Database Abstraction Layer

Powerful PHP database abstraction layer (DBAL) with many features for database schema introspection and management.

```
composer require doctrine/dbal
```

https://github.com/doctrine/dbal/

DBAL Documentation

The Doctrine DBAL documentation is a reference guide to everything you need to know about the database abstraction layer.

Getting Help

If this documentation is not helping to answer questions you have about the Doctrine DBAL, don't panic. You can get help from different sources:

- Slack chat room #dbal
- On Stack Overflow
- The Doctrine Mailing List
- Report a bug on GitHub.

Getting Started

The best way to get started is with the Introduction section in the documentation. Use the sidebar to browse other documentation for the Doctrine PHP DBAL.

https://groups.google.com/g/doctrine-user?pli=1

DBAL Documentation

The Doctrine DBAL documentation is a reference guide to everything you need to know about the database abstraction layer.

Getting Help

If this documentation is not helping to answer questions you have about the Doctrine DBAL, don't panic. You can get help from different sources:

- Slack chat room #dbal
- On Stack Overflow
- The Doctrine Mailing List
- Report a bug on GitHub.

Getting Started

The best way to get started is with the Introduction section in the documentation. Use the sidebar to browse other documentation for the Doctrine PHP DBAL.

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
- SQLite

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 class loader provided by Composer, to be able to autoload the classes:

<?php require_once 'vendor/autoload.php';
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

The DBAL consists of two layers: drivers and a wrapper. Each layer is mainly defined in terms of 3 components: `Connection`, `Statement` and `Result`. A `Doctrine\DBAL\Connection` wraps a `Doctrine\DBAL\Driver\Connection`, a `Doctrine\DBAL\Statement` wraps a `Doctrine\DBAL\Driver\Statement` and a `Doctrine\DBAL\Result` wraps a `Doctrine\DBAL\Driver\Result`.

`Doctrine\DBAL\Driver\Connection`, `Doctrine\DBAL\Driver\Statement` and `Doctrine\DBAL\Driver\Result` are just interfaces. These interfaces are implemented by concrete drivers.

What do the wrapper components 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.

Apart from the three main components, a DBAL driver should also provide an implementation of the `Doctrine\DBAL\Driver` interface that has two primary purposes:

1. Translate the DBAL connection parameters to the ones specific to the driver's connection class.
2. Act as a factory of other driver-specific components like platform, schema manager and exception converter.

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

Drivers

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

- `\Doctrine\DBAL\Driver\Connection`
- `\Doctrine\DBAL\Driver\Statement`
- `\Doctrine\DBAL\Driver\Result`

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 multiple MySQL extensions: pdo_mysql and mysqli.

Logging

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

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.

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

Getting a Connection

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

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

Or, using the simpler URL form:

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

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.

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=UTF8
```

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

```
pdo-mysql://localhost:4486/foo?charset=UTF8
```

In the example above, mind the dashes instead of the underscores in the URL scheme.

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:
```

> 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.

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.
  - `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
  - `sqlsrv`: A Microsoft SQL Server driver that uses the sqlsrv PHP extension.
  - `oci8`: An Oracle driver that uses the oci8 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.

Wrapper Class

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

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.

> 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.

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.
- `ssl_key` (string): The path name to the key file to use for SSL encryption.

- `ssl_cert` (string): The path name to the certificate file to use for SSL encryption.
- `ssl_ca` (string): The path name to the certificate authority file to use for SSL encryption.
- `ssl_capath` (string): The pathname to a directory that contains trusted SSL CA certificates in PEM format.
- `ssl_cipher` (string): A list of allowable ciphers to use for SSL encryption.
- `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.
- `default_dbname` (string): Override the default database (postgres) to connect to.
- `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: `https://www.postgresql.org/docs/9.4/static/libpq-connect.html#LIBPQ-CONNECT-SSLMODE`
- `sslrootcert` (string): specifies the name of a file containing SSL certificate authority (CA) certificate(s). If the file exists, the server's certificate will be verified to be signed by one of these authorities. See https://www.postgresql.org/docs/9.4/static/libpq-connect.html#LIBPQ-CONNECT-SSLROOTCERT
- `sslcert` (string): specifies the file name of the client SSL certificate. See `https://www.postgresql.org/docs/9.4/static/libpq-connect.html#LIBPQ-CONNECT-SSLCERT`
- `sslkey` (string): specifies the location for the secret key used for the client certificate. See `https://www.postgresql.org/docs/9.4/static/libpq-connect.html#LIBPQ-CONNECT-SSLKEY`
- `sslcrl` (string): specifies the file name of the SSL certificate revocation list (CRL). See `https://www.postgresql.org/docs/9.4/static/libpq-connect.html#LIBPQ-CONNECT-SSLCRL`
- `application_name` (string): Name of the application that is connecting to database. Optional. It will be displayed at `pg_stat_activity`.

