Doctrine Manual

Doctrine Core Team

1.0
November 26, 2008

Contents

Chapter 1

Getting started

1.1 Requirements

Doctrine requires PHP >= 5.2.3+. 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 or any other non official package, you may be required to perform
additional configurations.

1.2 Checking PDO driver installation

To know if your server supports the desirable PDO driver, you need to write a simple file with this content:

Listing 1.1:

```
<?php phpinfo(); ?>
```

Upload it to your server and execute this script.
You will notice a PDO box section scrolling down the page. Check if the driver you desire is active.

If your desired driver is not active, follow the PDO driver installation instruction[1] at PHP manual.

1.3 Installation

There are currently four different methods to install Doctrine.

- SVN (subversion)
- SVN externals
- Pear
- Zip-package

[1] http://php.net/manual/en/pdo.installation.php

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.

1.3.1 Sandbox Package

Doctrine also provides a special package which is a zero configuration Doctrine implementation. It
includes a fully featured command line interface for managing your schema files, migrations, database connections, data fixtures, and many other features. You can read about the sandbox package
and how to use it in the 12 chapter under the Sandbox section.

Below you will find the url to a tutorial on how to how to get started using
Doctrine with the sandbox package. With the sandbox and this tutorial you can get Doctrine up
and running in under 5 minutes. The tutorial offers example schema files, data fixtures, and a simple
script for managing a "User" model with Doctrine. Simple create, update, delete functionality.

The tutorial can be found here http://trac.phpdoctrine.org/wiki/MyFirstProject and the sandbox package
can be downloaded from here http://www.phpdoctrine.org/download

1.3.2 SVN

The installation of doctrine via SVN is very easy. Just get the latest revision of Doctrine from http://svn.phpdoctrine.org/branches/0.11.

In order to check out Doctrine in the current directory using the **svn** command line tool use the following code:

Listing 1.2:

```
svn co http://svn.phpdoctrine.org/branches/1.0 .
```

If you do not have a SVN client, chose one from the list below. Find the **Checkout** option and enter svn.phpdoctrine.org/branches/0.11 in the **path** or **repository url** parameter. There is no need for a username
or password to check out Doctrine.

- TortoiseSVN[2] a Windows application that integrates into Windows Explorer
- svnx[3] a Mac OS X GUI svn application
- Eclipse has SVN integration through the subeclipse[4] plugin

You can update to the latest version with

Listing 1.3:

```
svn update
```

in your doctrine directory.

[2] http://tortoisesvn.tigris.org/
[3] http://www.apple.com/downloads/macosx/development_tools/svnx.html
[4] http://subclipse.tigris.org/

1.3.3 SVN externals

If your project is under version control with SVN, you should set up doctrine via svn externals. You can do this with
the **svn** command line tool:

Listing 1.4:

```
svn pe svn:externals /path/to/project
```

You have to put the following line in the editor and save the changes.

Listing 1.5:

```
doctrine http://svn.phpdoctrine.org/tags/0.11.0
```

Afterwards you can download doctrine with

Listing 1.6:

```
svn update
```

1.3.4 PEAR

You can install Doctrine via PEAR with the following command:

Listing 1.7:

```
pear install http://pear.phpdoctrine.org/Doctrine-0.11.0
```

1.3.5 Zip-package

You can download Doctrine as a .zip or .tgz (for Linux) package from http://www.phpdoctrine.org/download.
Simply unzip it to your project directory with your favorite zip tool.

Under Linux you can extract the .tgz package with the following command line instruction:

Listing 1.8:

```
tar xzf Doctrine-0.11.0.tgz
```

1.4 Starting new project

Doctrine_Record is the basic component of every doctrine-based project. There should be atleast one Doctrine_Record for
each of your database tables. Doctrine_Record follows the [http://www.martinfowler.com/eaaCatalog/activeRecord.html
Active Record pattern]

Doctrine always adds a primary key column named 'id' to tables that doesn't have any primary keys specified. Only thing
you need to for creating database tables is defining a class which extends Doctrine_Record and setting a setTableDefinition
method with hasColumn() method calls and by exporting those classes.

Lets say we want to create a database table called 'user' with columns id(primary key), name, username, password and
created. Provided that you have already installed Doctrine these few lines of code are all you need:

User.php :

Listing 1.9:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        // set 'user' table columns, note that
        // id column is auto-created as no primary key is specified

        $this->hasColumn('name', 'string',30);
        $this->hasColumn('username', 'string',20);
        $this->hasColumn('password', 'string',16);
    }

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

?>
```

You can alternatively specify your Doctrine schema information as a YAML schema file. Below is an example user.yml file
which you can generate your Doctrine_Record from.

Listing 1.10:

```
---
User:
  actAs: [Timestampable]
  columns:
    name: string(30)
    username: string(20)
    password: string(16)
```

You can generate the php code from the yaml with the following code.

Listing 1.11:

```
<?php

Doctrine::generateModelsFromYaml('/path/to/user.yml', '/path/to/generate/models')
    ;

?>
```

Have a look in /path/to/generate/models/ and /path/to/generate/models/generated. You will see User.php and BaseUser.php.
User.php is for you to add your own custom functionality, and BaseUser.php is the code which is automatically regenerated
from the YAML schema file above each time.

Now that we have a Doctrine_Record class, we can export it to the database and create the tables. For exporting the user
class into database we need a simple build script:

Listing 1.12:

```
<?php

//require the base Doctrine class
require_once('path-to-doctrine/lib/Doctrine.php');

//register the autoloader
spl_autoload_register(array('Doctrine', 'autoload'));

require_once('User.php');

//set up a connection
Doctrine_Manager::connection('mysql://user:pass@localhost/test');

//export the classes
Doctrine::createTablesFromArray(array('User'));

?>
```

We now have a user model that supports basic CRUD opperations!

1.5 Working with existing databases

1.5.1 Introduction

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.

1.5.2 Making the first import

Let's consider we have a mysql database called test with a single table called 'file'.

The file table has been created with the following sql statement:

Listing 1.13:

```
CREATE TABLE file (
    id INT UNSIGNED AUTO_INCREMENT NOT NULL,
    name VARCHAR(150),
    size BIGINT,
    modified BIGINT,
    type VARCHAR(10),
    content TEXT,
    path TEXT,
    PRIMARY KEY(id))
```

Now we would like to convert it into Doctrine_Record class. It can be achieved easily with the following code snippet:

Listing 1.14:

```
<?php

require_once('path-to-doctrine/lib/Doctrine.php');
```

```
spl_autoload_register(array('Doctrine', 'autoload'));
Doctrine_Manager::connection('mysql://root:dc34@localhost/test');

// import method takes one parameter: the import directory (the directory where
// the generated record files will be put in
Doctrine::generateModelsFromDb('myrecords');

?>
```

That's it! Now there should be a file called BaseFile.php in your myrecords/generated directory. The file should look like:

Listing 1.15:

```
<?php

/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
abstract class BaseFile extends Doctrine_Record
{

  public function setTableDefinition()
  {
    $this->setTableName('file');
    $this->hasColumn('id', 'integer', 4, array('unsigned' => 1, 'values' =>
        array(), 'primary' => true, 'notnull' => true, 'autoincrement' => true))
        ;
    $this->hasColumn('name', 'string', 150, array('fixed' => false, 'values' =>
         array(), 'primary' => false, 'notnull' => false, 'autoincrement' =>
        false));
    $this->hasColumn('size', 'integer', 8, array('unsigned' => 0, 'values' =>
        array(), 'primary' => false, 'notnull' => false, 'autoincrement' =>
        false));
    $this->hasColumn('modified', 'integer', 8, array('unsigned' => 0, 'values' =
        >  array(), 'primary' => false, 'notnull' => false, 'autoincrement' =>
        false));
    $this->hasColumn('type', 'string', 10, array('fixed' => false, 'values' =>
        array(), 'primary' => false, 'notnull' => false, 'autoincrement' =>
        false));
    $this->hasColumn('content', 'string', null, array('fixed' => false, 'values'
         =>  array(), 'primary' => false, 'notnull' => false, 'autoincrement' =>
         false));
    $this->hasColumn('path', 'string', null, array('fixed' => false, 'values' =>
          array(), 'primary' => false, 'notnull' => false, 'autoincrement' =>
        false));
  }

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

}

?>
```

You should also have a file called File.php in your myrecords directory. The file should look like:

Listing 1.16:

```
<?php

/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
```

```
class File extends BaseFile
{

}

?>
```

Doctrine will automatically generate a skeleton Doctrine_Table class for the model at myrecords/UserTable.php. The file
should look like:

Listing 1.17:

```
<?php

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

}

?>
```

This is where you can put your custom finder methods which can be used by calling Doctrine::getTable('User').

1.6 Creating tables

1.6.1 Introduction

Doctrine supports exporting record classes into database. This means that based on the definitions given in your record
classes Doctrine will create the tables in your database.

Lets say we have a classes called User and Phonenumber with the following definitions:

Listing 1.18:

```
<?php

// file User.php
class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 20);
    }
    public function setUp()
    {
        $this->hasMany('Phonenumber', array('local' => 'id',
                                            'foreign' => 'user_id'));
    }
}
// file Phonenumber.php
class Phonenumber extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('phonenumber', 'string', 20);
        $this->hasColumn('user_id', 'integer');
```

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

?>
```

Now lets say these classes are in directory 'models/'. We can make Doctrine to iterate through this directory and
attach these classes into your database structure with the following script:

Listing 1.19:

```
<?php

require_once('path-to-doctrine/lib/Doctrine.php');

spl_autoload_register(array('Doctrine', 'autoload'));

//in order to export we need a database connection
Doctrine_Manager::connection('mysql://user:pass@localhost/test');

Doctrine::createTablesFromModels('models');

?>
```

This would execute the following queries on mysql.

Listing 1.20:

```
CREATE TABLE user (id BIGINT AUTO_INCREMENT, name VARCHAR(20), PRIMARY KEY(id),
    INDEX(id));
CREATE TABLE phonenumber (id INT AUTO_INCREMENT, phonenumber VARCHAR(20),
    user_id BIGINT, PRIMARY KEY(id), INDEX(user_id));
ALTER TABLE phonenumber ADD CONSTRAINT FOREIGN KEY (user_id) REFERENCES user(id)
     ON DELETE CASCADE;
```

Pay attention to the following things:

1. The autoincrement primary key columns are auto-added since we didn't specify any primary key columns

2. Doctrine auto-adds indexes to the referenced relation columns (this is needed in mysql)

1.6.2 Getting export queries

There might be situations where you don't want to execute the export queries immediately rather you want to get the
query strings and maybe attach them to a build.sql file. This can be easily achieved as follows:

Listing 1.21:

```
<?php

require_once('path-to-doctrine/lib/Doctrine.php');

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

```
Doctrine_Manager::connection('mgsql://user:pass@localhost/test');

$queries = Doctrine::generateSqlFromModels('models');

echo $queries;

?>
```

Consider the same situation and you want to get the string of sql queries needed to perform the exporting. It can be
achieved with Doctrine::generateSqlFromModels().

1.6.3 Export options

Listing 1.22:

```
<?php

// export everything, table definitions and constraints
$manager = Doctrine_Manager::getInstance();

$manager->setAttribute(Doctrine::ATTR_EXPORT, Doctrine::EXPORT_ALL);

// export classes without constraints

$manager->setAttribute(Doctrine::ATTR_EXPORT, Doctrine::EXPORT_TABLES ^
                                              Doctrine::EXPORT_CONSTRAINTS);

// turn off exporting

$manager->setAttribute(Doctrine::ATTR_EXPORT, Doctrine::EXPORT_NONE);

$sql = Doctrine::generateSqlFromModels();

?>
```

1.7 Generating models

Doctrine offers the ability to generate models from existing databases, or from YAML schema files. You already read about
generating models from an existing database in the 1.5
section.

Here is a simple example of how to generate your models from YAML schema files.

Create a schema_files/user.yml and place the following yaml in the file

Listing 1.23:

```
---
User:
  columns:
    username: string(255)
    password: string(255)
```

Now we can use a little script to generate the Doctrine_Record definition.

Listing 1.24:

```
<?php
```

```
require_once('/path/to/Doctrine.php');
spl_autoload_register(array('Doctrine', 'autoload'));

Doctrine::generateModelsFromYaml('/path/to/schema_files/', '/path/to/generate/
    models');

?>
```

Now you will have models/User.php and models/generated/BaseUser.php. User.php is for you to add custom code to, it is
only generated once, and BaseUser.php is regenerated each time you call generateModels-FromYaml()

1.8 Auto loading 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::autoload() we registered earlier with spl_autoload_register(array('Doctrine', 'autoload')).de Below are some examples
using the both types of model loading.

1.8.1 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

Here is an example of a basic Doctrine implementation using conservative model loading.

Listing 1.25:

```
<?php

//require the base Doctrine class
require_once('path-to-doctrine/lib/Doctrine.php');

//register the autoloader
spl_autoload_register(array('Doctrine', 'autoload'));

Doctrine_Manager::getInstance()->setAttribute('model_loading', 'conservative');
Doctrine::loadModels('/path/to/models'); // This call will not require the found
     .php files

$user = new User(); // This will invoke Doctrine::autoload() to include the file
     so the User class is present.

?>
```

1.8.2 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.

Here is an example of a basic Doctrine implementation using aggressive model loading.

Listing 1.26:

```
<?php

//require the base Doctrine class
require_once('path-to-doctrine/lib/Doctrine.php');

//register the autoloader
spl_autoload_register(array('Doctrine', 'autoload'));

Doctrine_Manager::getInstance()->setAttribute('model_loading', 'aggressive'); //
    Thi
Doctrine::loadModels('/path/to/models'); // This call will not require the found
    .php files

$user = new User(); // This will invoke Doctrine::autoload() to include the file
    so the User class is present.

?>
```

1.9 Command line interface

The command line interface is a collection of the most commonly used tasks in Doctrine available from a command line. The
command line interface can be read about more in the 12.3 section.

1.10 My first project tutorial

1.10.1 Introduction

This is a tutorial & how-to on creating your first project using the fully featured PHP Doctrine ORM. This tutorial
uses the the ready to go Doctrine sandbox package. It requires a web server, PHP and PDO + Sqlite.

1.10.2 Download

To get started, first download the latest Doctrine sandbox package: http://www.phpdoctrine.org/download. Second, extract
the downloaded file and you should have a directory named Doctrine-x.x.x-Sandbox. Inside of that directory is a simple
example implementation of a Doctrine based web application.

1.10.3 Package Contents

The files/directory structure should look like the following

Listing 1.27:

```
$ cd Doctrine-0.11.0-Sandbox
$ ls
config.php    doctrine     index.php    migrations    schema
data         doctrine.php    lib         models
```

The sandbox does not require any configuration, it comes ready to use with a sqlite database. Below is a description of
each of the files/directories and what its purpose is.

- doctrine - Shell script for executing the command line interface. Run with ./doctrine to see a list of command or

./doctrine help to see a detailed list of the commands

- doctrine.php - Php script which implements the Doctrine command line interface which is included in the above doctrine

shell script

- index.php - Front web controller for your web application
- migrations - Folder for your migration classes
- schema - Folder for your schema files
- models - Folder for your model files
- lib - Folder for the Doctrine core library files

1.10.4 Running the CLI

If you execute the doctrine shell script from the command line it will output the following:

Listing 1.28:

```
$ ./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
```

1.10.5 Defining Schema

Below is a sample yaml schema file to get started. You can place the yaml file in schemas/schema.yml. The command
line interface looks for all *.yml files in the schemas folder.

Listing 1.29:

```
---
User:
  columns:
    id:
      primary: true
      autoincrement: true
      type: integer(4)
    username: string(255)
    password: string(255)
  relations:
    Groups:
      class: Group
      refClass: UserGroup
      foreignAlias: Users

Group:
  tableName: groups
  columns:
    id:
      primary: true
      autoincrement: true
      type: integer(4)
    name: string(255)

UserGroup:
  columns:
    user_id: integer(4)
    group_id: integer(4)
  relations:
    User:
      onDelete: CASCADE
    Group:
      onDelete: CASCADE
```

1.10.6 Test Data Fixtures

Below is a sample yaml data fixtures file. You can place this file in data/fixtures/data.yml. The command line interface looks for all *.yml files in the data/fixtures folder.

Listing 1.30:

```
---
User:
  zyne:
    username: zYne-
    password: changeme
    Groups: [founder, lead, documentation]
  jwage:
    username: jwage
    password: changeme
    Groups: [lead, documentation]

Group:
  founder:
    name: Founder
  lead:
    name: Lead
  documentation:
    name: Documentation
```

1.10.7 Building Everything

Now that you have written your schema files and data fixtures, you can now build everything and begin working with your
models . Run the command below and your models will be generated in the models folder.

Listing 1.31:

```
$ ./doctrine build-all-reload
build-all-reload - Are you sure you wish to drop your databases? (y/n)
y
build-all-reload - Successfully dropped database for connection "sandbox" at
    path "/Users/jwage/Sites/doctrine/branches/0.11/tools/sandbox/sandbox.db"
build-all-reload - Generated models successfully from YAML schema
build-all-reload - Successfully created database for connection "sandbox" at
    path "/Users/jwage/Sites/doctrine/branches/0.11/tools/sandbox/sandbox.db"
build-all-reload - Created tables successfully
build-all-reload - Data was successfully loaded
```

Take a peak in the models folder and you will see that the model classes were generated for you. Now you can begin coding
in your index.php to play with Doctrine itself. Inside index.php place some code like the following for a simple test.

1.10.8 Running Tests

Listing 1.32:

```
<?php

$query = new Doctrine_Query();
$query->from('User u, u.Groups g');

$users = $query->execute();

echo '<pre>';
print_r($users->toArray(true));

?>
```

The print_r() should output the following data. You will notice that this is the data that we populated by placing
the yaml file in the data/fixtures files. You can add more data to the fixtures and rerun the build-all-reload
command to reinitialize the database.

Listing 1.33:

```
Array
(
    [0] => Array
        (
            [id] => 1
            [username] => zYne-
            [password] => changeme
            [Groups] => Array
                (
                    [0] => Array
                        (
                            [id] => 1
                            [name] => Founder
                        )

                    [1] => Array
                        (
                            [id] => 2
                            [name] => Lead
                        )

                    [2] => Array
                        (
                            [id] => 3
                            [name] => Documentation
                        )

                )

        )

    [1] => Array
        (
            [id] => 2
            [username] => jwage
            [password] => changeme
            [Groups] => Array
                (
                    [0] => Array
                        (
                            [id] => 2
                            [name] => Lead
                        )

                    [1] => Array
                        (
                            [id] => 3
                            [name] => Documentation
                        )

                )

        )

)
```

You can also issue DQL queries directly to your database by using the dql command line function. It is used like the
following.

Listing 1.34:

```
jwage:sandbox jwage$ ./doctrine dql "FROM User u, u.Groups g"
dql - executing: "FROM User u, u.Groups g" ()
dql - -
dql -   id: 1
dql -   username: zYne-
dql -   password: changeme
dql -   Groups:
dql -     -
dql -       id: 1
dql -       name: Founder
dql -     -
dql -       id: 2
dql -       name: Lead
dql -     -
dql -       id: 3
dql -       name: Documentation
dql - -
dql -   id: 2
dql -   username: jwage
dql -   password: changeme
dql -   Groups:
dql -     -
dql -       id: 2
dql -       name: Lead
dql -     -
dql -       id: 3
dql -       name: Documentation
```

1.10.9 User CRUD

Now we can demonstrate how to implement Doctrine in to a super simple module for managing users and passwords. Place
the following code in your index.php and pull it up in your browser. You will see the simple application.

Listing 1.35:

```
<?php

require_once('config.php');

Doctrine::loadModels('models');

$module = isset($_REQUEST['module']) ? $_REQUEST['module']:'users';
$action = isset($_REQUEST['action']) ? $_REQUEST['action']:'list';

if ($module == 'users') {
    $userId = isset($_REQUEST['id']) && $_REQUEST['id'] > 0 ? $_REQUEST['id']:
        null;
    $userTable = Doctrine::getTable('User');

    if ($userId === null) {
        $user = new User();
    } else {
        $user = $userTable->find($userId);
    }

    switch ($action) {
```

```
        case 'edit':
        case 'add':
            echo '<form action="index.php?module=users&action=save" method="POST
                ">
                  <fieldset>
                    <legend>User</legend>
                    <input type="hidden" name="id" value="' . $user->id . '" />
                    <label for="username">Username</label> <input type="text"
                        name="user[username]" value="' . $user->username . '" />
                    <label for="password">Password</label> <input type="text"
                        name="user[password]" value="' . $user->password . '" />
                    <input type="submit" name="save" value="Save" />
                  </fieldset
                  </form>';
            break;
        case 'save':
            $user->merge($_REQUEST['user']);
            $user->save();

            header('location: index.php?module=users&action=edit&id=' . $user->
                id);
            break;
        case 'delete':
            $user->delete();

            header('location: index.php?module=users&action=list');
            break;
        default:
            $query = new Doctrine_Query();
            $query->from('User u')
                  ->orderby('u.username');

            $users = $query->execute();

            echo '<ul>';
            foreach ($users as $user) {
                echo '<li><a href="index.php?module=users&action=edit&id=' .
                    $user->id . '">' . $user->username . '</a>   <a href="
                    index.php?module=users&action=delete&id=' . $user->id . '">[
                    X]</a></li>';
            }
            echo '</ul>';
    }

    echo '<ul>
            <li><a href="index.php?module=users&action=add">Add</a></li>
            <li><a href="index.php?module=users&action=list">List</a></li>
          </ul>';
} else {
    throw new Exception('Invalid module');
}

?>
```

Chapter 2

Connection management

2.1 DSN, the Data Source Name

2.2 Opening a new connection

Opening a new database connection in Doctrine is very easy. If you wish to use PDO (www.php.net/PDO) you can just
initalize a new PDO object:

Listing 2.1:

```
<?php

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

try {
    $dbh = new PDO($dsn, $user, $password);
    $conn = Doctrine_Manager::connection($dbh);
} catch (PDOException $e) {
    echo 'Connection failed: ' . $e->getMessage();
}

?>
```

Note: 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.

Listing 2.2:

```
<?php

$conn->setOption('username', 'username');
$conn->setOption('password', 'password');

?>
```

2.3 Lazy-connecting to database

Lazy-connecting to database can save a lot of resources. There might be many pages 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.

Listing 2.3:

```
<?php

// initalize a new Doctrine_Connection
$conn = Doctrine_Manager::connection('mysql://username:password@localhost/test')
    ;
// !! no actual database connection yet !!

// connects database and performs a query
$users = Doctrine_Query::create()
           ->from('User u')
           ->execute();

?>
```

2.4 Managing connections

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. The following example uses `openConnection()` second argument as
an optional connection alias.

Listing 2.4:

```
<?php

// Doctrine_Manager controls all the connections

$manager = Doctrine_Manager::getInstance();

// open first connection

$conn = $manager->openConnection('mysql://username:password@localhost/test', '
    connection 1');

?>
```

For convenience `Doctrine_Manager` provides static method `connection()` which opens new connection when arguments
are given to it and returns the current connection when no arguments have been speficied.

Listing 2.5:

```
<?php

// open first connection
```

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

$conn2 = Doctrine_Manager::connection();

// $conn2 == $conn

?>
```

The current connection is the lastly opened connection.

Listing 2.6:

```
<?php

// open second connection

$conn2 = $manager->openConnection('mysql://username2:password2@localhost/test2',
     'connection 2');

$manager->getCurrentConnection(); // $conn2

?>
```

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

Listing 2.7:

```
<?php

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

$manager->getCurrentConnection(); // $conn

?>
```

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

Listing 2.8:

```
<?php

// iterating through connections

foreach($manager as $conn) {

}

?>
```

2.5 Connection-component binding

Doctrine allows you to bind connections to components (= your ActiveRecord classes). This means everytime a component
issues a query or data is being fetched from the table the component is pointing at Doctrine will use the bound
connection.

Listing 2.9:

```
<?php

$conn = $manager->openConnection('mysql://username:password@localhost/dbname', '
    connection 1');

$conn2 = $manager->openConnection('mysql://username2:password2@localhost/dbname2
    ', 'connection 2');

$manager->bindComponent('User', 'connection 1');

$manager->bindComponent('Group', 'connection 2');

$q = Doctrine_Query::create();

// Doctrine uses 'connection 1' for fetching here
$users = $q->from('User u')->where('u.id IN (1,2,3)')->execute();

// Doctrine uses 'connection 2' for fetching here
$groups = $q->from('Group g')->where('g.id IN (1,2,3)')->execute();

?>
```

Chapter 3

Basic schema mapping

3.1 Introduction

This chapter and its subchapters tell you how to do basic schema mappings with Doctrine. After you've come in terms with
the concepts of this chapter you'll know how to:

1. Define columns for your record classes
2. Define table options
3. Define indexes
4. Define basic constraints and validators for columns

All column mappings within Doctrine are being done via the hasColumn() method of the Doctrine_Record. The hasColumn takes
4 arguments:

1. **column name** String that specifies the column name and optional alias. This is needed for all columns. If you want

to specify an alias for the column name you'll need to use the format '[columnName] as [columnAlias]'

1. **column type** String that specifies the column type. See the column types section.

1. **column length** 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.

1. **column constraints and validators** An array that specifies the list of constraints and validators applied to given

column.

Note that validators / column constraints and the column length fields are optional. The length may be omitted by using
null for the length argument, allowing doctrine to use a default length and permitting a fourth

argument for validation
or column constraints.

Lets take our first example. The following definition defines a class called Email which refers to a table called 'emails'.
The Email class has two columns id (an auto-incremented primary key column) and a string column called address.

Notice how we add two validators / constraints for the address column (notblank and email).
The notblank validator assures
that the address column isn't blank (so it must not contain space-characters only) whereas the email validator ensures that
the address is a valid email address.

Listing 3.1:

```php
<?php

class Email extends Doctrine_Record
{
    public function setTableDefinition()
    {
        // setting custom table name:
        $this->setTableName('emails');

        $this->hasColumn('address',         // name of the column
                         'string',          // column type
                         '200',             // column length
                         array('notblank' => true,
                               'email'    => true  // validators / constraints
                               )
                         );
    }
}

?>
```

Here is the same model specified as a YAML schema file

Listing 3.2:

```yaml
---
Email:
  tableName: emails
  columns:
    address:
      type: string(200)
      notblank: true
      email: true
```

Now lets create an export script for this class:

Listing 3.3:

```php
<?php

require_once('Email.php');
require_once('path-to-Doctrine/Doctrine.php');

require_once('path-to-doctrine/lib/Doctrine.php');

spl_autoload_register(array('Doctrine', 'autoload'));

// in order to export we need a database connection
$manager = Doctrine_Manager::getInstance();
```

```
$conn    = $manager->openConnection('mysql://user:pass@localhost/test');

$conn->export->exportClasses(array('Email'));

?>
```

The script would execute the following sql (we are using Mysql here as the database backend):

Listing 3.4:

```
CREATE TABLE emails (id BIGINT AUTO_INCREMENT, address VARCHAR(200), PRIMARY KEY
    (id)) ENGINE = INNODB;
```

3.2 Table and class naming

Doctrine automatically creates table names from the record class names. For this reason, it is recommended to name your
record classes using the following rules:

- Use `CamelCase` naming
- Underscores are allowed
- The first letter must be capitalized
- The class name cannot be one of the following (these keywords are reserved in DQL API):
- `ALL`, `AND`, `ANY`, `AS`, `ASC`, `AVG`, `BETWEEN`, `BIT_LENGTH`, `BY`, `CHARACTER_LENGTH`, `CHAR_LENGTH`, `COUNT`, `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `DELETE`, `DESC`, `DISTINCT`, `EMPTY`, `EXISTS`, `FALSE`, `FETCH`, `FROM`, `GROUP`, `HAVING`, `IN`, `INDEXBY`, `INNER`, `IS`, `JOIN`, `LEFT`, `LIKE`, `LOWER`, `MAX`, `MEMBER`, `MIN`, `MOD`, `NEW`, `NOT`, `NULL`, `OBJECT`, `OF`, `OR`, `ORDER`, `OUTER`, `POSITION`, `SELECT`, `SOME`, `SUM`, `TRIM`, `TRUE`, `UNKNOWN`, `UPDATE`, `UPPER` and `WHERE`.

