Doctrine Manual Version 1.2.2

Introduction

Code Examples

> The text in this book contains lots of PHP code examples. All starting and ending PHP tags have been removed to reduce the length of the book. Be sure to include the PHP tags when you copy and paste the examples.

What is Doctrine?

Doctrine is an object relational mapper (ORM) for PHP 5.2.3+ that sits on top of a powerful database abstraction layer (DBAL). One of its key features is the option to write database queries in a proprietary object oriented SQL dialect called Doctrine Query Language (DQL), inspired by Hibernates HQL. This provides developers with a powerful alternative to SQL that maintains flexibility without requiring unnecessary code duplication.

What is an ORM?

Object relational mapping is a technique used in programming languages when dealing with databases for translating incompatible data types in relational databases. This essentially allows for us to have a "virtual object database," that can be used from the programming language. Lots of free and commercial packages exist that allow this but sometimes developers chose to create their own ORM.

What is the Problem?

We are faced with many problems when building web applications. Instead of trying to explain it all it is best to read what Wikipedia has to say about object relational mappers.

Pulled from Wikipedia:

Data management tasks in object-oriented (OO) programming are typically implemented by manipulating objects, which are almost always non-scalar values. For example, consider an address book entry that represents a single person along with zero or more phone numbers and zero or more addresses. This could be modeled in an object-oriented implementation by a "person object" with "slots" to hold the data that comprise the entry: the person's name, a list (or array) of phone numbers, and a list of addresses. The list of phone numbers would itself contain "phone number objects" and so on. The address book entry is treated as a single value by the programming language (it can be referenced by a single variable, for instance). Various methods can be associated with the object, such as a method to return the preferred phone number, the home address, and so on.

However, many popular database products such as SQL DBMS can only store and manipulate scalar values such as integers and strings organized within tables.

The programmer must either convert the object values into groups of simpler values for storage in the database (and convert them back upon retrieval), or only use simple scalar values within the program. Object-relational mapping is used to implement the first approach.

The height of the problem is translating those objects to forms that can be stored in the database for easy retrieval, while preserving the properties of the objects and their relationships; these objects are then said to be persistent.

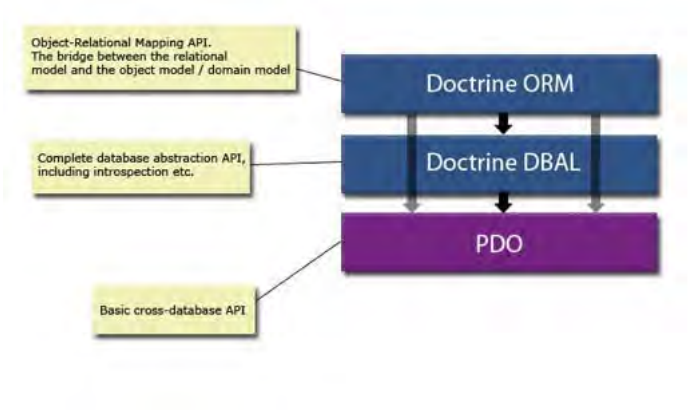
Minimum Requirements

Doctrine requires PHP >= 5.2.3+, although it doesn't require any external libraries. For database function call abstraction Doctrine uses PDO which comes bundled with the PHP official release that you get from www.php.net.

> If you use a 3 in 1 package under windows like Uniform Server, MAMP or any other non-official package, you may be required to perform additional configurations.

Basic Overview

Doctrine is a tool for object-relational mapping in PHP. It sits on top of PDO and is itself divided into two main layers, the DBAL and the ORM. The picture below shows how the layers of Doctrine work together.



The DBAL (Database Abstraction Layer) completes and extends the basic database abstraction/independence that is already provided by PDO. The DBAL library can be used standalone, if all you want is a powerful database abstraction layer on top of PDO. The ORM layer depends on the DBAL and therefore, when you load the ORM package the DBAL is already included.

Doctrine Explained

The following section tries to explain where Doctrine stands in the world of ORM tools. The Doctrine ORM is mainly built around the Active Record, Data Mapper and Meta Data Mapping patterns.

Through extending a specific base class named `Doctrine_Record`, all the child classes get the typical ActiveRecord interface (save/delete/etc.) and it allows Doctrine to easily participate in and monitor the lifecycles of your records. The real work, however, is mostly forwarded to other components, like the `Doctrine_Table` class. This class has the typical Data Mapper interface, `createQuery()`, `find(id)`, `findAll()`, `findBy*()`, `findOneBy*()` etc. So the ActiveRecord base class enables Doctrine to manage your records and provides them with the typical ActiveRecord interface whilst the mapping footwork is done elsewhere.

The ActiveRecord approach comes with its typical limitations. The most obvious is the enforcement for a class to extend a specific base class in order to be persistent (a `Doctrine_Record`). In general, the design of your domain model is pretty much restricted by the design of your relational model. There is an exception though. When dealing with inheritance structures, Doctrine provides some sophisticated mapping strategies which allow your domain model to diverge a bit from the relational model and therefore give you a bit more freedom.

Doctrine is in a continuous development process and we always try to add new features that provide more freedom in the modeling of the domain. However, as long as Doctrine remains mainly an ActiveRecord approach, there will always be a pretty large, (forced) similarity of these two models.

The current situation is depicted in the following picture.

Relational Bounds

As you see in the picture, the domain model can't drift far away from the bounds of the relational model.

After mentioning these drawbacks, it's time to mention some advantages of the ActiveRecord approach. Apart from the (arguably slightly) simpler programming model, it turns out that the strong similarity of the relational model and the Object Oriented (OO) domain model also has an advantage: It makes it relatively easy to provide powerful generation tools, that can create a basic domain model out of an existing relational schema. Further, as the domain model can't drift far from the relational model due to the reasons above, such generation and synchronization tools can easily be used throughout the development process. Such tools are one of Doctrine's strengths.

We think that these limitations of the ActiveRecord approach are not that much of a problem for the majority of web applications because the complexity of the business domains is often moderate, but we also admit that the ActiveRecord approach is certainly not suited for complex business logic (which is often approached using Domain-Driven Design) as it simply puts too many restrictions and has too much influence on your domain model.

Doctrine is a great tool to drive the persistence of simple or moderately complex domain models`(1)` and you may even find that it's a good choice for complex domain models if you consider the trade-off between making your domain model more database-centric and implementing all the mapping on your own (because at the time of this writing we are not aware of any powerful ORM tools for PHP that are not based on an ActiveRecord approach).

> `(1)` Note that complexity != size. A domain model can be pretty large without being complex and vice versa. Obviously, larger domain models have a greater probability of being complex.

Now you already know a lot about what Doctrine is and what it is not. If you would like to dive in now and get started right away, jump straight to the next chapter "Getting Started".

Key Concepts

The Doctrine Query Language (DQL) is an object query language. It let's you express queries for single objects or full object graphs, using the terminology of your domain model: class names, field names, relations between classes, etc. This is a powerful tool for retrieving or even manipulating objects without breaking the separation of the domain model (field names, class names, etc) from the relational model (table names, column names, etc). DQL looks very much like SQL and this is intended because it makes it relatively easy to grasp for people knowing SQL. There are, however, a few very important differences you should always keep in mind:

Take this example DQL query:

```
FROM User u
LEFT JOIN u.Phonenumbers where u.level > 1
```

The things to notice about this query:

- We select from **classes** and not **tables**. We are selecting from the `User` class/model.
- We join along **associations** (u.Phonenumbers)
- We can reference **fields** (u.level)
- There is no join condition (ON x.y = y.x). The associations between your classes and how these are expressed in the database are known to Doctrine (You need to make this mapping known to Doctrine, of course. How to do that is explained later in the Defining Models chapter.).

> DQL expresses a query in the terms of your domain model (your classes, the attributes they have, the relations they have to other classes, etc.).

It's very important that we speak about classes, fields and associations between classes here. `User` is **not** a table / table name . It may be that the name of the database table that the `User` class is mapped to is indeed named `User` but you should nevertheless adhere to this differentiation of terminology. This may sound nit picky since, due to the ActiveRecord approach, your relational model is often very similar to your domain model but it's really important. The column names are rarely the same as the field names and as soon as inheritance is involved, the relational model starts to diverge from the domain model. You can have a class `User` that is in fact mapped to several tables in the database. At this point it should be clear that talking about "selecting from the `User` table" is simply wrong then. And as Doctrine development continues there will be more features available that allow the two models to diverge even more.

Further Reading

For people new to object-relational mapping and (object-oriented) domain models we recommend the following literature:

The books by Martin Fowler cover a lot of the basic ORM terminology, the different approaches of modeling business logic and the patterns involved.

Another good read is about Domain Driven Design. Though serious Domain-Driven Design is currently not possible with Doctrine, this is an excellent resource for good domain modeling, especially in complex business domains, and the terminology around domain models that is pretty widespread nowadays is explained in depth (Entities, Value Objects, Repositories, etc).

Conclusion

Well, now that we have given a little educational reading about the methodologies and principals behind Doctrine we are pretty much ready to dive in to everything that is Doctrine. Lets dive in to setting up Doctrine in the Getting Started chapter.

Getting Started

Checking Requirements

First we need to make sure that you can run Doctrine on your server. We can do this one of two ways:

First create a small PHP script named `phpinfo.php` and upload it somewhere on your web server that is accessible to the web:

```
phpinfo();
```

Now execute it from your browser by going to `http://localhost/phpinfo.php`. You will see a list of information detailing your PHP configuration. Check that your PHP version is >= 5.2.3 and that you have PDO and the desired drivers installed.

> You can also check your PHP installation has the necessary requirements by running some commands from the terminal. We will demonstrate in the next example.

Check that your PHP version is >= 5.2.3 with the following command:

```
$ php -v
```

Now check that you have PDO and the desired drivers installed with the following command:

```
$ php -i
```

You could also execute the `phpinfo.php` from the command line and get the same result as the above example:

```
$ php phpinfo.php
```

> Checking the requirements are required in order to run the examples used throughout this documentation.

Installing

Currently it is possible to install Doctrine four different ways that are listed below:

- SVN (subversion)
- SVN externals
- PEAR Installer
- Download PEAR Package

It is recommended to download Doctrine via SVN (subversion), because in this case updating is easy. If your project is already under version control with SVN, you should choose SVN externals.

> If you wish to just try out Doctrine in under 5 minutes, the sandbox package is recommended. We will discuss the sandbox package in the next section.

Sandbox

Doctrine also provides a special package which is a zero configuration Doctrine implementation for you to test Doctrine without writing one line of code. You can download it from the download page.

> The sandbox implementation is not a recommend implementation for a production application. It's only purpose is for exploring Doctrine and running small tests.

SVN

It is highly recommended that you use Doctrine via SVN and the externals option. This option is the best as you will receive the latest bug fixes from SVN to ensure the best experience using Doctrine.

Installing

To install Doctrine via SVN is very easy. You can download any version of Doctrine from the SVN server: `http://svn.doctrine-project.org`

To check out a specific version you can use the following command from your terminal:

```
svn co http://svn.doctrine-project.org/branches/1.2 .
```

If you do not have a SVN client, chose one from the list below. Find the Checkout option and enter http://svn.doctrine-project.org/branches/1.2 in the path or repository url parameter. There is no need for a username or password to check out Doctrine.

- TortoiseSVN a Windows application that integrates into Windows Explorer
- svnx a Mac OS X GUI svn application
- Eclipse has SVN integration through the subeclipse plugin
- Versions a subversion client for the mac

Updating

Updating Doctrine with SVN is just as easy as installing. Simply execute the following command from your terminal:

```
$ svn update
```

SVN Externals

If your project is already under version control with SVN, then it is recommended that you use SVN externals to install Doctrine.

You can start by navigating to your checked out project in your terminal:

```
$ cd /var/www/my_project
```

Now that you are under your checked out project, you can execute the following command from your terminal and setup Doctrine as an SVN external:

```
$ svn propedit svn:externals lib/vendor
```

The above command will open your editor and you need to place the following text inside and save:

```
doctrine http://svn.doctrine-project.org/branches/1.2/lib
```

Now you can install Doctrine by doing an svn update:

```
$ svn update
```

It will download and install Doctrine at the following path: `/var/www/my_project/lib/vendor/doctrine`

> Don't forget to commit your change to the SVN externals.

```
$ svn commit
```

PEAR Installer

Doctrine also provides a PEAR server for installing and updating Doctrine on your servers. You can easily install Doctrine with the following command:

```
$ pear install pear.phpdoctrine.org/Doctrine-1.2.x
```

> Replace the above 1.2.x with the version you wish to install. For example "1.2.1".

Download Pear Package

If you do not wish to install via PEAR or do not have PEAR installed, you can always just manually download the package from the website. Once you download the package to your server you can extract it using the following command under linux.

```
$ tar xzf Doctrine-1.2.1.tgz
```

Implementing

Now that you have Doctrine in your hands, we are ready to implement Doctrine in to our application. This is the first step towards getting started with Doctrine.

First create a directory named `doctrine_test`. This is where we will place all our test code:

```
$ mkdir doctrine_test
$ cd doctrine_test
```

Including Doctrine Libraries

The first thing we must do is find the `Doctrine.php` file containing the core class so that we can require it in to our application. The `Doctrine.php` file is in the lib folder from when you downloaded Doctrine in the previous section.

We need to move the Doctrine libraries in to the `doctrine_test` directory into a folder in `doctrine_test/lib/vendor/doctrine`:

```
$ mkdir lib
$ mkdir lib/vendor
$ mkdir lib/vendor/doctrine
$ mv /path/to/doctrine/lib doctrine
```

Or if you are using SVN, you can use externals:

```
$ svn co http://svn.doctrine-project.org/branches/1.2/lib lib/vendor/doctrine
```

Now add it to your svn externals:

```
$ svn propedit svn:externals lib/vendor
```

It will open up your editor and place the following inside and save:

```
doctrine http://svn.doctrine-project.org/branches/1.2/lib
```

Now when you do SVN update you will get the Doctrine libraries updated:

```
$ svn update lib/vendor
```

Require Doctrine Base Class

We need to create a php script for bootstrapping Doctrine and all the configuration for it. Create a file named `bootstrap.php` and place the following code in the file:

```php
// bootstrap.php

/**
 * Bootstrap Doctrine.php, register autoloader specify
 * configuration attributes and load models.
 */

require_once(dirname(__FILE__) . '/lib/vendor/doctrine/Doctrine.php');
```

Register Autoloader

Now that we have the `Doctrine` class present, we need to register the class autoloader function in the bootstrap file:

```php
// bootstrap.php

// ...
spl_autoload_register(array('Doctrine', 'autoload'));
```

Lets also create the singleton `Doctrine_Manager` instance and assign it to a variable named `$manager`:

```php
// bootstrap.php

// ...
$manager = Doctrine_Manager::getInstance();
```

Autoloading Explained

> You can read about the PHP autoloading on the php website. Using the autoloader allows us to lazily load classes as they are requested instead of pre-loading all classes. This is a huge benefit to performance.

The way the Doctrine autoloader works is simple. Because our class names and paths are related, we can determine the path to a Doctrine class based on its name.

Imagine we have a class named `Doctrine_Some_Class` and we instantiate an instance of it:

```
$class = new Doctrine_Some_Class();
```

The above code will trigger a call to the `Doctrine_Core::autoload()` function and pass it the name of the class instantiated. The class name string is manipulated and transformed in to a path and required. Below is some pseudo code that shows how the class is found and required:

```
class Doctrine
{
    public function autoload($className)
    {
        $classPath = str_replace('_', '/', $className) . '.php';
        $path = '/path/to/doctrine/' . $classPath;
        require_once($path);
        return true;
    }
}
```

In the above example the `Doctrine_Some_Class` can be found at `/path/to/doctrine/Doctrine/Some/Class.php`.

> Obviously the real `Doctrine_Core::autoload()` function is a bit more complex and has some error checking to ensure the file exists but the above code demonstrates how it works.

Bootstrap File

> We will use this bootstrap class in later chapters and sections so be sure to create it!

The bootstrap file we have created should now look like the following:

```
// bootstrap.php

/**
 * Bootstrap Doctrine.php, register autoloader specify
 * configuration attributes and load models.
 */

require_once(dirname(__FILE__) . '/lib/vendor/doctrine/Doctrine.php');
spl_autoload_register(array('Doctrine', 'autoload'));
$manager = Doctrine_Manager::getInstance();
```

This new bootstrapping file will be referenced several times in this book as it is where we will make changes to our implementation as we learn how to use Doctrine step by step.

> The configuration attributes mentioned above are a feature in Doctrine used for configuring and controlling functionality. You will learn more about attributes and how to get/set them in the Configuration chapter.

Test Script

Now lets create a simple test script that we can use to run various tests as we learn about the features of Doctrine.

Create a new file in the `doctrine_test` directory named `test.php` and place the following code inside:

```
// test.php

require_once('bootstrap.php');

echo Doctrine_Core::getPath();
```

Now you can execute the test script from your command line. This is how we will perform tests with Doctrine throughout the chapters so make sure it is working for you! It should output the path to your Doctrine installation.

```
$ php test.php
/path/to/doctrine/lib
```

Conclusion

Phew! This was our first chapter where we actually got into some code. As you saw, first we were able to check that our server can actually run Doctrine. Then we learned all the different ways we can download and install Doctrine. Lastly we learned how to implement Doctrine by setting up a small test environment that we will use to perform some exercises in the remaining chapters of the book.

Now lets move on and get our first taste of Doctrine connections in the Introduction to Connections chapter.

Introduction to Connections

DSN, the Data Source Name

In order to connect to a database through Doctrine, you have to create a valid DSN (Data Source Name).

Doctrine supports both PEAR DB/MDB2 like data source names as well as PDO style data source names. The following section deals with PEAR like data source names. If you need more info about the PDO-style data source names see the documentation on PDO.

The DSN consists in the following parts:

DSN part	Description
phptype	Database backend used in PHP (i.e. mysql , pgsql etc.)
dbsyntax	Database used with regards to SQL syntax etc.
protocol	Communication protocol to use (i.e. tcp, unix etc.)
hostspec	Host specification (hostname[:port])
database	Database to use on the DBMS server
username	User name for login
password	Password for login
proto_opts	Maybe used with protocol
option	Additional connection options in URI query string format. Options are separated by ampersand (&). The Following table shows a non complete list of options:

List of options

Name	Description
charset	Some backends support setting the client charset.
new_link	Some RDBMS do not create new connections when connecting to the same host multiple times. This option will attempt to force a new connection.

The DSN can either be provided as an associative array or as a string. The string format of the supplied DSN is in its fullest form:

```
phptype(dbsyntax)://username:password@protocol+hostspec/database?option=value
```

Most variations are allowed:

```
phptype://username:password@protocol+hostspec:110//usr/db_file.db
phptype://username:password@hostspec/database
phptype://username:password@hostspec
phptype://username@hostspec
phptype://hostspec/database
phptype://hostspec
phptype:///database
phptype:///database?option=value&anotheroption=anothervalue
phptype(dbsyntax)
phptype
```

The currently supported PDO database drivers are:

Driver name	Supported databases
fbsql	FrontBase
ibase	InterBase / Firebird (requires PHP 5)
mssql	Microsoft SQL Server (NOT for Sybase. Compile PHP --with-mssql)
mysql	MySQL
mysqli	MySQL (supports new authentication protocol) (requires PHP 5)
oci	Oracle 7/8/9/10
pgsql	PostgreSQL

querysim	QuerySim
sqlite	SQLite 2

A second DSN format supported is

```
phptype(syntax)://user:pass@protocol(proto_opts)/database
```

If your database, option values, username or password contain characters used to delineate DSN parts, you can escape them via URI hex encodings:

Character	Hex Code
:	%3a
/	%2f
@	%40
+	%2b
(	%28
)	%29
?	%3f
=	%3d
&	%26

Please note, that some features may be not supported by all database drivers.

Examples

Example 1. Connect to database through a socket

```
mysql://user@unix(/path/to/socket)/pear
```

Example 2. Connect to database on a non standard port

```
pgsql://user:pass@tcp(localhost:5555)/pear
```

> If you use, the IP address `127.0.0.1`, the port parameter is ignored (default: 3306).

Example 3. Connect to SQLite on a Unix machine using options

```
sqlite:////full/unix/path/to/file.db?mode=0666
```

Example 4. Connect to SQLite on a Windows machine using options

```
sqlite:///c:/full/windows/path/to/file.db?mode=0666
```

Example 5. Connect to MySQLi using SSL

```
mysqli://user:pass@localhost/pear?key=client-key.pem&cert=client-cert.pem
```

Opening New Connections

Opening a new database connection in Doctrine is very easy. If you wish to use PDO you can just initialize a new PDO object.

Remember our `bootstrap.php` file we created in the Getting Started chapter? Under the code where we registered the Doctrine autoloader we are going to instantiate our new connection:

```
// bootstrap.php

// ...
$dsn = 'mysql:dbname=testdb;host=127.0.0.1';
$user = 'dbuser';
$password = 'dbpass';

$dbh = new PDO($dsn, $user, $password);
$conn = Doctrine_Manager::connection($dbh);
```

> Directly passing a PDO instance to `Doctrine_Manager::connection()` will not allow Doctrine to be aware of the username and password for the connection, since their is no way to retrieve it from an existing PDO instance. The username and password is required in order for Doctrine to be able to create and drop databases. To get around this you can manually set the username and password option directly on the `$conn` object.

```
// bootstrap.php

// ...
$conn->setOption('username', $user);
$conn->setOption('password', $password);
```

Lazy Database Connecting

Lazy-connecting to database can save a lot of resources. There might be many times where you don't need an actual database connection, hence its always recommended to use lazy-connecting (that means Doctrine will only connect to database when needed).

This feature can be very useful when using for example page caching, hence not actually needing a database connection on every request. Remember connecting to database is an expensive operation.

In the example below we will show you when you create a new Doctrine connection, the connection to the database isn't created until it is actually needed.

```
// bootstrap.php

// ...

// At this point no actual connection to the database is created
$conn = Doctrine_Manager::connection('mysql://username:password@localhost/test');

// The first time the connection is needed, it is instantiated
// This query triggers the connection to be created
$conn->execute('SHOW TABLES');
```

Testing your Connection

After reading the previous sections of this chapter, you should now know how to create a connection. So, lets modify our bootstrap file to include the initialization of a connection. For this example we will just be using a sqlite memory database but you can use whatever type of database connection you prefer.

Add your database connection to `bootstrap.php` and it should look something like the following:

```
/**
 * Bootstrap Doctrine.php, register autoloader and specify
 * configuration attributes
 */

require_once('../doctrine/branches/1.2/lib/Doctrine.php');
spl_autoload_register(array('Doctrine', 'autoload'));
$manager = Doctrine_Manager::getInstance();

$conn = Doctrine_Manager::connection('sqlite::memory:', 'doctrine');
```

To test the connection lets modify our `test.php` script and perform a small test. Since we create a variable name `$conn`, that variable is available to the test script so lets setup a small test to make sure our connection is working:

First lets create a test table and insert a record:

```
// test.php

// ...
$conn->export->createTable('test', array('name' => array('type' => 'string')));
$conn->execute('INSERT INTO test (name) VALUES (?)', array('jwage'));
```

Now lets execute a simple `SELECT` query from the `test` table we just created to make sure the data was inserted and that we can retrieve it:

```
// test.php

// ...
$stmt = $conn->prepare('SELECT * FROM test');
$stmt->execute();
$results = $stmt->fetchAll();
print_r($results);
```

Execute `test.php` from your terminal and you should see:

```
$ php test.php
Array
(
    [0] => Array
        (
            [name] => jwage
            [0] => jwage
        )

)
```

Conclusion

Great! Now we learned some basic operations of Doctrine connections. We have modified our Doctrine test environment to have a new connection. This is required because the examples in the coming chapters will require a connection.

Lets move on to the Configuration chapter and learn how you can control functionality and configurations using the Doctrine attribute system.

Configuration

Doctrine controls configuration of features and functionality using attributes. In this section we will discuss how to set and get attributes as well as an overview of what attributes exist for you to use to control Doctrine functionality.

Levels of Configuration

Doctrine has a three-level configuration structure. You can set configuration attributes at a global, connection and table level. If the same attribute is set on both lower level and upper level, the uppermost attribute will always be used. So for example if a user first sets default fetchmode in global level to `Doctrine_Core::FETCH_BATCH` and then sets a table fetchmode to `Doctrine_Core::FETCH_LAZY`, the lazy fetching strategy will be used whenever the records of that table are being fetched.

- **Global level** – The attributes set in global level will affect every connection and every table in each connection.
- **Connection level** – The attributes set in connection level will take effect on each table in that connection.
- **Table level** – The attributes set in table level will take effect only on that table.

In the following example we set an attribute at the global level:

```
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_VALIDATE, Doctrine_Core::VALIDATE_ALL);
```

In the next example above we override the global attribute on given connection:

```
// bootstrap.php

// ...
$conn->setAttribute(Doctrine_Core::ATTR_VALIDATE, Doctrine_Core::VALIDATE_NONE);
```

In the last example we override once again the connection level attribute in the table level:

```
// bootstrap.php

// ...
$table = Doctrine_Core::getTable('User');

$table->setAttribute(Doctrine_Core::ATTR_VALIDATE, Doctrine_Core::VALIDATE_ALL);
```

> We haven't introduced the above used `Doctrine_Core::getTable()` method. You will learn more about the table objects used in Doctrine in the Table section of the next chapter.

Defaults Attributes

Doctrine has a few specific attributes available that allow you to specify the default values of things that in the past were hardcoded values. Such as default column length, default column type, etc.

Default Column Options

It is possible to specify an array of default options to be used on every column in your model.

```
// bootstrap.php

// ...

$manager->setAttribute(Doctrine::ATTR_DEFAULT_COLUMN_OPTIONS,
  array('type' => 'string', 'length' => 255, 'notnull' => true));
```

Default Added Auto Id

You can customize the properties of the automatically added primary key in Doctrine models.

```
$manager->setAttribute(Doctrine::ATTR_DEFAULT_IDENTIFIER_OPTIONS,
  array('name' => '%s_id', 'type' => 'string', 'length' => 16));
```

> The `%s` string in the name is replaced with the table name.

Portability

Each database management system (DBMS) has it's own behaviors. For example, some databases capitalize field names in their output, some lowercase them, while others leave them alone. These quirks make it difficult to port your applications over to another database type. Doctrine strives to overcome these differences so your applications can switch between DBMS's without any changes. For example switching from sqlite to mysql.

The portability modes are bitwised, so they can be combined using | and removed using ^. See the examples section below on how to do this.

> You can read more about the bitwise operators on the PHP website.

Portability Mode Attributes

Below is a list of all the available portability attributes and the description of what each one does:

Name	Description
`PORTABILITY_ALL`	Turn on all portability features. This is the default setting.
`PORTABILITY_DELETE_COUNT`	Force reporting the number of rows deleted. Some DBMS's don't count the number of rows deleted when performingsimple `DELETE FROM` tablename queries. This mode tricks such DBMS's into telling the count by adding `WHERE 1=1` to the end of `DELETE` queries.
`PORTABILITY_EMPTY_TO_NULL`	Convert empty strings values to null in data in and output. Needed because Oracle considers empty strings to be null, while most other DBMS's know the difference between empty and null.
`PORTABILITY_ERRORS`	Makes certain error messages in certain drivers compatible with those from other DBMS's
`PORTABILITY_FIX_ASSOC_FIELD_NAMES`	This removes any qualifiers from keys in associative fetches. Some RDBMS, like for example SQLite, will by default use the fully qualified name for a column in assoc fetches if it is qualified in a query.
`PORTABILITY_FIX_CASE`	Convert names of tables and fields to lower or upper case in all methods. The case depends on the field_case option that may be set to either `CASE_LOWER` (default) or `CASE_UPPER`
`PORTABILITY_NONE`	Turn off all portability features.
`PORTABILITY_NUMROWS`	Enable hack that makes `numRows()` work in Oracle.
`PORTABILITY_EXPR`	Makes DQL API throw exceptions when non-portable expressions are being used.
`PORTABILITY_RTRIM`	Right trim the data output for all data fetches. This does not applied in drivers for RDBMS that automatically right trim values of fixed length character values, even if they do not right trim value of variable length character values.

Examples

Now we can use the `setAttribute()` method to enable portability for lowercasing and trimming with the following code:

```
// bootstrap.php

// ...
$conn->setAttribute(Doctrine_Core::ATTR_PORTABILITY,
        Doctrine_Core::PORTABILITY_FIX_CASE | Doctrine_Core::PORTABILITY_RTRIM);
```

Enable all portability options except trimming

```
// bootstrap.php

// ...
$conn->setAttribute(Doctrine_Core::ATTR_PORTABILITY,
        Doctrine_Core::PORTABILITY_ALL ^ Doctrine_Core::PORTABILITY_RTRIM);
```

Identifier quoting

You can quote the db identifiers (table and field names) with `quoteIdentifier()`. The delimiting style depends on which database driver is being used.

> Just because you CAN use delimited identifiers, it doesn't mean you SHOULD use them. In general, they end up causing way more problems than they solve. Anyway, it may be necessary when you have a reserved word as a field name (in this case, we suggest you to change it, if you can).

Some of the internal Doctrine methods generate queries. Enabling the `quote_identifier` attribute of Doctrine you can tell Doctrine to quote the identifiers in these generated queries. For all user supplied queries this option is irrelevant.

Portability is broken by using the following characters inside delimited identifiers:

Name	Character	Driver
backtick	`	MySQL
double quote	"	Oracle
brackets	[or]	Access

Delimited identifiers are known to generally work correctly under the following drivers: Mssql, Mysql, Oracle, Pgsql, Sqlite and Firebird.

When using the `Doctrine_Core::ATTR_QUOTE_IDENTIFIER` option, all of the field identifiers will be automatically quoted in the resulting SQL statements:

```
// bootstrap.php

// ...
$conn->setAttribute(Doctrine_Core::ATTR_QUOTE_IDENTIFIER, true);
```

Will result in a SQL statement that all the field names are quoted with the backtick '``' operator (in MySQL).

```
SELECT
*
FROM sometable
WHERE `id` = '123'
```

As opposed to:

```
SELECT
*
FROM sometable
WHERE id = '123'
```

Hydration Overwriting

By default Doctrine is configured to overwrite any local changes you have on your objects if you were to query for some objects which have already been queried for and modified.

```
$user = Doctrine_Core::getTable('User')->find(1);
$user->username = 'newusername';
```

Now I have modified the above object and if I were to query for the same data again, my local changes would be overwritten.

```
$user = Doctrine_Core::getTable('User')->find(1);
echo $user->username; // would output original username in database
```

You can disable this behavior by using the `ATTR_HYDRATE_OVERWRITE` attribute:

```
// bootstrap.php

// ...
$conn->setAttribute(Doctrine_Core::ATTR_HYDRATE_OVERWRITE, false);
```

Now if were to run the same test we ran above, the modified username would not be overwritten.

Configure Table Class

If you want to configure the class to be returned when using the `Doctrine_Core::getTable()` method you can set the `ATTR_TABLE_CLASS` attribute. The only requirement is that the class extends `Doctrine_Table`.

```
// bootstrap.php

// ...
$conn->setAttribute(Doctrine_Core::ATTR_TABLE_CLASS, 'MyTableClass');
```

Now the `MyTableClass` would look like the following:

```php
class MyTableClass extends Doctrine_Table
{
    public function myMethod()
    {
        // run some query and return the results
    }
}
```

Now when you do the following it will return an instance of `MyTableClass`:

```php
$user = $conn->getTable('MyModel')->myMethod();
```

If you want to customize the table class even further you can customize it for each model. Just create a class named `MyModelTable` and make sure it is able to be autoloaded.

```php
class MyModelTable extends MyTableClass
{
    public function anotherMethod()
    {
        // run some query and return the results
    }
}
```

Now when I execute the following code it will return an instance of `MyModelTable`:

```php
echo get_class($conn->getTable('MyModel')); // MyModelTable
```

Configure Query Class

If you would like to configure the base query class returned when you create new query instances, you can use the `ATTR_QUERY_CLASS` attribute. The only requirement is that it extends the `Doctrine_Query` class.

```php
// bootstrap.php

// ...
$conn->setAttribute(Doctrine_Core::ATTR_QUERY_CLASS, 'MyQueryClass');
```

The `MyQueryClass` would look like the following:

```php
class MyQueryClass extends Doctrine_Query
{

}
```

Now when you create a new query it will return an instance of `MyQueryClass`:

```php
$q = Doctrine_Core::getTable('User')
    ->createQuery('u');

echo get_class($q); // MyQueryClass
```

Configure Collection Class

Since you can configure the base query and table class, it would only make sense that you can also customize the collection class Doctrine should use. We just need to set the `ATTR_COLLECTION_CLASS` attribute.

```php
// bootstrap.php

// ...
$conn->setAttribute(Doctrine_Core::ATTR_COLLECTION_CLASS, 'MyCollectionClass');
```

The only requirement of the `MyCollectionClass` is that it must extend `Doctrine_Collection`:

```php
$phonenumbers = $user->Phonenumber;
echo get_class($phonenumbers); // MyCollectionClass
```

Disabling Cascading Saves

You can optionally disable the cascading save operations which are enabled by default for convenience with the `ATTR_CASCADE_SAVES` attribute. If you set this attribute to `false` it will only cascade and save if the record is dirty. This means that you can't cascade and save records who are dirty that are more than one level deep in the hierarchy, but you benefit with a significant performance
improvement.

```
$conn->setAttribute(Doctrine_Core::ATTR_CASCADE_SAVES, false);
```

Exporting

The export attribute is used for telling Doctrine what it should export when exporting classes to your database for creating your tables.

If you don't want to export anything when exporting you can use:

```
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_EXPORT, Doctrine_Core::EXPORT_NONE);
```

For exporting tables only (but not constraints) you can use on of the following:

```
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_EXPORT, Doctrine_Core::EXPORT_TABLES);
```

You can also use the following syntax as it is the same as the above:

```
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_EXPORT,
  Doctrine_Core::EXPORT_ALL ^ Doctrine_Core::EXPORT_CONSTRAINTS);
```

For exporting everything (tables and constraints) you can use:

```
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_EXPORT, Doctrine_Core::EXPORT_ALL);
```

Naming convention attributes

Naming convention attributes affect the naming of different database related elements such as tables, indexes and sequences. Basically every naming convention attribute has affect in both ways. When importing schemas from the database to classes and when exporting classes into database tables.

So for example by default Doctrine naming convention for indexes is `%s_idx`. Not only do the indexes you set get a special suffix, also the imported classes get their indexes mapped to their non-suffixed equivalents. This applies to all naming convention attributes.

Index name format

`Doctrine_Core::ATTR_IDXNAME_FORMAT` can be used for changing the naming convention of indexes. By default Doctrine uses the format `[name]_idx`. So defining an index called 'ageindex' will actually be converted into 'ageindex_idx'.

You can change the index naming convention with the following code:

```
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_IDXNAME_FORMAT, '%s_index');
```

Sequence name format

Similar to `Doctrine_Core::ATTR_IDXNAME_FORMAT`, `Doctrine_Core::ATTR_SEQNAME_FORMAT` can be used for changing the naming convention of sequences. By default Doctrine uses the format `[name]_seq`, hence creating a new sequence with the name of `mysequence` will lead into creation of sequence called `mysequence_seq`.

You can change the sequence naming convention with the following code:

```php
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_SEQNAME_FORMAT, '%s_sequence');
```

Table name format

The table name format can be changed the same as the index and sequence name format with the following code:

```php
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_TBLNAME_FORMAT, '%s_table');
```

Database name format

The database name format can be changed the same as the index, sequence and table name format with the following code:

```php
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_DBNAME_FORMAT, 'myframework_%s');
```

Validation attributes

Doctrine provides complete control over what it validates. The validation procedure can be controlled with `Doctrine_Core::ATTR_VALIDATE`.

The validation modes are bitwised, so they can be combined using `|` and removed using `^`. See the examples section below on how to do this.

Validation mode constants

Name	Description
`VALIDATE_NONE`	Turns off the whole validation procedure.
`VALIDATE_LENGTHS`	Makes Doctrine validate all field lengths.
`VALIDATE_TYPES`	Makes Doctrine validate all field types. Doctrine does loose type validation. This means that for example string with value '13.3' will not pass as an integer but '13' will.
`VALIDATE_CONSTRAINTS`	Makes Doctrine validate all field constraints such as `notnull`, `email` etc.
`VALIDATE_ALL`	Turns on all validations.

> Validation by default is turned off so if you wish for your data to be validated you will need to enable it. Some examples of how to change this configuration are provided below.

Examples

You can turn on all validations by using the `Doctrine_Core::VALIDATE_ALL` attribute with the following code:

```php
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_VALIDATE, Doctrine_Core::VALIDATE_ALL);
```

You can also configure Doctrine to validate lengths and types, but not constraints with the following code:

```php
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_VALIDATE,
  Doctrine_Core::VALIDATE_LENGTHS | Doctrine_Core::VALIDATE_TYPES);
```

Conclusion

Now we have gone over some of the most common attributes used to configure Doctrine. Some of these attributes may not apply to you ever or you may not understand what you could use them for now. As you read the next chapters you will see which attributes you do and don't need to use and things will begin to make more sense.

If you saw some attributes you wanted to change the value above, then you should have added it to your `bootstrap.php` file and it should look something like the following now:

```php
/**
 * Bootstrap Doctrine.php, register autoloader and specify
 * configuration attributes
 */

require_once('../doctrine/branches/1.2/lib/Doctrine.php');
spl_autoload_register(array('Doctrine', 'autoload'));
$manager = Doctrine_Manager::getInstance();

$conn = Doctrine_Manager::connection('sqlite::memory:', 'doctrine');

$manager->setAttribute(Doctrine_Core::ATTR_VALIDATE, Doctrine_Core::VALIDATE_ALL);
$manager->setAttribute(Doctrine_Core::ATTR_EXPORT, Doctrine_Core::EXPORT_ALL);
$manager->setAttribute(Doctrine_Core::ATTR_MODEL_LOADING, Doctrine_Core::MODEL_LOADING_CONSERVATIVE);
```

Now we are ready to move on to the next chapter where we will learn everything there is to know about Doctrine Connections.

Connections

Introduction

From the start Doctrine has been designed to work with multiple connections. Unless separately specified Doctrine always uses the current connection for executing the queries.

In this chapter we will demonstrate how to create and work with Doctrine connections.

Opening Connections

`Doctrine_Manager` provides the static method `Doctrine_Manager::connection()` which opens new connections.

In this example we will show you to open a new connection:

```
// test.php

// ...
$conn = Doctrine_Manager::connection('mysql://username:password@localhost/test', 'connection 1');
```

Retrieve Connections

If you use the `Doctrine_Manager::connection()` method and don't pass any arguments it will return the current connection:

```
// test.php

// ...
$conn2 = Doctrine_Manager::connection();

if ($conn === $conn2) {
    echo 'Doctrine_Manager::connection() returns the current connection';
}
```

Current Connection

The current connection is the last opened connection. In the next example we will show how you can get the current connection from the `Doctrine_Manager` instance:

```
// test.php

// ...
$conn2 = Doctrine_Manager::connection('mysql://username2:password2@localhost/test2', 'connection 2');

if ($conn2 === $manager->getCurrentConnection()) {
    echo 'Current connection is the connection we just created!';
}
```

Change Current Connection

You can change the current connection by calling `Doctrine_Manager::setCurrentConnection()`.

```
// test.php

// ...
$manager->setCurrentConnection('connection 1');

echo $manager->getCurrentConnection()->getName(); // connection 1
```

Iterating Connections

You can iterate over the opened connections by simply passing the manager object to a foreach clause. This is possible since `Doctrine_Manager` implements special `IteratorAggregate` interface.

> The `IteratorAggregate` is a special PHP interface for implementing iterators in to your objects.

```php
// test.php

// ...
foreach($manager as $conn) {
    echo $conn->getName() . "\n";
}
```

Get Connection Name

You can easily get the name of a `Doctrine_Connection` instance with the following code:

```php
// test.php

// ...
$conn = Doctrine_Manager::connection();

$name = $manager->getConnectionName($conn);

echo $name; // connection 1
```

Close Connection

You can easily close a connection and remove it from the Doctrine connection registry with the following code:

```php
// test.php

// ...
$conn = Doctrine_Manager::connection();

$manager->closeConnection($conn);
```

If you wish to close the connection but not remove it from the Doctrine connection registry you can use the following code instead:

```php
// test.php

// ...
$conn = Doctrine_Manager::connection();
$conn->close();
```

Get All Connections

You can retrieve an array of all the registered connections by using the `Doctrine_Manager::getConnections()` method like below:

```php
// test.php

// ...
$conns = $manager->getConnections();
foreach ($conns as $conn) {
    echo $conn->getName() . "\n";
}
```

The above is essentially the same as iterating over the `Doctrine_Manager` object like we did earlier. Here it is again:

```php
// test.php

// ...
foreach ($manager as $conn) {
    echo $conn->getName() . "\n";
}
```

Count Connections

You can easily get the number of connections from a `Doctrine_Manager` object since it implements the `Countable` interface.

```php
// test.php

// ...
$num = count($manager);

echo $num;
```

The above is the same as doing:

```php
// test.php

// ...
$num = $manager->count();
```

Creating and Dropping Database

When you create connections using Doctrine, you gain the ability to easily create and drop the databases related to those connections.

This is as simple as using some functions provided in the `Doctrine_Manager` or `Doctrine_Connection` classes.

The following code will iterate over all instantiated connections and call the `dropDatabases()`/`createDatabases()` function on each one:

```php
// test.php

// ...
$manager->createDatabases();

$manager->dropDatabases();
```

Drop/create database for specific connection

You can easily drop or create the database for a specific `Doctrine_Connection` instance by calling the `dropDatabase()`/`createDatabase()` function on the connection instance with the following code:

```php
// test.php

// ...
$conn->createDatabase();

$conn->dropDatabase();
```

Writing Custom Connections

Sometimes you might need the ability to create your own custom connection classes and utilize them. You may need to extend mysql, or write your own connection type completely. This is possible by writing a few classes and then registering the new connection type with Doctrine.

So in order to create a custom connection first we need to write the following classes.

```php
class Doctrine_Connection_Test extends Doctrine_Connection_Common
{
}

class Doctrine_Adapter_Test implements Doctrine_Adapter_Interface
{
    // ... all the methods defined in the interface
}
```

Now we register them with Doctrine:

```php
// bootstrap.php

// ...
$manager->registerConnectionDriver('test', 'Doctrine_Connection_Test');
```

With those few changes something like this is now possible:

```php
$conn = $manager->openConnection('test://username:password@localhost/dbname');
```

If we were to check what classes are used for the connection you will notice
that they are the classes we defined above.

```php
echo get_class($conn); // Doctrine_Connection_Test
echo get_class($conn->getDbh()); // Doctrine_Adapter_Test
```

Conclusion

Now that we have learned all about Doctrine connections we should be ready to dive right in to models in the Introduction to Models chapter. We will learn a little bit about Doctrine models first. Then we will start to have some fun and create our first test models and see what kind of magic Doctrine can provide for you.

Introduction to Models

Introduction

At the lowest level, Doctrine represents your database schema with a set of PHP classes. These classes define the schema and behavior of your model.

A basic model that represents a user in a web application might look something like this.

```
class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 255);
        $this->hasColumn('password', 'string', 255);
    }

    public function setUp()
    {
        $this->actAs('Timestampable');
    }
}
```

> We aren't actually going to use the above class definition, it is only meant to be an example. We will generate our first class definition from an existing database table later in this chapter.

Each `Doctrine_Record` child class can have a `setTableDefinition()` and `setUp()` method. The `setTableDefinition()` method is for defining columns, indexes and other information about the schema of tables. The `setUp()` method is for attaching behaviors and defining relationships between `Doctrine_Record` child classes. In the above example we are enabling the Timestampable behavior which adds some automagic functionality. You will learn more about what all can be used in these functions in the Defining Models chapter.

Generating Models

Doctrine offers ways to generate these classes to make it easier to get started using Doctrine.

> Generating from existing databases is only meant to be a convenience for getting started. After you generate from the database you will have to tweak it and clean things up as needed.

Existing Databases

A common case when looking for ORM tools like Doctrine is that the database and the code that access it is growing large/complex. A more substantial tool is needed than manual SQL code.

Doctrine has support for generating `Doctrine_Record` classes from your existing database. There is no need for you to manually write all the `Doctrine_Record` classes for your domain model.

Making the first import

Let's consider we have a mysql database called `doctrine_test` with a single table named `user`. The `user` table has been created with the following sql statement:

```
CREATE TABLE user (
   id bigint(20) NOT NULL auto_increment,
   first_name varchar(255) default NULL,
   last_name varchar(255) default NULL,
   username varchar(255) default NULL,
   password varchar(255) default NULL,
   type varchar(255) default NULL,
   is_active tinyint(1) default '1',
   is_super_admin tinyint(1) default '0',
   created_at TIMESTAMP,
   updated_at TIMESTAMP,
   PRIMARY KEY  (id)
) ENGINE=InnoDB
```

Now we would like to convert it into `Doctrine_Record` class. With Doctrine this is easy! Remember our test script we created in the Getting Started chapter? We're going to use that generate our models.

First we need to modify our `bootstrap.php` to use the MySQL database instead of sqlite memory:

```php
// bootstrap.php

// ...
$conn = Doctrine_Manager::connection('mysql://root:mys3cr3et@localhost/doctrine_test', 'doctrine');
// ...
```

> You can use the `$conn->createDatabase()` method to create the database if it does not already exist and the connected user has permission to create databases. Then use the above provided `CREATE TABLE` statement to create the table.

Now we need a place to store our generated classes so lets create a directory named `models` in the `doctrine_test` directory:

```
$ mkdir doctrine_test/models
```

Now we just need to add the code to our `test.php` script to generate the model classes:

```php
// test.php

// ...
Doctrine_Core::generateModelsFromDb('models', array('doctrine'), array('generateTableClasses' => true));
```

> The `generateModelsFromDb` method only requires one parameter and it is the import directory (the directory where the generated record files will be written to). The second argument is an array of database connection names to generate models for, and the third is the array of options to use for the model building.

That's it! Now there should be a file called `BaseUser.php` in your `doctrine_test/models/generated` directory. The file should look like the following:

```php
// models/generated/BaseUser.php

/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
abstract class BaseUser extends Doctrine_Record
{
  public function setTableDefinition()
  {
    $this->setTableName('user');
    $this->hasColumn('id', 'integer', 8, array('type' => 'integer', 'length' => 8, 'primary' => true, 'autoincrement' =>
    $this->hasColumn('first_name', 'string', 255, array('type' => 'string', 'length' => 255));
    $this->hasColumn('last_name', 'string', 255, array('type' => 'string', 'length' => 255));
    $this->hasColumn('username', 'string', 255, array('type' => 'string', 'length' => 255));
    $this->hasColumn('password', 'string', 255, array('type' => 'string', 'length' => 255));
    $this->hasColumn('type', 'string', 255, array('type' => 'string', 'length' => 255));
    $this->hasColumn('is_active', 'integer', 1, array('type' => 'integer', 'length' => 1, 'default' => '1'));
    $this->hasColumn('is_super_admin', 'integer', 1, array('type' => 'integer', 'length' => 1, 'default' => '0'));
    $this->hasColumn('created_at', 'timestamp', null, array('type' => 'timestamp', 'notnull' => true));
    $this->hasColumn('updated_at', 'timestamp', null, array('type' => 'timestamp', 'notnull' => true));
  }
}
```

You should also have a file called `User.php` in your `doctrine_test/models` directory. The file should look like the following:

```php
// models/User.php

/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
class User extends BaseUser
{

}
```

Doctrine will automatically generate a skeleton `Doctrine_Table` class for the model at `doctrine_test/models/UserTable.php` because we passed the option `generateTableClasses` with a value of `true`. The file should look like the following:

```php
// models/UserTable.php

/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
class UserTable extends Doctrine_Table
{

}
```

You can place custom functions inside the `User` and `UserTable` classes to customize the functionality of your models. Below are some examples:

```php
// models/User.php

// ...
class User extends BaseUser
{
    public function setPassword($password)
    {
        return $this->_set('password', md5($password));
    }
}
```

In order for the above `password` accessor overriding to work properly you must enabled the `auto_accessor_override` attribute in your `bootstrap.php` file like done below.

```php
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_AUTO_ACCESSOR_OVERRIDE, true);
```

Now when you try and set a users password it will be md5 encrypted. First we need to modify our `bootstrap.php` file to include some code for autoloading our models from the `models` directory:

```php
// bootstrap.php

// ...
Doctrine_Core::loadModels('models');
```

> The model loading is fully explained later in the Autoloading Models section of this chapter.

Now we can modify `test.php` to include some code which will test the changes we made to the `User` model:

```php
// test.php

// ...

$user = new User();
$user->username = 'jwage';
$user->password = 'changeme';

echo $user->password; // outputs md5 hash and not changeme
```

Now when you execute `test.php` from your terminal you should see the following:

```
$ php test.php
4cb9c8a8048fd02294477fcb1a41191a
```

Here is an example of some custom functions you might add to the `UserTable` class:

```php
// models/UserTable.php

// ...
class UserTable extends Doctrine_Table
{
    public function getCreatedToday()
    {
        $today = date('Y-m-d h:i:s', strtotime(date('Y-m-d')));
        return $this->createQuery('u')
            ->where('u.created_at > ?', $today)
            ->execute();
    }
}
```

In order for custom `Doctrine_Table` classes to be loaded you must enable the `autoload_table_classes` attribute in your `bootstrap.php` file like done below.

```php
// boostrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_AUTOLOAD_TABLE_CLASSES, true);
```

Now you have access to this function when you are working with the `UserTable` instance:

```php
// test.php

// ...
$usersCreatedToday = Doctrine_Core::getTable('User')->getCreatedToday();
```

Schema Files

You can alternatively manage your models with YAML schema files and generate PHP classes from them. First lets generate a YAML schema file from the existing models we already have to make things easier. Change `test.php` to have the following code inside:

```
// test.php
// ...
Doctrine_Core::generateYamlFromModels('schema.yml', 'models');
```

Execute the `test.php` script:

```
$ php test.php
```

Now you should see a file named `schema.yml` created in the root of the `doctrine_test` directory. It should look like the following:

```
---
User:
  tableName: user
  columns:
    id:
      type: integer(8)
      primary: true
      autoincrement: true
    is_active:
      type: integer(1)
      default: '1'
    is_super_admin:
      type: integer(1)
      default: '0'
    created_at:
      type: timestamp(25)
      notnull: true
    updated_at:
      type: timestamp(25)
      notnull: true
    first_name: string(255)
    last_name: string(255)
    username: string(255)
    password: string(255)
    type: string(255)
```

So now that we have a valid YAML schema file, we can now maintain our schema from here and generate the PHP classes from here. Lets create a new php script called `generate.php`. This script will re-generate everything and make sure the database is reinstantiated each time the script is called:

```
// generate.php

require_once('bootstrap.php');

Doctrine_Core::dropDatabases();
Doctrine_Core::createDatabases();
Doctrine_Core::generateModelsFromYaml('schema.yml', 'models');
Doctrine_Core::createTablesFromModels('models');
```

Now you can alter your `schema.yml` and re-generate your models by running the following command from your terminal:

```
$ php generate.php
```

Now that we have our YAML schema file setup and we can re-generate our models from the schema files lets cleanup the file a little and take advantage of some of the power of Doctrine:

```
---
User:
  actAs: [Timestampable]
  columns:
    is_active:
      type: integer(1)
      default: '1'
    is_super_admin:
      type: integer(1)
      default: '0'
    first_name: string(255)
    last_name: string(255)
    username: string(255)
    password: string(255)
    type: string(255)
```

Notice some of the changes we made:

1.) Removed the explicit `tableName` definition as it will default to user.

2.) Attached the `Timestampable` behavior.

3.) Removed `id` column as it is automatically added if no primary key is defined.

4.) Removed `updated_at` and `created_at` columns as they can be handled automatically by the `Timestampable` behavior.

Now look how much cleaner the YAML is and is because we take advantage of defaults and utilize core behaviors it is much less work we have to do ourselves.

Now re-generate your models from the YAML schema file:

```
$ php generate.php
```

You can learn more about YAML Schema Files in its dedicated chapter.

Manually Writing Models

You can optionally skip all the convenience methods and write your models manually using nothing but your own PHP code. You can learn all about the models syntax in the Defining Models chapter.

Autoloading Models

Doctrine offers two ways of loading models. We have conservative(lazy) loading, and aggressive loading. Conservative loading will not require the PHP file initially, instead it will cache the path to the class name and this path is then used in the `Doctrine_Core::modelsAutoload()`.

To use Doctrine model loading you need to register the model autoloader in your bootstrap:

```
// bootstrap.php
// ...
spl_autoload_register(array('Doctrine_Core', 'modelsAutoload'));
```

Below are some examples using the both types of model loading.

Conservative

Conservative model loading is going to be the ideal model loading method for a production environment. This method will lazy load all of the models instead of loading them all when model loading is executed.

Conservative model loading requires that each file contain only one class, and the file must be named after the class. For example, if you have a class named `User`, it must be contained in a file named `User.php`.

To use conservative model loading we need to set the model loading attribute to be conservative:

```
$manager->setAttribute(Doctrine_Core::ATTR_MODEL_LOADING, Doctrine_Core::MODEL_LOADING_CONSERVATIVE);
```

> We already made this change in an earlier step in the `bootstrap.php` file so you don't need to make this change again.

When we use the `Doctrine_Core::loadModels()` functionality all found classes will be cached internally so the autoloader can require them later.

```
Doctrine_Core::loadModels('models');
```

Now when we instantiate a new class, for example a `User` class, the autoloader will be triggered and the class is required.

```
// triggers call to Doctrine_Core::modelsAutoload() and the class is included
$user = new User();
```

Instantiating the class above triggers a call to `Doctrine_Core::modelsAutoload()` and the class that was found in the call to `Doctrine_Core::loadModels()` will be required and made available.

> Conservative model loading is recommended in most cases, specifically for production environments as you do not want to require every single model class even when it is not needed as this is unnecessary overhead. You only want to require it when it is needed.

Aggressive

Aggressive model loading is the default model loading method and is very simple, it will look for all files with a `.php` extension and will include it. Doctrine can not satisfy any inheritance and if your models extend another model, it cannot include them in the correct order so it is up to you to make sure all dependencies are satisfied in each class.

With aggressive model loading you can have multiple classes per file and the file name is not required to be related to the name of the class inside of the file.

The downside of aggressive model loading is that every php file is included in every request, so if you have lots of models it is recommended you use conservative model loading.