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.
- `connectstring` (string): Complete Easy Connect connection descriptor, see https://docs.oracle.com/database/121/NETAG/naming.htm. When using this option, you will still need to provide the `user` and `password` parameters, but the other parameters will no longer be used. Note that when using this parameter, the `getHost` and `getPort` methods from `Doctrine\DBAL\Connection` will no longer function as expected.
- `persistent` (boolean): Whether to establish a persistent connection.

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.

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.

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.

If you are running a MariaDB database, you should prefix the `serverVersion` with `mariadb-` (ex: `mariadb-10.2.12`).

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.

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.

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->fetchAssociative()) !== false) {

    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**.

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

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 colon (e.g. `:name1`). You cannot mix the positional and the named approach. You have to bind a parameter to each placeholder.

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.

> 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.

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, $types)` - Create a prepared statement for the passed SQL query, bind the given params with their binding types and execute the query. This method returns the executed prepared statement for iteration and is useful for SELECT statements.
- `executeStatement($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#executeStatement()` 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.

Binding Types

Doctrine DBAL extends PDOs handling of binding types in prepared statements considerably. Besides `Doctrine\DBAL\ParameterType` constants, you can make use of two very powerful additional features.

Doctrine\DBAL\Types Conversion

If you don't specify an integer (through one of `Doctrine\DBAL\ParameterType` constants) 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 are delegated to a method on the current database platform, which means this code works independent of the database you are using.

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

List of Parameters Conversion

> 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 these 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(
        ParameterType::INTEGER,
        ParameterType::INTEGER,
        ParameterType::INTEGER,
        ParameterType::INTEGER,
```

```
        ParameterType::INTEGER,
        ParameterType::INTEGER,
    )
);
```

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

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

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.

prepare()

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

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

/*
array(
  0 => array(
    'username' => 'jwage',
    'email' => 'j.wage@example.com'
  )
)
*/
```

executeStatement()

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

```
<?php

$count = $conn->executeStatement('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.

executeQuery()

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

```
<?php

$statement = $conn->executeQuery('SELECT * FROM user WHERE username = ?', array('jwage'));

$user = $statement->fetchAssociative();


/*

array(

  0 => 'jwage',

  1 => 'j.wage@example.com'

)

*/
```

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.

fetchAllAssociative()

Execute the query and fetch all results into an array:

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

/*
array(
  0 => array(
    'username' => 'jwage',
    'email' => 'j.wage@example.com'
  )
)
*/
```

fetchAllKeyValue()

Execute the query and fetch the first two columns into an associative array as keys and values respectively:

```
<?php
$users = $conn->fetchAllKeyValue('SELECT username, email FROM user');

/*
array(
  'jwage' => 'j.wage@example.com',
)
*/
```

> All additional columns will be ignored and are only allowed to be selected by DBAL for its internal purposes.

fetchAllAssociativeIndexed()

Execute the query and fetch the data as an associative array where the key represents the first column and the value is an associative array of the rest of the columns and their values:

```
<?php
$users = $conn->fetchAllAssociativeIndexed('SELECT id, username, email FROM user');

/*
array(
    1 => array(
      'username' => 'jwage',
      'email' => 'j.wage@example.com'
    )
)
*/
```

fetchNumeric()

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

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

/*
array(
  0 => 'jwage',
  1 => 'j.wage@example.com'
)
*/
```

fetchOne()

Retrieve only the value of the first column of the first result row.

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

fetchAssociative()

Retrieve associative array of the first result row.

```
<?php

$user = $conn->fetchAssociative('SELECT * FROM user WHERE username = ?', array('jwage'));

/*

array(

  'username' => 'jwage',

  'email' => 'j.wage@example.com'

)

*/
```

There are also convenience methods for data manipulation queries:

iterateKeyValue()

Execute the query and iterate over the first two columns as keys and values respectively:

```
<?php

foreach ($conn->iterateKeyValue('SELECT username, email FROM user') as $username => $email) {

    // ...

