETE statements using only GUIDs will work out perfectly as well, as they are by definition for unique rows. This feature allows you to build multi-tenant applications, even though they were not originally designed that way.

To enable filtering you can use the `SQLAzureShardManager#setFilteringEnabled()` method. This method is not part of the interface. You can also set a default value for filtering by passing it as the “shard-ing.filteringEnabled” parameter to `DriverManager#getConnection()`.

## 13.8 Generic SQL Sharding Support

Besides the custom SQL Azure support there is a generic implementation that works with all database drivers. It requires to specify all database connections and will switch between the different connections under the hood when using the `ShardManager` API. This is also the biggest drawback of this approach, since fan-out queries need to connect to all databases in a single request.

See the configuration for a sample sharding connection:

```
<?php
use Doctrine\DBAL\DriverManager;

$conn = DriverManager::getConnection(array(
    'wrapperClass' => 'Doctrine\Shards\DBAL\PoolingShardConnection',
    'driver'       => 'pdo_sqlite',
    'global'       => array('memory' => true),
    'shards'       => array(
        array('id' => 1, 'memory' => true),
        array('id' => 2, 'memory' => true),
    ),
    'shardChoser' => 'Doctrine\Shards\DBAL\ShardChoser\MultiTenantShardChoser',
));
```

You have to configure the following options:

- 'wrapperClass' - Selecting the PoolingShardConnection as above.
- 'global' - An array of database parameters that is used for connecting to the global database.
- 'shards' - An array of shard database parameters. You have to specify an 'id' parameter for each of the shard configurations.
- 'shardChoser' - Implementation of the `Doctrine\Shards\DBAL\ShardChoser\ShardChoser` interface.

The Shard Choser interface maps the distribution value to a shard-id. This gives you the freedom to implement your own strategy for sharding the data horizontally.

---

## SQLAzure Sharding Tutorial

---

---

**Note:** The sharding extension is currently in transition from a separate Project into DBAL. Class names may differ.

---

This tutorial builds upon the [Brian Swans tutorial](#) on SQLAzure Sharding and turns all the examples into examples using the Doctrine Sharding support.

It introduces SQL Azure Sharding, which is an abstraction layer in SQL Azure to support sharding. Many features for sharding are implemented on the database level, which makes it much easier to work with than generic sharding implementations.

For this tutorial you need an Azure account. You don't need to deploy the code on Azure, you can run it from your own machine against the remote database.

---

**Note:** You can look at the code from the 'examples/sharding' directory.

---

### 14.1 Install Doctrine

For this tutorial we will install Doctrine and the Sharding Extension through [Composer](#) which is the easiest way to install Doctrine. Composer is a new package manager for PHP. Download the `composer.phar` from their website and put it into a newly created folder for this tutorial. Now create a `composer.json` file in this project root with the following content:

```
{
    "require": {
        "doctrine/dbal": "2.2.2",
        "doctrine/shards": "0.2"
    }
}
```

Open up the commandline and switch to your tutorial root directory, then call `php composer.phar install`. It will grab the code and install it into the `vendor` subdirectory of your project. It also creates an autoloader, so that we don't have to care about this.

### 14.2 Setup Connection

The first thing to start with is setting up Doctrine and the database connection:

```
<?php
// bootstrap.php
use Doctrine\DBAL\DriverManager;
use Doctrine\Shards\DBAL\SQLAzure\SQLAzureShardManager;

require_once "vendor/autoload.php";

$conn = DriverManager::getConnection(array(
    'driver' => 'pdo_sqlsrv',
    'dbname' => 'SalesDB',
    'host' => 'tcp:dbname.windows.net',
    'user' => 'user@dbname',
    'password' => 'XXX',
    'platform' => new \Doctrine\DBAL\Platforms\SQLAzurePlatform(),
    'driverOptions' => array('MultipleActiveResultSets' => false),
    'sharding' => array(
        'federationName' => 'Orders_Federation',
        'distributionKey' => 'CustId',
        'distributionType' => 'integer',
    )
));

$shardManager = new SQLAzureShardManager($conn);
```

## 14.3 Create Database

Create a new database using the Azure/SQL Azure management console.

