



symfony and Doctrine with sfDoctrinePlugin

symfony 1.2

This PDF is brought to you by
SENSIOLABS 

License: GFDL

Version: doctrine-1.2-en-2010-03-21

Table of Contents

Chapter 1: Getting Started	4
What is Doctrine?	4
Enable Doctrine	4
Available Tasks	5
Chapter 2: Connections.....	7
Introduction	7
Supported Drivers	8
Data Source Name(DSN)	8
Doctrine Style	8
PDO Style	8
Import from Database.....	9
Multiple Connections.....	9
Connection Attributes.....	10
Build Everything	11
Chapter 3: Configuration	13
Attributes	13
Global	13
All Connections.....	13
Model.....	14
Configuring Model Building	14
Custom Doctrine Library Path	15
Chapter 4: Schema Files	17
Data Types	17
Options.....	20
Indexes.....	20
Relationships	21
One to One.....	21
One to Many	21
Many to Many.....	22
Cascading Operations.....	23
Database Level	23
Application Level	24
Behaviors	24
Core Behaviors	24
Nesting Behaviors.....	25
Inheritance.....	26
Concrete Inheritance.....	26
Simple Inheritance	27
Column Aggregation Inheritance	27
Global Schema Information	28

Plugin Schemas	29
Element Definitions	30
Root Elements	30
Columns	30
Relations	31
Inheritance	31
Indexes	32
Chapter 5: Data Fixtures	33
Introduction	33
Original	34
Linking Relationships	35
Many to Many	36
Inline	37
Chapter 6: Working with Data	38
Retrieving Data	38
DQL	38
Finders	44
Altering Data	45
Object Properties	45
Overriding Accessors and Mutators	45
Working with Relationships	45
Deleting Data	47
Chapter 7: Migrations	48
Available Migration Tasks	48
Starting Schema and Data Fixtures	48
Generating Migrations	49
From Database	49
From Models	49
Skeleton	50

Chapter 1

Getting Started



The symfony and Doctrine book does not explain in detail everything about Doctrine. It briefly touches on a few critical subjects that symfony utilizes the most. The Doctrine manual is still the best source of documentation for referencing everything that is Doctrine. It is recommended you read this book to get started and then utilize the Doctrine manual for referencing the specifics of using Doctrine. You can find the main Doctrine documentation [here](http://www.doctrine-project.org/documentation/1.0/en)¹.



What is Doctrine?

duplication.

Doctrine is a PHP ORM (object relational mapper) for PHP 5.2.3+ that sits on top of a powerful PHP DBAL (database abstraction layer). One of its key features is the ability to optionally write database queries in an OO (object oriented) SQL-dialect called DQL inspired by Hibernate's HQL. This provides developers with a powerful alternative to SQL that maintains a maximum of flexibility without requiring needless code

Enable Doctrine

In order to begin using Doctrine you must first enable it by editing your `config/ProjectConfiguration.class.php` `setup()` method to enable `sfDoctrinePlugin` and disable `sfPropelPlugin`.

```
Listing 1-1 public function setup()
{
    $this->enablePlugins(array('sfDoctrinePlugin'));
    $this->disablePlugins(array('sfPropelPlugin'));
}
```

If you prefer to have all plugins enabled by default, you can do the following:

```
Listing 1-2 public function setup()
{
    $this->enableAllPluginsExcept(array('sfPropelPlugin',
```

1. <http://www.doctrine-project.org/documentation/1.0/en>

```
'sfCompat10Plugin')));
}
```



Though not recommended, Doctrine and Propel can be used side-by-side. Simply enable `sfDoctrinePlugin` without disabling `sfPropelPlugin` and you are good to go.

Available Tasks

Below is the complete list of tasks available with the `sfDoctrinePlugin`. `sfDoctrinePlugin` offers all of the functionality of `sfPropelPlugin`, plus much more.

```
$ ./symfony list doctrine
Usage:
```

```
  symfony [options] task_name [arguments]
```

Options:

```
--dry-run      -n Do a dry run without executing actions.
--help         -H Display this help message.
--quiet        -q Do not log messages to standard output.
--trace        -t Turn on invoke/execute tracing, enable full backtrace.
--version      -V Display the program version.
```

Available tasks for the "doctrine" namespace:

```
:build-all          Generates Doctrine model, SQL and
initializes the database (doctrine-build-all)
:build-all-load     Generates Doctrine model, SQL, initializes
database, and load data (doctrine-build-all-load)
:build-all-reload   Generates Doctrine model, SQL, initializes
database, and load data (doctrine-build-all-reload)
:build-all-reload-test-all Generates Doctrine model, SQL, initializes
database, load data and run all test suites
(doctrine-build-all-reload-test-all)
:build-db            Creates database for current model
(doctrine-build-db)
:build-filters       Creates filter form classes for the current
model
:build-forms         Creates form classes for the current model
(doctrine-build-forms)
:build-model         Creates classes for the current model
(doctrine-build-model)
:build-schema        Creates a schema from an existing database
(doctrine-build-schema)
:build-sql           Creates SQL for the current model
(doctrine-build-sql)
:data-dump           Dumps data to the fixtures directory
(doctrine-dump-data)
:data-load           Loads data from fixtures directory
(doctrine-load-data)
:dql                 Execute a DQL query and view the results
(doctrine-dql)
:drop-db             Drops database for current model
(doctrine-drop-db)
:generate-admin      Generates a Doctrine admin module
:generate-migration Generate migration class
(doctrine-generate-migration)
```

Listing
1-3

```
:generate-migrations-db      Generate migration classes from existing
database connections (doctrine-generate-migrations-db,
doctrine-gen-migrations-from-db)
:generate-migrations-models  Generate migration classes from an existing
set of models (doctrine-generate-migrations-models,
doctrine-gen-migrations-from-models)
:generate-module             Generates a Doctrine module
(doctrine-generate-crud, doctrine:generate-crud)
:generate-module-for-route   Generates a Doctrine module for a route
definition
:insert-sql                  Inserts SQL for current model
(doctrine-insert-sql)
:migrate                      Migrates database to current/specified
version (doctrine-migrate)
:rebuild-db                  Creates database for current model
(doctrine-rebuild-db)
```

Chapter 2

Connections

Introduction

In this chapter we'll explain some things about Doctrine connections, how to configure multiple connections, bind models, and how to create and drop your databases and other connection related activities.

The default `config/databases.yml` should look like the following.

```
all:
  propel:
    class:      sfPropelDatabase
    param:
      dsn:      mysql:host=localhost;dbname=dbname
      username: user
```

*Listing
2-1*

The only difference between Propel and Doctrine here is that the class must be `sfDoctrineDatabase` instead of `sfPropelDatabase` and the connection name is `doctrine` instead of `propel`. Both Doctrine and Propel use PHP Data Objects (PDO) as the database abstraction layer.