}
```

> All additional columns will be ignored and are only allowed to be selected by DBAL for its internal purposes.

iterateAssociativeIndexed()

Execute the query and iterate over the result with the key representing the first column and the value being an associative array of the rest of the columns and their values:

```
<?php

foreach ($conn->iterateAssociativeIndexed('SELECT id, username, email FROM user') as $id => $data) {

    // ...

}
```

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)

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)

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:

quote()

Quote a value:

<?php

use Doctrine\DBAL\ParameterType;

$quoted = $conn->quote('value');

$quoted = $conn->quote('1234', ParameterType::INTEGER);

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();
```

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)
;
```

> The numerical parameters in the QueryBuilder API start with the needle `0`, not with `1` as in the PDO API.

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.

DISTINCT-Clause

The `SELECT` statement can be specified with a `DISTINCT` clause:

<?php
$queryBuilder ->select('name') ->distinct() ->from('users') ;

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.

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 = ?') ;

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') ;

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()`.

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.

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);

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 ()

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) ;

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()->and($queryBuilder->expr()->eq('username', '?'), $queryBuilder->expr()->eq('email', '?')));
The `and()` and `or()` 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.

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`.

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.

> 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.

> 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.

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
```

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.

Error handling

In order to handle errors related to deadlocks or lock wait timeouts, you can use Doctrine built-in transaction exceptions. All transaction exceptions where retrying makes sense have a marker interface: `Doctrine\DBAL\Exception\RetryableException`. A practical example is as follows:

```
<?php

try {
    // process stuff
} catch (\Doctrine\DBAL\Exception\RetryableException $e) {
    // retry the processing
}
```

If you need stricter control, you can catch the concrete exceptions directly:

- `Doctrine\DBAL\Exception\DeadlockException`: this can happen when each member of a group of actions is waiting for some other member to release a shared lock.
- `Doctrine\DBAL\Exception\LockWaitTimeoutException`: this exception happens when a transaction has to wait a considerable amount of time to obtain a lock, even if a deadlock is not involved.

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:

MySQL

- `MySQLPlatform` for version 5.0 and above.
- `MySQL57Platform` for version 5.7 (5.7.9 GA) and above.
- `MySQL80Platform` for version 8.0 (8.0 GA) and above.

MariaDB

- `MariaDb1027Platform` for version 10.2 (10.2.7 GA) and above.

Oracle

- `OraclePlatform` for all versions.

Microsoft SQL Server

- `SQLServer2012Platform` for version 2012 and above.

PostgreSQL

- `PostgreSQL94Platform` for version 9.4 and above.
- `PostgreSQL100Platform` for version 10.0 and above.

SQLite

- `SqlitePlatform` 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.

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');

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.

> 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.

> 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.

> 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.

> 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.

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.

> 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.

ascii_string

Similar to the `string` type but for binding non-unicode data. This type should be used with database vendors where a binding type mismatch can trigger an implicit cast and lead to performance problems.

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.

> 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.

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.

> 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.

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.

date_immutable

The immutable variant of the `date` type. Values retrieved from the database are always converted to PHP's `\DateTimeImmutable` object or `null` if no data is present.

> Available since version `2.6`.

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.

> 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.

datetime_immutable

The immutable variant of the `datetime` type. Values retrieved from the database are always converted to PHP's `\DateTimeImmutable` object or `null` if no data is present.

> Available since version `2.6`.

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.

datetimetz_immutable

The immutable variant of the `datetimetz` type. Values retrieved from the database are always converted to PHP's `\DateTimeImmutable` object or `null` if no data is present.

> Available since version `2.6`.

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.

time_immutable

The immutable variant of the `time` type. Values retrieved from the database are always converted to PHP's `\DateTimeImmutable` object or `null` if no data is present.

> Available since version `2.6`.

dateinterval

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

> See the Known Vendor Issue Known Vendor Issues section for details about the different handling of microseconds and timezones across all the different vendors.
>
> 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.

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.

> 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 an 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.

> 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 an 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.

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

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` or `null` types using PHP's `json_decode()` function.

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 an 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.

You should never rely on the order of your JSON object keys, as some vendors like MySQL sort the keys of its native JSON type using an internal order which is also subject to change.

json_array

This type is deprecated since 2.6, you should use `json` instead.

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.

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 an 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.

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.

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 an 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.