## 14.4 Create Schema

Doctrine has a powerful schema API. We don't need to use low-level DDL statements to generate the database schema. Instead you can use an Object-Oriented API to create the database schema and then have Doctrine turn it into DDL statements.

We will recreate Brians example schema with Doctrine DBAL. Instead of having to create federations and schema separately as in his example, Doctrine will do it all in one step:

```
<?php
// create_schema.php
use Doctrine\DBAL\Schema\Schema;
use Doctrine\Shards\DBAL\SQLAzure\SQLAzureSchemaSynchronizer;

require_once 'bootstrap.php';

$schema = new Schema();

$products = $schema->createTable('Products');
$products->addColumn('ProductID', 'integer');
$products->addColumn('SupplierID', 'integer');
$products->addColumn('ProductName', 'string');
$products->addColumn('Price', 'decimal', array('scale' => 2, 'precision' => 12));
$products->setPrimaryKey(array('ProductID'));
$products->addOption('azure.federated', true);
```

```

$customers = $schema->createTable('Customers');
$customers->addColumn('CustomerID', 'integer');
$customers->addColumn('CompanyName', 'string');
$customers->addColumn('FirstName', 'string');
$customers->addColumn('LastName', 'string');
$customers->setPrimaryKey(array('CustomerID'));
$customers->addOption('azure.federated', true);
$customers->addOption('azure.federatedOnColumnName', 'CustomerID');

$orders = $schema->createTable('Orders');
$orders->addColumn('CustomerID', 'integer');
$orders->addColumn('OrderID', 'integer');
$orders->addColumn('OrderDate', 'datetime');
$orders->setPrimaryKey(array('CustomerID', 'OrderID'));
$orders->addOption('azure.federated', true);
$orders->addOption('azure.federatedOnColumnName', 'CustomerID');

$orderItems = $schema->createTable('OrderItems');
$orderItems->addColumn('CustomerID', 'integer');
$orderItems->addColumn('OrderID', 'integer');
$orderItems->addColumn('ProductID', 'integer');
$orderItems->addColumn('Quantity', 'integer');
$orderItems->setPrimaryKey(array('CustomerID', 'OrderID', 'ProductID'));
$orderItems->addOption('azure.federated', true);
$orderItems->addOption('azure.federatedOnColumnName', 'CustomerID');

// Create the Schema + Federation:
$synchronizer = new SQLAzureSchemaSynchronizer($conn, $shardManager);
$synchronizer->createSchema($schema);

// Or jut look at the SQL:
echo implode("\n", $synchronizer->getCreateSchema($schema));

```

## 14.5 View Federation Members

To see how many shard instances (called Federation Members) your SQLAzure database currently has you can ask the `ShardManager` to enumerate all shards:

```

<?php
// view_federation_members.php
require_once "bootstrap.php";

$shards = $shardManager->getShards();
foreach ($shards as $shard) {
    print_r($shard);
}