To use aggressive model loading we need to set the model loading attribute to be aggressive:

```
$manager->setAttribute(Doctrine_Core::ATTR_MODEL_LOADING, Doctrine_Core::MODEL_LOADING_AGGRESSIVE);
```

> Aggressive is the default of the model loading attribute so explicitly setting it is not necessary if you wish to use it.

When we use the `Doctrine_Core::loadModels()` functionality all the classes found will be included right away:

```
Doctrine_Core::loadModels('/path/to/models');
```

Conclusion

This chapter is probably the most intense chapter so far but it is a good one. We learned a little about how to use models, how to generate models from existing databases, how to write our own models, and how to maintain our models as YAML schema files. We also modified our Doctrine test environment to implement some functionality for loading models from our models directory.

This topic of Doctrine models is so large that it warranted the chapters being split in to three pieces to make it easier on the developer to absorb all the information. In the next chapter we will really get in to the API we use to define our models.

Defining Models

As we mentioned before, at the lowest level in Doctrine your schema is represented by a set of php classes that map the schema meta data for your database tables.

In this chapter we will explain in detail how you can map your schema information using php code.

Columns

One problem with database compatibility is that many databases differ in their behavior of how the result set of a query is returned. MySQL leaves the field names unchanged, which means if you issue a query of the form `"SELECT myField FROM ..."` then the result set will contain the field `myField`.

Unfortunately, this is just the way MySQL and some other databases do it. Postgres for example returns all field names in lowercase whilst Oracle returns all field names in uppercase. "So what? In what way does this influence me when using Doctrine?", you may ask. Fortunately, you don't have to bother about that issue at all.

Doctrine takes care of this problem transparently. That means if you define a derived Record class and define a field called `myField` you will always access it through `$record->myField` (or `$record['myField']`, whatever you prefer) no matter whether you're using MySQL or Postgres or Oracle etc.

In short: You can name your fields however you want, using under_scores, camelCase or whatever you prefer.

> In Doctrine columns and column aliases are case sensitive. So when you are using columns in your DQL queries, the column/field names must match the case in your model definition.

Column Lengths

In Doctrine column length is an integer that specifies the column length. Some column types depend not only the given portable type but also on the given length. For example type string with length 1000 will be translated into native type TEXT on mysql.

The length is different depending on the type of column you are using:

- **integer** – Length is the the number of bytes the integer occupies.
- **string** – Number of the characters allowed in the string.
- **float/decimal** – Total number of characters allowed excluding the decimal.
- **enum** – If using native enum length does not apply but if using emulated enums then it is just the string length of the column value.

Column Aliases

Doctrine offers a way of setting column aliases. This can be very useful when you want to keep the application logic separate from the database logic. For example if you want to change the name of the database field all you need to change at your application is the column definition.

```php
// models/Book.php
class Book extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('bookTitle as title', 'string');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
Book:
  columns:
    bookTitle:
      name: bookTitle as title
      type: string
```

Now the column in the database is named bookTitle but you can access the property on your objects using title.

```php
// test.php

// ...
$book = new Book();
$book->title = 'Some book';
$book->save();
```

Default values

Doctrine supports default values for all data types. When default value is attached to a record column this means two things. First this value is attached to every newly created Record and when Doctrine creates your database tables it includes the default value in the create table statement.

```php
// models/generated/BaseUser.php

class User extends BaseUser
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 255, array('default' => 'default username'));

        // ...
    }

    // ...
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
# ...
  columns:
    username:
      type: string(255)
      default: default username
# ...
```

Now when you print the name on a brand new User record it will print the default value:

```php
// test.php

// ...
$user = new User();
echo $user->username; // default username
```

Data types

Introduction

All DBMS provide multiple choice of data types for the information that can be stored in their database table fields. However, the set of data types made available varies from DBMS to DBMS.

To simplify the interface with the DBMS supported by Doctrine, a base set of data types was defined. Applications may access them independently of the underlying DBMS.

The Doctrine applications programming interface takes care of mapping data types when managing database options. It is also able to convert that is sent to and received from the underlying DBMS using the respective driver.

The following data type examples should be used with Doctrine's `createTable()` method. The example array at the end of the data types section may be used with `createTable()` to create a portable table on the DBMS of choice (please refer to the main Doctrine documentation to find out what DBMS back ends are properly supported). It should also be noted that the following examples do not cover the creation and maintenance of indices, this chapter is only concerned with data types and the proper usage thereof.

It should be noted that the length of the column affects in database level type as well as application level validated length (the length that is validated with Doctrine validators).

Example 1. Column named 'content' with type 'string' and length 3000 results in database type 'TEXT' of which has database level length of 4000. However when the record is validated it is only allowed to have 'content' –column with maximum length of 3000.

Example 2. Column with type 'integer' and length 1 results in 'TINYINT' on many databases.

In general Doctrine is smart enough to know which integer/string type to use depending on the specified length.

Type modifiers

Within the Doctrine API there are a few modifiers that have been designed to aid in optimal table design. These are:

- The notnull modifiers
- The length modifiers
- The default modifiers
- unsigned modifiers for some field definitions, although not all DBMS's support this modifier for integer field types.
- collation modifiers (not supported by all drivers)
- fixed length modifiers for some field definitions.

Building upon the above, we can say that the modifiers alter the field definition to create more specific field types for specific usage scenarios. The notnull modifier will be used in the following way to set the default DBMS NOT NULL Flag on the field to true or false, depending on the DBMS's definition of the field value: In PostgreSQL the "NOT NULL" definition will be set to "NOT NULL", whilst in MySQL (for example) the "NULL" option will be set to "NO". In order to define a "NOT NULL" field type, we simply add an extra parameter to our definition array (See the examples in the following section)

```
'sometime' = array(
    'type'    => 'time',
    'default' => '12:34:05',
    'notnull' => true,
),
```

Using the above example, we can also explore the default field operator. Default is set in the same way as the notnull operator to set a default value for the field. This value may be set in any character set that the DBMS supports for text fields, and any other valid data for the field's data type. In the above example, we have specified a valid time for the "Time" data type, '12:34:05'. Remember that when setting default dates and times, as well as datetimes, you should research and stay within the epoch of your chosen DBMS, otherwise you will encounter difficult to diagnose errors!

```
'sometext' = array(
    'type'   => 'string',
    'length' => 12,
),
```

The above example will create a character varying field of length 12 characters in the database table. If the length definition is left out, Doctrine will create a length of the maximum allowable length for the data type specified, which may create a problem with some field types and indexing. Best practice is to define lengths for all or most of your fields.

Boolean

The boolean data type represents only two values that can be either 1 or 0. Do not assume that these data types are stored as integers because some DBMS drivers may implement this type with single character text fields for a matter of efficiency. Ternary logic is possible by using null as the third possible value that may be assigned to fields of this type.

> The next several examples are not meant for you to use and give them a try. They are simply for demonstrating purposes to show you how to use the different Doctrine data types using PHP code or YAML schema files.

```
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('booltest', 'boolean');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
Test:
  columns:
    booltest: boolean
```

Integer

The integer type is the same as integer type in PHP. It may store integer values as large as each DBMS may handle.

Fields of this type may be created optionally as unsigned integers but not all DBMS support it. Therefore, such option may be ignored. Truly portable applications should not rely on the availability of this option.

The integer type maps to different database type depending on the column length.

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('integertest', 'integer', 4, array(
                'unsigned' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    integertest:
      type: integer(4)
      unsigned: true
```

Float

The float data type may store floating point decimal numbers. This data type is suitable for representing numbers withina large scale range that do not require high accuracy. The scale and the precision limits of the values that may be stored in a database depends on the DBMS that it is used.

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('floattest', 'float');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    floattest: float
```

Decimal

The decimal data type may store fixed precision decimal numbers. This data type is suitable for representing numbers that require high precision and accuracy.

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('decimaltest', 'decimal');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    decimaltest: decimal
```

You can specify the length of the decimal just like you would set the `length` of any other column and you can specify the `scale` as an option in the third argument:

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('decimaltest', 'decimal', 18, array(
                'scale' => 2
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    decimaltest:
      type: decimal(18)
      scale: 2
```

String

The text data type is available with two options for the length: one that is explicitly length limited and another of undefined length that should be as large as the database allows.

The length limited option is the most recommended for efficiency reasons. The undefined length option allows very large fields but may prevent the use of indexes, nullability and may not allow sorting on fields of its type.

The fields of this type should be able to handle 8 bit characters. Drivers take care of DBMS specific escaping of characters of special meaning with the values of the strings to be converted to this type.

By default Doctrine will use variable length character types. If fixed length types should be used can be controlled via the fixed modifier.

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('stringtest', 'string', 200, array(
                'fixed' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    stringtest:
      type: string(200)
      fixed: true
```

Array

This is the same as the 'array' type in PHP.

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('arraytest', 'array', 10000);
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    arraytest: array(10000)
```

Object

Doctrine supports objects as column types. Basically you can set an object to a field and Doctrine handles automatically the serialization / unserialization of that object.

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('objecttest', 'object');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    objecttest: object
```

> The array and object types simply serialize the data when persisting to the database and unserialize the data when pulling from the database.

Blob

Blob (Binary Large OBject) data type is meant to store data of undefined length that may be too large to store in text fields, like data that is usually stored in fi les.

Blob fields are usually not meant to be used as parameters of query search clause (`WHERE`) unless the underlying DBMS supports a feature usually known as "full text search".

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('blobtest', 'blob');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    blobtest: blob
```

Clob

Clob (Character Large OBject) data type is meant to store data of undefined length that may be too large to store in text fields, like data that is usually stored in fi les.

Clob fields are meant to store only data made of printable ASCII characters whereas blob fields are meant to store all types of data.

Clob fields are usually not meant to be used as parameters of query search clause (`WHERE`) unless the underlying DBMS supports a feature usually known as "full text search".

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('clobtest', 'clob');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    clobtest: clob
```

Timestamp

The timestamp data type is a mere combination of the date and the time of the day data types. The representation of values of the time stamp type is accomplished by joining the date and time string values in a single string joined by a space. Therefore, the format template is `YYYY-MM-DD HH:MI:SS`. The represented values obey the same rules and ranges described for the date and time data types.

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('timestamptest', 'timestamp');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    timestamptest: timestamp
```

Time

The time data type may represent the time of a given moment of the day. DBMS independent representation of the time of the day is also accomplished by using text strings formatted according to the ISO-8601 standard.

The format defined by the ISO-8601 standard for the time of the day is HH:MI:SS where HH is the number of hour the day from 00 to 23 and MI and SS are respectively the number of the minute and of the second from 00 to 59. Hours, minutes and seconds numbered below 10 should be padded on the left with 0.

Some DBMS have native support for time of the day formats, but for others the DBMS driver may have to represent them as integers or text values. In any case, it is always possible to make comparisons between time values as well sort query results by fields of this type.

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('timetest', 'time');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    timetest: time
```

Date

The date data type may represent dates with year, month and day. DBMS independent representation of dates is accomplished by using text strings formatted according to the IS0-8601 standard.

The format defined by the ISO-8601 standard for dates is YYYY-MM-DD where YYYY is the number of the year (Gregorian calendar), MM is the number of the month from 01 to 12 and DD is the number of the day from 01 to 31. Months or days numbered below 10 should be padded on the left with 0.

Some DBMS have native support for date formats, but for others the DBMS driver may have to represent them as integers or text values. In any case, it is always possible to make comparisons between date values as well sort query results by fields of this type.

```php
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('datetest', 'date');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Test:
  columns:
    datetest: date
```

Enum

Doctrine has a unified enum type. The possible values for the column can be specified on the column definition with `Doctrine_Record::hasColumn()`

> If you wish to use native enum types for your DBMS if it supports it then you must set the following attribute:

```php
$conn->setAttribute(Doctrine_Core::ATTR_USE_NATIVE_ENUM, true);
```

Here is an example of how to specify the enum values:

```
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('enumtest', 'enum', null,
             array('values' => array('php', 'java', 'python'))
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
Test:
  columns:
    enumtest:
      type: enum
      values: [php, java, python]
```

Gzip

Gzip datatype is the same as string except that its automatically compressed when persisted and uncompressed when fetched. This datatype can be useful when storing data with a large compressibility ratio, such as bitmap images.

```
class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('gziptest', 'gzip');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
Test:
  columns:
    gziptest: gzip
```

> The family of php functions for compressing are used internally for compressing and uncompressing the contents of the gzip column type.

Examples

Consider the following definition:

```php
class Example extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('id', 'string', 32, array(
                'type' => 'string',
                'fixed' => 1,
                'primary' => true,
                'length' => '32'
            )
        );

        $this->hasColumn('someint', 'integer', 10, array(
                'type' => 'integer',
                'unsigned' => true,
                'length' => '10'
            )
        );

        $this->hasColumn('sometime', 'time', 25, array(
                'type' => 'time',
                'default' => '12:34:05',
                'notnull' => true,
                'length' => '25'
            )
        );

        $this->hasColumn('sometext', 'string', 12, array(
                'type' => 'string',
                'length' => '12'
            )
        );

        $this->hasColumn('somedate', 'date', 25, array(
                'type' => 'date',
                'length' => '25'
            )
        );

        $this->hasColumn('sometimestamp', 'timestamp', 25, array(
                'type' => 'timestamp',
                'length' => '25'
            )
        );

        $this->hasColumn('someboolean', 'boolean', 25, array(
                'type' => 'boolean',
                'length' => '25'
            )
        );

        $this->hasColumn('somedecimal', 'decimal', 18, array(
                'type' => 'decimal',
                'length' => '18'
            )
        );

        $this->hasColumn('somefloat', 'float', 2147483647, array(
                'type' => 'float',
                'length' => '2147483647'
            )
        );

        $this->hasColumn('someclob', 'clob', 2147483647, array(
                'type' => 'clob',
                'length' => '2147483647'
            )
        );

        $this->hasColumn('someblob', 'blob', 2147483647, array(
                'type' => 'blob',
                'length' => '2147483647'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
Example:
  tableName: example
  columns:
    id:
      type: string(32)
      fixed: true
      primary: true
    someint:
      type: integer(10)
      unsigned: true
    sometime:
      type: time(25)
      default: '12:34:05'
      notnull: true
    sometext: string(12)
    somedate: date(25)
    sometimestamp: timestamp(25)
    someboolean: boolean(25)
    somedecimal: decimal(18)
    somefloat: float(2147483647)
    someclob: clob(2147483647)
    someblob: blob(2147483647)
```

The above example will create the following database table in Pgsql:

Column	Type
id	character(32)
someint	integer
sometime	time without time zone
sometext	character or varying(12)
somedate	date
sometimestamp	timestamp without time zone
someboolean	boolean
somedecimal	numeric(18,2)
somefloat	double precision
someclob	text
someblob	bytea

The schema will create the following database table in Mysql:

Field	Type
id	char(32)
someint	integer
sometime	time
sometext	varchar(12)
somedate	date
sometimestamp	timestamp
someboolean	tinyint(1)
somedecimal	decimal(18,2)
somefloat	double
someclob	longtext
someblob	longblob

Relationships

Introduction

In Doctrine all record relations are being set with `Doctrine_Record::hasMany`, `Doctrine_Record::hasOne` methods. Doctrine supports almost all kinds of database relations from simple one-to-one foreign key relations to join table self-referencing relations.

Unlike the column definitions the `Doctrine_Record::hasMany` and `Doctrine_Record::hasOne` methods are placed within a method called setUp(). Both methods take two arguments: the first argument is a string containing the name of the class and optional alias, the second argument is an array consisting of relation options. The option array contains the following keys:

Name	Optional	Description

local	No	The local field of the relation. Local field is the linked field in the defining class.
foreign	No	The foreign field of the relation. Foreign field is the linked field in the linked class.
refClass	Yes	The name of the association class. This is only needed for many-to-many associations.
owningSide	Yes	Set to boolean true to indicate the owning side of the relation. The owning side is the side that owns the foreign key. There can only be one owning side in an association between two classes. Note that this option is required if Doctrine can't guess the owning side or it's guess is wrong. An example where this is the case is when both 'local' and 'foreign' are part of the identifier (primary key). It never hurts to specify the owning side in this way.
onDelete	Yes	The onDelete integrity action that is applied on the foreign key constraint when the tables are created by Doctrine.
onUpdate	Yes	The onUpdate integrity action that is applied on the foreign key constraint when the tables are created by Doctrine.
cascade	Yes	Specify application level cascading operations. Currently only delete is supported

So lets take our first example, say we have two classes Forum_Board and Forum_Thread. Here Forum_Board has many Forum_Threads, hence their relation is one-to-many. We don't want to write Forum_ when accessing relations, so we use relation aliases and use the alias Threads.

First lets take a look at the Forum_Board class. It has three columns: name, description and since we didn't specify any primary key, Doctrine auto-creates an id column for it.

We define the relation to the Forum_Thread class by using the hasMany() method. Here the local field is the primary key of the board class whereas the foreign field is the board_id field of the Forum_Thread class.

```
// models/Forum_Board.php

class Forum_Board extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 100);
        $this->hasColumn('description', 'string', 5000);
    }

    public function setUp()
    {
        $this->hasMany('Forum_Thread as Threads', array(
                'local' => 'id',
                'foreign' => 'board_id'
            )
        );
    }
}
```

> Notice the as keyword being used above. This means that the Forum_Board has a many relationship defined to Forum_Thread but is aliased as Threads.

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Forum_Board:
  columns:
    name: string(100)
    description: string(5000)
```

Then lets have a peek at the Forum_Thread class. The columns here are irrelevant, but pay attention to how we define the relation. Since each Thread can have only one Board we are using the hasOne() method. Also notice how we once again use aliases and how the local column here is board_id while the foreign column is the id column.

```php
// models/Forum_Thread.php

class Forum_Thread extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer');
        $this->hasColumn('board_id', 'integer');
        $this->hasColumn('title', 'string', 200);
        $this->hasColumn('updated', 'integer', 10);
        $this->hasColumn('closed', 'integer', 1);
    }

    public function setUp()
    {
        $this->hasOne('Forum_Board as Board', array(
                'local' => 'board_id',
                'foreign' => 'id'
            )
        );

        $this->hasOne('User', array(
                'local' => 'user_id',
                'foreign' => 'id'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
Forum_Thread:
  columns:
    user_id: integer
    board_id: integer
    title: string(200)
    updated: integer(10)
    closed: integer(1)
  relations:
    User:
      local: user_id
      foreign: id
      foreignAlias: Threads
    Board:
      class: Forum_Board
      local: board_id
      foreign: id
      foreignAlias: Threads
```

Now we can start using these classes. The same accessors that you've already used for properties are all available for relations.

First lets create a new board:

```php
// test.php

// ...
$board = new Forum_Board();
$board->name = 'Some board';
```

Now lets create a new thread under the board:

```php
// test.php

// ...
$board->Threads[0]->title = 'new thread 1';
$board->Threads[1]->title = 'new thread 2';
```

Each **Thread** needs to be associated to a user so lets create a new **User** and associate it to each **Thread**:

```php
$user = new User();
$user->username = 'jwage';
$board->Threads[0]->User = $user;
$board->Threads[1]->User = $user;
```

Now we can save all the changes with one call. It will save the new board as well as its threads:

```php
// test.php

// ...
$board->save();
```

Lets do a little inspecting and see the data structure that is created when you use the code from above. Add some code to `test.php` to output an array of the object graph we've just populated:

```
print_r($board->toArray(true));
```

> The `Doctrine_Record::toArray()` takes all the data of a `Doctrine_Record` instance and converts it to an array so you can easily inspect the data of a record. It accepts an argument named `$deep` telling it whether or not to include relationships. In this example we have specified {[true]} because we want to include the `Threads` data.

Now when you execute `test.php` with PHP from your terminal you should see the following:

```
$ php test.php
Array
(
    [id] => 2
    [name] => Some board
    [description] =>
    [Threads] => Array
        (
            [0] => Array
                (
                    [id] => 3
                    [user_id] => 1
                    [board_id] => 2
                    [title] => new thread 1
                    [updated] =>
                    [closed] =>
                    [User] => Array
                        (
                            [id] => 1
                            [is_active] => 1
                            [is_super_admin] => 0
                            [first_name] =>
                            [last_name] =>
                            [username] => jwage
                            [password] =>
                            [type] =>
                            [created_at] => 2009-01-20 16:41:57
                            [updated_at] => 2009-01-20 16:41:57
                        )

                )

            [1] => Array
                (
                    [id] => 4
                    [user_id] => 1
                    [board_id] => 2
                    [title] => new thread 2
                    [updated] =>
                    [closed] =>
                    [User] => Array
                        (
                            [id] => 1
                            [is_active] => 1
                            [is_super_admin] => 0
                            [first_name] =>
                            [last_name] =>
                            [username] => jwage
                            [password] =>
                            [type] =>
                            [created_at] => 2009-01-20 16:41:57
                            [updated_at] => 2009-01-20 16:41:57
                        )

                )

        )
)
```

> Notice how the auto increment primary key and foreign keys are automatically set by Doctrine internally. You don't have to worry about the setting of primary keys and foreign keys at all!

Foreign Key Associations

One to One

One-to-one relations are probably the most basic relations. In the following example we have two classes, `User` and `Email` with their relation being one-to-one.

First lets take a look at the `Email` class. Since we are binding a one-to-one relationship we are using the `hasOne()` method. Notice how we define the foreign key column (`user_id`) in the `Email` class. This is due to a fact that `Email` is owned by the `User` class and not the other way around. In fact you should always follow this convention – always place the foreign key in the owned class.

The recommended naming convention for foreign key columns is: `[tableName]_[primaryKey]`. As here the foreign table is 'user' and its primary key is 'id' we have named the foreign key column as 'user_id'.

```
// models/Email.php

class Email extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer');
        $this->hasColumn('address', 'string', 150);
    }

    public function setUp()
    {
        $this->hasOne('User', array(
                'local' => 'user_id',
                'foreign' => 'id'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Email:
  columns:
    user_id: integer
    address: string(150)
  relations:
    User:
      local: user_id
      foreign: id
      foreignType: one
```

> When using YAML schema files it is not required to specify the relationship on the opposite end(`User`) because the relationship is automatically flipped and added for you. The relationship will be named the name of the class. So in this case the relationship on the `User` side will be called `Email` and will be `many`. If you wish to customize this you can use the `foreignAlias` and `foreignType` options.

The `Email` class is very similar to the `User` class. Notice how the local and foreign columns are switched in the `hasOne()` definition compared to the definition of the `Email` class.

```
// models/User.php

class User extends BaseUser
{
    public function setUp()
    {
        parent::setUp();

        $this->hasOne('Email', array(
                'local' => 'id',
                'foreign' => 'user_id'
            )
        );
    }
}
```

> Notice how we override the `setUp()` method and call `parent::setUp()`. This is because the `BaseUser` class which is generated from YAML or from an existing database contains the main `setUp()` method and we override it in the `User` class to add an additional relationship.

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
User:
# ...
  relations:
    # ...
    Email:
      local: id
      foreign: user_id
```

One to Many and Many to One

One-to-Many and Many-to-One relations are very similar to One-to-One relations. The recommended conventions you came in terms with in the previous chapter also apply to one-to-many and many-to-one relations.

In the following example we have two classes: `User` and `Phonenumber`. We define their relation as one-to-many (a user can have many phonenumbers). Here once again the `Phonenumber` is clearly owned by the `User` so we place the foreign key in the `Phonenumber` class.

```
// models/User.php

class User extends BaseUser
{
    public function setUp()
    {
        parent::setUp();

        // ...

        $this->hasMany('Phonenumber as Phonenumbers', array(
                'local' => 'id',
                'foreign' => 'user_id'
            )
        );
    }
}

// models/Phonenumber.php

class Phonenumber extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer');
        $this->hasColumn('phonenumber', 'string', 50);
    }

    public function setUp()
    {
        $this->hasOne('User', array(
                'local' => 'user_id',
                'foreign' => 'id'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
User:
# ...
  relations:
    # ...
    Phonenumbers:
      type: many
      class: Phonenumber
      local: id
      foreign: user_id

Phonenumber:
  columns:
    user_id: integer
    phonenumber: string(50)
  relations:
    User:
      local: user_id
      foreign: id
```

Tree Structure

A tree structure is a self-referencing foreign key relation. The following definition is also called Adjacency List implementation in terms of hierarchical data concepts.

```php
// models/Task.php

class Task extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 100);
        $this->hasColumn('parent_id', 'integer');
    }

    public function setUp()
    {
        $this->hasOne('Task as Parent', array(
                'local' => 'parent_id',
                'foreign' => 'id'
            )
        );

        $this->hasMany('Task as Subtasks', array(
                'local' => 'id',
                'foreign' => 'parent_id'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
Task:
  columns:
    name: string(100)
    parent_id: integer
  relations:
    Parent:
      class: Task
      local: parent_id
      foreign: id
      foreignAlias: Subtasks
```

> The above implementation is purely an example and is not the most efficient way to store and retrieve hierarchical data. Check the `NestedSet` behavior included in Doctrine for the recommended way to deal with hierarchical data.

Join Table Associations

Many to Many

If you are coming from relational database background it may be familiar to you how many-to-many associations are handled: an additional association table is needed.

In many-to-many relations the relation between the two components is always an aggregate relation and the association table is owned by both ends. For example in the case of users and groups: when a user is being deleted, the groups he/she belongs to are not being deleted. However, the associations between this user and the groups he/she belongs to are instead being deleted. This removes the relation between the user and the groups he/she belonged to, but does not remove the user nor the groups.

Sometimes you may not want that association table rows are being deleted when user / group is being deleted. You can override this behavior by setting the relations to association component (in this case `Groupuser`) explicitly.

In the following example we have Groups and Users of which relation is defined as many-to-many. In this case we also need to define an additional class called `Groupuser`.

```php
class User extends BaseUser
{
    public function setUp()
    {
        parent::setUp();

        // ...

        $this->hasMany('Group as Groups', array(
                'local' => 'user_id',
                'foreign' => 'group_id',
                'refClass' => 'UserGroup'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
User:
# ...
  relations:
    # ...
    Groups:
      class: Group
      local: user_id
      foreign: group_id
      refClass: UserGroup
```

> The above `refClass` option is required when setting up many-to-many relationships.

```php
// models/Group.php

class Group extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->setTableName('groups');
        $this->hasColumn('name', 'string', 30);
    }

    public function setUp()
    {
        $this->hasMany('User as Users', array(
                'local' => 'group_id',
                'foreign' => 'user_id',
                'refClass' => 'UserGroup'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
Group:
  tableName: groups
  columns:
    name: string(30)
  relations:
    Users:
      class: User
      local: group_id
      foreign: user_id
      refClass: UserGroup
```

> Please note that `group` is a reserved keyword so that is why we renamed the table to `groups` using the `setTableName` method. The other option is to turn on identifier quoting using the `Doctrine_Core::ATTR_QUOTE_IDENTIFIER` attribute so that the reserved word is escaped with quotes.

```php
$manager->setAttribute(Doctrine_Core::Doctrine_Core::ATTR_QUOTE_IDENTIFIER, true);
```

```php
// models/UserGroup.php

class UserGroup extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer', null, array(
                'primary' => true
            )
        );

        $this->hasColumn('group_id', 'integer', null, array(
                'primary' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
UserGroup:
  columns:
    user_id:
      type: integer
      primary: true
    group_id:
      type: integer
      primary: true
```

Notice how the relationship is bi-directional. Both `User` has many `Group` and `Group` has many `User`. This is required by Doctrine in order for many-to-many relationships to fully work.

Now lets play around with the new models and create a user and assign it some groups. First create a new `User` instance:

```
// test.php

// ...
$user = new User();
```

Now add two new groups to the `User`:

```
// test.php

// ...
$user->Groups[0]->name = 'First Group';

$user->Groups[1]->name = 'Second Group';
```

Now you can save the groups to the database:

```
// test.php

// ...
$user->save();
```

Now you can delete the associations between user and groups it belongs to:

```
// test.php

// ...
$user->UserGroup->delete();

$groups = new Doctrine_Collection(Doctrine_Core::getTable('Group'));

$groups[0]->name = 'Third Group';

$groups[1]->name = 'Fourth Group';

$user->Groups[2] = $groups[0];
// $user will now have 3 groups

$user->Groups = $groups;
// $user will now have two groups 'Third Group' and 'Fourth Group'

$user->save();
```

Now if we inspect the `$user` object data with the `Doctrine_Record::toArray()`:

```
// test.php

// ...
print_r($user->toArray(true));
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [id] => 1
    [is_active] => 1
    [is_super_admin] => 0
    [first_name] =>
    [last_name] =>
    [username] => default username
    [password] =>
    [type] =>
    [created_at] => 2009-01-20 16:48:57
    [updated_at] => 2009-01-20 16:48:57
    [Groups] => Array
        (
            [0] => Array
                (
                    [id] => 3
                    [name] => Third Group
                )

            [1] => Array
                (
                    [id] => 4
                    [name] => Fourth Group
                )

        )

    [UserGroup] => Array
        (
        )

)
```

Self Referencing (Nest Relations)

Non-Equal Nest Relations

```php
// models/User.php

class User extends BaseUser
{
    public function setUp()
    {
        parent::setUp();

        // ...

        $this->hasMany('User as Parents', array(
                'local'    => 'child_id',
                'foreign'  => 'parent_id',
                'refClass' => 'UserReference'
            )
        );

        $this->hasMany('User as Children', array(
                'local'    => 'parent_id',
                'foreign'  => 'child_id',
                'refClass' => 'UserReference'
            )
        );
    }
}

// models/UserReference.php

class UserReference extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('parent_id', 'integer', null, array(
                'primary' => true
            )
        );

        $this->hasColumn('child_id', 'integer', null, array(
                'primary' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
# ...
  relations:
    # ...
    Parents:
      class: User
      local: child_id
      foreign: parent_id
      refClass: UserReference
      foreignAlias: Children

UserReference:
  columns:
    parent_id:
      type: integer
      primary: true
    child_id:
      type: integer
      primary: true
```

Equal Nest Relations

Equal nest relations are perfectly suitable for expressing relations where a class references to itself and the columns within the reference class are equal.

This means that when fetching related records it doesn't matter which column in the reference class has the primary key value of the main class.

The previous clause maybe hard to understand so lets take an example. We define a class called User which can have many friends. Notice here how we use the 'equal' option.

```php
// models/User.php

class User extends BaseUser
{
    public function setUp()
    {
        parent::setUp();

        // ...

        $this->hasMany('User as Friends', array(
                'local'    => 'user1',
                'foreign'  => 'user2',
                'refClass' => 'FriendReference',
                'equal'    => true,
            )
        );
    }
}

// models/FriendReference.php

class FriendReference extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user1', 'integer', null, array(
                'primary' => true
            )
        );

        $this->hasColumn('user2', 'integer', null, array(
                'primary' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
# ...
  relations:
    # ...
    Friends:
      class: User
      local: user1
      foreign: user2
      refClass: FriendReference
      equal: true

FriendReference:
  columns:
    user1:
      type: integer
      primary: true
    user2:
      type: integer
      primary: true
```

Now lets define 4 users: Jack Daniels, John Brandy, Mikko Koskenkorva and Stefan Beer with Jack Daniels and John Brandy being buddies and Mikko Koskenkorva being the friend of all of them.

```php
// test.php

// ...
$daniels = new User();
$daniels->username = 'Jack Daniels';

$brandy = new User();
$brandy->username = 'John Brandy';

$koskenkorva = new User();
$koskenkorva->username = 'Mikko Koskenkorva';

$beer = new User();
$beer->username = 'Stefan Beer';

$daniels->Friends[0] = $brandy;

$koskenkorva->Friends[0] = $daniels;
$koskenkorva->Friends[1] = $brandy;
$koskenkorva->Friends[2] = $beer;

$conn->flush();
```

> Calling `Doctrine_Connection::flush()` will trigger an operation that saves all unsaved objects and wraps it in a single transaction.

Now if we access for example the friends of Stefan Beer it would return one user 'Mikko Koskenkorva':

```php
// test.php

// ...
$beer->free();
unset($beer);
$user = Doctrine_Core::getTable('User')->findOneByUsername('Stefan Beer');

print_r($user->Friends->toArray());
```

Now when you execute `test.php` you will see the following:

```
$ php test.php
Array
(
    [0] => Array
        (
            [id] => 4
            [is_active] => 1
            [is_super_admin] => 0
            [first_name] =>
            [last_name] =>
            [username] => Mikko Koskenkorva
            [password] =>
            [type] =>
            [created_at] => 2009-01-20 16:53:13
            [updated_at] => 2009-01-20 16:53:13
        )

)
```

Foreign Key Constraints

Introduction

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. In other words foreign key constraints maintain the referential integrity between two related tables.

Say you have the product table with the following definition:

```
// models/Product.php

class Product extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string');
        $this->hasColumn('price', 'decimal', 18);
        $this->hasColumn('discounted_price', 'decimal', 18);
    }

    public function setUp()
    {
        $this->hasMany('Order as Orders', array(
                'local' => 'id',
                'foreign' => 'product_id'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Product:
  columns:
    name:
      type: string
    price:
      type: decimal(18)
    discounted_price:
      type: decimal(18)
  relations:
    Orders:
      class: Order
      local: id
      foreign: product_id
```

Let's also assume you have a table storing orders of those products. We want to ensure that the order table only contains orders of products that actually exist. So we define a foreign key constraint in the orders table that references the products table:

```
// models/Order.php

class Order extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->setTableName('orders');
        $this->hasColumn('product_id', 'integer');
        $this->hasColumn('quantity', 'integer');
    }

    public function setUp()
    {
        $this->hasOne('Product', array(
                'local' => 'product_id',
                'foreign' => 'id'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Order:
  tableName: orders
  columns:
    product_id: integer
    quantity: integer
  relations:
    Product:
      local: product_id
      foreign: id
```

> Foreign key columns are automatically indexed by Doctrine to ensure optimal performance when issuing queries involving the foreign key.

When exported the class `Order` would execute the following SQL:

```
CREATE TABLE orders (
    id integer PRIMARY KEY auto_increment,
    product_id integer REFERENCES products (id),
    quantity integer,
    INDEX product_id_idx (product_id)
)
```

Now it is impossible to create `orders` with a `product_id` that does not appear in the `product` table.

We say that in this situation the orders table is the referencing table and the products table is the referenced table. Similarly, there are referencing and referenced columns.

Foreign Key Names

When you define a relationship in Doctrine, when the foreign key is created in the database for you Doctrine will try to create a foreign key name for you. Sometimes though, this name may not be something you want so you can customize the name to use with the `foreignKeyName` option to your relationship setup.

```
// models/Order.php

class Order extends Doctrine_Record
{
    // ...

    public function setUp()
    {
        $this->hasOne('Product', array(
                'local' => 'product_id',
                'foreign' => 'id',
                'foreignKeyName' => 'product_id_fk'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Order:
  # ...
  relations:
    Product:
      local: product_id
      foreign: id
      foreignKeyName: product_id_fk
```

Integrity Actions

CASCADE

Delete or update the row from the parent table and automatically delete or update the matching rows in the child table. Both `ON DELETE CASCADE` and `ON UPDATE CASCADE` are supported. Between two tables, you should not define several `ON UPDATE CASCADE` clauses that act on the same column in the parent table or in the child table.

SET NULL

Delete or update the row from the parent table and set the foreign key column or columns in the child table to `NULL`. This is valid only if the foreign key columns do not have the `NOT NULL` qualifier specified. Both `ON DELETE SET NULL` and `ON UPDATE SET NULL` clauses are supported.

NO ACTION

In standard SQL, `NO ACTION` means no action in the sense that an attempt to delete or update a primary key value is not allowed to proceed if there is a related foreign key value in the referenced table.

RESTRICT

Rejects the delete or update operation for the parent table. `NO ACTION` and `RESTRICT` are the same as omitting the `ON DELETE` or `ON UPDATE` clause.

SET DEFAULT

In the following example we define two classes, `User` and `Phonenumber` with their relation being one-to-many. We also add a foreign key constraint with onDelete cascade action. This means that every time a `user` is being deleted its associated `phonenumbers` will also be deleted.

> The integrity constraints listed above are case sensitive and must be in upper case when being defined in your schema. Below is an example where the database delete cascading is used.

```
class Phonenumber extends Doctrine_Record
{
    // ...

    public function setUp()
    {
        parent::setUp();

        // ...

        $this->hasOne('User', array(
                'local' => 'user_id',
                'foreign' => 'id',
                'onDelete' => 'CASCADE'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Phonenumber:
# ...
  relations:
    # ...
    User:
      local: user_id
      foreign: id
      onDelete: CASCADE
```

> Notice how the integrity constraints are placed on the side where the foreign key exists. This is required in order for the integrity constraints to be exported to your database properly.

Indexes

Introduction

Indexes are used to find rows with specific column values quickly. Without an index, the database must begin with the first row and then read through the entire table to find the relevant rows.

The larger the table, the more this consumes time. If the table has an index for the columns in question, the database can quickly determine the position to seek to in the middle of the data file without having to look at all the data. If a table has 1,000 rows, this is at least 100 times faster than reading rows one-by-one.

Indexes come with a cost as they slow down the inserts and updates. However, in general you should **always** use indexes for the fields that are used in SQL where conditions.

Adding indexes

You can add indexes by using `Doctrine_Record::index`. An example of adding a simple index to field called name:

> The following index examples are not meant for you to actually add to your test Doctrine environment. They are only meant to demonstrate the API for adding indexes.

```php
class IndexTest extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string');

        $this->index('myindex', array(
                'fields' => array('name')
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
IndexTest:
  columns:
    name: string
  indexes:
    myindex:
      fields: [name]
```

An example of adding a multi-column index to field called `name`:

```php
class MultiColumnIndexTest extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string');
        $this->hasColumn('code', 'string');

        $this->index('myindex', array(
                'fields' => array('name', 'code')
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
MultiColumnIndexTest:
  columns:
    name: string
    code: string
  indexes:
    myindex:
      fields: [name, code]
```

An example of adding multiple indexes on same table:

```php
class MultipleIndexTest extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string');
        $this->hasColumn('code', 'string');
        $this->hasColumn('age', 'integer');

        $this->index('myindex', array(
                'fields' => array('name', 'code')
            )
        );

        $this->index('ageindex', array(
                'fields' => array('age')
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
MultipleIndexTest:
  columns:
    name: string
    code: string
    age: integer
  indexes:
    myindex:
      fields: [name, code]
    ageindex:
      fields: [age]
```

Index options

Doctrine offers many index options, some of them being database specific. Here is a full list of available options:

Name	Description
sorting	A string value that can be either 'ASC' or 'DESC'.
length	Index length (only some drivers support this).
primary	Whether or not the index is a primary index.
type	A string value that can be unique, 'fulltext', 'gist' or 'gin'.

Here is an example of how to create a unique index on the name column.

```
class MultipleIndexTest extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string');
        $this->hasColumn('code', 'string');
        $this->hasColumn('age', 'integer');

        $this->index('myindex', array(
                'fields' => array(
                    'name' => array(
                        'sorting' => 'ASC',
                        'length'  => 10),
                        'code'
                    ),
                'type' => 'unique',
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
MultipleIndexTest:
  columns:
    name: string
    code: string
    age: integer
  indexes:
    myindex:
      fields:
        name:
          sorting: ASC
          length: 10
        code: -
      type: unique
```

Special indexes

Doctrine supports many special indexes. These include Mysql FULLTEXT and Pgsql GiST indexes. In the following example we define a Mysql FULLTEXT index for the field 'content'.

```
// models/Article.php

class Article extends Doctrine_Record
{
    public function setTableDefinition()
    {
          $this->hasColumn('name', 'string', 255);
        $this->hasColumn('content', 'string');

        $this->option('type', 'MyISAM');

        $this->index('content', array(
                'fields' => array('content'),
                'type'   => 'fulltext'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
Article:
  options:
    type: MyISAM
  columns:
    name: string(255)
    content: string
  indexes:
    content:
      fields: [content]
      type: fulltext
```

> Notice how we set the table type to `MyISAM`. This is because the `fulltext` index type is only supported in `MyISAM` so you will receive an error if you use something like `InnoDB`.

Checks

You can create any kind of `CHECK` constraints by using the `check()` method of the `Doctrine_Record`. In the last example we add constraint to ensure that price is always higher than the discounted price.

```php
// models/Product.php

class Product extends Doctrine_Record
{
    public function setTableDefinition()
    {
        // ...

        $this->check('price > discounted_price');
    }

    // ...
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
Product:
# ...
  checks:
    price_check: price > discounted_price
```

Generates (in pgsql):

```sql
CREATE TABLE product (
    id INTEGER,
    price NUMERIC,
    discounted_price NUMERIC,
    PRIMARY KEY(id),
    CHECK (price >= 0),
    CHECK (price <= 1000000),
    CHECK (price > discounted_price))
```

> Some databases don't support `CHECK` constraints. When this is the case Doctrine simply skips the creation of check constraints.

If the Doctrine validators are turned on the given definition would also ensure that when a record is being saved its price is always greater than zero.

If some of the prices of the saved products within a transaction is below zero, Doctrine throws `Doctrine_Validator_Exception` and automatically rolls back the transaction.

Table Options

Doctrine offers various table options. All table options can be set via the `Doctrine_Record::option` function.

For example if you are using MySQL and want to use INNODB tables it can be done as follows:

```php
class MyInnoDbRecord extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string');

        $this->option('type', 'INNODB');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
MyInnoDbRecord:
  columns:
    name: string
  options:
    type: INNODB
```

In the following example we set the collate and character set options:

```php
class MyCustomOptionRecord extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string');

        $this->option('collate', 'utf8_unicode_ci');
        $this->option('charset', 'utf8');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
MyCustomOptionRecord:
  columns:
    name: string
  options:
    collate: utf8_unicode_ci
    charset: utf8
```

It is worth noting that for certain databases (Firebird, MySql and PostgreSQL) setting the charset option might not be enough for Doctrine to return data properly. For those databases, users are advised to also use the `setCharset` function of the database connection:

```php
$conn = Doctrine_Manager::connection();
$conn->setCharset('utf8');
```

Record Filters

Doctrine offers the ability to attach record filters when defining your models. A record filter is invoked whenever you access a property on a model that is invalid. So it allows you to essentially add properties dynamically to a model through the use of one of these filters.

To attach a filter you just need to add it in the `setUp()` method of your model definition:

```
class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 255);
        $this->hasColumn('password', 'string', 255);
    }

    public function setUp()
    {
        $this->hasOne('Profile', array(
            'local' => 'id',
            'foreign' => 'user_id'
        ));
        $this->unshiftFilter(new Doctrine_Record_Filter_Compound(array('Profile')));
    }
}

class Profile extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer');
        $this->hasColumn('first_name', 'string', 255);
        $this->hasColumn('last_name', 'string', 255);
    }

    public function setUp()
    {
        $this->hasOne('User', array(
            'local' => 'user_id',
            'foreign' => 'id'
        ));
    }
}
```

Now with the above example we can easily access the properties of the `Profile` relationship when using an instance of `User`. Here is an example:

```
$user = Doctrine_Core::getTable('User')
    ->createQuery('u')
    ->innerJoin('u.Profile p')
    ->where('p.username = ?', 'jwage')
    ->fetchOne();

echo $user->first_name . ' ' . $user->last_name;
```

When we ask for the `first_name` and `last_name` properties they do not exist on the `$user` instance so they are forwarded to the `Profile` relationship. It is the same as if you were to do the following:

```
echo $user->Profile->first_name . ' ' . $user->Profile->last_name;
```

You can write your own record filters pretty easily too. All that is required is you create a class which extends `Doctrine_Record_Filter` and implements the `filterSet()` and `filterGet()` methods. Here is an example:

```
class MyRecordFilter extends Doctrine_Record_Filter
{
    public function filterSet(Doctrine_Record $record, $name, $value)
    {
        // try and set the property

        throw new Doctrine_Record_UnknownPropertyException(sprintf('Unknown record property / related component "%s" on
    }

    public function filterGet(Doctrine_Record, $name)
    {
        // try and get the property

        throw new Doctrine_Record_UnknownPropertyException(sprintf('Unknown record property / related component "%s" on
    }
}
```

Now you can add the filter to your models:

```
class MyModel extends Doctrine_Record
{
    // ...

    public function setUp()
    {
        // ...

        $this->unshiftFilter(new MyRecordFilter());
    }
}
```

> Remember to be sure to throw an instance of the `Doctrine_Record_UnknownPropertyException` exception class if `filterSet()` or `filterGet()` fail to find the property.

Transitive Persistence

Doctrine offers both database and application level cascading operations. This section will explain in detail how to setup both application and database level cascades.

Application-Level Cascades

Since it can be quite cumbersome to save and delete individual objects, especially if you deal with an object graph, Doctrine provides application-level cascading of operations.

Save Cascades

You may already have noticed that `save()` operations are already cascaded to associated objects by default.

Delete Cascades

Doctrine provides a second application-level cascade style: delete. Unlike the `save()` cascade, the delete cascade needs to be turned on explicitly as can be seen in the following code snippet:

```
// models/User.php

class User extends BaseUser
{
    // ...

    public function setUp()
    {
        parent::setup();

        // ...

        $this->hasMany('Address as Addresses', array(
                'local' => 'id',
                'foreign' => 'user_id',
                'cascade' => array('delete')
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
User:
# ...
  relations:
    # ...
    Addresses:
      class: Address
      local: id
      foreign: user_id
      cascade: [delete]
```

The `cascade` option is used to specify the operations that are cascaded to the related objects on the application-level.

> Please note that the only currently supported value is `delete`, more options will be added in future releases of Doctrine.

In the example above, Doctrine would cascade the deletion of a `User` to it's associated `Addresses`. The following describes the generic procedure when you delete a record through `$record->delete()`:

1. Doctrine looks at the relations to see if there are any deletion cascades it needs to apply. If there are no deletion cascades, go to 3).

2. For each relation that has a delete cascade specified, Doctrine verifies that the objects that are the target of the cascade are loaded. That usually means that Doctrine fetches the related objects from the database if they're not yet loaded.(Exception: many-valued associations are always re-fetched from the database, to make sure all objects are loaded). For each associated object, proceed with step 1).

3. Doctrine orders all deletions and executes them in the most efficient way, maintaining referential integrity.

From this description one thing should be instantly clear: Application-level cascades happen on the object-level, meaning operations are cascaded from one object to another and in order to do that the participating objects need to be available.

This has some important implications:

- Application-level delete cascades don't perform well on many-valued associations when there are a lot of objects in the related collection (that is because they need to be fetched from the database, the actual deletion is pretty efficient).
- Application-level delete cascades do not skip the object lifecycle as database-level cascades do (see next chapter). Therefore all registered event listeners and other callback methods are properly executed in an application-level cascade.

Database-Level Cascades

Some cascading operations can be done much more efficiently at the database level. The best example is the delete cascade.

Database-level delete cascades are generally preferrable over application-level delete cascades except:

- Your database does not support database-level cascades (i.e. when using MySql with MYISAM tables).
- You have listeners that listen on the object lifecycle and you want them to get invoked.

Database-level delete cascades are applied on the foreign key constraint. Therefore they're specified on that side of the relation that owns the foreign key. Picking up the example from above, the definition of a database-level cascade would look as follows:

```
// models/Address.php

class Address extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer');
        $this->hasColumn('address', 'string', 255);
        $this->hasColumn('country', 'string', 255);
        $this->hasColumn('city', 'string', 255);
        $this->hasColumn('state', 'string', 2);
        $this->hasColumn('postal_code', 'string', 25);
    }

    public function setUp()
    {
        $this->hasOne('User', array(
                'local' => 'user_id',
                'foreign' => 'id',
                'onDelete' => 'CASCADE'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Address:
  columns:
    user_id: integer
    address: string(255)
    country: string(255)
    city: string(255)
    state: string(2)
    postal_code: string(25)
  relations:
    User:
      local: user_id
      foreign: id
      onDelete: CASCADE
```

The `onDelete` option is translated to proper DDL/DML statements when Doctrine creates your tables.

> Note that `'onDelete' => 'CASCADE'` is specified on the Address class, since the Address owns the foreign key (`user_id`) and database-level cascades are applied on the foreign key.

Currently, the only two supported database-level cascade styles are for `onDelete` and `onUpdate`. Both are specified on the side that owns the foreign key and applied to your database schema when Doctrine creates your tables.

Conclusion

Now that we know everything about how to define our Doctrine models, I think we are ready to move on to learning about how to work with models in your application.

Working with Models

Define Test Schema

> Remember to delete any existing schema information and models from previous chapters.

```
$ rm schema.yml
$ touch schema.yml
$ rm -rf models/*
```

For the next several examples we will use the following schema:

```php
// models/User.php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 255, array(
                'type' => 'string',
                'length' => '255'
            )
        );

        $this->hasColumn('password', 'string', 255, array(
                'type' => 'string',
                'length' => '255'
            )
        );
    }

    public function setUp()
    {
        $this->hasMany('Group as Groups', array(
                'refClass' => 'UserGroup',
                'local' => 'user_id',
                'foreign' => 'group_id'
            )
        );

        $this->hasOne('Email', array(
                'local' => 'id',
                'foreign' => 'user_id'
            )
        );

        $this->hasMany('Phonenumber as Phonenumbers', array(
                'local' => 'id',
                'foreign' => 'user_id'
            )
        );
    }
}

// models/Email.php

class Email extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer', null, array(
                'type' => 'integer'
            )
        );

        $this->hasColumn('address', 'string', 255, array(
                'type' => 'string',
                'length' => '255'
            )
        );
    }

    public function setUp()
    {
        $this->hasOne('User', array(
                'local' => 'user_id',
                'foreign' => 'id'
            )
        );
    }
}

// models/Phonenumber.php

class Phonenumber extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer', null, array(
                'type' => 'integer'
            )
        );

        $this->hasColumn('phonenumber', 'string', 255, array(
                'type' => 'string',
                'length' => '255'
            )
        );
        $this->hasColumn('primary_num', 'boolean');
    }

    public function setUp()
    {
        $this->hasOne('User', array(
                'local' => 'user_id',
                'foreign' => 'id'
            )
        );
    }
}

// models/Group.php
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

User:
  columns:
    username: string(255)
    password: string(255)
  relations:
    Groups:
      class: Group
      local: user_id
      foreign: group_id
      refClass: UserGroup
      foreignAlias: Users

Email:
  columns:
    user_id: integer
    address: string(255)
  relations:
    User:
      foreignType: one

Phonenumber:
  columns:
    user_id: integer
    phonenumber: string(255)
    primary_num: boolean
  relations:
    User:
      foreignAlias: Phonenumbers

Group:
  tableName: groups
  columns:
    name: string(255)

UserGroup:
  columns:
    user_id:
      type: integer
      primary: true
    group_id:
      type: integer
      primary: true
```

Now that you have your schema defined you can instantiate the database by simply running the `generate.php` script we so conveniently created in the previous chapter.

```
$ php generate.php
```

Dealing with Relations

Creating Related Records

Accessing related records in Doctrine is easy: you can use exactly the same getters and setters as for the record properties.

You can use any of the three ways above, however the last one is the recommended one for array portability purposes.

```php
// test.php

// ...
$user = new User();
$user['username'] = 'jwage';
$user['password'] = 'changeme';

$email = $user->Email;

$email = $user->get('Email');

$email = $user['Email'];
```

When accessing a one-to-one related record that doesn't exist, Doctrine automatically creates the object. That is why the above code is possible.

```php
// test.php

// ...
$user->Email->address = 'jonwage@gmail.com';
$user->save();
```

When accessing one-to-many related records, Doctrine creates a `Doctrine_Collection` for the related component. Lets say we have `users` and `phonenumbers` and their relation is one-to-many. You can add `phonenumbers` easily as shown above:

```
// test.php

// ...
$user->Phonenumbers[]->phonenumber = '123 123';
$user->Phonenumbers[]->phonenumber = '456 123';
$user->Phonenumbers[]->phonenumber = '123 777';
```

Now we can easily save the user and the associated phonenumbers:

```
// test.php

// ...
$user->save();
```

Another way to easily create a link between two related components is by using `Doctrine_Record::link()`. It often happens that you have two existing records that you would like to relate (or link) to one another. In this case, if there is a relation defined between the involved record classes, you only need the identifiers of the related record(s):

Lets create a few new `Phonenumber` objects and keep track of the new phone number identifiers:

```
// test.php

// ...
$phoneIds = array();

$phone1 = new Phonenumber();
$phone1['phonenumber'] = '555 202 7890';
$phone1->save();

$phoneIds[] = $phone1['id'];

$phone2 = new Phonenumber();
$phone2['phonenumber'] = '555 100 7890';
$phone2->save();

$phoneIds[] = $phone2['id'];
```

Let's link the phone numbers to the user, since the relation to `Phonenumbers` exists for the `User` record

```
// test.php

$user = new User();
$user['username'] = 'jwage';
$user['password'] = 'changeme';
$user->save();

$user->link('Phonenumbers', $phoneIds);
```

If a relation to the `User` record class is defined for the `Phonenumber` record class, you may even do this:

First create a user to work with:

```
// test.php

// ...
$user = new User();
$user['username'] = 'jwage';
$user['password'] = 'changeme';
$user->save();
```

Now create a new `Phonenumber` instance:

```
// test.php

// ...
$phone1 = new Phonenumber();
$phone1['phonenumber'] = '555 202 7890';
$phone1->save();
```

Now we can link the `User` to our `Phonenumber`:

```
// test.php

// ...
$phone1->link('User', array($user['id']));
```

We can create another phone number:

```php
// test.php

// ...
$phone2 = new Phonenumber();
$phone2['phonenumber'] = '555 100 7890';
$phone2->save();
```

Let's link this `Phonenumber` to our `User` too:

```php
// test.php

// ...
$phone2->link('User', array($user['id']));
```

Retrieving Related Records

You can retrieve related records by the very same `Doctrine_Record` methods as in the previous subchapter. Please note that whenever you access a related component that isn't already loaded Doctrine uses one `SQL SELECT` statement for the fetching, hence the following example executes three `SQL SELECT`s.

```php
// test.php

// ...
$user = Doctrine_Core::getTable('User')->find(1);

echo $user->Email['address'];

echo $user->Phonenumbers[0]->phonenumber;
```

Much more efficient way of doing this is using DQL. The following example uses only one SQL query for the retrieval of related components.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Email e')
    ->leftJoin('u.Phonenumbers p')
    ->where('u.id = ?', 1);

$user = $q->fetchOne();

echo $user->Email['address'];

echo $user->Phonenumbers[0]['phonenumber'];
```

Updating Related Records

You can update the related records by calling save for each related object / collection individually or by calling save on the object that owns the other objects. You can also call `Doctrine_Connection::flush` which saves all pending objects.

```php
// test.php

// ...
$user->Email['address'] = 'koskenkorva@drinkmore.info';

$user->Phonenumbers[0]['phonenumber'] = '123123';

$user->save();
```

> In the above example calling `$user->save()` saves the `email` and `phonenumber`.