Though Propel requires at least one connection named `propel`, Doctrine does not require that the connection be named `doctrine` so you can name it whatever you like.

You can configure the connections in `config/databases.yml` with the `configure:database` task like the following.

```
$ ./symfony configure:database --name=doctrine --class=sfDoctrineDatabase
"mysql:host=localhost;dbname=dbname" user secret
```

*Listing
2-2*

Now you will see a new connection defined like the following:

```
doctrine:
  class: sfDoctrineDatabase
  param:
    dsn: 'mysql:host=localhost;dbname=dbname'
    username: user
    password: secret
```

*Listing
2-3*



You need to completely remove the references to propel in `config/databases.yml` if you have the `sfPropelPlugin` disabled.

Supported Drivers

Doctrine supports all drivers which PDO supports. PHP must be compiled with both PDO and the `PDO_*` drivers you wish to use. Below is a list of databases PDO will work with.

Name	Description
MS SQL Server	Microsoft SQL Server and Sybase Functions (PDO_DBLIB)
Firebird/Interbase	Firebird/Interbase Functions (PDO_FIREBIRD)
IBM	IBM Functions (PDO_IBM)
Informix	Informix Functions (PDO_INFORMIX)
MySQL	MySQL Functions (PDO_MYSQL)
Oracle	Oracle Functions (PDO_OCI)
ODBC and DB2	ODBC and DB2 Functions (PDO_ODBC)
PostgreSQL	PostgreSQL Functions (PDO_PGSQL)
SQLite	SQLite Functions (PDO_SQLITE)



You can read more about PDO at <http://www.php.net/pdo>².

Data Source Name(DSN)

Doctrine offers two ways of specifying your DSN information. You can use the Doctrine style DSN or use the native PDO style.

Doctrine Style

Doctrine has a DSN syntax which is based off of PEAR MDB2.

Listing
2-4

```
all:
  doctrine:
    class:          sfDoctrineDatabase
    param:
      dsn:          driver://username:password@host/database_name
```

PDO Style

You may alternatively specify your DSN information in the PDO style syntax.

Listing
2-5

```
all:
  doctrine:
    class:          sfDoctrineDatabase
    param:
```

2. <http://www.php.net/pdo>

```
dsn:          driver:dbname=database_name;host=localhost
username:     username
password:     password
```



Using the PDO style syntax offers more flexibility and ability to specify non standard information about your connection to PDO. For example, when specifying non standard unix socket paths or ports to use when connecting, specifying it in PDO syntax is more flexible. The `configure:database` command also only works with the PDO style.

Import from Database

Doctrine has the ability to generate a schema file in `config/doctrine/schema.yml` from an existing database. Just configure your Doctrine connection for the database you wish to import and run the following command.



This is a good way to convert your Propel schema to Doctrine. Simply create your database using propel, and then generate the schema in Doctrine from your created database.

```
$ ./symfony doctrine:build-schema
>> doctrine generating yaml schema from database
```

*Listing
2-6*

Now have a look in `config/doctrine/schema.yml` and you will see the yaml for the database. In this example we have a user table.

```
CREATE TABLE user (id BIGINT AUTO_INCREMENT, username VARCHAR(255),
password VARCHAR(255), PRIMARY KEY(id)) ENGINE = INNODB;
```

*Listing
2-7*

The above mysql table would generate a yaml schema like the following in `config/doctrine/schema.yml`

```
User:
  tableName: user
  columns:
    id:
      type: integer(8)
      primary: true
      autoincrement: true
    username: string(255)
    password: string(255)
```

*Listing
2-8*

Multiple Connections

Doctrine offers the ability to have multiple connections. You can easily bind models to connections so that queries are executed on the appropriate connection. So first we need to add multiple connections with the `configure:database` command like the following.

```
$ ./symfony configure:database --name=master --class=sfDoctrineDatabase
"mysql:host=localhost;dbname=master" user secret
$ ./symfony configure:database --name=client --class=sfDoctrineDatabase
"mysql:host=localhost;dbname=client" user secret
```

*Listing
2-9*

Remove the original connection we created and your `config/databases.yml` will look like the following.

Listing 2-10

```
all:
  master:
    class: sfDoctrineDatabase
    param:
      dsn: 'mysql:host=localhost;dbname=master'
      username: user
      password: secret
  client:
    class: sfDoctrineDatabase
    param:
      dsn: 'mysql:host=localhost;dbname=client'
      username: user
      password: secret
```

Now say we have a `Client` model which you want to bind to the master database. You can simply do this directly in the definition of the model like below. Place the following YAML code in `config/doctrine/schema.yml`

Listing 2-11

```
Client:
  connection: master
  columns:
    name: string(255)
    username: string(255)
    password: string(255)
```

Now each `Client` can have `Stores` but they are saved in a separate database from the `Clients`.

Listing 2-12

```
Store:
  connection: client
  attributes:
    export: tables
  columns:
    name: string(255)
    description: string(500)
    client_id: integer
  relations:
    Client:
      foreignAlias: Stores
```



Because the tables are in separate databases the data can only be lazily loaded. Doctrine does not currently support generating sql for joining tables across databases. Also, notice the export attribute being set to tables. This tells Doctrine to only export the create table statement and not any foreign key constraints.

Connection Attributes

`sfDoctrinePlugin` allows you to specify connection attributes directly in the `config/databases.yml` file like the following.

Listing 2-13

```
doctrine:
  class: sfDoctrineDatabase
```

```

param:
  dsn: 'mysql:host=localhost;dbname=dbname'
  username: user
  password: secret
  attributes:
    use_dql_callbacks: true

```

The attributes you specify here will be set on the `Doctrine_Connection` instances when the connection is created.



Attributes in Doctrine are for configuring and controlling features. You can read more about attributes in the Doctrine documentation³.

Build Everything

Now that we have our connections and schema defined we can build everything with the following command.

```
$ ./symfony doctrine:build-all-reload
```

*Listing
2-14*

```

This command will remove all data in your database.
Are you sure you want to proceed? (y/N)

```

```

y
>> doctrine  dropping databases
>> doctrine  creating databases
>> doctrine  generating model classes
>> doctrine  generating sql for models
>> doctrine  generating form classes
>> doctrine  generating filter form classes
>> doctrine  created tables successfully
>> doctrine  loading data fixtures from "/Us...ymfony12doctrine/data/
fixtures"

```

Running the above commands is equal to running the following commands separately.

```
$ ./symfony doctrine:drop-db
```

*Listing
2-15*

```

This command will remove all data in your database.
Are you sure you want to proceed? (y/N)

```

```

y
>> doctrine  dropping databases
$ ./symfony doctrine:build-db
>> doctrine  creating databases
$ ./symfony doctrine:build-model
>> doctrine  generating model classes
$ ./symfony doctrine:build-sql
>> doctrine  generating sql for models
$ ./symfony doctrine:build-form
>> doctrine  generating form classes
$ ./symfony doctrine:build-filters
>> doctrine  generating filter form classes

```

3. <http://www.doctrine-project.com/documentation/manual/1.0/en/configuration>

```
$ ./symfony doctrine:insert-sql
>> doctrine  created tables successfully
$ ./symfony doctrine:data-load
>> doctrine  loading data fixtures from "/Us...ymfony12doctrine/data/
fixtures"
```



You can take a look at the models which were generated from your YAML schema files in `lib/model/doctrine` and `lib/model/doctrine/base`. The files in the generated folder are rewritten each time you build your models whereas the ones below the base directory are not. You may customize your models by editing the classes in `lib/model/doctrine`.