```

## 14.6 Insert Data

Now we want to insert some test data into the database to see the behavior when we split the shards. We use the same test data as Brian, but use the Doctrine API to insert them. To insert data into federated tables we have to select the shard we want to put the data into. We can use the `ShardManager` to execute this operation for us:

```
<?php
// insert_data.php
require_once "bootstrap.php";

$shardManager->selectShard(0);

$conn->insert("Products", array(
    "ProductID" => 386,
    "SupplierID" => 1001,
    "ProductName" => 'Titanium Extension Bracket Left Hand',
    "Price" => 5.25,
));
$conn->insert("Products", array(
    "ProductID" => 387,
    "SupplierID" => 1001,
    "ProductName" => 'Titanium Extension Bracket Right Hand',
    "Price" => 5.25,
));
$conn->insert("Products", array(
    "ProductID" => 388,
    "SupplierID" => 1001,
    "ProductName" => 'Fusion Generator Module 5 kV',
    "Price" => 10.50,
));
$conn->insert("Products", array(
    "ProductID" => 388,
    "SupplierID" => 1001,
    "ProductName" => 'Bypass Filter 400 MHz Low Pass',
    "Price" => 10.50,
));

$conn->insert("Customers", array(
    'CustomerID' => 10,
    'CompanyName' => 'Van Nuys',
    'FirstName' => 'Catherine',
    'LastName' => 'Abel',
));
$conn->insert("Customers", array(
    'CustomerID' => 20,
    'CompanyName' => 'Abercrombie',
    'FirstName' => 'Kim',
    'LastName' => 'Branch',
));
$conn->insert("Customers", array(
    'CustomerID' => 30,
    'CompanyName' => 'Contoso',
    'FirstName' => 'Frances',
    'LastName' => 'Adams',
));
$conn->insert("Customers", array(
    'CustomerID' => 40,
    'CompanyName' => 'A. Datum Corporation',
    'FirstName' => 'Mark',
    'LastName' => 'Harrington',
));
$conn->insert("Customers", array(
    'CustomerID' => 50,
    'CompanyName' => 'Adventure Works',
```

```

        'FirstName' => 'Keith',
        'LastName' => 'Harris',
    ));
$conn->insert("Customers", array(
    'CustomerID' => 60,
    'CompanyName' => 'Alpine Ski House',
    'FirstName' => 'Wilson',
    'LastName' => 'Pais',
));
$conn->insert("Customers", array(
    'CustomerID' => 70,
    'CompanyName' => 'Baldwin Museum of Science',
    'FirstName' => 'Roger',
    'LastName' => 'Harui',
));
$conn->insert("Customers", array(
    'CustomerID' => 80,
    'CompanyName' => 'Blue Yonder Airlines',
    'FirstName' => 'Pilar',
    'LastName' => 'Pinilla',
));
$conn->insert("Customers", array(
    'CustomerID' => 90,
    'CompanyName' => 'City Power & Light',
    'FirstName' => 'Kari',
    'LastName' => 'Hensien',
));
$conn->insert("Customers", array(
    'CustomerID' => 100,
    'CompanyName' => 'Coho Winery',
    'FirstName' => 'Peter',
    'LastName' => 'Brehm',
));

$conn->executeUpdate("DECLARE @orderId INT

    DECLARE @customerId INT

    SET @orderId = 10
    SELECT @customerId = CustomerId FROM Customers WHERE LastName = 'Hensien' and FirstName = 'Kari'

    INSERT INTO Orders (CustomerId, OrderId, OrderDate)
    VALUES (@customerId, @orderId, GetDate())

    INSERT INTO OrderItems (CustomerId, OrderID, ProductID, Quantity)
    VALUES (@customerId, @orderId, 388, 4)

    SET @orderId = 20
    SELECT @customerId = CustomerId FROM Customers WHERE LastName = 'Harui' and FirstName = 'Roger'

    INSERT INTO Orders (CustomerId, OrderId, OrderDate)
    VALUES (@customerId, @orderId, GetDate())

    INSERT INTO OrderItems (CustomerId, OrderID, ProductID, Quantity)
    VALUES (@customerId, @orderId, 389, 2)

    SET @orderId = 30
    SELECT @customerId = CustomerId FROM Customers WHERE LastName = 'Brehm' and FirstName = 'Peter'

```

```

INSERT INTO Orders (CustomerId, OrderId, OrderDate)
VALUES (@customerId, @orderId, GetDate())

INSERT INTO OrderItems (CustomerId, OrderID, ProductID, Quantity)
VALUES (@customerId, @orderId, 387, 3)

SET @orderId = 40
SELECT @customerId = CustomerId FROM Customers WHERE LastName = 'Pais' and FirstName = 'Wilson'

INSERT INTO Orders (CustomerId, OrderId, OrderDate)
VALUES (@customerId, @orderId, GetDate())

INSERT INTO OrderItems (CustomerId, OrderID, ProductID, Quantity)
VALUES (@customerId, @orderId, 388, 1)
);

```

This puts the data into the currently only existing federation member. We selected that federation member by picking 0 as distribution value, which is by definition part of the only existing federation.