Example: `My_PerfectClass`

If you need to use a different naming schema, you can override this using the `setTableName()` method in the
`setTableDefinition()` method.

3.3 Table options

Doctrine offers various table options. All table options can be set via `Doctrine_Record::option($optionNam $value)`.

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

Listing 3.5:

```
<?php

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

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

```
    }
}

?>
```

Listing 3.6:

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

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

Listing 3.7:

```
<?php

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

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

?>
```

Listing 3.8:

```
---
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:

Listing 3.9:

```
<?php

Doctrine_Manager::connection($name)->setCharset("utf8");

?>
```

Doctrine offers the ability to turn off foreign key constraints for specific Models.

Listing 3.10:

```
<?php

class MyCustomOptionRecord extends Doctrine_Record
{
    public function setTableDefinition()
    {
```

```
        $this->hasColumn('name', 'string');

        $this->setAttribute(Doctrine::ATTR_EXPORT, Doctrine::EXPORT_ALL ^
            Doctrine::EXPORT_CONSTRAINTS);
    }
}

?>
```

Listing 3.11:

```
---
MyCustomOptionRecord:
  columns:
    name: string
  attributes:
    export: [ all, constraints ]
```

3.4 Columns

3.4.1 Column naming

One problem with database compatibility is that many databases differ in their behaviour 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.

3.4.2 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.

Listing 3.12:

```
<?php

class Book extends Doctrine_Record
{
```

```
    public function setTableDefinition()
    {
        $this->hasColumn('bookTitle as title', 'string');
    }
}

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

?>
```

Listing 3.13:

```
---
Book:
  columns:
    name:
      name: bookTitle as title
      type: string
```

3.4.3 Default values

Doctrine supports default values for all data types. When default value is attached to a record column this means two of
things. First this value is attached to every newly created Record.

Listing 3.14:

```
<?php

class User extends Doctrine_record
{
    public function setTableDefinition(){
        $this->hasColumn('name', 'string', 50, array('default' => 'default name'
            ));
    }
}

$user = new User();
print $user->name; // default name

?>
```

Listing 3.15:

```
---
User:
  columns:
    name:
      type: string(50)
      default: default name
```

Also when exporting record class to database DEFAULT *value* is attached to column definition statement.

3.4.4 Data types

3.4.4.1 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, it was defined a base set of data types that applications
may access 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.

3.4.4.2 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.
- zerofill modifiers (not supported by all drivers)
- 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)

Listing 3.16:

```
<?php

'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!

Listing 3.17:

```
<?php

'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.

3.4.4.3 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.

Listing 3.18:

```
<?php

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

?>
```

Listing 3.19:

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

3.4.4.4 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.

Listing 3.20:

```
<?php

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

?>
```

Listing 3.21:

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

3.4.4.5 Float

The float data type may store floating point decimal numbers. This data type is suitable for representing numbers within
a 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.

Listing 3.22:

```php
<?php

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

?>
```

Listing 3.23:

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

3.4.4.6 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.

Listing 3.24:

```php
<?php

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

?>
```

Listing 3.25:

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

3.4.4.7 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.

Listing 3.26:

```
<?php

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

?>
```

Listing 3.27:

```
---
Test:
  stringtest:
    type: string(255)
    fixed: true
```

3.4.4.8 Array

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

Listing 3.28:

```
<?php

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

?>
```

Listing 3.29:

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

3.4.4.9 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.

Listing 3.30:

```
<?php
```

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

?>
```

Listing 3.31:

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

3.4.4.10 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 files.

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"

Listing 3.32:

```
<?php

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

?>
```

Listing 3.33:

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

3.4.4.11 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 files.

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"

Listing 3.34:

```
<?php

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

?>
```

Listing 3.35:

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

3.4.4.12 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

Listing 3.36:

```
<?php

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

?>
```

Listing 3.37:

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

3.4.4.13 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.

Listing 3.38:

```
<?php

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

?>
```

Listing 3.39:

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

3.4.4.14 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.

Listing 3.40:

```
<?php

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

?>
```

Listing 3.41:

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

3.4.4.15 Enum

Doctrine has a unified enum type. Enum typed columns automatically convert the string values into index numbers and vice versa. The possible values for the column can be specified with Doctrine_Record::setEnumValues(columnName, array values) or can be specified on the column definition with hasColumn()

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

Listing 3.42:

```php
<?php

$conn->setAttribute('use_native_enum', true);

?>
```

Listing 3.43:

```php
<?php

class Test extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('enumtest', 'enum', 4,
                         array(
                             'values' => array(
                                         'php',
                                         'java',
                                         'python'
                                         )
                               )
        );
    }
}

?>
```

Listing 3.44:

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

3.4.4.16 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.

Listing 3.45:

```
<?php

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

?>
```

Listing 3.46:

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

3.4.4.17 Examples

Consider the following definition:

Listing 3.47:

```
<?php

class Example extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumns(array(
                        'id' => array(
                            'type'     => 'string',
                            'length'   => 32,
                            'fixed' => true,
                        ),
                        'someint' => array(
                            'type'     => 'integer',
                            'length'   => 10,
                            'unsigned' => true,
                        ),
                        'sometext' => array(
                            'type'   => 'string',
                            'length' => 12,
                        ),
                        'somedate' => array(
                            'type' => 'date',
                        ),
                        'sometimestamp' => array(
                            'type' => 'timestamp',
                        ),
                        'someboolean' => array(
                            'type' => 'boolean',
                        ),
                        'somedecimal' => array(
                            'type' => 'decimal',
                        ),
                        'somefloat' => array(
                            'type' => 'float',
                        ),
                        'sometime' => array(
                            'type'    => 'time',
```

```
                'default' => '12:34:05',
                'notnull' => true,
            ),
            'someclob' => array(
                'type' => 'clob',
            ),
            'someblob' => array(
                'type' => 'blob',
            ));
}

?>
```

Listing 3.48:

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

The above example will create a database table as such in Pgsql:

Column	Type	Not Null	Default	comment	
id	character(32)				
somename	character	varying(12)			
somedate	date				
sometimestamp	timestamp without time zone				
someboolean	boolean				
somedecimal	numeric(18,2)				
somefloat	double precision				
sometime	time without time zone	NOT NULL	'12:34:05'		
someclob	text				
someblob	bytea				

And the following table in Mysql:

Field	Type	Collation	Attributes	Null	Default	comment
id	char(32)			YES		
somename	varchar(12)	latin1_swedish_ci		YES		
somedate	date			YES		

sometimestamp	timestamp without time zone			YES		
someboolean	tinyint(1)			YES		
somedecimal	decimal(18,2)			YES		
somefloat	double			YES		
sometime	time			NO	12:34:05	
someclob	longtext	latin1_swedish_ci		YES		
someblob	longblob		binary	YES		

3.5 Constraints and validators

3.5.1 Introduction

From PostgreSQL Documentation[1]*:*

> 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.
notnull		Checks if value is not null.
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.

[1] http://www.postgresql.org/docs/8.2/static/ddl-constraints.html

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

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.

Listing 3.49:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 255, array('minlength' => 12));
    }
}

?>
```

3.5.2 Notnull

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.

Listing 3.50:

```
<?php

class User extends Doctrine_Record
{
```

```
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 200, array('notnull' => true,
                                                      'primary' => true));
    }
}

?>
```

Listing 3.51:

```
---
User:
  columns:
    name:
      type: string(255)
      notnull: true
      primary: true
```

When this class gets exported to database the following SQL statement would get executed (in MySQL):

Listing 3.52:

```
CREATE TABLE user (name VARCHAR(200) NOT NULL, PRIMARY KEY(name))
```

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.

3.5.3 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 `name`.

Listing 3.53:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 200, array('unique' => true));
    }
```

```
}

?>
```

Listing 3.54:

```
---
User:
  columns:
    name:
      type: string(255)
      unique: true
```

Note: You should only use unique constraints for other than primary key columns. Primary key columns are always unique.

3.5.4 Check

Some of the Doctrine validators also act as database level check constraints. When a record with these validators is
exported additional CHECK constraints are being added to CREATE TABLE statement.

Consider the following example which uses 'min' validator:

Listing 3.55:

```
<?php

class Product extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('id', 'integer', 4, 'primary');
        $this->hasColumn('price', 'decimal', 18, array('min' => 0));
    }
}

?>
```

Listing 3.56:

```
---
Product:
  columns:
    id:
      type: integer(4)
      primary: true
    price:
      type: decimal(18)
      min: 0
```

When exported the given class definition would execute the following statement (in pgsql):

Listing 3.57:

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

So Doctrine optionally ensures even at the database level that the price of any product cannot be below zero.

You can also set the maximum value of a column by using the 'max' validator. This also creates the equivalent CHECK
constraint.

Listing 3.58:

```
<?php

class Product extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('id', 'integer', 4, array('primary' => true));
        $this->hasColumn('price', 'decimal', 18, array('min' => 0, 'max' =>
            1000000));
    }
}

?>
```

Listing 3.59:

```
---
Product:
  id:
    type: integer(4)
    primary: true
  price:
    type: decimal(18)
    min: 0
    max: 1000000
```

Generates (in pgsql):

Listing 3.60:

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

Lastly 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.

Listing 3.61:

```
<?php

class Product extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('id', 'integer', 4, array('primary' => true));
        $this->hasColumn('price', 'decimal', 18, array('min' => 0, 'max' =>
            1000000));
        $this->hasColumn('discounted_price', 'decimal', 18, array('min' => 0, '
            max' => 1000000));

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

```
}

?>
```

Listing 3.62:

```
---
Product:
  columns:
    id:
      type: integer(4)
      primary: true
    price:
      type: decimal(18)
      min: 18
      max: 1000000
    discounted_price:
      type: decimal(18)
      min: 0
      max: 1000000
  checks:
    check_1: price > discounted_price
```

Generates (in pgsql):

Listing 3.63:

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

> NOTE: some databases don't support CHECK constraints. When this is the case Doctrine simple 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.

3.6 Record identifiers

3.6.1 Introduction

Doctrine supports many kind of identifiers. For most cases it is recommended not to specify any primary keys
(Doctrine will then use field name `id` as an autoincremented primary key). When using table creation Doctrine is
smart enough to emulate the autoincrementation with sequences and triggers on databases that doesn't support it natively.

3.6.2 Natural

Natural identifier is a property or combination of properties that is unique and non-null. The use of natural identifiers
is encouraged.

Listing 3.64:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 200, array('primary' => true));
    }
}

?>
```

Listing 3.65:

```
---
User:
  columns:
    name:
      type: string(255)
      primary: true
```

3.6.3 Autoincremented

Autoincrement primary key is the most basic identifier and its usage is strongly encouraged. Sometimes you may want to use
some other name than `id` for your autoinc primary key. It can be specified as follows:

Listing 3.66:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('uid', 'integer', 20, array('primary' => true, '
            autoincrement' => true));
        $this->hasColumn('username', 'string');
    }
}

?>
```

Listing 3.67:

```
---
User:
  columns:
    uid:
      type: integer(20)
      primary: true
      autoincrement: true
    username: string
```

You should consider using autoincremented or sequential primary keys only when the record cannot be identified naturally
(in other words it doesn't have a natural identifier).

The following example shows why natural identifiers are more efficient.

Consider three classes Permission, Role and RolePermission. Roles having many permissions and vice versa (so their
relation is many-to-many). Now lets also assume that each role and permission are naturally identified by their names.

Now adding autoincremented primary keys to these classes would be simply stupid. It would require more data and it would
make the queries more inefficient. For example fetching all permissions for role 'Admin' would be done as follows
(when using autoinc pks):

Listing 3.68:

```
SELECT p.*
    FROM Permission p
        LEFT JOIN RolePermission rp ON rp.permission_id = p.id
        LEFT JOIN Role r ON rp.role_id = r.id
    WHERE r.name = 'Admin'
```

Now remember sql JOINS are always expensive and here we are using two of those. When using natural identifiers the query
would look like:

Listing 3.69:

```
SELECT p.*
    FROM Permission p
        LEFT JOIN RolePermission rp ON rp.permission_name = p.name
    WHERE rp.role_name = 'Admin'
```

Thats -1 JOIN !

3.6.4 Composite

Composite primary key can be used efficiently in association tables (tables that connect two components together). It is
not recommended to use composite primary keys in anywhere else as Doctrine does not support mapping relations on multiple
columns.

Due to this fact your doctrine-based system will scale better if it has autoincremented primary key even for association
tables.

Listing 3.70:

```
<?php

class UserGroup extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user_id', 'integer', 20, array('primary' => true));
        $this->hasColumn('group_id', 'integer', 20, array('primary' => true));
    }
```

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

?>
```

Listing 3.71:

```
---
UserGroup:
  columns:
    user_id:
      type: integer(20)
      primary: true
    group_id:
      type: integer(20)
      primary: true
  relations:
    User:
    Group:
```

3.6.5 Sequence

Doctrine supports sequences for generating record identifiers. Sequences are a way of offering unique IDs for data rows.
If you do most of your work with e.g. MySQL, think of sequences as another way of doing `AUTO_INCREMENT`.

Doctrine knows how to do sequence generation in the background so you don't have to worry about calling database specific
queries - Doctrine does it for you, all you need to do is define a column as a sequence column and optionally provide the
name of the sequence table and the id column name of the sequence table.

Consider the following record definition:

Listing 3.72:

```
<?php

class Book extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('id', 'integer', null, array('primary' => true, '
            sequence' => true));
        $this->hasColumn('name', 'string');
    }
}

?>
```

Listing 3.73:

```
---
Book:
  columns:
    id:
      type: integer
```

```
      primary: true
      sequence: true
    name: string
```

By default Doctrine uses the following format for sequence tables `[tablename]_seq`. If you wish to change this you can
use the following piece of code to change the formatting:

Listing 3.74:

```
<?php

$manager = Doctrine_Manager::getInstance();
$manager->setAttribute(Doctrine::ATTR_SEQNAME_FORMAT, '%s_my_seq');

?>
```

Doctrine uses column named id as the sequence generator column of the sequence table. If you wish to change this globally
(for all connections and all tables) you can use the following code:

Listing 3.75:

```
<?php

$manager = Doctrine_Manager::getInstance();
$manager->setAttribute(Doctrine::ATTR_SEQCOL_NAME, 'my_seq_column');

?>
```

In the following example we do not wish to change global configuration we just want to make the `id` column to use
sequence table called `book_sequence`. It can be done as follows:

Listing 3.76:

```
<?php

class Book extends Doctrine_Record {
    public function setTableDefinition()
    {
        $this->hasColumn('id', 'integer', null, array('primary', 'sequence' => '
            book_sequence'));
        $this->hasColumn('name', 'string');
    }
}

?>
```

Listing 3.77:

```
---
Book:
  columns:
    id:
      type: integer
      primary: true
      sequence: book_sequence
    name: string
```

Here we take the preceding example a little further: we want to have a custom sequence column. Here it goes:

Listing 3.78:

```
<?php

class Book extends Doctrine_Record {
    public function setTableDefinition()
    {
        $this->hasColumn('id', 'integer', null, array('primary', 'sequence' =>
            array('book_sequence', 'sequence')));
        $this->hasColumn('name', 'string');
    }
}

?>
```

Listing 3.79:

```
---
Book:
  columns:
    id:
      type: integer
      primary: true
      sequence: [book_sequence, sequence]
    name: string
```

3.7 Indexes

3.7.1 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.

3.7.2 Adding indexes

You can add indexes by simple calling `Doctrine_Record::index('indexName', $definition)` where `$definition` is the
definition array.

An example of adding a simple index to field called `name`:

Listing 3.80:

```
<?php

class IndexTest extends Doctrine_Record
{
    public function setTableDefinition()
    {
```

```
        $this->hasColumn('name', 'string');

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

?>
```

Listing 3.81:

```
---
IndexTest:
  columns:
    name: string
  indexes:
    myindex:
      fields: name
```

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

Listing 3.82:

```
<?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')));
    }
}

?>
```

Listing 3.83:

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

An example of adding a multiple indexes on same table:

Listing 3.84:

```
<?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')));
    }
}
```

```
?>
```

Listing 3.85:

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

3.7.3 Index options

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

Listing 3.86:

```
sorting     => string('ASC' / 'DESC')
        what kind of sorting does the index use (ascending / descending)

length      => integer
        index length (only some drivers support this)

primary     => boolean(true / false)
        whether or not the index is primary index

type        => string('unique',         -- supported by most drivers
                      'fulltext',       -- only availible on Mysql driver
                      'gist',           -- only availible on Pgsql driver
                      'gin')            -- only availible on Pgsql driver
```

Listing 3.87:

```
<?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' =>
                                  array('sorting' => 'ASC',
                                        'length'  => 10),
                                  'code'),
                      'type' => 'unique',
                      ));
    }
}

?>
```

Listing 3.88:

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

3.7.4 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'.

Listing 3.89:

```
<?php

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

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

?>
```

Listing 3.90:

```
---
Article:
  columns:
    name: string
    content: string
  indexes:
    content:
      fields: [content]
      type: fulltext
```

Chapter 4

Relations

4.1 Introduction

In Doctrine all record relations are being set with `hasMany`, `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 `hasMany` and `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:

- **local**, the local field of the relation. Local field is the linked field in the defining class.
- **foreign**, the foreign field of the relation. Foreign field is the linked field in the linked class.
- **refClass**, the name of the association class. This is only needed for many-to-many associations.
- **owningSide**, (optional) 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**, (optional) the onDelete integrity action that is applied on the foreign key constraint when the tables are created by Doctrine.
- **onUpdate**, (optional) the onUpdate integrity action that is applied on the foreign key constraint when the tables are created by Doctrine.

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.

Listing 4.1:

```
<?php

class Forum_Board extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 100);
        $this->hasColumn('description', 'string', 5000);
    }
    public function setUp()
    {
        // notice the 'as' keyword here
        $this->hasMany('Forum_Thread as Threads', array('local' => 'id',
                                                         'foreign' => 'board_id')
                                                           );
    }
}

?>
```

Listing 4.2:

```
---
Forum_Board:
  columns:
    name: string(100)
    description: string(5000)
  relations:
    Threads:
      class: Forum_Thread
      local: id
      foreign: board_id
      type: many
```

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.

Listing 4.3:

```
<?php

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

```
        // notice the 'as' keyword here
        $this->hasOne('Forum_Board as Board', array('local' => 'board_id',
                                                     'foreign' => 'id'));
    }
}

?>
```

Listing 4.4:

```
---
Forum_Thread:
  columns:
    board_id: integer(10)
    title: string(200)
    updated: integer(10)
    closed: integer(1)
  relations:
    Board:
      class: Forum_Board
      local: board_id
      foreign: id
```

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

Listing 4.5:

```
<?php

// first create a board
$board = new Forum_Board();
$board->name = 'Some board';

// lets create a new thread
$board->Thread[0]->title = 'new thread';

// save the changes
$board->save();

?>
```

4.2 Foreign key associations

4.2.1 One-To-One

One-to-one relations are propably 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'.

Listing 4.6:

```
<?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'));
    }
}

?>
```

Listing 4.7:

```
---
Email:
  columns:
    user_id: integer
    address: string(150)
  relations:
    User:
      local: user_id
      foreign: id
```

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

Listing 4.8:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string',50);
        $this->hasColumn('loginname', 'string',20);
        $this->hasColumn('password', 'string',16);
    }
    public function setUp()
    {
        $this->hasOne('Email', array('local' => 'id', 'foreign' => 'user_id'));
    }
}

?>
```

Listing 4.9:

```
---
User:
  columns:
    name: string(50)
    loginname: string(20)
    password: string(16)
  relations:
    Email:
      local: id
      foreign: user_id
```

4.2.2 One-to-Many, 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.

Listing 4.10:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 50);
        $this->hasColumn('loginname', 'string', 20);
        $this->hasColumn('password', 'string', 16);
    }

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

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

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

?>
```

Listing 4.11:

```
---
User:
  columns:
    name: string(50)
    loginname: string(20)
    password: string(16)
  relations:
    Phonenumber:
      local: id
      foreign: user_id

Phonenumber:
  columns:
    phonenumber: string(50)
    user_id: integer
```

```
  relations:
    User:
```

4.2.3 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.

However this mainly just serves as an example how the self-referencing can be done. The definition above is rarely a
good way of expressing hierarchical data, hence you should take a look at chapter 8
for how to set up efficient parent/child relations.

Listing 4.12:

```
<?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 Subtask', array('local' => 'id', 'foreign' => '
            parent_id'));
    }
}

?>
```

Listing 4.13:

```
---
Task:
  columns:
    name: string(100)
    parent_id: integer
  relations:
    Parent:
      class: Task
      local: parent_id
    Subtask:
      class: Task
      local: id
      foreign: parent_id
```

4.3 Join table associations

4.3.1 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 behaviour 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`.

Listing 4.14:

```
<?php

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

    public function setUp()
    {
        $this->hasMany('Group', array('local' => 'user_id',          // <- these
            are the column names
                                       'foreign' => 'group_id',      // <- in the
                                            association table
                                       'refClass' => 'GroupUser'));  // <- the
                                           following line is needed in many-to-
                                           many relations!
    }
}

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

    public function setUp()
    {
        $this->hasMany('User', array('local' => 'group_id',        // <- these
            are the column names
                                      'foreign' => 'user_id',      // <- in the
                                          association table
                                      'refClass' => 'GroupUser')); // <- the
                                          following line is needed in many-to-
                                          many relations!

        // group is reserved keyword so either do this or enable
            ATTR_QUOTE_IDENTIFIERS
        $this->setTableName('my_group');
    }
}
```

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

?>
```

Notice how the relationship is bi-directional. Both User hasMany Group and Group hasMany User. This is required by Doctrine in order for Many2Many relationships to fully work.

Now lets play around with the new models and create a user and assign it some groups.

Listing 4.15:

```
<?php

$user = new User();

// add two groups
$user->Group[0]->name = 'First Group';

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

// save changes into database
$user->save();

// deleting the associations between user and groups it belongs to

$user->Groupuser->delete();

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

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

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

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

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

?>
```

Listing 4.16:

```
---
User:
  columns:
    name: string(30)
  relations:
    Group:
      refClass: GroupUser
      local: user_id
      foreign: group_id

Group:
  tableName: my_group
  columns:
    name: string(30)
```

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

4.3.2 Self-referencing (Nest relations)

4.3.2.1 Non-equal nest relations

Listing 4.17:

```
<?php

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

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

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

    }
}

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));
    }
}

?>
```

Listing 4.18:

```
---
User:
  columns:
    name: string(30)
  relations:
    Parents:
      class: User
      refClass: UserReference
      local: parent_id
      foreign: child_id
```

```
    Children:
      class: User
      refClass: UserReference
      local: child_id
      foreign: parent_id

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

4.3.2.2 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.

Listing 4.19:

```
<?php

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

    public function setUp()
    {
        $this->hasMany('User as Friend', array('local'    => 'user1',
                                               'foreign'  => 'user2',
                                               'refClass' => 'UserReference',
                                               'equal'    => true,
                                                ));
    }
}

class UserReference extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('user1', 'integer', null, array('primary' => true));
        $this->hasColumn('user2', 'integer', null, array('primary' => true));
    }
}

?>
```

Listing 4.20:

```
---
User:
  columns:
    name: string(30)
  relations:
    Friend:
      class: User
      refClass: UserReference
      local: user1
      foreign: user2
      equal: true

UserReference:
  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.

Listing 4.21:

```php
<?php

$daniels = new User();
$daniels->name = 'Jack Daniels';

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

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

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

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

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

$conn->flush();

?>
```

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

4.4 Inheritance

Doctrine supports 3 types of inheritance strategies which can be mixed together.

4.4.1 Simple inheritance

Simple inheritance is the simpliest inheritance. In simple inheritance all the child classes share the same columns as the parent.

Listing 4.22:

```
<?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);
    }
}

class User extends Entity
{ }

class Group extends Entity
{ }

?>
```

Listing 4.23:

```
---
Entity:
  columns:
    name: string(30)
    username: string(20)
    password: string(16)
    created: integer(11)

User:
  inheritance:
    extends: Entity
    type: simple

Group:
  inheritance:
    extends: Entity
    type: simple
```

4.4.2 Concrete inheritance

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 above.

Listing 4.24:

```
<?php

class TextItem extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('topic', 'string', 100);
    }
}
```

```
class Comment extends TextItem
{
    public function setTableDefinition()
    {
        parent::setTableDefinition();

        $this->hasColumn('content', 'string', 300);
    }
}

?>
```

Listing 4.25:

```
---
TextItem:
  columns:
    topic: string(100)

Comment:
  inheritance:
    extends: TextItem
    type: concrete
  columns:
    content: string(300)
```

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.

Listing 4.26:

```
<?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);
    }
}

class User extends Entity
{
    public function setTableDefinition()
    {
        // the following method call is needed in
        // one-table-one-class inheritance
        parent::setTableDefinition();
    }
}

class Group extends Entity
{
    public function setTableDefinition()
```

```
    {
        // the following method call is needed in
        // one-table-one-class inheritance
        parent::setTableDefinition();
    }
}

?>
```

Listing 4.27:

```
---
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
```

4.4.3 Column aggregation inheritance

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
call the `Doctrine_Table::setSubclasses()` method from the parent class.

Listing 4.28:

```
<?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);

        // this column is used for column
        // aggregation inheritance
        $this->hasColumn('type', 'integer', 11);
        $this->setSubclasses(array(
            'User'  => array('type' => 1),
            'Group' => array('type' => 2)
        ));
```

```
    }
}

class User extends Entity {}
class Group extends Entity {}

?>
```

Listing 4.29:

```
---
Entity:
  columns:
    name: string(30)
    username: string(20)
    password: string(16)
    created: integer(11)

User:
  inheritance:
    extends: Entity
    type: column_aggregation
    keyField: type
    keyValue: 1

Group:
  inheritance:
    extends: Entity
    type: column_aggregation
    keyField: type
    keyValue: 2
```

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.

Listing 4.30:

```
<?php

$user = new User();
$user->name = 'Bjarte S. Karlsen';
$user->username = 'meus';
$user->password = 'rat';
$user->save();

$group = new Group();
$group->name = 'Users';
$group->username = 'users';
$group->password = 'password';
$group->save();

$q = Doctrine_Query::create();
$user = $q->from('Entity')->where('id = ?')->fetchOne(array($user->id));
assert($user instanceOf User);

$q = Doctrine_Query::create();
$group = $q->from('Entity')->where('id = ?')->fetchOne(array($group->id));
assert($group instanceOf Group);

?>
```

4.5 Foreign key constraints

4.5.1 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:

Listing 4.31:

```
<?php

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

?>
```

Listing 4.32:

```
---
Product:
  columns:
    id:
      type: integer
      primary: true
    name:
      type: string
    price:
      type: decimal(18)
```

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:

Listing 4.33:

```
<?php

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

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

```
        // foreign key columns should *always* have indexes

        $this->index('product_id', array('fields' => 'product_id'));
    }
}

?>
```

Listing 4.34:

```
---
Order:
  columns:
    order_id:
      type: integer
      primary: true
    product_id: integer
    quantity: integer
  relations:
    Product:
```

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

Listing 4.35:

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

Now it is impossible to create orders with product_no entries that do not appear in the products 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.