Here is what the `lib/model/doctrine/base/BaseClient.class.php` should look like.

Listing
2-16

```
<?php
// Connection Component Binding
Doctrine_Manager::getInstance()->bindComponent('Client', 'master');

/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
abstract class BaseClient extends sfDoctrineRecord
{
    public function setTableDefinition()
    {
        $this->setTableName('client');
        $this->hasColumn('name', 'string', 255, array('type' => 'string',
'length' => '255'));
        $this->hasColumn('username', 'string', 255, array('type' => 'string',
'length' => '255'));
        $this->hasColumn('password', 'string', 255, array('type' => 'string',
'length' => '255'));
    }

    public function setUp()
    {
        $this->hasMany('Store as Stores', array('local' => 'id',
'foreign' => 'client_id'));
    }
}
```



It is common practice to run the `./symfony doctrine:build-all-reload-test-all` command when developing. This will rebuild your entire environment and run the full test suite. This is a good command to run before committing new code to ensure no new regressions have occurred.



More can be read about connections in the Doctrine Manual here⁴.

4. <http://www.doctrine-project.org/documentation/manual/1-0/en/connections>

Chapter 3

Configuration

Attributes

Doctrine controls features and settings with attributes. The attributes can be defined at different levels of a hierarchy. Some attributes can be specified at all levels and others cannot. Below explains how you can specify attributes at each level. Attributes can be specified globally, on each connection, or on each individual model.

In symfony you can control the Doctrine configuration in your `config/ProjectConfiguration.class.php` or `apps/appname/config/appnameConfiguration.class.php`

Global

You can control global attributes by creating a `configureDoctrine()` in your configuration. All global attributes are set on the `Doctrine_Manager` singleton instance. This method is invoked when the `sfDoctrinePlugin config.php` is loaded. This is before any connections exist so only `Doctrine_Manager` attribute can be controlled at this point.

```
public function configureDoctrine(Doctrine_Manager $manager)
{
    $manager->setAttribute('use_dql_callbacks', true);
    $manager->setAttribute('use_native_enum', true);
}
```

*Listing
3-1*

All Connections

You can control per connection attributes by creating a `configureDoctrineConnection()` in your configuration class. This method is invoked in `sfDoctrineDatabase` as each connection is instantiated by symfony in the order they exist in `config/databases.yml`.

```
public function configureDoctrineConnection(Doctrine_Connection
$connection)
{
    $connection->setAttribute('use_dql_callbacks', true);
    $connection->setAttribute('use_native_enum', true);
}
```

*Listing
3-2*

You can also optionally specify connection attributes directly on the connection definition in `config/doctrine/databases.yml` like the following:

```

Listing 3-3 doctrine:
  class: sfDoctrineDatabase
  param:
    dsn: 'mysql:host=localhost;dbname=dbname'
    username: user
    password: secret
    attributes:
      use_dql_callbacks: true
      use_native_enum: true

```

You may also want to have a different configuration for each connection so you can create a specific function that is also invoked on each individual connection. If you have a connection named `master` then you will need to create a function named `configureDoctrineConnectionMaster()` in your `config/ProjectConfiguration.class.php` file.

```

Listing 3-4 public function configureDoctrineConnectionMaster(Doctrine_Connection
  $connection)
{
  $connection->setAttribute('use_dql_callbacks', false);
  $connection->setAttribute('use_native_enum', false);
}

```

In the above example we have enabled `use_dql_callbacks` and `use_native_enum` for every connection except the connection named `master` by enabling it for all connections and disabling the attributes specifically for that connection.

Model

The last level of the hierarchy for Doctrine is models. The attributes can be specified directly in the YAML definition of the model.

```

Listing 3-5 Store:
  connection: client
  attributes:
    export: tables
  columns:
    name: string(255)
    description: string(500)

```

You can also set attributes using php code in the generated model classes in `lib/model/doctrine`. Check out `lib/model/doctrine/Store.class.php` and override the `setTableDefinition()` to specify some additional attributes.

```

Listing 3-6 public function setTableDefinition()
{
  parent::setTableDefinition();
  $this->setAttribute('export', 'tables');
}

```

Configuring Model Building

`sfDoctrinePlugin` offers the ability to override some of the default model building options. These settings can be controlled using the `sfConfig` class using the parameter named `doctrine_model_builder_options`.

Here is an example of how you can change the base class used when generating models. You can set it to use a class named `myDoctrineRecord` or whatever you want. Just make sure that a class exists somewhere in your project for the symfony autoloading to find.

```
public function configureDoctrine(Doctrine_Manager $manager)
{
    $options = array('baseClassName' => 'myDoctrineRecord');
    sfConfig::set('doctrine_model_builder_options', $options);
}
```

*Listing
3-7*

Make sure you create the class. For example, `sfproject/lib/myDoctrineRecord.class.php` with the following php code.

```
class myDoctrineRecord extends sfDoctrineRecord
{
}
```

*Listing
3-8*

Now when you generate your models, all the classes will extend from `myDoctrineRecord` instead of `sfDoctrineRecord` so you can add custom functionality to all your models.

Here is a list of all the other options which can be changed to different values for the model building process.

Name	Description	Default
suffix	Suffix to use for generated classes	.class.php
generateBaseClasses	Whether or not to generate base classes	true
generateTableClasses	Whether or not to generate *Table classes	true
baseClassPrefix	Word to prefix base classes with	Base
baseClassesDirectory	Directory to generate base classes in	base
baseClassName	Super parent to extend models from	sfDoctrineRecord

Custom Doctrine Library Path

With `sfDoctrinePlugin` it is easy to swap out the version of Doctrine used by simply changing one configuration value.

Below you will find an example of how you can configure `sfDoctrinePlugin` to use a different version of Doctrine, for example 1.0.10.