## 14.7 Split Federation

Now lets split the federation, creating a second federation member. SQL Azure will automatically redistribute the data into the two federations after you executed this command.

```

<?php
// split_federation.php
require_once 'bootstrap.php';

$shardManager->splitFederation(60);

```

This little script uses the shard manager with a special method only existing on the SQL Azure implementation splitFederation. It accepts a value at which the split is executed.

If you reexecute the view\_federation\_members.php script you can now see that there are two federation members instead of just one as before. You can see with the rangeLow and rangeHigh parameters what customers and related entries are now served by which federation.

## 14.8 Inserting Data after Split

Now after we splitted the data we now have to make sure to be connected to the right federation before inserting data. Lets add a new customer with ID 55 and have him create an order.

```

<?php
// insert_data_aftersplit.php
require_once 'bootstrap.php';

$newCustomerId = 55;

$shardManager->selectShard($newCustomerId);

$conn->insert("Customers", array(
    "CustomerId" => $newCustomerId,
    "CompanyName" => "Microsoft",
    "FirstName" => "Brian",

```



```

        "LastName" => "Swan",
    ));

$conn->insert("Orders", array(
    "CustomerID" => 55,
    "OrderID" => 37,
    "OrderDate" => date('Y-m-d H:i:s'),
));

$conn->insert("OrderItems", array(
    "CustomerID" => 55,
    "OrderID" => 37,
    "ProductID" => 387,
    "Quantity" => 1,
));

```

As you can see its very important to pick the right distribution key in your sharded application. Otherwise you have to switch the shards very often, which is not really easy to work with. If you pick the sharding key right then it should be possible to select the shard only once per request for the major number of use-cases.

Fan-out the queries accross multiple shards should only be necessary for a small number of queries, because these kind of queries are complex.

## 14.9 Querying data with filtering off

To access the data you have to pick a shard again and then start selecting data from it.

```

<?php
// query_filtering_off.php
require_once "bootstrap.php";

$shardManager->selectShard(0);

$data = $conn->fetchAll('SELECT * FROM Customers');
print_r($data);

```

This returns all customers from the shard with distribution value 0. This will be all customers with id 10 to less than 60, since we split federations at 60.

## 14.10 Querying data with filtering on

One special feature of SQL Azure is the possibility to database level filtering based on the sharding distribution values. This means that SQL Azure will add WHERE clauses with `distributionkey=current distribution value` conditions to each distribution key.

```

<?php
// query_filtering_on.php
require_once "bootstrap.php";

$shardManager->setFilteringEnabled(true);
$shardManager->selectShard(55);

$data = $conn->fetchAll('SELECT * FROM Customers');
print_r($data);

```

Now you only get the customer with `id = 55`. The same holds for queries on the `Orders` and `OrderItems` table, which are restricted by customer `id = 55`.

---

## Supporting Other Databases

---

To support a database which is not currently shipped with Doctrine you have to implement the following interfaces and abstract classes:

- `\Doctrine\DBAL\Driver\Driver`
- `\Doctrine\DBAL\Driver\Statement`
- `\Doctrine\DBAL\Platforms\AbstractPlatform`
- `\Doctrine\DBAL\Schema\AbstractSchemaManager`

For an already supported platform but unsupported driver you only need to implement the first two interfaces, since the SQL Generation and Schema Management is already supported by the respective platform and schema instances. You can also make use of several Abstract Unittests in the `\Doctrine\Tests\DBAL` package to check if your platform behaves like all the others which is necessary for SchemaTool support, namely:

- `\Doctrine\Tests\DBAL\Platforms\AbstractPlatformTestCase`
- `\Doctrine\Tests\DBAL\Functional\Schema\AbstractSchemaManagerTestCase`

We would be very happy if any support for new databases would be contributed back to Doctrine to make it an even better product.