Clearing Related Records

You can clear a related records references from an object. This does not change the fact that these objects are related and won't change it in the database if you save. It just simply clears the reference in PHP of one object to another.

You can clear all references by doing the following:

```php
// test.php

// ...
$user->clearRelated();
```

Or you can clear a specific relationship:

```php
// test.php

// ...
$user->clearRelated('Email');
```

This is useful if you were to do something like the following:

```php
// test.php

// ...
if ($user->Email->exists()) {
    // User has e-mail
} else {
    // User does not have a e-mail
}

$user->clearRelated('Email');
```

Because Doctrine will automatically create a new `Email` object if the user does not have one, we need to clear that reference so that if we were to call `$user->save()` it wouldn't save a blank `Email` record for the `User`.

We can simplify the above scenario even further by using the `relatedExists()` method. This is so that you can do the above check with less code and not have to worry about clearing the unnecessary reference afterwards.

```php
if ($user->relatedExists('Email')) {
    // User has e-mail
} else {
    // User does not have a e-mail
}
```

Deleting Related Records

You can delete related records individually be calling `delete()` on a record or on a collection.

Here you can delete an individual related record:

```php
// test.php

// ...
$user->Email->delete();
```

You can delete an individual record from within a collection of records:

```php
// test.php

// ...
$user->Phonenumbers[3]->delete();
```

You could delete the entire collection if you wanted:

```php
// test.php

// ...
$user->Phonenumbers->delete();
```

Or can just delete the entire user and all related objects:

```php
// test.php

// ...
$user->delete();
```

Usually in a typical web application the primary keys of the related objects that are to be deleted come from a form. In this case the most efficient way of deleting the related records is using DQL DELETE statement. Lets say we have once again `Users` and `Phonenumbers` with their relation being one-to-many. Deleting the given `Phonenumbers` for given user id can be achieved as follows:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->delete('Phonenumber')
    ->addWhere('user_id = ?', 5)
    ->whereIn('id', array(1, 2, 3));

$numDeleted = $q->execute();
```

Sometimes you may not want to delete the `Phonenumber` records but to simply unlink the relations by setting the foreign key fields to null. This can of course be achieved with DQL but perhaps to most elegant way of doing this is by using `Doctrine_Record::unlink()`.

> Please note that the `unlink()` method is very smart. It not only sets the foreign fields for related `Phonenumbers` to null but it also removes all given `Phonenumber` references from the `User` object.

Lets say we have a `User` who has three `Phonenumbers` (with identifiers 1, 2 and 3). Now unlinking the `Phonenumbers` 1 and 3 can be achieved as easily as:

```
// test.php

// ...
$user->unlink('Phonenumbers', array(1, 3));

echo $user->Phonenumbers->count(); // 1
```

Working with Related Records

Testing the Existence of a Relation

The below example would return false because the relationship has not been instantiated yet:

```
// test.php

// ...
$user = new User();
if (isset($user->Email)) {
    // ...
}
```

Now the next example will return true because we instantiated the `Email` relationship:

```
// test.php

// ...
$obj->Email = new Email();

if(isset($obj->Email)) {
    // ...
}
```

Many-to-Many Relations

> Doctrine requires that Many-to-Many relationships be bi-directional. For example: both `User` must have many `Groups` and `Group` must have many `User`.

Creating a New Link

Lets say we have two classes `User` and `Group` which are linked through a `GroupUser` association class. When working with transient (new) records the fastest way for adding a `User` and couple of `Group`s for it is:

```
// test.php

// ...
$user = new User();
$user->username = 'Some User';
$user->Groups[0]->username = 'Some Group';
$user->Groups[1]->username = 'Some Other Group';
$user->save();
```

However in real world scenarios you often already have existing groups, where you want to add a given user. The most efficient way of doing this is:

```
// test.php

// ...
$groupUser = new GroupUser();
$groupUser->user_id = $userId;
$groupUser->group_id = $groupId;
$groupUser->save();
```

Deleting a Link

The right way to delete links between many-to-many associated records is by using the DQL DELETE statement. Convenient and recommended way of using DQL DELETE is through the Query API.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->delete('UserGroup')
    ->addWhere('user_id = ?', 5)
    ->whereIn('group_id', array(1, 2));

$deleted = $q->execute();
```

Another way to `unlink` the relationships between related objects is through the `Doctrine_Record::unlink` method. However, you should avoid using this method unless you already have the parent model, since it involves querying the database first.

```php
// test.php

// ...
$user = Doctrine_Core::getTable('User')->find(5);
$user->unlink('Group', array(1, 2));
$user->save();
```

You can also unlink ALL relationships to `Group` by omitting the second argument:

```php
// test.php

// ...
$user->unlink('Group');
```

While the obvious and convenient way of deleting a link between `User` and `Group` would be the following, you still should NOT do this:

```php
// test.php

// ...
$user = Doctrine_Core::getTable('User')->find(5);
$user->GroupUser->remove(0)->remove(1);
$user->save();
```

This is due to a fact that the call to `$user->GroupUser` loads all `Group` links for given `User`. This can be time-consuming task if the `User` belongs to many `Groups`. Even if the user belongs to few `groups` this will still execute an unnecessary SELECT statement.

Fetching Objects

Normally when you fetch data from database the following phases are executed:

- Sending the query to database
- Retrieve the returned data from the database

In terms of object fetching we call these two phases the 'fetching' phase. Doctrine also has another phase called hydration phase. The hydration phase takes place whenever you are fetching structured arrays / objects. Unless explicitly specified everything in Doctrine gets hydrated.

Lets consider we have `Users` and `Phonenumbers` with their relation being one-to-many. Now consider the following plain sql query:

```php
// test.php

// ...
$sql = 'SELECT u.id, u.username, p.phonenumber FROM user u LEFT JOIN phonenumber p ON u.id = p.user_id';
$results = $conn->getDbh()->fetchAll($sql);
```

If you are familiar with these kind of one-to-many joins it may be familiar to you how the basic result set is constructed. Whenever the user has more than one phonenumbers there will be duplicated data in the result set. The result set might look something like:

index	u.id	u.username	p.phonenumber
0	1	Jack Daniels	123 123
1	1	Jack Daniels	456 456
2	2	John Beer	111 111
3	3	John Smith	222 222
4	3	John Smith	333 333
5	3	John Smith	444 444

Here Jack Daniels has two `Phonenumbers`, John Beer has one whereas John Smith has three. You may notice how clumsy this result set is. Its hard to iterate over it as you would need some duplicate data checking logic here and there.

Doctrine hydration removes all duplicated data. It also performs many other things such as:

- Custom indexing of result set elements
- Value casting and preparation
- Value assignment listening
- Makes multi-dimensional array out of the two-dimensional result set array, the number of dimensions is equal to the number of nested joins

Now consider the DQL equivalent of the SQL query we used:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id, u.username, p.phonenumber')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p');

$results = $q->execute(array(), Doctrine_Core::HYDRATE_ARRAY);

print_r($results);
```

The structure of this hydrated array would look like:

```
$ php test.php
Array
(
    [0] => Array
        (
            [id] => 1
            [username] =>
            [Phonenumbers] => Array
                (
                    [0] => Array
                        (
                            [id] => 1
                            [phonenumber] => 123 123
                        )

                    [1] => Array
                        (
                            [id] => 2
                            [phonenumber] => 456 123
                        )

                    [2] => Array
                        (
                            [id] => 3
                            [phonenumber] => 123 777
                        )

                )

        )
    // ...
)
```

This structure also applies to the hydration of objects(records) which is the default hydration mode of Doctrine. The only differences are that the individual elements are represented as `Doctrine_Record` objects and the arrays converted into `Doctrine_Collection` objects. Whether dealing with arrays or objects you can:

- Iterate over the results using **foreach**
- Access individual elements using array access brackets
- Get the number of elements using **count()** function
- Check if given element exists using **isset()**
- Unset given element using **unset()**

You should always use array hydration when you only need to data for access-only purposes, whereas you should use the record hydration when you need to change the fetched data.

The constant O(n) performance of the hydration algorithm is ensured by a smart identifier caching solution.

> Doctrine uses an identity map internally to make sure that multiple objects for one record in a database don't ever exist. If you fetch an object and modify some of its properties, then re-fetch that same object later, the modified properties will be overwritten by default. You can change this behavior by changing the `ATTR_HYDRATE_OVERWRITE` attribute to `false`.

Sample Queries

Count number of records for a relationship:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.*, COUNT(DISTINCT p.id) AS num_phonenumbers')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')
    ->groupBy('u.id');

$users = $q->fetchArray();

echo $users[0]['Phonenumbers'][0]['num_phonenumbers'];
```

Retrieve Users and the Groups they belong to:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Groups g');

$users = $q->fetchArray();

foreach ($users[0]['Groups'] as $group) {
    echo $group['name'];
}
```

Simple WHERE with one parameter value:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->where('u.username = ?', 'jwage');

$users = $q->fetchArray();
```

Multiple WHERE with multiple parameters values:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')
    ->where('u.username = ? AND p.id = ?', array(1, 1));

$users = $q->fetchArray();
```

> You can also optionally use the `andWhere()` method to add to the existing where parts.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')
    ->where('u.username = ?', 1)
    ->andWhere('p.id = ?', 1);

$users = $q->fetchArray();
```

Using `whereIn()` convenience method:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->whereIn('u.id', array(1, 2, 3));

$users = $q->fetchArray();
```

The following is the same as above example:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->where('u.id IN (1, 2, 3)');

$users = $q->fetchArray();
```

Using DBMS function in your WHERE:

```php
// test.php

// ...
$userEncryptedKey = 'a157a558ac00449c92294c7fab684ae0';
$q = Doctrine_Query::create()
    ->from('User u')
    ->where("MD5(CONCAT(u.username, 'secret_key')) = ?", $userEncryptedKey);

$user = $q->fetchOne();

$q = Doctrine_Query::create()
    ->from('User u')
    ->where('LOWER(u.username) = LOWER(?)', 'jwage');

$user = $q->fetchOne();
```

Limiting result sets using aggregate functions. Limit to users with more than one phonenumber:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.*, COUNT(DISTINCT p.id) AS num_phonenumbers')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')
    ->having('num_phonenumbers > 1')
    ->groupBy('u.id');

$users = $q->fetchArray();
```

Join only primary phonenumbers using WITH:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Phonenumbers p WITH p.primary_num = ?', true);

$users = $q->fetchArray();
```

Selecting certain columns for optimization:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username, p.phone')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p');

$users = $q->fetchArray();
```

Using a wildcard to select all `User` columns but only one `Phonenumber` column:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.*, p.phonenumber')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p');

$users = $q->fetchArray();
```

Perform DQL delete with simple WHERE:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->delete('Phonenumber')
    ->addWhere('user_id = 5');

$deleted = $q->execute();
```

Perform simple DQL update for a column:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->update('User u')
    ->set('u.is_active', '?', true)
    ->where('u.id = ?', 1);

$updated = $q->execute();
```

Perform DQL update with DBMS function. Make all usernames lowercase:

```php
// test.php

// ...
$q = Doctrine_Query::create()
  ->update('User u')
  ->set('u.username', 'LOWER(u.username)');

$updated = $q->execute();
```

Using mysql LIKE to search for records:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->where('u.username LIKE ?', '%jwage%');

$users = $q->fetchArray();
```

Use the INDEXBY keyword to hydrate the data where the key of record entry is the name of the column you assign:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u INDEXBY u.username');

$users = $q->fetchArray();
```

Now we can print the user with the username of jwage:

```php
// test.php

// ...
print_r($users['jwage']);
```

Using positional parameters

```php
$q = Doctrine_Query::create()
    ->from('User u')
    ->where('u.username = ?', array('Arnold'));

$users = $q->fetchArray();
```

Using named parameters

```php
$q = Doctrine_Query::create()
    ->from('User u')
    ->where('u.username = :username', array(':username' => 'Arnold'));

$users = $q->fetchArray();
```

Using subqueries in your WHERE. Find users not in group named Group 2:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->where('u.id NOT IN (SELECT u.id FROM User u2 INNER JOIN u2.Groups g WHERE g.name = ?)', 'Group 2');

$users = $q->fetchArray();
```

> You can accomplish this without using subqueries. The two examples below would have the same result as the example above.

Use INNER JOIN to retrieve users who have groups, excluding the group named Group 2

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->innerJoin('u.Groups g WITH g.name != ?', 'Group 2')

$users = $q->fetchArray();
```

Use WHERE condition to retrieve users who have groups, excluding the group named Group 2

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Groups g')
    ->where('g.name != ?', 'Group 2');

$users = $q->fetchArray();
```

Doctrine has many different ways you can execute queries and retrieve the data. Below are examples of all the different ways you can execute a query:

First lets create a sample query to test with:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u');
```

You can use array hydration with the `fetchArray()` method:

```php
$users = $q->fetchArray();
```

You can also use array hydration by specifying the hydration method to the second argument of the `execute()` method:

```php
// test.php

// ...
$users = $q->execute(array(), Doctrine_Core::HYDRATE_ARRAY)
```

You can also specify the hydration method by using the `setHydrationMethod()` method:

```php
$users = $q->setHydrationMode(Doctrine_Core::HYDRATE_ARRAY)->execute(); // So is this
```

> Custom accessors and mutators will not work when hydrating data as anything except records. When you hydrate as an array it is only a static array of data and is not object oriented. If you need to add custom values to your hydrated arrays you can use the some of the events such as **`preHydrate`** and **`postHydrate`**

Sometimes you may want to totally bypass hydration and return the raw data that PDO returns:

```php
// test.php

// ...
$users = $q->execute(array(), Doctrine_Core::HYDRATE_NONE);
```

> More can be read about skipping hydration in the improving performance chapter.

If you want to just fetch one record from the query:

```
// test.php

// ...
$user = $q->fetchOne();

// Fetch all and get the first from collection
$user = $q->execute()->getFirst();
```

Field Lazy Loading

Whenever you fetch an object that has not all of its fields loaded from database then the state of this object is called proxy. Proxy objects can load the unloaded fields lazily.

In the following example we fetch all the Users with the `username` field loaded directly. Then we lazy load the password field:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username')
    ->from('User u')
    ->where('u.id = ?', 1)

$user = $q->fetchOne();
```

The following lazy-loads the `password` field and executes one additional database query to retrieve the value:

```
// test.php

// ...
$user->description;
```

Doctrine does the proxy evaluation based on loaded field count. It does not evaluate which fields are loaded on field-by-field basis. The reason for this is simple: performance. Field lazy-loading is very rarely needed in PHP world, hence introducing some kind of variable to check which fields are loaded would introduce unnecessary overhead to basic fetching.

Arrays and Objects

`Doctrine_Record` and `Doctrine_Collection` provide methods to facilitate working with arrays: `toArray()`, `fromArray()` and `synchronizeWithArray()`.

To Array

The `toArray()` method returns an array representation of your records or collections. It also accesses the relationships the objects may have. If you need to print a record for debugging purposes you can get an array representation of the object and print that.

```
// test.php

// ...
print_r($user->toArray());
```

If you do not want to include the relationships in the array then you need to pass the `$deep` argument with a value of `false`:

```
// test.php

// ...
print_r($user->toArray(false));
```

From Array

If you have an array of values you want to use to fill a record or even a collection, the `fromArray()` method simplifies this common task.

```php
// test.php

// ...
$data = array(
  'name' => 'John',
  'age' => '25',
  'Emails' => array(
    array('address' => 'john@mail.com'),
    array('address' => 'john@work.com')
);

$user = new User();
$user->fromArray($data);
$user->save();
```

Synchronize With Array

`synchronizeWithArray()` allows you to... well, synchronize a record with an array. So if have an array representation of your model and modify a field, modify a relationship field or even delete or create a relationship, this changes will be applied to the record.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.*, g.*')
    ->from('User u')
    ->leftJoin('u.Groups g')
    ->where('id = ?', 1);

$user = $q->fetchOne();
```

Now convert it to an array and modify some of the properties:

```php
// test.php

// ...
$arrayUser = $user->toArray(true);

$arrayUser['username'] = 'New name';
$arrayUser['Group'][0]['name'] = 'Renamed Group';
$arrayUser['Group'][] = array('name' => 'New Group');
```

Now use the same query to retrieve the record and synchronize the record with the `$arrayUser` variable:

```php
// test.php

// ...
$user = Doctrine_Query::create()
    ->select('u.*, g.*')
    ->from('User u')
    ->leftJoin('u.Groups g')
    ->where('id = ?', 1)
    ->fetchOne();

$user->synchronizeWithArray($arrayUser);
$user->save();
```

Overriding the Constructor

Sometimes you want to do some operations at the creation time of your objects. Doctrine doesn't allow you to override the `Doctrine_Record::__construct()` method but provides an alternative:

```php
class User extends Doctrine_Record
{
    public function construct()
    {
        $this->username = 'Test Name';
        $this->doSomething();
    }

    public function doSomething()
    {
        // ...
    }

    // ...
}
```

The only drawback is that it doesn't provide a way to pass parameters to the constructor.

Conclusion

By now we should know absolutely everything there is to know about models. We know how to create them, load them and most importantly we know how to use them and work with columns and relationships. Now we are ready to move on to learn about how to use the DQL (Doctrine Query Language).

DQL (Doctrine Query Language)

Introduction

Doctrine Query Language (DQL) is an Object Query Language created for helping users in complex object retrieval. You should always consider using DQL (or raw SQL) when retrieving relational data efficiently (eg. when fetching users and their phonenumbers).

In this chapter we will execute dozens of examples of how to use the Doctrine Query Language. All of these examples assume you are using the schemas defined in the previous chapters, primarily the Defining Models chapter. We will define one additional model for our testing purposes:

```
// models/Account.php

class Account extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 255);
        $this->hasColumn('amount', 'decimal');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Account:
  columns:
    name: string(255)
    amount: decimal
```

When compared to using raw SQL, DQL has several benefits:

- From the start it has been designed to retrieve records(objects) not result set rows
- DQL understands relations so you don't have to type manually sql joins and join conditions
- DQL is portable on different databases
- DQL has some very complex built-in algorithms like (the record limit algorithm) which can help the developer to efficiently retrieve objects
- It supports some functions that can save time when dealing with one-to-many, many-to-many relational data with conditional fetching.

If the power of DQL isn't enough, you should consider using the RawSql API for object population.

You may already be familiar with the following syntax:

> DO NOT USE THE FOLLOWING CODE. It uses many sql queries for object population.

```
// test.php

// ...
$users = Doctrine_Core::getTable('User')->findAll();

foreach($users as $user) {
    echo $user->username . " has phonenumbers: \n";

    foreach($user->Phonenumbers as $phonenumber) {
        echo $phonenumber->phonenumber . "\n";
    }
}
```

> Here is the same code but implemented more efficiently using only one SQL query for object population.

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Phonenumbers p');

echo $q->getSqlQuery();
```

Lets take a look at the SQL that would be generated by the above query:

```
SELECT
u.id AS u__id,
u.is_active AS u__is_active,
u.is_super_admin AS u__is_super_admin,
u.first_name AS u__first_name,
u.last_name AS u__last_name,
u.username AS u__username,
u.password AS u__password,
u.type AS u__type,
u.created_at AS u__created_at,
u.updated_at AS u__updated_at,
p.id AS p__id,
p.user_id AS p__user_id,
p.phonenumber AS p__phonenumber
FROM user u
LEFT JOIN phonenumber p ON u.id = p.user_id
```

Now lets execute the query and play with the data:

```
// test.php

// ...
$users = $q->execute();

foreach($users as $user) {
    echo $user->username . " has phonenumbers: \n";

    foreach($user->Phonenumbers as $phonenumber) {
        echo $phonenumber->phonenumber . "\n";
    }
}
```

> Using double quotes (") in DQL strings is discouraged. This is sensible in MySQL standard but in DQL it can be confused as an identifier. Instead it is recommended to use prepared statements for your values and it will be escaped properly.

SELECT queries

`SELECT` statement syntax:

```
SELECT
    [ALL | DISTINCT]
    <select_expr>,
...
    [
FROM <components>
    [
WHERE <where_condition>]
    [
GROUP BY <groupby_expr>
      [ASC | DESC],
... ]
    [
HAVING <where_condition>]
    [
ORDER BY <orderby_expr>
      [ASC | DESC],
...]
    [
LIMIT <row_count>
OFF
SET <offset>}]
```

The `SELECT` statement is used for the retrieval of data from one or more components.

Each `select_expr` indicates a column or an aggregate function value that you want to retrieve. There must be at least one `select_expr` in every `SELECT` statement.

First insert a few sample `Account` records:

```
// test.php

// ...
$account = new Account();
$account->name = 'test 1';
$account->amount = '100.00';
$account->save();

$account = new Account();
$account->name = 'test 2';
$account->amount = '200.00';
$account->save();
```

Be sure to execute `test.php`:

```
$ php test.php
```

Now you can test the selecting of the data with these next few sample queries:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('a.name')
    ->from('Account a');

echo $q->getSqlQuery();
```

Lets take a look at the SQL that would be generated by the above query:

```
SELECT
a.id AS a__id,
a.name AS a__name
FROM account a
```

```
// test.php

// ...
$accounts = $q->execute();
print_r($accounts->toArray());
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [0] => Array
        (
            [id] => 1
            [name] => test 1
            [amount] =>
        )

    [1] => Array
        (
            [id] => 2
            [name] => test 2
            [amount] =>
        )

)
```

An asterisk can be used for selecting all columns from given component. Even when using an asterisk the executed sql queries never actually use it (Doctrine converts asterisk to appropriate column names, hence leading to better performance on some databases).

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('a.*')
    ->from('Account a');

echo $q->getSqlQuery();
```

Compare the generated SQL from the last query example to the SQL generated by the query right above:

```
SELECT
a.id AS a__id,
a.name AS a__name,
a.amount AS a__amount
FROM account a
```

> Notice how the asterisk is replace by all the real column names that exist in the `Account` model.

Now lets execute the query and inspect the results:

```
// test.php

// ...
$accounts = $q->execute();
print_r($accounts->toArray());
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [0] => Array
        (
            [id] => 1
            [name] => test 1
            [amount] => 100.00
        )

    [1] => Array
        (
            [id] => 2
            [name] => test 2
            [amount] => 200.00
        )

)
```

`FROM` clause `components` indicates the component or components from which to retrieve records.

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username, p.*')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
u.username AS u__username,
p.id AS p__id,
p.user_id AS p__user_id,
p.phonenumber AS p__phonenumber
FROM user u
LEFT JOIN phonenumber p ON u.id = p.user_id
```

The `WHERE` clause, if given, indicates the condition or conditions that the records must satisfy to be selected. `where_condition` is an expression that evaluates to true for each row to be selected. The statement selects all rows if there is no `WHERE` clause.

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('a.name')
    ->from('Account a')
    ->where('a.amount > 2000');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
a.id AS a__id,
a.name AS a__name
FROM account a
WHERE a.amount > 2000
```

In the `WHERE` clause, you can use any of the functions and operators that DQL supports, except for aggregate (summary) functions. The `HAVING` clause can be used for narrowing the results with aggregate functions:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')
    ->having('COUNT(p.id) > 3');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
u.username AS u__username
FROM user u
LEFT JOIN phonenumber p ON u.id = p.user_id
HAVING COUNT(p.id) > 3
```

The `ORDER BY` clause can be used for sorting the results

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username')
    ->from('User u')
    ->orderBy('u.username');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
u.username AS u__username
FROM user u
ORDER BY u.username
```

The `LIMIT` and `OFFSET` clauses can be used for efficiently limiting the number of records to a given `row_count`

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username')
    ->from('User u')
    ->limit(20);

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
u.username AS u__username
FROM user u
LIMIT 20
```

Aggregate values

Aggregate value `SELECT` syntax:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id, COUNT(t.id) AS num_threads')
    ->from('User u, u.Threads t')
    ->where('u.id = ?', 1)
    ->groupBy('u.id');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
COUNT(f.id) AS f__0
FROM user u
LEFT JOIN forum__thread f ON u.id = f.user_id
WHERE u.id = ?
GROUP BY u.id
```

Now execute the query and inspect the results:

```
// test.php

// ...
$users = $q->execute();
```

You can easily access the `num_threads` data with the following code:

```php
// test.php

// ...
echo $users->num_threads . ' threads found';
```

UPDATE queries

`UPDATE` statement syntax:

```
UPDATE <component_name>
SET <col_name1> = <expr1> ,
<col_name2> = <expr2>
WHERE <where_condition>
ORDER BY <order_by>
LIMIT <record_count>
```

- The `UPDATE` statement updates columns of existing records in `component_name` with new values and returns the number of affected records.
- The `SET` clause indicates which columns to modify and the values they should be given.
- The optional `WHERE` clause specifies the conditions that identify which records to update. Without `WHERE` clause, all records are updated.
- The optional `ORDER BY` clause specifies the order in which the records are being updated.
- The `LIMIT` clause places a limit on the number of records that can be updated. You can use `LIMIT row_count` to restrict the scope of the `UPDATE`. A `LIMIT` clause is a **rows-matched restriction** not a rows-changed restriction. The statement stops as soon as it has found `record_count` rows that satisfy the `WHERE` clause, whether or not they actually were changed.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->update('Account')
    ->set('amount', 'amount + 200')
    ->where('id > 200');

// If you just want to set the amount to a value
$q->set('amount', '?', 500);

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
UPDATE account
SET amount = amount + 200
WHERE id > 200
```

Now to perform the update is simple. Just execute the query:

```php
// test.php

// ...
$rows = $q->execute();

echo $rows;
```

DELETE Queries

```
DELETE
FROM <component_name>
WHERE <where_condition>
ORDER BY <order_by>
LIMIT <record_count>
```

- The `DELETE` statement deletes records from `component_name` and returns the number of records deleted.
- The optional `WHERE` clause specifies the conditions that identify which records to delete. Without `WHERE` clause, all records are deleted.
- If the `ORDER BY` clause is specified, the records are deleted in the order that is specified.
- The `LIMIT` clause places a limit on the number of rows that can be deleted. The statement will stop as soon as it has deleted `record_count` records.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->delete('Account a')
    ->where('a.id > 3');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
DELETE
FROM account
WHERE id > 3
```

Now executing the `DELETE` query is just as you would think:

```php
// test.php

// ...
$rows = $q->execute();

echo $rows;
```

> When executing DQL UPDATE and DELETE queries the executing of a query returns the number of affected rows.

FROM clause

Syntax:

```
FROM <component_reference> [[LEFT | INNER] JOIN <component_reference>] ...
```

The `FROM` clause indicates the component or components from which to retrieve records. If you name more than one component, you are performing a join. For each table specified, you can optionally specify an alias.

Consider the following DQL query:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
```

Here `User` is the name of the class (component) and `u` is the alias. You should always use short aliases, since most of the time those make the query much shorther and also because when using for example caching the cached form of the query takes less space when short aliases are being used.

JOIN syntax

DQL JOIN Syntax:

```
JOIN <component_reference1>] [ON | WITH] <join_condition1> [INDEXBY] <map_condition1>,
[[LEFT | INNER] JOIN <component_reference2>] [ON | WITH] <join_condition2> [INDEXBY] <map_condition2>,
...
[[LEFT | INNER] JOIN <component_referenceN>] [ON | WITH] <join_conditionN> [INDEXBY] <map_conditionN>
```

DQL supports two kinds of joins INNER JOINs and LEFT JOINs. For each joined component, you can optionally specify an alias.

The default join type is `LEFT JOIN`. This join can be indicated by the use of either `LEFT JOIN` clause or simply `','`, hence the following queries are equal:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id, p.id')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p');

$q = Doctrine_Query::create()
    ->select('u.id, p.id')
    ->from('User u, u.Phonenumbers p');

echo $q->getSqlQuery();
```

> The recommended form is the first because it is more verbose and easier to read and understand what is being done.

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
u.id AS u__id,
p.id AS p__id
FROM user u
LEFT JOIN phonenumber p ON u.id = p.user_id
```

> Notice how the JOIN condition is automatically added for you. This is because Doctrine knows how `User` and `Phonenumber` are related so it is able to add it for you.

`INNER JOIN` produces an intersection between two specified components (that is, each and every record in the first component is joined to each and every record in the second component). So basically `INNER JOIN` can be used when you want to efficiently fetch for example all users which have one or more phonenumbers.

By default DQL auto-adds the primary key join condition:

```php
// test.php

// ...
$q = Doctrine_Query::create()
  ->select('u.id, p.id')
  ->from('User u')
  ->leftJoin('u.Phonenumbers p');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
u.id AS u__id,
p.id AS p__id
FROM User u
LEFT JOIN Phonenumbers p ON u.id = p.user_id
```

ON keyword

If you want to override this behavior and add your own custom join condition you can do it with the `ON` keyword. Consider the following DQL query:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id, p.id')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p ON u.id = 2');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
u.id AS u__id,
p.id AS p__id
FROM User u
LEFT JOIN Phonenumbers p ON u.id = 2
```

> Notice how the `ON` condition that would be normally automatically added is not present and the user specified condition is used instead.

WITH keyword

Most of the time you don't need to override the primary join condition, rather you may want to add some custom conditions. This can be achieved with the `WITH` keyword.

```
// test.php

// ...
$q = Doctrine_Query::create()
  ->select('u.id, p.id')
  ->from('User u')
  ->leftJoin('u.Phonenumbers p WITH u.id = 2');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
p.id AS p__id
FROM User u
LEFT JOIN Phonenumbers p ON u.id = p.user_id
AND u.id = 2
```

> Notice how the `ON` condition isn't completely replaced. Instead the conditions you specify are appended on to the automatic condition that is added for you.

The Doctrine_Query API offers two convenience methods for adding JOINS. These are called `innerJoin()` and `leftJoin()`, which usage should be quite intuitive as shown below:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u')
    ->leftJoin('u.Groups g')
    ->innerJoin('u.Phonenumbers p WITH u.id > 3')
    ->leftJoin('u.Email e');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
LEFT JOIN user_group u2 ON u.id = u2.user_id
LEFT JOIN groups g ON g.id = u2.group_id
INNER JOIN phonenumber p ON u.id = p.user_id
AND u.id > 3
LEFT JOIN email e ON u.id = e.user_id
```

INDEXBY keyword

The `INDEXBY` keyword offers a way of mapping certain columns as collection / array keys. By default Doctrine indexes multiple elements to numerically indexed arrays / collections. The mapping starts from zero. In order to override this behavior you need to use `INDEXBY` keyword as shown above:

```
// test.php

// ...
$q = Doctrine_Query::create()
  ->from('User u INDEXBY u.username');

$users = $q->execute();
```

> The `INDEXBY` keyword does not alter the generated SQL. It is simply used internally by `Doctrine_Query` to hydrate the data with the specified column as the key of each record in the collection.

Now the users in `$users` collection are accessible through their names:

```
// test.php

// ...
echo $user['jack daniels']->id;
```

The `INDEXBY` keyword can be applied to any given JOIN. This means that any given component can have each own indexing behavior. In the following we use distinct indexing for both `Users` and `Groups`.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u INDEXBY u.username')
    ->innerJoin('u.Groups g INDEXBY g.name');

$users = $q->execute();
```

Now lets print out the drinkers club's creation date.

```php
// test.php

// ...
echo $users['jack daniels']->Groups['drinkers club']->createdAt;
```

WHERE clause

Syntax:

```
WHERE <where_condition>
```

- The `WHERE` clause, if given, indicates the condition or conditions that the records must satisfy to be selected.
- `where_condition` is an expression that evaluates to true for each row to be selected.
- The statement selects all rows if there is no `WHERE` clause.
- When narrowing results with aggregate function values `HAVING` clause should be used instead of `WHERE` clause

You can use the `addWhere()`, `andWhere()`, `orWhere()`, `whereIn()`, `andWhereIn()`, `orWhereIn()`, `whereNotIn()`, `andWhereNotIn()`, `orWhereNotIn()` functions for building complex where conditions using `Doctrine_Query` objects.

Here is an example where we retrieve all active registered users or super administrators:

```php
// test.php

// ...
$q = Doctrine_Query::create()
  ->select('u.id')
  ->from('User u')
  ->where('u.type = ?', 'registered')
  ->andWhere('u.is_active = ?', 1)
  ->orWhere('u.is_super_admin = ?', 1);

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.type = ?
AND u.is_active = ?
OR u.is_super_admin = ?
```

Conditional expressions

Literals

Strings

A string literal that includes a single quote is represented by two single quotes; for example: ´´literal´s´´.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id, u.username')
    ->from('User u')
    ->where('u.username = ?', 'Vincent');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
u.username AS u__username
FROM user u
WHERE u.username = ?
```

> Because we passed the value of the `username` as a parameter to the `where()` method it is not included in the generated SQL. PDO handles the replacement when you execute the query. To check the parameters that exist on a `Doctrine_Query` instance you can use the `getParams()` method.

Integers

Integer literals support the use of PHP integer literal syntax.

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('a.id')
    ->from('User u')
    ->where('u.id = 4');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.id = 4
```

Floats

Float literals support the use of PHP float literal syntax.

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('a.id')
    ->from('Account a')
    ->where('a.amount = 432.123');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
a.id AS a__id
FROM account a
WHERE a.amount = 432.123
```

Booleans

The boolean literals are true and false.

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('a.id')
    ->from('User u')
    ->where('u.is_super_admin = true');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.is_super_admin = 1
```

Enums

The enumerated values work in the same way as string literals.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('a.id')
    ->from('User u')
    ->where("u.type = 'admin'");

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.type = 'admin'
```

Predefined reserved literals are case insensitive, although its a good standard to write them in uppercase.

Input parameters

Here are some examples of using positional parameters:

Single positional parameter:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u')
    ->where('u.username = ?', array('Arnold'));

echo $q->getSqlQuery();
```

> When the passed parameter for a positional parameter contains only one value you can simply pass a single scalar value instead of an array containing one value.

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.username = ?
```

Multiple positional parameters:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->where('u.id > ? AND u.username LIKE ?', array(50, 'A%'));

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE (u.id > ?
AND u.username LIKE ?)
```

Here are some examples of using named parameters:

Single named parameter:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u')
    ->where('u.username = :name', array(':name' => 'Arnold'));

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.username = :name
```

Named parameter with a LIKE statement:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u')
    ->where('u.id > :id', array(':id' => 50))
    ->andWhere('u.username LIKE :name', array(':name' => 'A%'));

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.id > :id
AND u.username LIKE :name
```

Operators and operator precedence

The operators are listed below in order of decreasing precedence.

Operator	Description
.	Navigation operator
	Arithmetic operators:
+, -	unary
*, /	multiplication and division
+, -	addition and subtraction
=, >, >=, <, <=, <> (not equal),	Comparison operators
[NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY	
	Logical operators:
NOT	
AND	
OR	

In expressions

Syntax:

```
<operand> IN (<subquery>|<value list>)
```

An `IN` conditional expression returns true if the **operand** is found from result of the **subquery** or if its in the specificied comma separated **value list**, hence the IN expression is always false if the result of the subquery is empty.

When **value list** is being used there must be at least one element in that list.

Here is an example where we use a subquery for the `IN`:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->where('u.id IN (SELECT u.id FROM User u INNER JOIN u.Groups g WHERE g.id = ?)', 1);

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.id IN (SELECT
u2.id AS u2__id
FROM user u2
INNER JOIN user_group u3 ON u2.id = u3.user_id
INNER JOIN groups g ON g.id = u3.group_id
WHERE g.id = ?)
```

Here is an example where we just use a list of integers:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u')
    ->whereIn('u.id', array(1, 3, 4, 5));

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.id IN (?,
?,
?,
?)
```

Like Expressions

Syntax:

```
string_expression [NOT] LIKE pattern_value [ESCAPE escape_character]
```

The string_expression must have a string value. The pattern_value is a string literal or a string-valued input parameter in which an underscore (_) stands for any single character, a percent (%) character stands for any sequence of characters (including the empty sequence), and all other characters stand for themselves. The optional escape_character is a single-character string literal or a character-valued input parameter (i.e., char or Character) and is used to escape the special meaning of the underscore and percent characters in `pattern_value`.

Examples:

- address.phone LIKE '12%3' is true for '123' '12993' and false for '1234'
- asentence.word LIKE 'l_se' is true for 'lose' and false for 'loose'
- aword.underscored LIKE '_%' ESCAPE '\' is true for '_foo' and false for 'bar'
- address.phone NOT LIKE '12%3' is false for '123' and '12993' and true for '1234'

If the value of the string_expression or pattern_value is NULL or unknown, the value of the LIKE expression is unknown. If the escape_characteris specified and is NULL, the value of the LIKE expression is unknown.

Find all users whose email ends with '@gmail.com':

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u')
    ->leftJoin('u.Email e')
    ->where('e.address LIKE ?', '%@gmail.com');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
LEFT JOIN email e ON u.id = e.user_id
WHERE e.address LIKE ?
```

Find all users whose name starts with letter 'A':

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u')
    ->where('u.username LIKE ?', 'A%');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.username LIKE ?
```

Exists Expressions

Syntax:

```
<operand> [NOT ]EXISTS (<subquery>)
```

The `EXISTS` operator returns `TRUE` if the subquery returns one or more rows and `FALSE` otherwise.

The `NOT EXISTS` operator returns `TRUE` if the subquery returns 0 rows and `FALSE` otherwise.

> For the next few examples we need to add the `ReaderLog` model.

```php
// models/ReaderLog.php

class ReaderLog extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('article_id', 'integer', null, array(
                'primary' => true
            )
        );

        $this->hasColumn('user_id', 'integer', null, array(
                'primary' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
ReaderLog:
  columns:
    article_id:
      type: integer
      primary: true
    user_id:
      type: integer
      primary: true
```

> After adding the `ReaderLog` model don't forget to run the `generate.php` script!

```
$ php generate.php
```

Now we can run some tests! First, finding all articles which have readers:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('a.id')
    ->from('Article a')
    ->where('EXISTS (SELECT r.id FROM ReaderLog r WHERE r.article_id = a.id)');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
a.id AS a__id
FROM article a
WHERE EXISTS (SELECT
r.id AS r__id
FROM reader_log r
WHERE r.article_id = a.id)
```

Finding all articles which don't have readers:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('a.id')
    ->from('Article a')
    ->where('NOT EXISTS (SELECT r.id FROM ReaderLog r WHERE r.article_id = a.id));

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
a.id AS a__id
FROM article a
WHERE NOT EXISTS (SELECT
r.id AS r__id
FROM reader_log r
WHERE r.article_id = a.id)
```

All and Any Expressions

Syntax:

```
operand comparison_operator ANY (subquery)
operand comparison_operator SOME (subquery)
operand comparison_operator ALL (subquery)
```

An ALL conditional expression returns true if the comparison operation is true for all values in the result of the subquery or the result of the subquery is empty. An ALL conditional expression is false if the result of the comparison is false for at least one row, and is unknown if neither true nor false.

```
$q = Doctrine_Query::create()
    ->from('C')
    ->where('C.col1 < ALL (FROM C2(col1))');
```

An ANY conditional expression returns true if the comparison operation is true for some value in the result of the subquery. An ANY conditional expression is false if the result of the subquery is empty or if the comparison operation is false for every value in the result of the subquery, and is unknown if neither true nor false.

```
$q = Doctrine_Query::create()
    ->from('C')
    ->where('C.col1 > ANY (FROM C2(col1))');
```

The keyword SOME is an alias for ANY.

```
$q = Doctrine_Query::create()
    ->from('C')
    ->where('C.col1 > SOME (FROM C2(col1))');
```

The comparison operators that can be used with ALL or ANY conditional expressions are =, <, <=, >, >=, <>. The result of the subquery must be same type with the conditional expression.

NOT IN is an alias for <> ALL. Thus, these two statements are equal:

```

FROM C
WHERE C.col1 <> ALL (
FROM C2(col1));

FROM C
WHERE C.col1 NOT IN (
FROM C2(col1));
```

```
$q = Doctrine_Query::create()
    ->from('C')
    ->where('C.col1 <> ALL (FROM C2(col1))');

$q = Doctrine_Query::create()
    ->from('C')
    ->where('C.col1 NOT IN (FROM C2(col1))');
```

Subqueries

A subquery can contain any of the keywords or clauses that an ordinary SELECT query can contain.

Some advantages of the subqueries:

- They allow queries that are structured so that it is possible to isolate each part of a statement.
- They provide alternative ways to perform operations that would otherwise require complex joins and unions.
- They are, in many people's opinion, readable. Indeed, it was the innovation of subqueries that gave people the original idea of calling the early SQL "Structured Query Language."

Here is an example where we find all users which don't belong to the group id 1:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u')
    ->where('u.id NOT IN (SELECT u2.id FROM User u2 INNER JOIN u2.Groups g WHERE g.id = ?)', 1);

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.id NOT IN (SELECT
u2.id AS u2__id
FROM user u2
INNER JOIN user_group u3 ON u2.id = u3.user_id
INNER JOIN groups g ON g.id = u3.group_id
WHERE g.id = ?)
```

Here is an example where we find all users which don't belong to any groups

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u')
    ->where('u.id NOT IN (SELECT u2.id FROM User u2 INNER JOIN u2.Groups g)');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.id NOT IN (SELECT
u2.id AS u2__id
FROM user u2
INNER JOIN user_group u3 ON u2.id = u3.user_id
INNER JOIN groups g ON g.id = u3.group_id)
```

Functional Expressions

String functions

The **CONCAT** function returns a string that is a concatenation of its arguments. In the example above we map the concatenation of users `first_name` and `last_name` to a value called `name`.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('CONCAT(u.first_name, u.last_name) AS name')
    ->from('User u');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
CONCAT(u.first_name,
u.last_name) AS u__0
FROM user u
```

Now we can execute the query and get the mapped function value:

```php
$users = $q->execute();

foreach($users as $user) {
    // here 'name' is not a property of $user,
    // its a mapped function value
    echo $user->name;
}
```

The second and third arguments of the **SUBSTRING** function denote the starting position and length of the substring to be returned. These arguments are integers. The first position of a string is denoted by 1. The **SUBSTRING** function returns a string.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username')
    ->from('User u')
    ->where("SUBSTRING(u.username, 0, 1) = 'z'");

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
u.username AS u__username
FROM user u
WHERE SUBSTRING(u.username
FROM 0 FOR 1) = 'z'
```

> Notice how the SQL is generated with the proper `SUBSTRING` syntax for the DBMS you are using!

The **TRIM** function trims the specified character from a string. If the character to be trimmed is not specified, it is assumed to be space (or blank). The optional trim_character is a single-character string literal or a character-valued input parameter (i.e., char or Character)[30]. If a trim specification is not provided, BOTH is assumed. The **TRIM** function returns the trimmed string.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username')
    ->from('User u')
    ->where('TRIM(u.username) = ?', 'Someone');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
u.username AS u__username
FROM user u
WHERE TRIM(u.username) = ?
```

The **LOWER** and **UPPER** functions convert a string to lower and upper case, respectively. They return a string.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username')
    ->from('User u')
    ->where("LOWER(u.username) = 'jon wage'");

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
u.username AS u__username
FROM user u
WHERE LOWER(u.username) = 'someone'
```

The **LOCATE** function returns the position of a given string within a string, starting the search at a specified position. It returns the first position at which the string was found as an integer. The first argument is the string to be located; the second argument is the string to be searched; the optional third argument is an integer that represents the string position at which the search is started (by default, the beginning of the string to be searched). The first position in a string is denoted by 1. If the string is not found, 0 is returned.

The **LENGTH** function returns the length of the string in characters as an integer.

Arithmetic functions

Availible DQL arithmetic functions:

```
ABS(simple_arithmetic_expression)
SQRT(simple_arithmetic_expression)
MOD(simple_arithmetic_expression, simple_arithmetic_expression)
```

- The **ABS** function returns the absolute value for given number.
- The **SQRT** function returns the square root for given number.
- The **MOD** function returns the modulus of first argument using the second argument.

Subqueries

Introduction

Doctrine allows you to use sub-dql queries in the FROM, SELECT and WHERE statements. Below you will find examples for all the different types of subqueries Doctrine supports.

Comparisons using subqueries

Find all the users which are not in a specific group.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u')
    ->where('u.id NOT IN (SELECT u.id FROM User u INNER JOIN u.Groups g WHERE g.id = ?)', 1);

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
WHERE u.id NOT IN (SELECT
u2.id AS u2__id
FROM user u2
INNER JOIN user_group u3 ON u2.id = u3.user_id
INNER JOIN groups g ON g.id = u3.group_id
WHERE g.id = ?)
```

Retrieve the users phonenumber in a subquery and include it in the resultset of user information.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->addSelect('(SELECT p.phonenumber FROM Phonenumber p WHERE p.user_id = u.id LIMIT 1) as phonenumber')
    ->from('User u');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
u.id AS u__id,
(SELECT
p.phonenumber AS p__phonenumber
FROM phonenumber p
WHERE p.user_id = u.id
LIMIT 1) AS u__0
FROM user u
```

GROUP BY, HAVING clauses

DQL GROUP BY syntax:

```
GROUP BY groupby_item {,
groupby_item}*
```

DQL HAVING syntax:

```
HAVING conditional_expression
```

`GROUP BY` and `HAVING` clauses can be used for dealing with aggregate functions. The Following aggregate functions are available on DQL: `COUNT`, `MAX`, `MIN`, `AVG`, `SUM`

Selecting alphabetically first user by name.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('MIN(a.amount)')
    ->from('Account a');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
MIN(a.amount) AS a__0
FROM account a
```

Selecting the sum of all Account amounts.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('SUM(a.amount)')
    ->from('Account a');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
SUM(a.amount) AS a__0
FROM account a
```

Using an aggregate function in a statement containing no `GROUP BY` clause, results in grouping on all rows. In the example below we fetch all users and the number of phonenumbers they have.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username')
    ->addSelect('COUNT(p.id) as num_phonenumbers')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')
    ->groupBy('u.id');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
u.id AS u__id,
u.username AS u__username,
COUNT(p.id) AS p__0
FROM user u
LEFT JOIN phonenumber p ON u.id = p.user_id
GROUP BY u.id
```

The `HAVING` clause can be used for narrowing the results using aggregate values. In the following example we fetch all users which have at least 2 phonenumbers

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username')
    ->addSelect('COUNT(p.id) as num_phonenumbers')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')
    ->groupBy('u.id')
    ->having('num_phonenumbers >= 2');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
u.id AS u__id,
u.username AS u__username,
COUNT(p.id) AS p__0
FROM user u
LEFT JOIN phonenumber p ON u.id = p.user_id
GROUP BY u.id
HAVING p__0 >= 2
```

You can access the number of phonenumbers with the following code:

```php
// test.php

// ...
$users = $q->execute();

foreach($users as $user) {
    echo $user->name . ' has ' . $user->num_phonenumbers . ' phonenumbers';
}
```

ORDER BY clause

Introduction

Record collections can be sorted efficiently at the database level using the ORDER BY clause.

Syntax:

```
, ...]
```

Examples:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')
    ->orderBy('u.username, p.phonenumber');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
u.id AS u__id,
u.username AS u__username
FROM user u
LEFT JOIN phonenumber p ON u.id = p.user_id
ORDER BY u.username,
p.phonenumber
```

In order to sort in reverse order you can add the `DESC` (descending) keyword to the name of the column in the `ORDER BY` clause that you are sorting by. The default is ascending order; this can be specified explicitly using the `ASC` keyword.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username')
    ->from('User u')
    ->leftJoin('u.Email e')
    ->orderBy('e.address DESC, u.id ASC');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
u.id AS u__id,
u.username AS u__username
FROM user u
LEFT JOIN email e ON u.id = e.user_id
ORDER BY e.address DESC,
u.id ASC
```

Sorting by an aggregate value

In the following example we fetch all users and sort those users by the number of phonenumbers they have.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username, COUNT(p.id) count')
    ->from('User u')
    ->innerJoin('u.Phonenumbers p')
    ->orderby('count');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
u.id AS u__id,
u.username AS u__username,
COUNT(p.id) AS p__0
FROM user u
INNER JOIN phonenumber p ON u.id = p.user_id
ORDER BY p__0
```

Using random order

In the following example we use random in the `ORDER BY` clause in order to fetch random post.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('t.id, RANDOM() AS rand')
    ->from('Forum_Thread t')
    ->orderby('rand')
    ->limit(1);

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
f.id AS f__id,
RAND() AS f__0
FROM forum__thread f
ORDER BY f__0
LIMIT 1
```

LIMIT and OFFSET clauses

Propably the most complex feature DQL parser has to offer is its `LIMIT` clause parser. Not only does the DQL `LIMIT` clause parser take care of `LIMIT` database portability it is capable of limiting the number of records instead of rows by using complex query analysis and subqueries.

Retrieve the first 20 users and all their associated phonenumbers:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.username, p.phonenumber')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')
    ->limit(20);

echo $q->getSqlQuery();
```

> You can also use the `offset()` method of the `Doctrine_Query` object in combination with the `limit()` method to produce your desired `LIMIT` and `OFFSET` in the executed SQL query.

The above call to `getSql()` would output the following SQL query:

```sql
SELECT
u.id AS u__id,
u.username AS u__username,
p.id AS p__id,
p.phonenumber AS p__phonenumber
FROM user u
LEFT JOIN phonenumber p ON u.id = p.user_id
LIMIT 20
```

Driver Portability

DQL `LIMIT` clause is portable on all supported databases. Special attention have been paid to following facts:

- Only Mysql, Pgsql and Sqlite implement LIMIT / OFFSET clauses natively
- In Oracle / Mssql / Firebird LIMIT / OFFSET clauses need to be emulated in driver specific way
- The limit-subquery-algorithm needs to execute to subquery separately in mysql, since mysql doesn't yet support LIMIT clause in subqueries
- Pgsql needs the order by fields to be preserved in SELECT clause, hence limit-subquery-algorithm needs to take this into consideration when pgsql driver is used
- Oracle only allows < 30 object identifiers (= table/column names/aliases), hence the limit subquery must use as short aliases as possible and it must avoid alias collisions with the main query.

The limit-subquery-algorithm

The limit-subquery-algorithm is an algorithm that DQL parser uses internally when one-to-many / many-to-many relational data is being fetched simultaneously. This kind of special algorithm is needed for the LIMIT clause to limit the number of records instead of sql result set rows.

This behavior can be overwritten using the configuration system (at global, connection or table level) using:

```
$table->setAttribute(Doctrine_Core::ATTR_QUERY_LIMIT, Doctrine_Core::LIMIT_ROWS);
$table->setAttribute(Doctrine_Core::ATTR_QUERY_LIMIT, Doctrine_Core::LIMIT_RECORDS); // revert
```

In the following example we have users and phonenumbers with their relation being one-to-many. Now lets say we want fetch the first 20 users and all their related phonenumbers.

Now one might consider that adding a simple driver specific LIMIT 20 at the end of query would return the correct results. Thats wrong, since we you might get anything between 1-20 users as the first user might have 20 phonenumbers and then record set would consist of 20 rows.

DQL overcomes this problem with subqueries and with complex but efficient subquery analysis. In the next example we are going to fetch first 20 users and all their phonenumbers with single efficient query. Notice how the DQL parser is smart enough to use column aggregation inheritance even in the subquery and how it's smart enough to use different aliases for the tables in the subquery to avoid alias collisions.

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id, u.username, p.*')
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')
    ->limit(20);

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
u.username AS u__username,
p.id AS p__id,
p.phonenumber AS p__phonenumber,
p.user_id AS p__user_id
FROM user u
LEFT JOIN phonenumber p ON u.id = p.user_id
WHERE u.id IN (SELECT
DISTINCT u2.id
FROM user u2
LIMIT 20)
```

In the next example we are going to fetch first 20 users and all their phonenumbers and only those users that actually have phonenumbers with single efficient query, hence we use an `INNER JOIN`. Notice how the DQL parser is smart enough to use the `INNER JOIN` in the subquery:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id, u.username, p.*')
    ->from('User u')
    ->innerJoin('u.Phonenumbers p')
    ->limit(20);

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
u.username AS u__username,
p.id AS p__id,
p.phonenumber AS p__phonenumber,
p.user_id AS p__user_id
FROM user u
INNER JOIN phonenumber p ON u.id = p.user_id
WHERE u.id IN (SELECT
DISTINCT u2.id
FROM user u2
INNER JOIN phonenumber p2 ON u2.id = p2.user_id
LIMIT 20)
```

Named Queries

When you are dealing with a model that may change, but you need to keep your queries easily updated, you need to find an easy way to define queries. Imagine for example that you change one field and you need to follow all queries in your application to make sure it'll not break anything.

Named Queries is a nice and effective way to solve this situation, allowing you to create `Doctrine_Queries` and reuse them without the need to keep rewritting them.

The Named Query support is built at the top of `Doctrine_Query_Registry` support. `Doctrine_Query_Registry` is a class for registering and naming queries. It helps with the organization of your applications queries and along with that it offers some very nice convenience stuff.

The queries are added using the `add()` method of the registry object. It takes two parameters, the query name and the actual DQL query.

```
// test.php

// ...
$r = Doctrine_Manager::getInstance()->getQueryRegistry();

$r->add('User/all', 'FROM User u');

$userTable = Doctrine_Core::getTable('User');

// find all users
$users = $userTable->find('all');
```

To simplify this support, `Doctrine_Table` support some accessors to `Doctrine_Query_Registry`.

Creating a Named Query

When you build your models with option `generateTableClasses` defined as true, each record class will also generate a `*Table` class, extending from `Doctrine_Table`.

Then, you can implement the method `construct()` to include your Named Queries:

```
class UserTable extends Doctrine_Table
{
    public function construct()
    {
        // Named Query defined using DQL string
        $this->addNamedQuery('get.by.id', 'SELECT u.username FROM User u WHERE u.id = ?');

        // Named Query defined using Doctrine_Query object
        $this->addNamedQuery(
            'get.by.similar.usernames', Doctrine_Query::create()
                ->select('u.id, u.username')
                ->from('User u')
                ->where('LOWER(u.username) LIKE LOWER(?)')
        );
    }
}
```

Accessing Named Query

To reach the `MyFooTable` class, which is a subclass of `Doctrine_Table`, you can do the following:

```
$userTable = Doctrine_Core::getTable('User');
```

To access the Named Query (will return you a `Doctrine_Query` instance, always):

```
$q = $userTable->createNamedQuery('get.by.id');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id,
u.username AS u__username
FROM user u
WHERE u.id = ?
```

Executing a Named Query

There are two ways to execute a Named Query. The first one is by retrieving the `Doctrine_Query` and then executing it normally, as a normal instance:

```
// test.php

// ...
$users = Doctrine_Core::getTable('User')
    ->createNamedQuery('get.by.similar.usernames')
    ->execute(array('%jon%wage%'));
```

You can also simplify the execution, by doing:

```
// test.php

// ...
$users = Doctrine_Core::getTable('User')
    ->find('get.by.similar.usernames', array('%jon%wage%'));
```

The method `find()` also accepts a third parameter, which is the hydration mode.