First we need to check out the version of Doctrine we want to use into `lib/vendor/doctrine`:

```
$ mkdir lib/vendor
$ cd lib/vendor
$ svn co http://svn.doctrine-project.org/tags/1.0.10/lib doctrine
```

*Listing
3-9*

Now we can configure the `sfDoctrinePlugin` to use that version of Doctrine instead of the one that comes bundled with the plugin. In your `ProjectConfiguration::setup()` method you need to change the value of the `sfDoctrinePlugin_doctrine_lib_path` with `sfConfig`, like the following:

```
public function setup()
{
    sfConfig::set('sfDoctrinePlugin_doctrine_lib_path',
```

*Listing
3-10*

```
sfConfig::get('sf_lib_dir') . '/vendor/doctrine/Doctrine.php');  
}
```



Not all versions of Doctrine are compatible with the Symfony `sfDoctrinePlugin`. For example using a different major version of Doctrine that is not bundled with the `sfDoctrinePlugin` is not guaranteed to work well, if at all.

TIP More can be read about configuration in the Doctrine Manual here⁵.

5. <http://www.doctrine-project.org/documentation/manual/1.0/en/configuration>

Chapter 4

Schema Files

In the previous chapters you've seen some various syntaxes for specifying your schema information in YAML files placed in `config/doctrine`. This chapter explains the syntaxes and how to specify all your schema meta data in YAML format.

Data Types

Doctrine offers several column data types. When you specify the portable Doctrine type it is automatically converted to the appropriate type of the DBMS you are using. Below is a list of the available column types that can be used as well as the type it is translated to when using the MySQL and pgSQL DBMS engines.



Doctrine data types are standardized and made portable across all DBMS. For the types that the DBMS do not support natively, Doctrine has the ability to convert the data on the way in to the and on the way out of the database. For example the Doctrine `array` and `object` types are `serialized()` on the way in and `unserialized()` on the way out.

Type	MySQL Type	pgSQL Type
integer	integer	int/serial
integer(1)	tinyint	smallint/serial
integer(2)	smallint	smallint/serial
integer(3)	mediumint	int/serial
integer(4)	int	int/serial
integer(5)	bigint	bigint/bigserial
float	double	float
double	double	float
decimal	decimal	numeric
char	char	char
varchar	varchar	varchar
string	varchar	varchar
array	text	text
object	text	text
blob	longblob	bytea
blob(255)	tinyblob	bytea

Type	MySQL Type	pgSQL Type
blob(65532)	blob	bytea
blob(16777215)	mediumblob	bytea
clob	longtext	text
clob(255)	tinytext	text
clob(65532)	text	text
clob(16777215)	mediumtext	text
timestamp	datetime	timestamp without timezone
time	time	time without timezone
date	date	date
gzip	text	text
boolean	tinyint(1)	boolean
bit	bit	varbit
varbit	n/a	varbit
inet	n/a	inet
enum	-see below-	-see below-

- Char(length) is used for string if “fixed” parameter is true. Length defaults to 255 if not provided.
- Any value up to and including the number shown in brackets will produce the specified column type
- Any integer with size greater than 4 will produce “bigint”
- In postgres, serial is used if “autoincrement” is set
- In Doctrine >= 1.1, timezone is not specified

The Doctrine enum type can either be emulated or you can use the native enum type if your DBMS supports it. It is off by default so you will need to enable an attribute to use native enums.

Before we enable the attribute Doctrine will generate SQL like the following and simply emulate the enum type and will make sure the value you specify is one of the valid specified values.

```
CREATE TABLE user (id BIGINT AUTO_INCREMENT, username VARCHAR(255),
password VARCHAR(255), user_type VARCHAR(255), PRIMARY KEY(id)) ENGINE =
INNODB;
```

Listing
4-1

Now lets specify the `use_native_enum` attribute on our connection so that Doctrine knows to generate the native enum sql for your DBMS.

```
all:
  doctrine:
    class: sfDoctrineDatabase
    param:
      dsn: 'mysql:host=localhost;dbname=symfony12doctrine'
      username: user
      attributes:
        use_native_enum: true
```

Listing
4-2

Now that we have enabled the attribute Doctrine generates the following SQL under MySQL:

```
CREATE TABLE user (id BIGINT AUTO_INCREMENT, username VARCHAR(255),
password VARCHAR(255), user_type ENUM('Normal', 'Administrator'), PRIMARY
KEY(id)) ENGINE = INNODB;
```

Listing
4-3

Below is a sample yaml schema file that implements each of the different column types.

```
User:
  columns:
    id:
      type: integer(4)
      primary: true
      autoincrement: true
    username: string(255)
    password: string(255)
    latitude: float
    longitude: float
    hourly_rate:
      type: decimal
      scale: 2
    groups_array: array
    session_object: object
    description: clob
    profile_image_binary_data: blob
    created_at: timestamp
    time_last_available: time
    date_last_available: date
    roles:
      type: enum
      values: [administrator, moderator, normal]
```

Listing
4-4

```

    default: normal
    html_header: gzip

```

Generates the following SQL with MySQL:

```

Listing 4-5 CREATE TABLE user (id INT AUTO_INCREMENT, username VARCHAR(255), password
VARCHAR(255), latitude DOUBLE, longitude DOUBLE, hourly_rate
DECIMAL(18,2), groups_array TEXT, session_object TEXT, description
LONGTEXT, profile_image_binary_data LONGBLOB, created_at DATETIME,
time_last_available TIME, date_last_available DATE, roles
ENUM('administrator', 'moderator', 'normal') DEFAULT 'normal', html_header
TEXT, PRIMARY KEY(id)) ENGINE = INNODB;

```

Options

Often you need to set options on your table for controlling things like charset, collation and table type in mysql. These can be controlled easily with options.

```

Listing 4-6 User:
options:
  type: MyISAM
  collate: utf8_unicode_ci
  charset: utf8
columns:
  username: string(255)
  password: string(255)

```

Generates the following SQL with MySQL:

```

Listing 4-7 CREATE TABLE user (id BIGINT AUTO_INCREMENT, username VARCHAR(255),
password VARCHAR(255), PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE
utf8_unicode_ci ENGINE = MyISAM;

```

Indexes

You can optimize your database by defining indexes on columns which are used in conditions on your queries. Below is an example of indexing the username column of a user table since it is common to do lookups on the table by the users username.

```

Listing 4-8 User:
columns:
  username: string(255)
  password: string(255)
indexes:
  username_index:
    fields: [username]
    type: unique

```

Generates the following SQL with MySQL:

```

Listing 4-9 CREATE TABLE user (id BIGINT AUTO_INCREMENT, username VARCHAR(255),
password VARCHAR(255), UNIQUE INDEX username_index_idx (username),
PRIMARY KEY(id)) ENGINE = INNODB;

```

You can also optionally specify unique directly on the column when dealing with single column unique indexes.

```
User:
  columns:
    username:
      type: string(255)
      unique: true
    password: string(255)
```

*Listing
4-10*

Generates the following SQL with MySQL:

```
CREATE TABLE user (id BIGINT AUTO_INCREMENT, username VARCHAR(255) UNIQUE,
password VARCHAR(255), PRIMARY KEY(id)) ENGINE = INNODB;
```

*Listing
4-11*



Indexes are automatically created on relationship foreign keys when the relationships are defined. The next section explains how to define relationships between foreign keys on your tables.