### 15.1 Implementation Steps in Detail

1. Add your driver shortcut to class-name *DoctrineDBALDriverManager*.
2. Make a copy of `tests/dbproperties.xml.dev` and adjust the values to your driver shortcut and testdatabase.
3. Create three new classes implementing `\Doctrine\DBAL\Driver\Driver`, `\Doctrine\DBAL\Driver\Statement` and `\Doctrine\DBAL\Driver`. You can take a look at the `\Doctrine\DBAL\Driver\OCI8` driver.
4. You can run the testsuite of your new database driver by calling “`cd tests/ && phpunit -c myconfig.xml Doctrine/Tess/AllTests.php`”
5. Start implementing `AbstractPlatform` and `AbstractSchemaManager`. Other implementations should serve as good example.



---

## Portability

---

There are often cases when you need to write an application or library that is portable across multiple different database vendors. The Doctrine ORM is one example of such a library. It is an abstraction layer over all the currently supported vendors (MySQL, Oracle, PostgreSQL, SQLite, SAP SQL Anywhere and Microsoft SQL Server). If you want to use the DBAL to write a portable application or library you have to follow lots of rules to make all the different vendors work the same.

There are many different layers that you need to take care of, here is a quick list:

- Returning of data is handled differently across vendors. Oracle converts empty strings to NULL, which means a portable application needs to convert all empty strings to null.
- Additionally some vendors pad CHAR columns to their length, whereas others don't. This means all strings returned from a database have to be passed through `rtrim()`.
- Case-sensitivity of column keys is handled differently in all databases, even depending on identifier quoting or not. You either need to know all the rules or fix the cases to lower/upper-case only.
- ANSI-SQL is not implemented fully by the different vendors. You have to make sure that the SQL you write is supported by all the vendors you are targeting.
- Some vendors use sequences for identity generation, some auto-increment approaches. Both are completely different (pre- and post-insert access) and therefore need special handling.
- Every vendor has a list of keywords that are not allowed inside SQL. Some even allow a subset of their keywords, but not at every position.
- Database types like dates, long text fields, booleans and many others are handled very differently between the vendors.
- There are differences with the regard to support of positional, named or both styles of parameters in prepared statements between all vendors.

For each point in this list there are different abstraction layers in Doctrine DBAL that you can use to write a portable application.

### 16.1 Connection Wrapper

This functionality is only implemented with Doctrine 2.1 upwards.

To handle all the points 1-3 you have to use a special wrapper around the database connection. The handling and differences to tackle are all taken from the great [PEAR MDB2 library](#).

Using the following code block in your initialization will:

- `rtrim()` all strings if necessary
- Convert all empty strings to null
- Return all associative keys in lower-case, using PDO native functionality or implemented in PHP userland (OCI8).

```
<?php
$params = array(
    // vendor specific configuration
    //...
    'wrapperClass' => 'Doctrine\DBAL\Portability\Connection',
    'portability' => \Doctrine\DBAL\Portability\Connection::PORTABILITY_ALL,
    'fetch_case' => \PDO::CASE_LOWER,
);
```

This sort of portability handling is pretty expensive because all the result rows and columns have to be looped inside PHP before being returned to you. This is why by default Doctrine ORM does not use this compability wrapper but implements another approach to handle assoc-key casing and ignores the other two issues.

## 16.2 Database Platform

Using the database platform you can generate bits of SQL for you, specifically in the area of SQL functions to achieve portability. You should have a look at all the different methods that the platforms allow you to access.

## 16.3 Keyword Lists

This functionality is only implemented with Doctrine 2.1 upwards.

Doctrine ships with lists of keywords for every supported vendor. You can access a keyword list through the schema manager of the vendor you are currently using or just instantiating it from the `Doctrine\DBAL\Platforms\Keywords` namespace.