Cross-Accessing Named Query

If that's not enough, Doctrine take advantage the `Doctrine_Query_Registry` and uses namespace queries to enable cross-access of Named Queries between objects. Suppose you have the `*Table` class instance of record `Article`. You want to call the "get.by.id" Named Query of record `User`. To access the Named Query, you have to do:

```
// test.php
// ...
$articleTable = Doctrine_Core::getTable('Article');
$users = $articleTable->find('User/get.by.id', array(1, 2, 3));
```

BNF

```
QL_statement ::= select_statement | update_statement | delete_statement
select_statement ::= select_clause from_clause [where_clause] [groupby_clause]
[having_clause] [orderby_clause]
update_statement ::= update_clause [where_clause]
delete_statement ::= delete_clause [where_clause]
from_clause ::=
FROM identification_variable_declaration
{, {identification_variable_declaration | collection_member_declaration}}*
identification_variable_declaration ::= range_variable_declaration { join | fetch_join }*
range_variable_declaration ::= abstract_schema_name [AS ] identification_variable
join ::= join_spec join_association_path_expression [AS ] identification_variable
fetch_join ::= join_specFETCH join_association_path_expression
association_path_expression ::=
collection_valued_path_expression | single_valued_association_path_expression
join_spec::= [LEFT [OUTER ] |INNER ]JOIN
join_association_path_expression ::= join_collection_valued_path_expression |
join_single_valued_association_path_expression
join_collection_valued_path_expression::=
identification_variable.collection_valued_association_field
join_single_valued_association_path_expression::=
identification_variable.single_valued_association_field
collection_member_declaration ::=
IN ( collection_valued_path_expression) [AS ] identification_variable
single_valued_path_expression ::=
state_field_path_expression | single_valued_association_path_expression
state_field_path_expression ::=
{identification_variable | single_valued_association_path_expression}.state_field
single_valued_association_path_expression ::=
identification_variable.{single_valued_association_field.}* single_valued_association_field
collection_valued_path_expression ::=
identification_variable.{single_valued_association_field.}*collection_valued_association_field
state_field ::= {embedded_class_state_field.}*simple_state_field
update_clause ::=UPDATE abstract_schema_name [[AS ] identification_variable]
SET update_item {, update_item}*
update_item ::= [identification_variable.]{state_field | single_valued_association_field} =
new_value
new_value ::=
simple_arithmetic_expression |
string_primary |
datetime_primary |

boolean_primary |
enum_primary
simple_entity_expression |
NULL
delete_clause ::=DELETE FROM abstract_schema_name [[AS ] identification_variable]
select_clause ::=SELECT [DISTINCT ] select_expression {, select_expression}*
select_expression ::=
single_valued_path_expression |
aggregate_expression |
identification_variable |
OBJECT( identification_variable) |
constructor_expression
constructor_expression ::=
NEW constructor_name( constructor_item {, constructor_item}*)
constructor_item ::= single_valued_path_expression | aggregate_expression
aggregate_expression ::=
{AVG |MAX |MIN |SUM }( [DISTINCT ] state_field_path_expression) |
COUNT ( [DISTINCT ] identification_variable | state_field_path_expression |
single_valued_association_path_expression)
where_clause ::=WHERE conditional_expression
groupby_clause ::=GROUP BY groupby_item {, groupby_item}*
groupby_item ::= single_valued_path_expression | identification_variable
having_clause ::=HAVING conditional_expression
orderby_clause ::=ORDER BY orderby_item {, orderby_item}*
orderby_item ::= state_field_path_expression [ASC |DESC ]
subquery ::= simple_select_clause subquery_from_clause [where_clause]
[groupby_clause] [having_clause]
subquery_from_clause ::=
FROM subselect_identification_variable_declaration
{, subselect_identification_variable_declaration}*
subselect_identification_variable_declaration ::=
identification_variable_declaration |
association_path_expression [AS ] identification_variable |
collection_member_declaration
simple_select_clause ::=SELECT [DISTINCT ] simple_select_expression
simple_select_expression::=
single_valued_path_expression |
aggregate_expression |
identification_variable
conditional_expression ::= conditional_term | conditional_expressionOR conditional_term
conditional_term ::= conditional_factor | conditional_termAND conditional_factor
conditional_factor ::= [NOT ] conditional_primary
conditional_primary ::= simple_cond_expression |( conditional_expression)
simple_cond_expression ::=
comparison_expression |
between_expression |
like_expression |
in_expression |
null_comparison_expression |
empty_collection_comparison_expression |

collection_member_expression |
exists_expression
between_expression ::=
arithmetic_expression [NOT ]BETWEEN
arithmetic_expressionAND arithmetic_expression |
string_expression [NOT ]BETWEEN string_expressionAND string_expression |
datetime_expression [NOT ]BETWEEN
datetime_expressionAND datetime_expression
in_expression ::=
state_field_path_expression [NOT ]IN ( in_item {, in_item}* | subquery)
in_item ::= literal | input_parameter
```

Magic Finders

Doctrine offers some magic finders for your Doctrine models that allow you to find a record by any column that is present in the model. This is helpful for simply finding a user by their username, or finding a group by the name of it. Normally this would require writing a `Doctrine_Query` instance and storing this somewhere so it can be reused. That is no longer needed for simple situations like that.

The basic pattern for the finder methods are as follows: `findBy%s($value)` or `findOneBy%s($value)`. The `%s` can be a column name or a relation alias. If you give a column name you must give the value you are looking for. If you specify a relationship alias, you can either pass an instance of the relation class to find, or give the actual primary key value.

First lets retrieve the `UserTable` instance to work with:

```
// test.php

// ...
$userTable = Doctrine_Core::getTable('User');
```

Now we can easily find a `User` record by its primary key by using the `find()` method:

```
// test.php

// ...
$user = $userTable->find(1);
```

Now if you want to find a single user by their username you can use the following magic finder:

```
// test.php

// ...
$user = $userTable->findOneByUsername('jonwage');
```

You can also easily find records by using the relationships between records. Because `User` has many `Phonenumbers` we can find those `Phonenumbers` by passing the `findBy**()` method a `User` instance:

```
// test.php

// ...
$phonenumberTable = Doctrine_Core::getTable('Phonenumber');
$phonenumbers = $phonenumberTable->findByUser($user);
```

The magic finders will even allow a little more complex finds. You can use the `And` and `Or` keywords in the method name to retrieve record by multiple properties.

```
$user = $userTable->findOneByUsernameAndPassword('jonwage', md5('changeme'));
```

You can even mix the conditions.

```
$users = $userTable->findByIsAdminAndIsModeratorOrIsSuperAdmin(true, true, true);
```

> These are very limited magic finders and it is always recommended to expand your queries to be manually written DQL queries. These methods are meant for only quickly accessing single records, no relationships, and are good for prototyping code quickly.

> The documented magic finders above are made possibly by using PHP's `__call()` overloading functionality. The undefined functions are forwarded to `Doctrine_Table::__call()` where the `Doctrine_Query` objects are built, executed and returned to the user.

Debugging Queries

The `Doctrine_Query` object has a few functions that can be used to help debug problems with the query:

Sometimes you may want to see the complete SQL string of your `Doctrine_Query` object:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u')
    ->orderBy('u.username');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
u.id AS u__id
FROM user u
ORDER BY u.username
```

> The SQL returned above by the `Doctrine_Query::getSql()` function does not replace the tokens with the parameters. This is the job of PDO and when we execute the query we pass the parameters to PDO where the replacement is executed. You can retrieve the array of parameters with the `Doctrine_Query::getParams()` method.

Get the array of parameters for the `Doctrine_Query` instance:

```
// test.php

// ...
print_r($q->getParams());
```

Conclusion

The Doctrine Query Language is by far one of the most advanced and helpful feature of Doctrine. It allows you to easily select very complex data from RDBMS relationships efficiently!

Now that we have gone over most of the major components of Doctrine and how to use them we are going to take a step back in the next chapter and look at everything from a birds eye view in the Component Overview chapter.

Component Overview

This chapter is intended to give you a birds eye view of all the main components that make up Doctrine and how they work together. We've discussed most of the components in the previous chapters but after this chapter you will have a better idea of all the components and what their jobs are.

Manager

The `Doctrine_Manager` class is a singleton and is the root of the configuration hierarchy and is used as a facade for controlling several aspects of Doctrine. You can retrieve the singleton instance with the following code.

```
// test.php

// ...
$manager = Doctrine_Manager::getInstance();
```

Retrieving Connections

```
// test.php

// ...
$connections = $manager->getConnections();
foreach ($connections as $connection) {
    echo $connection->getName() . "\n";
}
```

The `Doctrine_Manager` implements an iterator so you can simple loop over the $manager variable to loop over the connections.

```
// test.php

// ...
foreach ($manager as $connection) {
    echo $connection->getName() . "\n";
}
```

Connection

`Doctrine_Connection` is a wrapper for database connection. The connection is typically an instance of PDO but because of how Doctrine is designed, it is possible to design your own adapters that mimic the functionality that PDO provides.

The `Doctrine_Connection` class handles several things:

- Handles database portability things missing from PDO (eg. LIMIT / OFFSET emulation)
- Keeps track of `Doctrine_Table` objects
- Keeps track of records
- Keeps track of records that need to be updated / inserted / deleted
- Handles transactions and transaction nesting
- Handles the actual querying of the database in the case of INSERT / UPDATE / DELETE operations
- Can query the database using DQL. You will learn more about DQL in the DQL (Doctrine Query Language) chapter.
- Optionally validates transactions using Doctrine_Validator and gives full information of possible errors.

Available Drivers

Doctrine has drivers for every PDO-supported database. The supported databases are:

- FreeTDS / Microsoft SQL Server / Sybase
- Firebird/Interbase 6
- Informix
- Mysql
- Oracle
- Odbc

- PostgreSQL
- Sqlite

Creating Connections

```
// bootstrap.php

// ...
$conn = Doctrine_Manager::connection('mysql://username:password@localhost/test', 'connection 1');
```

> We have already created a new connection in the previous chapters. You can skip the above step and use the connection we've already created. You can retrieve it by using the `Doctrine_Manager::connection()` method.

Flushing the Connection

When you create new `User` records you can flush the connection and save all un-saved objects for that connection. Below is an example:

```
// test.php

// ...
$conn = Doctrine_Manager::connection();

$user1 = new User();
$user1->username = 'Jack';

$user2 = new User();
$user2->username = 'jwage';

$conn->flush();
```

Calling `Doctrine_Connection::flush()` will save all unsaved record instances for that connection. You could of course optionally call `save()` on each record instance and it would be the same thing.

```
// test.php

// ...
$user1->save();
$user2->save();
```

Table

`Doctrine_Table` holds the schema information specified by the given component (record). For example if you have a `User` class that extends `Doctrine_Record`, each schema definition call gets delegated to a unique table object that holds the information for later use.

Each `Doctrine_Table` is registered by `Doctrine_Connection`. You can retrieve the table object for each component easily which is demonstrated right below.

For example, lets say we want to retrieve the table object for the User class. We can do this by simply giving `User` as the first argument for the `Doctrine_Core::getTable()` method.

Getting a Table Object

In order to get table object for specified record just call `Doctrine_Record::getTable()`.

```
// test.php

// ...
$accountTable = Doctrine_Core::getTable('Account');
```

Getting Column Information

You can retrieve the column definitions set in `Doctrine_Record` by using the appropriate `Doctrine_Table` methods. If you need all information of all columns you can simply use:

```
// test.php

// ...
$columns = $accountTable->getColumns();

$columns = $accountTable->getColumns();
foreach ($columns as $column)
{
    print_r($column);
}
```

The above example would output the following when executed:

```
$ php test.php
Array
(
    [type] => integer
    [length] => 20
    [autoincrement] => 1
    [primary] => 1
)
Array
(
    [type] => string
    [length] => 255
)
Array
(
    [type] => decimal
    [length] => 18
)
```

Sometimes this can be an overkill. The following example shows how to retrieve the column names as an array:

```
// test.php

// ...
$names = $accountTable->getColumnNames();
print_r($names);
```

The above example would output the following when executed:

```
$ php test.php
Array
(
    [0] => id
    [1] => name
    [2] => amount
)
```

Getting Relation Information

You can also get an array of all the `Doctrine_Relation` objects by simply calling `Doctrine_Table::getRelations()` like the following:

```
// test.php

// ...
$userTable = Doctrine_Core::getTable('User');

$relations = $userTable->getRelations();

foreach ($relations as $name => $relation) {
    echo $name . ":\n";
    echo "Local - " . $relation->getLocal() . "\n";
    echo "Foreign - " . $relation->getForeign() . "\n\n";
}
```

The above example would output the following when executed:

```
$ php test.php
Email:
Local - id
Foreign - user_id

Phonenumbers:
Local - id
Foreign - user_id

Groups:
Local - user_id
Foreign - group_id

Friends:
Local - user1
Foreign - user2

Addresses:
Local - id
Foreign - user_id

Threads:
Local - id
Foreign - user_id
```

You can get the `Doctrine_Relation` object for an individual relationship by using the `Doctrine_Table::getRelation()` method.

```php
// test.php

// ...
$relation = $userTable->getRelation('Phonenumbers');

echo 'Name: ' . $relation['alias'] . "\n";
echo 'Local - ' . $relation['local'] . "\n";
echo 'Foreign - ' . $relation['foreign'] . "\n";
echo 'Relation Class - ' . get_class($relation);
```

The above example would output the following when executed:

```
$ php test.php
Name: Phonenumbers
Local - id
Foreign - user_id
Relation Class - Doctrine_Relation_ForeignKey
```

> Notice how in the above examples the `$relation` variable holds an instance of `Doctrine_Relation_ForeignKey` yet we can access it like an array. This is because, like many Doctrine classes, it implements `ArrayAccess`.

You can debug all the information of a relationship by using the `toArray()` method and using `print_r()` to inspect it.

```php
// test.php

// ...
$array  = $relation->toArray();
print_r($array);
```

Finder Methods

`Doctrine_Table` provides basic finder methods. These finder methods are very fast to write and should be used if you only need to fetch data from one database table. If you need queries that use several components (database tables) use `Doctrine_Connection::query()`.

You can easily find an individual user by its primary key by using the `find()` method:

```php
// test.php

// ...
$user = $userTable->find(2);
print_r($user->toArray());
```

The above example would output the following when executed:

```
$ php test.php
Array
(
    [id] => 2
    [is_active] => 1
    [is_super_admin] => 0
    [first_name] =>
    [last_name] =>
    [username] => jwage
    [password] =>
    [type] =>
    [created_at] => 2009-01-21 13:29:12
    [updated_at] => 2009-01-21 13:29:12
)
```

You can also use the `findAll()` method to retrieve a collection of all `User` records in the database:

```php
// test.php

// ...
foreach ($userTable->findAll() as $user) {
    echo $user->username . "\n";
}
```

The above example would output the following when executed:

```
$ php test.php
Jack
jwage
```

> The `findAll()` method is not recommended as it will return all records in the database and if you need to retrieve information from relationships it will lazily load that data causing high query counts. You can learn how to retrieve records and their related records efficiently by reading the DQL (Doctrine Query Language) chapter.

You can also retrieve a set of records with a DQL where condition by using the `findByDql()` method:

```
// test.php

// ...
$users = $userTable->findByDql('username LIKE ?', '%jw%');

foreach($users as $user) {
    echo $user->username . "\n";
}
```

The above example would output the following when executed:

```
$ php test.php
jwage
```

Doctrine also offers some additional magic finder methods that can be read about in the Magic Finders section of the DQL chapter.

> All of the finders below provided by `Doctrine_Table` use instances of `Doctrine_Query` for executing the queries. The objects are built dynamically internally and executed.

Custom Table Classes

Adding custom table classes is very easy. Only thing you need to do is name the classes as [componentName]Table and make them extend `Doctrine_Table`. So for the `User` model we would create a class like the following:

```
// models/UserTable.php

class UserTable extends Doctrine_Table
{
}
```

Custom Finders

You can add custom finder methods to your custom table object. These finder methods may use fast `Doctrine_Table` finder methods or DQL API (`Doctrine_Query::create()`).

```
// models/UserTable.php

class UserTable extends Doctrine_Table
{
    public function findByName($name)
    {
        return Doctrine_Query::create()
            ->from('User u')
            ->where('u.name LIKE ?', "%$name%")
            ->execute();
    }
}
```

Doctrine will check if a child `Doctrine_Table` class called `UserTable` exists when calling `getTable()` and if it does, it will return an instance of that instead of the default `Doctrine_Table`.

> In order for custom `Doctrine_Table` classes to be loaded you must enable the `autoload_table_classes` attribute in your `bootstrap.php` file like done below.

```
// boostrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_AUTOLOAD_TABLE_CLASSES, true);
```

Now when we ask for the `User` table object we will get the following:

```
$userTable = Doctrine_Core::getTable('User');

echo get_class($userTable); // UserTable

$users = $userTable->findByName("Jack");
```

> The above example where we add a `findByName()` method is made possible automatically by the magic finder methods. You can read about them in the Magic Finders section of the DQL chapter.

Record

Doctrine represents tables in your RDBMS with child `Doctrine_Record` classes. These classes are where you define your schema information, options, attributes, etc. Instances of these child classes represents records in the database and you can get and set properties on these objects.

Properties

Each assigned column property of `Doctrine_Record` represents a database table column. You will learn more about how to define your models in the Defining Models chapter.

Now accessing the columns is easy:

```
// test.php

// ...
$userTable = Doctrine_Core::getTable('User');
$user = $userTable->find(1);
```

Access property through overloading

```
// test.php

// ...
echo $user->username;
```

Access property with get()

```
// test.php

// ...
echo $user->get('username);
```

Access property with ArrayAccess

```
// test.php

// ...
echo $user['username'];
```

> The recommended way to access column values is by using the ArrayAccess as it makes it easy to switch between record and array fetching when needed.

Iterating through the properties of a record can be done in similar way as iterating through an array - by using the `foreach` construct. This is possible since `Doctrine_Record` implements a magic `IteratorAggregate` interface.

```
// test.php

// ...
foreach ($user as $field => $value) {
    echo $field . ': ' . $value . "\n";
}
```

As with arrays you can use the `isset()` for checking if given property exists and `unset()` for setting given property to null.

We can easily check if a property named 'name' exists in a if conditional:

```
// test.php

// ...
if (isset($user['username'])) {

}
```

If we want to unset the name property we can do it using the `unset()` function in php:

```php
// test.php

// ...
unset($user['username']);
```

When you have set values for record properties you can get an array of the modified fields and values using `Doctrine_Record::getModified()`

```php
// test.php

// ...
$user['username'] = 'Jack Daniels';

print_r($user->getModified());
```

The above example would output the following when executed:

```
$ php test.php
Array
(
    [username] => Jack Daniels
)
```

You can also simply check if a record is modified by using the `Doctrine_Record::isModified()` method:

```php
// test.php

// ...
echo $user->isModified() ? 'Modified':'Not Modified';
```

Sometimes you may want to retrieve the column count of given record. In order to do this you can simply pass the record as an argument for the `count()` function. This is possible since `Doctrine_Record` implements a magic Countable interface. The other way would be calling the `count()` method.

```php
// test.php

// ...
echo $record->count();
echo count($record);
```

`Doctrine_Record` offers a special method for accessing the identifier of given record. This method is called `identifier()` and it returns an array with identifier field names as keys and values as the associated property values.

```php
// test.php

// ...
$user['username'] = 'Jack Daniels';
$user->save();

print_r($user->identifier()); // array('id' => 1)
```

A common case is that you have an array of values which you need to assign to a given record. It may feel awkward and clumsy to set these values separately. No need to worry though, `Doctrine_Record` offers a way for merging a given array or record to another

The `merge()` method iterates through the properties of the given record or array and assigns the values to the object

```php
// test.php

// ...
$values = array(
    'username' => 'someone',
    'age'  => 11,
);

$user->merge($values);

echo $user->username; // someone
echo $user->age;  // 11
```

You can also merge a one records values in to another like the following:

```php
// test.php

// ...
$user1 = new User();
$user1->username = 'jwage';

$user2 = new User();
$user2->merge($user1);

echo $user2->username; // jwage
```

`Doctrine_Record` also has a `fromArray()` method which is identical to `merge()` and only exists for consistency with the `toArray()` method.

Updating Records

Updating objects is very easy, you just call the `Doctrine_Record::save()` method. The other way is to call `Doctrine_Connection::flush()` which saves all objects. It should be noted though that flushing is a much heavier operation than just calling save method.

```
// test.php

// ...
$userTable = Doctrine_Core::getTable('User');

$user = $userTable->find(2);

if ($user !== false) {
    $user->username = 'Jack Daniels';

    $user->save();
}
```

Sometimes you may want to do a direct update. In direct update the objects aren't loaded from database, rather the state of the database is directly updated. In the following example we use DQL UPDATE statement to update all users.

Run a query to make all user names lowercase:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->update('User u')
    ->set('u.username', 'LOWER(u.name)');

$q->execute();
```

You can also run an update using objects if you already know the identifier of the record. When you use the `Doctrine_Record::assignIdentifier()` method it sets the record identifier and changes the state so that calling `Doctrine_Record::save()` performs an update instead of insert.

```
// test.php

// ...
$user = new User();
$user->assignIdentifier(1);
$user->username = 'jwage';
$user->save();
```

Replacing Records

Replacing records is simple. If you instantiate a new object and save it and then late instantiate another new object with the same primary key or unique index value which already exists in the database, then it will replace/update that row in the database instead of inserting a new one. Below is an example.

First, imagine a `User` model where username is a unique index.

```
// test.php

// ...
$user = new User();
$user->username = 'jwage';
$user->password = 'changeme';
$user->save();
```

Issues the following query

```
INSERT INTO user (username, password) VALUES (?,?) ('jwage', 'changeme')
```

Now lets create another new object and set the same username but a different password.

```
// test.php

// ...
$user = new User();
$user->username = 'jwage';
$user->password = 'newpassword';
$user->replace();
```

Issues the following query

```
REPLACE INTO user (id,username,password) VALUES (?,?,?)    (null, 'jwage', 'newpassword')
```

The record is replaced/updated instead of a new one being inserted

Refreshing Records

Sometimes you may want to refresh your record with data from the database, use `Doctrine_Record::refresh()`.

```
// test.php

// ...
$user = Doctrine_Core::getTable('User')->find(2);
$user->username = 'New name';
```

Now if you use the `Doctrine_Record::refresh()` method it will select the data from the database again and update the properties of the instance.

```
// test.php

// ...
$user->refresh();
```

Refreshing relationships

The `Doctrine_Record::refresh()` method can also refresh the already loaded record relationships, but you need to specify them on the original query.

First lets retrieve a `User` with its associated `Groups`:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Groups')
    ->where('id = ?');

$user = $q->fetchOne(array(1));
```

Now lets retrieve a `Group` with its associated `Users`:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('Group g')
    ->leftJoin('g.Users')
    ->where('id = ?');

$group = $q->fetchOne(array(1));
```

Now lets link the retrieved `User` and `Group` through a `UserGroup` instance:

```
// test.php

// ...
$userGroup = new UserGroup();
$userGroup->user_id = $user->id;
$userGroup->group_id = $group->id;
$userGroup->save();
```

You can also link a `User` to a `Group` in a much simpler way, by simply adding the `Group` to the `User`. Doctrine will take care of creating the `UserGroup` instance for you automatically:

```
// test.php

// ...
$user->Groups[] = $group;
$user->save()
```

Now if we call `Doctrine_Record::refresh(true)` it will refresh the record and its relationships loading the newly created reference we made above:

```
// test.php

// ...
$user->refresh(true);
$group->refresh(true);
```

You can also lazily refresh all defined relationships of a model using `Doctrine_Record::refreshRelated()`:

```
// test.php

// ...
$user = Doctrine_Core::getTable('User')->findOneByName('jon');
$user->refreshRelated();
```

If you want to refresh an individual specified relationship just pass the name of a relationship to the `refreshRelated()` function and it will lazily load the relationship:

```
// test.php

// ...
$user->refreshRelated('Phonenumber');
```

Deleting Records

Deleting records in Doctrine is handled by `Doctrine_Record::delete()`, `Doctrine_Collection::delete()` and `Doctrine_Connection::delete()` methods.

```
// test.php

// ...
$userTable = Doctrine_Core::getTable("User");

$user = $userTable->find(2);

// deletes user and all related composite objects
if($user !== false) {
    $user->delete();
}
```

If you have a `Doctrine_Collection` of `User` records you can call `delete()` and it will loop over all records calling `Doctrine_Record::delete()` for you.

```
// test.php

// ...
$users = $userTable->findAll();
```

Now you can delete all users and their related composite objects by calling `Doctrine_Collection::delete()`. It will loop over all `Users` in the collection calling delete one each one:

```
// test.php

// ...
$users->delete();
```

Using Expression Values

There might be situations where you need to use SQL expressions as values of columns. This can be achieved by using `Doctrine_Expression` which converts portable DQL expressions to your native SQL expressions.

Lets say we have a class called event with columns `timepoint(datetime)` and `name(string)`. Saving the record with the current timestamp can be achieved as follows:

```
// test.php

// ...
$user = new User();
$user->username = 'jwage';
$user->updated_at = new Doctrine_Expression('NOW()');
$user->save();
```

The above code would issue the following SQL query:

```
INSERT INTO user (username, updated_at_) VALUES ('jwage', NOW())
```

> When you use `Doctrine_Expression` with your objects in order to get the updated value you will have to manually call `refresh()` to get the updated value from the database.

```php
// test.php

// ...
$user->refresh();
```

Getting Record State

Every `Doctrine_Record` has a state. First of all records can be transient or persistent. Every record that is retrieved from database is persistent and every newly created record is considered transient. If a `Doctrine_Record` is retrieved from database but the only loaded property is its primary key, then this record has a state called proxy.

Every transient and persistent `Doctrine_Record` is either clean or dirty. `Doctrine_Record` is clean when none of its properties are changed and dirty when at least one of its properties has changed.

A record can also have a state called locked. In order to avoid infinite recursion in some rare circular reference cases Doctrine uses this state internally to indicate that a record is currently under a manipulation operation.

Below is a table containing all the different states a record can be in with a short description of it:

Name	Description
`Doctrine_Record::STATE_PROXY`	Record is in proxy state meaning its persistent but not all of its properties are loaded from the database.
`Doctrine_Record::STATE_TCLEAN`	Record is transient clean, meaning its transient and none of its properties are changed.
`Doctrine_Record::STATE_TDIRTY`	Record is transient dirty, meaning its transient and some of its properties are changed.
`Doctrine_Record::STATE_DIRTY`	Record is dirty, meaning its persistent and some of its properties are changed.
`Doctrine_Record::STATE_CLEAN`	Record is clean, meaning its persistent and none of its properties are changed.
`Doctrine_Record::STATE_LOCKED`	Record is locked.

You can easily get the state of a record by using the `Doctrine_Record::state()` method:

```php
// test.php

// ...
$user = new User();

if ($user->state() == Doctrine_Record::STATE_TDIRTY) {
    echo 'Record is transient dirty';
}
```

> The above object is `TDIRTY` because it has some default values specified in the schema. If we use an object that has no default values and instantiate a new instance it will return `TCLEAN`.

```php
// test.php

// ...
$account = new Account();

if ($account->state() == Doctrine_Record::STATE_TCLEAN) {
    echo 'Record is transient clean';
}
```

Getting Object Copy

Sometimes you may want to get a copy of your object (a new object with all properties copied). Doctrine provides a simple method for this: `Doctrine_Record::copy()`.

```php
// test.php

// ...
$copy = $user->copy();
```

Notice that copying the record with `copy()` returns a new record (state `TDIRTY`) with the values of the old record, and it copies the relations of that record. If you do not want to copy the relations too, you need to use `copy(false)`.

Get a copy of user without the relations

```php
// test.php

// ...
$copy = $user->copy(false);
```

Using the PHP `clone` functionality simply uses this `copy()` functionality internally:

```
// test.php

// ...
$copy = clone $user;
```

Saving a Blank Record

By default Doctrine doesn't execute when `save()` is being called on an unmodified record. There might be situations where you want to force-insert the record even if it has not been modified. This can be achieved by assigning the state of the record to `Doctrine_Record::STATE_TDIRTY`.

```
// test.php

// ...
$user = new User();
$user->state('TDIRTY');
$user->save();
```

> When setting the state you can optionally pass a string for the state and it will be converted to the appropriate state constant. In the example above, `TDIRTY` is actually converted to `Doctrine_Record::STATE_TDIRTY`.

Mapping Custom Values

There might be situations where you want to map custom values to records. For example values that depend on some outer sources and you only want these values to be available at runtime not persisting those values into database. This can be achieved as follows:

```
// test.php

// ...
$user->mapValue('isRegistered', true);

$user->isRegistered; // true
```

Serializing

Sometimes you may want to serialize your record objects (possibly for caching purposes):

```
// test.php

// ...
$string = serialize($user);

$user = unserialize($string);
```

Checking Existence

Very commonly you'll need to know if given record exists in the database. You can use the `exists()` method for checking if given record has a database row equivalent:

```
// test.php

// ...
$record = new User();

echo $record->exists() ? 'Exists':'Does Not Exist'; // Does Not Exist

$record->username = 'someone';
$record->save();

echo $record->exists() ? 'Exists':'Does Not Exist'; // Exists
```

Function Callbacks for Columns

`Doctrine_Record` offers a way for attaching callback calls for column values. For example if you want to trim certain column, you can simply use:

```
// test.php

// ...
$record->call('trim', 'username');
```

Collection

`Doctrine_Collection` is a collection of records (see Doctrine_Record). As with records the collections can be deleted and saved using `Doctrine_Collection::delete()` and `Doctrine_Collection::save()` accordingly.

When fetching data from database with either DQL API (see `Doctrine_Query`) or rawSql API (see `Doctrine_RawSql`) the methods return an instance of `Doctrine_Collection` by default.

The following example shows how to initialize a new collection:

```
// test.php

// ...
$users = new Doctrine_Collection('User');
```

Now add some new data to the collection:

```
// test.php

// ...
$users[0]->username = 'Arnold';
$users[1]->username = 'Somebody';
```

Now just like we can delete a collection we can save it:

```
$users->save();
```

Accessing Elements

You can access the elements of `Doctrine_Collection` with `set()` and `get()` methods or with ArrayAccess interface.

```
// test.php

// ...
$userTable = Doctrine_Core::getTable('User');
$users = $userTable->findAll();
```

Accessing elements with ArrayAccess interface

```
// test.php

// ...
$users[0]->username = "Jack Daniels";
$users[1]->username = "John Locke";
```

Accessing elements with get()

```
echo $users->get(1)->username;
```

Adding new Elements

When accessing single elements of the collection and those elements (records) don't exist Doctrine auto-adds them.

In the following example we fetch all users from database (there are 5) and then add couple of users in the collection.

As with PHP arrays the indexes start from zero.

```
// test.php

// ...
$users = $userTable->findAll();

echo count($users); // 5

$users[5]->username = "new user 1";
$users[6]->username = "new user 2";
```

You could also optionally omit the 5 and 6 from the array index and it will automatically increment just as a PHP array would:

```
// test.php

// ...
$users[]->username = 'new user 3'; // key is 7
$users[]->username = 'new user 4'; // key is 8
```

Getting Collection Count

The `Doctrine_Collection::count()` method returns the number of elements currently in the collection.

```
// test.php

// ...
$users = $userTable->findAll();

echo $users->count();
```

Since `Doctrine_Collection` implements Countable interface a valid alternative for the previous example is to simply pass the collection as an argument for the count() function.

```
// test.php

// ...
echo count($users);
```

Saving the Collection

Similar to `Doctrine_Record` the collection can be saved by calling the `save()` method. When `save()` gets called Doctrine issues `save()` operations an all records and wraps the whole procedure in a transaction.

```
// test.php

// ...
$users = $userTable->findAll();

$users[0]->username = 'Jack Daniels';

$users[1]->username = 'John Locke';

$users->save();
```

Deleting the Collection

Doctrine Collections can be deleted in very same way is Doctrine Records you just call `delete()` method. As for all collections Doctrine knows how to perform single-shot-delete meaning it only performs one database query for the each collection.

For example if we have collection of users. When deleting the collection of users doctrine only performs one query for this whole transaction. The query would look something like:

```
DELETE
FROM user
WHERE id IN (1,2,3,
... ,N)
```

Key Mapping

Sometimes you may not want to use normal indexing for collection elements. For example in some cases mapping primary keys as collection keys might be useful. The following example demonstrates how this can be achieved.

Map the `id` column

```
// test.php

// ....
$userTable = Doctrine_Core::getTable('User');

$userTable->setAttribute(Doctrine_Core::ATTR_COLL_KEY, 'id');
```

Now user collections will use the values of id column as element indexes:

```
// test.php

// ...
$users = $userTable->findAll();

foreach($users as $id => $user) {
    echo $id . $user->username;
}
```

You may want to map the `name` column:

```php
// test.php

// ...
$userTable = Doctrine_Core::getTable('User');

$userTable->setAttribute(Doctrine_Core::ATTR_COLL_KEY, 'username');
```

Now user collections will use the values of `name` column as element indexes:

```php
// test.php

// ...
$users = $userTable->findAll();

foreach($users as $username => $user) {
    echo $username . ' - ' . $user->created_at . "\n";
}
```

> Note this would only be advisable if the `username` column is specified as unique in your schema otherwise you will have cases where data cannot be hydrated properly due to duplicate collection keys.

Loading Related Records

Doctrine provides means for efficiently retrieving all related records for all record elements. That means when you have for example a collection of users you can load all phonenumbers for all users by simple calling the `loadRelated()` method.

However, in most cases you don't need to load related elements explicitly, rather what you should do is try to load everything at once by using the DQL API and JOINS.

The following example uses three queries for retrieving users, their phonenumbers and the groups they belong to.

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u');

$users = $q->execute();
```

Now lets load phonenumbers for all users:

```php
// test.php

// ...
$users->loadRelated('Phonenumbers');

foreach($users as $user) {
    echo $user->Phonenumbers[0]->phonenumber;
    // no additional db queries needed here
}
```

The `loadRelated()` works an any relation, even associations:

```php
// test.php

// ...
$users->loadRelated('Groups');

foreach($users as $user) {
    echo $user->Groups[0]->name;
}
```

The example below shows how to do this more efficiently by using the DQL API.

Write a `Doctrine_Query` that loads everything in one query:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Phonenumbers p')
    ->leftJoin('u.Groups g');

$users = $q->execute();
```

Now when we use the `Phonenumbers` and `Groups` no additional database queries are needed:

```php
// test.php

// ...
foreach($users as $user) {
    echo $user->Phonenumbers[0]->phonenumber;
    echo $user->Groups[0]->name;
}
```

Validator

Validation in Doctrine is a way to enforce your business rules in the model part of the MVC architecture. You can think of this validation as a gateway that needs to be passed right before data gets into the persistent data store. The definition of these business rules takes place at the record level, that means in your active record model classes (classes derived from `Doctrine_Record`). The first thing you need to do to be able to use this kind of validation is to enable it globally. This is done through the `Doctrine_Manager`.

```php
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_VALIDATE, Doctrine_Core::VALIDATE_ALL);
```

Once you enabled validation, you'll get a bunch of validations automatically:

- Data type validations: All values assigned to columns are checked for the right type. That means if you specified a column of your record as type 'integer', Doctrine will validate that any values assigned to that column are of this type. This kind of type validation tries to be as smart as possible since PHP is a loosely typed language. For example 2 as well as "7" are both valid integers whilst "3f" is not. Type validations occur on every column (since every column definition needs a type).
- Length validation: As the name implies, all values assigned to columns are validated to make sure that the value does not exceed the maximum length.

You can combine the following constants by using bitwise operations: `VALIDATE_ALL`, `VALIDATE_TYPES`, `VALIDATE_LENGTHS`, `VALIDATE_CONSTRAINTS`, `VALIDATE_NONE`.

For example to enable all validations except length validations you would use:

```php
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_VALIDATE, VALIDATE_ALL & ~VALIDATE_LENGTHS);
```

You can read more about this topic in the Data Validation chapter.

More Validation

The type and length validations are handy but most of the time they're not enough. Therefore Doctrine provides some mechanisms that can be used to validate your data in more detail.

Validators are an easy way to specify further validations. Doctrine has a lot of predefined validators that are frequently needed such as `email`, `country`, `ip`, `range` and `regexp` validators. You find a full list of available validators in the Data Validation chapter. You can specify which validators apply to which column through the 4th argument of the `hasColumn()` method. If that is still not enough and you need some specialized validation that is not yet available as a predefined validator you have three options:

- You can write the validator on your own.
- You can propose your need for a new validator to a Doctrine developer.
- You can use validation hooks.

The first two options are advisable if it is likely that the validation is of general use and is potentially applicable in many situations. In that case it is a good idea to implement a new validator. However if the validation is special it is better to use hooks provided by Doctrine:

- `validate()` (Executed every time the record gets validated)
- `validateOnInsert()` (Executed when the record is new and gets validated)
- `validateOnUpdate()` (Executed when the record is not new and gets validated)

If you need a special validation in your active record you can simply override one of these methods in your active record class (a descendant of `Doctrine_Record`). Within these methods you can use all the power of PHP to validate your fields. When a field does not pass your validation you can then add errors to the record's error stack. The following code snippet shows an example of how to define validators together with custom validation:

```php
// models/User.php

class User extends BaseUser
{
    protected function validate()
    {
        if ($this->username == 'God') {
            // Blasphemy! Stop that! ;-)
            // syntax: add(<fieldName>, <error code/identifier>)
            $errorStack = $this->getErrorStack();
            $errorStack->add('name', 'You cannot use this username!');
        }
    }
}

// models/Email.php

class Email extends BaseEmail
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        // validators 'email' and 'unique' used
        $this->hasColumn('address','string', 150, array('email', 'unique'));
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
Email:
  columns:
    address:
      type: string(150)
      email: true
      unique: true
```

Valid or Not Valid

Now that you know how to specify your business rules in your models, it is time to look at how to deal with these rules in the rest of your application.

Implicit Validation

Whenever a record is going to be saved to the persistent data store (i.e. through calling `$record->save()`) the full validation procedure is executed. If errors occur during that process an exception of the type `Doctrine_Validator_Exception` will be thrown. You can catch that exception and analyze the errors by using the instance method `Doctrine_Validator_Exception::getInvalidRecords()`. This method returns an ordinary array with references to all records that did not pass validation. You can then further explore the errors of each record by analyzing the error stack of each record. The error stack of a record can be obtained with the instance method `Doctrine_Record::getErrorStack()`. Each error stack is an instance of the class `Doctrine_Validator_ErrorStack`. The error stack provides an easy to use interface to inspect the errors.

Explicit Validation

You can explicitly trigger the validation for any record at any time. For this purpose `Doctrine_Record` provides the instance method `Doctrine_Record::isValid()`. This method returns a boolean value indicating the result of the validation. If the method returns false, you can inspect the error stack in the same way as seen above except that no exception is thrown, so you simply obtain the error stack of the record that didnt pass validation through `Doctrine_Record::getErrorStack()`.

The following code snippet shows an example of handling implicit validation which caused a `Doctrine_Validator_Exception`.

```php
// test.php

// ...
$user = new User();

try {
    $user->username = str_repeat('t', 256);
    $user->Email->address = "drink@@notvalid..";
    $user->save();
} catch(Doctrine_Validator_Exception $e) {
    $userErrors = $user->getErrorStack();
    $emailErrors = $user->Email->getErrorStack();

    foreach($userErrors as $fieldName => $errorCodes) {
        echo $fieldName . " - " . implode(', ', $errorCodes) . "\n";
    }

    foreach($emailErrors as $fieldName => $errorCodes) {
        echo $fieldName . " - " . implode(', ', $errorCodes) . "\n";
    }
}
```

> You could also use `$e->getInvalidRecords()`. The direct way used above is just more simple when you know the records you're dealing with.

You can also retrieve the error stack as a nicely formatted string for easy use in your applications:

```php
// test.php

// ...
echo $user->getErrorStackAsString();
```

It would output an error string that looks something like the following:

```
Validation failed in class User

  1 field had validation error:

    * 1 validator failed on username (length)
```

Profiler

`Doctrine_Connection_Profiler` is an event listener for `Doctrine_Connection`. It provides flexible query profiling. Besides the SQL strings the query profiles include elapsed time to run the queries. This allows inspection of the queries that have been performed without the need for adding extra debugging code to model classes.

`Doctrine_Connection_Profiler` can be enabled by adding it as an event listener for Doctrine_Connection.

```php
// test.php

// ...
$profiler = new Doctrine_Connection_Profiler();

$conn = Doctrine_Manager::connection();
$conn->setListener($profiler);
```

Basic Usage

Perhaps some of your pages is loading slowly. The following shows how to build a complete profiler report from the connection:

```php
// test.php

// ...
$time = 0;
foreach ($profiler as $event) {
    $time += $event->getElapsedSecs();
    echo $event->getName() . " " . sprintf("%f", $event->getElapsedSecs()) . "\n";
    echo $event->getQuery() . "\n";
    $params = $event->getParams();
    if( ! empty($params)) {
        print_r($params);
    }
}
echo "Total time: " . $time  . "\n";
```

> Frameworks like symfony, Zend, etc. offer web debug toolbars that use this functionality provided by Doctrine for reporting the number of queries executed on every page as well as the time it takes for each query.

Locking Manager

> The term 'Transaction' does not refer to database transactions here but to the general meaning of this term.

Locking is a mechanism to control concurrency. The two most well known locking strategies are optimistic and pessimistic locking. The following is a short description of these two strategies from which only pessimistic locking is currently supported by Doctrine.

Optimistic Locking

The state/version of the object(s) is noted when the transaction begins. When the transaction finishes the noted state/version of the participating objects is compared to the current state/version. When the states/versions differ the objects have been modified by another transaction and the current transaction should fail. This approach is called 'optimistic' because it is assumed that it is unlikely that several users will participate in transactions on the same objects at the same time.

Pessimistic Locking

The objects that need to participate in the transaction are locked at the moment the user starts the transaction. No other user can start a transaction that operates on these objects while the locks are active. This ensures that the user who starts the transaction can be sure that no one else modifies the same objects until he has finished his work.

Doctrine's pessimistic offline locking capabilities can be used to control concurrency during actions or procedures that take several HTTP request and response cycles and/or a lot of time to complete.

Examples

The following code snippet demonstrates the use of Doctrine's pessimistic offline locking capabilities.

At the page where the lock is requested get a locking manager instance:

```
// test.php

// ...
$lockingManager = new Doctrine_Locking_Manager_Pessimistic();
```

> Ensure that old locks which timed out are released before we try to acquire our lock 300 seconds = 5 minutes timeout. This can be done by using the `releaseAgedLocks()` method.

```
// test.php

// ...
$user = Doctrine_Core::getTable('User')->find(1);

try
{
    $lockingManager->releaseAgedLocks(300);

    $gotLock = $lockingManager->getLock($user, 'jwage');

    if ($gotLock)
    {
        echo "Got lock!";
    }
    else
    {
        echo "Sorry, someone else is currently working on this record";
    }
} catch(Doctrine_Locking_Exception $dle) {
    echo $dle->getMessage();
    // handle the error
}
```

At the page where the transaction finishes get a locking manager instance:

```php
// test.php

// ...
$user = Doctrine_Core::getTable('User')->find(1);

$lockingManager = new Doctrine_Locking_Manager_Pessimistic();

try
{
    if ($lockingManager->releaseLock($user, 'jwage'))
    {
        echo "Lock released";
    }
    else
    {
        echo "Record was not locked. No locks released.";
    }
}
catch(Doctrine_Locking_Exception $dle)
{
    echo $dle->getMessage();
    // handle the error
}
```

Technical Details

The pessimistic offline locking manager stores the locks in the database (therefore 'offline'). The required locking table is automatically created when you try to instantiate an instance of the manager and the `ATTR_CREATE_TABLES` is set to TRUE. This behavior may change in the future to provide a centralized and consistent table creation procedure for installation purposes.

Views

Database views can greatly increase the performance of complex queries. You can think of them as cached queries. `Doctrine_View` provides integration between database views and DQL queries.

Using Views

Using views on your database using Doctrine is easy. We provide a nice `Doctrine_View` class which provides functionality for creating, dropping and executing views.

The `Doctrine_View` class integrates with the `Doctrine_Query` class by saving the SQL that would be executed by `Doctrine_Query`.

First lets create a new `Doctrine_Query` instance to work with:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Phonenumber p')
    ->limit(20);
```

Now lets create the `Doctrine_View` instance and pass it the `Doctrine_Query` instance as well as a `name` for identifying that database view:

```php
// test.php

// ...
$view  = new Doctrine_View($q, 'RetrieveUsersAndPhonenumbers');
```

Now we can easily create the view by using the `Doctrine_View::create()` method:

```php
// test.php

// ...
try {
    $view->create();
} catch (Exception $e) {}
```

Alternatively if you want to drop the database view you use the `Doctrine_View::drop()` method:

```php
// test.php

// ...
try {
    $view->drop();
} catch (Exception $e) {}
```

Using views are extremely easy. Just use the `Doctrine_View::execute()` for executing the view and returning the results just as a normal `Doctrine_Query` object would:

```
// test.php

// ...
$users  = $view->execute();

foreach ($users as $user) {
    print_r($us->toArray());
}
```

Conclusion

We now have been exposed to a very large percentage of the core functionality provided by Doctrine. The next chapters of this book are documentation that cover some of the optional functionality that can help make your life easier on a day to day basis.

Lets move on to the next chapter where we can learn about how to use native SQL to hydrate our data in to arrays and objects instead of the Doctrine Query Language.

Native SQL

Introduction

`Doctrine_RawSql` provides a convenient interface for building raw sql queries. Similar to `Doctrine_Query`, `Doctrine_RawSql` provides means for fetching arrays and objects. Whichever way you prefer.

Using raw sql for fetching might be useful when you want to utilize database specific features such as query hints or the `CONNECT` keyword in Oracle.

Creating a `Doctrine_RawSql` object is easy:

```
// test.php

// ...
$q = new Doctrine_RawSql();
```

Optionally a connection parameter can be given and it accepts an instance of `Doctrine_Connection`. You learned how to create connections in the Connections chapter.

```
// test.php

// ...
$conn = Doctrine_Manager::connection();
$q = new Doctrine_RawSql($conn);
```

Component Queries

The first thing to notice when using `Doctrine_RawSql` is that you always have to place the fields you are selecting in curly brackets {}. Also for every selected component you have to call `addComponent()`.

The following example should clarify the usage of these:

```
// test.php

// ...
$q->select('{u.*}')
  ->from('user u')
  ->addComponent('u', 'User');

$users = $q->execute();
print_r($users->toArray());
```

> Note above that we tell that `user` table is bound to class called `User` by using the `addComponent()` method.

Pay attention to following things:

- Fields must be in curly brackets.
- For every selected table there must be one `addComponent()` call.

Fetching from Multiple Components

When fetching from multiple components the `addComponent()` calls become a bit more complicated as not only do we have to tell which tables are bound to which components, we also have to tell the parser which components belongs to which.

In the following example we fetch all `users` and their `phonenumbers`. First create a new `Doctrine_RawSql` object and add the select parts:

```
// test.php

// ...
$q = new Doctrine_RawSql();
$q->select('{u.*}, {p.*}');
```

Now we need to add the `FROM` part to the query with the join to the phonenumber table from the user table and map everything together:

```php
// test.php

// ...
$q->from('user u LEFT JOIN phonenumber p ON u.id = p.user_id')
```

Now we tell that `user` table is bound to class called `User` we also add an alias for `User` class called `u`. This alias will be used when referencing the `User` class.

```php
// test.php

// ...
$q->addComponent('u', 'User u');
```

Now we add another component that is bound to table `phonenumber`:

```php
// test.php

// ...
$q->addComponent('p', 'u.Phonenumbers p');
```

> Notice how we reference that the `Phonenumber` class is the User's phonenumber.

Now we can execute the `Doctrine_RawSql` query just like if you were executing a `Doctrine_Query` object:

```php
// test.php

// ...
$users = $q->execute();
echo get_class($users) . "\n";
echo get_class($users[0]) . "\n";
echo get_class($users[0]['Phonenumbers'][0]) . "\n";
```

The above example would output the following when executed:

```
$ php test.php
Doctrine_Collection
User
Phonenumber
```

Conclusion

This chapter may or may not be useful for you right now. In most cases the Doctrine Query Language is plenty sufficient for retrieving the complex data sets you require. But if you require something outside the scope of what `Doctrine_Query` is capable of then `Doctrine_RawSql` can help you.

In the previous chapters you've seen a lot of mention about YAML schema files and have been given examples of schema files but haven't really been trained on how to write your own. The next chapter explains in great detail how to maintain your models as YAML Schema Files.

YAML Schema Files

Introduction

The purpose of schema files is to allow you to manage your model definitions directly from a YAML file rather then editing php code. The YAML schema file is parsed and used to generate all your model definitions/classes. This makes Doctrine model definitions much more portable.

Schema files support all the normal things you would write with manual php code. Component to connection binding, relationships, attributes, templates/behaviors, indexes, etc.

Abbreviated Syntax

Doctrine offers the ability to specify schema in an abbreviated syntax. A lot of the schema parameters have values they default to, this allows us to abbreviate the syntax and let Doctrine just use its defaults. Below is an example of schema taking advantage of all the abbreviations.

> The `detect_relations` option will attempt to guess relationships based on column names. In the example below Doctrine knows that `User` has one `Contact` and will automatically define the relationship between the models.

```yaml
---
detect_relations: true

User:
  columns:
    username: string
    password: string
    contact_id: integer

Contact:
  columns:
    first_name: string
    last_name: string
    phone: string
    email: string
    address: string
```

Verbose Syntax

Here is the 100% verbose form of the above schema:

```yaml
---
User:
  columns:
    username:
      type: string(255)
    password:
      type: string(255)
    contact_id:
      type: integer
  relations:
    Contact:
      class: Contact
      local: contact_id
      foreign: id
      foreignAlias: User
      foreignType: one
      type: one

Contact:
  columns:
    first_name:
      type: string(255)
    last_name:
      type: string(255)
    phone:
      type: string(255)
    email:
      type: string(255)
    address:
      type: string(255)
  relations:
    User:
      class: User
      local: id
      foreign: contact_id
      foreignAlias: Contact
      foreignType: one
      type: one
```

In the above example we do not define the `detect_relations` option, instead we manually define the relationships so we have complete control over the configuration of the local/foreign key, type and alias of the relationship on each side.

Relationships

When specifying relationships it is only necessary to specify the relationship on the end where the foreign key exists. When the schema file is parsed, it reflects the relationship and builds the opposite end automatically. If you specify the other end of the relationship manually, the auto generation will have no effect.

Detect Relations

Doctrine offers the ability to specify a `detect_relations` option as you saw earlier. This feature provides automatic relationship building based on column names. If you have a `User` model with a `contact_id` and a class with the name `Contact` exists, it will automatically create the relationships between the two.

Customizing Relationships

Doctrine only requires that you specify the relationship on the end where the foreign key exists. The opposite end of the relationship will be reflected and built on the opposite end. The schema syntax offers the ability to customize the relationship alias and type of the opposite end. This is good news because it means you can maintain all the relevant relationship information in one place. Below is an example of how to customize the alias and type of the opposite end of the relationship. It demonstrates the relationships `User` has one `Contact` and `Contact` has one `User` as `UserModel`. Normally it would have automatically generated `User` has one `Contact` and `Contact` has many `User`. The `foreignType` and `foreignAlias` options allow you to customize the opposite end of the relationship.

```
---
User:
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    contact_id:
      type: integer(4)
    username:
      type: string(255)
    password:
      type: string(255)
  relations:
    Contact:
      foreignType: one
      foreignAlias: UserModel

Contact:
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    name:
      type: string(255)
```

You can quickly detect and create the relationships between two models with the detect_relations option like below.

```
---
detect_relations: true

User:
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    avatar_id:
      type: integer(4)
    username:
      type: string(255)
    password:
      type: string(255)

Avatar:
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    name:
      type: string(255)
    image_file:
      type: string(255)
```

The resulting relationships would be `User` has one `Avatar` and `Avatar` has many `User`.

One to One

```yaml
---
User:
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    contact_id:
      type: integer(4)
    username:
      type: string(255)
    password:
      type: string(255)
  relations:
    Contact:
      foreignType: one

Contact:
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    name:
      type: string(255)
```

One to Many

```yaml
---
User:
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    contact_id:
      type: integer(4)
    username:
      type: string(255)
    password:
      type: string(255)

Phonenumber:
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    name:
      type: string(255)
    user_id:
      type: integer(4)
  relations:
    User:
      foreignAlias: Phonenumbers
```

Many to Many

```yaml
---
User:
  columns:
    id:
      type: integer(4)
      autoincrement: true
      primary: true
    username:
      type: string(255)
    password:
      type: string(255)
  attributes:
    export: all
    validate: true

Group:
  tableName: group_table
  columns:
    id:
      type: integer(4)
      autoincrement: true
      primary: true
    name:
      type: string(255)
  relations:
    Users:
      foreignAlias: Groups
      class: User
      refClass: GroupUser

GroupUser:
  columns:
    group_id:
      type: integer(4)
      primary: true
    user_id:
      type: integer(4)
      primary: true
  relations:
    Group:
      foreignAlias: GroupUsers
    User:
      foreignAlias: GroupUsers
```

This creates a set of models where `User` has many `Groups`, `Group` has many `Users`, `GroupUser` has one `User` and `GroupUser` has one `Group`.

Features & Examples

Connection Binding

If you're not using schema files to manage your models, you will normally use this code to bind a component to a connection name with the following code:

Create a connection with code like below:

```
Doctrine_Manager::connection('mysql://jwage:pass@localhost/connection1', 'connection1');
```

Now somewhere in your Doctrine bootstrapping of Doctrine you would bind the model to that connection:

```
Doctrine_Manager::connection()->bindComponent('User', 'conn1');
```

Schema files offer the ability to bind it to a specific connection by specifying the connection parameter. If you do not specify the connection the model will just use the current connection set on the `Doctrine_Manager` instance.

```yaml
---
User:
  connection: connection1
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    contact_id:
      type: integer(4)
    username:
      type: string(255)
    password:
      type: string(255)
```

Attributes

Doctrine offers the ability to set attributes for your generated models directly in your schema files similar to how you would if you were manually writing your Doctrine_Record child classes.