Relationships

Doctrine offers the ability to map the relationships which exist in your database to the ORM so that it can be the most help when working with your data.

One to One

Here is a simple example of how to define a one-to-one relation between a User and Profile model.

```
Profile:
  columns:
    user_id: integer
    name: string(255)
    email_address:
      type: string(255)
      email: true
  relations:
    User:
      local: user_id
      foreign: id
      type: one
      foreignType: one
```

*Listing
4-12*

Generates the following SQL with MySQL:

```
CREATE TABLE profile (id BIGINT AUTO_INCREMENT, user_id BIGINT, name
VARCHAR(255), email_address VARCHAR(255), INDEX user_id_idx (user_id),
PRIMARY KEY(id)) ENGINE = INNODB;
ALTER TABLE profile ADD FOREIGN KEY (user_id) REFERENCES user(id);
```

*Listing
4-13*

One to Many

Here is a simple example of how to define a one-to-many relation between a User and Phonenumbrer model.

Listing 4-14 Phonenumbers:

```

columns:
  user_id: integer
  phonenumbers: string(255)
relations:
  User:
    foreignAlias: Phonenumbers
    local: user_id
    foreign: id
    type: one
    foreignType: many

```

Generates the following SQL with MySQL:

Listing 4-15

```

CREATE TABLE phonenumbers (id BIGINT AUTO_INCREMENT, user_id BIGINT,
phonenumbers VARCHAR(255), INDEX user_id_idx (user_id), PRIMARY KEY(id))
ENGINE = INNODB;
ALTER TABLE phonenumbers ADD FOREIGN KEY (user_id) REFERENCES user(id);

```

Many to Many

Here is a simple example of how to define a many-to-many relation between a BlogPost and Tag model.

Listing 4-16 BlogPost:

```

columns:
  user_id: integer
  title: string(255)
  body: clob
relations:
  User:
    local: user_id
    foreign: id
    type: one
    foreignType: one
    foreignAlias: BlogPosts
  Tags:
    class: Tag
    foreignAlias: BlogPosts
    refClass: BlogPostTag
    local: blog_post_id
    foreign: tag_id

```

```

Tag:
columns:
  name: string(255)

```

```

BlogPostTag:
columns:
  blog_post_id:
    type: integer
    primary: true
  tag_id:
    type: integer
    primary: true
relations:
  BlogPost:
    local: blog_post_id

```

```

    foreign: id
    foreignAlias: BlogPostTags
  Tag:
    local: tag_id
    foreign: id
    foreignAlias: BlogPostTags

```

Generates the following SQL with MySQL:

```

CREATE TABLE blog_post (id BIGINT AUTO_INCREMENT, user_id BIGINT, title
VARCHAR(255), body LONGTEXT, INDEX user_id_idx (user_id), PRIMARY KEY(id))
ENGINE = INNODB;
CREATE TABLE blog_post_tag (blog_post_id BIGINT, tag_id BIGINT, PRIMARY
KEY(blog_post_id, tag_id)) ENGINE = INNODB;
CREATE TABLE tag (id BIGINT AUTO_INCREMENT, name VARCHAR(255), PRIMARY
KEY(id)) ENGINE = INNODB;
ALTER TABLE blog_post ADD FOREIGN KEY (user_id) REFERENCES user(id);
ALTER TABLE blog_post_tag ADD FOREIGN KEY (tag_id) REFERENCES tag(id);
ALTER TABLE blog_post_tag ADD FOREIGN KEY (blog_post_id) REFERENCES
blog_post(id);

```

*Listing
4-17*

Cascading Operations

When saving objects in Doctrine it is cascaded to associated objects by default. Deleting is slightly different. Doctrine has the ability to do both application and database level cascading deletes.

Database Level

Doctrine also has the ability to export cascading operations to the database level. Below is an example of how to setup a model with some cascading options.

```

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

Phonenumber:
  columns:
    user_id: integer
    phonenumber: string(255)
  relations:
    User:
      foreignAlias: Phonenumbers
      local: user_id
      foreign: id
      type: one
      foreignType: many
      onDelete: CASCADE

```

*Listing
4-18*

Generates the following SQL with MySQL:

```

CREATE TABLE phonenumber (id BIGINT AUTO_INCREMENT, user_id BIGINT,
phonenumber VARCHAR(255), INDEX user_id_idx (user_id), PRIMARY KEY(id))
ENGINE = INNODB;

```

*Listing
4-19*

```
ALTER TABLE phonenumber ADD FOREIGN KEY (user_id) REFERENCES user(id) ON
DELETE CASCADE;
```



Database level cascading is specified on the side where the foreign key lives.

Application Level

Unlike the `save()` operations the `delete()` cascading needs to be turned on explicitly. Here is an example:



Application level cascading `save()` and `delete()` does not apply when doing DQL update and delete statements, only when calling `save()` and `delete()` on your objects.

Listing
4-20

```
User:
  columns:
    username: string(255)
    password: string(255)
  relations:
    Phonenumbers:
      class: Phonenumbers
      local: id
      foreign: id
      type: many
      foreignType: one
      cascade: [delete]

Phonenumber:
  columns:
    user_id: integer
    phonenumber: string(255)
  relations:
    User:
      foreignAlias: Phonenumbers
      local: user_id
      foreign: id
      type: one
      foreignType: many
```



Application level cascading deletes differ from database level in that they are defined on the side where the relationship you wish to cascade on is defined. This is different than database level cascades where you always specify it on the side where the foreign key lives.

Behaviors

One great feature of Doctrine is the ability to have plug n' play behavior. These behaviors can be easily included in your model definitions and you inherit functionality automatically.

Core Behaviors

Here is a list of behavior bundled with Doctrine core. You can use any of the behaviors in your models without writing any code.

Name	Description
Geographical	Adds latitude and longitude to your model and offers functionality for calculating miles/kilometers between records.
I18n	Adds internationalization capabilities to your models.
NestedSet	Turn your models in to a traversable tree.
Searchable	Index all the data in your models and make it searchable.
Sluggable	Add a <code>slug</code> field to your models and have it automatically create a slug based on your configuration.
SoftDelete	Never really delete a record. Will simply set a deleted flag instead and filter all deleted records from select queries.
Timestampable	Add a <code>created_at</code> and <code>updated_at</code> column to your models have Doctrine set them when inserting and updating records.
Versionable	Turn your models in to an audit log and record all changes. Offers the ability to revert back to previous versions easily.