---

## Caching

---

A Doctrine\DBAL\Statement can automatically cache result sets.

For this to work an instance of Doctrine\Common\Cache\Cache must be provided. This can be set on the configuration object (optionally it can also be passed at query time):

```
<?php
$cache = new \Doctrine\Common\Cache\ArrayCache();
$config = $conn->getConfiguration();
$config->setResultCacheImpl($cache);
```

To get the result set of a query cached it is necessary to pass a Doctrine\DBAL\Cache\QueryCacheProfile instance to the executeQuery or executeCacheQuery instance. The difference between these two methods is that the former does not require this instance, while the later has this instance as a required parameter:

```
<?php
$stmt = $conn->executeQuery($query, $params, $types, new QueryCacheProfile(0, "some key"));
$stmt = $conn->executeCacheQuery($query, $params, $types, new QueryCacheProfile(0, "some key"));
```

It is also possible to pass in a the Doctrine\Common\Cache\Cache instance into the constructor of Doctrine\DBAL\Cache\QueryCacheProfile in which case it overrides the default cache instance:

```
<?php
$cache = new \Doctrine\Common\Cache\FilesystemCache(__DIR__);
new QueryCacheProfile(0, "some key", $cache);
```

In order for the data to actually be cached its necessary to ensure that the entire result set is read (easiest way to ensure this is to use fetchAll) and the statement object is closed:

```
<?php
$stmt = $conn->executeCacheQuery($query, $params, $types, new QueryCacheProfile(0, "some key"));
$data = $stmt->fetchAll();
$stmt->closeCursor(); // at this point the result is cached
```

**Warning:** When using the cache layer not all fetch modes are supported. See the code of the [ResultCacheStatement](#) for details.





---

## Known Vendor Issues

---

This section describes known compatability issues with all the supported database vendors:

### 18.1 PostgreSQL

#### 18.1.1 DateTime, DateTimeTz and Time Types

Postgres has a variable return format for the datatype `TIMESTAMP(n)` and `TIME(n)` if microseconds are allowed ( $n > 0$ ). Whenever you save a value with `microseconds = 0`, PostgreSQL will return this value in the format:

```
2010-10-10 10:10:10 (Y-m-d H:i:s)
```

However if you save a value with microseconds it will return the full representation:

```
2010-10-10 10:10:10.123456 (Y-m-d H:i:s.u)
```

Using the `DateTime`, `DateTimeTz` or `Time` type with microseconds enabled columns can lead to errors because internally types expect the exact format ‘Y-m-d H:i:s’ in combination with `DateTime::createFromFormat()`. This method is twice as fast as passing the date to the constructor of `DateTime`.

This is why Doctrine always wants to create the time related types without microseconds:

- `DateTime` to `TIMESTAMP(0) WITHOUT TIME ZONE`
- `DateTimeTz` to `TIMESTAMP(0) WITH TIME ZONE`
- `Time` to `TIME(0) WITHOUT TIME ZONE`

If you do not let Doctrine create the date column types and rather use types with microseconds you have to replace the “`DateTime`”, “`DateTimeTz`” and “`Time`” types with a more liberal `DateTime` parser that detects the format automatically:

```
use Doctrine\DBAL\Types\Type;

Type::overrideType('datetime', 'Doctrine\DBAL\Types\VarDateTimeType');
Type::overrideType('datetimetz', 'Doctrine\DBAL\Types\VarDateTimeType');
Type::overrideType('time', 'Doctrine\DBAL\Types\VarDateTimeType');
```

#### 18.1.2 Timezones and DateTimeTz

Postgres does not save the actual Timezone Name but UTC-Offsets. The difference is subtle but can be potentially very nasty. Derick Rethans explains it very well [in a blog post of his](#).

## 18.2 MySQL

### 18.2.1 DateTimeTz

MySQL does not support saving timezones or offsets. The DateTimeTz type therefore behave like the DateTime type.

