Doctrine DBAL Documentation

Release 2.1.0

Roman Borschel, Guilherme Blanco, Benjamin Eberlei, Jonathan

April 29, 2012

Contents

Contents:

CHAPTER 1

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

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

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

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

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

CHAPTER 2

Architecture

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

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

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

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

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

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

2.1 Drivers

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

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

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

2.2 Platforms

The platforms abstract the generation of queries and which database features a platform supports. The `\Doctrine\DBAL\Platforms\AbstractPlatform` defines the common denominator of what a database platform has to publish to the userland, to be fully supportable by Doctrine. This includes the SchemaTool, Transaction Isolation and many other features. The Database platform for MySQL for example can be used by all 3 MySQL extensions, PDO, Mysqli and ext/mysql.

2.3 Logging

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

2.4 Schema

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

2.5 Types

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

CHAPTER 3

Configuration

3.1 Getting a Connection

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

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

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

The following sections describe the available connection parameters in detail.

3.1.1 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.
  - `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`: An MSSQL driver that uses pdo_sqlsrv PDO
  - `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.

3.1.2 Wrapper Class

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

3.1.3 Connection Details

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

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

pdo_sqlite

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

pdo_mysql

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

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.

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.
- `charset` (string): The charset used when connecting to the database.

pdo_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.

3.1.4 Custom Platform

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

3.1.5 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.

CHAPTER 4

Data Retrieval And Manipulation

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

4.1 Data Retrieval

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

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

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

```php
<?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
<?php

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

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

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

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

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

4.1.1 Dynamic Parameters and Prepared Statements

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1.2 Using Prepared Statements

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

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

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

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

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

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

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

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

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

4.2 Binding Types

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

4.2.1 DoctrineDBALTypes Conversion

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

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

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

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

4.2.2 List of Parameters Conversion

Note: This is a Doctrine 2.1 feature.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3 API

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

4.3.1 prepare()

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

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

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

4.3.2 executeUpdate()

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

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

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

4.3.3 executeQuery()

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

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

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

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

4.3.4 fetchAll()

Execute the query and fetch all results into an array:

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

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

4.3.5 fetchArray()

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

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

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

4.3.6 fetchColumn()

Retrieve only the given column of the first result row.

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

4.3.7 fetchAssoc()

Retrieve assoc row of the first result row.

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

There are also convenience methods for data manipulation queries:

4.3.8 delete()

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

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

4.3.9 insert()

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

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

4.3.10 update()

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

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

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

4.3.11 quote()

Quote a value:

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

4.3.12 quoteIdentifier()

Quote an identifier according to the platform details.

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

CHAPTER 5

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
<?php

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

CHAPTER 6

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

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

6.1 Transaction Nesting

A `Doctrine\DBAL\Connection` also adds support for nesting transactions, or rather propagating transaction control up the call stack. For that purpose, the `Connection` class keeps an internal counter that represents the nesting level and is increased/decreased as `beginTransaction()`, `commit()` and

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

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

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

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

```php
<?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 guaruanteed to the inner transaction is that it still happens atomically, all or nothing, the transaction just gets a wider scope and the control is handed to the outer scope.

Note: The transaction nesting described here is a debated feature that has its critics. Form your own opinion. We recommend avoiding nesting transaction blocks when possible, and most of the time, it is possible. Transaction control should mostly be left to a service layer and not be handled in data access objects or similar.

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

CHAPTER 7

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
<?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`.

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
<?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.

CHAPTER 8

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. The following PHP Types are abstracted across all the supported database vendors:

- Integer
- SmallInt
- BigInt
- String (string with maximum length, for example 255)
- Text (strings without maximum length)
- Decimal (restricted floats, *NOTE* Only works with a setlocale() configuration that uses decimal points!)
- Boolean
- DateTime
- Date (DateTime instance where only Y-m-d get persisted)
- Time (DateTime instance where only H:i:s get persisted)
- Array (serialized into a text field for all vendors by default)
- Object (serialized into a text field for all vendors by default)
- Float (*NOTE* Only works with a setlocale() configuration that uses decimal points!)

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()`.

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

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

8.1 Detection of Database Types

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

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

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

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

8.2 Custom Mapping Types

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

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

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

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

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

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

/**
 * My custom datatype.
 */
class MoneyType extends Type
{
    const MONEY = 'money'; // modify to match your type name

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

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

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

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

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

Then you override the methods convertToPhpValueSQL and convertToDatabaseValueSQL :

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

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

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

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

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

CHAPTER 9

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

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

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

9.1 listDatabases()

Retrieve an array of databases on the configured connection:

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

9.2 listSequences()

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

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

Or if you want to manually specify a database name:

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

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

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

9.3 listTableColumns()

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

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

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

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

9.4 listTableDetails()

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

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

9.5 listTableForeignKeys()

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

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

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

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

9.6 listTableIndexes()

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

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

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

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

9.7 listTables()

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

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

9.8 listViews()

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

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

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

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

9.9 createSchema()

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

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

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

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

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

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

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

CHAPTER 10

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 completly 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"));
$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" => "CAS

$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 occour 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 occour with missing links of foreign keys.

CHAPTER 11

Events

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

11.1 PostConnect Event

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

Doctrine ships with one implementation for the "PostConnect" event:

- Doctrine\DBAL\Event\Listeners\OracleSessionInit allows to specify any number of Oracle Session related 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
<?php
$evm = new EventManager();
$evm->addEventSubscriber(new OracleSessionInit(array(
    'NLS_TIME_FORMAT' => 'HH24:MI:SS',
));

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

Security

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

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

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

12.1 User input in your queries

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

12.1.1 Wrong: String Concatenation

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

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

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

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

> <?php // DQL is not safe against arbitrary user-input as well: $dql = “SELECT u FROM User u WHERE u.username = ‘” . $_GET[’username’] . “”’;

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

12.1.2 Right: Prepared Statements

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

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

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

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

```php
<?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 alot:

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

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

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

12.1.3 Right: Quoting/Escaping values

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

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

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

12.2 Non-ASCII compatible Charsets in MySQL

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

CHAPTER 13

Supporting Other Databases

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

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

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

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

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

13.1 Implementation Steps in Detail

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

CHAPTER 14

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 MSSQL). 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:

1. 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.
2. 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()`.
3. 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.
4. 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.
5. 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.
6. 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.
7. Database types like dates, long text fields, booleans and many others are handled very differently between the vendors.
8. 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.

14.1 Connection Wrapper

This functionality is only implemented with Doctrine 2.1 upwards.

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

Using the following code block in your initialization will:

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

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

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

14.2 Database Platform

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

14.3 Keyword Lists

This functionality is only implemented with Doctrine 2.1 upwards.

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

CHAPTER 15

Known Vendor Issues

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

15.1 PostgreSQL

15.1.1 DateTime, DateTimeTz and Time Types

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

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

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

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

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

15.1.2 Timezones and DateTimeTz

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

15.2 MySQL

15.2.1 DateTimeTz

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

15.3 Sqlite

15.3.1 DateTimeTz

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

15.4 IBM DB2

15.4.1 DateTimeTz

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

15.5 Oracle

15.5.1 DateTimeTz

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

15.5.2 OCI8: SQL Queries with Question Marks

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

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

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

```
SELECT * FROM users WHERE name = 'bar:oci1'
```

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

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

15.5.3 OCI-LOB instances

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

15.6 Microsoft SQL Server

15.6.1 Unique and NULL

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

CHAPTER 16

Indices and tables

- *search*