While the built-in `text` type of MySQL and MariaDB can store binary data, `mysqldump` cannot properly export `text` fields containing binary data. This will cause creating and restoring of backups fail silently. A workaround is to `serialize()`/`unserialize()` and `base64_encode()`/`base64_decode()` PHP objects and store them into a `text` field manually.

Because the built-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.

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 Type				
smallint	`integer`	**MySQL** **PostgreSQL** **Oracle** **SQL Server** **SQLite**	*all* *all* *all* *all* *all*	`SMALLINT UNSIGNED` [10]_ `AUTO_`
integer	`integer`	**MySQL** **PostgreSQL** **Oracle** **SQL Server** **SQLite**	*all* *all* *all* *all* *all*	`INT UNSIGNED` [10]_ `AUTO_INCRE` `INT` [12]_
bigint	`string` [8]_	**MySQL** **PostgreSQL** **Oracle** **SQL Server** **SQLite**	*all* *all* *all* *all* *all*	`BIGINT UNSIGNED` [10]_ `AUTO_IN`

decimal [7]_	`string` [9]_	**MySQL** --- **PostgreSQL** --- **Oracle** --- **SQL Server** --- **SQLite**	*all* --- *all*	`NUMERIC(p, s)` `UNSIGNED` [10]_ `NUMERIC(p, s)`
float	`float`	**MySQL** --- **PostgreSQL** --- **Oracle** --- **SQL Server** --- **SQLite**	*all* --- *all*	`DOUBLE PRECISION` `UNSIGNED` [1 `DOUBLE PRECISION`
string [2]_ [5]_	`string`	**MySQL** --- **PostgreSQL** --- **SQLite** --- **Oracle** --- **SQL Server**	*all* --- *all* --- *all*	`VARCHAR(n)` [3]_ --- --- ---
ascii_string	`string`	**SQL Server** **MySQL** --- **PostgreSQL** --- **Oracle** --- **SQLite** --- **SQL Server**		`VARCHAR(n)` `CHAR(n)`
text	`string`	**MySQL** --- **PostgreSQL** --- **Oracle** --- **SQLite** --- **SQL Server**	*all* --- *all all* --- *all*	`TINYTEXT` [16]_ --- --- ---
guid	`string`	**MySQL** --- **Oracle** --- **SQLite** --- **SQL Server** --- **PostgreSQL**	*all* --- *all all*	`VARCHAR(255)` [1]_ --- `UUID`
binary [2]_ [6]_	`resource`	**MySQL** --- **SQL Server** --- **Oracle**	*all all* --- *all* --- *all*	`VARBINARY(n)` [3]_ --- ---

		---		
		PostgreSQL		

		SQLite		
		MySQL		
		---		`TINYBLOB` [16]_ ---
		Oracle	*all* ---	---
		---	*all* ---	---
blob	`resource`	**SQLite**	*all* ---	---
		---	*all*	---
		SQL Server		---

		PostgreSQL		
		MySQL		

		PostgreSQL	*all* ---	
		---	*all* ---	`TINYINT(1)` ---
boolean	`boolean`	**SQLite**	*all all*	---

		SQL Server		

		Oracle		
		MySQL		

		PostgreSQL		
		---	*all* ---	
date	`\DateTime`	**Oracle**	"all"	`DATE`

		SQLite		

		SQL Server		
		MySQL		

		SQL Server	*all* ---	
		---	*all* ---	`DATETIME` [13]_ ---
datetime	`\DateTime`	**SQLite**	*all* ---	---
		---	*all*	---
		PostgreSQL		

		Oracle		
		MySQL		

		SQLite	*all* ---	
		---	"all" ---	`DATETIME` [14]_ [15]_ ---
datetimetz	`\DateTime`	**SQL Server**	*all*	`TIME ZONE`

		PostgreSQL		

		Oracle		
		MySQL	*all* ---	`TIME` ---
time	`\DateTime`	---	*all* ---	---

		SQLite --- **PostgreSQL** --- **Oracle** --- **SQL Server** **MySQL** --- **PostgreSQL** --- **Oracle** --- **SQLite** --- **SQL Server** **MySQL** [1]_ --- **PostgreSQL** --- **Oracle** --- **SQLite** --- **SQL Server** **MySQL** --- **PostgreSQL** --- **Oracle** --- **SQLite** --- **SQL Server**	*all* --- "all"	
array [1]_ --- **simple array** [1]_	`array`		*all* --- *all all* --- *all*	`TINYTEXT` [16]_ --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
json_array	`array`		*all* --- *all* --- *all* --- *all*	`TINYTEXT` [16]_ --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
object [1]_	`object`		*all* --- *all all* --- *all*	`TINYTEXT` [16]_ --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---

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.