You can easily enable a behavior by using the `actAs` functionality. Below is an example of how to use the `Sluggable` behavior.

```
BlogPost:
  actAs:
    Sluggable:
      fields: [title]
      unique: true
  columns:
    user_id: integer
    title: string(255)
    body: clob
```

Listing
4-21

The above example will automatically add a `slug` column to the model and will set the value of the `slug` column based on the value of the `title` column and make sure the value is unique. If a `slug` already exists in the database with the same value then 1, 2, 3, etc. is appended to the end.

Generates the following SQL with MySQL:

```
CREATE TABLE blog_post (id BIGINT AUTO_INCREMENT, user_id BIGINT, title
VARCHAR(255), body LONGTEXT, slug VARCHAR(255), UNIQUE INDEX sluggable_idx
(slug), INDEX user_id_idx (user_id), PRIMARY KEY(id)) ENGINE = INNODB;
```

Listing
4-22



You can also write your own behaviors. Check out the source code of the existing behaviors to get a peek at how they work. They can be found in `SF_ROOT/plugins/sfDoctrinePlugin/lib/doctrine/Doctrine/Template`. And you can read more about Doctrine behaviors in the manual⁶.

Nesting Behaviors

Doctrine offers the ability to easily nest behaviors. For example you may want to have a `Sluggable` behavior on your auto-generated model with the `I18n` behavior.

6. <http://www.doctrine-project.org/documentation/manual/1-0/en/behaviors>

```

Listing 4-23 Gallery:
  actAs:
    I18n:
      fields: [title, description]
    actAs:
      Sluggable:
        fields: [title]
  columns:
    title: string(255)
    description: clob

```

Now the GalleryTranslation model which is automatically generated will have a slug column which is automatically set for you based on the translated title column. You can mix your behaviors together but remember some behaviors will not always play together as they are developed standalone and are not aware of each other.

Inheritance

Another great feature of Doctrine is the ability to use native PHP OOP inheritance with your models. It supports three different inheritance strategies which can be used independently or mixed together. Below are some examples of the different inheritance strategies.

Inheritance Types

Name	Description
Concrete	Each child class has a separate table has all the columns of its parents
Simple	Each child class shares the same table and columns as its parents
Column Aggregation	All columns must be defined in the parent and each child class is determined by a type column

Below are some examples of the three different inheritance strategies supported by Doctrine.

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.

```

Listing 4-24 TextItem:
  columns:
    topic: string(100)

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

```

Generates the following SQL with MySQL:

```

Listing 4-25 CREATE TABLE text_item (id BIGINT AUTO_INCREMENT, topic VARCHAR(100),
PRIMARY KEY(id)) ENGINE = INNODB;
CREATE TABLE comment (id BIGINT AUTO_INCREMENT, topic VARCHAR(100),
content TEXT, PRIMARY KEY(id)) ENGINE = INNODB;

```

Simple Inheritance

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

Entity:

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

*Listing
4-26*

User:

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

Group:

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

Generates the following SQL with MySQL:

```
CREATE TABLE entity (id BIGINT AUTO_INCREMENT, name VARCHAR(30), username
VARCHAR(20), password VARCHAR(16), created BIGINT, PRIMARY KEY(id)) ENGINE
= INNODB;
```

*Listing
4-27*

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 automatically added which tells whether an entity is a group or a user.

Entity:

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

*Listing
4-28*

User:

```
inheritance:
  extends: Entity
  type: column_aggregation
```

Group:

```
inheritance:
  extends: Entity
  type: column_aggregation
```

Generates the following SQL with MySQL:

```
CREATE TABLE entity (id BIGINT AUTO_INCREMENT, name VARCHAR(30), username
VARCHAR(20), password VARCHAR(16), created BIGINT, type VARCHAR(255),
PRIMARY KEY(id)) ENGINE = INNODB;
```

*Listing
4-29*

Global Schema Information

Doctrine schemas allow you to specify certain parameters that will apply to all of the models defined in the schema file. Below you can find an example on what global parameters you can set for schema files.

List of global parameters:

Name	Description
connection	Name of the connection to bind the models to
attributes	Array of attributes to apply to the models
actAs	Array of actAs behaviors and options to enable on the models
options	Array of table options to apply to the models
inheritance	Inheritance options to apply to the models

Here is a sample schema file which implements some global schema information:

Listing
4-30

```

connection: conn_name1
actAs: [Timestampable]
options:
  type: INNODB

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)

```

Generates the following SQL with MySQL:

Listing
4-31

```

CREATE TABLE contact (id INT AUTO_INCREMENT, name VARCHAR(255), created_at
DATETIME, updated_at DATETIME, PRIMARY KEY(id)) ENGINE = INNODB;
CREATE TABLE user (id INT AUTO_INCREMENT, contact_id INT, username
VARCHAR(255), password VARCHAR(255), created_at DATETIME, updated_at
DATETIME, INDEX contact_id_idx (contact_id), PRIMARY KEY(id)) ENGINE =

```

```

INNOODB;
ALTER TABLE user ADD FOREIGN KEY (contact_id) REFERENCES contact(id);

```

All of the settings at the top will be applied to every model which is defined in that yaml file.

Plugin Schemas

With symfony plugins, using Doctrine schemas are no different than using them in your main config/doctrine folder. The plugin should also have the same config/doctrine directory containing YAML files. It is not necessary to specify any package parameter like you have to with Propel. The plugin is smart enough to know it is a part of a plugin because of its location.

The models, forms, filters, etc. are all generated in to sub-folders for the plugin to make organization and maintenance of your models easier. For example in sfDoctrineGuardPlugin sfGuardUser is generated as follows.

```

lib/
  model/
    doctrine/
      sfDoctrineGuardPlugin/
        sfGuardUser.class.php
        sfGuardUserTable.class.php
        base
          BasesfGuardUser.class.php
      form/
        doctrine/
          BaseFormDoctrine.class.php
          sfDoctrineGuardPlugin/
            sfGuardUserForm.class.php
            base
              BasesfGuardUserForm.class.php
  plugins/
    sfDoctrineGuardPlugin/
      lib/
        model/
          doctrine/
            PluginsfGuardUser.class.php
            PluginsfGuardUserTable.class.php
        form/
          doctrine/
            PluginsfGuardUserForm.class.php

```

Listing
4-32

The hierarchy of the generated classes are as follows.

Name	Extends	Description
sfGuardUser	PluginsfGuardUser	Top level model class for all your custom project functionality.
PluginsfGuardUser	BasesfGuardUser	Plugin level model class for functionality bundled with the plugin.
BasesfGuardUser	sfDoctrineRecord	Generated base model class containing schema meta data.
sfGuardUserTable	PluginsfGuardUserTable	Top level table class for custom project functionality.