```yaml
---
User:
  connection: connection1
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    contact_id:
      type: integer(4)
    username:
      type: string(255)
    password:
      type: string(255)
  attributes:
    export: none
    validate: false
```

Enums

To use enum columns in your schema file you must specify the type as enum and specify an array of values for the possible enum values.

```yaml
---
TvListing:
 tableName: tv_listing
 actAs: [Timestampable]
 columns:
   notes:
     type: string
   taping:
     type: enum
     length: 4
     values: ['live', 'tape']
   region:
     type: enum
     length: 4
     values: ['US', 'CA']
```

ActAs Behaviors

You can attach behaviors to your models with the `actAs` option. You can specify something like the following:

```yaml
---
User:
  connection: connection1
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    contact_id:
      type: integer(4)
    username:
      type: string(255)
    password:
      type: string(255)
  actAs:
    Timestampable:
    Sluggable:
      fields: [username]
      name: slug    # defaults to 'slug'
      type: string  # defaults to 'clob'
      length: 255   # defaults to null. clob doesn't require a length
```

> The options specified on the Sluggable behavior above are optional as they will use defaults values if you do not specify anything. Since they are defaults it is not necessary to type it out all the time.

```yaml
---
User:
  connection: connection1
  columns:
# ...
  actAs: [Timestampable, Sluggable]
```

Listeners

If you have a listener you'd like attached to a model, you can specify them directly in the yml as well.

```yaml
---
User:
  listeners: [ MyCustomListener ]
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    contact_id:
      type: integer(4)
    username:
      type: string(255)
    password:
      type: string(255)
```

The above syntax will generated a base class that looks something like the following:

```php
class BaseUser extends Doctrine_Record
{
    // ...

    public setUp()
    {
        // ...
        $this->addListener(new MyCustomListener());
    }
}
```

Options

Specify options for your tables and when Doctrine creates your tables from your models the options will be set on the create table statement.

```yaml
---
User:
  connection: connection1
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    contact_id:
      type: integer(4)
    username:
      type: string(255)
    password:
      type: string(255)
  options:
    type: INNODB
    collate: utf8_unicode_ci
    charset: utf8
```

Indexes

Please see the Indexes section of the chapter for more information about indexes and their options.

```yaml
---
UserProfile:
  columns:
    user_id:
      type: integer
      length: 4
      primary: true
      autoincrement: true
    first_name:
      type: string
      length: 20
    last_name:
      type: string
      length: 20
  indexes:
    name_index:
      fields:
        first_name:
          sorting: ASC
          length: 10
          primary: true
        last_name: []
      type: unique
```

This is the PHP line of code that is auto-generated inside `setTableDefinition()` inside your base model class for the index definition used above:

```
$this->index('name_index', array(
        'fields' => array(
            'first_name' => array(
                'sorting'  => 'ASC',
                'length'   => '10',
                'primary'  => true
            ),
            'last_name' => array()),
        'type' => 'unique'
    )
);
```

Inheritance

Below we will demonstrate how you can setup the different types of inheritance using YAML schema files.

Simple Inheritance

```
---
Entity:
  columns:
    name: string(255)
    username: string(255)
    password: string(255)

User:
  inheritance:
    extends: Entity
    type: simple

Group:
  inheritance:
    extends: Entity
    type: simple
```

> Any columns or relationships defined in models that extend another in simple inheritance will be moved to the parent when the PHP classes are built.

You can read more about this topic in the Inheritance - Simple chapter.

Concrete Inheritance

```
---
TextItem:
  columns:
    topic: string(255)

Comment:
  inheritance:
    extends: TextItem
    type: concrete
  columns:
    content: string(300)
```

You can read more about this topic in the Inheritance - Concrete chapter.

Column Aggregation Inheritance

> Like simple inheritance, any columns or relationships added to the children will be automatically removed and moved to the parent when the PHP classes are built.

First lets defined a model named `Entity` that our other models will extend from:

```
---
Entity:
  columns:
    name: string(255)
    type: string(255)
```

> The type column above is optional. It will be automatically added when it is specified in the child class.

Now lets create a `User` model that extends the `Entity` model:

```yaml
---
User:
  inheritance:
    extends: Entity
    type: column_aggregation
    keyField: type
    keyValue: User
  columns:
    username: string(255)
    password: string(255)
```

> The `type` option under the `inheritance` definition is optional as it is implied if you specify a `keyField` or `keyValue`. If the `keyField` is not specified it will default to add a column named `type`. The `keyValue` will default to the name of the model if you do not specify anything.

Again lets create another model that extends `Entity` named `Group`:

```yaml
---
Group:
  inheritance:
    extends: Entity
    type: column_aggregation
    keyField: type
    keyValue: Group
  columns:
    description: string(255)
```

> The `User username` and `password` and the `Group description` columns will be automatically moved to the parent `Entity`.

You can read more about this topic in the Inheritance - Column Aggregation chapter.

Column Aliases

If you want the ability alias a column name as something other than the column name in the database this is easy to accomplish with Doctrine. We simple use the syntax "`column_name as field_name`" in the name of our column:

```yaml
---
User:
  columns:
    login:
      name: login as username
      type: string(255)
    password:
      type: string(255)
```

The above example would allow you to access the column named `login` from the alias `username`.

Packages

Doctrine offers the "package" parameter which will generate the models in to sub folders. With large schema files this will allow you to better organize your schemas in to folders.

```yaml
---
User:
  package: User
  columns:
    username: string(255)
```

The model files from this schema file would be put in a folder named User. You can specify more sub folders by doing "package: User.Models" and the models would be in User/Models

Package Custom Path

You can also completely by pass the automatic generation of packages to the appropriate path by specifying a completely custom path to generate the package fi les:

```yaml
---
User:
  package: User
  package_custom_path: /path/to/generate/package
  columns:
    username: string(255)
```

Global Schema Information

Doctrine schemas allow you to specify certain parameters that will apply to all of the models defined in the schema file. Below you can find an example on what global parameters you can set for schema files.

List of global parameters:

Name	Description
`connection`	Name of connection to bind the models to.
`attributes`	Array of attributes for models.
`actAs`	Array of behaviors for the models to act as.
`options`	Array of tables options for the models.
`package`	Package to put the models in.
`inheritance`	Array of inheritance information for models
`detect_relations`	Whether or not to try and detect foreign key relations

Now here is an example schema where we use some of the above global parameters:

```yaml
---
connection: conn_name1
actAs: [Timestampable]
options:
  type: INNODB
package: User
detect_relations: true

User:
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    contact_id:
      type: integer(4)
    username:
      type: string(255)
    password:
      type: string(255)

Contact:
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    name:
      type: string(255)
```

All of the settings at the top will be applied to every model which is defined in that YAML file.

Using Schema Files

Once you have defined your schema files you need some code to build the models from the YAML definition.

```php
$options = array(
  'packagesPrefix'  =>  'Plugin',
  'baseClassName'   =>  'MyDoctrineRecord',
  'suffix'          =>  '.php'
);

Doctrine_Core::generateModelsFromYaml('/path/to/yaml', '/path/to/model', $options);
```

The above code will generate the models for `schema.yml` at `/path/to/generate/models`.

Below is a table containing the different options you can use to customize the building of models. Notice we use the `packagesPrefix`, `baseClassName` and `suffix` options above.

Name	Default	Description
`packagesPrefix`	`Package`	What to prefix the middle package models with.
`packagesPath`	`#models_path#/packages`	Path to write package files.
`packagesFolderName`	`packages`	The name of the folder to put packages in, inside of the packages path.
`generateBaseClasses`	`true`	Whether or not to generate abstract base models containing the definition and a top level class which is empty extends the base.
`generateTableClasses`	`true`	Whether or not to generate a table class for each model.

baseClassPrefix	Base	The prefix to use for generated base class.
baseClassesDirectory	generated	Name of the folder to generate the base class definitions in.
baseTableClassName	Doctrine_Table	The base table class to extend the other generated table classes from.
baseClassName	Doctrine_Record	Name of the base Doctrine_Record class.
classPrefix		The prefix to use on all generated classes.
classPrefixFiles	true	Whether or not to use the class prefix for the generated file names as well.
pearStyle	false	Whether or not to generated PEAR style class names and file names. This option if set to true will replace underscores(_) with the DIRECTORY_SEPARATOR in the path to the generated class file.
suffix	.php	Extension for your generated models.
phpDocSubpackage		The phpDoc subpackage name to generate in the doc blocks.
phpDocName		The phpDoc author name to generate in the doc blocks.
phpDocEmail		The phpDoc e-mail to generate in the doc blocks.

Conclusion

Now that we have learned all about YAML Schema files we are ready to move on to a great topic regarding Data Validation. This is an important topic because if you are not validating user inputted data yourself then we want Doctrine to validate data before being persisted to the database.

Data Validation

Introduction

Pulled directly from PostgreSQL Documentation:

> Data types are a way to limit the kind of data that can be stored in a table. For many applications, however, the constraint they provide is too coarse. For example, a column containing a product price should probably only accept positive values. But there is no standard data type that accepts only positive numbers. Another issue is that you might want to constrain column data with respect to other columns or rows. For example, in a table containing product information, there should be only one row for each product number.

Doctrine allows you to define *portable* constraints on columns and tables. Constraints give you as much control over the data in your tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised. This applies even if the value came from the default value definition.

Doctrine constraints act as database level constraints as well as application level validators. This means double security: the database doesn't allow wrong kind of values and neither does the application.

Here is a full list of available validators within Doctrine:

validator(arguments)	constraints	description
`notnull`	`NOT NULL`	Ensures the 'not null' constraint in both application and database level
`email`		Checks if value is valid email.
`notblank`	`NOT NULL`	Checks if value is not blank.
`nospace`		Checks if value has no space chars.
`past`	`CHECK` constraint	Checks if value is a date in the past.
`future`		Checks if value is a date in the future.
`minlength(length)`		Checks if value satisfies the minimum length.
`country`		Checks if value is a valid country code.
`ip`		Checks if value is valid IP (internet protocol) address.
`htmlcolor`		Checks if value is valid html color.
`range(min, max)`	`CHECK` constraint	Checks if value is in range specified by arguments.
`unique`	`UNIQUE` constraint	Checks if value is unique in its database table.
`regexp(expression)`		Checks if value matches a given regexp.
`creditcard`		Checks whether the string is a well formated credit card number
`digits(int, frac)`	Precision and scale	Checks if given value has int number of integer digits and frac number of fractional digits
`date`		Checks if given value is a valid date.
`readonly`		Checks if a field is modified and if it is returns false to force a field as readonly
`unsigned`		Checks if given integer value is unsigned.
`usstate`		Checks if given value is a valid US state code.

Below is an example of how you use the validator and how to specify the arguments for the validators on a column.

In our example we will use the `minlength` validator.

```php
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('username', 'string', 255, array(
                'minlength' => 12
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
  columns:
    username:
      type: string(255)
      minlength: 12
# ...
```

Examples

Not Null

A not-null constraint simply specifies that a column must not assume the null value. A not-null constraint is always written as a column constraint.

The following definition uses a `notnull` constraint for column name. This means that the specified column doesn't accept null values.

```php
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('username', 'string', 255, array(
                'notnull' => true,
                'primary' => true,
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
  columns:
    username:
      type: string(255)
      notnull: true
      primary: true
# ...
```

When this class gets exported to database the following SQL statement would get executed (in MySQL):

```sql
CREATE TABLE user (username VARCHAR(255) NOT NULL,
PRIMARY KEY(username))
```

The `notnull` constraint also acts as an application level validator. This means that if Doctrine validators are turned on, Doctrine will automatically check that specified columns do not contain null values when saved.

If those columns happen to contain null values `Doctrine_Validator_Exception` is raised.

Email

The e-mail validator simply validates that the inputted value is indeed a valid e-mail address and that the MX records for the address domain resolve as a valid e-mail address.

```
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('email', 'string', 255, array(
                'email'   => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
User:
  columns:
# ...
    email:
      type: string(255)
      email: true
# ...
```

Now when we try and create a user with an invalid email address it will not validate:

```
// test.php

// ...
$user = new User();
$user->username = 'jwage';
$user->email = 'jonwage';

if ( ! $user->isValid()) {
    echo 'User is invalid!';
}
```

The above code will throw an exception because `jonwage` is not a valid e-mail address. Now we can take this even further and give a valid e-mail address format but an invalid domain name:

```
// test.php

// ...
$user = new User();
$user->username = 'jwage';
$user->email = 'jonwage@somefakedomainiknowdoesntexist.com';

if ( ! $user->isValid()) {
    echo 'User is invalid!';
}
```

Now the above code will still fail because the domain `somefakedomainiknowdoesntexist.com` does not exist and the php function `checkdnsrr()` returned false.

Not Blank

The not blank validator is similar to the not null validator except that it will fail on empty strings or strings with white space.

```php
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('username', 'string', 255, array(
                'notblank'    => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
  columns:
    username:
      type: string(255)
      notblank: true
# ...
```

Now if we try and save a `User` record with a username that is a single blank white space, validation will fail:

```php
// test.php

// ...
$user = new User();
$user->username = ' ';

if ( ! $user->isValid()) {
    echo 'User is invalid!';
}
```

No Space

The no space validator is simple. It checks that the value doesn't contain any spaces.

```php
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('username', 'string', 255, array(
                'nospace'    => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
  columns:
    username:
      type: string(255)
      nospace: true
# ...
```

Now if we try and save a `User` with a `username` that has a space in it, the validation will fail:

```php
$user = new User();
$user->username = 'jon wage';

if ( ! $user->isValid()) {
    echo 'User is invalid!';
}
```

Past

The past validator checks if the given value is a valid date in the past. In this example we'll have a `User` model with a `birthday` column and we want to validate that the date is in the past.

```php
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('birthday', 'timestamp', null, array(
                'past' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
  columns:
# ...
    birthday:
      type: timestamp
      past: true
# ...
```

Now if we try and set a birthday that is not in the past we will get a validation error.

Future

The future validator is the opposite of the past validator and checks if the given value is a valid date in the future. In this example we'll have a `User` model with a `next_appointment_date` column and we want to validate that the date is in the future.

```php
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('next_appointment_date', 'timestamp', null, array(
                'future' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
  columns:
# ...
    next_appointment_date:
      type: timestamp
      future: true
# ...
```

Now if we try and set an appointment date that is not in the future we will get a validation error.

Min Length

The min length does exactly what it says. It checks that the value string length is greater than the specified minimum length. In this example we will have a `User` model with a `password` column where we want to make sure the length of the `password` is at least 5 characters long.

```
// models/User.php

class User extends BaseUser
{
    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('password', 'timestamp', null, array(
                'minlength' => 5
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
User:
  columns:
# ...
    password:
      type: timestamp
      minlength: 5
# ...
```

Now if we try and save a `User` with a `password` that is shorter than 5 characters, the validation will fail.

```
// test.php

// ...
$user = new User();
$user->username = 'jwage';
$user->password = 'test';

if ( ! $user->isValid()) {
    echo 'User is invalid because "test" is only 4 characters long!';
}
```

Country

The country validator checks if the given value is a valid country code.

```
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('country', 'string', 2, array(
                'country' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
  columns:
# ...
    country:
      type: string(2)
      country: true
# ...
```

Now if you try and save a `User` with an invalid country code the validation will fail.

```php
// test.php

// ...
$user = new User();
$user->username = 'jwage';
$user->country_code = 'zz';

if ( ! $user->isValid()) {
    echo 'User is invalid because "zz" is not a valid country code!';
}
```

IP Address

The ip address validator checks if the given value is a valid ip address.

```php
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('ip_address', 'string', 15, array(
                'ip' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
  columns:
# ...
    ip_address:
      type: string(15)
      ip: true
# ...
```

Now if you try and save a `User` with an invalid ip address the validation will fail.

```php
$user = new User();
$user->username = 'jwage';
$user->ip_address = '123.123';

if ( ! $user->isValid()) {
    echo 'User is invalid because "123.123" is not a valid ip address
}
```

HTML Color

The html color validator checks that the given value is a valid html hex color.

```php
// models/User.php

class User extends BaseUser
{
    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('favorite_color', 'string', 7, array(
                'htmlcolor' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
  columns:
# ...
    favorite_color:
      type: string(7)
      htmlcolor: true
# ...
```

Now if you try and save a `User` with an invalid html color value for the `favorite_color` column the validation will fail.

```php
// test.php

// ...
$user = new User();
$user->username = 'jwage';
$user->favorite_color = 'red';

if ( ! $user->isValid()) {
    echo 'User is invalid because "red" is not a valid hex color';
}
```

Range

The range validator checks if value is within given range of numbers.

```php
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('age', 'integer', 3, array(
                'range' => array(10, 100)
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
  columns:
# ...
    age:
      type: integer(3)
      range: [10, 100]
# ...
```

Now if you try and save a `User` with an age that is less than 10 or greater than 100, the validation will fail.

```php
// test.php

// ...
$user = new User();
$user->username = 'jwage';
$user->age = '3';

if ( ! $user->isValid()) {
    echo 'User is invalid because "3" is less than the minimum of "10"';
}
```

You can use the `range` validator to validate max and min values by omitting either one of the `0` or `1` keys of the range array. Below is an example:

```php
// models/User.php

class User extends BaseUser
{
    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('age', 'integer', 3, array(
                'range' => array(1 => 100)
            )
        );
    }
}
```

The above would make it so that age has a max of 100. To have a minimum value simple specify `0` instead of `1` in the range array.

The YAML syntax for this would look like the following:

```yaml
---
# schema.yml

# ...
User:
  columns:
# ...
    age:
      type: integer(3)
      range:
        1: 100
# ...
```

Unique

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table.

In general, a unique constraint is violated when there are two or more rows in the table where the values of all of the columns included in the constraint are equal. However, two null values are not considered equal in this comparison. That means even in the presence of a unique constraint it is possible to store duplicate rows that contain a null value in at least one of the constrained columns. This behavior conforms to the SQL standard, but some databases do not follow this rule. So be careful when developing applications that are intended to be portable.

The following definition uses a `unique` constraint for column username.

```php
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('username', 'string', 255, array(
                'unique' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
User:
  columns:
    username:
      type: string(255)
      unique: true
# ....
```

> You should only use unique constraints for columns other than the primary key because they are always unique already.

Regular Expression

The regular expression validator is a simple way to validate column values against your own provided regular expression. In this example we will make sure the username contains only valid letters or numbers.

```
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('username', 'string', 255, array(
                'regexp' => '/[a-zA-Z0-9]/'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
User:
  columns:
    username:
      type: string(255)
      regexp: '/^[a-zA-Z0-9]+$/'
# ...
```

Now if we were to try and save a `User` with a `username` that has any other character than a letter or number in it, the validation will fail:

```
// test.php

// ...
$user = new User();
$user->username = '[jwage';

if ( ! $user->isValid()) {
    echo 'User is invalid because the username contains a [ character';
}
```

Credit Card

The credit card validator simply checks that the given value is indeed a valid credit card number.

```
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('cc_number', 'integer', 16, array(
                'creditcard' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
User:
  columns:
# ...
    cc_number:
      type: integer(16)
      creditcard: true
# ...
```

Read Only

The read only validator will fail validation if you modify a column that has the `readonly` validator enabled on it.

```
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('readonly_value', 'string', 255, array(
                'readonly' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
User:
  columns:
# ...
    readonly_value:
      type: integer(16)
      readonly: true
# ...
```

Now if I ever try and modify the column named `readonly_value` from a `User` object instance, validation will fail.

Unsigned

The unsigned validator checks that the given integer value is unsigned.

```
// models/User.php

class User extends BaseUser
{
    // ...

    public function setTableDefinition()
    {
        parent::setTableDefinition();

        // ...

        $this->hasColumn('age', 'integer', 3, array(
                'unsigned' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
User:
  columns:
# ...
    age:
      type: integer(3)
      unsigned: true
# ...
```

Now if I try and save a `User` with a negative age the validation will fail:

```php
// test.php

// ...
$user = new User();
$user->username = 'jwage';
$user->age = '-100';

if ( ! $user->isValid()) {
    echo 'User is invalid because -100 is signed';
}
```

US State

The us state validator checks that the given string is a valid US state code.

```php
// models/State.php

class State extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 255);
        $this->hasColumn('code', 'string', 2, array(
                'usstate' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
State:
  columns:
    name: string(255)
    code:
      type: string(2)
      usstate: true
```

Now if I try and save a `State` with an invalid state code then the validation will fail.

```php
$state = new State();
$state->name = 'Tennessee';
$state->code = 'ZZ';

if ( ! $state->isValid()) {
    echo 'State is invalid because "ZZ" is not a valid state code';
}
```

Conclusion

If we want Doctrine to validate our data before being persisted to the database, now we have the knowledge on how to do it. We can use the validators that are provided with the Doctrine core to perform common validations of our data.

The next chapter is an important one as we will discuss a great feature of Doctrine, Inheritance! Inheritance is a great way accomplish complex functionality with minimal code. After we discuss inheritance we will move on to a custom strategy that provides even better functionality than inheritance, called Behaviors.

Data Hydrators

Doctrine has a concept of data hydrators for transforming your `Doctrine_Query` instances to a set of PHP data for the user to take advantage of. The most obvious way to hydrate the data is to put it into your object graph and return models/class instances. Sometimes though you want to hydrate the data to an array, use no hydration or return a single scalar value. This chapter aims to document and demonstrate the different hydration types and even how to write your own new hydration types.

Core Hydration Methods

Doctrine offers a few core hydration methods to help you with the most common hydration needs.

Record

The first type is `HYDRATE_RECORD` and is the default hydration type. It will take the data from your queries and hydrate it into your object graph. With this type this type of thing is possible.

```
$q = Doctrine_Core::getTable('User')
    ->createQuery('u')
    ->leftJoin('u.Email e')
    ->where('u.username = ?', 'jwage');

$user = $q->fetchOne(array(), Doctrine_Core::HYDRATE_RECORD);

echo $user->Email->email;
```

The data for the above query was retrieved with one query and was hydrated into the object graph by the record hydrator. This makes it much easier to work with data since you are dealing with records and not result sets from the database which can be difficult to work with.

Array

The array hydration type is represented by the `HYDRATE_ARRAY` constant. It is identical to the above record hydration except that instead of hydrating the data into the object graph using PHP objects it uses PHP arrays. The benefit of using arrays instead of objects is that they are much faster and the hydration does not take as long.

So if you were to run the same example you would have access to the same data but it would be via a PHP array.

```
$q = Doctrine_Core::getTable('User')
    ->createQuery('u')
    ->leftJoin('u.Email e')
    ->where('u.username = ?', 'jwage');

$user = $q->fetchOne(array(), Doctrine_Core::HYDRATE_ARRAY);

echo $user['Email']['email'];
```

Scalar

The scalar hydration type is represented by the `HYDRATE_SCALAR` constant and is
a very fast and efficient way to hydrate your data. The downside to this method is that it does not hydrate your data into the object graph, it returns a flat rectangular result set which can be difficult to work with when dealing with lots of records.

```
$q = Doctrine_Core::getTable('User')
    ->createQuery('u')
    ->where('u.username = ?', 'jwage');

$user = $q->fetchOne(array(), Doctrine_Core::HYDRATE_SCALAR);

echo $user['u_username'];
```

The above query would produce a data structure that looks something like the following:

```
$user = array(
    'u_username' => 'jwage',
    'u_password' => 'changeme',
    // ...
);
```

If the query had a many relationship joined than the data for the user would be duplicated for every record that exists for that user. This is the downside as it is difficult to work with when dealing with lots of records.

Single Scalar

Often times you want a way to just return a single scalar value. This is possible with the single scalar hydration method and is represented by the `HYDRATE_SINGLE_SCALAR` attribute.

With this hydration type we could easily count the number of phonenumbers a user has with the following:

```
$q = Doctrine_Core::getTable('User')
    ->createQuery('u')
    ->select('COUNT(p.id)')
    ->leftJoin('u.Phonenumber p')
    ->where('u.username = ?', 'jwage');

$numPhonenumbers = $q->fetchOne(array(), Doctrine_Core::HYDRATE_SINGLE_SCALAR);

echo $numPhonenumbers;
```

This is much better than hydrating the data with a more complex method and grabbing the value from those results. With this it is very fast and efficient to get the data you really want.

On Demand

If you wish to use a less memory intensive hydration method you can use the on demand hydration which is represented by the `HYDRATE_ON_DEMAND` constant. It will only hydrate one records graph at a time so that means less memory footprint overall used.

```
// Returns instance of Doctrine_Collection_OnDemand
$result = $q->execute(array(), Doctrine_Core::HYDRATE_ON_DEMAND);
foreach ($result as $obj) {
    // ...
}
```

`Doctrine_Collection_OnDemand` hydrates each object one at a time as you iterate over it so this results in less memory being used because we don't have to first load all the data from the database to PHP then convert it to the entire data structure to return.

Nested Set Record Hierarchy

For your models which use the nested set behavior you can use the record hierarchy hydration method to hydrate your nested set tree into an actual hierarchy of nested objects.

```
$categories = Doctrine_Core::getTable('Category')
    ->createQuery('c')
    ->execute(array(), Doctrine_Core::HYDRATE_RECORD_HIERARCHY);
```

Now you can access the children of a record by accessing the mapped value property named `__children`. It is named with the underscores prefixed to avoid any naming conflicts.

```
foreach ($categories->getFirst()->get('__children') as $child) {
    // ...
}
```

Nested Set Array Hierarchy

If you wish to hydrate the nested set hierarchy into arrays instead of objects you can do so using the `HYDRATE_ARRAY_HIERARCHY` constant. It is identical to the `HYDRATE_RECORD_HIERARCHY` except that it uses PHP arrays instead of objects.

```
$categories = Doctrine_Core::getTable('Category')
    ->createQuery('c')
    ->execute(array(), Doctrine_Core::HYDRATE_ARRAY_HIERARCHY);
```

Now you can do the following:

```
foreach ($categories[0]['__children'] as $child) {
    // ...
}
```

Writing Hydration Method

Doctrine offers the ability to write your own hydration methods and register them with Doctrine for use. All you need to do is write a class that extends `Doctrine_Hydrator_Abstract` and register it with `Doctrine_Manager`.

First lets write a sample hydrator class:

```
class Doctrine_Hydrator_MyHydrator extends Doctrine_Hydrator_Abstract
{
    public function hydrateResultSet($stmt)
    {
        $data = $stmt->fetchAll(PDO::FETCH_ASSOC);
        // do something to with $data
        return $data;
    }
}
```

To use it make sure we register it with `Doctrine_Manager`:

```
// bootstrap.php

// ...
$manager->registerHydrator('my_hydrator', 'Doctrine_Hydrator_MyHydrator');
```

Now when you execute your queries you can pass `my_hydrator` and it will use your class to hydrate the data.

```
$q->execute(array(), 'my_hydrator');
```

Inheritance

Doctrine supports three types of inheritance strategies which can be mixed together. The three types are simple, concrete and column aggregation. You will learn about these three different types of inheritance and how to use them in this chapter.

For this chapter lets delete all our existing schemas and models from our test environment we created and have been using in the earlier chapters:

```
$ rm schema.yml
$ touch schema.yml
$ rm -rf models/*
```

Simple

Simple inheritance is the easiest and simplest inheritance to use. In simple inheritance all the child classes share the same columns as the parent and all information is stored in the parent table.

```
// models/Entity.php

class Entity extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 30);
        $this->hasColumn('username', 'string', 20);
        $this->hasColumn('password', 'string', 16);
        $this->hasColumn('created_at', 'timestamp');
        $this->hasColumn('update_at', 'timestamp');
    }
}
```

Now lets create a `User` model that extends `Entity`:

```
// models/User.php

class User extends Entity
{ }
```

Do the same thing for the `Group` model:

```
// models/Group.php

class Group extends Entity
{ }
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Entity:
  columns:
    name: string(30)
    username: string(20)
    password: string(16)
    created_at: timestamp
    updated_at: timestamp

User:
  inheritance:
    extends: Entity
    type: simple

Group:
  inheritance:
    extends: Entity
    type: simple
```

Lets check the SQL that is generated by the above models:

```php
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('Entity', 'User', 'Group'));
echo $sql[0];
```

The above code would output the following SQL query:

```
CREATE TABLE entity (id BIGINT AUTO_INCREMENT,
username VARCHAR(20),
password VARCHAR(16),
created_at DATETIME,
updated_at DATETIME,
name VARCHAR(30),
PRIMARY KEY(id)) ENGINE = INNODB
```

> When using YAML schema files you are able to define columns in the child classes but when the YAML is parsed the columns are moved to the parent for you automatically. This is only a convenience to you so that you can organize your columns easier.

Concrete

Concrete inheritance creates separate tables for child classes. However in concrete inheritance each class generates a table which contains all columns (including inherited columns). In order to use concrete inheritance you'll need to add explicit `parent::setTableDefinition()` calls to child classes as shown below.

```php
// models/TextItem.php

class TextItem extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('topic', 'string', 100);
    }
}
```

Now lets create a model named `Comment` that extends `TextItem` and add an extra column named `content`:

```php
// models/Comment.php

class Comment extends TextItem
{
    public function setTableDefinition()
    {
        parent::setTableDefinition();

        $this->hasColumn('content', 'string', 300);
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
TextItem:
  columns:
    topic: string(100)

Comment:
  inheritance:
    extends: TextItem
    type: concrete
  columns:
    content: string(300)
```

Lets check the SQL that is generated by the above models:

```php
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('TextItem', 'Comment'));
echo $sql[0] . "\n";
echo $sql[1];
```

The above code would output the following SQL query:

```
CREATE TABLE text_item (id BIGINT AUTO_INCREMENT,
topic VARCHAR(100),
PRIMARY KEY(id)) ENGINE = INNODB
CREATE TABLE comment (id BIGINT AUTO_INCREMENT,
topic VARCHAR(100),
content TEXT,
PRIMARY KEY(id)) ENGINE = INNODB
```

In concrete inheritance you don't necessarily have to define additional columns, but in order to make Doctrine create separate tables for each class you'll have to make iterative `setTableDefinition()` calls.

In the following example we have three database tables called `entity`, `user` and `group`. `Users` and `groups` are both `entities`. The only thing we have to do is write 3 classes (`Entity`, `Group` and `User`) and make iterative `setTableDefinition()` method calls.

```
// models/Entity.php

class Entity extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 30);
        $this->hasColumn('username', 'string', 20);
        $this->hasColumn('password', 'string', 16);
        $this->hasColumn('created', 'integer', 11);
    }
}

// models/User.php

class User extends Entity
{
    public function setTableDefinition()
    {
        // the following method call is needed in
        // concrete inheritance
        parent::setTableDefinition();
    }
}

// models/Group.php
class Group extends Entity
{
    public function setTableDefinition()
    {
        // the following method call is needed in
        // concrete inheritance
        parent::setTableDefinition();
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
Entity:
  columns:
    name: string(30)
    username: string(20)
    password: string(16)
    created: integer(11)

User:
  inheritance:
    extends: Entity
    type: concrete

Group:
  tableName: groups
  inheritance:
    extends: Entity
    type: concrete
```

Lets check the SQL that is generated by the above models:

```
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('Entity', 'User', 'Group'));
echo $sql[0] . "\n";
echo $sql[1] . "\n";
echo $sql[2] . "\n";
```

The above code would output the following SQL query:

```
CREATE TABLE user (id BIGINT AUTO_INCREMENT,
name VARCHAR(30),
username VARCHAR(20),
password VARCHAR(16),
created BIGINT,
PRIMARY KEY(id)) ENGINE = INNODB
CREATE TABLE groups (id BIGINT AUTO_INCREMENT,
name VARCHAR(30),
username VARCHAR(20),
password VARCHAR(16),
created BIGINT,
PRIMARY KEY(id)) ENGINE = INNODB
CREATE TABLE entity (id BIGINT AUTO_INCREMENT,
name VARCHAR(30),
username VARCHAR(20),
password VARCHAR(16),
created BIGINT,
PRIMARY KEY(id)) ENGINE = INNODB
```

Column Aggregation

In the following example we have one database table called `entity`. `Users` and `groups` are both `entities` and they share the same database table.

The `entity` table has a column called `type` which tells whether an `entity` is a `group` or a `user`. Then we decide that `users` are type 1 and groups type 2.

The only thing we have to do is to create 3 records (the same as before) and add the call to the `Doctrine_Table::setSubclasses()` method from the parent class.

```
// models/Entity.php

class Entity extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 30);
        $this->hasColumn('username', 'string', 20);
        $this->hasColumn('password', 'string', 16);
        $this->hasColumn('created_at', 'timestamp');
        $this->hasColumn('update_at', 'timestamp');

        $this->setSubclasses(array(
                'User'  => array('type' => 1),
                'Group' => array('type' => 2)
            )
        );
    }
}

// models/User.php
class User extends Entity
{ }

// models/Group.php
class Group extends Entity
{ }
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
Entity:
  columns:
    username: string(20)
    password: string(16)
    created_at: timestamp
    updated_at: timestamp

User:
  inheritance:
    extends: Entity
    type: column_aggregation
    keyField: type
    keyValue: 1

Group:
  inheritance:
    extends: Entity
    type: column_aggregation
    keyField: type
    keyValue: 2
```

Lets check the SQL that is generated by the above models:

```
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('Entity', 'User', 'Group'));
echo $sql[0];
```

The above code would output the following SQL query:

```
CREATE TABLE entity (id BIGINT AUTO_INCREMENT,
username VARCHAR(20),
password VARCHAR(16),
created_at DATETIME,
updated_at DATETIME,
type VARCHAR(255),
PRIMARY KEY(id)) ENGINE = INNODB
```

> Notice how the `type` column was automatically added. This is how column aggregation inheritance knows which model each record in the database belongs to.

This feature also enable us to query the `Entity` table and get a `User` or `Group` object back if the returned object matches the constraints set in the parent class.

See the code example below for an example of this. First lets save a new `User` object:

```
// test.php

// ...
$user = new User();
$user->name = 'Bjarte S. Karlsen';
$user->username = 'meus';
$user->password = 'rat';
$user->save();
```

Now lets save a new `Group` object:

```
// test.php

// ...
$group = new Group();
$group->name = 'Users';
$group->username = 'users';
$group->password = 'password';
$group->save();
```

Now if we query the `Entity` model for the id of the `User` we created, the `Doctrine_Query` will return an instance of `User`.

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('Entity e')
    ->where('e.id = ?');

$user = $q->fetchOne(array($user->id));

echo get_class($user); // User
```

If we do the same thing as above but for the `Group` record, it will return an instance of `Group`.

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('Entity e')
    ->where('e.id = ?');

$group = $q->fetchOne(array($group->id));

echo get_class($group); // Group
```

> The above is possible because of the `type` column. Doctrine knows which class each record was created by, so when data is being hydrated it can be hydrated in to the appropriate sub-class.

We can also query the individual `User` or `Group` models:

```
$q = Doctrine_Query::create()
    ->select('u.id')
    ->from('User u');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
e.id AS e__id
FROM entity e
WHERE (e.type = '1')
```

> Notice how the `type` condition was automatically added to the query so that it will only return records that are of type `User`.

Conclusion

Now that we've learned about how to take advantage of PHPs inheritance features with our models we can move on to learning about Doctrine Behaviors. This is one of the most sophisticated and useful features in Doctrine for accomplishing complex models with small and easy to maintain code.

Behaviors

Introduction

Many times you may find classes having similar things within your models. These things may contain anything related to the schema of the component itself (relations, column definitions, index definitions etc.). One obvious way of re-factoring the code is having a base class with some classes extending it.

However inheritance solves only a fraction of things. The following sections show how using `Doctrine_Template` is much more powerful and flexible than using inheritance.

`Doctrine_Template` is a class template system. Templates are basically ready-to-use little components that your Record classes can load. When a template is being loaded its `setTableDefinition()` and `setUp()` methods are being invoked and the method calls inside them are being directed into the class in question.

This chapter describes the usage of various behaviors available for Doctrine. You'll also learn how to create your own behaviors. In order to grasp the concepts of this chapter you should be familiar with the theory behind `Doctrine_Template` and `Doctrine_Record_Generator`. We will explain what these classes are shortly.

When referring to behaviors we refer to class packages that use templates, generators and listeners extensively. All the introduced components in this chapter can be considered `core` behaviors, that means they reside at the Doctrine main repository.

Usually behaviors use generators side-to-side with template classes (classes that extend `Doctrine_Template`). The common workflow is:

- A new template is being initialized
- The template creates the generator and calls `initialize()` method
- The template is attached to given class

As you may already know templates are used for adding common definitions and options to record classes. The purpose of generators is much more complex. Usually they are being used for creating generic record classes dynamically. The definitions of these generic classes usually depend on the owner class. For example the columns of the `AuditLog` versioning class are the columns of the parent class with all the sequence and autoincrement definitions removed.

Simple Templates

In the following example we define a template called `TimestampBehavior`. Basically the purpose of this template is to add date columns 'created' and 'updated' to the record class that loads this template. Additionally this template uses a listener called Timestamp listener which updates these fields based on record actions.

```
// models/TimestampListener.php

class TimestampListener extends Doctrine_Record_Listener
{
    public function preInsert(Doctrine_Event $event)
    {
        $event->getInvoker()->created = date('Y-m-d', time());
        $event->getInvoker()->updated = date('Y-m-d', time());
    }

    public function preUpdate(Doctrine_Event $event)
    {
        $event->getInvoker()->updated = date('Y-m-d', time());
    }
}
```

Now lets create a child `Doctrine_Template` named `TimestampTemplate` so we can attach it to our models with the `actAs()` method:

```
// models/TimestampBehavior.php

class TimestampTemplate extends Doctrine_Template
{
    public function setTableDefinition()
    {
        $this->hasColumn('created', 'date');
        $this->hasColumn('updated', 'date');

        $this->addListener(new TimestampListener());
    }
}
```

Lets say we have a class called `BlogPost` that needs the timestamp functionality. All we need to do is to add `actAs()` call in the class definition.

```
class BlogPost extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 200);
        $this->hasColumn('body', 'clob');
    }

    public function setUp()
    {
        $this->actAs('TimestampBehavior');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
BlogPost:
  actAs: [TimestampBehavior]
  columns:
    title: string(200)
    body: clob
```

Now when we try and utilize the `BlogPost` model you will notice that the `created` and `updated` columns were added for you and automatically set when saved:

```
$blogPost = new BlogPost();
$blogPost->title = 'Test';
$blogPost->body = 'test';
$blogPost->save();

print_r($blogPost->toArray());
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [id] => 1
    [title] => Test
    [body] => test
    [created] => 2009-01-22
    [updated] => 2009-01-22
)
```

> The above described functionality is available via the `Timestampable` behavior that we have already talked about. You can go back and read more about it in the Timestampable section of this chapter.

Templates with Relations

Many times the situations tend to be much more complex than the situation in the previous chapter. You may have model classes with relations to other model classes and you may want to replace given class with some extended class.

Consider we have two classes, `User` and `Email`, with the following definitions:

```php
class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 255);
        $this->hasColumn('password', 'string', 255);
    }

    public function setUp()
    {
        $this->hasMany('Email', array(
                'local' => 'id',
                'foreign' => 'user_id'
            )
        );
    }
}

class Email extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('address', 'string');
        $this->hasColumn('user_id', 'integer');
    }

    public function setUp()
    {
        $this->hasOne('User', array(
                'local' => 'user_id',
                'foreign' => 'id'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
User:
  columns:
    username: string(255)
    password: string(255)

Email:
  columns:
    address: string
    user_id: integer
  relations:
    User:
```

Now if we extend the `User` and `Email` classes and create, for example, classes `ExtendedUser` and `ExtendedEmail`, the `ExtendedUser` will still have a relation to the `Email` class and not the `ExtendedEmail` class. We could of course override the `setUp()` method of the `User` class and define relation to the `ExtendedEmail` class, but then we lose the whole point of inheritance. `Doctrine_Template` can solve this problem elegantly with its dependency injection solution.

In the following example we'll define two templates, `UserTemplate` and `EmailTemplate`, with almost identical definitions as the `User` and `Email` class had.

```php
// models/UserTemplate.php

class UserTemplate extends Doctrine_Template
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 255);
        $this->hasColumn('password', 'string', 255);
    }

    public function setUp()
    {
        $this->hasMany('EmailTemplate as Emails', array(
                'local' => 'id',
                'foreign' => 'user_id'
            )
        );
    }
}
```

Now lets define the `EmailTemplate`:

```php
// models/EmailTemplate.php

class EmailTemplate extends Doctrine_Template
{
    public function setTableDefinition()
    {
        $this->hasColumn('address', 'string');
        $this->hasColumn('user_id', 'integer');
    }

    public function setUp()
    {
        $this->hasOne('UserTemplate as User', array(
                'local' => 'user_id',
                'foreign' => 'id'
            )
        );
    }
}
```

Notice how we set the relations. We are not pointing to concrete Record classes, rather we are setting the relations to templates. This tells Doctrine that it should try to find concrete Record classes for those templates. If Doctrine can't find these concrete implementations the relation parser will throw an exception, but before we go ahead of things, here are the actual record classes:

```php
class User extends Doctrine_Record
{
    public function setUp()
    {
        $this->actAs('UserTemplate');
    }
}

class Email extends Doctrine_Record
{
    public function setUp()
    {
        $this->actAs('EmailTemplate');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
User:
  actAs: [UserTemplate]

Email:
  actAs: [EmailTemplate]
```

Now consider the following code snippet. This does NOT work since we haven't yet set any concrete implementations for the templates.

```php
// test.php

// ...
$user = new User();
$user->Emails; // throws an exception
```

The following version works. Notice how we set the concrete implementations for the templates globally using `Doctrine_Manager`:

```php
// bootstrap.php

// ...
$manager->setImpl('UserTemplate', 'User')
        ->setImpl('EmailTemplate', 'Email');
```

Now this code will work and won't throw an exception like it did before:

```php
$user = new User();
$user->Emails[0]->address = 'jonwage@gmail.com';
$user->save();

print_r($user->toArray(true));
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [id] => 1
    [username] =>
    [password] =>
    [Emails] => Array
        (
            [0] => Array
                (
                    [id] => 1
                    [address] => jonwage@gmail.com
                    [user_id] => 1
                )

        )

)
```

> The implementations for the templates can be set at manager, connection and even at the table level.

Delegate Methods

Besides from acting as a full table definition delegate system, `Doctrine_Template` allows the delegation of method calls. This means that every method within the loaded templates is available in the record that loaded the templates. Internally the implementation uses magic method called `__call()` to achieve this functionality.

Lets add to our previous example and add some custom methods to the `UserTemplate`:

```php
// models/UserTemplate.php

class UserTemplate extends Doctrine_Template
{
    // ...

    public function authenticate($username, $password)
    {
        $invoker = $this->getInvoker();
        if ($invoker->username == $username && $invoker->password == $password) {
            return true;
        } else {
            return false;
        }
    }
}
```

Now take a look at the following code and how we can use it:

```php
$user = new User();
$user->username = 'jwage';
$user->password = 'changeme';

if ($user->authenticate('jwage', 'changemte')) {
    echo 'Authenticated successfully!';
} else {
    echo 'Could not authenticate user!';
}
```

You can also delegate methods to `Doctrine_Table` classes just as easily. But, to avoid naming collisions the methods for table classes must have the string `TableProxy` appended to the end of the method name.

Here is an example where we add a new finder method:

```php
// models/UserTemplate.php

class UserTemplate extends Doctrine_Template
{
    // ...

    public function findUsersWithEmailTableProxy()
    {
        return Doctrine_Query::create()
            ->select('u.username')
            ->from('User u')
            ->innerJoin('u.Emails e')
            ->execute();
    }
}
```

Now we can access that function from the `Doctrine_Table` object for the `User` model:

```php
$userTable = Doctrine_Core::getTable('User');
$users = $userTable->findUsersWithEmail();
```

> Each class can consists of multiple templates. If the templates contain similar definitions the most recently loaded template always overrides the former.

Creating Behaviors

This subchapter provides you the means for creating your own behaviors. Lets say we have various different Record classes that need to have one-to-many emails. We achieve this functionality by creating a generic behavior which creates Email classes on the fly.

We start this task by creating a behavior called `EmailBehavior` with a `setTableDefinition()` method. Inside the `setTableDefinition()` method various helper methods can be used for easily creating the dynamic record definition. Commonly the following methods are being used:

```php
public function initOptions()
public function buildLocalRelation()
public function buildForeignKeys(Doctrine_Table $table)
public function buildForeignRelation($alias = null)
public function buildRelation() // calls buildForeignRelation() and buildLocalRelation()
```

```php
class EmailBehavior extends Doctrine_Record_Generator
{
    public function initOptions()
    {
        $this->setOption('className', '%CLASS%Email');

        // Some other options
        // $this->setOption('appLevelDelete', true);
        // $this->setOption('cascadeDelete', false);
    }

    public function buildRelation()
    {
        $this->buildForeignRelation('Emails');
        $this->buildLocalRelation();
    }

    public function setTableDefinition()
    {
        $this->hasColumn('address', 'string', 255, array(
                'email'   => true,
                'primary' => true
            )
        );
    }
}
```

Core Behaviors

For the next several examples using the core behaviors lets delete all our existing schemas and models from our test environment we created and have been using in the earlier chapters:

```
$ rm schema.yml
$ touch schema.yml
$ rm -rf models/*
```

Introduction

Doctrine comes bundled with some templates that offer out of the box functionality for your models. You can enable these templates in your models very easily. You can do it directly in your `Doctrine_Record`s or you can specify them in your YAML schema if you are managing your models with YAML.

In the next several examples we will demonstrate some of the behaviors that come bundled with Doctrine.

Versionable

Lets create a `BlogPost` model that we want to have the ability to have versions:

```php
// models/BlogPost.php

class BlogPost extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 255);
        $this->hasColumn('body', 'clob');
    }

    public function setUp()
    {
        $this->actAs('Versionable', array(
                'versionColumn' => 'version',
                'className' => '%CLASS%Version',
                'auditLog' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
BlogPost:
  actAs:
    Versionable:
      versionColumn: version
      className: %CLASS%Version
      auditLog: true
  columns:
    title: string(255)
    body: clob
```

> The `auditLog` option can be used to turn off the audit log history. This is when you want to maintain a version number but not maintain the data at each version.

Lets check the SQL that is generated by the above models:

```php
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('BlogPost'));
echo $sql[0] . "\n";
echo $sql[1];
```

The above code would output the following SQL query:

```sql
CREATE TABLE blog_post_version (id BIGINT,
title VARCHAR(255),
body LONGTEXT,
version BIGINT,
PRIMARY KEY(id,
version)) ENGINE = INNODB
CREATE TABLE blog_post (id BIGINT AUTO_INCREMENT,
title VARCHAR(255),
body LONGTEXT,
version BIGINT,
PRIMARY KEY(id)) ENGINE = INNODB
ALTER TABLE blog_post_version ADD FOREIGN KEY (id) REFERENCES blog_post(id) ON UPDATE CASCADE ON DELETE CASCADE
```

> Notice how we have 2 additional statements we probably didn't expect to see. The behavior automatically created a `blog_post_version` table and related it to `blog_post`.

Now when we insert or update a `BlogPost` the version table will store all the old versions of the record and allow you to revert back at anytime. When you instantiate a `BlogPost` for the first time this is what is happening internally:

- It creates a class called `BlogPostVersion` on-the-fly, the table this record is pointing at is `blog_post_version`
- Everytime a `BlogPost` object is deleted / updated the previous version is stored into `blog_post_version`
- Everytime a `BlogPost` object is updated its version number is increased.

Now lets play around with the `BlogPost` model:

```php
$blogPost = new BlogPost();
$blogPost->title = 'Test blog post';
$blogPost->body = 'test';
$blogPost->save();

$blogPost->title = 'Modified blog post title';
$blogPost->save();

print_r($blogPost->toArray());
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [id] => 1
    [title] => Modified blog post title
    [body] => test
    [version] => 2
)
```

> Notice how the value of the `version` column is `2`. This is because we have saved 2 versions of the `BlogPost` model. We can easily revert to another version by using the `revert()` method that the behavior includes.

Lets revert back to the first version:

```
$blogPost->revert(1);
print_r($blogPost->toArray());
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [id] => 2
    [title] => Test blog post
    [body] => test
    [version] => 1
)
```

> Notice how the value of the `version` column is set to 1 and the `title` is back to the original value was set it to when creating the `BlogPost`.

Timestampable

The Timestampable behavior will automatically add a `created_at` and `updated_at` column and automatically set the values when a record is inserted and updated.

Since it is common to want to know the date a post is made lets expand our `BlogPost` model and add the `Timestampable` behavior to automatically set these dates for us.

```
// models/BlogPost.php

class BlogPost extends Doctrine_Record
{
    // ...

    public function setUp()
    {
        $this->actAs('Timestampable');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
BlogPost:
  actAs:
# ...
    Timestampable:
# ...
```

If you are only interested in using only one of the columns, such as a `created_at` timestamp, but not a an `updated_at` field, set the `disabled` to true for either of the fields as in the example below.

```
---
BlogPost:
  actAs:
# ...
    Timestampable:
      created:
        name: created_at
        type: timestamp
        format: Y-m-d H
      updated:
        disabled: true
# ...
```

Now look what happens when we create a new post:

```
$blogPost = new BlogPost();
$blogPost->title = 'Test blog post';
$blogPost->body = 'test';
$blogPost->save();

print_r($blogPost->toArray());
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [id] => 1
    [title] => Test blog post
    [body] => test
    [version] => 1
    [created_at] => 2009-01-21 17:54:23
    [updated_at] => 2009-01-21 17:54:23
)
```

> Look how the `created_at` and `updated_at` values were automatically set for you!

Here is a list of all the options you can use with the `Timestampable` behavior on the created side of the behavior:

Name	Default	Description
`name`	`created_at`	**The name of the column.**
`type`	`timestamp`	**The column type.**
`options`	`array()`	**Any additional options for the column.**
`format`	`Y-m-d H:i:s`	**The format of the timestamp if you don't use the timestamp column type. The date is built using PHP's date() function.**
`disabled`	`false`	**Whether or not to disable the created date.**
`expression`	`NOW()`	**Expression to use to set the column value.**

Here is a list of all the options you can use with the `Timestampable` behavior on the updated side of the behavior that are not possible on the created side:

Name	Default	Description
`onInsert`	`true`	**Whether or not to set the updated date when the record is first inserted.**

Sluggable

The `Sluggable` behavior is a nice piece of functionality that will automatically add a column to your model for storing a unique human readable identifier that can be created from columns like title, subject, etc. These values can be used for search engine friendly urls.

Lets expand our `BlogPost` model to use the `Sluggable` behavior because we will want to have nice URLs for our posts:

```
// models/BlogPost.php

class BlogPost extends Doctrine_Record
{
    // ...

    public function setUp()
    {
        // ...