4.5.2 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 everytime a users is being deleted its associated
phonenumbers will also be deleted.

Listing 4.36:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 50);
        $this->hasColumn('loginname', 'string', 20);
        $this->hasColumn('password', 'string', 16);
    }
    public function setUp()
    {
        $this->index('id', array('fields' => 'id'));

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

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

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

?>
```

Listing 4.37:

```
---
User:
  columns:
    name: string(50)
    loginname: string(20)
    password: string(16)
  indexes:
    id:
      fields: id
  relations:
```

```
    Phonenumber:
      local: id
      foreign: user_id

Phonenumber:
  columns:
    phonenumber: string(50)
    user_id: integer
  indexes:
    user_id_idx:
      fields: user_id
  relations:
    User:
      local: user_id
      foreign: id
      onDelete: CASCADE
```

4.6 Transitive Persistence

4.6.1 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.

4.6.1.1 Save cascades

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

4.6.1.2 Delete cascades

Starting with Doctrine 0.11, 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:

Listing 4.38:

```php
<?php

class User extends Doctrine_Record
{
    //...
    public function setUp()
    {
        $this->hasOne('Address as address', array(
                'local' => 'id',
                'foreign' => 'user_id',
                'cascade' => array('delete') // application-level delete cascade
                ));
    }
}

?>
```

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

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 Address. 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 asociated 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.

4.6.2 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:

Listing 4.39:

```
<?php

class Address extends Doctrine_Record
{
    //...
    public function setUp()
```

```
    {
        $this->hasOne('User as user', array(
                'local' => 'user_id', // this is the foreign key in the database
                'foreign' => 'id',
                'onDelete' => 'CASCADE' // database-level delete cascade,
                    applied on the foreign key
                ));
    }
}

?>
```

The 'onDelete' => 'CASCADE' 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 'onDelete' => 'CASCADE' and 'onUpdate' => 'CASCADE'. Both are specified on the side that owns the foreign key and applied to your database schema when Doctrine creates your tables.

Chapter 5

Schema Files

5.1 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.

5.2 Short Hand Syntax

Doctrine offers the ability to specify schema in a short hand 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 shorthand features.

Note: the detect_relations option will attempt to guess relationships based on column names. In the
example below Doctrine knows that User hasOne Contact and will automatically define the relationship.

Listing 5.1:

```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
```

5.3 Expanded Syntax

Here is the none short hand form of the above schema.

Listing 5.2:

```
---
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 and foreign key, the type
and alias of relationship on each end of the relationship.

5.4 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 inflects 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.

5.4.1 Detect Relations

Doctrine offers the ability to specify a detect_relations options. 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.

5.4.2 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 inflected 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 hasOne Contact and Contact hasOne User as UserModel. Normally it would have
automatically generated User hasOne Contact and Contact hasMany User. The foreignType and foreignAlias
keywords allow you to customize the foreign relationship.

Listing 5.3:

```
---
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.

Listing 5.4:

```
---
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 will be User hasOne Avatar and Avatar hasMany User.

5.4.3 One to One

Listing 5.5:

```
---
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)
```

5.4.4 One to Many

Listing 5.6:

```
---
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
```

5.4.5 Many to Many

Listing 5.7:

```
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 hasMany Groups, Group hasMany Users, GroupUser hasOne User
and GroupUser hasOne Group.

5.5 Features & Examples

5.5.1 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:

Listing 5.8:

```
<?php

Doctrine_Manager::connection('mysql://jwage:pass@localhost/connection1', '
    connection1');

?>
```

Now somewhere in your Doctrine bootstrapping of Doctrine oyu would bind the model to that connection

Listing 5.9:

```
<?php

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 instance.

Listing 5.10:

```
---
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)
```

5.5.2 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.

Listing 5.11:

```
---
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
```

5.5.3 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.

Listing 5.12:

```
---
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']
```

5.5.4 ActAs

You can attach behaviors to your models with the actAs option. You can specify

Listing 5.13:

```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)
# This is an optional syntax which will just use the the defaults which are
    specified below
# Since they are defaults it is not necessary to type it out
#  actAs: [Timestampable, Sluggable]
  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
```

5.5.5 Listeners

If you have a listener you'd like attached to a model, you can specify them directly in the yml as well.

Listing 5.14:

```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 as follows.

Listing 5.15:

```php
<?php

class BaseUser extends Doctrine_Record
{
   ...

   public setUp()
   {
       ...
       $this->addListener(new MyCustomListener());
   }
}
```

5.5.6 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.

Listing 5.16:

```
---
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
```

5.5.7 Indexes

Please see chapter 3 for more information about indexes and
their options.

schema.yml

Listing 5.17:

```
---
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.

Listing 5.18:

```
<?php

$this->index('name_index', array('fields' => array('first_name' => array('
    sorting'  => 'ASC',
                                                  'length'
                                                  =>
                                                  '10'
                                                  ,
                                                  'primary'
                                                  =>
                                                  true
                                                  ),
                                     'last_name' => array()),
                        'type' => 'unique'));

?>
```

5.5.8 Inheritance

Simple inheritance. Read about more about 4.4.1. Any columns or relations added to the children classes will be moved to the parent.

Listing 5.19:

```
---
Entity:
  columns:
    name: string(255)
    username: string(255)
    password: string(255)

User:
  inheritance:
    extends: Entity
    type: simple

Group:
  inheritance:
    extends: Entity
    type: simple
```

Concrete inheritance. Read about more about 4.4.2

Listing 5.20:

```
---
TextItem:
  columns:
    topic: string(255)

Comment:
  inheritance:
    extends: TextItem
    type: concrete
  columns:
    content: string(300)
```

Column aggregation inheritance. Read about more about 4.4.3.
Like simple inheritance, any columns or relationships added to the children will be automatically

removed and moved to
the parent.

Listing 5.21:

```
---
Entity:
  columns:
    name: string(255)
    type: string(255)          # Optional, will be automatically added when it is
         specified in the child class

User:
  inheritance:
    extends: Entity
    type: column_aggregation  # Optional, it is implied if a keyField or
        keyValue is present
    keyField: type             # Optional, will default to 'type' and add it to
        the parent class if type is column aggregation
    keyValue: User             # Optional, will default to the name of the child
        class if type is column aggregation
  columns:
    username: string(255)
    password: string(255)

Group:
  inheritance:
    extends: Entity
    type: column_aggregation
    keyField: type
    keyValue: Group
  columns:
    description: string(255)
```

5.5.9 Column Aliases

If you want the ability alias a column name as something other than the column name in the database
this is possible with the " as alias_name" string after the column name.

Example:

Listing 5.22:

```
---
User:
  columns:
    login:
      name: login as username
      type: string(255)
    password:
      type: string(255)
```

The above example would allow you to access the login column name from the alias "username".

5.5.10 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.

Listing 5.23:

```
---
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

5.5.11 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:

connection
attributes
templates
actAs
options
package
inheritance
detect_relations

Listing 5.24:

```
---
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.

5.6 Using Schema Files

Once you have defined your schema files you need some code to

Listing 5.25:

```
<?php

// The options are completely optional. Only use this if you need something
    beyond the default configuration for model generation
$options = array('packagesPrefix'        =>  'Package',             // What to
    prefix the middle package models with
                 'packagesPath'          =>  '',                    // this
                     defaults to the "#models_path#/packages"
                 '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
                 'baseClassesDirectory'  =>  'generated',           // Name of
                     the folder to generate the base class definitions in
                 'baseClassName'         =>  'Doctrine_Record',     // Name of
                     the base Doctrine_Record class
                 'suffix'                =>  '.php');               // Extension
                      for your generated models

// This code will generate the models for schema.yml at /path/to/generate/models
Doctrine::generateModelsFromYaml('/path/to/directory/with/yaml/schema/files', '/
    path/to/generate/models', $options);

?>
```

Chapter 6

Working with objects

6.1 Dealing with relations

6.1.1 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.

Listing 6.1:

```
<?php

$user->Email;

$user->get('Email');

$user['Email'];

?>
```

When accessing a one-to-one related record that doesn't exist, Doctrine automatically creates the object. So for example
the following code is possible:

Listing 6.2:

```
<?php

$user = new User();
$user->name = 'some user';

$user->Email->address = 'some@one.info';
// saves the user and the associated email
$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:

Listing 6.3:

```php
<?php

$user = new User();
$user->name = 'some user';

$user->Phonenumber[]->phonenumber = '123 123';
$user->Phonenumber[]->phonenumber = '456 123';
$user->Phonenumber[]->phonenumber = '123 777';

// saves the user and the associated phonenumbers
$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):

Listing 6.4:

```php
<?php

// We keep track of the new phone number identifiers

$phoneIds = array();

// Some phone numbers are created...

$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'];

// Some user is created...

$user = new User();
$user['name'] = 'Werner Mollentze';
$user->save();

// Let's link the phone numbers to the user, since the relation to Phonenumber
    exists for the User record...

$user->link('Phonenumber', $phoneIds);

?>
```

If a relation to the User record class is defined for the Phonenumber record class, you may even do this:

Listing 6.5:

```php
<?php
```

```
// Some user is created...

$user = new User();
$user['name'] = 'wernerm';
$user->save();

// Some phone numbers are created and linked to the User on-the-fly...
// This is possible if a relation to User exists for the Phonenumber record

$phone1 = new Phonenumber();
$phone1['phonenumber'] = '555 202 7890';
$phone1->save();

// Let's link this Phonenumber to our User...

$phone1->link('User', array($user['id']));

// We create another phone number...

$phone2 = new Phonenumber();
$phone2['phonenumber'] = '555 100 7890';
$phone2->save();

// Let's link this Phonenumber to our User too...

$phone2->link('User', array($user['id']));

?>
```

6.1.2 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 4 SQL SELECTs.

Listing 6.6:

```
<?php

$user = Doctrine::getTable('User')->find(5);

print $user->Email['address'];

print $user->Phonenumber[0]->phonenumber;

print $user->Group[0]->name;

?>
```

Much more efficient way of doing this is using DQL. The following example uses only one SQL query for the retrieval of
related components.

Listing 6.7:

```
<?php

$user = Doctrine_Query::create()
        ->from('User u')
        ->leftJoin('u.Email e')
        ->leftJoin('u.Phonenumber p')
```

```
        ->leftJoin('u.Group g')
        ->execute();

print $user->Email['address'];

print $user->Phonenumber[0]->phonenumber;

print $user->Group[0]->name;

?>
```

6.1.3 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.

Listing 6.8:

```
<?php

$user->Email['address'] = 'koskenkorva@drinkmore.info';

$user->Phonenumber[0]->phonenumber = '123123';

$user->save();

// saves the email and phonenumber

?>
```

6.1.4 Deleting related records

You can delete related records individually be calling `delete()` on a record or on a collection.

Listing 6.9:

```
<?php

$user->Email->delete();

$user->Phonenumber[3]->delete();

// deleting user and all related objects:

$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:

Listing 6.10:

```
<?php

$deleted = Doctrine_Query::create()
            ->delete()
            ->from('Phonenumber')
            ->addWhere('user_id = ?', array($userId))
            ->whereIn('id', $phonenumberIds);
            ->execute();
// print out the number of deleted phonenumbers
print $deleted;

?>
```

Sometimes you may not want to delete the phonenumber records but to simply unlink the relations by setting the foreing
key fields to null. This can ofcourse 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 3 Phonenumbers (with identifiers 1, 2 and 3). Now unlinking the Phonenumbers 1 and 3
can be achieved as easily as:

Listing 6.11:

```
<?php

$user->unlink('Phonenumber', array(1, 3));

$user->Phonenumber->count(); // 1

?>
```

6.1.5 Working with related records

6.1.5.1 Testing the existance of a relation

Listing 6.12:

```
<?php

$obj = new Model();
if(isset($obj->Relation)) { // returns false
        ...
}
$obj->Relation = new Relation();
if(isset($obj->Relation)) { // returns true
        ...
}

?>
```

6.2 Many-to-Many relations

Note: Doctrine requires that Many-to-Many relationships be bi-directional. For example: both User must have many Group and Group must have many User. This is required by Doctrine in order for Many-to-Many relationships to fully work.

6.2.1 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 Groups for it is:

Listing 6.13:

```
<?php

$user = new User();
$user->name = 'Some User';
$user->Group[0]->name = 'Some Group';
$user->Group[1]->name = '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:

Listing 6.14:

```
<?php

$gu = new GroupUser();
$gu->user_id = $userId;
$gu->group_id = $groupId;
$gu->save();

?>
```

6.2.2 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 trhough the Query API.

Listing 6.15:

```
<?php

$deleted = Doctrine_Query::create()
           ->delete()
           ->from('GroupUser')
           ->addWhere('user_id = 5')
           ->whereIn('group_id', $groupIds);
           ->execute();
// print out the deleted links
print $deleted;

?>
```

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.

Listing 6.16:

```
<?php

$user = Doctrine::getTable('User')->find(5);
$user->unlink('Group', array(0, 1));
$user->save();

// you can also unlink ALL relationships to Group
$user->unlink('Group');

?>
```

While the obvious and convinient way of deleting a link between User and Group would be the following, you still should
NOT do this:

Listing 6.17:

```
<?php

$user = Doctrine::getTable('User')->find(5);
$user->GroupUser
     ->remove(0)
     ->remove(1);
$user->save();

?>
```

This is due to a fact that $user->GroupUser loads all group links for given user. This can time-consuming task if user
belongs to many groups. Even if the user belongs to few groups this will still execute an unnecessary SELECT statement.

6.3 Fetching objects

Normally when you fetch data from database the following phases are executed:

1. Sending the query to database
2. 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 fecthing 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:

Listing 6.18:

```
<?php
```

```
$dbh->fetchAll('SELECT u.id, u.name, p.phonenumber FROM user u LEFT JOIN
    phonenumber p ON u.id = p.user_id');

?>
```

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:

Listing 6.19:

```
index  | u.id | u.name        | 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 2 phonenumbers, John Beer has one whereas John Smith has 3 phonenumbers. You may notice how clumsy
this result set is. Its hard to iterate over it as you would need some duplicate data checkings here and there.

Doctrine hydration removes all duplicated data. It also performs many other things such as:

1. Custom indexing of result set elements
2. Value casting and preparation
3. Value assignment listening
4. 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:

Listing 6.20:

```
<?php

$array = Doctrine_Query::create()
          ->select('u.id, u.name, p.phonenumber')
          ->from('User u')
          ->leftJoin('u.Phonenumber p')
          ->execute(array(), Doctrine::HYDRATE_ARRAY);

?>
```

The structure of this hydrated array would look like:

Listing 6.21:

```
array(0 => array('id' => 1,
                 'name' => 'Jack Daniels',
                 'Phonenumber' =>
                    array(0 => array('phonenumber' => '123 123'),
                          1 => array('phonenumber' => '456 456'))),
      1 => array('id' => 2,
```

```
                  'name' => 'John Beer',
                  'Phonenumber' =>
                     array(0 => array('phonenumber' => '111 111'))),
      2 => array('id' => 3,
                  'name' => 'John Smith',
                  'Phonenumber' =>
                     array(0 => array('phonenumber' => '111 111')),
                           2 => array('phonenumber' => '222 222'),
                           3 => array('phonenumber' => '333 333'))));
```

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:

1. Iterate over the results using *foreach*
2. Access individual elements using array access brackets
3. Get the number of elements using *count()* function
4. Check if given element exists using *isset()*
5. 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.

6.3.1 Sample Queries

All of the below queries were executed with the following schema and data fixtures:

Schema

Listing 6.22:

```
---
User:
  actAs:
    Timestampable:
    Sluggable:
      fields: [username]
  columns:
    email_address: string(255)
    username: string(255)
    password: string(255)
  relations:
    Groups:
      class: Group
      refClass: UserGroup
      foreignAlias: Users

Phonenumber:
  actAs: [Timestampable]
  columns:
    user_id: integer
```

```
    phone: string(255)
    primary_num:
      type: boolean
      default: false
  relations:
    User:
      foreignAlias: Phonenumbers

Group:
  # required to renamed to groups because group is a reserved word in mysql
  tableName: groups
  columns:
    name: string(255)

UserGroup:
  columns:
    user_id: integer
    group_id: integer
```

Data fixtures

Listing 6.23:

```
---
User:
  User_1:
    username: jwage
    password: changeme
    Phonenumbers:
      Phonenumber_1:
        phone: 6155139185
        primary_num: true
      Phonenumber_2:
        phone: 6153137679
    Groups: [Group_1]

Group:
  Group_1:
    name: Group 1
  Group_2:
    name: Group 2
```

Count number of records for a relationship

Listing 6.24:

```
<?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

Listing 6.25:

```
<?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

Listing 6.26:

```
<?php

$q = Doctrine_Query::create()
        ->from('User u')
        ->where('u.username = ?', 'jwage');
$users = $q->fetchArray();

?>
```

Multiple WHERE with multiple parameters values

Listing 6.27:

```
<?php

$q = Doctrine_Query::create()
        ->from('User u')
        ->where('u.is_active = ? AND u.is_online = ?', array(1, 1));
$users = $q->fetchArray();

// You can also optionally use the addWhere() to add to the existing where parts
$q = Doctrine_Query::create()
        ->from('User u')
        ->where('u.is_active = ?', 1)
        ->addWhere('u.is_online = ?', 1);
$users = $q->fetchArray();

?>
```

Using whereIn() convenience method

Listing 6.28:

```
<?php

$q = Doctrine_Query::create()
        ->from('User u')
        ->whereIn('u.id', array(1, 2, 3))
$users = $q->fetchArray();

// This is the same as above
$q = Doctrine_Query::create()
        ->from('User u')
        ->where('u.id IN (1, 2, 3)');
$users = $q->fetchArray();

?>
```

Using DBMS function in your WHERE

Listing 6.29:

```
<?php

$userEncryptedKey = 'a157a558ac00449c92294c7fab684ae0';
$q = Doctrine_Query::create()
      ->from('User u')
      ->where("MD5(CONCAT(u.username, 'secret_user_key')) = ?",
         $userEncryptedKey);
$user = $q->fetchOne();

$q = Doctrine_Query::create()
        ->from('User u')
        ->where('LOWER(u.username) = LOWER(?)', 'jwage');
$user = $q->fetchOne();

?>
```

Limiting resultsets using aggregate functions

Listing 6.30:

```
<?php

// Users with more than 1 phonenumber
$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

Listing 6.31:

```
<?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

Listing 6.32:

```
<?php

$q = Doctrine_Query::create()
        ->select('u.username, p.phone')
        ->from('User u')
        ->leftJoin('u.Phonenumbers p');
$users = $q->fetchArray();

?>
```

Using wildcards to select all columns

Listing 6.33:

```
<?php
```

```php
// Select all User columns but only the phone phonenumber column
$q = Doctrine_Query::create()
        ->select('u.*, p.phone')
        ->from('User u')
        ->leftJoin('u.Phonenumbers p');
$users = $q->fetchArray();

?>
```

Perform DQL delete with simple WHERE

Listing 6.34:

```php
<?php

// Delete phonenumbers for user id = 5
$deleted = Doctrine_Query::create()
            ->delete()
            ->from('Phonenumber')
            ->addWhere('user_id = 5')
            ->execute();

?>
```

Perfom simple DQL update for a column

Listing 6.35:

```php
<?php

// Set user id = 1 to active
Doctrine_Query::create()
    ->update('User u')
    ->set('u.is_active', '?', true)
    ->where('u.id = ?', 1)
    ->execute();

?>
```

Perform DQL update with dbms functions

Listing 6.36:

```php
<?php

// Make all usernames lowercase
Doctrine_Query::create()
  ->update('User u')
  ->set('u.username', 'LOWER(u.username)')
  ->execute();

?>
```

Using mysql LIKE to search for records

Listing 6.37:

```php
<?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

Listing 6.38:

```
<?php

$q = Doctrine_Query::create()
        ->from('User u INDEXBY u.username');

$users = $q->fetchArray();
print_r($users['jwage']); // Will print the user with the username of jwage

?>
```

Using positional and named parameters

Listing 6.39:

```
<?php

// Positional parameters
$q = Doctrine_Query::create()
          ->from('User u')
          ->where('u.username = ?', array('Arnold'));
$users = $q->fetchArray();

// Named parameters
$q = Doctrine_Query::create()
          ->from('User u')
          ->where('u.username = :username', array(':username' => 'Arnold'));
$users = $q->fetchArray();

?>
```

Using subqueries in your WHERE

Listing 6.40:

```
<?php

// Find uers not in group named Group 2
$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 subqueries like the 2 below
// This is similar as above
$q = Doctrine_Query::create()
        ->from('User u')
        ->innerJoin('u.Groups g WITH g.name != ?', 'Group 2')
$users = $q->fetchArray();

// or this
$q = Doctrine_Query::create()
        ->from('User u')
        ->leftJoin('u.Groups g')
        ->where('g.name != ?', 'Group 2');

?>
```

Doctrine has many different ways you can execute queries and retrieve the data. Below is a list of all
the different ways you can execute queries.

Listing 6.41:

```
<?php

$q = Doctrine_Query::create()
        ->from('User u');

// Array hydration
$users = $q->fetchArray();                                      // Fetch the
    results as a hydrated array
$users = $q->execute(array(), Doctrine::HYDRATE_ARRAY);         // This is
    the same as above
$users = $q->setHydrationMode(Doctrine::HYDRATE_ARRAY)->execute(); // So is this

// No hydration
$users = $q->execute(array(), Doctrine::HYDRATE_NONE);          // Execute
    the query with plain PDO and no hydration
$users = $q->setHydrationMode(Doctrine::HYDRATE_NONE)->execute();  // This is
    the same as above

// Fetch one
$user = $q->fetchOne();

// Fetch all and get the first from collection
$user = $q->execute()->getFirst();

?>
```

6.3.2 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.

Lets say we have a User class with the following definition:

Listing 6.42:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 20);
        $this->hasColumn('password', 'string', 16);
        $this->hasColumn('description', 'string');
    }
}

?>
```

In the following example we fetch all the Users with the fields name and password loaded directly. Then we lazy-load a
huge field called description for one user.

Listing 6.43:

```
<?php

$users = Doctrine_Query::create()
          ->select('u.name, u.password')
          ->from('User u')
          ->execute();
```

```
// the following lazy-loads the description fields and executes one additional
    database query
$users[0]->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.

6.4 Arrays and objects

Doctrine_Records and Doctrine_Collections provide methods to facilitate working with arrays: `toArray()`,
`fromArray()` and `synchronizeWithArray()`.

6.4.1 toArray

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.

Listing 6.44:

```
<?php

print_r ($user->toArray()); // toArray(false) if you don't want to get the
    relations

?>
```

6.4.2 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.

Listing 6.45:

```
<?php

// If you have an array like this
$data = array(
  'name' => 'John',
  'age' => '25',
  'Emails' => array('john@mail.com', 'john@work.com')
);

// you can populate a user record with an Emails relationship like this
$user = new User();
```

```
$user->fromArray($data);
$user->Emails->count(); // --> 2

?>
```

6.4.3 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.

Listing 6.46:

```
<?php

$user = Doctrine_Query::create()
    ->select('u.*, g.*')
    ->from('User u')
    ->leftJoin('u.Groups g')
    ->where('id = ?', 1)
    ->fetchOne();

// Display this object on a cool javascript form that allows you to:

$arrayUser['name'] = 'New name'; // modify a field
$arrayUser['Group'][0]['name'] = 'Renamed Group'; // modify a field on a
    relation
$arrayUser['Group'][] = array('name' => 'New Group'); // create a new relation
unset($arrayUser['Group'][1]); // even remove a relation

// submit the form and on the next script use the same query to retrieve the
    record

$user = Doctrine_Query::create()
    ->select('u.*, g.*')
    ->from('User u')
    ->leftJoin('u.Groups g')
    ->where('id = ?', 1)
    ->fetchOne();

// sanitize the form input an get the data

$user->synchronizeWithArray($arrayUser);
$user->save(); // all changes get applied to the user object

?>
```

6.5 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:

Listing 6.47:

```
<?php

class User extends Doctrine_Record
```

```
{
    public function construct()
    {
        $this->name = 'Test Name';
        $this->do_something();
    }
}

?>
```

The only drawback is that it doesn't provide a way to pass parameters to the constructor.

Chapter 7

Component overview

7.1 Record

Doctrine_Record is one of the most essential components of Doctrine ORM. The class is a wrapper for database row but
along with that it speficies what relations it has
on other components and what columns it has. It may access the related components, hence its refered as an ActiveRecord.

The classes that inherit Doctrine_Record are called components. There should be atleast one component for each
database table.

You can instantiate and use your models like the following below.

Listing 7.1:

```
<?php

$user = new User();

// records support array access
$user['name'] = 'John Locke';

// save user into database
$user->save();

?>
```

Every record has an object identifier, which is an internal unique identifier. You can get the object identifier with
the oid() method. Basically two objects are considered the same if they share the same object identifier.

7.1.1 Properties

Each assigned column property of Doctrine_Record represents a database table column. As you've learned in the
previous chapters the column definitions can be achieved with the hasColumn() method. Now accessing the columns is
easy. You can use any of the means described above. The recommended way is using the ArrayAccess as it makes it easy
to switch between record and array fetching when needed.

Listing 7.2:

```php
<?php

$table = Doctrine::getTable('User');

$user = $table->find(3);

// access property through overloading

$name = $user->name;

// access property with get()

$name = $user->get("name");

// access property with ArrayAccess interface

$name = $user['name'];

?>
```

Iterating trhough 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.

Listing 7.3:

```php
<?php

foreach ($user as $field => $value) {

}

?>
```

As with arrays you can use the isset() for checking if given property exists and unset() for setting given property to
null.

Listing 7.4:

```php
<?php

// checking if property called 'name' exists
if (isset($user['name'])) {

}

// unsetting name property
unset($user['name']);

?>
```

When you have set values for record properties you can get an array of the modified fields and values using
`Doctrine_Record::modifiedFields()`

Listing 7.5:

```php
<?php

$user['name'] = 'Jack Daniels';
```

```php
$user['age'] = 100;

print_r($user->getModified()); // array('name' => 'Jack Daniels', 'age' => 100);

$user->isModified(); // true

?>
```

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.

Listing 7.6:

```php
<?php

// get the number of columns

$colCount = $record->count();

$colCount = count($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.

Listing 7.7:

```php
<?php

$user['name'] = 'Jack Daniels';

$user->save();

$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 given
array to property values.

The merge() method iterates through the properties of given record and assigns the values of given array to the
associated properties.

Listing 7.8:

```php
<?php

$values = array('name' => 'someone',
                'age'  => 11,
                'unknownproperty' => '...');

// notice that here the unknownproperty won't get assigned
// as the User class doesn't have a column with that name
```

```
$user->merge($values);

print $user->name; // someone
print $user->age; // 11

print $user->unknownproperty; // throws exception

?>
```

7.1.2 Retrieving existing records

Doctrine provides many ways for record retrieval. The fastest ways for retrieving existing records are the finder
methods provided by `Doctrine_Table`. If you need to use more complex queries take a look at the DQL API.

Listing 7.9:

```
<?php

$table = Doctrine::getTable("User");

// find by primary key

$user = $table->find(2);
if ($user !== false)
    print $user->name;

// get all users
$users = $table->findAll();
foreach($users as $user) {
    print $user->name;
}

// finding by dql
$users = $table->findByDql("name LIKE '%John%'");
foreach($users as $user) {
    print $user->created;
}

// finding with magic accessors
$user = $table->findOneByName('jon'); // find user named jon
$users = $table->findByAge(10); // find users with age of 10

// finding objects with DQL

$users = Doctrine_Query::create()->from('User u')->where("u.name LIKE '%John%'")
    ->execute();

?>
```

7.1.3 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.