Name	Extends	Description
PluginsfGuardUserTable	Doctrine_Table	Plugin level table class for functionality bundled with the plugin.
sfGuardUserForm	PluginsfGuardUserForm	Top level form class for all your custom project functionality.
PluginsfGuardUserForm	BasesfGuardUserForm	Plugin level form class for functionality bundled with the plugin.
BasesfGuardUserForm	BaseFormDoctrine	Generated base form class containing form widgets and validators.
BaseFormDoctrine	sfFormDoctrine	Generated base form class which all generated forms extend.

Element Definitions

Below is a list with all the allowed element names and a brief definition for each one.

Root Elements

Name	Description
abstract	Whether or not to make the generated class abstract. Defaults to false. When a class is abstract it is not exported to the database.
className	Name of the class to generate
tableName	Name of the table in your DBMS to use.
connection	Name of the Doctrine_Connection instance to bind the model to.
columns	Column definitions.
relations	Relationship definitions.
indexes	Index definitions.
attributes	Attribute definitions.
actAs	ActAs definitions.
options	Option definitions.
inheritance	Array for inheritance definition
listeners	Array defining listeners to attach
checks	Checks to run at application level as well as exporting to your DBMS

Columns

Name	Description
name	Name of the column.
fixed	Whether or not the column is fixed.
primary	Whether or not the column is a part of the primary key.
autoincrement	Whether or not the column is an autoincrement column.
type	Doctrine data type of the column
length	Length of the column

Name	Description
default	Default value of the column
scale	Scale of the column. Used for the <code>decimal</code> type.
values	List of values for the <code>enum</code> type.
comment	Comment for the column.
sequence	Sequence definition for column.
zerofill	Whether or not to make the column fill empty characters with zeros
extra	Array of extra information to store with the column definition
unsigned	Unsigned modifiers for some field definitions, although not all DBMS's support this modifier for integer field types.

Relations

Name	Description
class	Name of class to use for relationship.
alias	Alias to use to identify relationship.
type	The relationship type. Value can be either <code>one</code> or <code>many</code> and it defaults to <code>one</code> .
refClass	Middle reference class to use for many to many relationships.
local	The local field name used in the relationship.
foreign	the foreign field name used in the relationship.
foreignAlias	The alias of the opposite end of the relationship. Only allowed when <code>autoComplete</code> is set to <code>true</code> .
foreignType	The type of the opposite end of the relationship. Only allowed when <code>autoComplete</code> is set to <code>true</code> .
autoComplete	Whether or not to add the relationship to the opposite end making it bi-directional. Defaults to <code>true</code> .
cascade	Application level cascading options.
onDelete	Database level cascading delete value.
onUpdate	Database level cascading update value.
equal	Whether or not the relationship is a equal nested many to many.
owningSide	-
refClassRelationAlias	-

Inheritance

Name	Description
type	Type of inheritance to use. Allowed values are <code>concrete</code> , <code>column_aggregation</code> , and <code>simple</code> .
extends	Name of the class to extend.
keyField	Name of the field to use as the key for <code>column_aggregation</code> inheritance.
keyValue	Value to fill the <code>keyField</code> with for <code>column_aggregation</code> inheritance.

Indexes

Name	Description
-------------	--------------------

name	Name of the index to create.
------	------------------------------

fields	Array of fields to use in the index.
--------	--------------------------------------

unique	Whether or not the index is unique.
--------	-------------------------------------



More can be read about schema files in the Doctrine Manual [here](http://www.doctrine-project.org/documentation/manual/1_0/en/yaml-schema-files)⁷.

⁷ http://www.doctrine-project.org/documentation/manual/1_0/en/yaml-schema-files

Chapter 5

Data Fixtures

Introduction

Doctrine offers the ability to load small sets of sample test data by using a simple YAML syntax for specifying data to be loaded in to your object relationship hierarchy. It supports easily creating information for your tables and linking foreign keys between records.



The examples demonstrated in this chapter use the following simple User and Phonenummer schema which should be placed in `config/doctrine/schema.yml`.

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

Phonenumber:
  columns:
    user_id: integer
    phonenumber: string(25)
  relations:
    User:
      foreignAlias: Phonenummers

Profile:
  columns:
    name: string(255)
    about: string(500)
    user_id: integer
  relations:
    User:
      foreignType: one
```

*Listing
5-1*



In `sfDoctrinePlugin`, when linking records in data fixtures you use the relationship name, unlike `sfPropelPlugin` where you use the foreign key name. You also have the ability to specify the data fixtures inline. Meaning, a block of YAML that represents a `Doctrine_Record` instance can have nested data structures that define the relationship graph for that `Doctrine_Record` child. Later in this chapter will demonstrate both the original and inline style data fixtures.

Original

Create `data/fixtures/user.yml` and load the following YAML code.

```
Listing 5-2 User:
  User_1:
    username: jwage
    password: changeme
  User_2:
    username: fabpot
    password: changeme
  User_3:
    username: dwhittle
    password: changeme
```

Run the following commands to rebuild the database.

```
Listing 5-3 $ ./symfony doctrine:build-all-reload
```

Now run a simple DQL query to inspect that the data was loaded properly.

```
Listing 5-4 $ ./symfony doctrine:dql "FROM User u"
>> doctrine executing dql query
DQL: FROM User u
found 3 results
-
  id: '1'
  username: jwage
  password: changeme
-
  id: '2'
  username: fabpot
  password: changeme
-
  id: '3'
  username: dwhittle
  password: changeme
```



Setting Date Values in Data Fixtures

The `sfYaml` parser will automatically convert valid dates in to unix timestamps unless you specifically wrap it in single quotes forcing it to be a string type as far as the parser is concerned. If you do not use single quotes when setting `date` or `timestamp` column types the Doctrine validation will fail because of the value being passed to the `Doctrine_Record` being a unix timestamp.

Here is an example of how you can set the `created_at` column of a `User` model.

```
Listing 5-5 User:
  User_1:
    username: jwage
    password: changeme
    created_at: '2008-12-17 00:01:00'
```

Linking Relationships

Create `data/fixtures/phonenumbers.yml` and load the following YAML data fixtures.

```
Phonenumber:
  Phonenumber_1:
    phonenumber: 6155139185
    User: User_1
  Phonenumber_2:
    phonenumber: 1234567890
    User: User_2
  Phonenumber_3:
    phonenumber: 0987654321
    User: User_3
```

*Listing
5-6*

Rebuild the database and run another DQL query to inspect the loaded data fixtures.

```
$ ./symfony doctrine:build-all-reload
```

*Listing
5-7*

Now inspect the data with a more complex query that joins the User Phonenumber records.

```
$ ./symfony doctrine:dql "FROM User u, u.Phonenumbers p"
>> doctrine executing dql query
DQL: FROM User u, u.Phonenumbers p
found 3 results
-
  id: '1'
  username: jwage
  password: changeme
  Phonenumbers:
    -
      id: '1'
      phonenumber: 6155139185
      user_id: '1'
    -
      id: '2'
      phonenumber: 1234567890
      user_id: '2'
    -
      id: '3'
      phonenumber: 0987654321
      user_id: '3'
```

*Listing
5-8*

Many to Many

Use the following YAML schema file in `config/doctrine/schema.yml` with the example data fixtures.