> 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.

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 an SQL declaration. You can modify the SQL declaration Doctrine will produce. At first, to enable this feature, you must override the `canRequireSQLConversion` method:

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

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

```
<?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 using 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:

> 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.

listDatabases()

Retrieve an array of databases on the configured connection:

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

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";
}
```

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";
}
```

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');
```

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";

}

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

<?php

foreach ($foreignKeys as $foreignKey) {

    echo $foreignKey->getName() . ': ' . $foreignKey->getLocalTableName() ."\n";

}

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";
}
```

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";
    }
}
```

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";
}
```

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

$schema = 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"));

$myTable->setComment('Some comment');
```

```
$schema->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();
$schemaDiff = $comparator->compare($fromSchema, $toSchema);


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

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.

> This chapter is far from being completely documented.

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 only by MySQL. Defaults to `false`.
- **comment** (integer|string): The column comment. Supported by MySQL, PostgreSQL, Oracle and SQL Server. Defaults to `null`.

Vendor specific options

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

- **columnDefinition** (string): The custom column declaration SQL snippet to use instead of the generated SQL by Doctrine. Defaults to `null`. This can 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.
- **collation** (string): The collation to use for the column. Supported by MySQL, PostgreSQL, Sqlite and 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.

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 enviroment 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);
```

Schema Events

There are multiple events in Doctrine DBAL that are triggered on schema changes of the database. It is possible to add your own event listener to be able to run your own code before changes to the database are committed. An instance of `Doctrine\Common\EventManager` can also be added to Platforms.

A event listener class can contain one or more methods to schema events. These methods must be named like the events itself.

```
<?php

$evm = new EventManager();

$eventName = Events::onSchemaCreateTable;

$evm->addEventListener($eventName, new MyEventListener());
```

```
<?php

$evm = new EventManager();

$eventNames = array(Events::onSchemaCreateTable, Events::onSchemaCreateTableColumn);

$evm->addEventListener($eventNames, new MyEventListener());
```

The following events are available.

OnSchemaCreateTable Event

`Doctrine\DBAL\Events::onSchemaCreateTable` is triggered before every create statement that is executed by one of the Platform instances and injects an instance of `Doctrine\DBAL\Event\SchemaCreateTableEventArgs` as event argument for event listeners.

```
<?php

class MyEventListener

{

   public function onSchemaCreateTable(SchemaCreateTableEventArgs $event)

   {

      // Your EventListener code

   }

}
```

```
$evm = new EventManager();

$evm->addEventListener(Events::onSchemaCreateTable, new MyEventListener());
```

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

It allows you to access the `Doctrine\DBAL\Schema\Table` instance and its columns, the used Platform and provides a way to add additional SQL statements.

OnSchemaCreateTableColumn Event

`Doctrine\DBAL\Events::onSchemaCreateTableColumn` is triggered on every new column before a create statement that is executed by one of the Platform instances and injects an

instance of `Doctrine\DBAL\Event\SchemaCreateTableColumnEventArgs` as event argument for event listeners.

```
<?php

class MyEventListener

{

   public function onSchemaCreateTableColumn(SchemaCreateTableColumnEventArgs $event)

   {

      // Your EventListener code

   }

}


$evm = new EventManager();

$evm->addEventListener(Events::onSchemaCreateTableColumn, new MyEventListener());


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

It allows you to access the `Doctrine\DBAL\Schema\Table` instance, the affected `Doctrine\DBAL\Schema\Column`, the used Platform and provides a way to add additional SQL statements.

OnSchemaDropTable Event

`Doctrine\DBAL\Events::onSchemaDropTable` is triggered before a drop table statement that is executed by one of the Platform instances and injects an instance of `Doctrine\DBAL\Event\SchemaDropTableEventArgs` as event argument for event listeners.

```
<?php

class MyEventListener

{

   public function onSchemaDropTable(SchemaDropTableEventArgs $event)

   {

      // Your EventListener code

   }

}


$evm = new EventManager();

$evm->addEventListener(Events::onSchemaDropTable, new MyEventListener());
```

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

It allows you to access the `Doctrine\DBAL\Schema\Table` instance, the used Platform and provides a way to set an additional SQL statement.

OnSchemaAlterTable Event

`Doctrine\DBAL\Events::onSchemaAlterTable` is triggered before every alter statement that is executed by one of the Platform instances and injects an instance of `Doctrine\DBAL\Event\SchemaAlterTableEventArgs` as event argument for event listeners.