Listing 7.10:

```
<?php

$table = Doctrine::getTable('User');

$user = $table->find(2);

if($user !== false) {
    $user->name = '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.

Listing 7.11:

```
<?php

// make all usernames lowercased
Doctrine_Query::create()
  ->update('User u')
  ->set('u.name', 'LOWER(u.name)')
  ->execute();

?>
```

7.1.4 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.

Listing 7.12:

```
<?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.

Listing 7.13:

```
<?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

?>
```

7.1.5 Refreshing records

Sometimes you may want to refresh your record with data from the database, use `Doctrine_Record::refresh()`.

Listing 7.14:

```
<?php

$user = Doctrine::getTable('User')->find(2);
$user->name = 'New name';
// oups, I want to refresh the name
$user->refresh();

?>
```

7.1.5.1 Refreshing relationships

The `Doctrine_Record::refresh($deep = false)` method can also refresh the already loaded record relationships, but you need to specify them on the original query.

Listing 7.15:

```
<?php

$user = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Groups')
    ->where('id = ?')
    ->fetchOne(array(1));

$group = Doctrine_Query::create()
    ->from('Group g')
    ->leftJoin('g.Users')
    ->where('id = ?')
    ->fetchOne(array(1));

$userGroup = new UserGroup();
$userGroup->user_id = $user->id;
$userGroup->group_id = $group->id;
$userGroup->save();

// get new group on user
$user->refresh(true);
// get new user on group
$group->refresh(true);

?>
```

You can also lazily refresh individual relationships or all defined relationships of a model using `Doctrine_Record::refreshRelated($name = null)`.

Listing 7.16:

```php
<?php

$user = Doctrine::getTable('User')->findOneByName('jon');
$user->refreshRelated(); // Will lazily load all defined relationships

$user->refreshRelated('Phonenumber'); // Will lazily load the Phonenumber
    relationship

?>
```

7.1.6 Deleting records

Deleting records in Doctrine is handled by `Doctrine_Record::delete()`, `Doctrine_Collection::delete()` and
`Doctrine_Connection::delete()` methods.

Listing 7.17:

```php
<?php

$table = Doctrine::getTable("User");

$user = $table->find(2);

// deletes user and all related composite objects
if($user !== false)
    $user->delete();

$users = $table->findAll();

// delete all users and their related composite objects
$users->delete();

?>
```

7.1.7 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 timepoint can be achieved as follows:

Listing 7.18:

```php
<?php

$event = new Event();
$event->name = 'Rock festival';
$event->timepoint = new Doctrine_Expression('NOW()');

$event->save();

?>
```

The last line would execute sql (in sqlite):

Listing 7.19:

```sql
INSERT INTO event (name, timepoint) VALUES ('Rock festival', 'NOW()')
```

7.1.8 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 atleast 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.

Listing 7.20:

```
<?php

$state = $record->state();

switch($state):
    case Doctrine_Record::STATE_PROXY:
        // record is in proxy state,
        // meaning its persistent but not all of its properties are
        // loaded from the database
    break;
    case Doctrine_Record::STATE_TCLEAN:
        // record is transient clean,
        // meaning its transient and
        // none of its properties are changed
    break;
    case Doctrine_Record::STATE_TDIRTY:
        // record is transient dirty,
        // meaning its transient and
        // some of its properties are changed
    break;
    case Doctrine_Record::STATE_DIRTY:
        // record is dirty,
        // meaning its persistent and
        // some of its properties are changed
    break;
    case Doctrine_Record::STATE_CLEAN:
        // record is clean,
        // meaning its persistent and
        // none of its properties are changed
    break;
    case Doctrine_Record::STATE_LOCKED:
        // record is locked
    break;
endswitch;

?>
```

7.1.9 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().

Listing 7.21:

```
<?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).

Listing 7.22:

```
<?php

// get a copy of user without the relations
$copy = $user->copy(false);

?>
```

7.1.10 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.

Listing 7.23:

```
<?php

$user = new User();
$user->state('TDIRTY');
$user->save();

$user->id; // 1

?>
```

7.1.11 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 availible at runtime not persisting those values into database. This can be
achieved as follows:

Listing 7.24:

```
<?php

$user->mapValue('isRegistered', true);

$user->isRegistered; // true

?>
```

7.1.12 Serializing

Sometimes you may want to serialize your record objects (possibly for caching purposes). Records can be serialized, but
remember: Doctrine cleans all relations, before doing this. So remember to persist your objects into database before
serializing them.

Listing 7.25:

```
<?php

$string = serialize($user);

$user = unserialize($string);

?>
```

7.1.13 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.

Listing 7.26:

```
<?php

$record = new User();

$record->exists(); // false

$record->name = 'someone';
$record->save();

$record->exists(); // true

?>
```

7.1.14 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 type:

Listing 7.27:

```
<?php

$record->call('trim', 'column1');

?>
```

7.2 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:

Listing 7.28:

```php
<?php

$conn = Doctrine_Manager::getInstance()
        ->openConnection(new PDO("dsn", "username", "pw"));

// initalizing a new collection
$users = new Doctrine_Collection(Doctrine::getTable('User'));

// alternative (propably easier)
$users = new Doctrine_Collection('User');

// adding some data
$users[0]->name = 'Arnold';

$users[1]->name = 'Somebody';

// finally save it!
$users->save();

?>
```

7.2.1 Accessing elements

You can access the elements of `Doctrine_Collection` with `set()` and `get()` methods or with `ArrayAccess`
interface.

Listing 7.29:

```php
<?php

$table = Doctrine::getTable("User");

$users = $table->findAll();

// accessing elements with ArrayAccess interface

$users[0]->name = "Jack Daniels";

$users[1]->name = "John Locke";

// accessing elements with get()

print $users->get(1)->name;

?>
```

7.2.2 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.

Listing 7.30:

```
<?php

$users = $table->findAll();

print count($users); // 5

$users[5]->name = "new user 1";
$users[6]->name = "new user 2";

?>
```

7.2.3 Getting collection count

The `Doctrine_Collection` method `count()` returns the number of elements currently in the collection.

Listing 7.31:

```
<?php

$users = $table->findAll();

print $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.

Listing 7.32:

```
print count($users); // Doctrine_Collection implements Countable interface
```

7.2.4 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.

Listing 7.33:

```
<?php

$users = $table->findAll();

$users[0]->name = 'Jack Daniels';

$users[1]->name = 'John Locke';

$users->save();

?>
```

7.2.5 Deleting 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:

Listing 7.34:

```
DELETE FROM user WHERE id IN (1,2,3, ... ,N)
```

7.2.6 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.

Listing 7.35:

```
<?php

// mapping id column

$user = new User();

$user->setAttribute(Doctrine::ATTR_COLL_KEY, 'id');

// now user collections will use the values of
// id column as element indexes

$users = Doctrine::getTable('User')->findAll();

foreach($users as $id => $user) {
    print $id . $user->name;
}

// mapping name column

$user = new User();

$user->setAttribute(Doctrine::ATTR_COLL_KEY, 'name');

// now user collections will use the values of
// name column as element indexes

$users = Doctrine::getTable('User')->findAll();

foreach($users as $name => $user) {
    print $name . $user->type;
}

?>
```

7.2.7 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.

Listing 7.36:

```
<?php

$users = Doctrine_Query::create()
          ->from('User u')
          ->execute();

// now lets load phonenumbers for all users

$users->loadRelated('Phonenumber');

foreach($users as $user) {
    print $user->Phonenumber[0]->phonenumber;
    // no additional db queries needed here
}

// the loadRelated works an any relation, even associations:

$users->loadRelated('Group');

foreach($users as $user) {
    print $user->Group[0]->name;
}

?>
```

The example below shows how to do this more efficiently by using the DQL API.

Listing 7.37:

```
<?php

// load everything here
$users = Doctrine_Query::create()
          ->from('User u')
          ->leftJoin('u.Phonenumber p')
          ->leftJoin('u.Group g')
          ->execute();

foreach($users as $user) {
    // no additional db queries needed here
    print $user->Phonenumber->phonenumber;

    print $user->Group->name;
}

?>
```

7.3 Connection

Doctrine_Connection is a wrapper for database connection. It 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 the DQL API (see `Doctrine_Query`)
- Optionally validates transactions using `Doctrine_Validator` and gives full information of possible errors.

7.3.1 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

7.3.2 Getting a table object

In order to get table object for specified record just call `Doctrine_Record::getTable()` or `Doctrine_Connection::getTable()`.

Listing 7.38:

```php
<?php

$manager = Doctrine_Manager::getInstance();

// open new connection

$conn = $manager->openConnection(new PDO('dsn','username','password'));

// getting a table object

$table = Doctrine::getTable('User');

?>
```

7.3.3 Flushing the connection

Creating new record (database row) is very easy. You can either use the `Doctrine_Connection::create()` or
`Doctrine_Table::create()` method to do this or just simply use the new operator.

Listing 7.39:

```
<?php

$user = new User();
$user->name = 'Jack';

$group = $conn->create('Group');
$group->name = 'Drinking Club';

// saves all the changed objects into database

$conn->flush();

?>
```

7.3.4 Querying the database

`Doctrine_Connection::query()` is a simple method for efficient object retrieval. It takes one parameter (DQL query)
and optionally prepared statement params.

Listing 7.40:

```
<?php

// select all users

$users = Doctrine_Query::create()
            ->from('User u')
            ->execute();

// select all users where user email is jackdaniels@drinkmore.info

$users = Doctrine_Query::create()
            ->from('User u')
            ->leftJoin('u.Email e')
            ->where('e.address = ?', 'jackdaniels@drinkmore.info')
            ->execute();

// using prepared statements

$users = Doctrine_Query::create()
            ->from('User u')
            ->where('u.name = ?', array('Jack'))
            ->execute();

?>
```

7.4 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, which means you can retrieve the tables from the
connection by calling the getTable() method with the appropriate component name.

For example, lets say we want to retrieve the table object for the User class. We can do this by simply giving the 'User'
as the first argument for the getTable() method.

Listing 7.41:

```
// get the current connection
$conn = Doctrine_Manager::connection();

$table = Doctrine::getTable('User');
```

7.4.1 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:

Listing 7.42:

```
<?php

// getting all information of all columns
$columns = $table->getColumns();

?>
```

Sometimes this can be an overkill. The following example shows how to retrieve the column names as an array:

Listing 7.43:

```
// getting column names
$names = $table->getColumnNames();
```

7.4.2 Getting relation information

7.4.3 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().

Listing 7.44:

```
<?php

$table = Doctrine::getTable('User');

// find by primary key

$user = $table->find(2);
```

```php
if($user !== false)
    print $user->name;

// get all users
foreach($table->findAll() as $user) {
    print $user->name;
}

// finding by dql
foreach($table->findByDql("name LIKE '%John%'") as $user) {
    print $user->created;
}

?>
```

7.4.3.1 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 inherit `Doctrine_Table`.

Listing 7.45:

```php
<?php

// valid table object

class UserTable extends Doctrine_Table
{
}

// not valid [doesn't extend Doctrine_Table]
class GroupTable { }

?>
```

7.4.4 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_Connection::query()`).

Listing 7.46:

```php
<?php

class UserTable extends Doctrine_Table {
    /**
     * you can add your own finder methods here
     */
    public function findByName($name) {
        return Doctrine_Query::create()
                ->from('User u')
                ->where('u.name LIKE ?', "%$name%")
                ->execute();
    }
}
class User extends Doctrine_Record { }

$conn = Doctrine_Manager::getInstance()->openConnection('mysql://username:
    password@localhost/dbname');
```

```
// doctrine will now check if a class called UserTable exists
// and if it inherits Doctrine_Table

$table    = Doctrine::getTable('User');

print get_class($table); // UserTable

$users    = $table->findByName("Jack");

?>
```

7.5 Validators

7.5.1 Introduction

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` (see the code below).

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.

Listing 7.47:

```
<?php

// turning on validation
Doctrine_Manager::getInstance()->setAttribute(Doctrine::ATTR_VALIDATE, Doctrine:
    :VALIDATE_ALL);

?>
```

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:

Listing 7.48:

```
VALIDATE_ALL & ~VALIDATE_LENGTHS
```

7.5.2 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 at the bottom of this page. 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 thess methods you can use all the power of PHP to validate your fields.
When a field doesnt 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:

Listing 7.49:

```
<?php

class User extends Doctrine_Record
```

```
{
    public function setUp()
    {
        $this->ownsOne('Email', array('local' => 'email_id'));
    }
    public function setTableDefinition()
    {
        // no special validators used only types
        // and lengths will be validated
        $this->hasColumn('name', 'string', 15);
        $this->hasColumn('email_id', 'integer');
        $this->hasColumn('created', 'integer', 11);
    }
    // Our own validation
    protected function validate()
    {
        if ($this->name == 'God') {
            // Blasphemy! Stop that! ;-)
            // syntax: add(<fieldName>, <error code/identifier>)
            $this->getErrorStack()->add('name', 'forbiddenName');
        }
    }
}
class Email extends Doctrine_Record
{
    public function setTableDefinition()
    {
        // validators 'email' and 'unique' used
        $this->hasColumn("address","string",150, array("email", "unique"));
    }
}

?>
```

7.5.3 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.

7.5.3.1 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_Erro`
The error stack provides an easy to use interface to inspect the errors.

7.5.3.2 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`.

Listing 7.50:

```php
<?php

try {
    $user->name = "this is an example of too long name";
    $user->Email->address = "drink@@notvalid..";
    $user->save();
} catch(Doctrine_Validator_Exception $e) {
    // Note: you could also use $e->getInvalidRecords(). The direct way
    // used here is just more simple when you know the records you're dealing
        with.
    $userErrors = $user->getErrorStack();
    $emailErrors = $user->Email->getErrorStack();

    /* Inspect user errors */
    foreach($userErrors as $fieldName => $errorCodes) {
        switch ($fieldName) {
            case 'name':
                // $user->name is invalid. inspect the error codes if needed.
            break;
        }
    }

    /* Inspect email errors */
    foreach($emailErrors as $fieldName => $errorCodes) {
        switch ($fieldName) {
            case 'address':
                // $user->Email->address is invalid. inspect the error codes if
                    needed.
            break;
        }
    }
}

?>
```

7.6 Profiler

7.6.1 Introduction

`Doctrine_Connection_Profiler` is an eventlistener 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 eventlistener for `Doctrine_Connection`.

Listing 7.51:

```
<?php

$conn = Doctrine_Manager::connection($dsn);

$profiler = new Doctrine_Connection_Profiler();

$conn->setListener($profiler);

?>
```

7.6.2 Basic usage

Perhaps some of your pages is loading slowly. The following shows how to build a complete profiler report from the
connection:

Listing 7.52:

```
<?php

$time = 0;
foreach ($profiler as $event) {
    $time += $event->getElapsedSecs();
    echo $event->getName() . " " . sprintf("%f", $event->getElapsedSecs()) . "<
        br>\n";
    echo $event->getQuery() . "<br>\n";
    $params = $event->getParams();
    if( ! empty($params)) {
        var_dump($params);
    }
}
echo "Total time: " . $time  . "<br>\n";

?>
```

7.7 Locking manager

7.7.1 Introduction

[**Note**: The term 'Transaction' doesnt refer to database transactions here but to the general meaning of this term]

[**Note**: This component is in **Alpha State**]

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 noone 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.

7.7.2 Examples

The following code snippet demonstrates the use of Doctrine's pessimistic offline locking capabilities.

At the page where the lock is requested...

Listing 7.53:

```
<?php

// Get a locking manager instance
$lockingMngr = new Doctrine_Locking_Manager_Pessimistic();

try
{
    // Ensure that old locks which timed out are released
    // before we try to acquire our lock
    // 300 seconds = 5 minutes timeout
    $lockingMngr->releaseAgedLocks(300);

    // Try to get the lock on a record
    $gotLock = $lockingMngr->getLock(
     // The record to lock. This can be any Doctrine_Record
                        $myRecordToLock,
    // The unique identifier of the user who is trying to get the lock
                       'Bart Simpson'
                 );

    if($gotLock)
    {
        echo "Got lock!";
        // ... proceed
    }
    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...

Listing 7.54:

```
<?php

// Get a locking manager instance
$lockingMngr = new Doctrine_Locking_Manager_Pessimistic();

try
{
    if($lockingMngr->releaseLock($myRecordToUnlock, 'Bart Simpson'))
    {
        echo "Lock released";
    }
    else
    {
        echo "Record was not locked. No locks released.";
    }
}
catch(Doctrine_Locking_Exception $dle)
{
    echo $dle->getMessage();
    // handle the error
}

?>
```

7.7.3 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 behaviour may change in the future to provide a centralised and consistent table creation procedure for
installation purposes.

7.8 View

7.8.1 Introduction

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.

7.8.2 Managing views

Listing 7.55:

```php
<?php

$conn = Doctrine_Manager::getInstance()
        ->openConnection(new PDO("dsn","username","password"));

$query = Doctrine_Query::create($conn);
$query->from('User.Phonenumber')->limit(20);

$view  = new Doctrine_View($query, 'MyView');

// creating a database view
$view->create();

// dropping the view from the database
$view->drop();

?>
```

7.8.3 Using views

Listing 7.56:

```php
<?php

$conn = Doctrine_Manager::getInstance()
        ->openConnection(new PDO("dsn","username","password"));

$query = Doctrine_Query::create($conn);
$query->from('User u, u.Phonenumber')->limit(20);

// hook the query into appropriate view
$view  = new Doctrine_View($query, 'MyView');

// now fetch the data from the view
$coll  = $view->execute();

?>
```

Chapter 8

Hierarchical data

8.1 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[1].

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 in the following chapters, or see

- http://www.dbazine.com/oracle/or-articles/tropashko4
- http://dev.mysql.com/tech-resources/articles/hierarchical-data.html

[1] http://en.wikipedia.org/wiki/Tree_data_structure

8.2 Nested set

8.2.1 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

8.2.2 Setting up

To set up your model as Nested Set, you must add the following code to your model's table definition.

Listing 8.1:

```php
<?php

...
    public function setTableDefinition()
    {
        ...

        $this->actAs('NestedSet');

       ...
    }
...

?>
```

"actAs" is a convenience method that loads templates that are shipped with Doctrine(Doctrine_Template_* classes). The
more general alternative would look like this:

Listing 8.2:

```php
<?php

...
    public function setTableDefinition()
    {
        ...

        $this->actAs('Doctrine_Template_NestedSet');

       ...
    }
...

?>
```

Detailed information on Doctrine's templating model can be found in chapter [doc class-templates :index :name]. These
templates add some functionality to your model. In the example of the nested set, your model gets 3 additional
fields: "lft", "rgt", "level". You never need to care about "lft" and "rgt". 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 indendation.

You must never assign values to lft, rgt, level. These are managed transparently by the nested set implementation.

8.2.3 More than 1 tree in a single table

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 based upon the set up above:

Listing 8.3:

```
<?php

...
    public function setTableDefinition()
    {
        ...
        $options = array('hasManyRoots'     => true,         // enable many roots
                         'rootColumnName'   => 'root_id');   // set root column
                             name, defaults to 'root_id'
        $this->actAs('NestedSet', $options);

       ...
    }
...

?>
```

The rootColumnName is the column that is used to differentiate between trees. When you create a new node to insert it
into an existing tree you dont need to care about this field. This is done by the nested set implementation. However,
when you want to create a new root node you have the option to set the "root_id" manually. The nested set implementation
will recognize that. In the same way you can move nodes between different trees without caring about the "root_id". All
of this is handled for you.

8.2.4 Working with the tree(s)

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 2 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:

Listing 8.4:

```
<?php

// Assuming $conn is an instance of some Doctrine_Connection
  $treeObject = Doctrine::getTable('MyNestedSetModel')->getTree();
  // ... the full tree interface is available on $treeObject

  // Assuming $entity is an instance of MyNestedSetModel
  $nodeObject = $entity->getNode();
  // ... 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.

8.2.4.1 Creating a root node

Listing 8.5:

```
<?php

...
$root = new MyNestedSetModel();
$root->name = 'root';
$treeObject = Doctrine::getTable('MyNestedSetModel')->getTree();
$treeObject->createRoot($root); // calls $root->save() internally
...

?>
```

8.2.4.2 Inserting a node

Listing 8.6:

```
<?php

...
// Assuming $someOtherRecord is an instance of MyNestedSetModel
$record = new MyNestedSetModel();
$record->name = 'somenode';
$record->getNode()->insertAsLastChildOf($someOtherRecord); // calls $record->
    save() internally
...

?>
```

8.2.4.3 Deleting a node

Listing 8.7:

```
<?php

...
// Assuming $record is an instance of MyNestedSetModel
$record->getNode()->delete();
// calls $record->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 dont want to delete them.

8.2.4.4 Moving a node

Listing 8.8:

```
<?php

...
// Assuming $record and $someOtherRecord are both instances of MyNestedSetModel
$record->getNode()->moveAsLastChildOf($someOtherRecord);
...

?>
```

There are 4 move methods: moveAsLastChildOf($other), moveAsFirstChildOf($other), moveAsPrevSiblingOf($other) and
moveAsNextSiblingOf($other). The method names are self-explanatory.

8.2.4.5 Examining a node

Listing 8.9:

```
<?php

...
// Assuming $record is an instance of MyNestedSetModel
$isLeaf = $record->getNode()->isLeaf(); // true/false
$isRoot = $record->getNode()->isRoot(); // true/false
...

?>
```

8.2.4.6 Examining and retrieving siblings

Listing 8.10:

```
<?php

...
// Assuming $record is an instance of MyNestedSetModel
$hasNextSib = $record->getNode()->hasNextSibling(); // true/false
$haPrevSib = $record->getNode()->hasPrevSibling(); // true/false
```

```
$nextSib = $record->getNode()->getNextSibling(); // returns false if there is no
     next sibling, otherwise returns the sibling
$prevSib = $record->getNode()->getPrevSibling(); // returns false if there is no
     previous sibling, otherwise returns the sibling

$siblings = $record->getNode()->getSiblings(); // an array of all siblings
...

?>
```

8.2.4.7 Examining and retrieving children / parents / descendants / ancestors

Listing 8.11:

```
<?php

...
// Assuming $record is an instance of MyNestedSetModel
$hasChildren = $record->getNode()->hasChildren(); // true/false
$hasParent = $record->getNode()->hasParent(); // true/false

$firstChild = $record->getNode()->getFirstChild(); // returns false if there is
    no first child, otherwise returns the child
$lastChild = $record->getNode()->getLastChild(); // returns false if there is no
     lase child, otherwise returns the child
$parent = $record->getNode()->getParent(); // returns false if there is no
    parent, otherwise returns the parent

$children = $record->getNode()->getChildren(); // returns false if there are no
    children, otherwise returns the children
// !!! IMPORATNT: getChildren() returns only the direct descendants. If you want
     all descendants, use getDescendants() !!!

$descendants = $record->getNode()->getDescendants(); // returns false if there
    are no descendants, otherwise returns the descendants
$ancestors = $record->getNode()->getAncestors(); // returns false if there are
    no ancestors, otherwise returns the ancestors

$numChildren = $record->getNode()->getNumberChildren(); // returns the number of
     children
$numDescendants = $record->getNode()->getNumberDescendants(); // returns the
    number of descendants

...

?>
```

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).

8.2.4.8 Simple Example: Displaying a tree

Listing 8.12:

```
<?php

...
$treeObject = Doctrine::getTable('MyNestedSetModel')->getTree();
$tree = $treeObject->fetchTree();
foreach ($tree as $node) {
    echo str_repeat('  ', $node['level']) . $node['name'] . '<br />';
}
...

?>
```

8.2.5 Advanced usage

The previous sections have explained the basic usage of Doctrine's nested set implementation. This section will go one
step further.

8.2.5.1 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. Take a look at the following code snippet:

Listing 8.13:

```
<?php

$query = Doctrine_Query::create();
            ->select("cat.name, hp.name, m.name")->from("Category cat")
            ->leftJoin("cat.hottestProduct hp")
            ->leftJoin("hp.manufacturer m");

$treeObject = Doctrine::getTable('Category')->getTree();
$treeObject->setBaseQuery($query);
$tree = $treeObject->fetchTree();
$treeObject->resetBaseQuery();

?>
```

There it is, the tree with all the related data you need, all in one query.

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 do:

Listing 8.14:

```php
<?php

$query = Doctrine_Query::create();
            ->select("base.name, hp.name, m.name")->from("Category base")
            ->leftJoin("base.hottestProduct hp")
            ->leftJoin("hp.manufacturer m")
            ->setHydrationMode(Doctrine::HYDRATE_ARRAY);

$treeObject = Doctrine::getTable('Category')->getTree();
$treeObject->setBaseQuery($query);
$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.

8.3 Examples

Listing 8.15:

```php
<?php

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

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

?>
```

Listing 8.16:

```yaml
---
Category:
  actAs: [NestedSet]
  columns:
    name: string(255)
```

Chapter 9

Configuration

9.1 Introduction

Listing 9.1:

```php
<?php

$manager = Doctrine_Manager::getInstance();

$manager->setAttribute(Doctrine::ATTR_LISTENER, new MyListener());

?>
```

9.2 Levels of configuration

Doctrine has a three-level configuration structure. You can set configuration attributes in 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 user first sets default fetchmode in global level to `Doctrine::FETCH_BATCH` and then sets `example`
table fetchmode to `Doctrine::FETCH_LAZY`, the lazy fetching strategy will be used whenever the records of 'example'
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:

Listing 9.2:

```php
<?php

// setting a global level attribute
$manager = Doctrine_Manager::getInstance();
```

```
$manager->setAttribute(Doctrine::ATTR_VALIDATE, Doctrine::VALIDATE_ALL);

?>
```

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

Listing 9.3:

```
<?php

// setting a connection level attribute
// (overrides the global level attribute on this connection)

$conn = $manager->openConnection(new PDO('dsn', 'username', 'pw'));

$conn->setAttribute(Doctrine::ATTR_VALIDATE, Doctrine::VALIDATE_NONE);

?>
```

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

Listing 9.4:

```
<?php

// setting a table level attribute
// (overrides the connection/global level attribute on this table)

$table = Doctrine::getTable('User');

$table->setAttribute(Doctrine::ATTR_LISTENER, new UserListener());

?>
```

9.3 General attributes

9.3.1 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 scripts
over to another server type. Doctrine strives to overcome these differences so your program can switch between DBMS's
without any changes.

You control which portability modes are enabled by using the portability configuration option. Configuration options
are set via `factory()` and `setOption()`.

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

9.3.1.1 Portability Mode Constants

`Doctrine::PORTABILITY_ALL` **(default)** turn on all portability features. this is the default setting.

`Doctrine::PORTABILITY_DELETE_COUNT` Force reporting the number of rows deleted. Some DBMS's don't count the

number of rows deleted when performing simple `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.

`Doctrine::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.

`Doctrine::PORTABILITY_ERRORS` Makes certain error messages in certain drivers compatible with those from other

DBMS's

`Doctrine::PORTABILITY_FIX_ASSOC_FIELD_NAMES` This removes any qualifiers from keys in associative fetches. some

RDBMS , like for example SQLite, will be default use the fully qualified name for a column in assoc fetches if it is
qualified in a query.

`Doctrine::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`

`Doctrine::PORTABILITY_NONE` Turn off all portability features

`Doctrine::PORTABILITY_NUMROWS` Enable hack that makes `numRows()` work in Oracle

`Doctrine::PORTABILITY_EXPR` Makes DQL API throw exceptions when non-portable expressions are being used.

`Doctrine::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.

9.3.1.2 Examples

Using `setAttribute()` to enable portability for lowercasing and trimming

Listing 9.5:

```
<?php

$conn->setAttribute('portability',
        Doctrine::PORTABILITY_FIX_CASE | Doctrine::PORTABILITY_RTRIM);

?>
```

Using `setAttribute()` to enable all portability options except trimming

Listing 9.6:

```
<?php

$conn->setAttribute('portability',
        Doctrine::PORTABILITY_ALL ^ Doctrine::PORTABILITY_RTRIM);

?>
```

9.3.2 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.