        $this->actAs('Sluggable', array(
                'unique'    => true,
                'fields'    => array('username'),
                'canUpdate' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
BlogPost:
  actAs:
# ...
    Sluggable:
      unique: true
      fields: [title]
      canUpdate: true
# ...
```

Now look what happens when we create a new post. The slug column will automatically be set for us:

```
$blogPost = new BlogPost();
$blogPost->title = 'Test blog post';
$blogPost->body = 'test';
$blogPost->save();

print_r($blogPost->toArray());
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [id] => 1
    [title] => Test blog post
    [body] => test
    [version] => 1
    [created_at] => 2009-01-21 17:57:05
    [updated_at] => 2009-01-21 17:57:05
    [slug] => test-blog-post
)
```

> Notice how the value of the `slug` column was automatically set based on the value of the `title` column. When a slug is created, by default it is `urlized` which means all non-url-friendly characters are removed and white space is replaced with hyphens(-).

The unique flag will enforce that the slug created is unique. If it is not unique an auto incremented integer will be appended to the slug before saving to database.

The `canUpdate` flag will allow the users to manually set the slug value to be used when building the url friendly slug.

Here is a list of all the options you can use on the `Sluggable` behavior:

Name	Default	Description
`name`	`slug`	The name of the slug column.
`alias`	`null`	The alias of the slug column.
`type`	`string`	The type of the slug column.
`length`	`255`	The length of the slug column.
`unique`	`true`	Whether or not unique slug values are enforced.
`options`	`array()`	Any other options for the slug column.
`fields`	`array()`	The fields that are used to build slug value.
`uniqueBy`	`array()`	The fields that make determine a unique slug.
`uniqueIndex`	`true`	Whether or not to create a unique index.
`canUpdate`	`false`	Whether or not the slug can be updated.
`builder`	`array('Doctrine_Inflector', 'urlize')`	The `Class::method()` used to build the slug.
`indexName`	`sluggable`	The name of the index to create.

I18n

`Doctrine_I18n` package is a behavior for Doctrine that provides internationalization support for record classes. In the following example we have a `NewsItem` class with two fields `title` and `content`. We want to have the field `title` with different languages support. This can be achieved as follows:

```php
class NewsItem extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 255);
        $this->hasColumn('body', 'blog');
    }

    public function setUp()
    {
        $this->actAs('I18n', array(
                'fields' => array('title', 'body')
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
NewsItem:
  actAs:
    I18n:
      fields: [title, body]
  columns:
    title: string(255)
    body: clob
```

Below is a list of all the options you can use with the `I18n` behavior:

Name	Default	Description
`className`	`%CLASS%Translation`	The name pattern to use for generated class.
`fields`	`array()`	The fields to internationalize.
`type`	`string`	The type of `lang` column.
`length`	`2`	The length of the `lang` column.
`options`	`array()`	Other options for the `lang` column.

Lets check the SQL that is generated by the above models:

```php
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('NewsItem'));
echo $sql[0] . "\n";
echo $sql[1];
```

The above code would output the following SQL query:

```sql
CREATE TABLE news_item_translation (id BIGINT,
title VARCHAR(255),
body LONGTEXT,
lang CHAR(2),
PRIMARY KEY(id,
lang)) ENGINE = INNODB
CREATE TABLE news_item (id BIGINT AUTO_INCREMENT,
PRIMARY KEY(id)) ENGINE = INNODB
```

> Notice how the field `title` is not present in the `news_item` table. Since its present in the translation table it would be a waste of resources to have that same field in the main table. Basically Doctrine always automatically removes all translated fields from the main table.

Now the first time you initialize a new `NewsItem` record Doctrine initializes the behavior that builds the followings things:

1. Record class called `NewsItemTranslation`
2. Bi-directional relations between `NewsItemTranslation` and `NewsItem`

Lets take a look at how we can manipulate the translations of the `NewsItem`:

```php
// test.php

// ...
$newsItem = new NewsItem();
$newsItem->Translation['en']->title = 'some title';
$newsItem->Translation['en']->body = 'test';
$newsItem->Translation['fi']->title = 'joku otsikko';
$newsItem->Translation['fi']->body = 'test';
$newsItem->save();

print_r($newsItem->toArray());
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [id] => 1
    [Translation] => Array
        (
            [en] => Array
                (
                    [id] => 1
                    [title] => some title
                    [body] => test
                    [lang] => en
                )
            [fi] => Array
                (
                    [id] => 1
                    [title] => joku otsikko
                    [body] => test
                    [lang] => fi
                )

        )

)
```

How do we retrieve the translated data now? This is easy! Lets find all items and their Finnish translations:

```
// test.php

// ...
$newsItems = Doctrine_Query::create()
    ->from('NewsItem n')
    ->leftJoin('n.Translation t')
    ->where('t.lang = ?')
    ->execute(array('fi'));

echo $newsItems[0]->Translation['fi']->title;
```

The above example would produce the following output:

```
$ php test.php
joku otsikko
```

NestedSet

The `NestedSet` behavior allows you to turn your models in to a nested set tree structure where the entire tree structure can be retrieved in one efficient query. It also provided a nice interface for manipulating the data in your trees.

Lets take a `Category` model for example where the categories need to be organized in a hierarchical tree structure:

```
// models/Category.php

class Category extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 255);
    }

    public function setUp()
    {
        $this->actAs('NestedSet', array(
                'hasManyRoots' => true,
                'rootColumnName' => 'root_id'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Category:
  actAs:
    NestedSet:
      hasManyRoots: true
      rootColumnName: root_id
  columns:
    name: string(255)
```

Lets check the SQL that is generated by the above models:

```php
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('Category'));
echo $sql[0];
```

The above code would output the following SQL query:

```sql
CREATE TABLE category (id BIGINT AUTO_INCREMENT,
name VARCHAR(255),
root_id INT,
lft INT,
rgt INT,
level SMALLINT,
PRIMARY KEY(id)) ENGINE = INNODB
```

> Notice how the `root_id`, `lft`, `rgt` and `level` columns are automatically added. These columns are used to organize the tree structure and are handled automatically for you internally.

We won't discuss the `NestedSet` behavior in 100% detail here. It is a very large behavior so it has its own dedicated chapter.

Searchable

The `Searchable` behavior is a fulltext indexing and searching tool. It can be used for indexing and searching both database and files.

Imagine we have a `Job` model for job postings and we want it to be easily searchable:

```php
// models/Job.php

class Job extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 255);
        $this->hasColumn('description', 'clob');
    }

    public function setUp()
    {
        $this->actAs('Searchable', array(
                'fields' => array('title', 'content')
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Job:
  actAs:
    Searchable:
      fields: [title, description]
  columns:
    title: string(255)
    description: clob
```

Lets check the SQL that is generated by the above models:

```php
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('Job'));
echo $sql[0] . "\n";
echo $sql[1] . "\n";
echo $sql[2];
```

The above code would output the following SQL query:

```sql
CREATE TABLE job_index (id BIGINT,
keyword VARCHAR(200),
field VARCHAR(50),
position BIGINT,
PRIMARY KEY(id,
keyword,
field,
position)) ENGINE = INNODB
CREATE TABLE job (id BIGINT AUTO_INCREMENT,
title VARCHAR(255),
description LONGTEXT,
PRIMARY KEY(id)) ENGINE = INNODB
ALTER TABLE job_index ADD FOREIGN KEY (id) REFERENCES job(id) ON UPDATE CASCADE ON DELETE CASCADE
```

> Notice how the `job_index` table is automatically created for you and a foreign key between `job` and `job_index` was automatically created.

Because the `Searchable` behavior is such a large topic, we have more information on this that can be found in the Searching chapter.

Geographical

The below is only a demo. The Geographical behavior can be used with any data record for determining the number of miles or kilometers between 2 records.

```
// models/Zipcode.php

class Zipcode extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('zipcode', 'string', 255);
        $this->hasColumn('city', 'string', 255);
        $this->hasColumn('state', 'string', 2);
        $this->hasColumn('county', 'string', 255);
        $this->hasColumn('zip_class', 'string', 255);
    }

    public function setUp()
    {
        $this->actAs('Geographical');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Zipcode:
  actAs: [Geographical]
  columns:
    zipcode: string(255)
    city: string(255)
    state: string(2)
    county: string(255)
    zip_class: string(255)
```

Lets check the SQL that is generated by the above models:

```
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('Zipcode'));
echo $sql[0];
```

The above code would output the following SQL query:

```
CREATE TABLE zipcode (id BIGINT AUTO_INCREMENT,
zipcode VARCHAR(255),
city VARCHAR(255),
state VARCHAR(2),
county VARCHAR(255),
zip_class VARCHAR(255),
latitude DOUBLE,
longitude DOUBLE,
PRIMARY KEY(id)) ENGINE = INNODB
```

> Notice how the Geographical behavior automatically adds the `latitude` and `longitude` columns to the records used for calculating distance between two records. Below you will find some example usage.

First lets retrieve two different zipcode records:

```
// test.php

// ...
$zipcode1 = Doctrine_Core::getTable('Zipcode')->findOneByZipcode('37209');
$zipcode2 = Doctrine_Core::getTable('Zipcode')->findOneByZipcode('37388');
```

Now we can get the distance between those two records by using the `getDistance()` method that the behavior provides:

```
// test.php

// ...
echo $zipcode1->getDistance($zipcode2, $kilometers = false);
```

> The 2nd argument of the `getDistance()` method is whether or not to return the distance in kilometers. The default is false.

Now lets get the 50 closest zipcodes that are not in the same city:

```
// test.php

// ...
$q = $zipcode1->getDistanceQuery();

$q->orderby('miles asc')
  ->addWhere($q->getRootAlias() . '.city != ?', $zipcode1->city)
  ->limit(50);

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
z.id AS z__id,
z.zipcode AS z__zipcode,
z.city AS z__city,
z.state AS z__state,
z.county AS z__county,
z.zip_class AS z__zip_class,
z.latitude AS z__latitude,
z.longitude AS z__longitude,
((ACOS(SIN(* PI() / 180) * SIN(z.latitude * PI() / 180) + COS(* PI() / 180) * COS(z.latitude * PI() / 180) * COS((-
((ACOS(SIN(* PI() / 180) * SIN(z.latitude * PI() / 180) + COS(* PI() / 180) * COS(z.latitude * PI() / 180) * COS((-
FROM zipcode z
WHERE z.city != ?
ORDER BY z__0 asc
LIMIT 50
```

> Notice how the above SQL query includes a bunch of SQL that we did not write. This was automatically added by the behavior to calculate the number of miles between records.

Now we can execute the query and use the calculated number of miles values:

```
// test.php

// ...
$result = $q->execute();

foreach ($result as $zipcode) {
  echo $zipcode->city . " - " . $zipcode->miles . "<br/>";
  // You could also access $zipcode->kilometers
}
```

Get some sample zip code data to test this

http://www.populardata.com/zip_codes.zip

Download and import the csv file with the following function:

```php
// test.php

// ...
function parseCsvFile($file, $columnheadings = false, $delimiter = ',', $enclosure = "\"")
{
    $row = 1;
    $rows = array();
    $handle = fopen($file, 'r');

    while (($data = fgetcsv($handle, 1000, $delimiter, $enclosure)) !== FALSE) {

        if (!($columnheadings == false) && ($row == 1)) {
            $headingTexts = $data;
        } elseif (!($columnheadings == false)) {
            foreach ($data as $key => $value) {
                unset($data[$key]);
                $data[$headingTexts[$key]] = $value;
            }
            $rows[] = $data;
        } else {
            $rows[] = $data;
        }
        $row++;
    }

    fclose($handle);
    return $rows;
}

$array = parseCsvFile('zipcodes.csv', false);

foreach ($array as $key => $value) {
    $zipcode = new Zipcode();
    $zipcode->fromArray($value);
    $zipcode->save();
}
```

SoftDelete

The `SoftDelete` behavior is a very simple yet highly desired model behavior which overrides the `delete()` functionality and adds a `deleted_at` column. When `delete()` is called, instead of deleting the record from the database, a delete_at date is set. Below is an example of how to create a model with the `SoftDelete` behavior being used.

```php
// models/SoftDeleteTest.php

class SoftDeleteTest extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', null, array(
                'primary' => true
            )
        );
    }

    public function setUp()
    {
        $this->actAs('SoftDelete');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
SoftDeleteTest:
  actAs: [SoftDelete]
  columns:
    name:
      type: string(255)
      primary: true
```

Lets check the SQL that is generated by the above models:

```php
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('SoftDeleteTest'));
echo $sql[0];
```

The above code would output the following SQL query:

```sql
CREATE TABLE soft_delete_test (name VARCHAR(255),
deleted_at DATETIME DEFAULT NULL,
PRIMARY KEY(name)) ENGINE = INNODB
```

Now lets put the behavior in action.

> You are required to enable DQL callbacks in order for all executed queries to have the dql callbacks executed on them. In the SoftDelete behavior they are used to filter the select statements to exclude all records where the deleted_at flag is set with an additional WHERE condition.

Enable DQL Callbacks

```
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_USE_DQL_CALLBACKS, true);
```

Now save a new record so we can test the `SoftDelete` functionality:

```
// test.php

// ...
$record = new SoftDeleteTest();
$record->name = 'new record';
$record->save();
```

Now when we call `delete()` the `deleted_at` flag will be set to true:

```
// test.php

// ...
$record->delete();

print_r($record->toArray());
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [name] => new record
    [deleted_at] => 2009-09-01 00:59:01
)
```

Also, when we select some data the query is modified for you:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('SoftDeleteTest t');

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
s.name AS s__name,
s.deleted_at AS s__deleted_at
FROM soft_delete_test s
WHERE (s.deleted_at IS NULL)
```

> Notice how the where condition is automatically added to only return the records that have not been deleted.

Now if we execute the query:

```
// test.php

// ...
$count = $q->count();
echo $count;
```

The above would be echo 0 because it would exclude the record saved above because the delete flag was set.

Nesting Behaviors

Below is an example of several behaviors to give a complete wiki database that is versionable, searchable, sluggable, and full I18n.

```php
class Wiki extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 255);
        $this->hasColumn('content', 'string');
    }

    public function setUp()
    {
        $options = array('fields' => array('title', 'content'));
        $auditLog = new Doctrine_Template_Versionable($options);
        $search = new Doctrine_Template_Searchable($options);
        $slug = new Doctrine_Template_Sluggable(array(
                'fields' => array('title')
            )
        );
        $i18n = new Doctrine_Template_I18n($options);

        $i18n->addChild($auditLog)
             ->addChild($search)
             ->addChild($slug);

        $this->actAs($i18n);

        $this->actAs('Timestampable');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
WikiTest:
  actAs:
    I18n:
      fields: [title, content]
      actAs:
        Versionable:
          fields: [title, content]
        Searchable:
          fields: [title, content]
        Sluggable:
          fields: [title]
  columns:
    title: string(255)
    content: string
```

> The above example of nesting behaviors is currently broken in Doctrine. We are working furiously to come up with a backwards compatible fix. We will announce when the fix is ready and update the documentation accordingly.

Generating Files

By default with behaviors the classes which are generated are evaluated at run-time and no files containing the classes are ever written to disk. This can be changed with a configuration option. Below is an example of how to configure the I18n behavior to generate the classes and write them to files instead of evaluating them at run-time.

```php
class NewsArticle extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 255);
        $this->hasColumn('body', 'string', 255);
        $this->hasColumn('author', 'string', 255);
    }

    public function setUp()
    {
        $this->actAs('I18n', array(
                'fields'           => array('title', 'body'),
                'generateFiles'    => true,
                'generatePath'     => '/path/to/generate'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
NewsArticle:
  actAs:
    I18n:
      fields: [title, body]
      generateFiles: true
      generatePath: /path/to/generate
  columns:
    title: string(255)
    body: string(255)
    author: string(255)
```

Now the behavior will generate a file instead of generating the code and using **eval()** to evaluate it at runtime.

Querying Generated Classes

If you want to query the auto generated models you will need to make sure the model with the behavior attached is loaded and initialized. You can do this by using the static **Doctrine_Core::initializeModels()** method.

For example if you want to query the translation table for a **BlogPost** model you will need to run the following code:

```php
Doctrine_Core::initializeModels(array('BlogPost'));

$q = Doctrine_Query::create()
    ->from('BlogPostTranslation t')
    ->where('t.id = ? AND t.lang = ?', array(1, 'en'));

$translations = $q->execute();
```

> This is required because the behaviors are not instantiated until the model is instantiated for the first time. The above **initializeModels()** method instantiates the passed models and makes sure the information is properly loaded in to the array of loaded models.

Conclusion

By now we should know a lot about Doctrine behaviors. We should know how to write our own for our models as well as how to use all the great behaviors that come bundled with Doctrine.

Now we are ready to move on to discuss the Searchable behavior in more detail in the Searching chapter. As it is a large topic we have devoted an entire chapter to it.

Searching

Introduction

Searching is a huge topic, hence an entire chapter has been devoted to a behavior called `Searchable`. It is a fulltext indexing and searching tool. It can be used for indexing and searching both database and files.

Consider we have a class called `NewsItem` with the following definition:

```
// models/NewsItem.php

class NewsItem extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 255);
        $this->hasColumn('body', 'clob');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
NewsItem:
  columns:
    title: string(255)
    body: clob
```

Now lets say we have an application where users are allowed to search for different news items, an obvious way to implement this would be building a form and based on that forms submitted values build DQL queries such as:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('NewsItem i')
    ->where('n.title LIKE ? OR n.content LIKE ?');
```

As the application grows these kind of queries become very slow. For example when using the previous query with parameters `%framework%` and `%framework%` (this would be equivalent of 'find all news items whose title or content contains word 'framework') the database would have to traverse through each row in the table, which would naturally be very very slow.

Doctrine solves this with its search component and inverse indexes. First lets alter our definition a bit:

```
// models/NewsItem.php

class NewsItem extends Doctrine_Record
{
    // ...

    public function setUp()
    {
        $this->actAs('Searchable', array(
                'fields' => array('title', 'content')
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
NewsItem:
  actAs:
    Searchable:
      fields: [title, content]
# ...
```

Lets check the SQL that is generated by the above models:

```
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('NewsItem'));
echo $sql[0] . "\n";
echo $sql[1] . "\n";
echo $sql[2];
```

The above code would output the following SQL query:

```
CREATE TABLE news_item_index (id BIGINT,
keyword VARCHAR(200),
field VARCHAR(50),
position BIGINT,
PRIMARY KEY(id,
keyword,
field,
position)) ENGINE = INNODB
CREATE TABLE news_item (id BIGINT AUTO_INCREMENT,
title VARCHAR(255),
body LONGTEXT,
PRIMARY KEY(id)) ENGINE = INNODB
ALTER TABLE news_item_index ADD FOREIGN KEY (id) REFERENCES news_item(id) ON UPDATE CASCADE ON DELETE CASCADE
```

Here we tell Doctrine that `NewsItem` class acts as searchable (internally Doctrine loads `Doctrine_Template_Searchable`) and fields `title` and `content` are marked as fulltext indexed fields. This means that every time a `NewsItem` is added or updated Doctrine will:

1. Update the inverse search index or
2. Add new pending entry to the inverse search index (sometimes it can be efficient to update the inverse search index in batches)

Index structure

The structure of the inverse index Doctrine uses is the following:

[(string) keyword] [(string) field] [(integer) position] [(mixed) [foreign_keys]]

Column	Description
`keyword`	**The keyword in the text that can be searched for.**
`field`	**The field where the keyword was found.**
`position`	**The position where the keyword was found.**
`[foreign_keys]`	**The foreign keys of the record being indexed.**

In the `NewsItem` example the `[foreign_keys]` would simply contain one field named `id` with foreign key references to `NewsItem(id)` and with `onDelete => CASCADE` constraint.

An example row in this table might look something like:

keyword	fi eld	position	id
`database`	`title`	`3`	`1`

In this example the word `database` is the third word of the `title` field of `NewsItem` with id of `1`.

Index Building

Whenever a searchable record is being inserted into database Doctrine executes the index building procedure. This happens in the background as the procedure is being invoked by the search listener. The phases of this procedure are:

1. Analyze the text using a `Doctrine_Search_Analyzer` based class
2. Insert new rows into index table for all analyzed keywords

Sometimes you may not want to update the index table directly when new searchable entries are added. Rather you may want to batch update the index table in certain intervals. For disabling the direct update functionality you'll need to set the batchUpdates option to true when you attach the behavior:

```php
// models/NewsItem.php

class NewsItem extends Doctrine_Record
{
    // ...

    public function setUp()
    {
        $this->actAs('Searchable', array(
                'fields' => array('title', 'content')
                'batchUpdates' => true
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
# schema.yml

# ...
NewsItem:
  actAs:
    Searchable:
      fields: [title, content]
      batchUpdates: true
# ...
```

The actual batch updating procedure can be invoked with the `batchUpdateIndex()` method. It takes two optional arguments: `limit` and `offset`. Limit can be used for limiting the number of batch indexed entries while the offset can be used for setting the first entry to start the indexing from.

First lets insert a new `NewsItem` records:

```php
// test.php

// ...
$newsItem = new NewsItem();
$newsItem->title = 'Test';
$newsItem->body = 'test';
$newsItem->save();
```

> If you don't have batch updates enabled then the index will be automatically updated for you when you insert or update `NewsItem` records. If you do have batch updates enabled then you can perform the batch updates by using the following code:

```php
// test.php

// ...
$newsItemTable = Doctrine_Core::getTable('NewsItem');
$newsItemTable->batchUpdateIndex();
```

Text Analyzers

By default Doctrine uses `Doctrine_Search_Analyzer_Standard` for analyzing the text. This class performs the following things:

- Strips out stop-keywords (such as 'and', 'if' etc.) As many commonly used words such as 'and', 'if' etc. have no relevance for the search, they are being stripped out in order to keep the index size reasonable.
- Makes all keywords lowercased. When searching words 'database' and 'DataBase' are considered equal by the standard analyzer, hence the standard analyzer lowercases all keywords.
- Replaces all non alpha-numeric marks with whitespace. In normal text many keywords might contain non alpha-numeric chars after them, for example 'database.'. The standard analyzer strips these out so that 'database' matches 'database.'.
- Replaces all quotation marks with empty strings so that "O'Connor" matches "oconnor"

You can write your own analyzer class by making a class that implements `Doctrine_Search_Analyzer_Interface`. Here is an example where we create an analyzer named `MyAnalyzer`:

```php
// models/MyAnalyzer.php

class MyAnalyzer implements Doctrine_Search_Analyzer_Interface
{
    public function analyze($text)
    {
        $text = trim($text);
        return $text;
    }
}
```

> The search analyzers must only contain one method named `analyze()` and it should return the modified inputted text to be indexed.

This analyzer can then be applied to the search object as follows:

```
// test.php

// ...
$newsItemTable = Doctrine_Core::getTable('NewsItem');
$search = $newsItemTable
    ->getTemplate('Doctrine_Template_Searchable')
    ->getPlugin();

$search->setOption('analyzer', new MyAnalyzer());
```

Query language

`Doctrine_Search` provides a query language similar to Apache Lucene. The `Doctrine_Search_Query` converts human readable, easy-to-construct search queries to their complex DQL equivalents which are then converted to SQL like normal.

Performing Searches

Here is a simple example to retrieve the record ids and relevance data.

```
// test.php

// ...
$newsItemTable = Doctrine_Core::getTable('NewsItem');

$results = $newsItemTable->search('test');
print_r($results);
```

The above code executes the following query:

```
SELECT
COUNT(keyword) AS relevance,
id
FROM article_index
WHERE id IN (SELECT
id
FROM article_index
WHERE keyword = ?)
AND id IN (SELECT
id
FROM article_index
WHERE keyword = ?)
GROUP BY id
ORDER BY relevance DESC
```

The output of the code above would be the following:

```
$ php test.php
Array
(
    [0] => Array
        (
            [relevance] => 1
            [id] => 1
        )

)
```

Now you can use those results in another query to retrieve the actual `NewsItem` objects:

```
// test.php

// ...
$ids = array();
foreach ($results as $result) {
    $ids[] = $result['id'];
}

$q = Doctrine_Query::create()
    ->from('NewsItem i')
    ->whereIn('i.id', $ids);

$newsItems = $q->execute();

print_r($newsItems->toArray());
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [0] => Array
        (
            [id] => 1
            [title] => Test
            [body] => test
        )

)
```

You can optionally pass the `search()` function a query object to modify with a where condition subquery to limit the results using the search index.

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->from('NewsItem i');

$q = Doctrine_Core::getTable('Article')
    ->search('test', $q);

echo $q->getSqlQuery();
```

The above call to `getSql()` would output the following SQL query:

```
SELECT
n.id AS n__id,
n.title AS n__title,
n.body AS n__body
FROM news_item n
WHERE n.id IN (SELECT
id
FROM news_item_index
WHERE keyword = ?
GROUP BY id)
```

Now we can execute the query and get the `NewsItem` objects:

```
// test.php

// ...
$newsItems = $q->execute();

print_r($newsItems->toArray());
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [0] => Array
        (
            [id] => 1
            [title] => Test
            [body] => test
        )

)
```

File searches

As stated before `Doctrine_Search` can also be used for searching files. Lets say we have a directory which we want to be searchable. First we need to create an instance of `Doctrine_Search_File` which is a child of `Doctrine_Search` providing some extra functionality needed for the file searches.

```
// test.php

// ...
$search = new Doctrine_Search_File();
```

Second thing to do is to generate the index table. By default Doctrine names the database index class as `FileIndex`.

Lets check the SQL that is generated by the above models created:

```
// test.php

// ...
$sql = Doctrine_Core::generateSqlFromArray(array('FileIndex'));
```

The above code would output the following SQL query:

```
CREATE TABLE file_index (url VARCHAR(255),
keyword VARCHAR(200),
field VARCHAR(50),
position BIGINT,
PRIMARY KEY(url,
keyword,
field,
position)) ENGINE = INNODB
```

You can create the actual table in the database by using the `Doctrine_Core::createTablesFromArray()` method:

```
// test.php

// ...
Doctrine_Core::createTablesFromArray(array('FileIndex'));
```

Now we can start using the file searcher. In this example lets just index the `models` directory:

```
// test.php

// ...
$search->indexDirectory('models');
```

The `indexDirectory()` iterates recursively through given directory and analyzes all files within it updating the index table as necessary.

Finally we can start searching for pieces of text within the indexed files:

```
// test.php

// ...
$results = $search->search('hasColumn');
print_r($results);
```

The above example would produce the following output:

```
$ php test.php
Array
(
    [0] => Array
        (
            [relevance] => 2
            [url] => models/generated/BaseNewsItem.php
        )

)
```

Conclusion

Now that we have learned all about the `Searchable` behavior we are ready to learn in more detail about the `NestedSet` behavior in the Hierarchical Data chapter. The `NestedSet` is a large topic like the `Searchable` behavior so it got its own dedicated chapter as well.

Hierarchical Data

Introduction

Most users at one time or another have dealt with hierarchical data in a SQL database and no doubt learned that the management of hierarchical data is not what a relational database is intended for. The tables of a relational database are not hierarchical (like XML), but are simply a flat list. Hierarchical data has a parent-child relationship that is not naturally represented in a relational database table.

For our purposes, hierarchical data is a collection of data where each item has a single parent and zero or more children (with the exception of the root item, which has no parent). Hierarchical data can be found in a variety of database applications, including forum and mailing list threads, business organization charts, content management categories, and product categories.

In a hierarchical data model, data is organized into a tree-like structure. The tree structure allows repeating information using parent/child relationships. For an explanation of the tree data structure, see here.

There are three major approaches to managing tree structures in relational databases, these are:

- the adjacency list model
- the nested set model (otherwise known as the modified pre-order tree traversal algorithm)
- materialized path model

These are explained in more detail at the following links:

- http://www.dbazine.com/oracle/or-articles/tropashko4
- http://dev.mysql.com/tech-resources/articles/hierarchical-data.html

Nested Set

Introduction

Nested Set is a solution for storing hierarchical data that provides very fast read access. However, updating nested set trees is more costly. Therefore this solution is best suited for hierarchies that are much more frequently read than written to. And because of the nature of the web, this is the case for most web applications.

For more detailed information on the Nested Set, read here:

- http://www.sitepoint.com/article/hierarchical-data-database/2
- http://dev.mysql.com/tech-resources/articles/hierarchical-data.html

Setting Up

To set up your model as Nested Set, you must add some code to the `setUp()` method of your model. Take this `Category` model below for example:

```
// models/Category.php
class Category extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 255);
    }

    public function setUp()
    {
        $this->actAs('NestedSet');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Category:
  actAs: [NestedSet]
  columns:
    name: string(255)
```

Detailed information on Doctrine's templating model can be found in chapter 16 Behaviors. These templates add some functionality to your model. In the example of the nested set, your model gets 3 additional fields: `lft`, `rgt` and `level`. You never need to care about the `lft` and `rgt` fields. These are used internally to manage the tree structure. The `level` field however, is of interest for you because it's an integer value that represents the depth of a node within it's tree. A level of 0 means it's a root node. 1 means it's a direct child of a root node and so on. By reading the `level` field from your nodes you can easily display your tree with proper indention.

> You must never assign values to `lft`, `rgt`, `level`. These are managed transparently by the nested set implementation.

Multiple Trees

The nested set implementation can be configured to allow your table to have multiple root nodes, and therefore multiple trees within the same table.

The example below shows how to setup and use multiple roots with the `Category` model:

```
// models/Category.php

class Category extends Doctrine_Record
{
    // ...

    public function setUp()
    {
        $options = array(
            'hasManyRoots'     => true,
            'rootColumnName'   => 'root_id'
        );
        $this->actAs('NestedSet', $options);
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
Category:
  actAs:
    NestedSet:
      hasManyRoots: true
      rootColumnName: root_id
  columns:
    name: string(255)
```

The `rootColumnName` is the column used to differentiate between trees. When you create a new root node you have the option to set the `root_id` manually, otherwise Doctrine will assign a value for you.

In general use you do not need to deal with the `root_id` directly. For example, when you insert a new node into an existing tree or move a node between trees Doctrine transparently handles the associated `root_id` changes for you.

Working with Trees

After you successfully set up your model as a nested set you can start working with it. Working with Doctrine's nested set implementation is all about two classes: `Doctrine_Tree_NestedSet` and `Doctrine_Node_NestedSet`. These are nested set implementations of the interfaces `Doctrine_Tree_Interface` and `Doctrine_Node_Interface`. Tree objects are bound to your table objects and node objects are bound to your record objects. This looks as follows:

The full tree interface is available by using the following code:

```
// test.php

// ...
$treeObject = Doctrine_Core::getTable('Category')->getTree();
```

In the next example `$category` is an instance of `Category`:

```php
// test.php

// ...
$nodeObject = $category->getNode();
```

With the above code the full node interface is available on `$nodeObject`.

In the following sub-chapters you'll see code snippets that demonstrate the most frequently used operations with the node and tree classes.

Creating a Root Node

```php
// test.php

// ...
$category = new Category();
$category->name = 'Root Category 1';
$category->save();

$treeObject = Doctrine_Core::getTable('Category')->getTree();
$treeObject->createRoot($category);
```

Inserting a Node

In the next example we're going to add a new `Category` instance as a child of the root `Category` we created above:

```php
// test.php

// ...
$child1 = new Category();
$child1->name = 'Child Category 1';

$child2 = new Category();
$child2->name = 'Child Category 1';

$child1->getNode()->insertAsLastChildOf($category);
$child2->getNode()->insertAsLastChildOf($category);
```

Deleting a Node

Deleting a node from a tree is as simple as calling the `delete()` method on the node object:

```php
// test.php

// ...
$category = Doctrine_Core::getTable('Category')->findOneByName('Child Category 1');
$category->getNode()->delete();
```

> The above code calls `$category->delete()` internally. It's important to delete on the node and not on the record. Otherwise you may corrupt the tree.

Deleting a node will also delete all descendants of that node. So make sure you move them elsewhere before you delete the node if you don't want to delete them.

Moving a Node

Moving a node is simple. Doctrine offers several methods for moving nodes around between trees:

```php
// test.php

// ...
$category = new Category();
$category->name = 'Root Category 2';
$category->save();

$categoryTable = Doctrine_Core::getTable('Category');

$treeObject = $categoryTable->getTree();
$treeObject->createRoot($category);

$childCategory = $categoryTable->findOneByName('Child Category 1');
$childCategory->getNode()->moveAsLastChildOf($category);
...
```

Below is a list of the methods available for moving nodes around:

- moveAsLastChildOf($other)
- moveAsFirstChildOf($other)
- moveAsPrevSiblingOf($other)

- moveAsNextSiblingOf($other).

The method names should be self-explanatory to you.

Examining a Node

You can examine the nodes and what type of node they are by using some of the following functions:

```
// test.php

// ...
$isLeaf = $category->getNode()->isLeaf();
$isRoot = $category->getNode()->isRoot();
```

> The above used functions return true/false depending on whether or not they are a leaf or root node.

Examining and Retrieving Siblings

You can easily check if a node has any next or previous siblings by using the following methods:

```
// test.php

// ...
$hasNextSib = $category->getNode()->hasNextSibling();
$hasPrevSib = $category->getNode()->hasPrevSibling();
```

You can also retrieve the next or previous siblings if they exist with the following methods:

```
// test.php

// ...
$nextSib = $category->getNode()->getNextSibling();
$prevSib = $category->getNode()->getPrevSibling();
```

> The above methods return false if no next or previous sibling exists.

If you want to retrieve an array of all the siblings you can simply use the `getSiblings()` method:

```
// test.php

// ...
$siblings = $category->getNode()->getSiblings();
```

Examining and Retrieving Descendants

You can check if a node has a parent or children by using the following methods:

```
// test.php

// ...
$hasChildren = $category->getNode()->hasChildren();
$hasParent = $category->getNode()->hasParent();
```

You can retrieve a nodes first and last child by using the following methods:

```
// test.php

// ...
$firstChild = $category->getNode()->getFirstChild();
$lastChild = $category->getNode()->getLastChild();
```

Or if you want to retrieve the parent of a node:

```
// test.php

// ...
$parent = $category->getNode()->getParent();
```

You can get the children of a node by using the following method:

```php
// test.php

// ...
$children = $category->getNode()->getChildren();
```

> The `getChildren()` method returns only the direct descendants. If you want all descendants, use the `getDescendants()` method.

You can get the descendants or ancestors of a node by using the following methods:

```php
// test.php

// ...
$descendants = $category->getNode()->getDescendants();
$ancestors = $category->getNode()->getAncestors();
```

Sometimes you may just want to get the number of children or descendants. You can use the following methods to accomplish this:

```php
// test.php

// ...
$numChildren = $category->getNode()->getNumberChildren();
$numDescendants = $category->getNode()->getNumberDescendants();
```

The `getDescendants()` and `getAncestors()` both accept a parameter that you can use to specify the `depth` of the resulting branch. For example `getDescendants(1)` retrieves only the direct descendants (the descendants that are 1 level below, that's the same as `getChildren()`). In the same fashion `getAncestors(1)` would only retrieve the direct ancestor (the parent), etc. `getAncestors()` can be very useful to efficiently determine the path of this node up to the root node or up to some specific ancestor (i.e. to construct a breadcrumb navigation).

Rendering a Simple Tree

> The next example assumes you have `hasManyRoots` set to false so in order for the below example to work properly you will have to set that option to false. We set the value to true in a earlier section.

```php
// test.php

// ...
$treeObject = Doctrine_Core::getTable('Category')->getTree();
$tree = $treeObject->fetchTree();

foreach ($tree as $node) {
    echo str_repeat('  ', $node['level']) . $node['name'] . "\n";
}
```

Advanced Usage

The previous sections have explained the basic usage of Doctrine's nested set implementation. This section will go one step further.

Fetching a Tree with Relations

If you're a demanding software developer this question may already have come into your mind: "How do I fetch a tree/branch with related data?". Simple example: You want to display a tree of categories, but you also want to display some related data of each category, let's say some details of the hottest product in that category. Fetching the tree as seen in the previous sections and simply accessing the relations while iterating over the tree is possible but produces a lot of unnecessary database queries. Luckily, `Doctrine_Query` and some flexibility in the nested set implementation have come to your rescue. The nested set implementation uses `Doctrine_Query` objects for all it's database work. By giving you access to the base query object of the nested set implementation you can unleash the full power of `Doctrine_Query` while using your nested set.

First lets create the query we want to use to retrieve our tree data with:

```php
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('c.name, p.name, m.name')
    ->from('Category c')
    ->leftJoin('c.HottestProduct p')
    ->leftJoin('p.Manufacturer m');
```

Now we need to set the above query as the base query for the tree:

```php
$treeObject = Doctrine_Core::getTable('Category')->getTree();
$treeObject->setBaseQuery($q);
$tree = $treeObject->fetchTree();
```

There it is, the tree with all the related data you need, all in one query.

> If you don't set your own base query then one will be automatically created for you internally.

When you are done it is a good idea to reset the base query back to normal:

```
// test.php

// ...
$treeObject->resetBaseQuery();
```

You can take it even further. As mentioned in the chapter Improving Performance you should only fetch objects when you need them. So, if we need the tree only for display purposes (read-only) we can use the array hydration to speed things up a bit:

```
// test.php

// ...
$q = Doctrine_Query::create()
    ->select('c.name, p.name, m.name')
    ->from('Category c')
    ->leftJoin('c.HottestProduct p')
    ->leftJoin('p.Manufacturer m')
    ->setHydrationMode(Doctrine_Core::HYDRATE_ARRAY);

$treeObject = Doctrine_Core::getTable('Category')->getTree();
$treeObject->setBaseQuery($q);
$tree = $treeObject->fetchTree();
$treeObject->resetBaseQuery();
```

Now you got a nicely structured array in `$tree` and if you use array access on your records anyway, such a change will not even effect any other part of your code. This method of modifying the query can be used for all node and tree methods (`getAncestors()`, `getDescendants()`, `getChildren()`, `getParent()`, ...). Simply create your query, set it as the base query on the tree object and then invoke the appropriate method.

Rendering with Indention

Below you will find an example where all trees are rendered with proper indention. You can retrieve the roots using the `fetchRoots()` method and retrieve each individual tree by using the `fetchTree()` method.

```
// test.php

// ...
$treeObject = Doctrine_Core::getTable('Category')->getTree();
$rootColumnName = $treeObject->getAttribute('rootColumnName');

foreach ($treeObject->fetchRoots() as $root) {
  $options = array(
      'root_id' => $root->$rootColumnName
  );
  foreach($treeObject->fetchTree($options) as $node) {
    echo str_repeat(' ', $node['level']) . $node['name'] . "\n";
  }
}
```

After doing all the examples above the code above should render as follows:

```
$ php test.php
Root Category 1
Root Category 2
 Child Category 1
```

Conclusion

Now that we have learned all about the `NestedSet` behavior and how to manage our hierarchical data using Doctrine we are ready to learn about Data Fixtures. Data fixtures are a great tool for loading small sets of test data in to your applications to be used for unit and functional tests or to populate your application with its initial data.

Data Fixtures

Data fixtures are meant for loading small sets of test data through your models to populate your database with data to test against. The data fixtures are often used side by side with some kind of unit/functional testing suite.

Importing

Importing data fixtures is just as easy as dumping. You can use the `loadData()` function:

```
Doctrine_Core::loadData('/path/to/data.yml');
```

You can either specify an individual yml file like we have done above, or you can specify an entire directory:

```
Doctrine_Core::loadData('/path/to/directory');
```

If you want to append the imported data to the already existing data then you need to use the second argument of the `loadData()` function. If you don't specify the second argument as true then the data will be purged before importing.

Here is how you can append instead of purging:

```
Doctrine_Core::loadData('/path/to/data.yml', true);
```

Dumping

You can dump data to fixtures file in many different formats to help you get started with writing your data fixtures. You can dump your data fixtures to one big YAML file like the following:

```
Doctrine_Core::dumpData('/path/to/data.yml');
```

Or you can optionally dump all data to individual files. One YAML file per model like the following:

```
Doctrine_Core::dumpData('/path/to/directory', true);
```

Implement

Now that we know a little about data fixtures lets implement them in to our test environment we created and have been using through the previous chapters so that we can test the example fixtures used in the next sections.

First create a directory in your `doctrine_test` directory named `fixtures` and create a file named `data.yml` inside:

```
$ mkdir fixtures
$ touch fixtures/data.yml
```

Now we need to just modify our `generate.php` script to include the code for loading the data fixtures. Add the following code to the bottom of `generate.php`:

```
// generate.php

// ...
Doctrine_Core::loadData('fixtures');
```

Writing

You can write your fixtures files manually and load them in to your applications. Below is a sample `data.yml` fixtures file. You can also split your data fixtures file up in to multiple files. Doctrine will read all fixtures files and parse them, then load all data.

For the next several examples we will use the following models:

```php
// models/Resouce.php

class Resource extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 255);
        $this->hasColumn('resource_type_id', 'integer');
    }

    public function setUp()
    {
        $this->hasOne('ResourceType as Type', array(
                'local' => 'resource_type_id',
                'foreign' => 'id'
            )
        );

        $this->hasMany('Tag as Tags', array(
                'local' => 'resource_id',
                'foreign' => 'tag_id',
                'refClass' => 'ResourceTag'
            )
        );
    }
}

// models/ResourceType.php

class ResourceType extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 255);
    }

    public function setUp()
    {
        $this->hasMany('Resource as Resouces', array(
                'local' => 'id',
                'foreign' => 'resource_type_id'
            )
        );
    }
}

// models/Tag.php

class Tag extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 255);
    }

    public function setUp()
    {
        $this->hasMany('Resource as Resources', array(
                'local' => 'tag_id',
                'foreign' => 'resource_id',
                'refClass' => 'ResourceTag'
            )
        );
    }
}

// models/ResourceTag.php

class ResourceTag extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('resource_id', 'integer');
        $this->hasColumn('tag_id', 'integer');
    }
}

// models/Category.php

class BaseCategory extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 255, array(
                'type' => 'string',
                'length' => '255'
            )
        );
    }

    public function setUp()
    {
        $this->actAs('NestedSet');
    }
}

class BaseArticle extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 255, array(
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

Resource:
  columns:
    name: string(255)
    resource_type_id: integer
  relations:
    Type:
      class: ResourceType
      foreignAlias: Resources
    Tags:
      class: Tag
      refClass: ResourceTag
      foreignAlias: Resources

ResourceType:
  columns:
    name: string(255)

Tag:
  columns:
   name: string(255)

ResourceTag:
  columns:
    resource_id: integer
    tag_id: integer

Category:
  actAs: [NestedSet]
  columns:
    name: string(255)

Article:
  actAs:
    I18n:
      fields: [title, body]
  columns:
    title: string(255)
    body: clob
```

All row keys across all YAML data fixtures must be unique. For example below tutorial, doctrine, help, cheat are all unique.

```
---
# fixtures/data.yml

Resource:
  Resource_1:
    name: Doctrine Video Tutorial
    Type: Video
    Tags: [tutorial, doctrine, help]
  Resource_2:
    name: Doctrine Cheat Sheet
    Type: Image
    Tags: [tutorial, cheat, help]

ResourceType:
  Video:
    name: Video
  Image:
    name: Image

Tag:
  tutorial:
    name: tutorial
  doctrine:
    name: doctrine
  help:
    name: help
  cheat:
    name: cheat
```

You could optionally specify the Resources each tag is related to instead of specifying the Tags a Resource has.

```yaml
---
# fixtures/data.yml

# ...
Tag:
  tutorial:
    name: tutorial
    Resources: [Resource_1, Resource_2]
  doctrine:
    name: doctrine
    Resources: [Resource_1]
  help:
    name: help
    Resources: [Resource_1, Resource_2]
  cheat:
    name: cheat
    Resources: [Resource_1]
```

Fixtures For Nested Sets

Writing a fixtures file for a nested set tree is slightly different from writing regular fixtures files. The structure of the tree is defined like the following:

```yaml
---
# fixtures/data.yml

# ...
Category:
  Category_1:
    name: Categories # the root node
    children:
      Category_2:
        name: Category 1
      Category_3:
        name: Category 2
        children:
          Category_4:
            name: Subcategory of Category 2
```

> When writing data fixtures for the NestedSet you must either specify at least a `children` element of the first data block or specify `NestedSet: true` under the model which is a NestedSet in order for the data fixtures to be imported using the NestedSet api.

```yaml
---
# fixtures/data.yml

# ...
Category:
  NestedSet: true
  Category_1:
    name: Categories
# ...
```

Or simply specifying the children keyword will make the data fixtures importing using the NestedSet api.

```yaml
---
# fixtures/data.yml

# ...
Category:
  Category_1:
    name: Categories
    children: []
# ...
```

If you don't use one of the above methods then it is up to you to manually specify the lft, rgt and level values for your nested set records.

Fixtures For I18n

The fixtures for the `I18n` aren't anything custom since the `I18n` really is just a normal set of relationships that are built on the fly dynamically:

```yaml
---
# fixtures/data.yml

# ...
Article:
  Article_1:
    Translation:
      en:
        title: Title of article
        body: Body of article
      fr:
        title: French title of article
        body: French body of article
```

Conclusion

By now we should be able to write and load our own data fixtures in our application. So, now we will move on to learning about the underlying Database Abstraction Layer in Doctrine. This layer is what makes all the previously discussed functionality possible. You can use this layer standalone apart from the ORM. In the next chapter we'll explain how you can use the DBAL by itself.

Database Abstraction Layer

The Doctrine Database Abstraction Layer is the underlying framework that the ORM uses to communicate with the database and send the appropriate SQL depending on which database type you are using. It also has the ability to query the database for information like what table a database has or what fields a table has. This is how Doctrine is able to generate your models from existing databases so easily.

This layer can be used independently of the ORM. This might be of use for example if you have an existing application that uses PDO directly and you want to port it to use the Doctrine Connections and DBAL. At a later phase you could begin to use the ORM for new things and rewrite old pieces to use the ORM.

The DBAL is composed of a few different modules. In this chapter we will discuss the different modules and what their jobs are.

Export

The Export module provides methods for managing database structure. The methods can be grouped based on their responsibility: create, edit (alter or update), list or delete (drop) database elements. The following document lists the available methods, providing examples of their use.

Introduction

Every schema altering method in the Export module has an equivalent which returns the SQL that is used for the altering operation. For example `createTable()` executes the query / queries returned by `createTableSql()`.

In this chapter the following tables will be created, altered and finally dropped, in a database named `events_db`:

events

Name	Type	Primary	Auto Increment
id	integer	true	true
name	string(255)	false	false
datetime	timestamp	false	false

people

Name	Type	Primary	Auto Increment
id	integer	true	true
name	string(255)	false	false

event_participants

Name	Type	Primary	Auto Increment
event_id	integer	true	false
person_id	string(255)	true	false

Creating Databases

It is simple to create new databases with Doctrine. It is only a matter of calling the `createDatabase()` function with an argument that is the name of the database to create.

```
// test.php
// ...
$conn->export->createDatabase('events_db');
```

Now lets change the connection in our `bootstrap.php` file to connect to the new `events_db`:

```php
// bootstrap.php

/**
 * Bootstrap Doctrine.php, register autoloader and specify
 * configuration attributes
 */

// ...

$conn = Doctrine_Manager::connection('mysql://root:@localhost/events_db', 'doctrine');

// ...
```

Creating Tables

Now that the database is created and we've re-configured our connection, we can proceed with adding some tables. The method `createTable()` takes three parameters: the table name, an array of field definition and some extra options (optional and RDBMS-specific).

Now lets create the `events` table:

```php
// test.php

//
$definition = array(
    'id' => array(
        'type' => 'integer',
        'primary' => true,
        'autoincrement' => true
    ),
    'name' => array(
        'type' => 'string',
        'length' => 255
    ),
    'datetime' => array(
        'type' => 'timestamp'
    )
);

$conn->export->createTable('events', $definition);
```

The keys of the definition array are the names of the fields in the table. The values are arrays containing the required key `type` as well as other keys, depending on the value of `type`. The values for the `type` key are the same as the possible Doctrine datatypes. Depending on the datatype, the other options may vary.

Datatype	length	default	not null	unsigned	autoincrement
`string`	x	x	x		
`boolean`		x	x		
`integer`	x	x	x	x	x
`decimal`		x	x		
`float`		x	x		
`timestamp`		x	x		
`time`		x	x		
`date`		x	x		
`clob`	x		x		
`blob`	x		x		

And now we can go ahead and create the `people` table:

```php
// test.php

// ...
$definition = array(
    'id' => array(
        'type' => 'integer',
        'primary' => true,
        'autoincrement' => true
    ),
    'name' => array(
        'type' => 'string',
        'length' => 255
    )
);

$conn->export->createTable('people', $definition);
```

You can also specify an array of options as the third argument to the `createTable()` method:

```php
// test.php

// ...
$options = array(
    'comment' => 'Repository of people',
    'character_set' => 'utf8',
    'collate' => 'utf8_unicode_ci',
    'type'    => 'innodb',
);

// ...

$conn->export->createTable('people', $definition, $options);
```

Creating Foreign Keys

Creating the `event_participants` table with a foreign key:

```php
// test.php

// ...
$options = array(
    'foreignKeys' => array(
        'events_id_fk' => array(
            'local'   => 'event_id',
            'foreign' => 'id',
            'foreignTable' => 'events',
            'onDelete' => 'CASCADE',
        )
    ),
    'primary' => array('event_id', 'person_id'),
);

$definition = array(
    'event_id' => array(
        'type' => 'integer',
        'primary' => true
    ),
    'person_id' => array(
        'type' => 'integer',
        'primary' => true
    ),
);

$conn->export->createTable('event_participants', $definition, $options);
```

> In the above example notice how we omit a foreign key for the `person_id`. In that example we omit it so we can show you how to add an individual foreign key to a table in the next example. Normally it would be best to have both foreign keys defined on the in the `foreignKeys`.

Now lets add the missing foreign key in the `event_participants` table the on `person_id` column:

```php
// test.php

// ...
$definition = array('local'   => 'person_id',
                    'foreign' => 'id',
                    'foreignTable' => 'people',
                    'onDelete' => 'CASCADE');

$conn->export->createForeignKey('event_participants', $definition);
```

Altering table

`Doctrine_Export` drivers provide an easy database portable way of altering existing database tables.

```php
// test.php

// ...
$alter  = array(
    'add' => array(
        'new_column' => array(
            'type' => 'string',
            'length' => 255
        ),
        'new_column2' => array(
            'type' => 'string',
            'length' => 255
        )
    )
);

echo $conn->export->alterTableSql('events', $alter);
```

The above call to `alterTableSql()` would output the following SQL query:

```
ALTER TABLE events ADD new_column VARCHAR(255),
ADD new_column2 VARCHAR(255)
```

> If you only want execute generated sql and not return it, use the `alterTable()` method.

```
// test.php
// ...
$conn->export->alterTable('events', $alter);
```

The `alterTable()` method requires two parameters and has an optional third:

Name	Type	Description
$name	string	Name of the table that is intended to be changed.
$changes	array	Associative array that contains the details of each type of change that is intended to be performed.

An optional third parameter (default: false):

Name	Type	Description
$check	boolean	Check if the DBMS can actually perform the operation before executing.

The types of changes that are currently supported are defined as follows:

Change	Description
name	New name for the table.
add	Associative array with the names of fields to be added as indexes of the array. The value of each entry of the array should be set to another associative array with the properties of the fields to be added. The properties of the fields should be the same as defined by the Doctrine parser.
remove	Associative array with the names of fields to be removed as indexes of the array. Currently the values assigned to each entry are ignored. An empty array should be used for future compatibility.
rename	Associative array with the names of fields to be renamed as indexes of the array. The value of each entry of the array should be set to another associative array with the entry named name with the new field name and the entry named Declaration that is expected to contain the portion of the field declaration already in DBMS specific SQL code as it is used in the `CREATE TABLE` statement.
change	Associative array with the names of the fields to be changed as indexes of the array. Keep in mind that if it is intended to change either the name of a field and any other properties, the change array entries should have the new names of the fields as array indexes.

The value of each entry of the array should be set to another associative array with the properties of the fields to that are meant to be changed as array entries. These entries should be assigned to the new values of the respective properties. The properties of the fields should be the same as defined by the Doctrine parser.

```php
// test.php

// ...
$alter = array('name' => 'event',
               'add' => array(
                        'quota' => array(
                            'type' => 'integer',
                            'unsigned' => 1
                            )
                        ),
               'remove' => array(
                        'new_column2' => array()
                        ),
               'change' => array(
                        'name' => array(
                            'length' => '20',
                            'definition' => array(
                                'type' => 'string',
                                'length' => 20
                                )
                            )
                        ),
               'rename' => array(
                        'new_column' => array(
                            'name' => 'gender',
                            'definition' => array(
                                'type' => 'string',
                                'length' => 1,
                                'default' => 'M'
                                )
                            )
                        )

               );

$conn->export->alterTable('events', $alter);
```

> Notice how we renamed the table to `event`, lets rename it back to `events`. We only renamed it to demonstrate the functionality and we will need the table to be named `events` for the next examples.

```php
// test.php

// ...
$alter = array(
    'name' => 'events'
);

$conn->export->alterTable('event', $alter);
```

Creating Indexes

To create an index, the method `createIndex()` is used, which has similar signature as `createConstraint()`, so it takes table name, index name and a definition array. The definition array has one key named `fields` with a value which is another associative array containing fields that will be a part of the index. The fields are defined as arrays with possible keys: sorting, with values ascending and descending length, integer value

Not all RDBMS will support index sorting or length, in these cases the drivers will ignore them. In the test events database, we can assume that our application will show events occuring in a specific timeframe, so the selects will use the datetime field in `WHERE` conditions. It will help if there is an index on this field.

```php
// test.php

// ...
$definition = array(
    'fields' => array(
        'datetime' => array()
    )
);

$conn->export->createIndex('events', 'datetime', $definition);
```

Deleting database elements

For every `create*()` method as shown above, there is a corresponding `drop*()` method to delete a database, a table, field, index or constraint. The `drop*()` methods do not check if the item to be deleted exists, so it's developer's responsibility to check for exceptions using a try catch block:

```php
// test.php

// ...
try {
    $conn->export->dropSequence('nonexisting');
} catch(Doctrine_Exception $e) {

}
```

You can easily drop a constraint with the following code:

```php
// test.php

// ...
$conn->export->dropConstraint('events', 'PRIMARY', true);
```

> The third parameter gives a hint that this is a primary key constraint.

```php
// test.php

// ...
$conn->export->dropConstraint('event_participants', 'event_id');
```

You can easily drop an index with the following code:

```php
$conn->export->dropIndex('events', 'event_timestamp');
```

> It is recommended to not actually execute the next two examples. In the next section we will need the `events_db` to be intact for our examples to work.

Drop a table from the database with the following code:

```php
// test.php

// ...
$conn->export->dropTable('events');
```

We can drop the database with the following code:

```php
// test.php

// ...
$conn->export->dropDatabase('events_db');
```

Import

The import module allows you to inspect a the contents of a database connection and learn about the databases and schemas in each database.

Introduction

To see what's in the database, you can use the `list*()` family of functions in the Import module.

Name	Description
`listDatabases()`	List the databases
`listFunctions()`	List the available functions.
`listSequences($dbName)`	List the available sequences. Takes optional database name as a parameter. If not supplied, the currently selected database is assumed.
`listTableConstraints($tableName)`	Lists the available tables. takes a table name
`listTableColumns($tableName)`	List the columns available in a table.
`listTableIndexes($tableName)`	List the indexes defined in a table.
`listTables($dbName)`	List the tables in a database.
`listTableTriggers($tableName)`	List the triggers in a table.
`listTableViews($tableName)`	List the views available in a table.
`listUsers()`	List the users for the database.
`listViews($dbName)`	List the views available for a database.

Below you will find examples on how to use the above listed functions:

Listing Databases

```php
// test.php

// ...
$databases = $conn->import->listDatabases();
print_r($databases);
```

Listing Sequences

```
// test.php

// ...
$sequences = $conn->import->listSequences('events_db');
print_r($sequences);
```

Listing Constraints

```
// test.php

// ...
$constraints = $conn->import->listTableConstraints('event_participants');
print_r($constraints);
```

Listing Table Columns

```
// test.php

// ...
$columns = $conn->import->listTableColumns('events');
print_r($columns);
```

Listing Table Indexes

```
// test.php

// ...
$indexes = $conn->import->listTableIndexes('events');
print_r($indexes);
```

Listing Tables

```
$tables = $conn->import->listTables();
print_r($tables);
```

Listing Views

> Currently there is no method to create views, so let's do it manually.

```
$sql = "CREATE VIEW names_only AS SELECT name FROM people";
$conn->exec($sql);

$sql = "CREATE VIEW last_ten_events AS SELECT * FROM events ORDER BY id DESC LIMIT 0,10";
$conn->exec($sql);
```

Now we can list the views we just created:

```
$views = $conn->import->listViews();
print_r($views);
```

DataDict

Introduction

Doctrine uses the `DataDict` module internally to convert native RDBMS types to Doctrine types and the reverse. `DataDict` module uses two methods for the conversions:

- `getPortableDeclaration()`, which is used for converting native RDBMS type declaration to portable Doctrine declaration
- `getNativeDeclaration()`, which is used for converting portable Doctrine declaration to driver specific type declaration

Getting portable declaration

```
// test.php

// ...
$declaration = $conn->dataDict->getPortableDeclaration('VARCHAR(255)');

print_r($declaration);
```

The above example would output the following:

```
$ php test.php
Array
(
    [type] => Array
        (
            [0] => string
        )

    [length] => 255
    [unsigned] =>
    [fixed] =>
)
```

Getting Native Declaration

```
// test.php

// ...
$portableDeclaration = array(
    'type' => 'string',
    'length' => 20,
    'fixed' => true
);

$nativeDeclaration = $conn->dataDict->getNativeDeclaration($portableDeclaration);

echo $nativeDeclaration;
```

The above example would output the following:

```
$ php test.php
CHAR(20)
```

Drivers

Mysql

Setting table type

```
// test.php

// ...
$fields = array(
    'id' => array(
        'type' => 'integer',
        'autoincrement' => true
    ),
    'name' => array(
        'type' => 'string',
        'fixed' => true,
        'length' => 8
    )
);
```

> The following option is mysql specific and skipped by other drivers.

```
$options = array('type' => 'INNODB');

$sql = $conn->export->createTableSql('test_table', $fields);
echo $sql[0];
```

The above will output the following SQL query:

```
CREATE TABLE test_table (id INT AUTO_INCREMENT,
name CHAR(8)) ENGINE = INNODB
```

Conclusion

This chapter is indeed a nice one. The Doctrine DBAL is a great tool all by itself. It is probably one of the most fully featured and easy to use PHP database abstraction layers available today.