```
<?php
class MyEventListener
{
    public function onSchemaAlterTable(SchemaAlterTableEventArgs $event)
    {
        // Your EventListener code
    }
}

$evm = new EventManager();
$evm->addEventListener(Events::onSchemaAlterTable, new MyEventListener());

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

It allows you to access the `Doctrine\DBAL\Schema\TableDiff` instance, the used Platform and provides a way to add additional SQL statements.

OnSchemaAlterTableAddColumn Event

`Doctrine\DBAL\Events::onSchemaAlterTableAddColumn` is triggered on every altered column before every alter statement that is executed by one of the Platform instances and injects an instance of `Doctrine\DBAL\Event\SchemaAlterTableAddColumnEventArgs` as event argument for event listeners.

```
<?php
class MyEventListener
{
    public function
onSchemaAlterTableAddColumn(SchemaAlterTableAddColumnEventArgs $event)
    {
        // Your EventListener code
    }
}

$evm = new EventManager();
$evm->addEventListener(Events::onSchemaAlterTableAddColumn, new
MyEventListener());

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

It allows you to access the `Doctrine\DBAL\Schema\TableDiff` instance, the affected `Doctrine\DBAL\Schema\Column`, the used Platform and provides a way to add additional SQL statements.

OnSchemaAlterTableRemoveColumn Event

`Doctrine\DBAL\Events::onSchemaAlterTableRemoveColumn` is triggered on every column that is going to be removed before every alter-drop statement that is executed by one of the Platform instances and injects an instance of `Doctrine\DBAL\Event\SchemaAlterTableRemoveColumnEventArgs` as event argument for event listeners.

```
<?php
class MyEventListener
{
    public function
onSchemaAlterTableRemoveColumn(SchemaAlterTableRemoveColumnEventArgs $event)
    {
        // Your EventListener code
    }
}

$evm = new EventManager();
$evm->addEventListener(Events::onSchemaAlterTableRemoveColumn, new
MyEventListener());

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

It allows you to access the `Doctrine\DBAL\Schema\TableDiff` instance, the affected `Doctrine\DBAL\Schema\Column`, the used Platform and provides a way to add additional SQL statements.

OnSchemaAlterTableChangeColumn Event

`Doctrine\DBAL\Events::onSchemaAlterTableChangeColumn` is triggered on every column that is going to be changed before every alter statement that is executed by one of the Platform instances and injects an instance of `Doctrine\DBAL\Event\SchemaAlterTableRemoveColumnEventArgs` as event argument for event listeners.

```
<?php
class MyEventListener
{
    public function
onSchemaAlterTableChangeColumn(SchemaAlterTableChangeColumnEventArgs $event)
    {
        // Your EventListener code
    }
}

$evm = new EventManager();
$evm->addEventListener(Events::onSchemaAlterTableChangeColumn, new
MyEventListener());

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

It allows you to access the `Doctrine\DBAL\Schema\TableDiff` instance, a `Doctrine\DBAL\Schema\ColumnDiff` of the affected column, the used Platform and provides a way to add additional SQL statements.

OnSchemaAlterTableRenameColumn Event

`Doctrine\DBAL\Events::onSchemaAlterTableRenameColumn` is triggered on every column that is going to be renamed before every alter statement that is executed by one of the Platform instances and injects an instance of `Doctrine\DBAL\Event\SchemaAlterTableRenameColumnEventArgs` as event argument for event listeners.

```
<?php
class MyEventListener
{
    public function
onSchemaAlterTableRenameColumn(SchemaAlterTableRenameColumnEventArgs $event)
    {
        // Your EventListener code
    }
}

$evm = new EventManager();
$evm->addEventListener(Events::onSchemaAlterTableRenameColumn, new
MyEventListener());

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

It allows you to access the `Doctrine\DBAL\Schema\TableDiff` instance, the old column name and the new column in form of a `Doctrine\DBAL\Schema\Column` object, the used Platform and provides a way to add additional SQL statements.