NOTE: 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:

- backtick (`) – due to MySQL
- double quote (") – due to Oracle
- brackets ([or]) – due to Access

Delimited identifiers are known to generally work correctly under the following drivers:

- Mssql
- Mysql
- Oracle
- Pgsql

- Sqlite
- Firebird

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

Listing 9.7:

```
<?php

$conn->setAttribute(Doctrine::ATTR_QUOTE_IDENTIFIER, true);

?>
```

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

Listing 9.8:

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

as opposed to:

Listing 9.9:

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

9.3.3 Exporting

The export attribute is used for telling Doctrine what it should export when exporting classes.

If you don't want to export anything when calling export() you can use:

Listing 9.10:

```
<?php

$manager->setAttribute(Doctrine::ATTR_EXPORT, Doctrine::EXPORT_NONE);

?>
```

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

Listing 9.11:

```
<?php

$manager->setAttribute(Doctrine::ATTR_EXPORT, Doctrine::EXPORT_TABLES);
// or you can use
$manager->setAttribute(Doctrine::ATTR_EXPORT, Doctrine::EXPORT_ALL ^ Doctrine::
    EXPORT_CONSTRAINTS);

?>
```

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

Listing 9.12:

```
<?php

$manager->setAttribute(Doctrine::ATTR_EXPORT, Doctrine::EXPORT_ALL);

?>
```

9.3.4 Event listener

Listing 9.13:

```
<?php

// setting default event listener
$manager->setAttribute(Doctrine::ATTR_LISTENER, new MyListener());

?>
```

9.4 Naming convention attributes

Naming convention attributes affect on 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.

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.

9.4.1 Index name format

Doctrine::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'.

Listing 9.14:

```
<?php

// changing the index naming convention
$manager->setAttribute(Doctrine::ATTR_IDXNAME_FORMAT, '%s_index');

?>
```

9.4.2 Sequence name format

Similar to Doctrine::ATTR_IDXNAME_FORMAT, Doctrine::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'.

Listing 9.15:

```
<?php

// changing the sequence naming convention
$manager->setAttribute(Doctrine::ATTR_IDXNAME_FORMAT, '%s_sequence');

?>
```

9.4.3 Table name format

Listing 9.16:

```
<?php

// changing the table naming convention
$manager->setAttribute(Doctrine::ATTR_TBLNAME_FORMAT, '%s_table');

?>
```

9.4.4 Database name format

Listing 9.17:

```
<?php

// changing the database naming convention
$manager->setAttribute(Doctrine::ATTR_DBNAME_FORMAT, 'myframework_%s');

?>
```

9.5 Validation attributes

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

9.5.1 Validation mode constants

{Doctrine::VALIDATE_NONE} Turns off the whole validation procedure. This is the default value.

{Doctrine::VALIDATE_LENGTHS} Makes Doctrine validate all field lengths.

{Doctrine::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.

{Doctrine::VALIDATE_CONSTRAINTS} Makes Doctrine validate all field constraints such as notnull, email etc.

{Doctrine::VALIDATE_ALL} Turns on all validations.

9.5.2 Examples

Turning on all validations:

Listing 9.18:

```php
<?php

$manager->setAttribute(Doctrine::ATTR_VALIDATE, Doctrine::VALIDATE_ALL);

?>
```

Validating lengths and types, but not constraints:

Listing 9.19:

```php
<?php

$manager->setAttribute(Doctrine::ATTR_VALIDATE, Doctrine::VALIDATE_LENGTHS |
    Doctrine::VALIDATE_TYPES);

?>
```

Chapter 10

Data fixtures

Doctrine Data uses the Doctrine Parser for the dumping and loading of fixtures data so it is possible to use any of the
formats available in the Parser. Currently yml is the only fully supported format but xml and others are next.

10.1 Exporting

You can export data to fixtures file in many different formats

Listing 10.1:

```
<?php

// A few ways exist for specifying where you export the data

// Dump to one large fixture file
$data = new Doctrine_Data();
$data->exportData('data.yml', 'yml');

// Dump to individual files. One file per model. 4th argument true specifies to
    dump to individual files
$data = new Doctrine_Data();
$data->exportData('path/to/directory', 'yml', true);

?>
```

10.2 Importing

You can import data from fixtures files in many different formats

Listing 10.2:

```
<?php

// Path can be in a few different formats
$path = 'path/to/data.yml'; // Path directly to one yml file
$path = array('data.yml', 'data2.yml', 'more.yml'); // Array of yml file paths
$path = array('directory1', 'directory2', 'directory3'); // Array of directories
     which contain yml files. It will find
all files with an extension of .yml

// Specify the format of the data you are importing
$format = 'yml'; // xml, yml, json
```

```
$models = array('User', 'Phonenumber'); // you can optionally specify an array
    of the models you wish to import the data
for, by default it loads data for all the available loaded models and the data
    that exists

$data = new Doctrine_Data();
$data->importData($path, $format, $models);

?>
```

10.3 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.

Imagine a schema with the following relationships:

Listing 10.3:

```
<?php

Resource hasMany Tag as Tags
Resource hasOne ResourceType as Type
ResourceType hasMany Resource as Resources
Tag hasMany Resource as Resources

?>
```

Note: All row keys across all yaml data fixtures must be unique. For example below tutorial, doctrine, help, cheat are all unique.

Listing 10.4:

```
---
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.

Listing 10.5:

```
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]
```

Here is how you would write code to load the data from that data.yml file

Listing 10.6:

```
<?php

$data = new Doctrine_Data();
$data->importData('data.yml', 'yml');

?>
```

10.4 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 this:

Listing 10.7:

```
---
Category:
  Category_1:
    title: Categories # the root node
    children:
      Category_2:
        title: Category 1
      Category_3:
        title: Category 2
        children:
          Category_4:
            title: Subcategory of Category 2
```

10.5 Fixtures For I18n

Listing 10.8:

```
Article:
  Article_1:
    name: Test article
    Translation:
      en:
```

```
      title: Title of article
      body: Body of article
    fr:
      title: French title of article
      body: French body of article
```

Chapter 11

DQL (Doctrine Query Language)

11.1 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).

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 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:

Listing 11.1:

```
<?php

// DO NOT USE THE FOLLOWING CODE
// (uses many sql queries for object population)

$users = Doctrine::getTable('User')->findAll();

foreach($users as $user) {
    print $user->name . ' has phonenumbers: ';
```

```
    foreach($user->Phonenumber as $phonenumber) {
        print $phonenumber . ' ';
    }
}

?>
```

However you should not use it. Below is the same behaviour implemented much more efficiently:

Listing 11.2:

```
<?php

// same thing implemented much more efficiently:
// (using only one sql query for object population)

$users = Doctrine_Query::create()
          ->from('User u')
          ->leftJoin('u.Phonenumber p')
          ->execute();

foreach($users as $user) {
    print $user->name . ' has phonenumbers: ';

    foreach($user->Phonenumber as $phonenumber) {
        print $phonenumber . ' ';
    }
}

?>
```

11.2 SELECT queries

SELECT statement syntax:

Listing 11.3:

```
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> OFFSET <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.

Listing 11.4:

```
SELECT a.name, a.amount FROM Account a
```

- 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).

Listing 11.5:
```
SELECT a.* FROM Account a
```

- `FROM` clause `components` indicates the component or components from which to retrieve records.

Listing 11.6:
```
SELECT a.* FROM Account a

SELECT u.*, p.*, g.* FROM User u LEFT JOIN u.Phonenumber p LEFT JOIN u.Group g
```

- 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.

Listing 11.7:
```
SELECT a.* 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

Listing 11.8:
```
SELECT u.* FROM User u LEFT JOIN u.Phonenumber p HAVING COUNT(p.id) > 3
```

- The `ORDER BY` clause can be used for sorting the results

Listing 11.9:
```
SELECT u.* FROM User u ORDER BY u.name
```

- The `LIMIT` and `OFFSET` clauses can be used for efficiently limiting the number of records to a given `row_count`

Listing 11.10:
```
SELECT u.* FROM User u LIMIT 20
```

11.2.1 DISTINCT keyword

11.2.2 Aggregate values

Aggregate value `SELECT` syntax:

Listing 11.11:

```
<?php

// SELECT u.*, COUNT(p.id) num_posts FROM User u, u.Posts p WHERE u.id = 1 GROUP
     BY u.id

$query = Doctrine_Query::create();

$query->select('u.id, COUNT(p.id) num_posts')
      ->from('User u, u.Posts p')
      ->where('u.id = ?', 1)
      ->groupby('u.id');

$users = $query->execute();

echo $users->Posts[0]->num_posts . ' posts found';

?>
```

11.3 UPDATE queries

`UPDATE` statement syntax:

Listing 11.12:

```
UPDATE //component_name//
    SET //col_name1//=//expr1// [, //col_name2//=//expr2// ...]
    [WHERE //where_condition//]
    [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.

Listing 11.13:

```
<?php

$q = Doctrine_Query::create();

$rows = $q->update('Account')
          ->set('amount', 'amount + ?', '200')
          ->where('id > 200')
          ->execute();

print $rows; // the number of affected rows

?>
```

11.4 DELETE queries

Listing 11.14:

```
DELETE FROM <component_name>
    [WHERE <where_condition>]
    [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.

Listing 11.15:

```
<?php

$q = Doctrine_Query::create();

$rows = $q->delete()
          ->from('Account a')
          ->where('a.id > ?', 3)
          ->execute();

print $rows; // the number of affected rows

?>
```

11.5 FROM clause

Syntax:

Listing 11.16:

```
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:

Listing 11.17:

```
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.

The following example shows how to fetch all records from class 'User'.

Listing 11.18:

```
<?php

$users = Doctrine_Query::create()
         ->from('User u')
         ->execute();

?>
```

11.6 JOIN syntax

DQL JOIN Syntax:

Listing 11.19:

```
[[LEFT | INNER] 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:

Listing 11.20:
```
SELECT u.*, p.* FROM User u LEFT JOIN u.Phonenumber

SELECT u.*, p.* FROM User u, u.Phonenumber p
```

The recommended form is the first one.

- `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.

Listing 11.21:
```
SELECT u.*, p.* FROM User u INNER JOIN u.Phonenumber p
```

By default DQL auto-adds the primary key join condition, so for DQL query:

Listing 11.22:
```
SELECT u.id, p.id FROM User u LEFT JOIN u.Phonenumber
```

Would have a SQL equivalent:

Listing 11.23:
```
SELECT u.id AS u__id, p.id AS p__id FROM User u LEFT JOIN Phonenumber p ON u.id
    = p.user_id
```

11.6.1 ON keyword

If you want to override this behaviour and add your own custom join condition you can do it with the `ON` keyword.
Consider the following DQL query:

Listing 11.24:
```
SELECT u.id, p.id FROM User u LEFT JOIN u.Phonenumber ON u.id = 2
```

This query would be converted into SQL:

Listing 11.25:
```
SELECT u.id AS u__id, p.id AS p__id FROM User u LEFT JOIN Phonenumber p ON u.id
    = 2
```

11.6.2 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.

DQL:

Listing 11.26:
```
SELECT u.id, p.id FROM User u LEFT JOIN u.Phonenumber WITH u.id = 2
```

SQL:

Listing 11.27:

```
SELECT u.id AS u__id, p.id AS p__id FROM User u LEFT JOIN Phonenumber p ON u.id
    = p.user_id AND u.id = 2
```

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:

Listing 11.28:

```
<?php

$q = Doctrine_Query::create();
$q->from('User u')
  ->leftJoin('u.Group g')
  ->innerJoin('u.Phonenumber p WITH u.id > 3')
  ->leftJoin('u.Email e');

$users = $q->execute();

?>
```

11.7 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 behaviour you need to use INDEXBY keyword as shown above:

Listing 11.29:

```
<?php

$q = Doctrine_Query::create();
$q->from('User u INDEXBY u.name');

$users = $q->execute();

?>
```

Now the users in $users collection are accessible through their names.

Listing 11.30:

```
<?php

print $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
behaviour. In the following we use distinct indexing for both Users and Groups.

Listing 11.31:

```
<?php
```

```
$q = Doctrine_Query::create();
$q->from('User u INDEXBY u.name')->innerJoin('u.Group g INDEXBY g.name');

$users = $q->execute();

?>
```

Now lets print out the drinkers club's creation date.

Listing 11.32:

```
<?php

print $users['jack daniels']->Group['drinkers club']->createdAt;

?>
```

11.8 WHERE clause

Syntax:

Listing 11.33:

```
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

11.9 Conditional expressions

11.9.1 Literals

Strings

A string literal is enclosed in single quotes; for example: 'literal'. A string literal that includes a single quote
is represented by two single quotes; for example: 'literal''s'.

Listing 11.34:

```
FROM User WHERE User.name = 'Vincent'
```

Integers

Integer literals support the use of PHP integer literal syntax.

Listing 11.35:

```
FROM User WHERE User.id = 4
```

Floats

Float literals support the use of PHP float literal syntax.

Listing 11.36:

```
FROM Account WHERE Account.amount = 432.123
```

Booleans

The boolean literals are true and false.

Listing 11.37:

```
FROM User WHERE User.admin = true

FROM Session WHERE Session.is_authed = false
```

Enums

The enumerated values work in the same way as string literals.

Listing 11.38:

```
FROM User WHERE User.type = 'admin'
```

Predefined reserved literals are case insensitive, although its a good standard to write them in uppercase.

11.9.2 Input parameters

Listing 11.39:

```
<?php

// POSITIONAL PARAMETERS:
$users = Doctrine_Query::create()
          ->from('User u')
          ->where('u.name = ?', array('Arnold'))
          ->execute();

$users = Doctrine_Query::create()
          ->from('User u')
          ->where('u.id > ? AND u.name LIKE ?', array(50, 'A%'))
          ->execute();

// NAMED PARAMETERS:

$users = Doctrine_Query::create()
          ->from('User u')
          ->where('u.name = :name', array(':name' => 'Arnold'))
          ->execute();

$users = Doctrine_Query::create()
          ->from('User u')
          ->where('u.id > :id AND u.name LIKE :name', array(':id' => 50, ':name'
              => 'A%'))
          ->execute();

?>
```

11.9.3 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	

11.9.4 Between expressions

11.9.5 In expressions

Syntax:

Listing 11.40:

```
<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.

Listing 11.41:

```
FROM C1 WHERE C1.col1 IN (FROM C2(col1));

FROM User WHERE User.id IN (1,3,4,5)
```

The keyword IN is an alias for = ANY. Thus, these two statements are equal:

Listing 11.42:

```
FROM C1 WHERE C1.col1 = ANY (FROM C2(col1));
FROM C1 WHERE C1.col1 IN    (FROM C2(col1));
```

11.9.6 Like Expressions

Syntax:

Listing 11.43:

```
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.

Listing 11.44:

```
<?php

// finding all users whose email ends with '@gmail.com'
$users = Doctrine_Query::create()
          ->from('User u')
          ->leftJoin('u.Email e')
          ->where('e.address LIKE ?', '%@gmail.com')
          ->execute();

// finding all users whose name starts with letter 'A'
$users = Doctrine_Query::create()
          ->from('User u')
          ->where('u.name LIKE ?', 'A%')
          ->execute();

?>
```

11.9.7 Null Comparison Expressions

11.9.8 Empty Collection Comparison Expressions

11.9.9 Collection Member Expressions

11.9.10 Exists Expressions

Syntax:

Listing 11.45:

```
<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.

Finding all articles which have readers:

Listing 11.46:

```
FROM Article a
  WHERE EXISTS (SELECT r.id FROM ReaderLog r
                WHERE r.article_id = a.id)
```

Finding all articles which don't have readers:

Listing 11.47:

```
FROM Article a
  WHERE NOT EXISTS (SELECT r.id FROM ReaderLog r
                WHERE r.article_id = a.id)
```

11.9.11 All and Any Expressions

Syntax:

Listing 11.48:

```
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.

Listing 11.49:

```
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.

Listing 11.50:

```
FROM C WHERE C.col1 > ANY (FROM C2(col1))
```

The keyword SOME is an alias for ANY.

Listing 11.51:

```
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:

Listing 11.52:

```
FROM C WHERE C.col1 <> ALL (FROM C2(col1));
FROM C WHERE C.col1 NOT IN (FROM C2(col1));
```

11.9.12 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."

Listing 11.53:

```
<?php

// finding all users which don't belong to any group 1
$users = Doctrine_Query::create()
           ->from('User u')
           ->where('u.id NOT IN (SELECT u.id FROM User u INNER JOIN u.Group g
              WHERE g.id = ?)')
           ->execute();

// finding all users which don't belong to any groups
// Notice:
// the usage of INNER JOIN
// the usage of empty brackets preceding the Group component

$users = Doctrine_Query::create()
           ->from('User u')
           ->where('u.id NOT IN (SELECT u.id FROM User u INNER JOIN u.Group g)')
           ->execute();

?>
```

11.10 Functional Expressions

11.10.1 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 firstname and lastname to a value called name

Listing 11.54:

```
<?php

$q = Doctrine_Query::create();

$users = $q->select('CONCAT(u.firstname, u.lastname) name')->from('User u')->
    execute();
```

```
foreach($users as $user) {
    // here 'name' is not a property of $user,
    // its a mapped function value
    print $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.

Listing 11.55:

```
<?php

$q = Doctrine_Query::create();

$users = $q->select('u.name')->from('User u')->where("SUBSTRING(u.name, 0, 1) =
    'z'")->execute();

foreach($users as $user) {
    print $user->name;
}

?>
```

- 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.

Listing 11.56:

```
<?php

$q = Doctrine_Query::create();

$users = $q->select('u.name')->from('User u')->where("TRIM(u.name) = 'Someone'")
    ->execute();

foreach($users as $user) {
    print $user->name;
}

?>
```

- The *LOWER* and *UPPER* functions convert a string to lower and upper case, respectively. They return a string.

Listing 11.57:

```
<?php

$q = Doctrine_Query::create();

$users = $q->select('u.name')->from('User u')->where("LOWER(u.name) = 'someone'"
    )->execute();

foreach($users as $user) {
    print $user->name;
}

?>
```

- 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.

11.10.2 Arithmetic functions

Availible DQL arithmetic functions:

Listing 11.58:

```
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.

11.10.3 Datetime functions

11.11 Subqueries

11.11.1 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.

11.11.2 Comparisons using subqueries

Find all the users which are not in a specific group.

Listing 11.59:

```
<?php

$users = Doctrine_Query::create()
          ->from('User u')
          ->where('u.id NOT IN (SELECT u.id FROM User u INNER JOIN u.Group g
              WHERE g.id = ?)')
          ->execute(1);

?>
```

11.11.3 Conditional expressions

11.11.3.1 ANY, IN and SOME

11.11.3.2 ALL

11.11.3.3 EXISTS and NOT EXISTS

11.11.4 Correlated subqueries

11.11.5 Subqueries in FROM clause

11.11.6 Subqueries in SELECT clause

Retrieve the users phonenumber in a subquery and include it in the resultset of user information.

Listing 11.60:

```
<?php

$users = Doctrine_Query::create()
          ->select('u.*, (SELECT p.phonenumber FROM Phonenumber p WHERE p.
              user_id = u.id LIMIT 1) as phonenumber')
          ->from('User u')
          ->execute();

?>
```

11.12 GROUP BY, HAVING clauses

11.13 ORDER BY clause

11.13.1 Introduction

Record collections can be sorted efficiently at the database level using the ORDER BY clause.

Syntax:

Listing 11.61:

```
[ORDER BY {ComponentAlias.columnName}
        [ASC | DESC], ...]
```

Examples:

Listing 11.62:

```
FROM User u LEFT JOIN u.Phonenumber p
  ORDER BY u.name, p.phonenumber

FROM User u, u.Email e
  ORDER BY e.address, u.id
```

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.

Listing 11.63:

```
FROM User u LEFT JOIN u.Email e
  ORDER BY e.address DESC, u.id ASC;
```

11.13.2 Sorting by an aggregate value

In the following example we fetch all users and sort those users by the number of phonenumbers they have.

Listing 11.64:

```
<?php

$q = Doctrine_Query::create();

$users = $q->select('u.*, COUNT(p.id) count')
           ->from('User u')
           ->innerJoin('u.Phonenumber p')
           ->orderby('count');

?>
```

11.13.3 Using random order

In the following example we use random in the ORDER BY clause in order to fetch random post.

Listing 11.65:

```
<?php

$q = Doctrine_Query::create();

$posts = $q->select('p.*, RANDOM() rand')
           ->from('Post p')
           ->orderby('rand')
           ->limit(1)
           ->execute();

$randomPost = $posts[0];

?>
```

11.14 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.

Listing 11.66:

```
<?php

// retrieve the first 20 users and all their associated phonenumbers

$users = Doctrine_Query::create()
          ->select('u.*, p.*')
          ->from('User u')
          ->leftJoin('u.Phonenumber p')
          ->limit(20)
          ->execute();

foreach($users as $user) {
    print ' --- '.$user->name.' --- \n';

    foreach($user->Phonenumber as $p) {
        print $p->phonenumber.'\n';
    }
}

?>
```

11.14.1 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.

11.14.2 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 behaviour can be overwritten using the configuration system (at global, connection or table level) using:

Listing 11.67:

```
<?php

$table->setAttribute(Doctrine::ATTR_QUERY_LIMIT, Doctrine::LIMIT_ROWS);
$table->setAttribute(Doctrine::ATTR_QUERY_LIMIT, Doctrine::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.

DQL QUERY:

Listing 11.68:

```
SELECT u.id, u.name, p.* FROM User u LEFT JOIN u.Phonenumber p LIMIT 20
```

SQL QUERY:

Listing 11.69:

```
SELECT
    e.id AS e__id,
    e.name AS e__name,
    p.id AS p__id,
    p.phonenumber AS p__phonenumber,
    p.entity_id AS p__entity_id
FROM entity e
LEFT JOIN phonenumber p ON e.id = p.entity_id
WHERE e.id IN (
SELECT DISTINCT e2.id
FROM entity e2
WHERE (e2.type = 0) LIMIT 20) AND (e.type = 0)
```

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.

DQL QUERY:

Listing 11.70:

```
SELECT u.id, u.name, p.* FROM User u LEFT JOIN u.Phonenumber p LIMIT 20
```

SQL QUERY:

Listing 11.71:

```
SELECT
    e.id AS e__id,
    e.name AS e__name,
    p.id AS p__id,
    p.phonenumber AS p__phonenumber,
    p.entity_id AS p__entity_id
FROM entity e
LEFT JOIN phonenumber p ON e.id = p.entity_id
WHERE e.id IN (
SELECT DISTINCT e2.id
FROM entity e2
INNER JOIN phonenumber p2 ON e2.id = p2.entity_id
WHERE (e2.type = 0) LIMIT 20) AND (e.type = 0)
```

11.15 Examples

11.16 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.

Listing 11.72:

```
<?php

$r = Doctrine_Manager::getInstance()->getQueryRegistry();
```

```
$r->add('User/all', 'FROM User u');

$userTable = Doctrine::getTable('User');

// find all users
$users = $userTable->find('all');

?>
```

To simplify this support, Doctrine_Table support some accessors to Doctrine_Query_Registry.

11.16.1 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:

Listing 11.73:

```
<?php

class MyFooTable extends Doctrine_Table
{
    public function construct()
    {
        // Named Query defined using DQL string
        $this->addNamedQuery('get.by.id', 'SELECT f.* FROM MyFoo f WHERE f.id =
            ?');

        // Named Query defined using Doctrine_Query object
        $this->addNamedQuery(
            'get.by.similar.names', Doctrine_Query::create()
                ->select('f.id, f.value0')
                ->from('MyFoo f')
                ->where('LOWER(f.name) LIKE LOWER(?)')
        );
    }
}

?>
```

11.16.2 Accessing Named Query

To reach the MyFooTable class, which is a subclass of Doctrine_Table, you can do the following:

Listing 11.74:

```
<?php

$MyFooTableInstance = Doctrine::getTable('MyFoo');

?>
```

To access the Named Query (will return you a Doctrine_Query instance, always):

Listing 11.75:

```
<?php

$query = $MyFooTableInstance->createNamedQuery('get.by.id');

?>
```

11.16.3 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:

Listing 11.76:

```
<?php

$fooItems = Doctrine::getTable('MyFoo')
    ->createNamedQuery('get.by.similar.names')
    ->execute(array('%jon%wage%'));

?>
```

You can also simplify the execution, by doing:

Listing 11.77:

```
<?php

$fooItems = Doctrine::getTable('MyFoo')
    ->find('get.by.similar.names', array('%jon%wage%'));

?>
```

The method find() also accepts a third parameter, which is the hydration mode.

11.16.4 Cross-Acessing Named Query

If that's not enough, Doctrine take advantage the Doctrine_Query_Registry and uses namespaced queries to enable
cross-access of Named Queries between objects.
Suppose you have the *Table class instance of record "MyBar". You want to call the "get.by.id" Named Query of record
"MyFoo". To acess the Named Query, you have to do:

Listing 11.78:

```
<?php

$MyBarTable = Doctrine::getTable('MyBar');

// ...

$MyFooItems = $MyBarTable->find('MyFoo/get.by.id', array(1, 2, 3), Doctrine::
   HYDRATE_RECORD);

// MyFooItems is a Doctrine_Collection, with MyFoo records (Doctrine_Record).

?>
```

11.17 BNF

Listing 11.79:

```
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
like_expression ::=
string_expression [NOT ]LIKE pattern_value [ESCAPE escape_character]
null_comparison_expression ::=
{single_valued_path_expression | input_parameter}IS [NOT ] NULL
empty_collection_comparison_expression ::=
collection_valued_path_expressionIS [NOT] EMPTY
collection_member_expression ::= entity_expression
[NOT ]MEMBER [OF ] collection_valued_path_expression
exists_expression::= [NOT ]EXISTS (subquery)
all_or_any_expression ::= {ALL |ANY |SOME } (subquery)
comparison_expression ::=
string_expression comparison_operator {string_expression | all_or_any_expression
    } |
boolean_expression {= |<> } {boolean_expression | all_or_any_expression} |
enum_expression {= |<> } {enum_expression | all_or_any_expression} |
datetime_expression comparison_operator
{datetime_expression | all_or_any_expression} |
entity_expression {= |<> } {entity_expression | all_or_any_expression} |
arithmetic_expression comparison_operator
{arithmetic_expression | all_or_any_expression}
comparison_operator ::== |> |>= |< |<= |<>
arithmetic_expression ::= simple_arithmetic_expression | (subquery)
simple_arithmetic_expression ::=
```

```
arithmetic_term | simple_arithmetic_expression {+ |- } arithmetic_term
arithmetic_term ::= arithmetic_factor | arithmetic_term {* |/ }
    arithmetic_factor
arithmetic_factor ::= [{+ |- }] arithmetic_primary
arithmetic_primary ::=
state_field_path_expression |
numeric_literal |
(simple_arithmetic_expression) |
input_parameter |
functions_returning_numerics |
aggregate_expression
string_expression ::= string_primary | (subquery)
string_primary ::=
state_field_path_expression |
string_literal |
input_parameter |
functions_returning_strings |
aggregate_expression

datetime_expression ::= datetime_primary | (subquery)
datetime_primary ::=
state_field_path_expression |
input_parameter |
functions_returning_datetime |
aggregate_expression
boolean_expression ::= boolean_primary | (subquery)
boolean_primary ::=
state_field_path_expression |
boolean_literal |
input_parameter |
enum_expression ::= enum_primary | (subquery)
enum_primary ::=
state_field_path_expression |
enum_literal |
input_parameter |
entity_expression ::=
single_valued_association_path_expression | simple_entity_expression
simple_entity_expression ::=
identification_variable |
input_parameter
functions_returning_numerics::=
LENGTH( string_primary) |
LOCATE( string_primary, string_primary[, simple_arithmetic_expression]) |
ABS( simple_arithmetic_expression) |
SQRT( simple_arithmetic_expression) |
MOD( simple_arithmetic_expression, simple_arithmetic_expression) |
SIZE( collection_valued_path_expression)
functions_returning_datetime ::=
  CURRENT_DATE |
  CURRENT_TIME |
  CURRENT_TIMESTAMP
functions_returning_strings ::=
CONCAT( string_primary, string_primary) |
SUBSTRING( string_primary,
simple_arithmetic_expression, simple_arithmetic_expression)|
TRIM( [[trim_specification] [trim_character]FROM ] string_primary) |
LOWER( string_primary) |
UPPER( string_primary)
trim_specification ::=LEADING | TRAILING | BOTH
```

11.18 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.

Examples:

Listing 11.80:

```php
<?php

// The normal find by primary key method
$userTable = Doctrine::getTable('User');

$user = $userTable->find(1);

// Find one user by the username
$userTable = Doctrine::getTable('User');

$user = $userTable->findOneByUsername('jonwage');

// Find phonenumbers for the user above
$phoneTable = Doctrine::getTable('Phonenumber');

$phonenumbers = $phoneTable->findByUser($user);

?>
```

Chapter 12

Utilities

12.1 Pagination

12.1.1 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.

12.1.2 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:

Listing 12.1:

```php
<?php

// 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:

Listing 12.2:
```
<?php

$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:

Listing 12.3:
```
<?php

$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:

Listing 12.4:
```
<?php

$pager->setCountQuery($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:

Listing 12.5:

```
<?php

$pager->setCountQueryParams([$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:

Listing 12.6:

```
<?php

// 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());

?>
```

12.1.3 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`).