Now we are ready to move on and learn about how to use Transactions.

Transactions

Introduction

A database transaction is a unit of interaction with a database management system or similar system that is treated in a coherent and reliable way independent of other transactions that must be either entirely completed or aborted. Ideally, a database system will guarantee all of the ACID (Atomicity, Consistency, Isolation, and Durability) properties for each transaction.

- **Atomicity** refers to the ability of the DBMS to guarantee that either all of the tasks of a transaction are performed or none of them are. The transfer of funds can be completed or it can fail for a multitude of reasons, but atomicity guarantees that one account won't be debited if the other is not credited as well.
- **Consistency** refers to the database being in a legal state when the transaction begins and when it ends. This means that a transaction can't break the rules, or **integrity constraints**, of the database. If an integrity constraint states that all accounts must have a positive balance, then any transaction violating this rule will be aborted.
- **Isolation** refers to the ability of the application to make operations in a transaction appear isolated from all other operations. This means that no operation outside the transaction can ever see the data in an intermediate state; a bank manager can see the transferred funds on one account or the other, but never on both – even if she ran her query while the transfer was still being processed. More formally, isolation means the transaction history (or schedule) is serializable. For performance reasons, this ability is the most often relaxed constraint. See the isolation article for more details.
- **Durability** refers to the guarantee that once the user has been notified of success, the transaction will persist, and not be undone. This means it will survive system failure, and that the database system has checked the integrity constraints and won't need to abort the transaction. Typically, all transactions are written into a log that can be played back to recreate the system to its state right before the failure. A transaction can only be deemed committed after it is safely in the log.

In Doctrine all operations are wrapped in transactions by default. There are some things that should be noticed about how Doctrine works internally:

- Doctrine uses application level transaction nesting.
- Doctrine always executes `INSERT` / `UPDATE` / `DELETE` queries at the end of transaction (when the outermost commit is called). The operations are performed in the following order: all inserts, all updates and last all deletes. Doctrine knows how to optimize the deletes so that delete operations of the same component are gathered in one query.

First we need to begin a new transation:

```
$conn->beginTransaction();
```

Next perform some operations which result in queries being executed:

```
$user = new User();
$user->name = 'New user';
$user->save();

$user = Doctrine_Core::getTable('User')->find(5);
$user->name = 'Modified user';
$user->save();
```

Now we can commit all the queries by using the `commit()` method:

```
$conn->commit();
```

Nesting

You can easily nest transactions with the Doctrine DBAL. Check the code below for a simple example demonstrating nested transactions.

First lets create a standard PHP function named `saveUserAndGroup()`:

```
function saveUserAndGroup(Doctrine_Connection $conn, User $user, Group $group)
{
    $conn->beginTransaction();

    $user->save();

    $group->save();

    $conn->commit();
}
```

Now we make use of the function inside another transaction:

```
try {
    $conn->beginTransaction();

    saveUserAndGroup($conn,$user,$group);
    saveUserAndGroup($conn,$user2,$group2);
    saveUserAndGroup($conn,$user3,$group3);

    $conn->commit();
} catch(Doctrine_Exception $e) {
    $conn->rollback();
}
```

> Notice how the three calls to `saveUserAndGroup()` are wrapped in a transaction, and each call to the function starts its own nested transaction.

Savepoints

Doctrine supports transaction savepoints. This means you can set named transactions and have them nested.

The `Doctrine_Transaction::beginTransaction($savepoint)` sets a named transaction savepoint with a name of `$savepoint`. If the current transaction has a savepoint with the same name, the old savepoint is deleted and a new one is set.

```
try {
    $conn->beginTransaction();
    // do some operations here

    // creates a new savepoint called mysavepoint
    $conn->beginTransaction('mysavepoint');
    try {
        // do some operations here

        $conn->commit('mysavepoint');
    } catch(Exception $e) {
        $conn->rollback('mysavepoint');
    }
    $conn->commit();
} catch(Exception $e) {
    $conn->rollback();
}
```

The `Doctrine_Transaction::rollback($savepoint)` rolls back a transaction to the named savepoint. Modifications that the current transaction made to rows after the savepoint was set are undone in the rollback.

> Mysql, for example, does not release the row locks that were stored in memory after the savepoint.

Savepoints that were set at a later time than the named savepoint are deleted.

The `Doctrine_Transaction::commit($savepoint)` removes the named savepoint from the set of savepoints of the current transaction.

All savepoints of the current transaction are deleted if you execute a commit or if a rollback is being called without savepoint name parameter.

```
try {
    $conn->beginTransaction();
    // do some operations here

    // creates a new savepoint called mysavepoint
    $conn->beginTransaction('mysavepoint');

    // do some operations here

    $conn->commit();   // deletes all savepoints
} catch(Exception $e) {
    $conn->rollback(); // deletes all savepoints
}
```

Isolation Levels

A transaction isolation level sets the default transactional behavior. As the name 'isolation level' suggests, the setting determines how isolated each transation is, or what kind of locks are associated with queries inside a transaction. The four available levels are (in ascending order of strictness):

`READ UNCOMMITTED`
Barely transactional, this setting allows for so-called 'dirty reads', where queries inside one transaction are affected by uncommitted changes in another transaction.

`READ COMMITTED`

Committed updates are visible within another transaction. This means identical queries within a transaction can return differing results. This is the default in some DBMS's.

`REPEATABLE READ`
Within a transaction, all reads are consistent. This is the default of Mysql INNODB engine.

`SERIALIZABLE`
Updates are not permitted in other transactions if a transaction has run an ordinary `SELECT` query.

To get the transaction module use the following code:

```
$tx = $conn->transaction;
```

Set the isolation level to READ COMMITTED:

```
$tx->setIsolation('READ COMMITTED');
```

Set the isolation level to SERIALIZABLE:

```
$tx->setIsolation('SERIALIZABLE');
```

> Some drivers (like Mysql) support the fetching of current transaction isolation level. It can be done as follows:

```
$level = $tx->getIsolation();
```

Conclusion

Transactions are a great feature for ensuring the quality and consistency of your database. Now that you know about transactions we are ready to move on and learn about the events sub-framework.

The events sub-framework is a great feature that allows you to hook in to core methods of Doctrine and alter the operations of internal functionality without modifying one line of core code.

Event Listeners

Introduction

Doctrine provides flexible event listener architecture that not only allows listening for different events but also for altering the execution of the listened methods.

There are several different listeners and hooks for various Doctrine components. Listeners are separate classes whereas hooks are empty template methods within the listened class.

Hooks are simpler than event listeners but they lack the separation of different aspects. An example of using `Doctrine_Record` hooks:

```
// models/BlogPost.php

class BlogPost extends Doctrine_Record
{
    // ...

    public function preInsert($event)
    {
        $invoker = $event->getInvoker();

        $invoker->created = date('Y-m-d', time());
    }
}
```

> By now we have defined lots of models so you should be able to define your own `setTableDefinition()` for the `BlogPost` model or even create your own custom model!

Now you can use the above model with the following code assuming we added a `title`, `body` and `created` column to the model:

```
// test.php

// ...
$blog = new BlogPost();
$blog->title = 'New title';
$blog->body = 'Some content';
$blog->save();

echo $blog->created;
```

The above example will output the current date as PHP knows it.

Each listener and hook method takes one parameter `Doctrine_Event` object. `Doctrine_Event` object holds information about the event in question and can alter the execution of the listened method.

For the purposes of this documentation many method tables are provided with column named `params` indicating names of the parameters that an event object holds on given event. For example the `preCreateSavepoint` event has one parameter with the name of the created `savepoint`, which is quite intuitively named as `savepoint`.

Connection Listeners

Connection listeners are used for listening the methods of `Doctrine_Connection` and its modules (such as `Doctrine_Transaction`). All listener methods take one argument `Doctrine_Event` which holds information about the listened event.

Creating a New Listener

There are three different ways of defining a listener. First you can create a listener by making a class that inherits `Doctrine_EventListener`:

```
class MyListener extends Doctrine_EventListener
{
    public function preExec(Doctrine_Event $event)
    {

    }
}
```

Note that by declaring a class that extends `Doctrine_EventListener` you don't have to define all the methods within the `Doctrine_EventListener_Interface`. This is due to a fact that `Doctrine_EventListener` already has empty skeletons for all these methods.

Sometimes it may not be possible to define a listener that extends `Doctrine_EventListener` (you might have a listener that inherits some other base class). In this case you can make it implement `Doctrine_EventListener_Interface`.

```
class MyListener implements Doctrine_EventListener_Interface
{
    public function preTransactionCommit(Doctrine_Event $event) {}
    public function postTransactionCommit(Doctrine_Event $event) {}

    public function preTransactionRollback(Doctrine_Event $event) {}
    public function postTransactionRollback(Doctrine_Event $event) {}

    public function preTransactionBegin(Doctrine_Event $event) {}
    public function postTransactionBegin(Doctrine_Event $event) {}

    public function postConnect(Doctrine_Event $event) {}
    public function preConnect(Doctrine_Event $event) {}

    public function preQuery(Doctrine_Event $event) {}
    public function postQuery(Doctrine_Event $event) {}

    public function prePrepare(Doctrine_Event $event) {}
    public function postPrepare(Doctrine_Event $event) {}

    public function preExec(Doctrine_Event $event) {}
    public function postExec(Doctrine_Event $event) {}

    public function preError(Doctrine_Event $event) {}
    public function postError(Doctrine_Event $event) {}

    public function preFetch(Doctrine_Event $event) {}
    public function postFetch(Doctrine_Event $event) {}

    public function preFetchAll(Doctrine_Event $event) {}
    public function postFetchAll(Doctrine_Event $event) {}

    public function preStmtExecute(Doctrine_Event $event) {}
    public function postStmtExecute(Doctrine_Event $event) {}
}
```

> All listener methods must be defined here otherwise PHP throws fatal error.

The third way of creating a listener is a very elegant one. You can make a class that implements `Doctrine_Overloadable`. This interface has only one method: `__call()`, which can be used for catching *all* the events.

```
class MyDebugger implements Doctrine_Overloadable
{
    public function __call($methodName, $args)
    {
        echo $methodName . ' called !';
    }
}
```

Attaching listeners

You can attach the listeners to a connection with setListener().

```
$conn->setListener(new MyDebugger());
```

If you need to use multiple listeners you can use addListener().

```
$conn->addListener(new MyDebugger());
$conn->addListener(new MyLogger());
```

Pre and Post Connect

All of the below listeners are invoked in the `Doctrine_Connection` class. And they are all passed an instance of `Doctrine_Event`.

Methods	Listens	Params
`preConnect()`	`connection()`	
`postConnect()`	`connection()`	

Transaction Listeners

All of the below listeners are invoked in the `Doctrine_Transaction` class. And they are all passed an instance of `Doctrine_Event`.

Methods	Listens	Params
preTransactionBegin()	beginTransaction()	
postTransactionBegin()	beginTransaction()	
preTransactionRollback()	rollback()	
postTransactionRollback()	rollback()	
preTransactionCommit()	commit()	
postTransactionCommit()	commit()	
preCreateSavepoint()	createSavepoint()	savepoint
postCreateSavepoint()	createSavepoint()	savepoint
preRollbackSavepoint()	rollbackSavepoint()	savepoint
postRollbackSavepoint()	rollbackSavepoint()	savepoint
preReleaseSavepoint()	releaseSavepoint()	savepoint
postReleaseSavepoint()	releaseSavepoint()	savepoint

```
class MyTransactionListener extends Doctrine_EventListener
{
    public function preTransactionBegin(Doctrine_Event $event)
    {
        echo 'beginning transaction... ';
    }

    public function preTransactionRollback(Doctrine_Event $event)
    {
        echo 'rolling back transaction... ';
    }
}
```

Query Execution Listeners

All of the below listeners are invoked in the Doctrine_Connection and Doctrine_Connection_Statement classes. And they are all passed an instance of Doctrine_Event.

Methods	Listens	Params
prePrepare()	prepare()	query
postPrepare()	prepare()	query
preExec()	exec()	query
postExec()	exec()	query, rows
preStmtExecute()	execute()	query
postStmtExecute()	execute()	query
preExecute()	execute() *	query
postExecute()	execute() *	query
preFetch()	fetch()	query, data
postFetch()	fetch()	query, data
preFetchAll()	fetchAll()	query, data
postFetchAll()	fetchAll()	query, data

> preExecute() and postExecute() only get invoked when Doctrine_Connection::execute() is being called without prepared statement parameters. Otherwise Doctrine_Connection::execute() invokes prePrepare(), postPrepare(), preStmtExecute() and postStmtExecute().

Hydration Listeners

The hydration listeners can be used for listening to resultset hydration procedures. Two methods exist for listening to the hydration procedure: preHydrate() and postHydrate().

If you set the hydration listener on connection level the code within the preHydrate() and postHydrate() blocks will be invoked by all components within a multi-component resultset. However if you add a similar listener on table level it only gets invoked when the data of that table is being hydrated.

Consider we have a class called User with the following fields: first_name, last_name and age. In the following example we create a listener that always builds a generated field called full_name based on first_name and last_name fields.

```php
// test.php

// ...
class HydrationListener extends Doctrine_Record_Listener
{
    public function preHydrate(Doctrine_Event $event)
    {
        $data = $event->data;

        $data['full_name'] = $data['first_name'] . ' ' . $data['last_name'];
        $event->data = $data;
    }
}
```

Now all we need to do is attach this listener to the `User` record and fetch some users:

```php
// test.php

// ...
$userTable = Doctrine_Core::getTable('User');
$userTable->addRecordListener(new HydrationListener());

$q = Doctrine_Query::create()
    ->from('User');

$users = $q->execute();

foreach ($users as $user) {
    echo $user->full_name;
}
```

Record Listeners

`Doctrine_Record` provides listeners very similar to `Doctrine_Connection`. You can set the listeners at global, connection and table level.

Here is a list of all available listener methods:

All of the below listeners are invoked in the `Doctrine_Record` and `Doctrine_Validator` classes. And they are all passed an instance of `Doctrine_Event`.

Methods	Listens
`preSave()`	`save()`
`postSave()`	`save()`
`preUpdate()`	`save()` when the record state is `DIRTY`
`postUpdate()`	`save()` when the record state is `DIRTY`
`preInsert()`	`save()` when the record state is `TDIRTY`
`postInsert()`	`save()` when the record state is `TDIRTY`
`preDelete()`	`delete()`
`postDelete()`	`delete()`
`preValidate()`	`validate()`
`postValidate()`	`validate()`

Just like with connection listeners there are three ways of defining a record listener: by extending `Doctrine_Record_Listener`, by implementing `Doctrine_Record_Listener_Interface` or by implementing `Doctrine_Overloadable`.

In the following we'll create a global level listener by implementing `Doctrine_Overloadable`:

```php
class Logger implements Doctrine_Overloadable
{
    public function __call($m, $a)
    {
        echo 'caught event ' . $m;

        // do some logging here...
    }
}
```

Attaching the listener to manager is easy:

```php
$manager->addRecordListener(new Logger());
```

Note that by adding a manager level listener it affects on all connections and all tables / records within these connections. In the following we create a connection level listener:

```
class Debugger extends Doctrine_Record_Listener
{
    public function preInsert(Doctrine_Event $event)
    {
        echo 'inserting a record ...';
    }

    public function preUpdate(Doctrine_Event $event)
    {
        echo 'updating a record...';
    }
}
```

Attaching the listener to a connection is as easy as:

```
$conn->addRecordListener(new Debugger());
```

Many times you want the listeners to be table specific so that they only apply on the actions on that given table.

Here is an example:

```
class Debugger extends Doctrine_Record_Listener
{
    public function postDelete(Doctrine_Event $event)
    {
        echo 'deleted ' . $event->getInvoker()->id;
    }
}
```

Attaching this listener to given table can be done as follows:

```
class MyRecord extends Doctrine_Record
{
    // ...

    public function setUp()
    {
        $this->addListener(new Debugger());
    }
}
```

Record Hooks

All of the below listeners are invoked in the `Doctrine_Record` and `Doctrine_Validator` classes. And they are all passed an instance of `Doctrine_Event`.

Methods	Listens
`preSave()`	`save()`
`postSave()`	`save()`
`preUpdate()`	`save()` when the record state is `DIRTY`
`postUpdate()`	`save()` when the record state is `DIRTY`
`preInsert()`	`save()` when the record state is `TDIRTY`
`postInsert()`	`save()` when the record state is `TDIRTY`
`preDelete()`	`delete()`
`postDelete()`	`delete()`
`preValidate()`	`validate()`
`postValidate()`	`validate()`

Here is a simple example where we make use of the `preInsert()` and `preUpdate()` methods:

```php
class BlogPost extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 200);
        $this->hasColumn('content', 'string');
        $this->hasColumn('created', 'date');
        $this->hasColumn('updated', 'date');
    }

    public function preInsert($event)
    {
        $this->created = date('Y-m-d', time());
    }

    public function preUpdate($event)
    {
        $this->updated = date('Y-m-d', time());
    }
}
```

DQL Hooks

Doctrine allows you to attach record listeners globally, on each connection, or on specific record instances. `Doctrine_Query` implements `preDql*()` hooks which are checked for on any attached record listeners and checked for on the model instance itself whenever a query is executed. The query will check all models involved in the `from` part of the query for any hooks which can alter the query that invoked the hook.

Here is a list of the hooks you can use with DQL:

Methods	Listens
`preDqlSelect()`	`from()`
`preDqlUpdate()`	`update()`
`preDqlDelete()`	`delete()`

Below is an example record listener attached directly to the model which will implement the `SoftDelete` functionality for the `User` model.

> The SoftDelete functionality is included in Doctrine as a behavior. This code is used to demonstrate how to use the select, delete, and update DQL listeners to modify executed queries. You can use the SoftDelete behavior by specifying `$this->actAs('SoftDelete')` in your `Doctrine_Record::setUp()` definition.

```
class UserListener extends Doctrine_EventListener
{
    /**
     * Skip the normal delete options so we can override it with our own
     *
     * @param Doctrine_Event $event
     * @return void
     */
    public function preDelete(Doctrine_Event $event)
    {
        $event->skipOperation();
    }

    /**
     * Implement postDelete() hook and set the deleted flag to true
     *
     * @param Doctrine_Event $event
     * @return void
     */
    public function postDelete(Doctrine_Event $event)
    {
        $name = $this->_options['name'];
        $event->getInvoker()->$name = true;
        $event->getInvoker()->save();
    }

    /**
     * Implement preDqlDelete() hook and modify a dql delete query so it updates the deleted flag
     * instead of deleting the record
     *
     * @param Doctrine_Event $event
     * @return void
     */
    public function preDqlDelete(Doctrine_Event $event)
    {
        $params = $event->getParams();
        $field = $params['alias'] . '.deleted';
        $q = $event->getQuery();
        if ( ! $q->contains($field)) {
            $q->from('')->update($params['component'] . ' ' . $params['alias']);
            $q->set($field, '?', array(false));
            $q->addWhere($field . ' = ?', array(true));
        }
    }

    /**
     * Implement preDqlDelete() hook and add the deleted flag to all queries for which this model
     * is being used in.
     *
     * @param Doctrine_Event $event
     * @return void
     */
    public function preDqlSelect(Doctrine_Event $event)
    {
        $params = $event->getParams();
        $field = $params['alias'] . '.deleted';
        $q = $event->getQuery();
        if ( ! $q->contains($field)) {
            $q->addWhere($field . ' = ?', array(false));
        }
    }
}
```

All of the above methods in the listener could optionally be placed in the user class below. Doctrine will check there for the hooks as well:

```
class User extends Doctrine_Record
{
    // ...

    public function preDqlSelect()
    {
        // ...
    }

    public function preDqlUpdate()
    {
        // ...
    }

    public function preDqlDelete()
    {
        // ...
    }
}
```

In order for these dql callbacks to be checked, you must explicitly turn them on. Because this adds a small amount of overhead for each query, we have it off by default. We already enabled this attribute in an earlier chapter.

Here it is again to refresh your memory:

```
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_USE_DQL_CALLBACKS, true);
```

Now when you interact with the User model it will take in to account the deleted flag:

Delete user through record instance:

```
$user = new User();
$user->username = 'jwage';
$user->password = 'changeme';
$user->save();
$user->delete();
```

> The above call to `$user->delete()` does not actually delete the record instead it sets the deleted flag to true.

```
$q = Doctrine_Query::create()
    ->from('User u');

echo $q->getSqlQuery();
```

```
SELECT
u.id AS u__id,
u.username AS u__username,
u.password AS u__password,
u.deleted AS u__deleted
FROM user u
WHERE u.deleted = ?
```

> Notice how the `"u.deleted = ?"` was automatically added to the where condition with a parameter value of **true**.

Chaining Listeners

Doctrine allows chaining of different event listeners. This means that more than one listener can be attached for listening the same events. The following example attaches two listeners for given connection:

In this example `Debugger` and `Logger` both inherit `Doctrine_EventListener`:

```
$conn->addListener(new Debugger());
$conn->addListener(new Logger());
```

The Event object

Getting the Invoker

You can get the object that invoked the event by calling `getInvoker()`:

```
class MyListener extends Doctrine_EventListener
{
    public function preExec(Doctrine_Event $event)
    {
        $event->getInvoker(); // Doctrine_Connection
    }
}
```

Event Codes

`Doctrine_Event` uses constants as event codes. Below is the list of all available event constants:

- `Doctrine_Event::CONN_QUERY`
- `Doctrine_Event::CONN_EXEC`
- `Doctrine_Event::CONN_PREPARE`
- `Doctrine_Event::CONN_CONNECT`
- `Doctrine_Event::STMT_EXECUTE`
- `Doctrine_Event::STMT_FETCH`

- `Doctrine_Event::STMT_FETCHALL`
- `Doctrine_Event::TX_BEGIN`
- `Doctrine_Event::TX_COMMIT`
- `Doctrine_Event::TX_ROLLBACK`
- `Doctrine_Event::SAVEPOINT_CREATE`
- `Doctrine_Event::SAVEPOINT_ROLLBACK`
- `Doctrine_Event::SAVEPOINT_COMMIT`
- `Doctrine_Event::RECORD_DELETE`
- `Doctrine_Event::RECORD_SAVE`
- `Doctrine_Event::RECORD_UPDATE`
- `Doctrine_Event::RECORD_INSERT`
- `Doctrine_Event::RECORD_SERIALIZE`
- `Doctrine_Event::RECORD_UNSERIALIZE`
- `Doctrine_Event::RECORD_DQL_SELECT`
- `Doctrine_Event::RECORD_DQL_DELETE`
- `Doctrine_Event::RECORD_DQL_UPDATE`

Here are some examples of hooks being used and the code that is returned:

```
class MyListener extends Doctrine_EventListener
{
    public function preExec(Doctrine_Event $event)
    {
        $event->getCode(); // Doctrine_Event::CONN_EXEC
    }
}

class MyRecord extends Doctrine_Record
{
    public function preUpdate(Doctrine_Event $event)
    {
        $event->getCode(); // Doctrine_Event::RECORD_UPDATE
    }
}
```

Getting the Invoker

The method `getInvoker()` returns the object that invoked the given event. For example for event `Doctrine_Event::CONN_QUERY` the invoker is a `Doctrine_Connection` object.

Here is an example of using the record hook named `preUpdate()` that is invoked when a `Doctrine_Record` instance is saved and an update is issued to the database:

```
class MyRecord extends Doctrine_Record
{
    public function preUpdate(Doctrine_Event $event)
    {
        $event->getInvoker(); // Object(MyRecord)
    }
}
```

Skip Next Operation

`Doctrine_Event` provides many methods for altering the execution of the listened method as well as for altering the behavior of the listener chain.

For some reason you may want to skip the execution of the listened method. It can be done as follows (note that `preExec()` could be any listener method):

```
class MyListener extends Doctrine_EventListener
{
    public function preExec(Doctrine_Event $event)
    {
        // some business logic, then:

        $event->skipOperation();
    }
}
```

Skip Next Listener

When using a chain of listeners you might want to skip the execution of the next listener. It can be achieved as follows:

```
class MyListener extends Doctrine_EventListener
{
    public function preExec(Doctrine_Event $event)
    {
        // some business logic, then:

        $event->skipNextListener();
    }
}
```

Conclusion

Event listeners are a great feature in Doctrine and combined with Behaviors they can provide some very complex functionality with a minimal amount of code.

Now we are ready to move on to discuss the best feature in Doctrine for improving performance, Caching.

Caching

Introduction

`Doctrine_Cache` offers an intuitive and easy-to-use query caching solution. It provides the following things:

- Multiple cache backends to choose from (including Memcached, APC and Sqlite)
- Advanced options for fine-tuning. `Doctrine_Cache` has many options for fine-tuning performance.

All the cache drivers are instantiated like the following:

```
// bootstrap.php

// ...
$options = array();
$cacheDriver = new Doctrine_Cache_Memcache($options);
```

> Each driver has its own possible values for the `$options` array.

Drivers

Memcache

Memcache driver stores cache records into a memcached server. Memcached is a high-performance, distributed memory object caching system. In order to use this backend, you need a memcached daemon and the memcache PECL extension.

You can instantiate the Memcache cache driver with the following code:

```
// bootstrap.php

// ...
$servers = array(
    'host' => 'localhost',
    'port' => 11211,
    'persistent' => true
);

$cacheDriver = new Doctrine_Cache_Memcache(array(
        'servers' => $servers,
        'compression' => false
    )
);
```

> Memcache allows multiple servers.

Available options for Memcache driver:

Option	Data Type	Default Value	Description
`servers`	`array`	`array(array('host' => 'localhost','port' => 11211, 'persistent' => true))`	An array of memcached servers ; each memcached server is described by an associative array : `'host' => (string)` : the name of the memcached server, `'port' => (int)` : the port of the memcached server, `'persistent' => (bool)` : use or not persistent connections to this memcached server
`compression`	`boolean`	`false`	`true` if you want to use on-the-fly compression

APC

The Alternative PHP Cache (APC) is a free and open opcode cache for PHP. It was conceived to provide a free, open, and robust framework for caching and optimizing PHP intermediate code. The APC cache driver of Doctrine stores cache records in shared memory.

You can instantiate the APC cache driver with the following code:

```php
// bootstrap.php

// ...
$cacheDriver = new Doctrine_Cache_Apc();
```

Db

Db caching backend stores cache records into given database. Usually some fast flat-file based database is used (such as sqlite).

You can instantiate the database cache driver with the following code:

```php
// bootstrap.php

// ...
$cacheConn = Doctrine_Manager::connection(new PDO('sqlite::memory:'));
$cacheDriver = new Doctrine_Cache_Db(array('connection' => $cacheConn));
```

Query Cache & Result Cache

Introduction

Doctrine provides means for caching the results of the DQL parsing process, as well as the end results of DQL queries (the data). These two caching mechanisms can greatly increase performance. Consider the standard workflow of DQL query execution:

- Init new DQL query
- Parse DQL query
- Build database specific SQL query
- Execute the SQL query
- Build the result set
- Return the result set

Now these phases can be very time consuming, especially phase 4 which sends the query to your database server. When Doctrine query cache is being used only the following phases occur:

- Init new DQL query
- Execute the SQL query (grabbed from the cache)
- Build the result set
- Return the result set

If a DQL query has a valid cache entry the cached SQL query is used, otherwise the phases 2-3 are executed normally and the result of these steps is then stored in the cache. The query cache has no disadvantages, since you always get a fresh query result.

> You should always use query cache in a production environment. That said, you can easily use it during development, too. Whenever you change a DQL query and execute it the first time Doctrine sees that it has been modified and will therefore create a new cache entry, so you don't even need to invalidate the cache.

It's worth noting that the effectiveness of the query cache greatly relies on the usage of prepared statements (which are used by Doctrine by default anyway). You should not directly embed dynamic query parts and always use placeholders instead.

When using a result cache things get even better. Then your query process looks as follows (assuming a valid cache entry is found):

- Init new DQL query
- Return the result set

As you can see, the result cache implies the query cache shown previously. You should always consider using a result cache if the data returned by the query does not need to be up-to-date at any time.

Query Cache

Using the Query Cache

You can set a connection or manager level query cache driver by using the `Doctrine_Core::ATTR_QUERY_CACHE` attribute. Setting a connection level cache driver means that all queries executed with this connection use the specified cache driver whereas setting a manager level cache driver means that all connections (unless overridden at connection level) will use the given cache driver.

Setting a manager level query cache driver:

```php
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_QUERY_CACHE, $cacheDriver);
```

> The value of `$cacheDriver` above could be any of the drivers we instantiated in the previous section of this chapter.

Setting a connection level cache driver:

```
// bootstrap.php

// ...
$conn->setAttribute(Doctrine_Core::ATTR_QUERY_CACHE, $cacheDriver);
```

Fine Tuning

In the previous chapter we used global caching attributes. These attributes can be overriden at the query level. You can override the cache driver by calling `useQueryCache()` and pass it an instance of a valid Doctrine cache driver. This rarely makes sense for the query cache but is possible:

```
$q = Doctrine_Query::create()
    ->useQueryCache(new Doctrine_Cache_Apc());
```

Result Cache

Using the Result Cache

You can set a connection or manager level result cache driver by using `Doctrine_Core::ATTR_RESULT_CACHE`. Setting a connection level cache driver means that all queries executed with this connection use the specified cache driver whereas setting a manager level cache driver means that all connections (unless overridden at connection level) will use the given cache driver.

Setting a manager level cache driver:

```
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_RESULT_CACHE, $cacheDriver);
```

Setting a connection level cache driver:

```
// bootstrap.php

// ...
$conn->setAttribute(Doctrine_Core::ATTR_RESULT_CACHE, $cacheDriver);
```

Usually the cache entries are valid for only some time. You can set global value for how long the cache entries should be considered valid by using `Doctrine_Core::ATTR_RESULT_CACHE_LIFESPAN`.

Set the lifespan as one hour (60 seconds * 60 minutes = 1 hour = 3600 secs):

```
// bootstrap.php

// ...
$manager->setAttribute(Doctrine_Core::ATTR_RESULT_CACHE_LIFESPAN, 3600);
```

Now as we have set a cache driver for use we can make a DQL query use it by calling the `useResultCache()` method:

Fetch blog post titles and the number of comments:

```
$q = Doctrine_Query::create()
    ->select('b.title, COUNT(c.id) count')
    ->from('BlogPost b')
    ->leftJoin('b.Comments c')
    ->limit(10)
    ->useResultCache(true);

$blogPosts = $q->execute();
```

Fine Tuning

In the previous chapter we used global caching attributes. These attributes can be overriden at the query level. You can override the cache driver by calling `useCache()` and pass it an instance of a valid Doctrine cache driver.

```
$q = Doctrine_Query::create()
    ->useResultCache(new Doctrine_Cache_Apc());
```

Also you can override the lifespan attribute by calling `setResultCacheLifeSpan()`:

```
$q = Doctrine_Query::create()
    ->setResultCacheLifeSpan(60 * 30);
```

Conclusion

Using the caching feature of Doctrine is highly recommended in both development and production environments. There are no adverse affects to using it and it will only help the performance of your application.

The caching feature is the second to last feature we will discuss in this book before wrapping things up by discussing things like the technologies used in Doctrine, coding standards and unit testing. Lets move on to discuss the last feature of Doctrine, Migrations.

Migrations

The Doctrine migration package allows you to easily update your production databases through a nice programmatic interface. The changes are done in a way so that your database is versioned and you can walk backwards and forwards through the database versions.

Performing Migrations

Before we learn how to create the migration classes lets take a look at how we can run migrations so that we can implement them in our Doctrine test environment in the next section.

First lets create a new instance of `Doctrine_Migration` and pass it the path to our migration classes:

```
$migration = new Doctrine_Migration('/path/to/migration_classes', $conn);
```

> Notice the second argument to the `Doctrine_Migration` constructor. You can pass an optional `Doctrine_Connection` instance. If you do not pass the connection for the migration class to use, it will simply grab the current connection.

Now we can migrate to the latest version by calling the `migrate()` method:

```
$migration->migrate();
```

If you want to migrate to a specific version you can pass an argument to `migrate()`. For example we can migrate to version 3 from 0:

```
$migration->migrate(3);
```

Now you can migrate back to version 0 from 3:

```
$migration->migrate(0);
```

If you want to get the current version of the database you can use the `getCurrentVersion()` method:

```
echo $migration->getCurrentVersion();
```

> Omitting the version number argument for the `migrate()` method means that internally Doctrine will try and migrate to the latest class version number that it could find in the directory you passed.

> **Transactions in Migrations**
> Internally Doctrine does not wrap migration versions in transactions. It is up to you as the developer to handle exceptions and transactions in your migration classes. Remember though, that very few databases support transactional DDL. So on most databases, even if you wrap the migrations in a transaction, any DDL statements like create, alter, drop, rename, etc., will take effect anyway.

Implement

Now that we know how to perform migrations lets implement a little script in our Doctrine test environment named `migrate.php`.

First we need to create a place for our migration classes to be stored so lets create a directory named `migrations`:

```
$ mkdir migrations
```

Now create the `migrate.php` script in your favorite editor and place the following code inside:

```
// migrate.php

require_once('bootstrap.php');

$migration = new Doctrine_Migration('migrations');
$migration->migrate();
```

Writing Migration Classes

Migration classes consist of a simple class that extends from `Doctrine_Migration`. You can define a `up()` and `down()` method that is meant for doing and undoing changes to a database for that migration version.

> The name of the class can be whatever you want, but the name of the file which contains the class must have a prefix containing a number that is used for loading the migrations in the correct order.

Below are a few examples of some migration classes you can use to build your database starting from version 1.

For the first version lets create a new table named `migration_test`:

```
// migrations/1_add_table.php

class AddTable extends Doctrine_Migration_Base
{
    public function up()
    {
        $this->createTable('migration_test', array('field1' => array('type' => 'string')));
    }

    public function down()
    {
        $this->dropTable('migration_test');
    }
}
```

Now lets create a second version where we add a new column to the table we added in the previous version:

```
// migrations/2_add_column.php

class AddColumn extends Doctrine_Migration_Base
{
    public function up()
    {
        $this->addColumn('migration_test', 'field2', 'string');
    }

    public function down()
    {
        $this->removeColumn('migration_test', 'field2');
    }
}
```

Finally, lets change the type of the `field1` column in the table we created previously:

```
// migrations/3_change_column.php

class ChangeColumn extends Doctrine_Migration_Base
{
    public function up()
    {
        $this->changeColumn('migration_test', 'field2', 'integer');
    }

    public function down()
    {
        $this->changeColumn('migration_test', 'field2', 'string');
    }
}
```

Now that we have created the three migration classes above we can run our `migrate.php` script we implemented earlier:

```
$ php migrate.php
```

If you look in the database you will see that we have the table named `migrate_test` created and the version number in the `migration_version` is set to three.

If you want to migrate back to where we started you can pass a version number to the `migrate()` method in the `migrate.php` script:

```
// migrate.php

// ...
$migration = new Doctrine_Migration('migrations');
$migration->migrate(0);
```

Now run the `migrate.php` script:

```
$ php migrate.php
```

If you look in the database now, everything we did in the `up()` methods has been reversed by the contents of the `down()` method.

Available Operations

Here is a list of the available methods you can use to alter your database in your migration classes.

Create Table

```
// ...
    public function up()
    {
        $columns = array(
            'id' => array(
                'type' => 'integer',
                'unsigned' => 1
                'notnull' => 1
                'default' => 0
            ),
            'name' => array(
                'type' => 'string',
                'length' => 12
            ),
            'password' => array(
                'type' => 'string',
                'length' => 12
            )
        );

        $options = array(
            'type'     => 'INNODB',
            'charset'  => 'utf8'
        );

        $this->createTable('table_name', $columns, $options);
    }
// ...
```

> You might notice already that the data structures used to manipulate the your schema are the same as the data structures used with the database abstraction layer. This is because internally the migration package uses the database abstraction layer to perform the operations specified in the migration classes.

Drop Table

```
// ...
    public function down()
    {
        $this->dropTable('table_name');
    }
// ...
```

Rename Table

```
// ...
    public function up()
    {
        $this->renameTable('old_table_name', 'new_table_name');
    }
// ...
```

Create Constraint

```
// ...
    public function up()
    {
        $definition = array(
            'fields' => array(
                'username' => array()
            ),
            'unique' => true
        );

        $this->createConstraint('table_name', 'constraint_name', $definition);
    }
// ...
```

Drop Constraint

Now the opposite `down()` would look like the following:

```
// ...
    public function down()
    {
        $this->dropConstraint('table_name', 'constraint_name');
    }
// ...
```

Create Foreign Key

```
// ...
    public function up()
    {
        $definition = array(
            'local'         => 'email_id',
            'foreign'       => 'id',
            'foreignTable'  => 'email',
            'onDelete'      => 'CASCADE'
        );

        $this->createForeignKey('table_name', 'email_foreign_key', $definition);
    }
// ...
```

The valid options for the `$definition` are:

Name	Description
name	Optional constraint name
local	The local field(s)
foreign	The foreign reference field(s)
foreignTable	The name of the foreign table
onDelete	Referential delete action
onUpdate	Referential update action
deferred	Deferred constraint checking

Drop Foreign Key

```
// ...
    public function down()
    {
        $this->dropForeignKey('table_name', 'email_foreign_key');
    }
// ...
```

Add Column

```
// ...
    public function up()
    {
        $this->addColumn('table_name', 'column_name', 'string', $options);
    }
// ...
```

Rename Column

> Some DBMS like sqlite do not implement the rename column operation. An exception is thrown if you try and rename a column when using a sqlite connection.

```
// ...
    public function up()
    {
        $this->renameColumn('table_name', 'old_column_name', 'new_column_name');
    }
// ...
```

Change Column

Change any aspect of an existing column:

```
// ...
    public function up()
    {
        $options = array('length' => 1);
        $this->changeColumn('table_name', 'column_name', 'tinyint', $options);
    }
// ...
```

Remove Column

```
// ...
    public function up()
    {
        $this->removeColumn('table_name', 'column_name');
    }
// ...
```

Irreversible Migration

> Sometimes you may perform some operations in the `up()` method that cannot be reversed. For example if you remove a column from a table. In this case you need to throw a new `Doctrine_Migration_IrreversibleMigrationException` exception.

```
// ...
    public function down()
    {
        throw new Doctrine_Migration_IrreversibleMigrationException(
            'The remove column operation cannot be undone!'
        );
    }
// ...
```

Add Index

```
// ...
    public function up()
    {
        $options = array('fields' => array(
                            'username' => array(
                                'sorting' => 'ascending'
                            ),
                            'last_login' => array()));

        $this->addIndex('table_name', 'index_name', $options)
    }
// ...
```

Remove Index

```
// ...
    public function down()
    {
        $this->removeIndex('table_name', 'index_name');
    }
// ...
```

Pre and Post Hooks

Sometimes you may need to alter the data in the database with your models. Since you may create a table or make a change, you have to do the data altering after the `up()` or `down()` method is processed. We have hooks in place for this named `preUp()`, `postUp()`, `preDown()`, and `postDown()`. Define these methods and they will be triggered:

```
// migrations/1_add_table.php

class AddTable extends Doctrine_Migration_Base
{
    public function up()
    {
        $this->createTable('migration_test', array('field1' => array('type' => 'string')));
    }

    public function postUp()
    {
        $migrationTest = new MigrationTest();
        $migrationTest->field1 = 'Initial record created by migrations';
        $migrationTest->save();
    }

    public function down()
    {
        $this->dropTable('migration_test');
    }
}
```

> The above example assumes you have created and made available the `MigrationTest` model. Once the `up()` method is executed the `migration_test` table is created so the `MigrationTest` model can be used. We have provided the definition of this model below.

```
// models/MigrationTest.php

class MigrationTest extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('field1', 'string');
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```
---
# schema.yml

# ...
MigrationTest:
  columns:
    field1: string
```

Up/Down Automation

In Doctrine migrations it is possible most of the time to automate the opposite of a migration method. For example if you create a new column in the up of a migration, we should be able to easily automate the down since all we need to do is remove the column that was created. This is possible by using the `migrate()` function for both the `up` and `down`.

The `migrate()` method accepts an argument of $direction and it will either have a value of `up` or `down`. This value is passed to the first argument of functions like `column`, `table`, etc.

Here is an example where we automate the adding and removing of a column

```
class MigrationTest extends Doctrine_Migration_Base
{
    public function migrate($direction)
    {
        $this->column($direction, 'table_name', 'column_name', 'string', '255');
    }
}
```

Now when we run up with the above migration, the column will be added and when we run down the column will be removed.

Here is a list of the following migration methods that can be automated:

Automate Method Name	Automates
table()	createTable()/dropTable()
constraint()	createConstraint()/dropConstraint()
foreignKey()	createForeignKey()/dropForeignKey()
column()	addColumn()/removeColumn()
index()	addInex()/removeIndex()

Generating Migrations

Doctrine offers the ability to generate migration classes a few different ways. You can generate a set of migrations to re-create an existing database, or generate migration classes to create a database for an existing set of models. You can even generate migrations from differences between two sets of schema information.

From Database

To generate a set of migrations from the existing database connections it is simple, just use `Doctrine_Core::generateMigrationsFromDb()`.

```
Doctrine_Core::generateMigrationsFromDb('/path/to/migration/classes');
```

From Existing Models

To generate a set of migrations from an existing set of models it is just as simple as from a database, just use `Doctrine_Core::generateMigrationsFromModels()`.

```
Doctrine_Core::generateMigrationsFromModels('/path/to/migration/classes', '/path/to/models');
```

Diff Tool

Sometimes you may want to alter your models and be able to automate the migration process for your changes. In the past you would have to write the migration classes manually for your changes. Now with the diff tool you can make your changes then generate the migration classes for the changes.

The diff tool is simple to use. It accepts a "from" and a "to" and they can be one of the following:

- Path to yaml schema files
- Name of an existing database connection
- Path to an existing set of models

A simple example would be to create two YAML schema files, one named `schema1.yml` and another named `schema2.yml`.

The `schema1.yml` contains a simple `User` model:

```
---
# schema1.yml

User:
  columns:
    username: string(255)
    password: string(255)
```

Now imagine we modify the above schema and want to add a `email_address` column:

```
---
# schema1.yml

User:
  columns:
    username: string(255)
    password: string(255)
    email_address: string(255)
```

Now we can easily generate a migration class which will add the new column to our database:

```
Doctrine_Core::generateMigrationsFromDiff('/path/to/migration/classes', '/path/to/schema1.yml', '/path/to/schema2.yml')
```

This will produce a file at the path `/path/to/migration/classes/1236199329_version1.php`

```
class Version1 extends Doctrine_Migration_Base
{
    public function up()
    {
        $this->addColumn('user', 'email_address', 'string', '255', array ());
    }

    public function down()
    {
        $this->removeColumn('user', 'email_address');
    }
}
```

Now you can easily migrate your database and add the new column!

Conclusion

Using migrations is highly recommended for altering your production database schemas as it is a safe and easy way to make changes to your schema.

Migrations are the last feature of Doctrine that we will discuss in this book. The final chapters will discuss some other topics that will help you be a better Doctrine developers on a day-to-day basis. First lets discuss some of the other Utilities that Doctrine provides.

Extensions

The Doctrine extensions are a way to create reusable Doctrine extensions that can be dropped into any project and enabled. An extension is nothing more than some code that follows the Doctrine standards for code naming, autoloading, etc.

In order to use the extensions you must first configure Doctrine to know the path to where your extensions live:

```
Doctrine_Core::setExtensionsPath('/path/to/extensions');
```

Lets checkout an existing extension from SVN to have a look at it. We'll have a look at the `Sortable` extension which bundles a behavior for your models which give you up and down sorting capabilities.

```
$ svn co http://svn.doctrine-project.org/extensions/Sortable/branches/1.2-1.0/ /path/to/extensions/Sortable
```

If you have a look at `/path/to/extensions/Sortable` you will see a directory structure that looks like the following:

```
Sortable/
  lib/
    Doctrine/
      Template/
        Listener/
          Sortable.php
        Sortable.php
  tests/
    run.php
    Template/
      SortableTestCase.php
```

To test that the extension will run on your machine you can run the test suite for the extension. All you need to do is set the `DOCTRINE_DIR` environment variable.

```
$ export DOCTRINE_DIR=/path/to/doctrine
```

> The above path to Doctrine must be the path to the main folder, not just the lib folder. In order to run the tests it must have access to the `tests` directory included with Doctrine.

It is possible now to run the tests for the `Sortable` extension:

```
$ cd /path/to/extensions/Sortable/tests
$ php run.php
```

You should see the tests output the following showing the tests were successful:

```
Doctrine Unit Tests
===================
Doctrine_Template_Sortable_TestCase.........................................passed

Tested: 1 test cases.
Successes: 26 passes.
Failures: 0 fails.
Number of new Failures: 0
Number of fixed Failures: 0

Tests ran in 1 seconds and used 13024.9414062 KB of memory
```

Now if you want to use the extension in your project you will need register the extension with Doctrine and setup the extension autoloading mechanism.

First lets setup the extension autoloading.

```
// bootstrap.php

// ...
spl_autoload_register(array('Doctrine', 'extensionsAutoload'));
```

Now you can register the extension and the classes inside that extension will be autoloaded.

```
$manager->registerExtension('Sortable');
```

If you need to register an extension from a different location you can specify the full path to the extension directory as the second argument to the `registerExtension()` method.

Utilities

Pagination

Introduction

In real world applications, display content from database tables is a commom task. Also, imagine that this content is a search result containing thousands of items. Undoubtely, it will be a huge listing, memory expensive and hard for users to find the right item. That is where some organization of this content display is needed and pagination comes in rescue.

Doctrine implements a highly flexible pager package, allowing you to not only split listing in pages, but also enabling you to control the layout of page links. In this chapter, we'll learn how to create pager objects, control pager styles and at the end, overview the pager layout object – a powerful page links displayer of Doctrine.

Working with Pager

Paginating queries is as simple as effectively do the queries itself. `Doctrine_Pager` is the responsible to process queries and paginate them. Check out this small piece of code:

```
// Defining initial variables
$currentPage = 1;
$resultsPerPage = 50;

// Creating pager object
$pager = new Doctrine_Pager(
      Doctrine_Query::create()
            ->from( 'User u' )
            ->leftJoin( 'u.Group g' )
            ->orderby( 'u.username ASC' ),
      $currentPage, // Current page of request
      $resultsPerPage // (Optional) Number of results per page. Default is 25
);
```

Until this place, the source you have is the same as the old `Doctrine_Query` object. The only difference is that now you have 2 new arguments. Your old query object plus these 2 arguments are now encapsulated by the `Doctrine_Pager` object. At this stage, `Doctrine_Pager` defines the basic data needed to control pagination. If you want to know that actual status of the pager, all you have to do is to check if it's already executed:

```
$pager->getExecuted();
```

If you try to access any of the methods provided by `Doctrine_Pager` now, you'll experience `Doctrine_Pager_Exception` thrown, reporting you that Pager was not yet executed. When executed, `Doctrine_Pager` offer you powerful methods to retrieve information. The API usage is listed at the end of this topic.

To run the query, the process is similar to the current existent `Doctrine_Query` execute call. It even allow arguments the way you usually do it. Here is the PHP complete syntax, including the syntax of optional parameters:

```
$items = $pager->execute([$args = array() [, $fetchType = null]]);

foreach ($items as $item) {
    // ...
}
```

There are some special cases where the return records query differ of the counter query. To allow this situation, `Doctrine_Pager` has some methods that enable you to count and then to execute. The first thing you have to do is to define the count query:

```
$pager->setCountQuery($query [, $params = null]);

// ...

$rs = $pager->execute();
```

The first param of `setCountQuery` can be either a valid `Doctrine_Query` object or a DQL string. The second argument you can define the optional parameters that may be sent in the counter query. If you do not define the params now, you're still able to define it later by calling the `setCountQueryParams`:

```
$pager->setCountQueryParams([$params = array() [, $append = false]]);
```

This method accepts 2 parameters. The first one is the params to be sent in count query and the second parameter is if the `$params` should be appended to the list or if it should override the list of count query parameters. The default behavior is to override the list. One last thing to mention about count query is, if you do not define any parameter for count query, it will still send the parameters you define in `$pager->execute()` call.

Count query is always enabled to be accessed. If you do not define it and call `$pager->getCountQuery()`, it will return the "fetcher" query to you.

If you need access the other functionalities that `Doctrine_Pager` provides, you can access them through the API:

```
// Returns the check if Pager was already executed
$pager->getExecuted();

// Return the total number of itens found on query search
$pager->getNumResults();

// Return the first page (always 1)
$pager->getFirstPage();

// Return the total number of pages
$pager->getLastPage();

// Return the current page
$pager->getPage();

// Defines a new current page (need to call execute again to adjust offsets and values)
$pager->setPage($page);

// Return the next page
$pager->getNextPage();

// Return the previous page
$pager->getPreviousPage();

// Return the first indice of current page
$pager->getFirstIndice();

// Return the last indice of current page
$pager->getLastIndice();

// Return true if it's necessary to paginate or false if not
$pager->haveToPaginate();

// Return the maximum number of records per page
$pager->getMaxPerPage();

// Defined a new maximum number of records per page (need to call execute again to adjust offset and values)
$pager->setMaxPerPage($maxPerPage);

// Returns the number of itens in current page
$pager->getResultsInPage();

// Returns the Doctrine_Query object that is used to make the count results to pager
$pager->getCountQuery();

// Defines the counter query to be used by pager
$pager->setCountQuery($query, $params = null);

// Returns the params to be used by counter Doctrine_Query (return $defaultParams if no param is defined)
$pager->getCountQueryParams($defaultParams = array());

// Defines the params to be used by counter Doctrine_Query
$pager->setCountQueryParams($params = array(), $append = false);

// Return the Doctrine_Query object
$pager->getQuery();

// Return an associated Doctrine_Pager_Range_* instance
$pager->getRange($rangeStyle, $options = array());
```

Controlling Range Styles

There are some cases where simple paginations are not enough. One example situation is when you want to write page links listings. To enable a more powerful control over pager, there is a small subset of pager package that allows you to create ranges.

Currently, Doctrine implements two types (or styles) of ranges: Sliding (`Doctrine_Pager_Range_Sliding`) and Jumping (`Doctrine_Pager_Range_Jumping`).

Sliding

Sliding page range style, the page range moves smoothly with the current page. The current page is always in the middle, except in the first and last pages of the range. Check out how does it work with a chunk length of 5 items:

```
Listing 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Page 1: o-------|
Page 2: |-o-----|
Page 3: |---o---|
Page 4:   |---o---|
Page 5:     |---o---|
Page 6:       |---o---|
Page 7:         |---o---|
Page 8:           |---o---|
```

Jumping

In Jumping page range style, the range of page links is always one of a fixed set of "frames": 1–5, 6–10, 11–15, and so on.

```
Listing 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Page 1: o-------|
Page 2: |-o-----|
Page 3: |---o---|
Page 4: |-----o-|
Page 5: |-------o
Page 6:           o---------|
Page 7:           |-o-------|
Page 8:           |---o-----|
```

Now that we know how the different of styles of pager range works, it's time to learn how to use them:

```php
$pagerRange = new Doctrine_Pager_Range_Sliding(
    array(
          'chunk' => 5 // Chunk length
    ),
    $pager // Doctrine_Pager object we learned how to create in previous topic
);
```

Alternatively, you can use:

```php
$pagerRange = $pager->getRange(
      'Sliding',
      array(
            'chunk' => 5
      )
);
```

What is the advantage to use this object, instead of the `Doctrine_Pager`? Just one; it allows you to retrieve ranges around the current page.

Look at the example:

```php
// Retrieves the range around the current page
// In our example, we are using sliding style and we are at page 1
$pages = $pager_range->rangeAroundPage();

// Outputs: [1][2][3][4][5]
echo '['. implode('][', $pages) .']';
```

If you build your `Doctrine_Pager` inside the range object, the API gives you enough power to retrieve information related to `Doctrine_Pager_Range` subclass instance:

```php
// Return the Pager associated to this Pager_Range
$pager_range->getPager();

// Defines a new Doctrine_Pager (automatically call _initialize protected method)
$pager_range->setPager($pager);

// Return the options assigned to the current Pager_Range
$pager_range->getOptions();

// Returns the custom Doctrine_Pager_Range implementation offset option
$pager_range->getOption($option);

// Check if a given page is in the range
$pager_range->isInRange($page);

// Return the range around the current page (obtained from Doctrine_Pager
// associated to the $pager_range instance)
$pager_range->rangeAroundPage();
```

Advanced layouts with pager

Until now, we learned how to create paginations and how to retrieve ranges around the current page. To abstract the business logic involving the page links generation, there is a powerful component called `Doctrine_Pager_Layout`. The main idea of this component is to abstract php logic and only leave HTML to be defined by Doctrine developer.

`Doctrine_Pager_Layout` accepts 3 obrigatory arguments: a `Doctrine_Pager` instance, a `Doctrine_Pager_Range` subclass instance and a string which is the URL to be assigned as {%url} mask in templates. As you may see, there are two types of "variables" in `Doctrine_Pager_Layout`:

Mask

A piece of string that is defined inside template as replacements. They are defined as **{%mask_name}** and are replaced by what you define in options or what is defined internally by `Doctrine_Pager_Layout` component. Currently, these are the internal masks available:

- **{%page}** Holds the page number, exactly as page_number, but can be overwritable by `addMaskReplacement()` to behavior like another mask or value
- **{%page_number}** Stores the current page number, but cannot be overwritable
- **{%url}** Available only in `setTemplate()` and `setSelectedTemplate()` methods. Holds the processed URL, which was defined in constructor

Template

As the name explains itself, it is the skeleton of HTML or any other resource that is applied to each page returned by `Doctrine_Pager_Range::rangeAroundPage()` subclasses. There are 3 distinct templates that can be defined:

- `setTemplate()` Defines the template that can be used in all pages returned by `Doctrine_Pager_Range::rangeAroundPage()` subclass call
- `setSelectedTemplate()` Template that is applied when it is the page to be processed is the current page you are. If nothing is defined (a blank string or no definition), the template you defined in `setTemplate()` is used
- `setSeparatorTemplate()` Separator template is the string that is applied between each processed page. It is not included before the first call and after the last one. The defined template of this method is not affected by options and also it cannot process masks

Now we know how to create the `Doctrine_Pager_Layout` and the types that are around this component, it is time to view the basic usage:

Creating the pager layout is simple:

```
$pagerLayout = new Doctrine_Pager_Layout(
      new Doctrine_Pager(
            Doctrine_Query::create()
                  ->from( 'User u' )
                  ->leftJoin( 'u.Group g' )
                  ->orderby( 'u.username ASC' ),
            $currentPage,
            $resultsPerPage
      ),
      new Doctrine_Pager_Range_Sliding(array(
            'chunk' => 5
      )),
      'http://wwww.domain.com/app/User/list/page,{%page_number}'
);
```

Assigning templates for page links creation:

```
$pagerLayout->setTemplate('[<a href="{%url}">{%page}</a>]');
$pagerLayout->setSelectedTemplate('[{%page}]');

// Retrieving Doctrine_Pager instance
$pager = $pagerLayout->getPager();

// Fetching users
$users = $pager->execute(); // This is possible too!

// Displaying page links
// Displays: [1][2][3][4][5]
// With links in all pages, except the $currentPage (our example, page 1)
$pagerLayout->display();
```

Explaining this source, the first part creates the pager layout instance. Second, it defines the templates for all pages and for the current page. The last part, it retrieves the `Doctrine_Pager` object and executes the query, returning in variable `$users`. The last part calls the displar without any optional mask, which applies the template in all pages found by `Doctrine_Pager_Range::rangeAroundPage()` subclass call.