OnSchemaColumnDefinition Event

`Doctrine\DBAL\Events::onSchemaColumnDefinition` is triggered on a schema update and is executed for every existing column definition of the database before changes are applied. An instance of `Doctrine\DBAL\Event\SchemaColumnDefinitionEventArgs` is injected as argument for event listeners.

```
<?php
class MyEventListener
{
    public function onSchemaColumnDefinition(SchemaColumnDefinitionEventArgs
$event)
    {
        // Your EventListener code
    }
}

$evm = new EventManager();
$evm->addEventListener(Events::onSchemaColumnDefinition, new MyEventListener());

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

It allows you to access the table column definitions of the current database, table name, Platform and `Doctrine\DBAL\Connection` instance. Columns, that are about to be added, are not listed.

OnSchemaIndexDefinition Event

`Doctrine\DBAL\Events::onSchemaIndexDefinition` is triggered on a schema update and is executed for every existing index definition of the database before changes are applied. An

instance of `Doctrine\DBAL\Event\SchemaIndexDefinitionEventArgs` is injected as argument for event listeners.

```
<?php
class MyEventListener
{
    public function onSchemaIndexDefinition(SchemaIndexDefinitionEventArgs
$event)
    {
        // Your EventListener code
    }
}

$evm = new EventManager();
$evm->addEventListener(Events::onSchemaIndexDefinition, new MyEventListener());

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

It allows you to access the table index definitions of the current database, table name, Platform and `Doctrine\DBAL\Connection` instance. Indexes, that are about to be added, are not listed.

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.

SQL Injection: Safe and Unsafe APIs for User Input

A database library naturally 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/DDL SQL statements

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

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:

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 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.

Right: Prepared Statements

You should always use prepared statements to execute your queries. Prepared statements is a two-step procedure, separating the 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 placeholders 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
$dql = "SELECT u FROM User u WHERE u.username = ?1";
$query = $em->createQuery($dql);
$query->setParameter(1, $_GET['username']);
$data = $query->getResult();

// DQL Prepared Statements: Named
$dql = "SELECT u FROM User u WHERE u.username = :name";
$query = $em->createQuery($dql);
$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 `executeStatement` 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.

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']);
```

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

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

For an already supported platform but unsupported driver you only need to implement the first three 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 unit tests in the `\Doctrine\DBAL\Tests` package to check if your platform behaves like all the others which is necessary for SchemaTool support, namely:

- `\Doctrine\DBAL\Tests\Platforms\AbstractPlatformTestCase`
- `\Doctrine\DBAL\Tests\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.

Implementation Steps in Detail

1. Add your driver shortcut to the `Doctrine\DBAL\DriverManager` class.
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\Connection`, `\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 test suite of your new database driver by calling `phpunit`. You can set your own settings in the phpunit.xml file.

5. Start implementing AbstractPlatform and AbstractSchemaManager. Other implementations should serve as good examples.

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 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.

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

use Doctrine\DBAL\ColumnCase;
use Doctrine\DBAL\Portability\Connection as PortableConnection;

$params = array(
    // vendor specific configuration
    //...
    'wrapperClass' => PortableConnection::class,
    'portability'  => PortableConnection::PORTABILITY_ALL,
    'fetch_case'   => ColumnCase::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.

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.

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 `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 (the easiest way to ensure this is to use one of the `fetchAll*()` methods):

<?php

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

$data = $stmt→fetchAllAssociative();

When using the cache layer not all fetch modes are supported. See the code of the `Doctrine\DBAL\Cache\ResultCacheStatement` for details.

Known Vendor Issues

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

PostgreSQL

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 (and immutable variants) 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 replace the DateTime, DateTimeTz and Time types (and immutable variants) 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');

Type::overrideType('datetime_immutable', 'Doctrine\DBAL\Types\
VarDateTimeImmutableType');
Type::overrideType('datetimetz_immutable', 'Doctrine\DBAL\Types\
VarDateTimeImmutableType');
Type::overrideType('time_immutable', 'Doctrine\DBAL\Types\
VarDateTimeImmutableType');
```

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.

MySQL

DateTimeTz

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

Sqlite

Buffered Queries and Isolation

Be careful if you execute a `SELECT` query and do not iterate over the statements results immediately. `UPDATE` statements executed before iteration affect only the rows that have not been buffered into PHP memory yet. This breaks the SERIALIZABLE transaction isolation property that SQLite supposedly has.