12.1.3.1 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 12.7:

```
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---|
```

12.1.3.2 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 12.8:

```
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:

Listing 12.9:

```php
<?php

$pager_range = 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:

Listing 12.10:

```php
<?php

$pager_range = $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:

Listing 12.11:

```php
<?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:

Listing 12.12:

```php
<?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();

?>
```

12.1.4 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
2 types of "variables" in `Doctrine_Pager_Layout`:

12.1.4.1 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

12.1.4.2 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:

Listing 12.13:

```php
<?php

// Creating pager layout
$pager_layout = 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
$pager_layout->setTemplate('[<a href="{%url}">{%page}</a>]');
$pager_layout->setSelectedTemplate('[{%page}]');
```

```
// Retrieving Doctrine_Pager instance
$pager = $pager_layout->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)
$pager_layout->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.

Listing 12.14:

```
<?php

// Creating pager layout
$pager_layout = new Doctrine_Pager_Layout(
    new Doctrine_Pager(
        Doctrine_Query::create()
            ->from( 'User u' )
            ->leftJoin( 'u.Group g' )
            ->where( 'LOWER(u.username) LIKE LOWER(?)', array( '%'.$_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 template processment
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.

Listing 12.15:

```php
<?php

// Assigning templates for page links creation
$pager_layout->setTemplate('[<a href="{%url}">{%page}</a>]');
$pager_layout->setSelectedTemplate('[{%page}]');

// Fetching users
$users = $pager_layout->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:

Listing 12.16:

```php
<?php

// Displaying page links
$pager_layout->display( array(
    'search' => urlencode($_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:

Listing 12.17:

```php
<?php

// Return the Pager associated to the Pager_Layout
$pager_layout->getPager();

// Return the Pager_Range associated to the Pager_Layout
$pager_layout->getPagerRange();

// Return the URL mask associated to the Pager_Layout
$pager_layout->getUrlMask();

// Return the template associated to the Pager_Layout
$pager_layout->getTemplate();

// Return the current page template associated to the Pager_Layout
$pager_layout->getSelectedTemplate();

// Defines the Separator template, applied between each page
```

```
$pager_layout->setSeparatorTemplate($separatorTemplate);

// Return the current page template associated to the Pager_Layout
$pager_layout->getSeparatorTemplate();

// Handy method to execute the query without need to retrieve the Pager instance
$pager_layout->execute($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.

12.1.5 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:

Listing 12.18:

```
<?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:

Listing 12.19:

```
<?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:

Listing 12.20:

```
<?php

// CLASS:

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:

$pager_layout = new sfDoctrinePagerLayout(
    $pager,
    new Doctrine_Pager_Range_Sliding(array('chunk' => 5)),
    '@hostHistoryList?page={%page_number}'
);

$pager_layout->setTemplate('[{link_to}{%page}{/link_to}]');

?>
```

12.2 Facade

12.2.1 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.

12.2.2 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.

Listing 12.21:

```
<?php

// Turn debug on/off and check for whether it is on/off
Doctrine::debug(true);

if (Doctrine::debug()) {
    echo 'debugging is on';
} else {
    echo 'debugging is off';
}

// Get the path to your Doctrine libraries
$path = Doctrine::getPath();

// 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
$models = Doctrine::loadModels('/path/to/models', Doctrine::
    MODEL_LOADING_CONSERVATIVE); // or Doctrine::MODEL_LOADING_AGGRESSIVE
print_r($models);

// Get array of all the models loaded and present to Doctrine
$models = Doctrine::getLoadedModels();

// Pass an array of classes to the above method and it will filter out the ones
    that are not Doctrine_Records
$models = Doctrine::filterInvalidModels(array('User', 'Formatter', '
    Doctrine_Record'));
print_r($models); // would return array('User') because Formatter and
    Doctrine_Record are not valid

// Get Doctrine_Connection object for an actual table name
$conn = Doctrine::getConnectionByTableName('user'); // returns the connection
    object that the table name is associated
with.

// Generate YAML schema from an existing database
Doctrine::generateYamlFromDb('/path/to/dump/schema.yml', array('connection_name'
    ), $options);

// Generate your models from an existing database
Doctrine::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::generateModelsFromYaml('/path/to/schema.yml', '/path/to/generate/
    models', $options);

// Create the tables supplied in the array
Doctrine::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::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::generateSqlFromModels('/path/to/models');

// Generate array of sql statements to create the array of passed models
Doctrine::generateSqlFromArray(array('User', 'Phonenumber'));

// Generate YAML schema from an existing set of models
Doctrine::generateYamlFromModels('/path/to/schema.yml', '/path/to/models');

// Create all databases for connections.
// Array of connection names is optional
Doctrine::createDatabases(array('connection_name'));

// Drop all databases for connections
// Array of connection names is optional
Doctrine::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 need
// to specify a folder instead of a file.
Doctrine::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 loading
Doctrine::loadData('/path/to/fixture/files', true);

// Run a migration process for a set of migration classes
$num = 5; // migrate to version #5
Doctrine::migration('/path/to/migrations', $num);

// Generate a blank migration class template
Doctrine::generateMigrationClass('ClassName', '/path/to/migrations');

// Generate all migration classes for an existing database
Doctrine::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::generateMigrationsFromModels('/path/to/migrations', '/path/to/models')
    ;

// Get Doctrine_Table instance for a model
$userTable = Doctrine::getTable('User');

// Compile doctrine in to a single php file
$drivers = array('mysql'); // specify the array of drivers you want to include
    in this compiled version
Doctrine::compile('/path/to/write/compiled/doctrine', $drivers);

// Dump doctrine objects for debugging
$conn = Doctrine_Manager::connection();
Doctrine::dump($conn);
```

```
?>
```

12.2.3 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.

Listing 12.22:

```
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.

12.3 Command Line Interface

12.3.1 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.

12.3.2 Tasks

Below is a list of available tasks for managing your Doctrine implementation.

Listing 12.23:

```
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.

Listing 12.24:

```
<?php

$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());
}

?>
```

12.3.3 Usage

File named "doctrine" that is set to executable

Listing 12.25:

```
#!/usr/bin/env php
<?php
chdir(dirname(__FILE__));
include('doctrine.php');
```

Actual php file named "doctrine.php" that implements the Doctrine_Cli.

Listing 12.26:

```
<?php

// 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
```

```
required for you to enter them.

$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.

Listing 12.27:

```
./doctrine generate-models-yaml
./doctrine create-tables
```

12.4 Sandbox

12.4.1 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.

Listing 12.28:

```
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.

Chapter 13

Native SQL

13.1 Introduction

Doctrine_RawSql provides convient interface for building raw sql queries. Similar to Doctrine_Query, Doctrine_RawSql
provides means for fetching arrays and objects, the 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 Doctrine_RawSql object is easy:

Listing 13.1:

```
<?php

$q = new Doctrine_RawSql();

?>
```

Optionally a connection parameter can be given:

Listing 13.2:

```
<?php

$q = new Doctrine_RawSql($conn); // here $conn is an instance of
    Doctrine_Connection

?>
```

13.2 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:

Listing 13.3:

```
<?php

$q = new Doctrine_RawSql();
```

```
$q->select('{u.*}')
  ->from('user')
  ->addComponent('user', 'User'); // here we tell that user table is bound to
      class called 'User'

$users = $q->execute();
$users[0]; // User object

?>
```

Pay attention to following things:

1. Fields must be in curly brackets
2. For every selected table there must be one addComponent call

13.3 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.

Consider the following model:

Listing 13.4:

```
<?php

// file User.php
class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', 20);
    }
    public function setUp()
    {
        $this->hasMany('Phonenumber', array('local' => 'id',
                                            'foreign' => 'user_id'));
    }
}
// file Phonenumber.php
class Phonenumber extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('phonenumber', 'string', 20);
        $this->hasColumn('user_id', 'integer');
    }
    public function setUp()
    {
        $this->hasOne('User', array('local' => 'user_id',
                                    'foreign' => 'id',
                                    'onDelete' => 'CASCADE'));
    }
}

?>
```

In the following example we fetch all users and their phonenumbers:

Listing 13.5:

```php
<?php

$q = new Doctrine_RawSql();

$q->select('{u.*}, {p.*}')
  ->from('user u LEFT JOIN phonenumber p ON u.id = p.user_id')
  // here 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 to User class
  ->addComponent('u', 'User u')
  // here we add another component that is bound to table phonenumber
  // notice how we reference that the Phonenumber class is "User's phonenumber"
  ->addComponent('p', 'u.Phonenumber p');

$users = $q->execute();
$users[0]; // User object

?>
```

Chapter 14

Transactions

14.1 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[1] 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[2] 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[3] 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,

---

1 http://en.wikipedia.org/wiki/Atomicity
2 http://en.wikipedia.org/wiki/Database_consistency
3 http://en.wikipedia.org/wiki/Isolation_%28computer_science%29

isolation means the transaction history (or schedule[4]) is
serializable[5]. For performance reasons, this ability is the most often
relaxed constraint. See the isolation[6] article for more
details.

- Durability[7] 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[8] has checked the integrity constraints and won't
need to abort the transaction. Typically, all transactions are written into a
log[9] 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.

- *from wikipedia*[10]

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.

Listing 14.1:

```php
<?php

$conn->beginTransaction();

$user = new User();
$user->name = 'New user';
$user->save();

$user = Doctrine::getTable('User')->find(5);
$user->name = 'Modified user';
$user->save();

$conn->commit(); // all the queries are executed here

?>
```

[4] http://en.wikipedia.org/wiki/Schedule_%28computer_science%29
[5] http://en.wikipedia.org/wiki/Serializability
[6] http://en.wikipedia.org/wiki/Isolation_%28computer_science%29
[7] http://en.wikipedia.org/wiki/Durability_%28computer_science%29
[8] http://en.wikipedia.org/wiki/Database_system
[9] http://en.wikipedia.org/wiki/Database_log
[10] http://www.wikipedia.org

14.2 Nesting

Listing 14.2:

```
<?php

function saveUserAndGroup(Doctrine_Connection $conn, User $user, Group $group) {
    $conn->beginTransaction();

    $user->save();

    $group->save();

    $conn->commit();
}

try {
    $conn->beginTransaction();

    saveUserAndGroup($conn,$user,$group);
    saveUserAndGroup($conn,$user2,$group2);
    saveUserAndGroup($conn,$user3,$group3);

    $conn->commit();
} catch(Doctrine_Exception $e) {
    $conn->rollback();
}

?>
```

14.3 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.

Listing 14.3:

```
<?php

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.

NOTE: 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.

Listing 14.4:

```
<?php

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
}

?>
```

14.4 Locking strategies

14.4.1 Pessimistic locking

14.4.2 Optimistic locking

14.5 Lock modes

14.6 Isolation levels

A transaction isolation level sets the default transactional behaviour. 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.

Listing 14.5:

```
<?php

$tx = $conn->transaction; // get the transaction module

// sets the isolation level to READ COMMITTED
$tx->setIsolation('READ COMMITTED');

// sets 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();

?>
```

14.7 Deadlocks

Chapter 15

Caching

15.1 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.

Initializing a new cache driver instance:

Listing 15.1:

```
<?php

$cacheDriver = new Doctrine_Cache_Memcache($options);

?>
```

15.2 Drivers

15.2.1 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.

Listing 15.2:

```
<?php

// memcache allows multiple servers
$servers = array('host' => 'localhost',
                 'port' => 11211,
                 'persistent' => true);

$cacheDriver = new Doctrine_Cache_Memcache(array('servers' => $servers,
                                                  'compression' => false));

?>
```

Available options for Memcache driver:

Option	Data Type	Default Value	Description
servers	array	array(array('host' => 'localhost','port' => 11211, 'persistent' => true))	

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

15.2.2 APC

The Alternative PHP Cache (APC) is a free and open opcode cache for PHP. It was conceived of 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.

Listing 15.3:

```
<?php

$cacheDriver = new Doctrine_Cache_Apc();

?>
```

15.2.3 Db

Db caching backend stores cache records into given database. Usually some fast flat-file based database is used (such as
sqlite).

Initializing sqlite cache driver can be done as above:

Listing 15.4:

```
<?php

$conn  = Doctrine_Manager::connection(new PDO('sqlite::memory:'));

$cacheDriver = new Doctrine_Cache_Sqlite(array('connection' => $conn));

?>
```

15.3 Query Cache & Result Cache

15.3.1 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:

1. Init new DQL query
2. Parse DQL query
3. Build database specific SQL query
4. Execute the SQL query
5. Build the result set
6. 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:

1. Init new DQL query
2. Execute the SQL query (grabbed from the cache)
3. Build the result set
4. 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 therefore always use it 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 is has been modified and will therefore create a new cache entry, so you
dont even need to invalidate the cache. It's worth noting that the effectiveness of the query cache greatly relies on
the usage of prepared staments (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):

1. Init new DQL query
2. 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.

15.3.2 Query Cache

15.3.2.1 Using the query cache

You can set a connection or manager level query cache driver by using Doctrine::ATTR_QUERY_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 query cache driver:

Listing 15.5:

```php
<?php

$manager = Doctrine_Manager::getInstance();

$manager->setAttribute(Doctrine::ATTR_QUERY_CACHE, $cacheDriver);

?>
```

Setting a connection level cache driver:

Listing 15.6:

```php
<?php

$manager = Doctrine_Manager::getInstance();
$conn    = $manager->openConnection('pgsql://user:pass@localhost/test');

$conn->setAttribute(Doctrine::ATTR_QUERY_CACHE, $cacheDriver);

?>
```

15.3.2.2 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 with a valid cacheDriver. This rarely makes sense for the query cache
but is possible:

Listing 15.7:

```php
<?php

$query = Doctrine_Query::create()
           ->useQueryCache(new Doctrine_Cache_Apc());

?>
```

15.3.3 Result Cache

15.3.3.1 Using the result cache

You can set a connection or manager level result cache driver by using Doctrine::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:

Listing 15.8:

```
<?php

$manager = Doctrine_Manager::getInstance();

$manager->setAttribute(Doctrine::ATTR_RESULT_CACHE, $cacheDriver);

?>
```

Setting a connection level cache driver:

Listing 15.9:

```
<?php

$manager = Doctrine_Manager::getInstance();
$conn    = $manager->openConnection('pgsql://user:pass@localhost/test');

$conn->setAttribute(Doctrine::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::ATTR_RESULT_CACHE_LIFESPAN.

Listing 15.10:

```
<?php

$manager = Doctrine_Manager::getInstance();

// set the lifespan as one hour (60 seconds * 60 minutes = 1 hour = 3600 secs)
$manager->setAttribute(Doctrine::ATTR_RESULT_CACHE_LIFESPAN, 3600);

?>
```

Now as we have set a cache driver for use we can make a DQL query to use it:

Listing 15.11:

```
<?php

$query = Doctrine_Query::create();

// fetch blog titles and the number of comments
$query->select('b.title, COUNT(c.id) count')
      ->from('Blog b')
      ->leftJoin('b.Comments c')
      ->limit(10)
      ->useResultCache(true);

$entries = $query->execute();

?>
```

15.3.3.2 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 with a valid cacheDriver:

Listing 15.12:

```php
<?php

$query = Doctrine_Query::create();

$query->useResultCache(new Doctrine_Cache_Apc());

?>
```

Also you can override the lifespan attribute by calling setResultCacheLifeSpan():

Listing 15.13:

```php
<?php

$query = Doctrine_Query::create();

// set the lifespan as half an hour
$query->setResultCacheLifeSpan(60 * 30);

?>
```

Chapter 16

Event listeners

16.1 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 eventlisteners but they lack the separation of different aspects. An example of using
Doctrine_Record hooks:

Listing 16.1:

```php
<?php

class Blog extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 200);
        $this->hasColumn('content', 'string');
        $this->hasColumn('created', 'date');
    }
    public function preInsert($event)
    {
        $this->created = date('Y-m-d', time());
    }
}

// initialize connection etc, then:

$blog = new Blog();
$blog->title = 'New title';
$blog->content = 'Some content';
$blog->save();

$blog->created; // 2007-06-20 (format: YYYY-MM-DD)

?>
```

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
the name of the created savepoint, which is quite intuitively named as savepoint.

16.2 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.

16.2.1 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:

Listing 16.2:

```
<?php

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.

Listing 16.3:

```
<?php

class MyListener implements Doctrine_EventListener_Interface
{
    // notice: all listener methods must be defined here
    // (otherwise PHP throws fatal error)

    public function preExec(Doctrine_Event $event)
    { }
    public function postExec(Doctrine_Event $event)
    { }

    // ...
}
```

```
?>
```

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.

Listing 16.4:

```
<?php

class MyDebugger implements Doctrine_Overloadable
{
    public function __call($methodName, $args)
    {
        print $methodName . ' called !';
    }
}

?>
```

16.2.2 Attaching listeners

You can attach the listeners to a connection with setListener().

Listing 16.5:

```
<?php

$conn->setListener(new MyDebugger());

?>
```

If you need to use multiple listeners you can use addListener().

Listing 16.6:

```
<?php

$conn->addListener(new MyDebugger());
$conn->addListener(new MyLogger());

?>
```

16.2.3 preConnect, postConnect

16.2.4 Transaction listeners

Methods	Listens	Params
preTransactionBegin(Doctrine_Event $event)	Doctrine_Transaction::beginTransaction()	
postTransactionBegin(Doctrine_Event $event)	Doctrine_Transaction::beginTransaction()	
preTransactionRollback(Doctrine_Event $event)	Doctrine_Transaction::rollback()	
postTransactionRollback(Doctrine_Event $event)	Doctrine_Transaction::rollback()	

preTransactionCommit(Doctrine_Event $event)	Doctrine_Transaction::commit()	
postTransactionCommit(Doctrine_Event $event)	Doctrine_Transaction::commit()	
preCreateSavepoint(Doctrine_Event $event)	Doctrine_Transaction::createSavepoint()	savepoint
postCreateSavepoint(Doctrine_Event $event)	Doctrine_Transaction::createSavepoint()	savepoint
preRollbackSavepoint(Doctrine_Event $event)	Doctrine_Transaction::rollbackSavepoint()	savepoint
postRollbackSavepoint(Doctrine_Event $event)	Doctrine_Transaction::rollbackSavepoint()	savepoint
preReleaseSavepoint(Doctrine_Event $event)	Doctrine_Transaction::releaseSavepoint()	savepoint
postReleaseSavepoint(Doctrine_Event $event)	Doctrine_Transaction::releaseSavepoint()	savepoint

Listing 16.7:

```
<?php

class MyTransactionListener extends Doctrine_EventListener
{
    public function preTransactionBegin(Doctrine_Event $event)
    {
        print 'beginning transaction... ';
    }

    public function preTransactionRollback(Doctrine_Event $event)
    {
        print 'rolling back transaction... ';
    }
}

?>
```

16.2.5 Query execution listeners

Methods	Listens	Params
prePrepare(Doctrine_Event $event)	Doctrine_Connection::prepare()	query
postPrepare(Doctrine_Event $event)	Doctrine_Connection::prepare()	query
preExec(Doctrine_Event $event)	Doctrine_Connection::exec()	query
postExec(Doctrine_Event $event)	Doctrine_Connection::exec()	query, rows
preStmtExecute(Doctrine_Event $event)	Doctrine_Connection_Statement::execute()	query
postStmtExecute(Doctrine_Event $event)	Doctrine_Connection_Statement::execute()	query
preExecute(Doctrine_Event $event)	Doctrine_Connection::execute() *	query
postExecute(Doctrine_Event $event)	Doctrine_Connection::execute() *	query
preFetch(Doctrine_Event $event)	Doctrine_Connection::fetch()	query, data

postFetch(Doctrine_Event $event)	Doctrine_Connection::fetch()	query, data
preFetchAll(Doctrine_Event $event)	Doctrine_Connection::fetchAll()	query, data
postFetchAll(Doctrine_Event $event)	Doctrine_Connection::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.

16.3 Query listeners

The query listeners can be used for listening the DQL query building and resultset hydration procedures. Couple of
methods exist for listening 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: firstname, lastname and age. In the following example
we create a listener that always builds a generated field called fullname based on firstname and lastname fields.

Listing 16.8:

```
<?php

class HydrationListener extends Doctrine_Record_Listener
{
    public function preHydrate(Doctrine_Event $event)
    {
        $data = $event->data;

        $data['fullname'] = $data['firstname'] . ' ' . $data['lastname'];
        $event->data = $data;
    }
}

?>
```

Now all we need to do is attach this listener to the User record and fetch some users.

Listing 16.9:

```
<?php

$user = new User();
$user->addListener(new HydrationListener());

$users = Doctrine_Query::create()
          ->from('User')
```

```
        ->execute();

foreach ($users as $user) {
    print $user->fullname;
}

?>
```

16.4 Record listeners

Doctrine_Record provides listeners very similar to Doctrine_Connection. You can set the listeners at global, connection
and record(=table) level.

Here is a list of all available listener methods:

Methods	Listens
preSave(Doctrine_Event $event)	Doctrine_Record::save()
postSave(Doctrine_Event $event)	Doctrine_Record::save()
preUpdate(Doctrine_Event $event)	Doctrine_Record::save() when the record state is DIRTY
preDqlUpdate(Doctrine_Event $event)	Doctrine_Query::create()->update('User')->set('name', '?', 'jwage')->execute()
postUpdate(Doctrine_Event $event)	Doctrine_Record::save() when the record state is DIRTY
preInsert(Doctrine_Event $event)	Doctrine_Record::save() when the record state is TDIRTY
postInsert(Doctrine_Event $event)	Doctrine_Record::save() when the record state is TDIRTY
preDelete(Doctrine_Event $event)	Doctrine_Record::delete()
preDqlDelete(Doctrine_Event $event)	Doctrine_Query::create()->delete()->from('User')->execute()
postDelete(Doctrine_Event $event)	Doctrine_Record::delete()
preValidate(Doctrine_Event $event)	Doctrine_Validator::validate()
postValidate(Doctrine_Event $event)	Doctrine_Validator::validate()
preDqlSelect(Doctrine_Event $event)	Doctrine_Query::create()->from('User u')->execute()

Just like with connection listeners there are three ways of defining a record listener: by extending Doctrine_Record_Listener, by implement Doctrine_Record_Listener_Interface or by implementing Doctrine_Overloadable.
In the following we'll create a global level listener by implementing Doctrine_Overloadable:

Listing 16.10:

```
<?php

class Logger extends Doctrine_Overloadable
{
    public function __call($m, $a)
    {
        print 'catched event ' . $m;

        // do some logging here...
    }
}
```

```
?>
```

Attaching the listener to manager is easy:

Listing 16.11:

```
<?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:

Listing 16.12:

```
<?php

class Debugger extends Doctrine_Record_Listener
{
    public function preInsert(Doctrine_Event $event)
    {
        print 'inserting a record ...';
    }
    public function preUpdate(Doctrine_Event $event)
    {
        print 'updating a record...';
    }
}

?>
```

Attaching the listener to a connection is as easy as:

Listing 16.13:

```
<?php

$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:

Listing 16.14:

```
<?php

class Debugger extends Doctrine_Record_Listener
{
    public function postDelete(Doctrine_Event $event)
    {
        print 'deleted ' . $event->getInvoker()->id;
    }
}

?>
```

Attaching this listener to given table can be done as follows:

Listing 16.15:

```
<?php

class MyRecord extends Doctrine_Record
{
    public function setTableDefinition()
    {
        // some definitions
    }

    public function setUp()
    {
        $this->addListener(new Debugger());
    }
}

?>
```

16.5 Record hooks

Methods	Listens
preSave(Doctrine_Event $event)	Doctrine_Record::save()
postSave(Doctrine_Event $event)	Doctrine_Record::save()
preUpdate(Doctrine_Event $event)	Doctrine_Record::save() when the record state is DIRTY
preDqlUpdate(Doctrine_Event $event)	Doctrine_Query::create()->update('User')->set('name', '?', 'jwage')->execute()
postUpdate(Doctrine_Event $event)	Doctrine_Record::save() when the record state is DIRTY
preInsert(Doctrine_Event $event)	Doctrine_Record::save() when the record state is TDIRTY
postInsert(Doctrine_Event $event)	Doctrine_Record::save() when the record state is TDIRTY
preDelete(Doctrine_Event $event)	Doctrine_Record::delete()
preDqlDelete(Doctrine_Event $event)	Doctrine_Query::create()->delete()->from('User')->execute()
postDelete(Doctrine_Event $event)	Doctrine_Record::delete()
preValidate(Doctrine_Event $event)	Doctrine_Validator::validate()
postValidate(Doctrine_Event $event)	Doctrine_Validator::validate()
preDqlSelect(Doctrine_Event $event)	Doctrine_Query::create()->from('User u')->execute()

Example 1. Using insert and update hooks

Listing 16.16:

```
<?php

class Blog 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());
    }
}

?>
```

16.6 Chaining listeners

Doctrine allows chaining of different eventlisteners. This means that more than one listener can be attached for
listening the same events. The following example attaches two listeners for given connection:

Listing 16.17:

```
<?php

// here Debugger and Logger both inherit Doctrine_EventListener

$conn->addListener(new Debugger());
$conn->addListener(new Logger());

?>
```

16.7 The Event object

16.7.1 Getting the invoker

You can get the object that invoked the event by calling getInvoker():

Listing 16.18:

```
<?php

class MyListener extends Doctrine_EventListener
{
    public function preExec(Doctrine_Event $event)
    {
        $event->getInvoker(); // Doctrine_Connection
    }
}

?>
```

16.7.2 Event codes

Doctrine_Event uses constants as event codes. Above 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

Listing 16.19:

```
<?php

class MyListener extends Doctrine_EventListener
{
    public function preExec(Doctrine_Event $event)
    {
        $event->getCode(); // Doctrine_Event::CONN_EXEC
    }
}

?>
```

- 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

Listing 16.20:

```
<?php

class MyRecord extends Doctrine_Record
{
    public function preUpdate(Doctrine_Event $event)
    {
        $event->getCode(); // Doctrine_Event::RECORD_UPDATE
    }
}

?>
```

16.7.3 getInvoker()

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. Example:

Listing 16.21:

```
<?php

class MyRecord extends Doctrine_Record
{
    public function preUpdate(Doctrine_Event $event)
    {
        $event->getInvoker(); // Object(MyRecord)
    }
}

?>
```

16.7.4 skipOperation()

Doctrine_Event provides many methods for altering the execution of the listened method as well as for altering the
behaviour 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):

Listing 16.22:

```
<?php

class MyListener extends Doctrine_EventListener
{
    public function preExec(Doctrine_Event $event)
    {
        // some business logic, then:

        $event->skipOperation();
    }
}

?>
```

Query

16.7.5 skipNextListener()

When using a chain of listeners you might want to skip the execution of the next listener. It can be achieved as follows:

Listing 16.23:

```
<?php

class MyListener extends Doctrine_EventListener
{
    public function preExec(Doctrine_Event $event)
    {
```

```
        // some business logic, then:

        $event->skipNextListener();
    }
}

?>
```

16.8 DQL Query Listeners

Doctrine allows you to attach record listeners globally, on each connection, or on specific record instances. Doctrine_Query implements post/pre DQL 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 involed in the "from" part of the query for any hooks which can alter any part of the
query.