## 18.3 Sqlite

### 18.3.1 DateTime

Unlike most database management systems, Sqlite does not convert supplied datetime strings to an internal storage format before storage. Instead, Sqlite stores them as verbatim strings (i.e. as they are entered) and expects the user to use the `DATETIME()` function when reading data which then converts the stored values to datetime strings. Because Doctrine is not using the `DATETIME()` function, you may end up with “Could not convert database value ... to Doctrine Type datetime.” exceptions when trying to convert database values to `\DateTime` objects using

```
\Doctrine\DBAL\Types\Type::getType('datetime')->convertToPhpValue(...)
```

### 18.3.2 DateTimeTz

Sqlite does not support saving timezones or offsets. The DateTimeTz type therefore behave like the DateTime type.

### 18.3.3 Reverse engineering primary key order

SQLite versions < 3.7.16 only return that a column is part of the primary key, but not the order. This is only a problem with tables where the order of the columns in the table is not the same as the order in the primary key. Tables created with Doctrine use the order of the columns as defined in the primary key.

## 18.4 IBM DB2

### 18.4.1 DateTimeTz

DB2 does not save the actual Timezone Name but UTC-Offsets. The difference is subtle but can be potentially very nasty. Derick Rethans explains it very well [in a blog post of his](#).

## 18.5 Oracle

### 18.5.1 DateTimeTz

Oracle does not save the actual Timezone Name but UTC-Offsets. The difference is subtle but can be potentially very nasty. Derick Rethans explains it very well [in a blog post of his](#).

### 18.5.2 OCI8: SQL Queries with Question Marks

We had to implement a question mark to named parameter translation inside the OCI8 DBAL Driver. It works as a very simple parser with two states: Inside Literal, Outside Literal. From our perspective it should be working in all cases, but you have to be careful with certain queries:

```
SELECT * FROM users WHERE name = 'bar?'
```

Could in case of a bug with the parser be rewritten into:

```
SELECT * FROM users WHERE name = 'bar:ocil'
```

For this reason you should always use prepared statements with Oracle OCI8, never use string literals inside the queries. A query for the user ‘bar?’ should look like:

```
$sql = 'SELECT * FROM users WHERE name = ?'
$stmt = $conn->prepare($sql);
$stmt->bindValue(1, 'bar?');
$stmt->execute();
```

### 18.5.3 OCI-LOB instances

Doctrine 2 always requests CLOB columns as strings, so that you as a developer never get access to the OCI-LOB instance. Since we are using prepared statements for all write operations inside the ORM, using strings instead of the OCI-LOB does not cause any problems.

## 18.6 Microsoft SQL Server

### 18.6.1 Unique and NULL

Microsoft SQL Server takes Unique very seriously. There is only ever one NULL allowed contrary to the standard where you can have multiple NULLs in a unique column.

### 18.6.2 DateTime, DateTimeTz and Time Types

SQL Server has a variable return format for the datatype DATETIME(n) if microseconds are allowed (n > 0). Whenever you save a value with microseconds = 0.

If you do not let Doctrine create the date column types and rather use types with microseconds you have to replace the “DateTime”, “DateTimeTz” and “Time” types with a more liberal DateTime parser that detects the format automatically:

```
use Doctrine\DBAL\Types\Type;

Type::overrideType('datetime', 'Doctrine\DBAL\Types\VarDateTime');
Type::overrideType('datetimez', 'Doctrine\DBAL\Types\VarDateTime');
Type::overrideType('time', 'Doctrine\DBAL\Types\VarDateTime');
```

### 18.6.3 PDO\_SQLSRV: VARBINARY/BLOB columns

The PDO\_SQLSRV driver currently has a bug when binding values to VARBINARY/BLOB columns with bindValue in prepared statements. This raises an implicit conversion from data type error as it tries to convert

a character type value to a binary type value even if you explicitly define the value as `\PDO::PARAM_LOB` type. Therefore it is highly encouraged to use the native `sqlsrv` driver instead which does not have this limitation.

---

## Indices and tables

---

- [search](#)