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(...)
```

DateTimeTz

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

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.

IBM DB2

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.

Oracle

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.

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.

Microsoft SQL Server

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.

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 replace the DateTime, DateTimeTz and Time types (and immutable variants) 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('datetimetz', 'Doctrine\DBAL\Types\VarDateTime');
Type::overrideType('time', 'Doctrine\DBAL\Types\VarDateTime');

Type::overrideType('datetime_immutable', 'Doctrine\DBAL\Types\
VarDateTimeImmutableType');
Type::overrideType('datetimetz_immutable', 'Doctrine\DBAL\Types\
VarDateTimeImmutableType');
Type::overrideType('time_immutable', 'Doctrine\DBAL\Types\
VarDateTimeImmutableType');
```

Tutorial: Doctrine DBAL (PHP Database Abstraction Layer)

This tutorial will show you how to connect a database using Doctrine DBAL. The examples are for Erdiko, but could be applied to any PHP framework that uses composer.

Installation

If you have not installed Erdiko, please go to http://erdiko.org/getStarted.html#installation

To install Erdiko via Composer. simply run

```
composer create-project erdiko/erdiko my-project-name
```

After you have installed Erdiko (or your other favorite framework), it is very easy to install Doctrine.

via composer:

```
composer require doctrine/dbal 2.3.*
```

Alternatively, you can do it by hand by modifying the composer.json file in the root folder. You just need to add this line:

```
{“require”: {“doctrine/dbal”: “2.3.4”}}
```

Then run ‘composer update’ to install Doctrine DBAL.

Basic usage

1. Getting a connection

We can get a connection through the DoctrineDBALDriverManager class.

```
$connectionParams = array(
    ‘dbname’ => ‘database_name’,
    ‘user’ => ‘user_name’,
    ‘password’ => ‘user_password’,
    ‘host’ => ‘localhost’,
    ‘driver’ => ‘pdo_mysql’
);
$conn = DoctrineDBALDriverManager::getConnection($connectionParams, $config);
```

Now, you are ready to retrieve and manipulate data.

2. Data Retrieval And Manipulation

After you have established a connection with database, it is easy to manipulation data.

In this tutorial, we create a table “Products" and create three fields for it.

The three fields are Name, Qty, and Price.

Inserting Data

```
$sql = “INSERT INTO Products (Name, Qty, Price) VALUES (‘Mango’, ’10’,5)”;
$stmt = $conn->query($sql);
```

Retrieving Data

```
$sql = “SELECT * FROM Products”;

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

while ($row = $stmt->fetch()) {

   echo $row[‘Name’].’, Qty: ’.$row[‘Qty’].’, Price: ’.$row[‘Price’];

}
```

The output should be “*Mango, Qty:10, Price:5”.*

Advanced usage:

To perform fancy data manipulation or query, we will need to set up the class loader before establishing a connection.

Setting up class loader

We can open the Example controller(Example.php) under Erdiko/app/controllers/.

In the Example controller, we will need to add the following line before the class scope and it will allow the program to access the class ClassLoader.

```
use DoctrineCommonClassLoader;
```

Then, we can add the following code to a page.

```
$classLoader = new ClassLoader(‘Doctrine’);

$classLoader->register();

$config = new DoctrineDBALConfiguration();
```

Note:

It is not a good practice to set connection parameters every time we get a connection with database. A better way to do it would be storing all these parameters to a config file and creating a data model to read the config file.

For example, we can create a db.json config file under Erdiko/app/config/local/

In the db.json, we can set the connection parameters.

```
{

    “data":{

       “dbname": ‘database_name’,
```

```
        “user": ‘user_name’,
        “password": ‘user_password’,
        “host": ‘localhost’,
        “driver": ‘pdo_mysql’
    }
}
```

Then, we can create a data model and it should looking something like below.

```
public function getDbConfig($dbConfig)
{
    $config = Erdiko::getConfig(“local/db”);
    $connectionParams = array(
        ‘dbname’ => $config[“data"][“dbname”],
        ‘user’ =>  $config[“data"][“user”],
        ‘password’ =>  $config[“data"][“password”],
        ‘host’ =>  $config[“data"][“host”],
        ‘driver’ =>  $config[“data"][“driver”],
    );
    return $connectionParams;
}
```

The getDbConfig function will make the program easy to maintain and also reduce the amount of code.

Now, getting a connection will be only required the following line:

```
$conn =
DoctrineDBALDriverManager::getConnection($connectionParams, getDbConfig(‘data’
));composer create-project erdiko/erdiko project-name
```

https://www.arroyolabs.com/2014/09/tutorial-doctrine-dbal-php-database-abstraction/