Below is an example record listener attached directly to the model which will implement the SoftDelete
functionality for the User model.

Note: 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.

Listing 16.24:

```
<?php

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';
        $query = $event->getQuery();
        if ( ! $query->contains($field)) {
            $query->from('')->update($params['component'] . ' ' . $params['alias
                ']);
            $query->set($field, '?', array(false));
            $query->addWhere($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';
        $query = $event->getQuery();
        if ( ! $query->contains($field)) {
            $query->addWhere($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 also.

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 255);
        $this->hasColumn('password', 'string', 255);
        $this->hasColumn('deleted', 'boolean', 1);
    }

    public function setUp()
    {
        $this->addListener(new UserListener());
    }

    // These methods are options because they are in the listener now, but could
        be moved here
    // 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. You can turn it on with the following attribute:

Listing 16.25:

```
<?php

Doctrine_Manager::getInstance()->setAttribute('use_dql_callbacks', true);

?>
```

Now when you interact with the User model it will take in to account the deleted flag.

Listing 16.26:

```
<?php

// Delete user with object
$user = new User();
$user->username = 'jwage';
$user->password = 'changeme';
$user->save();   // results in $user->id == 1
$user->delete(); // Does not delete record, instead sets deleted flag to 1

// Query for user and deleted flag is included in the where condition
    automatically
$q = Doctrine_Query::create()->from('User u');

echo $q->getSql();
// 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 = ?

?>
```

Note the "u.deleted = ?" was automatically added to the where condition with a parameter value of true.

Chapter 17

Behaviors

17.1 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 refactoring the
code is having a base class with some classes extending it.

However inheritance solves only a fraction of things. The following subchapters show how many times using
Doctrine_Template is much more powerful and flexible than using inheritance.

Doctrine_Template is a class templating 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.

17.2 Core Behaviors

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_Records or you can specify them in your
yaml schema if you are managing your models with a yaml schema file.

17.2.1 Versionable

Listing 17.1:

```php
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 125);
        $this->hasColumn('password', 'string', 255);
    }
```

```
    public function setUp()
    {
        $this->actAs('Versionable', array('versionColumn' => 'version', '
            className' => '%CLASS%Version'));
    }
}

?>
```

Listing 17.2:

```
---
User:
  actAs:
    Versionable:
      versionColumn: version
      className: %CLASS%Version
      auditLog: true # Can be used to optionally turn off the audit log history
          table
  columns:
    username:
      type: string(125)
    password:
      type: string(255)
```

17.2.2 Timestampable

The 2nd argument array is not required. It defaults to all the values that are present in the example below.

Listing 17.3:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 125);
        $this->hasColumn('password', 'string', 255);
    }

    public function setUp()
    {
        $options = array('created' =>  array('name'          => 'created_at',
                // Name of created column
                                             'type'          => 'timestamp',
                                                        // Doctrine column data
                                                 type
                                             'options'       => array(),
                                                            // Array of options for
                                                  column
                                             'format'        => 'Y-m-d H:i:s',
                                                      // Format of date used with
                                                 PHP date() function(default)
                                             'disabled'      => false,
                                                                // Disable the
                                                 created column(default)
                                             'expression'    => 'NOW()'),
                                                          // Update column with
                                                 database expression(default=
                                                 false)
                         'updated' =>  array('name'          => 'updated_at',
                              // Name of updated column(default)
```

```
                                                'type'          => 'timestamp',
                                                        // Doctrine column data
                                                    type(default)
                                                'options'       => array(),
                                                            // Array of options for
                                                     column(default)
                                                'format'        => 'Y-m-d H:i:s',
                                                        // Format of date used with
                                                    PHP date() function(default)
                                                'disabled'      => false,
                                                            // Disable the
                                                    updated column(default)
                                                'expression'    => 'NOW()',
                                                            // Use a database
                                                    expression to set column(
                                                    default=false)
                                                'onInsert'      => true));
                                                            // Whether or not to
                                                    set column onInsert(default)

        $this->actAs('Timestampable', $options);
    }
}

?>
```

Listing 17.4:

```
---
User:
  actAs:
    Timestampable:
      created:
        name: created_at
        type: timestamp
        format:Y-m-d H:i:s
      updated:
        name: updated_at
        type: timestamp
        format: Y-m-d H:i:s
  columns:
    username:
      type: string(125)
    password:
      type: string(255)
```

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 flag disabled=>true for either of the fields as in the example below.

Listing 17.5:

```
---
User:
  actAs:
    Timestampable:
      created:
        name: created_at
        type: timestamp
        format:Y-m-d H:i:s
      updated:
        disabled: true
  columns:
    username:
      type: string(125)
    password:
```

```
      type: string(255)
```

17.2.3 Sluggable

If you do not specify the columns to create the slug from, it will default to just using the __toString() method on the
model.

Listing 17.6:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 125);
        $this->hasColumn('password', 'string', 255);
    }

    public function setUp()
    {
        $this->actAs('Sluggable', array('unique'    => true,
                                        'fields'    => array('username'),
                                        'canUpdate' => true));
    }
}

?>
```

Listing 17.7:

```
---
User:
  actAs:
    Sluggable:
      unique: true
      fields: [username]
      canUpdate: true
  columns:
    username:
      type: string(125)
    password:
      type: string(255)
```

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.

17.2.4 I18n

Doctrine_I18n package is a plugin 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:

Listing 17.8:

```
<?php

class NewsItem extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 200);
        $this->hasColumn('content', 'string');
    }

    public function setUp()
    {
        $this->actAs('I18n', array('fields' => array('title')));
    }
}

?>
```

Now the first time you initialize a new NewsItem record Doctrine initializes the plugin that builds the followings
things:

1. Record class called NewsItemTranslation
2. Bi-directional relations between NewsItemTranslation and NewsItem

17.2.4.1 Creating the I18n table

The I18n table can be created as follows:

Listing 17.9:

```
<?php

$conn->setAttribute(Doctrine::ATTR_EXPORT, Doctrine::EXPORT_ALL);

$conn->export->exportClasses(array('NewsItem'));

?>
```

The following code example executes two sql statements. When using mysql those statements would look like:

Listing 17.10:

```
CREATE TABLE news_item (id INT NOT NULL AUTO_INCREMENT, content TEXT)
CREATE TABLE news_item_translation (id INT NOT NULL, title VARCHAR(200), lang
    VARCHAR(20))
```

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.

17.2.4.2 Using I18n

In the following example we add some data with finnish and english translations:

Listing 17.11:

```
<?php

$item = new NewsItem();
$item->content = 'This is some content. This field is not being translated.';

$item->Translation['FI']->title = 'Joku otsikko';
$item->Translation['EN']->title = 'Some title';
$item->save();

?>
```

Now lets find all items and their finnish translations:

Listing 17.12:

```
<?php

$items = Doctrine_Query::create()
         ->from('NewsItem n')
         ->leftJoin('n.Translation t INDEXBY t.lang')
         ->where('t.lang = ?')
         ->execute(array('FI'));

$items[0]->Translation['FI']->title; // 'joku otsikko'

?>
```

17.2.5 NestedSet

Listing 17.13:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 125);
        $this->hasColumn('password', 'string', 255);
    }

    public function setUp()
    {
        $this->actAs('NestedSet', array('hasManyRoots' => true, 'rootColumnName'
            => 'root_id'));
    }
}

?>
```

Listing 17.14:

```
---
User:
  actAs:
    NestedSet:
      hasManyRoots: true
      rootColumnName: root_id
  columns:
    username:
      type: string(125)
    password:
      type: string(255)
```

17.2.6 Searchable

Listing 17.15:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 125);
        $this->hasColumn('password', 'string', 255);
    }

    public function setUp()
    {
        $this->actAs('Searchable', array('fields' => array('title', 'content')))
            ;
    }
}

?>
```

Listing 17.16:

```
---
User:
  actAs:
    Searchable:
      fields: [title, content]
  columns:
    username:
      type: string(125)
    password:
      type: string(255)
```

17.2.7 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.

Listing 17.17:

```
<?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()
  {
    parent::setUp();
    $this->actAs('Geographical');
  }
}

?>
```

Listing 17.18:

```
Zipcode:
  actAs: [Geographical]
  columns:
    zipcode: string(255)
    city: string(255)
    state: string(2)
    county: string(255)
    zip_class: string(255)
```

The geographical plugin automatically adds the latitude and longitude columns to the records used for calculating distance
between 2 records.

Usage

Listing 17.19:

```
<?php

$zipcode1 = Doctrine::getTable('Zipcode')->findOneByZipcode('37209');
$zipcode2 = Doctrine::getTable('Zipcode')->findOneByZipcode('37388');

// get distance between to zipcodes
echo $zipcode1->getDistance($zipcode2, $kilometers = false);

// Get the 50 closest zipcodes that are not in the same city
$query = $zipcode1->getDistanceQuery();
$query->orderby('miles asc');
$query->addWhere($query->getRootAlias() . '.city != ?', $zipcode1->city);
$query->limit(50);

$result = $query->execute();

foreach ($result as $zipcode) {
  echo $zipcode->city . " - " . $zipcode->miles . "<br/>"; // $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 code

Listing 17.20:

```
<?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();
}

?>
```

This chapter describes the usage of various plugins availible for Doctrine. You'll also learn how to create your own
plugins. In order to grasp the concepts of this chapter you should already be familiar with the theory behind
Doctrine_Template and Doctrine_Record_Generator. When refering to plugins we refer to class packages that use
templates, generators and listeners extensively. All the introduced components in this chapter can be considered
'core' plugins, that means they reside at the Doctrine main repository. There are other official plugins too which
can be found at the homesite of the Sensei project (www.sensei-project.org).

Usually plugins use generators side-to-side with template classes (classes that extend Doctrine_Template). The common
workflow is:

1. A new template is being initiliazed
2. The template creates the generator and calls initialize() method
3. 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.

17.2.8 Versionable

Doctrine versionable behavior provides a full versioning solution. Lets say we have a NewsItem class that we want to be versioned.
This functionality can be applied by simply adding $this->actAs('Versionable') into your record setup.

Listing 17.21:

```
<?php

class NewsItem extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 200);
        $this->hasColumn('content', 'string');
        // the versioning plugin needs version column
        $this->hasColumn('version', 'integer');
    }

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

?>
```

Now when we have defined this record to be versionable, Doctrine does internally the following things:

- It creates a class called NewsItemVersion on-the-fly, the table this record is pointing at is news_item_version
- Everytime a NewsItem object is deleted / updated the previous version is stored into news_item_version
- Everytime a NewsItem object is updated its version number is increased.

17.2.8.1 Creating the version table

As with all other plugins, the plugin-table, in this case the table that holds the different versions, can be created
by enabling Doctrine::EXPORT_PLUGINS. The easiest way to set this is by setting the value of Doctrine::ATTR_EXPORT to
Doctrine::EXPORT_ALL. The following example shows the usage:

Listing 17.22:

```
<?php

$conn->setAttribute(Doctrine::ATTR_EXPORT, Doctrine::EXPORT_ALL);

$conn->export->exportClasses(array('NewsItem'));

?>
```

The following code example executes two sql statements. When using mysql those statements would look like:

Listing 17.23:

```
CREATE TABLE news_item (id INT NOT NULL AUTO_INCREMENT, title VARCHAR(200),
    content TEXT, version INTEGER)
CREATE TABLE news_item_version (id INT NOT NULL, title VARCHAR(200), content
    TEXT, version INTEGER)
```

17.2.8.2 Using versioning

Listing 17.24:

```
<?php

$newsItem = new NewsItem();
$newsItem->title = 'No news is good news';
$newsItem->content = 'All quiet on the western front';

$newsItem->save();
$newsItem->version; // 1

$newsItem->title = 'A different title';
$newsItem->save();
$newsItem->version; // 2

?>
```

17.2.8.3 Reverting changes

Doctrine_Record provides a method called revert() which can be used for reverting to specified version. Internally
Doctrine queries the version table and fetches the data for given version. If the given version is not found a
Doctrine_Record_Exception is being thrown.

Listing 17.25:

```
<?php

$newsItem->revert(1);

$newsItem->title; // No news is good news

?>
```

17.2.8.4 Advanced usage

There are many options for the versioning plugin. Sometimes you may want to use other version column than 'version'.
This can be achieved by giving the options parameter to actAs() method.

Listing 17.26:

```
<?php

class NewsItem extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 200);
        $this->hasColumn('content', 'string');
        // the versioning plugin needs version column
        $this->hasColumn('news_version', 'integer');
    }

    public function setUp()
    {
        $this->actAs('Versionable', array('versionColumn' => 'news_version'));
    }
```

```
}

?>
```

You can also control the name of the versioning record and the name of the version table with option attributes
'className' and 'tableName'.

17.2.9 Soft-delete

Soft-delete is a very simple model behavior which will overrides the delete() functionality and adds a delete
column. When delete() is called, instead of deleting the record from the database, a delete flag is set to 1.
Below is an example of how to create a model with the SoftDelete behavior being used.

Listing 17.27:

```
<?php

class SoftDeleteTest extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string', null, array('primary' => true));
    }

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

?>
```

Listing 17.28:

```
---
SoftDeleteTest:
  actAs: [SoftDelete]
  columns:
    name:
      type: string
      primary: true
```

Now lets put the plugin in action.

Note: 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 flag is set with an additional WHERE condition.

Listing 17.29:

```
<?php

// Enable dql callbacks.
Doctrine_Manager::getInstance()->setAttribute('use_dql_callbacks', true);

// save a new record
$record = new SoftDeleteTest();
```

```
$record->name = 'new record';
$record->save();

$record->delete();
var_dump($record->deleted); // true

// Querying for records excludes deleted records automatically.
$q = Doctrine_Query::create()
        ->from('SoftDeleteTest t');

echo $q->count(); // This would be 0, it would exclude the record saved above
    because the delete flag was set
// The following SQL would have been executed to get this count
// SELECT COUNT(DISTINCT s.name) AS num_results FROM soft_delete_test s WHERE s.
    deleted = ? GROUP BY s.name

echo $q->getSql(); // SELECT s.name AS s__name, s.something AS s__something, s.
    deleted AS s__deleted FROM soft_delete_test s WHERE s.deleted = ?

?>
```

17.3 Simple templates

In the following example we define a template called TimestampTemplate. 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.

Listing 17.30:

```
<?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()->created = date('Y-m-d', time());
        $event->getInvoker()->updated = date('Y-m-d', time());
    }
}
class TimestampTemplate extends Doctrine_Template
{
    public function setTableDefinition()
    {
        $this->hasColumn('created', 'date');
        $this->hasColumn('updated', 'date');

        $this->setListener(new TimestampListener());
    }
}

?>
```

Lets say we have a class called Blog that needs the timestamp functionality. All we need to do is to add actAs()
call in the class definition.

Listing 17.31:

```
<?php

class Blog extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 200);
        $this->hasColumn('content', 'string');
    }
    public function setUp()
    {
        $this->actAs('TimestampTemplate');
    }
}

?>
```

17.4 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:

Listing 17.32:

```
<?php

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

?>
```

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 - 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.

Listing 17.33:

```
<?php

class UserTemplate extends Doctrine_Template
{
    public function setTableDefinition()
    {
        $this->hasColumn('name', 'string');
    }
    public function setUp()
    {
        $this->hasMany('EmailTemplate as Email', array('local' => 'id', 'foreign
            ' => 'user_id'));
    }
}
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:

Listing 17.34:

```
<?php

class User extends Doctrine_Record
{
    public function setUp()
    {
        $this->actAs('UserTemplate');
    }
}
class Email extends Doctrine_Record
{
    public function setUp()
    {
```

```
        $this->actAs('EmailTemplate');
    }
}

?>
```

Now consider the following code snippet. This does NOT work since we haven't yet set any concrete implementations for the templates.

Listing 17.35:

```
<?php

$user = new User();
$user->Email; // throws an exception

?>
```

The following version works. Notice how we set the concrete implementations for the templates globally using Doctrine_Manager.

Listing 17.36:

```
<?php

$manager = Doctrine_Manager::getInstance();
$manager->setImpl('UserTemplate', 'User')
        ->setImpl('EmailTemplate', 'Email');

$user = new User();
$user->Email;

?>
```

The implementations for the templates can be set at manager, connection and even at the table level.

17.5 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 take an example: we have a User class that loads authentication functionality through a template.

Listing 17.37:

```
<?php

class User extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('fullname', 'string', 30);
    }
    public function setUp()
```

```
    {
        $this->actAs('AuthTemplate');
    }
}
class AuthTemplate extends Doctrine_Template
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 16);
        $this->hasColumn('password', 'string', 16);
    }
    public function login($username, $password)
    {
        // some login functionality here
    }
}

?>
```

Now you can simply use the methods found in AuthTemplate within the User class as shown above.

Listing 17.38:

```
<?php

$user = new User();

$user->login($username, $password);

?>
```

You can get the record that invoked the delegate method by using the getInvoker() method of Doctrine_Template.
Consider the AuthTemplate example. If we want to have access to the User object we just need to do the following:

Listing 17.39:

```
<?php

class AuthTemplate extends Doctrine_Template
{
    public function setTableDefinition()
    {
        $this->hasColumn('username', 'string', 16);
        $this->hasColumn('password', 'string', 16);
    }
    public function login($username, $password)
    {
        // do something with the Invoker object here
        $object = $this->getInvoker();
    }
}

?>
```

17.6 Multiple Templates

Each class can consists of multiple templates. If the templates contain similar definitions the most recently loaded
template always overrides the former.

17.7 Creating plugins

This subchapter provides you the means for creating your own plugins. Lets say we have various different Record
classes that need to have one-to-many emails. We achieve this functionality by creating a generic plugin which
creates Email classes on the fly.

We start this task by creating a plugin called EmailPlugin with 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:

public function initOptions()
public function buildLocalRelation()
public function buildForeignKeys(Doctrine_Table $table)
public function buildForeignRelation($alias = null)
public function buildRelation() // calls buildForeignRelation() and buildLocalRelation()

Listing 17.40:

```
<?php

class EmailPlugin extends Doctrine_Record_Generator
{
    public function initOptions()
    {
        $this->setOption('className', '%CLASS%Email');
    }

    public function buildRelation()
    {
          $this->buildForeignRelation('Emails');
        $this->buildLocalRelation();
    }

    public function setTableDefinition()
    {
        $this->hasColumn('address', 'string', 255, array('email'  => true,
                                                         'primary' => true));
    }
}

?>
```

17.8 Nesting plugins

Below is an example of several behaviors to give a complete wiki database that is versionable, searchable, sluggable,
and full I18n.

Listing 17.41:

```
<?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');
    }
}

?>
```

Listing 17.42:

```
---
WikiTest:
  actAs:
    I18n:
      fields: [title, content]
      actAs:
        Versionable:
          fields: [title, content]
        Searchable:
          fields: [title, content]
        Sluggable:
          fields: [title]
  columns:
    title: string(255)
    content: string
```

17.9 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.

Listing 17.43:

```
<?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'));
    }
}

?>
```

Now the behavior will generate a file instead of generating the code and using eval() to evaluate it at runtime.

Chapter 18

File parser

The parser is built to allow dumping and loading from many different formats. Currently xml, yml, json and php
serialization are the only formats supported. You can specify the data to load/dump in with the $type argument on
dump() and load()

18.1 Dumping

Dumping array to yml variable

Listing 18.1:

```
<?php

$array = array('test' => array('key' => 'value'), 'test2' => 'test');

// Dump the array to yml and return, set to $yml(does not write to file).
    Replace null with a path to a yml file if
you wish to write to disk
$yml = Doctrine_Parser::dump($array, 'yml');

?>
```

$yml would contain the following

Listing 18.2:

```
---
test:
  key: value
test2: test
```

Dumping array to yml file

Listing 18.3:

```
<?php

$array = array('test' => array('key' => 'value'), 'test2' => 'test');

// Dump the above array to test.yml using yml parser
Doctrine_Parser::dump($array, 'yml', 'test.yml');

?>
```

A file named test.yml would be created and would contain the following

Listing 18.4:

```
---
test:
  key: value
test2: test
```

18.2 Loading

Loading and parsing data from a yml file to a php array

Listing 18.5:

```
<?php

$array = array('test' => array('key' => 'value'), 'test2' => 'test');

// We dump the above array to test.yml using the yml parser dumper
Doctrine_Parser::dump($array, 'yml', 'test.yml');

// Now we reload that dumped yaml file back to the original array format using
    the yml parser loder
$array = Doctrine_Parser::load('test.yml', 'yml');

print_r($array);

?>
```

The print_r() would output the following

Listing 18.6:

```
Array
(
    [test] => Array
        (
            [key] => value
        )

    [test2] => test
)
```

Chapter 19

Migration

The Doctrine Migration tools allow you to migrate databases and it issues alter table statements directly to your databases when you need to deploy database changes.

19.1 Writing Migration Classes

Migration classes consist of a simple class that extends from Doctrine_Migration. You can define a public up() and down() method that is meant for doing and undoing changes to a database for that
migration step. The class name is completely arbitrary, but the name of the file which contains the
class must have a prefix containing the number it represents in the migration process.
Example: XXX_representative_name.class.php

Listing 19.1:

```
<?php

// 001_add_table.class.php
class AddTable extends Doctrine_Migration
{
    public function up()
    {
        $this->createTable('migration_test', array('field1' => array('type' => '
            string')));
    }

    public function down()
    {
        $this->dropTable('migration_test');
    }
}

// 002_add_column.class.php
class AddColumn extends Doctrine_Migration
{
    public function up()
    {
        $this->addColumn('migration_test', 'field1', 'string');
    }

    public function down()
    {
        $this->renameColumn('migration_test', 'field1', 'field2');
    }
}
```

```
// 003_change_column.class.php
class ChangeColumn extends Doctrine_Migration
{
    public function up()
    {
        $this->changeColumn('migration_test', 'field1', 'integer');
    }

    public function down()
    {
        $this->changeColumn('migration_test', 'field1', 'string');
    }
}

?>
```

19.1.1 Methods

Here is a list of the available methods you can use to alter your database in your migration classes

Create a new table

Listing 19.2:

```
<?php

// Name of the table to create
$tableName = 'new_table';

// Sample array of fields for the table
$fields = array('id' => array(
                    'type' => 'integer',
                    'unsigned' => 1
                    'notnull' => 1
                    'default' => 0
                ),
                'name' => array(
                    'type' => 'text',
                    'length' => 12
                ),
                'password' => array(
                    'type' => 'text',
                    'length' => 12
                ));

// Array of table options
$options = array('type'     => 'INNODB',
                 'charset'  => 'utf8');

$this->createTable($tableName, $fields, $options);

?>
```

Drop an existing table

Listing 19.3:

```
<?php

// Name of the table to drop
$tableName = 'new_table';
$this->dropTable($tableName);
```

```
?>
```

Rename an existing table

Listing 19.4:

```
<?php

// Old name of table
$oldTableName = 'users';

// New name of table
$newTableName = 'user';

$this->renameTable($oldTableName, $newTableName);

?>
```

Create a new database constraint

Listing 19.5:

```
<?php

// Name of the table to create the constraint on
$tableName = 'user';

// Name of the constraint to create
$constraintName = 'unique_username';

// Definition array
$definition = array('fields' => array('username' => array())
                    'unique' => true); // 'primary' => true

$this->createConstraint($tableName, $constraintName, $definition);

?>
```

Listing 19.6:

```
<?php

// Name of the table where the constraint lives
$tableName = 'user';

// Name of the constraint to drop
$constraintName = 'unique_username';

// Whether or not this constraint is a primary constraint
$primary = false;

$this->dropConstraint($tableName, $constraintName, $primary = false);

?>
```

Create a foreign key

Listing 19.7:

```
<?php

// Name of the table to create the foreign key on
$tableName = 'user';

// Definition of the foreign key
```

```
$definition = array('name'          => 'email_foreign_key'
                    'local'         => 'email_id',
                    'foreign'       => 'id',
                    'foreignTable'  => 'email',
                    'onDelete'      => 'CASCADE');

$this->createForeignKey($tableName, $definition);

?>
```

The valid options for the $definition are:

key	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 a foreign key

Listing 19.8:

```
<?php

// Name of the table where the foreign key exists
$tableName = 'user';

// Name of the foreign key
$fkName = 'email_foreign_key';

$this->dropForeignKey($tableName, $fkName);

?>
```

Add a new column to a table

Listing 19.9:

```
<?php

// Name of the table to add the column to
$tableName = 'user';

// Name of the column to add
$columnName = 'email_address';

// Data type for column
$type = 'string';

// Array of options for column
$options = array('length' => '255');

$this->addColumn($tableName, $columnName, $type, $options);

?>
```

Rename an existing column on a table

Listing 19.10:

```php
<?php

// Name of the table where the column to rename exists
$tableName = 'user';

// Old name of the column
$oldColumnName = 'login';

// New name of the column
$newColumnName = 'username';

$this->renameColumn($tableName, $oldColumnName, $newColumnName);

?>
```

Change any aspect of an existing column

Listing 19.11:

```php
<?php

// Name of the table where the column to change exists
$tableName = 'user';

// Name of the column to change
$columnName = 'is_active';

// Type to change the column to
$type = 'tinyint';

// Array of options to change for the column
$options = array('length' => 1);

$this->changeColumn($tableName, $columnName, $type, $options);

?>
```

Remove an existing column from a table

Listing 19.12:

```php
<?php

// Name of the table where the column to remove exists
$tableName = 'user';

// Name of the column to remove
$columnName = 'num_logins';

$this->removeColumn($tableName, $columnName)

?>
```

Add an index to a table

Listing 19.13:

```php
<?php

// Name of the table to create the index on
$tableName = 'user';

// Name of the index to create
$indexName = 'username_last_loginx';

$options = array('fields' => array(
```

```
                    'username' => array(
                        'sorting' => 'ascending'
                    ),
                    'last_login' => array()));

$this->addIndex($tableName, $indexName, array $options = array())

?>
```

Remove an existing index from a table

Listing 19.14:

```
<?php

// Name of the table to remove the index from
$tableName = 'user';

// Name of the index to remove
$indexName = 'username_last_loginx';

$this->removeIndex($tableName, $indexName)

?>
```

19.1.2 Altering Data

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 after the migration version is executed.

Listing 19.15:

```
<?php

// XXX_add_user.class.php
class AddUser extends Doctrine_Migration
{
    public function up()
    {
        $this->createTable('migration_test', array('field1' => array('type' => '
            string')));
    }

    public function postUp()
    {
      $migrationTest = new MigrationTest();
      $migrationTest->field1 = 'test';
      $migrationTest->save();
    }

    public function down()
    {
        $this->dropTable('migration_test');
    }

    public function postDown()
    {
```

```
        $migrationTest = Doctrine::getTable('MigrationTest')->findOneByField1('
            test');
        $migrationTest->delete();
    }
}

?>
```

19.2 Performing Migrations

Listing 19.16:

```
<?php

$migration = new Doctrine_Migration('/path/to/migration_classes');

// Assume current version is 0
$migration->migrate(3); // takes you from 0 to 3
$migration->migrate(0); // takes you from 3 to 0

echo $migration->getCurrentVersion(); // 0

?>
```

This functionality is can also be accessed from the Doctrine command line interface.

Chapter 20

Searching

20.1 Introduction

Searching is a huge topic, hence an entire chapter has been devoted to a plugin called Doctrine_Search. Doctrine_Search
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:

Listing 20.1:

```
<?php

class NewsItem extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 200);
        $this->hasColumn('content', 'string');
    }
}

?>
```

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 form build DQL queries such as:

Listing 20.2:

```
SELECT n.* FROM NewsItem n 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:

Listing 20.3:

```
<?php
```

```
class NewsItem extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('title', 'string', 200);
        $this->hasColumn('content', 'string');
    }
    public function setUp()
    {
        $this->actAs('Searchable', array('fields' => array('title', 'content'))
            ;
    }
}

?>
```

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 everytime 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)

Sometimes you may want to alter the search object options afterwards. The search object can be accessed as follows:

Listing 20.4:

```
<?php

$search = Doctrine::getTable('NewsItem')
              ->getTemplate('Searchable')
              ->getPlugin();

?>
```

20.2 Index structure

The structure of the inverse index Doctrine uses is the following:

[(string) keyword] [(string) field] [(integer) position] [(mixed) [foreign_keys]]

- **keyword** is the keyword in the text that can be searched for
- **field** is the field where the keyword was found
- **position** is the position where the keyword was found
- **[foreign_keys]** either one or multiple fields depending on the owner component (here NewsItem)

In the NewsItem example the [foreign_keys] would simply contain one field id with foreign key references to NewsItem(id)
and with onDelete => CASCADE constraint.