As you may see, there is no need to use other masks except the internals ones. Lets suppose we implement a new functionality to search for Users in our existent application, and we need to support this feature in pager layout too. To simplify our case, the search parameter is named "search" and is received through `$_GET` superglobal array. The first change we need to do is tho adjust the `Doctrine_Query` object and also the URL, to allow it to be sent to other pages.

Creating the pager layout:

```
$pagerLayout = new Doctrine_Pager_Layout(
      new Doctrine_Pager(
            Doctrine_Query::create()
                  ->from( 'User u' )
                  ->leftJoin( 'u.Group g' )
                  ->where('LOWER(u.username) LIKE LOWER(?)', array( '%'.$_GET['search'].'%' ) )
                  ->orderby( 'u.username ASC' ),
            $currentPage,
            $resultsPerPage
      ),
      new Doctrine_Pager_Range_Sliding(array(
            'chunk' => 5
      )),
      'http://wwww.domain.com/app/User/list/page,{%page_number}?search={%search}'
);
```

Check out the code and notice we added a new mask, called `{%search}`. We'll need to send it to the template processing at a later stage. We then assign the templates, just as defined before, without any change. And also, we do not need to change execution of query.

Assigning templates for page links creation:

```
$pagerLayout->setTemplate('[<a href="{%url}">{%page}</a>]');
$pagerLayout->setSelectedTemplate('[{%page}]');

// Fetching users
$users = $pagerLayout->execute();

foreach ($users as $user) {
    // ...
}
```

The method `display()` is the place where we define the custom mask we created. This method accepts 2 optional arguments: one array of optional masks and if the output should be returned instead of printed on screen. In our case, we need to define a new mask, the `{%search}`, which is the search offset of `$_GET` superglobal array. Also, remember that since it'll be sent as URL, it needs to be encoded. Custom masks are defined in key => value pairs. So all needed code is to define an array with the offset we desire and the value to be replaced:

```
// Displaying page links
$pagerLayout->display( array(
    'search' => urlencode($_GET['search'])
) );
```

`Doctrine_Pager_Layout` component offers accessors to defined resources. There is not need to define pager and pager range as variables and send to the pager layout. These instances can be retrieved by these accessors:

```
// Return the Pager associated to the Pager_Layout
$pagerLayout->getPager();

// Return the Pager_Range associated to the Pager_Layout
$pagerLayout->getPagerRange();

// Return the URL mask associated to the Pager_Layout
$pagerLayout->getUrlMask();

// Return the template associated to the Pager_Layout
$pagerLayout->getTemplate();

// Return the current page template associated to the Pager_Layout
$pagerLayout->getSelectedTemplate();

// Defines the Separator template, applied between each page
$pagerLayout->setSeparatorTemplate($separatorTemplate);

// Return the current page template associated to the Pager_Layout
$pagerLayout->getSeparatorTemplate();

// Handy method to execute the query without need to retrieve the Pager instance
$pagerLayout->execute($params = array(), $hydrationMode = null);
```

There are a couple of other methods that are available if you want to extend the `Doctrine_Pager_Layout` to create you custom layouter. We will see these methods in the next section.

Customizing pager layout

`Doctrine_Pager_Layout` does a really good job, but sometimes it is not enough. Let's suppose a situation where you have to create a layout of pagination like this one:

<< < 1 2 3 4 5 > >>

Currently, it is impossible with raw `Doctrine_Pager_Layout`. But if you extend it and use the available methods, you can achieve it. The base Layout class provides you some methods that can be used to create your own implementation. They are:

```php
// $this refers to an instance of Doctrine_Pager_Layout

// Defines a mask replacement. When parsing template, it converts replacement
// masks into new ones (or values), allowing to change masks behavior on the fly
$this->addMaskReplacement($oldMask, $newMask, $asValue = false);

// Remove a mask replacement
$this->removeMaskReplacement($oldMask);

// Remove all mask replacements
$this->cleanMaskReplacements();

// Parses the template and returns the string of a processed page
$this->processPage($options = array()); // Needs at least page_number offset in $options array

// Protected methods, although very useful

// Parse the template of a given page and return the processed template
$this->_parseTemplate($options = array());

// Parse the url mask to return the correct template depending of the options sent
// Already process the mask replacements assigned
$this->_parseUrlTemplate($options = array());

// Parse the mask replacements of a given page
$this->_parseReplacementsTemplate($options = array());

// Parse the url mask of a given page and return the processed url
$this->_parseUrl($options = array());

// Parse the mask replacements, changing from to-be replaced mask with new masks/values
$this->_parseMaskReplacements($str);
```

Now that you have a small tip of useful methods to be used when extending `Doctrine_Pager_Layout`, it's time to see our implemented class:

```php
class PagerLayoutWithArrows extends Doctrine_Pager_Layout
{
    public function display($options = array(), $return = false)
    {
        $pager = $this->getPager();
        $str = '';

        // First page
        $this->addMaskReplacement('page', '&laquo;', true);
        $options['page_number'] = $pager->getFirstPage();
        $str .= $this->processPage($options);

        // Previous page
        $this->addMaskReplacement('page', '&lsaquo;', true);
        $options['page_number'] = $pager->getPreviousPage();
        $str .= $this->processPage($options);

        // Pages listing
        $this->removeMaskReplacement('page');
        $str .= parent::display($options, true);

        // Next page
        $this->addMaskReplacement('page', '&rsaquo;', true);
        $options['page_number'] = $pager->getNextPage();
        $str .= $this->processPage($options);

        // Last page
        $this->addMaskReplacement('page', '&raquo;', true);
        $options['page_number'] = $pager->getLastPage();
        $str .= $this->processPage($options);

        // Possible wish to return value instead of print it on screen
        if ($return) {
            return $str;
        }

        echo $str;
    }
}
```

As you may see, I have to manual process the items <<, <, > and >>. I override the **{%page}** mask by setting a raw value to it (raw value is achieved by setting the third parameter as true). Then I define the only MUST HAVE information to process the page and call it. The return is the template processed as a string. I do it to any of my custom buttons.

Now supposing a totally different situation. Doctrine is framework agnostic, but many of our users use it together with Symfony. `Doctrine_Pager` and subclasses are 100% compatible with Symfony, but `Doctrine_Pager_Layout` needs some tweaks to get it working with Symfony's `link_to` helper function. To allow this usage with `Doctrine_Pager_Layout`, you have to extend it and add your custom processor over it. For example purpose (it works in Symfony), I used **{link_to}…{/link_to}** as a template processor to do this job. Here is the extended class and usage in Symfony:

```
class sfDoctrinePagerLayout extends Doctrine_Pager_Layout
{
    public function __construct($pager, $pagerRange, $urlMask)
    {
        sfLoader::loadHelpers(array('Url', 'Tag'));
        parent::__construct($pager, $pagerRange, $urlMask);
    }

    protected function _parseTemplate($options = array())
    {
        $str = parent::_parseTemplate($options);

        return preg_replace(
            '/\{link_to\}(.*?)\{\/link_to\}/', link_to('$1', $this->_parseUrl($options)), $str
        );
    }
}
```

Usage:

```
$pagerLayout = new sfDoctrinePagerLayout(
    $pager,
    new Doctrine_Pager_Range_Sliding(array('chunk' => 5)),
    '@hostHistoryList?page={%page_number}'
);

$pagerLayout->setTemplate('[{link_to}{%page}{/link_to}]');
```

Facade

Creating & Dropping Databases

Doctrine offers the ability to create and drop your databases from your defined Doctrine connections. The only trick to using it is that the name of your Doctrine connection must be the name of your database. This is required due to the fact that PDO does not offer a method for retrieving the name of the database you are connected to. So in order to create and drop the database Doctrine itself must be aware of the name of the database.

Convenience Methods

Doctrine offers static convenience methods available in the main Doctrine class. These methods perform some of the most used functionality of Doctrine with one method. Most of these methods are using in the `Doctrine_Task` system. These tasks are also what are executed from the `Doctrine_Cli`.

```php
// Turn debug on/off and check for whether it is on/off
Doctrine_Core::debug(true);

if (Doctrine_Core::debug()) {
    echo 'debugging is on';
} else {
    echo 'debugging is off';
}

// Get the path to your Doctrine libraries
$path = Doctrine_Core::getPath();

// Set the path to your Doctrine libraries if it is some non-default location
Doctrine_Core::setPath('/path/to/doctrine/libs');

// Load your models so that they are present and loaded for Doctrine to work with
// Returns an array of the Doctrine_Records that were found and loaded
$models = Doctrine_Core::loadModels('/path/to/models', Doctrine_Core::MODEL_LOADING_CONSERVATIVE); // or Doctrine_Core:
print_r($models);

// Get array of all the models loaded and present to Doctrine
$models = Doctrine_Core::getLoadedModels();

// Pass an array of classes to the above method and it will filter out the ones that are not Doctrine_Records
$models = Doctrine_Core::filterInvalidModels(array('User', 'Formatter', 'Doctrine_Record'));
print_r($models); // would return array('User') because Formatter and Doctrine_Record are not valid

// Get Doctrine_Connection object for an actual table name
$conn = Doctrine_Core::getConnectionByTableName('user'); // returns the connection object that the table name is associ
with.

// Generate YAML schema from an existing database
Doctrine_Core::generateYamlFromDb('/path/to/dump/schema.yml', array('connection_name'), $options);

// Generate your models from an existing database
Doctrine_Core::generateModelsFromDb('/path/to/generate/models', array('connection_name'), $options);

// Array of options and the default values
$options = array('packagesPrefix'        =>  'Package',
                 'packagesPath'          =>  '',
                 'packagesFolderName'    =>  'packages',
                 'suffix'                =>  '.php',
                 'generateBaseClasses'   =>  true,
                 'baseClassesPrefix'     =>  'Base',
                 'baseClassesDirectory'  =>  'generated',
                 'baseClassName'         =>  'Doctrine_Record');

// Generate your models from YAML schema
Doctrine_Core::generateModelsFromYaml('/path/to/schema.yml', '/path/to/generate/models', $options);

// Create the tables supplied in the array
Doctrine_Core::createTablesFromArray(array('User', 'Phoneumber'));

// Create all your tables from an existing set of models
// Will generate sql for all loaded models if no directory is given
Doctrine_Core::createTablesFromModels('/path/to/models');

// Generate string of sql commands from an existing set of models
// Will generate sql for all loaded models if no directory is given
Doctrine_Core::generateSqlFromModels('/path/to/models');

// Generate array of sql statements to create the array of passed models
Doctrine_Core::generateSqlFromArray(array('User', 'Phonenumber'));

// Generate YAML schema from an existing set of models
Doctrine_Core::generateYamlFromModels('/path/to/schema.yml', '/path/to/models');

// Create all databases for connections.
// Array of connection names is optional
Doctrine_Core::createDatabases(array('connection_name'));

// Drop all databases for connections
// Array of connection names is optional
Doctrine_Core::dropDatabases(array('connection_name'));

// Dump all data for your models to a yaml fixtures file
// 2nd argument is a bool value for whether or not to generate individual fixture files for each model. If true you nee
// to specify a folder instead of a file.
Doctrine_Core::dumpData('/path/to/dump/data.yml', true);

// Load data from yaml fixtures files
// 2nd argument is a bool value for whether or not to append the data when loading or delete all data first before load
Doctrine_Core::loadData('/path/to/fixture/files', true);

// Run a migration process for a set of migration classes
$num = 5; // migrate to version #5
Doctrine_Core::migration('/path/to/migrations', $num);

// Generate a blank migration class template
Doctrine_Core::generateMigrationClass('ClassName', '/path/to/migrations');

// Generate all migration classes for an existing database
Doctrine_Core::generateMigrationsFromDb('/path/to/migrations');

// Generate all migration classes for an existing set of models
// 2nd argument is optional if you have already loaded your models using loadModels()
Doctrine_Core::generateMigrationsFromModels('/path/to/migrations', '/path/to/models');

// Get Doctrine_Table instance for a model
$userTable = Doctrine_Core::getTable('User');
```

Tasks

Tasks are classes which bundle some of the core convenience methods in to tasks that can be easily executed by setting the required arguments. These tasks are directly used in the Doctrine command line interface.

```
BuildAll
BuildAllLoad
BuildAllReload
Compile
CreateDb
CreateTables
Dql
DropDb
DumpData
Exception
GenerateMigration
GenerateMigrationsDb
GenerateMigrationsModels
GenerateModelsDb
GenerateModelsYaml
GenerateSql
GenerateYamlDb
GenerateYamlModels
LoadData
Migrate
RebuildDb
```

You can read below about how to execute Doctrine Tasks standalone in your own scripts.

Command Line Interface

Introduction

The Doctrine Cli is a collection of tasks that help you with your day to do development and testing with your Doctrine implementation. Typically with the examples in this manual, you setup php scripts to perform whatever tasks you may need. This Cli tool is aimed at providing an out of the box solution for those tasks.

Tasks

Below is a list of available tasks for managing your Doctrine implementation.

```
$ ./doctrine
Doctrine Command Line Interface

./doctrine build-all
./doctrine build-all-load
./doctrine build-all-reload
./doctrine compile
./doctrine create-db
./doctrine create-tables
./doctrine dql
./doctrine drop-db
./doctrine dump-data
./doctrine generate-migration
./doctrine generate-migrations-db
./doctrine generate-migrations-models
./doctrine generate-models-db
./doctrine generate-models-yaml
./doctrine generate-sql
./doctrine generate-yaml-db
./doctrine generate-yaml-models
./doctrine load-data
./doctrine migrate
./doctrine rebuild-db
```

The tasks for the CLI are separate from the CLI and can be used standalone. Below is an example.

```
$task = new Doctrine_Task_GenerateModelsFromYaml();

$args = array('yaml_schema_path' => '/path/to/schema',
              'models_path'      => '/path/to/models');

$task->setArguments($args);

try {
  if ($task->validate()) {
    $task->execute();
  }
} catch (Exception $e) {
  throw new Doctrine_Exception($e->getMessage());
}
```

Usage

File named "doctrine" that is set to executable

```
#!/usr/bin/env php
[php]

chdir(dirname(__FILE__));
include('doctrine.php');

?>
```

Actual php file named "doctrine.php" that implements the `Doctrine_Cli`.

```
// Include your Doctrine configuration/setup here, your connections, models, etc.

// Configure Doctrine Cli
// Normally these are arguments to the cli tasks but if they are set here the arguments will be auto-filled and are not

$config = array('data_fixtures_path'  =>  '/path/to/data/fixtures',
                'models_path'         =>  '/path/to/models',
                'migrations_path'     =>  '/path/to/migrations',
                'sql_path'            =>  '/path/to/data/sql',
                'yaml_schema_path'    =>  '/path/to/schema');

$cli = new Doctrine_Cli($config);
$cli->run($_SERVER['argv']);
```

Now you can begin executing commands.

```
./doctrine generate-models-yaml
./doctrine create-tables
```

Sandbox

Installation

You can install the sandbox by downloading the special sandbox package from http://www.phpdoctrine.org/download or you can install it via svn below.

```
svn co http://www.phpdoctrine.org/svn/branches/0.11 doctrine
cd doctrine/tools/sandbox
chmod 0777 doctrine

./doctrine
```

The above steps should give you a functioning sandbox. Execute the ./doctrine command without specifying a task will show you an index of all the available cli tasks in Doctrine.

Conclusion

I hope some of these utilities discussed in this chapter are of use to you. Now lets discuss how Doctrine maintains stability and avoids regressions by using Unit Testing.

Unit Testing

Doctrine is programmatically tested using UnitTests. You can read more about unit testing here on Wikipedia.

Running tests

In order to run the tests that come with doctrine you need to check out the entire project, not just the lib folder.

```
$ svn co http://svn.doctrine-project.org/branches/1.2 /path/to/co/doctrine
```

Now change directory to the checked out copy of doctrine.

```
$ cd /path/to/co/doctrine
```

You should see the following files and directories listed.

```
CHANGELOG
COPYRIGHT
lib/
LICENSE
package.xml
tests/
tools/
vendor/
```

> It is not uncommon for the test suite to have fails that we are aware of. Often Doctrine will have test cases for bugs or enhancement requests that cannot be committed until later versions. Or we simply don't have a fix for the issue yet and the test remains failing. You can ask on the mailing list or in IRC for how many fails should be expected in each version of Doctrine.

CLI

To run tests on the command line, you must have php-cli installed.

Navigate to the `/path/to/co/doctrine/tests` folder and execute the `run.php` script:

```
$ cd /path/to/co/doctrine/tests
$ php run.php
```

This will print out a progress bar as it is running all the unit tests. When it is finished it will report to you what has passed and failed.

The CLI has several options available for running specific tests, groups of tests or filtering tests against class names of test suites. Run the following command to check out these options.

```
$ php run.php -help
```

You can run an individual group of tests like this:

```
$ php run.php --group data_dict
```

Browser

You can run the unit tests in the browser by navigating to `doctrine/tests/run.php`. Options can be set through `_GET` variables.

For example:

- `http://localhost/doctrine/tests/run.php`
- `http://localhost/doctrine/tests/run.php?filter=Limit&group[]=query&group[]=record`

> Please note that test results may very depending on your environment. For example if `php.ini` `apc.enable_cli` is set to 0 then some additional tests may fail.

Writing Tests

When writing your test case, you can copy `TemplateTestCase.php` to start off. Here is a sample test case:

```
class Doctrine_Sample_TestCase extends Doctrine_UnitTestCase
{
    public function prepareTables()
    {
        $this->tables[] = "MyModel1";
        $this->tables[] = "MyModel2";
        parent::prepareTables();
    }

    public function prepareData()
    {
      $this->myModel = new MyModel1();
      //$this->myModel->save();
    }

    public function testInit()
    {

    }

    // This produces a failing test
    public function testTest()
    {
        $this->assertTrue($this->myModel->exists());
        $this->assertEqual(0, 1);
        $this->assertIdentical(0, '0');
        $this->assertNotEqual(1, 2);
        $this->assertTrue((5 < 1));
        $this->assertFalse((1 > 2));
    }
}

class Model1 extends Doctrine_Record
{
}

class Model2 extends Doctrine_Record
{
}
```

> The model definitions can be included directly in the test case file or they can be put in `/path/to/co/doctrine/tests/models` and they will be autoloaded for you.

Once you are finished writing your test be sure to add it to `run.php` like the following.

```
$test->addTestCase(new Doctrine_Sample_TestCase());
```

Now when you execute run.php you will see the new failure reported to you.

Ticket Tests

In Doctrine it is common practice to commit a failing test case for each individual ticket that is reported in trac. These test cases are automatically added to run.php by reading all test case files found in the `/path/to/co/doctrine/tests/Ticket/` folder.

You can create a new ticket test case easily from the CLI:

```
$ php run.php --ticket 9999
```

If the ticket number 9999 doesn't already exist then the blank test case class will be generated for you at `/path/to/co/doctrine/tests/Ticket/9999TestCase.php`.

```
class Doctrine_Ticket_9999_TestCase extends Doctrine_UnitTestCase
{
}
```

Methods for testing

Assert Equal

```
// ...
    public function test1Equals1()
    {
        $this->assertEqual(1, 1);
    }
// ...
```

Assert Not Equal

```
// ...
    public function test1DoesNotEqual2()
    {
        $this->assertNotEqual(1, 2);
    }
// ...
```

Assert Identical

The `assertIdentical()` method is the same as the `assertEqual()` except that its logic is stricter and uses the `===` for comparing the two values.

```
// ...
    public function testAssertIdentical()
    {
        $this->assertIdentical(1, '1');
    }
// ...
```

> The above test would fail obviously because the first argument is the number 1 casted as PHP type integer and the second argument is the number 1 casted as PHP type string.

Assert True

```
// ...
    public function testAssertTrue()
    {
        $this->assertTrue(5 > 2);
    }
// ...
```

Assert False

```
// ...
    public function testAssertFalse()
    {
        $this->assertFalse(5 < 2);
    }
// ...
```

Mock Drivers

Doctrine uses mock drivers for all drivers other than sqlite. The following code snippet shows you how to use mock drivers:

```
class Doctrine_Sample_TestCase extends Doctrine_UnitTestCase
{
    public function testInit()
    {
        $this->dbh = new Doctrine_Adapter_Mock('oracle');
        $this->conn = Doctrine_Manager::getInstance()->openConnection($this->dbh);
    }
}
```

Now when you execute queries they won't actually be executed against a real database. Instead they will be collected in an array and you will be able to analyze the queries that were executed and make test assertions against them.

```
class Doctrine_Sample_TestCase extends Doctrine_UnitTestCase
{
    // ...

    public function testMockDriver()
    {
        $user = new User();
        $user->username = 'jwage';
        $user->password = 'changeme';
        $user->save();

        $sql = $this->dbh->getAll();

        // print the sql array to find the query you're looking for
        // print_r($sql);

        $this->assertEqual($sql[0], 'INSERT INTO user (username, password) VALUES (?, ?)');
    }
}
```

Test Class Guidelines

Every class should have at least one TestCase equivalent and they should inherit `Doctrine_UnitTestCase`. Test classes should refer to a class or an aspect of a class, and they should be named accordingly.

Some examples:

- `Doctrine_Record_TestCase` is a good name because it refers to the `Doctrine_Record` class
- `Doctrine_Record_State_TestCase` is also good, because it refers to the state aspect of the `Doctrine_Record` class.
- `Doctrine_PrimaryKey_TestCase` is a bad name, because it's too generic.

Test Method Guidelines

Methods should support agile documentation and should be named so that if it fails, it is obvious what failed. They should also give information of the system they test

For example the method test name `Doctrine_Export_Pgsql_TestCase::testCreateTableSupportsAutoincPks()` is a good name.

Test method names can be long, but the method content should not be. If you need several assert-calls, divide the method into smaller methods. There should never be assertions within any loops, and rarely within functions.

> Commonly used testing method naming convention `TestCase::test[methodName]` is **not** allowed in Doctrine. So in this case `Doctrine_Export_Pgsql_TestCase::testCreateTable()` would not be allowed!

Conclusion

Unit testing in a piece of software like Doctrine is so incredible important. Without it, it would be impossible to know if a change we make has any kind of negative affect on existing working use cases. With our collection of unit tests we can be sure that the changes we make won't break existing functionality.

Now lets move on to learn about how we can improve performance when using Doctrine.

Improving Performance

Introduction

Performance is a very important aspect of all medium to large sized applications. Doctrine is a large abstraction library that provides a database abstraction layer as well as object-relational mapping. While this provides a lot of benefits like portability and ease of development it's inevitable that this leads to drawbacks in terms of performance. This chapter tries to help you to get the best performance out of Doctrine.

Compile

Doctrine is quite big framework and usually dozens of files are being included on each request. This brings a lot of overhead. In fact these file operations are as time consuming as sending multiple queries to database server. The clean separation of class per file works well in developing environment, however when project goes commercial distribution the speed overcomes the clean separation of class per file -convention.

Doctrine offers method called `compile()` to solve this issue. The compile method makes a single file of most used Doctrine components which can then be included on top of your script. By default the file is created into Doctrine root by the name `Doctrine.compiled.php`.

Compiling is a method for making a single file of most used doctrine runtime components including the compiled file instead of multiple files (in worst cases dozens of files) can improve performance by an order of magnitude. In cases where this might fail, a `Doctrine_Exception` is throw detailing the error.

Lets create a compile script named `compile.php` to handle the compiling of Doctrine:

```
// compile.php

require_once('/path/to/doctrine/lib/Doctrine.php');
spl_autoload_register(array('Doctrine', 'autoload'));

Doctrine_Core::compile('Doctrine.compiled.php');
```

Now we can execute `compile.php` and a file named `Doctrine.compiled.php` will be generated in the root of your `doctrine_test` folder:

```
$ php compile.php
```

If you wish to only compile in the database drivers you are using you can pass an array of drivers as the second argument to `compile()`. For this example we are only using MySQL so lets tell Doctrine to only compile the `mysql` driver:

```
// compile.php
// ...
Doctrine_Core::compile('Doctrine.compiled.php', array('mysql'));
```

Now you can change your `bootstrap.php` script to include the compiled Doctrine:

```
// bootstrap.php
// ...
require_once('Doctrine.compiled.php');
// ...
```

Conservative Fetching

Maybe the most important rule is to be conservative and only fetch the data you actually need. This may sound trivial but laziness or lack of knowledge about the possibilities that are available often lead to a lot of unnecessary overhead.

Take a look at this example:

```
$record = $table->find($id);
```

How often do you find yourself writing code like that? It's convenient but it's very often not what you need. The example above will pull all columns of the record out of the database and populate the newly created object with that data. This not only means unnecessary network traffic but also means that Doctrine has to populate data into objects that is never used.

I'm sure you all know why a query like the following is not ideal:

```
SELECT
*
FROM my_table
```

The above is bad in any application and this is also true when using Doctrine. In fact it's even worse when using Doctrine because populating objects with data that is not needed is a waste of time.

Another important rule that belongs in this category is: **Only fetch objects when you really need them**. Doctrine has the ability to fetch "array graphs" instead of object graphs. At first glance this may sound strange because why use an object-relational mapper in the first place then? Take a second to think about it. PHP is by nature a precedural language that has been enhanced with a lot of features for decent OOP. Arrays are still the most efficient data structures you can use in PHP. Objects have the most value when they're used to accomplish complex business logic. It's a waste of resources when data gets wrapped in costly object structures when you have no benefit of that. Take a look at the following code that fetches all comments with some related data for an article, passing them to the view for display afterwards:

```
$q = Doctrine_Query::create()
    ->select('b.title, b.author, b.created_at')
    ->addSelect('COUNT(t.id) as num_comments')
    ->from('BlogPost b')
    ->leftJoin('b.Comments c')
    ->where('b.id = ?')
    ->orderBy('b.created_at DESC');

$blogPosts = $q->execute(array(1));
```

Now imagine you have a view or template that renders the most recent blog posts:

```
[php] foreach ($blogPosts as $blogPost):
  <li>
    <strong>
      [php] echo $blogPost['title'] ?>
    </strong>

    - Posted on [php] echo $blogPost['created_at'] ?>
    by [php] echo $blogPost['author'] ?>.

    <small>
      ([php] echo $blogPost['num_comments'] ?>)
    </small>
  </li>
[php] endforeach; ?>

?>
```

Can you think of any benefit of having objects in the view instead of arrays? You're not going to execute business logic in the view, are you? One parameter can save you a lot of unnecessary processing:

```
// ...

$blogPosts = $q->execute(array(1), Doctrine_Core::HYDRATE_ARRAY);
```

If you prefer you can also use the `setHydrationMethod()` method:

```
// ...

$q->setHydrationMode(Doctrine_Core::HYDRATE_ARRAY);

$blogPosts = $q->execute(array(1));
```

The above code will hydrate the data into arrays instead of objects which is much less expensive.

> One great thing about array hydration is that if you use the `ArrayAccess` on your objects you can easily switch your queries to use array hydration and your code will work exactly the same. For example the above code we wrote to render the list of the most recent blog posts would work when we switch the query behind it to array hydration.

Sometimes, you may want the direct output from PDO instead of an object or an array. To do this, set the hydration mode to `Doctrine_Core::HYDRATE_NONE`. Here's an example:

```
$q = Doctrine_Query::create()
    ->select('SUM(d.amount)')
    ->from('Donation d');

$results = $q->execute(array(), Doctrine_Core::HYDRATE_NONE);
```

You will need to print the results and find the value in the array depending on your DQL query:

```
print_r($results);
```

In this example the result would be accessible with the following code:

```
$total = $results[0][1];
```

> There are two important differences between `HYDRATE_ARRAY` and `HYDRATE_NONE` which you should consider before choosing which to use. `HYDRATE_NONE` is the fastest but the result is an array with numeric keys and so results would be referenced as `$result[0][0]` instead of `$result[0]['my_field']` with `HYDRATE_ARRAY`. Best practice would to use `HYDRATE_NONE` when retrieving large record sets or when doing many similar queries. Otherwise, `HYDRATE_ARRAY` is more comfortable and should be preferred.

Bundle your Class Files

When using Doctrine or any other large OO library or framework the number of files that need to be included on a regular HTTP request rises significantly. 50-100 includes per request are not uncommon. This has a significant performance impact because it results in a lot of disk operations. While this is generally no issue in a dev environment, it's not suited for production. The recommended way to handle this problem is to bundle the most-used classes of your libraries into a single file for production, stripping out any unnecessary whitespaces, linebreaks and comments. This way you get a significant performance improvement even without a bytecode cache (see next section). The best way to create such a bundle is probably as part of an automated build process i.e. with Phing.

Use a Bytecode Cache

A bytecode cache like APC will cache the bytecode that is generated by php prior to executing it. That means that the parsing of a file and the creation of the bytecode happens only once and not on every request. This is especially useful when using large libraries and/or frameworks. Together with file bundling for production this should give you a significant performance improvement. To get the most out of a bytecode cache you should contact the manual pages since most of these caches have a lot of configuration options which you can tweak to optimize the cache to your needs.

Free Objects

As of version 5.2.5, PHP is not able to garbage collect object graphs that have circular references, e.g. Parent has a reference to Child which has a reference to Parent. Since many doctrine model objects have such relations, PHP will not free their memory even when the objects go out of scope.

For most PHP applications, this problem is of little consequence, since PHP scripts tend to be short-lived. Longer-lived scripts, e.g. bulk data importers and exporters, can run out of memory unless you manually break the circular reference chains. Doctrine provides a `free()` function on `Doctrine_Record`, `Doctrine_Collection`, and `Doctrine_Query` which eliminates the circular references on those objects, freeing them up for garbage collection. Usage might look like:

Free objects when mass inserting records:

```
for ($i = 0; $i < 1000; $i++)
{
    $object = createBigObject();
    $object->save();
    $object->free(true);
}
```

You can also free query objects in the same way:

```
for ($i = 0; $i < 1000; $i++)
{
  $q = Doctrine_Query::create()
      ->from('User u');

  $results = $q->fetchArray();
  $q->free();
}
```

Or even better if you can reuse the same query object for each query in the loop that would be ideal:

```
$q = Doctrine_Query::create()
    ->from('User u');

for ($i = 0; $i < 1000; $i++)
{
  $results = $q->fetchArray();
  $q->free();
}
```

Other Tips

Helping the DQL parser

There are two possible ways when it comes to using DQL. The first one is writing the plain DQL queries and passing them to `Doctrine_Connection::query($dql)`. The second one is to use a `Doctrine_Query` object and its fluent interface. The latter should be preferred for all but very simple queries. The reason is that using the `Doctrine_Query` object and it's methods makes the life of the DQL parser a little bit easier. It reduces the amount of query parsing that needs to be done and is therefore faster.

Efficient relation handling

When you want to add a relation between two components you should **NOT** do something like the following:

> The following example assumes a many-many between `Role` and `User`.

```
$role = new Role();
$role->name = 'New Role Name';

$user->Roles[] = $newRole;
```

> The above code will load all roles of the user from the database if they're not yet loaded! Just to add one new link!

The following is the recommended way instead:

```
$userRole = new UserRole();
$userRole->role_id = $role_id;
$userRole->user_id = $user_id;
$userRole->save();
```

Conclusion

Lots of methods exist for improving performance in Doctrine. It is highly recommended that you consider some of the methods described above.

Now lets move on to learn about some of the technology used in Doctrine.

Technology

Introduction

Doctrine is a product of the work of many people. Not just the people who have coded and documented this software are the only ones responsible for this great framework. Other ORMs in other languages are a major resource for us as we can learn from what they have already done.

> Doctrine has also borrowed pieces of code from other open source projects instead of re-inventing the wheel. Two of the projects borrowed from are symfony and the Zend Framework. The relevant license information can be found in the root of Doctrine when you download it in a file named `LICENSE`.

Architecture

Doctrine is divided into three main packages: CORE, ORM and DBAL. Below is a list of some of the main classes that make up each of the packages.

Doctrine CORE

- Doctrine
- Doctrine_Manager
- Doctrine_Connection
- Doctrine_Compiler
- Doctrine_Exception
- Doctrine_Formatter
- Doctrine_Object
- Doctrine_Null
- Doctrine_Event
- Doctrine_Overloadable
- Doctrine_Configurable
- Doctrine_EventListener

Doctrine DBAL

- Doctrine_Expression_Driver
- Doctrine_Export
- Doctrine_Import
- Doctrine_Sequence
- Doctrine_Transaction
- Doctrine_DataDict

Doctrine DBAL is also divided into driver packages.

Doctrine ORM

- Doctrine_Record
- Doctrine_Table
- Doctrine_Relation
- Doctrine_Expression
- Doctrine_Query
- Doctrine_RawSql
- Doctrine_Collection
- Doctrine_Tokenizer

Other miscellaneous packages.

- Doctrine_Validator
- Doctrine_Hook
- Doctrine_View

There are also behaviors for Doctrine:

- Geographical
- I18n
- NestedSet
- Searchable
- Sluggable
- SoftDelete
- Timestampable
- Versionable

Design Patterns Used

`GoF (Gang of Four)` design patterns used:

- Singleton, for forcing only one instance of `Doctrine_Manager`
- Composite, for leveled configuration
- Factory, for connection driver loading and many other things
- Observer, for event listening
- Flyweight, for efficient usage of validators
- Iterator, for iterating through components (Tables, Connections, Records etc.)
- State, for state-wise connections
- Strategy, for algorithm strategies

Enterprise application design patterns used:

- Active Record, Doctrine is an implementation of this pattern
- UnitOfWork, for maintaining a list of objects affected in a transaction
- Identity Field, for maintaining the identity between record and database row
- Metadata Mapping, for Doctrine DataDict
- Dependent Mapping, for mapping in general, since all records extend `Doctrine_Record` which performs all mappings
- Foreign Key Mapping, for one-to-one, one-to-many and many-to-one relationships
- Association Table Mapping, for association table mapping (most commonly many-to-many relationships)
- Lazy Load, for lazy loading of objects and object properties
- Query Object, DQL API is actually an extension to the basic idea of Query Object pattern

Speed

- **Lazy initialization** - For collection elements
- **Subselect fetching** - Doctrine knows how to fetch collections efficiently using a subselect.
- **Executing SQL statements later, when needed** : The connection never issues an INSERT or UPDATE until it is actually needed. So if an exception occurs and you need to abort the transaction, some statements will never actually be issued. Furthermore, this keeps lock times in the database as short as possible (from the late UPDATE to the transaction end).
- **Join fetching** - Doctrine knows how to fetch complex object graphs using joins and subselects
- **Multiple collection fetching strategies** - Doctrine has multiple collection fetching strategies for performance tuning.
- **Dynamic mixing of fetching strategies** - Fetching strategies can be mixed and for example users can be fetched in a batch collection while users' phonenumbers are loaded in offset collection using only one query.
- **Driver specific optimizations** - Doctrine knows things like bulk-insert on mysql.
- **Transactional single-shot delete** - Doctrine knows how to gather all the primary keys of the pending objects in delete list and performs only one sql delete statement per table.
- **Updating only the modified columns.** - Doctrine always knows which columns have been changed.
- **Never inserting/updating unmodified objects.** - Doctrine knows if the the state of the record has changed.

- **PDO for database abstraction** - PDO is by far the fastest availible database abstraction layer for php.

Conclusion

This chapter should have given you a complete birds eye view of all the components of Doctrine and how they are organized. Up until now you have seen them all used a part from each other but the separate lists of the three main packages should have made things very clear for you if it was not already.

Now we are ready to move on and learn about how to deal with Doctrine throwing exceptions in the Exceptions and Warnings chapter.

Exceptions and Warnings

Manager exceptions

`Doctrine_Manager_Exception` is thrown if something failed at the connection management

```
try {
    $manager->getConnection('unknown');
} catch (Doctrine_Manager_Exception) {
    // catch errors
}
```

Relation exceptions

Relation exceptions are being thrown if something failed during the relation parsing.

Connection exceptions

Connection exceptions are being thrown if something failed at the database level. Doctrine offers fully portable database error handling. This means that whether you are using sqlite or some other database you can always get portable error code and message for the occurred error.

```
try {
    $conn->execute('SELECT * FROM unknowntable');
} catch (Doctrine_Connection_Exception $e) {
    echo 'Code : ' . $e->getPortableCode();
    echo 'Message : ' . $e->getPortableMessage();
}
```

Query exceptions

An exception will be thrown when a query is executed if the DQL query is invalid in some way.

Conclusion

Now that you know how to deal with Doctrine throwing exceptions lets move on and show you some real world schemas that would be used in common web applications found today on the web.

Real World Examples

User Management System

In almost all applications you need to have some kind of security or authentication system where you have users, roles, permissions, etc. Below is an example where we setup several models that give you a basic user management and security system.

```php
class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 255, array(
                'unique' => true
            )
        );
        $this->hasColumn('password', 'string', 255);
    }
}

class Role extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 255);
    }
}

class Permission extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 255);
    }
}

class RolePermission extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('role_id', 'integer', null, array(
                'primary' => true
            )
        );
        $this->hasColumn('permission_id', 'integer', null, array(
                'primary' => true
            )
        );
    }

    public function setUp()
    {
        $this->hasOne('Role', array(
                'local' => 'role_id',
                'foreign' => 'id'
            )
        );
        $this->hasOne('Permission', array(
                'local' => 'permission_id',
                'foreign' => 'id'
            )
        );
    }
}

class UserRole extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer', null, array(
                'primary' => true
            )
        );
        $this->hasColumn('role_id', 'integer', null, array(
                'primary' => true
            )
        );
    }

    public function setUp()
    {
        $this->hasOne('User', array(
                'local' => 'user_id',
                'foreign' => 'id'
            )
        );
        $this->hasOne('Role', array(
                'local' => 'role_id',
                'foreign' => 'id'
            )
        );
    }
}

class UserPermission extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer', null, array(
                'primary' => true
            )
        );
        $this->hasColumn('permission_id', 'integer', null, array(
                'primary' => true
            )
        );
    }

    public function setUp()
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
User:
  columns:
    username: string(255)
    password: string(255)
  relations:
    Roles:
      class: Role
      refClass: UserRole
      foreignAlias: Users
    Permissions:
      class: Permission
      refClass: UserPermission
      foreignAlias: Users

Role:
  columns:
    name: string(255)
  relations:
    Permissions:
      class: Permission
      refClass: RolePermission
      foreignAlias: Roles

Permission:
  columns:
    name: string(255)

RolePermission:
  columns:
    role_id:
      type: integer
      primary: true
    permission_id:
      type: integer
      primary: true
  relations:
    Role:
    Permission:

UserRole:
  columns:
    user_id:
      type: integer
      primary: true
    role_id:
      type: integer
      primary: true
  relations:
    User:
    Role:

UserPermission:
  columns:
    user_id:
      type: integer
      primary: true
    permission_id:
      type: integer
      primary: true
  relations:
    User:
    Permission:
```

Forum Application

Below is an example of a forum application where you have categories, boards, threads and posts:

```
class Forum_Category extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('root_category_id', 'integer', 10);
        $this->hasColumn('parent_category_id', 'integer', 10);
        $this->hasColumn('name', 'string', 50);
        $this->hasColumn('description', 'string', 99999);
    }

    public function setUp()
    {
        $this->hasMany('Forum_Category as Subcategory', array(
                'local' => 'parent_category_id',
                'foreign' => 'id'
            )
        );
        $this->hasOne('Forum_Category as Rootcategory', array(
                'local' => 'root_category_id',
                'foreign' => 'id'
            )
        );
    }
}

class Forum_Board extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('category_id', 'integer', 10);
        $this->hasColumn('name', 'string', 100);
        $this->hasColumn('description', 'string', 5000);
    }

    public function setUp()
    {
        $this->hasOne('Forum_Category as Category', array(
                'local' => 'category_id',
                'foreign' => 'id'
            )
        );
        $this->hasMany('Forum_Thread as Threads',  array(
                'local' => 'id',
                'foreign' => 'board_id'
            )
        );
    }
}

class Forum_Entry extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('author', 'string', 50);
        $this->hasColumn('topic', 'string', 100);
        $this->hasColumn('message', 'string', 99999);
        $this->hasColumn('parent_entry_id', 'integer', 10);
        $this->hasColumn('thread_id', 'integer', 10);
        $this->hasColumn('date', 'integer', 10);
    }

    public function setUp()
    {
        $this->hasOne('Forum_Entry as Parent',  array(
                'local' => 'parent_entry_id',
                'foreign' => 'id'
            )
        );
        $this->hasOne('Forum_Thread as Thread', array(
                'local' => 'thread_id',
                'foreign' => 'id'
            )
        );
    }
}

class Forum_Thread extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('board_id', 'integer', 10);
        $this->hasColumn('updated', 'integer', 10);
        $this->hasColumn('closed', 'integer', 1);
    }

    public function setUp()
    {
        $this->hasOne('Forum_Board as Board', array(
                'local' => 'board_id',
                'foreign' => 'id'
            )
        );

        $this->hasMany('Forum_Entry as Entries', array(
                'local' => 'id',
                'foreign' => thread_id'
            )
        );
    }
}
```

Here is the same example in YAML format. You can read more about YAML in the YAML Schema Files chapter:

```yaml
---
Forum_Category:
  columns:
    root_category_id: integer(10)
    parent_category_id: integer(10)
    name: string(50)
    description: string(99999)
  relations:
    Subcategory:
      class: Forum_Category
      local: parent_category_id
      foreign: id
    Rootcategory:
      class: Forum_Category
      local: root_category_id
      foreign: id

Forum_Board:
  columns:
    category_id: integer(10)
    name: string(100)
    description: string(5000)
  relations:
    Category:
      class: Forum_Category
      local: category_id
      foreign: id
    Threads:
      class: Forum_Thread
      local: id
      foreign: board_id

Forum_Entry:
  columns:
    author: string(50)
    topic: string(100)
    message: string(99999)
    parent_entry_id: integer(10)
    thread_id: integer(10)
    date: integer(10)
  relations:
    Parent:
      class: Forum_Entry
      local: parent_entry_id
      foreign: id
    Thread:
      class: Forum_Thread
      local: thread_id
      foreign: id

Forum_Thread:
  columns:
    board_id: integer(10)
    updated: integer(10)
    closed: integer(1)
  relations:
    Board:
      class: Forum_Board
      local: board_id
      foreign: id
    Entries:
      class: Forum_Entry
      local: id
      foreign: thread_id
```

Conclusion

I hope that these real world schema examples will help you with using Doctrine in the real world in your application. The last chapter of this book will discuss the coding standards used in Doctrine and are recommended for you to use in your application as well. Remember, consistency in your code is key!

Coding Standards

PHP File Formatting

General

For files that contain only PHP code, the closing tag ("`?>`") is never permitted. It is not required by PHP. Not including it prevents trailing whitespace from being accidentally injected into the output.

> Inclusion of arbitrary binary data as permitted by `__HALT_COMPILER()` is prohibited from any Doctrine framework PHP file or files derived from them. Use of this feature is only permitted for special installation scripts.

Indentation

Use an indent of 4 spaces, with no tabs.

Maximum Line Length

The target line length is 80 characters, i.e. developers should aim keep code as close to the 80-column boundary as is practical. However, longer lines are acceptable. The maximum length of any line of PHP code is 120 characters.

Line Termination

Line termination is the standard way for Unix text files to represent the end of a line. Lines must end only with a linefeed (LF). Linefeeds are represented as ordinal 10, or hexadecimal 0x0A.

You should not use carriage returns (CR) like Macintosh computers (0x0D) and do not use the carriage return/linefeed combination (CRLF) as Windows computers (0x0D, 0x0A).

Naming Conventions

Classes

The Doctrine ORM Framework uses the same class naming convention as PEAR and Zend framework, where the names of the classes directly map to the directories in which they are stored. The root level directory of the Doctrine Framework is the "Doctrine/" directory, under which all classes are stored hierarchially.

Class names may only contain alphanumeric characters. Numbers are permitted in class names but are discouraged. Underscores are only permitted in place of the path separator, eg. the filename "Doctrine/Table/Exception.php" must map to the class name "`Doctrine_Table_Exception`".

If a class name is comprised of more than one word, the first letter of each new word must be capitalized. Successive capitalized letters are not allowed, e.g. a class "XML_Reader" is not allowed while "Xml_Reader" is acceptable.

Interfaces

Interface classes must follow the same conventions as other classes (see above).

They must also end with the word "Interface" (unless the interface is approved not to contain it such as `Doctrine_Overloadable`). Some examples:

Examples

- `Doctrine_Adapter_Interface`
- `Doctrine_EventListener_Interface`

Filenames

For all other files, only alphanumeric characters, underscores, and the dash character ("-") are permitted. Spaces are prohibited.

Any file that contains any PHP code must end with the extension ".php". These examples show the acceptable filenames for containing the class names from the examples in the section above:

- `Doctrine/Adapter/Interface.php`
- `Doctrine/EventListener/Interface`

File names must follow the mapping to class names described above.

Functions and Methods

Function names may only contain alphanumeric characters and underscores are not permitted. Numbers are permitted in function names but are highly discouraged. They must always start with a lowercase letter and when a function name consists of more than one word, the first letter of each new word must be capitalized. This is commonly called the "studlyCaps" or "camelCaps" method. Verbosity is encouraged and function names should be as verbose as is practical to enhance the understandability of code.

For object-oriented programming, accessors for objects should always be prefixed with either "get" or "set". This applies to all classes except for `Doctrine_Record` which has some accessor methods prefixed with 'obtain' and 'assign'. The reason for this is that since all user defined ActiveRecords inherit `Doctrine_Record`, it should populate the get / set namespace as little as possible.

> Functions in the global scope ("floating functions") are NOT permmitted. All static functions should be wrapped in a static class.

Variables

Variable names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in variable names but are discouraged. They must always start with a lowercase letter and follow the "camelCaps" capitalization convention. Verbosity is encouraged. Variables should always be as verbose as practical. Terse variable names such as "$i" and "$n" are discouraged for anything other than the smallest loop contexts. If a loop contains more than 20 lines of code, the variables for the indices need to have more descriptive names. Within the framework certain generic object variables should always use the following names:

Object type	Variable name
`Doctrine_Connection`	$conn
`Doctrine_Collection`	$coll
`Doctrine_Manager`	$manager
`Doctrine_Query`	$q

There are cases when more descriptive names are more appropriate (for example when multiple objects of the same class are used in same context), in that case it is allowed to use different names than the ones mentioned.

Constants

Constants may contain both alphanumeric characters and the underscore. They must always have all letters capitalized. For readablity reasons, words in constant names must be separated by underscore characters. For example, `ATTR_EXC_LOGGING` is permitted but `ATTR_EXCLOGGING` is not.Constants must be defined as class members by using the "const" construct. Defining constants in the global scope with "define" is NOT permitted.

```
class Doctrine_SomeClass
{
    const MY_CONSTANT = 'something';
}
echo $Doctrine_SomeClass::MY_CONSTANT;
```

Record Columns

All record columns must be in lowercase and usage of underscores(_) are encouraged for columns that consist of more than one word.

```
class User
{
    public function setTableDefinition()
    {
        $this->hasColumn('home_address', 'string');
    }
}
```

Foreign key fields must be in format `[table_name]_[column]`. The next example is a field that is a foreign key that points to `user(id)`:

```php
class Phonenumber extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer');
    }
}
```

Coding Style

PHP Code Demarcation

PHP code must always be delimited by the full-form, standard PHP tags and short tags are never allowed. For files containing only PHP code, the closing tag must always be omitted

Strings

When a string is literal (contains no variable substitutions), the apostrophe or "single quote" must always used to demarcate the string:

Literal String

```php
$string = 'something';
```

When a literal string itself contains apostrophes, it is permitted to demarcate the string with quotation marks or "double quotes". This is especially encouraged for SQL statements:

String Containing Apostrophes

```php
$sql = "SELECT id, name FROM people WHERE name = 'Fred' OR name = 'Susan'";
```

Variable Substitution

Variable substitution is permitted using the following form:

```php
// variable substitution
$greeting = "Hello $name, welcome back!";
```

String Concatenation

Strings may be concatenated using the "." operator. A space must always be added before and after the "." operator to improve readability:

```php
$framework = 'Doctrine' . ' ORM ' . 'Framework';
```

Concatenation Line Breaking

When concatenating strings with the "." operator, it is permitted to break the statement into multiple lines to improve readability. In these cases, each successive line should be padded with whitespace such that the "."; operator is aligned under the "=" operator:

```php
$sql = "SELECT id, name FROM user "
     . "WHERE name = ? "
     . "ORDER BY name ASC";
```

Arrays

Negative numbers are not permitted as indices and a indexed array may be started with any non-negative number, however this is discouraged and it is recommended that all arrays have a base index of 0. When declaring indexed arrays with the array construct, a trailing space must be added after each comma delimiter to improve readability. It is also permitted to declare multiline indexed arrays using the "array" construct. In this case, each successive line must be padded with spaces. When declaring associative arrays with the array construct, it is encouraged to break the statement into multiple lines. In this case, each successive line must be padded with whitespace such that both the keys and the values are aligned:

```php
$sampleArray = array('Doctrine', 'ORM', 1, 2, 3);

$sampleArray = array(1, 2, 3,
                     $a, $b, $c,
                     56.44, $d, 500);

$sampleArray = array('first'  => 'firstValue',
                     'second' => 'secondValue');
```

Classes

Classes must be named by following the naming conventions. The brace is always written next line after the class name (or interface declaration). Every class must have a documentation block that conforms to the PHPDocumentor standard. Any code within a class must be indented four spaces and only one class is permitted per PHP file. Placing additional code in a class file is NOT permitted.

This is an example of an acceptable class declaration:

```
/**
 * Documentation here
 */
class Doctrine_SampleClass
{
    // entire content of class
    // must be indented four spaces
}
```

Functions and Methods

Methods must be named by following the naming conventions and must always declare their visibility by using one of the private, protected, or public constructs. Like classes, the brace is always written next line after the method name. There is no space between the function name and the opening parenthesis for the arguments. Functions in the global scope are strongly discouraged. This is an example of an acceptable function declaration in a class:

```
/**
 * Documentation Block Here
 */
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar()
    {
        // entire content of function
        // must be indented four spaces
    }

    public function bar2()
    {

    }
}
```

> Functions must be separated by only ONE single new line like is done above between the `bar()` and `bar2()` methods.

Passing by-reference is permitted in the function declaration only:

```
/**
 * Documentation Block Here
 */
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar(&$baz)
    {
    }
}
```

Call-time pass by-reference is prohibited. The return value must not be enclosed in parentheses. This can hinder readability and can also break code if a method is later changed to return by reference.

```
/**
 * Documentation Block Here
 */
class Foo
{
    /**
     * WRONG
     */
    public function bar() {
        return($this->bar);
    }

    /**
     * RIGHT
     */
    public function bar()
    {
        return $this->bar;
    }
}
```

Function arguments are separated by a single trailing space after the comma delimiter. This is an example of an acceptable function call for a function that takes three arguments:

```
threeArguments(1, 2, 3);
```

Call-time pass by-reference is prohibited. See above for the proper way to pass function arguments by-reference. For functions whose arguments permitted arrays, the function call may include the array construct and can be split into multiple lines to improve readability. In these cases, the standards for writing arrays still apply:

```
threeArguments(array(1, 2, 3), 2, 3);

threeArguments(array(1, 2, 3, 'Framework',
                     'Doctrine', 56.44, 500), 2, 3);
```

Control Statements

Control statements based on the if and elseif constructs must have a single space before the opening parenthesis of the conditional, and a single space after the closing parenthesis. Within the conditional statements between the parentheses, operators must be separated by spaces for readability. Inner parentheses are encouraged to improve logical grouping of larger conditionals. The opening brace is written on the same line as the conditional statement. The closing brace is always written on its own line. Any content within the braces must be indented four spaces.

```
if ($foo != 2) {
    $foo = 2;
}
```

For if statements that include elseif or else, the formatting must be as in these examples:

```
if ($foo != 1) {
    $foo = 1;
} else {
    $foo = 3;
}

if ($foo != 2) {
    $foo = 2;
} elseif ($foo == 1) {
    $foo = 3;
} else {
    $foo = 11;
}
```

When ! operand is being used it must use the following formatting:

```
if ( ! $foo) {

}
```

Control statements written with the switch construct must have a single space before the opening parenthesis of the conditional statement, and also a single space after the closing parenthesis. All content within the switch statement must be indented four spaces. Content under each case statement must be indented an additional four spaces but the breaks must be at the same indentation level as the case statements.

```
switch ($case) {
    case 1:
    case 2:
    break;
    case 3:
    break;
    default:
    break;
}
```

The construct default may never be omitted from a switch statement.

Inline Documentation

Documentation Format:

All documentation blocks ("docblocks") must be compatible with the phpDocumentor format. Describing the phpDocumentor format is beyond the scope of this document. For more information, visit: http://phpdoc.org/

Every method, must have a docblock that contains at a minimum:

- A description of the function
- All of the arguments
- All of the possible return values
- It is not necessary to use the @access tag because the access level is already known from the public, private, or protected construct used to declare the function.

If a function/method may throw an exception, use @throws:

```
/*
 * Test function
 *
 * @throws Doctrine_Exception
 */
public function test()
{
  throw new Doctrine_Exception('This function did not work');
}
```

Conclusion

This is the last chapter of **Doctrine ORM for PHP - Guide to Doctrine for PHP**. I really hope that this book was a useful piece of documentation and that you are now comfortable with using Doctrine and will be able to come back to easily reference things as needed.

As always, follow the Doctrine :)

Thanks, Jon