An example row in this table might look something like:

keyword	field	position	id
database	title	3	1

In this example the word database is the third word of the title field of NewsItem 1.

20.3 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.

Listing 20.5:

```
<?php

$search->setOption('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.

Listing 20.6:

```
<?php

$newsItem = new NewsItem();

$newsItem->batchUpdateIndex();

?>
```

20.4 Text analyzers

By default Doctrine uses Doctrine_Search_Analyzer_Standard for analyzing the text. This class performs the following
things:

1. 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.

2. Makes all keywords lowercased
When searching words 'database' and 'DataBase' are considered equal by the standard analyzer, hence the standard
analyzer lowercases all keywords.
3. 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.'.
4. 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. This analyzer
can then be applied to the search object as follows:

Listing 20.7:

```
<?php

$search->setOption('analyzer', new MyAnalyzer());

?>
```

20.5 Query language

Doctrine_Search provides a query language similar to Apache Lucene. The parsed behind Doctrine_Search_Query converts
human readable, easy-to-construct search queries to their complex sql equivalents.

20.6 Performing Searches

Here is a simple example to retrieve the record ids and relevance data.

Listing 20.8:

```
<?php

$results = Doctrine::getTable('Article a')->search('php orm');

// Executes the following query and returns an associative array
// 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

print_r($results); // Will print an array of record ids and the relevance of
    each

?>
```

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.

Listing 20.9:

```
<?php

$query = Doctrine_Query::create()
    ->from('Article a');
```

```
$articles = Doctrine::getTable('Article a')->search('php orm', $query)->
    fetchArray();

// Executes the following query
// SELECT a.id AS a__id, a.title AS a__title, a.body AS a__body FROM article a
    WHERE a.id IN(SELECT id FROM article_index WHERE id IN (SELECT id FROM
    article_index WHERE keyword = ?) && id IN (SELECT id FROM article_index
    WHERE keyword = ?) GROUP BY id)

print_r($articles); // Will print the articles which have the keywords php or
    orm in them.

?>
```

20.7 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.

Listing 20.10:

```
<?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.

Listing 20.11:

```
<?php

$search->buildDefinition(); // builds to table and record class definitions

$conn->export->exportClasses(array('FileIndex'));

?>
```

Now we can start using the file searcher. First lets index some directory:

Listing 20.12:

```
<?php

$search->indexDirectory('myfiles');

?>
```

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:

Listing 20.13:

```
<?php

$resultSet = $search->search('database orm');

?>
```

Chapter 21

Database abstraction

21.1 Modules

21.2 Export

21.2.1 Introduction

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.

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(id, name, datetime);
people(id, name);
event_participants(event_id, person_id);

21.2.2 Creating a database

Listing 21.1:

```
<?php

$conn->export->createDatabase('events_db');

?>
```

21.2.3 Creating tables

Now that the database is created, 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:

Listing 21.2:

```
<?php

$definition = array (
    'id' => array (
        'type' => 'integer',
        'unsigned' => 1,
        'notnull' => 1,
        'default' => 0,
    ),
    '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		

Creating the people table:

Listing 21.3:

```
<?php

$options = array(
    'comment' => 'Repository of people',
    'character_set' => 'utf8',
    'collate' => 'utf8_unicode_ci',
    'type'    => 'innodb',
);
$definition = array (
    'id' => array (
        'type' => 'integer',
        'unsigned' => 1,
        'notnull' => 1,
        'default' => 0,
    ),
    'name' => array (
```

```
        'type' => 'string',
        'length' => 255
    )
);
$conn->export->createTable('people', $definition, $options);

?>
```

21.2.4 Creating foreign keys

Creating the event_participants table with a foreign key:

Listing 21.4:

```
<?php

$options = array(
    'foreignKeys' => array('local'   => 'event_id',
                           'foreign' => 'id'
                           'foreignTable' => 'events'
                           'onDelete' => 'CASCADE'),
    'primary' => array('event_id', 'person_id'),
);
$definition = array (
    'event_id' => array (
        'type' => 'integer',
        'unsigned' => 1,
        'notnull' => 1,
        'default' => 0,
    ),
    'person_id' => array (
        'type' => 'integer',
        'unsigned' => 1,
        'notnull' => 1,
        'default' => 0,
    ),
);

$conn->export->createTable('event_participants', $definition, $options);

?>
```

Now lets say we want to add foreign key on person_id too. This can be achieved as follows:

Listing 21.5:

```
<?php

$definition = array('local'   => 'person_id',
                    'foreign' => 'id'
                    'foreignTable' => 'people'
                    'onDelete' => 'CASCADE'))

$conn->export->createForeignKey('event_participants', $definition);

?>
```

21.2.5 Altering table

Doctrine_Export drivers provide an easy database portable way of altering existing database tables.

NOTE: if you only want to get the generated sql (and not execute it) use Doctrine_Export::alterTableSql()

Listing 21.6:

```
<?php

$dbh   = new PDO('dsn','username','pw');
$conn  = Doctrine_Manager::getInstance()
        ->openConnection($dbh);

$a     = array('add' => array('name' => array('type' => 'string', 'length' =>
    255)));

$conn->export->alterTableSql('mytable', $a);

// On mysql this method returns:
// ALTER TABLE mytable ADD COLUMN name VARCHAR(255)

?>
```

Doctrine_Export::alterTable() takes two parameters:

string *$name* name of the table that is intended to be changed.

array *$changes* associative array that contains the details of each type of change that is intended to be performed.

An optional third parameter (default: false) is accepted in alterTable and alterTableSql; it is named *$check* and it
identifies if the DBMS driver can perform the requested table alterations if the value is true or actually perform them
otherwise.

The types of changes that are currently supported are defined as follows:

- *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.

Listing 21.7:

```
<?php

$a = array('name' => 'userlist',
           'add' => array(
                    'quota' => array(
                        'type' => 'integer',
                        'unsigned' => 1
                        )
                    ),
            'remove' => array(
                    'file_limit' => array(),
                    'time_limit' => array()
                    ),
            'change' => array(
                    'name' => array(
                        'length' => '20',
                        'definition' => array(
                            'type' => 'string',
                            'length' => 20
                            )
                        )
                    ),
            'rename' => array(
                    'sex' => array(
                        'name' => 'gender',
                        'definition' => array(
                            'type' => 'string',
                            'length' => 1,
                            'default' => 'M'
                            )
                        )
                    )

            );

$dbh  = new PDO('dsn','username','pw');
$conn = Doctrine_Manager::getInstance()->openConnection($dbh);
```

```
$conn->export->alterTable('mytable', $a);

?>
```

21.2.6 Creating indices

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 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.

Listing 21.8:

```
<?php

$definition = array(
    'fields' => array(
        'datetime' => array()
    )
);
$conn->export->createIndex('events', 'event_timestamp', $definition);

?>
```

21.2.7 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.

Listing 21.9:

```
<?php

// drop a sequence
try {
    $conn->export->dropSequence('nonexisting');
} catch(Doctrine_Exception $e) {

}
// another sequence
$result = $conn->export->dropSequence('people');

// drop a constraint
```

```
$conn->export->dropConstraint('events', 'PRIMARY', true);
// note: the third parameter gives a hint
//       that this is a primary key constraint
$conn->export->dropConstraint('event_participants', 'unique_participant');

// drop an index
$conn->export->dropIndex('events', 'event_timestamp');

// drop a table
$conn->export->dropTable('events');

// drop the database already!
$conn->export->dropDatabase('events_db');

?>
```

21.3 Import

21.3.1 Introduction

To see what's in the database, you can use the list*() family of functions in the Import module.

- listDatabases()
- listFunctions()
- listSequences(): takes optional database name as a parameter. If not supplied, the currently selected database is assumed.
- listTableConstraints(): takes a table name
- listTableColumns(): takes a table name
- listTableIndexes(): takes a table name
- listTables(): takes an optional database name
- listTableTriggers(): takes a table name
- listTableViews(): takes a table name
- listUsers()
- listViews(): takes an optional database name

21.3.2 Listing databases

Listing 21.10:

```
<?php

$dbs = $conn->import->listDatabases();
print_r($dbs);

?>
```

21.3.3 Listing sequences

Listing 21.11:

```
<?php

$seqs = $conn->import->listSequences('events_db');
print_r($seqs);

?>
```

21.3.4 Listing constraints

Listing 21.12:

```
<?php

$cons = $conn->import->listTableConstraints('event_participants');

?>
```

21.3.5 Listing table fields

Listing 21.13:

```
<?php

$fields = $conn->import->listTableColumns('events');
print_r($fields);
/*
prints:
Array
(
    [0] => id
    [1] => name
    [2] => datetime
)
*/

?>
```

21.3.6 Listing table indices

Listing 21.14:

```
<?php

$idx = $conn->import->listTableIndexes('events');
print_r($idx);
/*
prints:
Array
(
    [0] => event_timestamp
)
*/

?>
```

21.3.7 Listing tables

Listing 21.15:

```
<?php

$tables = $conn->import->listTables();
print_r($tables);
/*
prints:
Array
(
    [0] => event_participants
    [1] => events
    [2] => people
)
*/

?>
```

21.3.8 Listing views

Listing 21.16:

```
<?php

// currently there is no method to create a view,
// 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);
// list views
$views = $conn->import->listViews();
print_r($views);
/*
prints:
Array
(
    [0] => last_ten_events
    [1] => names_only
)
*/

?>
```

21.4 DataDict

21.4.1 Introduction

Doctrine uses DataDict module internally to convert native RDBMS types to Doctrine types and the reverse. DataDict module
uses two methods for the conversions:
1. getPortableDeclaration(), which is used for converting native RDBMS type declaration to portable Doctrine declaration
2. getNativeDeclaration(), which is used for converting portable Doctrine declaration to driver specific type declaration

21.4.2 Getting portable declaration

Listing 21.17:

```
<?php

$dbh  = new PDO('mysql:host=localhost;dbname=test', 'username', 'pw');
$conn = Doctrine_Manager::getInstance()->openConnection($dbh);

$decl = $conn->dataDict->getPortableDeclaration('VARCHAR(255)');

print_r($decl);
/*
array('type' => 'string',
      'length' => 255,
      'fixed'  => false,
      'unsigned' => false
      );
*/

?>
```

21.4.3 Getting native declaration

Listing 21.18:

```
<?php

$dbh  = new PDO('mysql:host=localhost;dbname=test', 'username', 'pw');
$conn = Doctrine_Manager::getInstance()->openConnection($dbh);

$portableDecl = array('type' => 'string',
                      'length' => 20,
                      'fixed' => true);
$nativeDecl = $conn->dataDict->getNativeDeclaration($portableDecl);

print $nativeDecl; // CHAR(20)

?>
```

21.5 Drivers

21.5.1 Mysql

21.5.1.1 Setting table type

Listing 21.19:

```
<?php

$dbh  = new PDO('dsn','username','pw');
$conn = Doctrine_Manager::getInstance()->openConnection($dbh);

$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' => 'MYISAM');

$conn->export->createTable('mytable', $fields);

// on mysql this executes query:
// CREATE TABLE mytable (id INT AUTO_INCREMENT PRIMARY KEY,
//          name CHAR(8));

?>
```

Chapter 22

Improving Performance

22.1 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.

22.2 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.

Listing 22.1:

```php
<?php

Doctrine::compile();
// on some other script:
require_once('path_to_doctrine/Doctrine.compiled.php');
```

```
?>
```

22.3 Fetch only what you need

Maybe the most important rule is to 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:

Listing 22.2:

```
<?php

$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 pretty sure you all know why

Listing 22.3:

```
SELECT * FROM ...
```

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**. Until recently
this statement would make no sense at all but one of the recent additions to Doctrine is 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 prodecural 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 pseudo-code that
fetches all comments with some related data for an article, passing them to the view for display afterwards:

Listing 22.4:

```
<?php

$comments = $query->select("c.id, ...")->from("Comment c")
        ->leftJoin("c.foo f")
```

```
        ->leftJoin("f.bar b")
        ->where("c.article_id = ?")
        ->execute(array(1));
$view->comments = $comments;

?>
```

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:

Listing 22.5:

```
<?php

... ->execute(array(1), Doctrine::HYDRATE_ARRAY);

?>
```

This will return a bunch of nested php arrays. It could look something like this, assuming we fetched some comments:

Listing 22.6:

```
array(5) (
    [0] => array(
        'title' => 'Title1',
        'message' => 'Hello there! I like donuts!',
        'author' => array(
            'first_name' => 'Bart',
            'last_name' => 'Simpson'
        )
    ),
    [1] => array(
        'title' => 'Title2',
        'message' => 'Hullo!',
        'author' => array(
            'first_name' => 'Homer',
            'last_name' => 'Simpson'
        )
    ),
    ...
)
```

Here 'author' is a related component of a 'comment' and thus results in a sub-array. If you always use the array
syntax for accessing data, then the switch to array fetching requires nothing more than adding the additional parameter.
The following code works regardless of the fetching style:

Listing 22.7:

```
<?php

foreach ($comments as $comment) {
    echo $comment['title'] . '<br />';
    echo $comment['message'] . '<br />';
    echo $comment['author']['first_name'] . ' - ' . $comment['author']['
       last_name'] . '<br />';
}

?>
```

Array fetching is the best choice whenever you need data read-only like passing it to the view for display. And from my experience, most of the time when you fetch a large amount of data it's only for display purposes. And these are exactly the cases where you get the best performance payoff when fetching arrays instead of objects.

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::HYDRATE_NONE**. Here's an example:

Listing 22.8:

```
<?php

$total = Doctrine_Query::create()
  ->select('SUM(d.amount)')
  ->from('Donation d')
  ->execute(array(), Doctrine::HYDRATE_NONE);

?>
```

22.4 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.

22.5 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.

22.6 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:

Listing 22.9:

```
<?php

for ($i=0; $i<1000; $i++)
{
  $object = createBigObject();
  $object->save();
  $object->free(true); // Free the object and all of its relations

  $query = Doctrine_Query::create()->...;
  $results = $query->fetchArray();
  $query->free();
}

?>
```

22.7 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:

Listing 22.10:

```
<?php

// Assuming a many-many between role - user
$user->roles[] = $newRole;

?>
```

This will load all roles of the user from the database if they're not yet loaded! Just to add one new link! Do
this instead:

Listing 22.11:

```
<?php

// Assuming a many-many between role - user, where UserRoleXref is the cross-
    reference table
$ref = new UserRoleXref();
$ref->role_id = $role_id;
$ref->user_id = $user_id;
$ref->save();

?>
```

Chapter 23

Technology

23.1 Architecture

Doctrine is divided into 3 main 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

There are also plugins for Doctrine:

- Doctrine_Validator
- Doctrine_Hook
- Doctrine_View
- Doctrine_Tree + Doctrine_Node

23.2 Design patterns used

GoF (Gang of Four) design patterns used:

- Singleton[1], for forcing only one instance of `Doctrine_Manager`
- Composite[2], for leveled configuration
- Factory[3], for connection driver loading and many other things
- Observer[4], for event listening
- Flyweight[5], for efficient usage of validators
- Iterator[6], for iterating through components (Tables, Connections,

Records etc.)

- State[7], for state-wise connections
- Strategy[8], for algorithm strategies

Enterprise application design patterns used:

- Active Record[9], Doctrine is an implementation of this pattern
- UnitOfWork[10], for maintaining a list of objects affected in a

[1] http://www.dofactory.com/Patterns/PatternSingleton.aspx
[2] http://www.dofactory.com/Patterns/PatternComposite.aspx
[3] http://www.dofactory.com/Patterns/PatternFactory.aspx
[4] http://www.dofactory.com/Patterns/PatternObserver.aspx
[5] http://www.dofactory.com/Patterns/PatternFlyweight.aspx
[6] http://www.dofactory.com/Patterns/PatternFlyweight.aspx
[7] http://www.dofactory.com/Patterns/PatternState.aspx
[8] http://www.dofactory.com/Patterns/PatternStrategy.aspx
[9] http://www.martinfowler.com/eaaCatalog/activeRecord.html
[10] http://www.martinfowler.com/eaaCatalog/unitOfWork.html

transaction

- Identity Field[11], for maintaining the identity between record

and database row

- Metadata Mapping[12], for Doctrine DataDict
- Dependent Mapping[13], for mapping in general, since all

records extend `Doctrine_Record` which performs all mappings

- Foreign Key Mapping[14], for one-to-one, one-to-many and

many-to-one relationships

- Association Table Mapping[15], for association table

mapping (most commonly many-to-many relationships)

- Lazy Load[16], for lazy loading of objects and object properties
- Query Object[17], DQL API is actually an extension to the basic

idea of Query Object pattern

23.3 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.

[11] http://www.martinfowler.com/eaaCatalog/identityField.html
[12] http://www.martinfowler.com/eaaCatalog/metadataMapping.html
[13] http://www.martinfowler.com/eaaCatalog/dependentMapping.html
[14] http://www.martinfowler.com/eaaCatalog/foreignKeyMapping.html
[15] http://www.martinfowler.com/eaaCatalog/associationTableMapping.html
[16] http://www.martinfowler.com/eaaCatalog/lazyLoad.html
[17] http://www.martinfowler.com/eaaCatalog/queryObject.html

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.

23.4 Internal optimizations

Chapter 24

Exceptions and warnings

24.1 Manager exceptions

Doctrine_Manager_Exception is thrown if something failed at the connection management

Listing 24.1:

```
<?php

try {
    $manager->getConnection('unknown');
} catch (Doctrine_Manager_Exception) {
    // catch errors
}

?>
```

24.2 Relation exceptions

Relation exceptions are being thrown if something failed during the relation parsing.

24.3 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.

Listing 24.2:

```
<?php

try {
    $conn->execute('SELECT * FROM unknowntable');
} catch (Doctrine_Connection_Exception $e) {
    print 'Code : ' . $e->getPortableCode();
    print 'Message : ' . $e->getPortableMessage();
}

?>
```

24.4 Query exceptions

An exception will be thrown when a query is executed if the DQL query is invalid in some way.

Chapter 25

Real world examples

25.1 User management system

Listing 25.1:

```
<?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()
    {
        $this->hasOne('User', array('local' => 'user_id', 'foreign' => 'id'));
        $this->hasOne('Permission', array('local' => 'permission_id', 'foreign'
            => 'id'));
    }
}

?>
```

Listing 25.2:

```
---
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:
```

25.2 Forum application

Listing 25.3:

```
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->ownsMany('Forum_Entry as Entries', array('local' => 'id', '
            foreign' => thread_id'));
    }
}
```

Listing 25.4:

```
---
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
```

Chapter 26

Coding standards

26.1 Overview

26.1.1 Scope

26.1.2 Goals

26.2 PHP File Formatting

26.2.1 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.

IMPORTANT: 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.

26.2.2 Indentation

Use an indent of 4 spaces, with no tabs.

26.2.3 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.

26.2.4 Line termination

- Line termination is the standard way for Unix text files. Lines must end only with a linefeed (LF). Linefeeds are

represented as ordinal 10, or hexadecimal 0x0A.

- Do not use carriage returns (CR) like Macintosh computers (0x0D).
- Do not use the carriage return/linefeed combination (CRLF) as Windows computers (0x0D, 0x0A).

26.3 Naming Conventions

26.3.1 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.

26.3.2 Interfaces

- Interface classes must follow the same conventions as other classes (see above), however must end with the word

"`Interface`" (unless the interface is approved not to contain it such as `Doctrine_Overloadable`). Some examples:
* `Doctrine_Db_EventListener_Interface`
* `Doctrine_EventListener_Interface`

26.3.3 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/Db.php`
* `Doctrine/Connection/Transaction.php`

- File names must follow the mapping to class names described above.

26.3.4 Functions and methods

- Function names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in function

names but are discouraged.

- Function names must always start with a lowercase letter. 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. 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.

26.3.5 Variables

All variables must satisfy the following conditions:

- Variable names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in variable

names but are discouraged.

- Variable names 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	$query
Doctrine_Db	$db

- 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.

26.3.6 Constants

Following rules must apply to all constants used within Doctrine framework:

- Constants may contain both alphanumeric characters and the underscore.
- Constants 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.

Listing 26.1:

```php
<?php

class Doctrine_SomeClass
{
    const MY_CONSTANT = 'something';
}
print Doctrine_SomeClass::MY_CONSTANT;

?>
```

26.3.7 Record columns

- All record columns must be in lowercase
- Usage of _ is encouraged for columns that consist of more than one word

Listing 26.2:

```
<?php

class User
{
    public function setTableDefinition()
    {
        $this->hasColumn('home_address', 'string');
    }
}

?>
```

- Foreign key fields must be in format [tablename]_[column]

Listing 26.3:

```
<?php

class Phonenumber
{
    public function setTableDefinition()
    {
        // this field is a foreign key that points to user(id)
        $this->hasColumn('user_id', 'integer');
    }
}

?>
```

26.4 Coding Style

26.4.1 PHP code demarcation

- PHP code must always be delimited by the full-form, standard PHP tags
- Short tags are never allowed. For files containing only PHP code, the closing tag must always be omitted

26.4.2 Strings

- When a string is literal (contains no variable substitutions), the apostrophe or "single quote" must always used to

demarcate the string:

Listing 26.4:

```
<?php

// literal string
$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:

Listing 26.5:

```
<?php

// string contains apostrophes
$sql = "SELECT id, name FROM people WHERE name = 'Fred' OR name = 'Susan'";

?>
```

- Variable substitution is permitted using the following form:

Listing 26.6:

```
<?php

// variable substitution
$greeting = "Hello $name, welcome back!";

?>
```

- Strings may be concatenated using the "." operator. A space must always be added before and after the "." operator

to improve readability:

Listing 26.7:

```
<?php

// concatenation
$framework = 'Doctrine' . ' ORM ' . 'Framework';

?>
```

- 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:

Listing 26.8:

```
<?php

// concatenation line breaking
$sql = "SELECT id, name FROM user "
     . "WHERE name = ? "
     . "ORDER BY name ASC";

?>
```

26.4.3 Arrays

- Negative numbers are not permitted as indices.
- An 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:

Listing 26.9:

```
<?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');

?>
```

26.4.4 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.
- 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:

Listing 26.10:

```php
<?php

/**
 * Documentation here
 */
class Doctrine_SampleClass
{
    // entire content of class
    // must be indented four spaces
}

?>
```

26.4.5 Functions and methods

- Methods must be named by following the naming conventions.
- Methods 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:

Listing 26.11:

```php
<?php

/**
 * Documentation Block Here
 */
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar()
    {
        // entire content of function
        // must be indented four spaces
    }
}

?>
```

- Passing by-reference is permitted in the function declaration only:

Listing 26.12:

```
<?php

/**
 * 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.

Listing 26.13:

```
<?php

/**
 * 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:

Listing 26.14:

```
<?php

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:

Listing 26.15:

```
<?php

threeArguments(array(1, 2, 3), 2, 3);

threeArguments(array(1, 2, 3, 'Framework',
                     'Doctrine', 56.44, 500), 2, 3);

?>
```

26.4.6 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.

Listing 26.16:

```
<?php

if ($foo != 2) {
    $foo = 2;
}

?>
```

- For `if` statements that include `elseif` or `else`, the formatting must be as in these examples:

Listing 26.17:

```php
<?php

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:

Listing 26.18:

```php
<?php

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.

Listing 26.19:

```php
<?php

switch ($case) {
    case 1:
    case 2:
    break;
    case 3:
    break;
    default:
    break;
}

?>
```

- The construct default may never be omitted from a switch statement.

26.4.7 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/

Methods:

* 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`:
* `@throws exceptionclass [description]`

26.5 Testing

Doctrine is programmatically tested using UnitTests - see http://en.wikipedia.org/wiki/Unit_testing.

26.5.1 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.

26.5.1.1 CLI

To run tests on the command line, you must have php-cli installed.

Navigate to the DOCTRINE_PATH/tests folder and run the following command:

Listing 26.20:

```
php run.php
```

This should print out a progress indicator and a report of any test failures.

The CLI testrunner has several options for coverage reports, running a group of tests, and filtering tests against classnames of testsuites. Run "php run.php -help" for more details on these options.

26.5.1.2 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://example.com/doctrine/tests/run.php
- http://example.com/doctrine/tests/run.php?filter=Limit\&group[]=query\&group[]=record

26.5.2 Writing tests

When writing your test case, you can copy `TemplateTestCase.php` to start off. Include your test in run.php:

Listing 26.21:

```
<?php

$test->addTestCase(new Doctrine_Sample_TestCase());

?>
```

Here is a sample test case:

Listing 26.22:

```
<?php
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));
    }
}

// You can optionally put your model classes right in the test, or put them in
    the models folder and they will be autoloaded.

class Model1 extends Doctrine_Record
{
}

class Model2 extends Doctrine_Record
{
}
```

If you execute run.php you should you see your failing test.

26.5.2.1 Methods for testing

Listing 26.23:

```
<?php

public function assertEqual($value, $value2)
public function assertIdentical($value, $value2)
public function assertNotEqual($value, $value2)
public function assertTrue($expr)
public function assertFalse($expr)

?>
```

26.5.2.2 Mock drivers

Doctrine uses mock drivers for all drivers other than sqlite. The following code snippet shows you how to use mock drivers:

Listing 26.24:

```
<?php
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
            );
    }
}
```

26.5.2.3 Test Class Guidelines

- Every class should have at least one `TestCase` equivalent
- All testcase classes 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.

26.5.2.4 Test Method Guidelines

- Methods should support agile documentation
- Test methods should be named so that if it fails, it is obvious what failed.
- Test method names should give information of the system they test
- Example: `Doctrine_Export_Pgsql_TestCase::testCreateTableSupportsAutoincPks()` is a good test 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.

NOTE: 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!