```

Listing 5-9
BlogPost:
  columns:
    title: string(255)
    body: clob
  relations:
    Tags:
      class: Tag
      refClass: BlogPostTag
      foreignAlias: BlogPosts

BlogPostTag:
  columns:
    blog_post_id:
      type: integer
      primary: true
    tag_id:
      type: integer
      primary: true
  relations:
    BlogPost:
      foreignAlias: BlogPostTags
    Tag:
      foreignAlias: BlogPostTags

Tag:
  columns:
    name: string(255)

```

Load the below data fixtures in to `data/fixtures/data.yml`

```

Listing 5-10
BlogPost:
  BlogPost_1:
    title: Test Blog Post
    body: This is the body of the test blog post
    Tags: [test, php, doctrine, orm]

Tag:
  test:
    name: test
  php:
    name: php
  doctrine:
    name: doctrine
  orm:
    name: orm

```

Rebuild the database again and run a DQL query to see the loaded data.

```

Listing 5-11
$ ./symfony doctrine:build-all-reload

```

Now inspect the data with another DQL query that fetches all `BlogPost` records and the related `Tags`

```

$ ./symfony doctrine:dql "FROM BlogPost p, p.Tags"
>> doctrine executing dql query
DQL: FROM BlogPost p, p.Tags
found 1 results
-
  id: '1'
  title: 'Test Blog Post'
  body: 'This is the body of the test blog post'
  Tags:
  -
    id: '1'
    name: test
  -
    id: '2'
    name: php
  -
    id: '3'
    name: doctrine
  -
    id: '4'
    name: orm

```

*Listing
5-12*

Inline

Doctrine offers the ability to specify data fixture relationships inline like below.

```

User:
  User_1:
    username: jwage
    password: changeme
    Phonenumbers:
      Phonenummer_1:
        6155139185

BlogPost:
  BlogPost_1:
    title: Test Blog Post
    body: This is the body of the test blog post
    Tags:
      test:
        name: test
      php:
        name: php
      doctrine:
        name: doctrine
      orm:
        name: orm

```

*Listing
5-13*

This alternative syntax can greatly reduce the length and complexity of your data fixtures.



More can be read about data fixtures in the Doctrine Manual here⁸.

8. <http://www.doctrine-project.org/documentation/manual/1-0/en/data-fixtures>

Chapter 6

Working with Data

Retrieving Data

In Doctrine you are able to retrieve complex results from your RDBMS and hydrate them into array or object data structures which represent your relationship structure. This is done by using the Doctrine Query Language. It is the best way to retrieve all your data in the lowest amount of queries possible. For convenience when working with single tables we offer some simple finder methods as well that dynamically build and execute these queries.

DQL

Doctrine uses DQL for retrieving data and offers a complete `Doctrine_Query` API for building them. Below you'll find a complete list of the methods that can be used as well as examples utilizing all of them.

Query API

Common API

Function Name	SQL	Appends	Description
<code>where('u.username = ?', 'jwage')</code>	<code>u.username = ?</code>	No	Set the <code>WHERE</code> and override any existing <code>WHERE</code> conditions
<code>andWhere('u.username = ?', 'jwage')</code>	<code>AND u.username = ?</code>	Yes	Add a <code>WHERE</code> condition that is appended with an <code>AND</code>
<code>whereIn('u.id', array(1, 2, 3))</code>	<code>AND u.id IN (?, ?, ?)</code>	Yes	Add a <code>AND IN WHERE</code> condition that is appended
<code>andWhereIn('u.id', array(1, 2, 3))</code>	<code>^</code>	Yes	Convenience/proxy method for <code>whereIn()</code>
<code>orWhereIn('u.id', array(1, 2, 3))</code>	<code>OR u.id IN (?, ?, ?)</code>	Yes	Add a <code>OR IN WHERE</code> condition that is appended
<code>whereNotIn('u.id', array(1, 2, 3))</code>	<code>AND u.id NOT IN (?, ?, ?)</code>	Yes	Add a <code>AND NOT IN WHERE</code> condition that is appended
<code>andWhereNotIn('u.id', array(1, 2, 3))</code>	<code>^</code>	Yes	Convenience/proxy method for <code>whereNotIn()</code>
<code>orWhereNotIn('u.id', array(1, 2, 3))</code>	<code>OR u.id NOT IN (?, ?, ?)</code>	Yes	Add a <code>OR NOT IN WHERE</code> condition that is appended

Function Name	SQL	Appends	Description
orWhere('u.username = ?', 'jwage')	OR u.username = ?	Yes	Add a OR WHERE condition that is appended
groupBy('u.id')	GROUP BY u.id, u.username	No	Set the GROUP BY and override any existing GROUP BY
addGroupBy('u.username')	GROUP BY u.username	Yes	Add a GROUP BY that is appended
having('num_phonenumbers > 0')	HAVING num_phonenumbers > 0	No	Set the HAVING and override any existing HAVING
addHaving('u.username = ?', 'jwage')	HAVING u.username = ?	Yes	Add a HAVING that is appended

Select API

Function Name	Description
distinct(\$flag = true)	Set the flag to be a distinct select
select('u.id, u.username, COUNT(p.id) as num_phonenumbers')	Set the SELECT and override any existing select
addSelect('u.email_address')	Add a select that is appended
from('User u, u.Phonenum p')	Set the FROM and override any existing FROM and joins
leftJoin('u.Phonenum p')	Add a LEFT JOIN that is appended to the FROM
innerJoin('u.Profile p')	Add a INNER JOIN that is appended to the FROM
addFrom('u.Phonenum p')	Add a FROM join that is appended to the FROM
orderBy('u.username')	Set the ORDER BY and override any existing ORDER BY
addOrderBy('u.is_active = ?', 1)	Add a ORDER BY that is appended
limit(20)	Set the number of records to limit the result set to
offset(5)	Set the number to offset the limit of records from

Update API

Function Name	Description
forUpdate(\$flag = true)	Change a query to use FOR UPDATE
update('User u')	Specify the model name to UPDATE
set('u.username = ?', 'jwage')	Set new values for the UPDATE query. The first argument is the data to modify, the second is the expression to put directly in the DQL string (can be ? or a DBMS function), and the third is the new value.

Delete API

Function Name	Description
delete()	Change a query to be a delete

Create New Query

Create new query from Doctrine_Table instance.

```
Listing 6-1 $q = Doctrine::getTable('User')->createQuery('u')
            ->where('u.username = ?', 'jwage');
```

Create new query manually

```
Listing 6-2 $q = Doctrine_Query::create()
            ->from('User u')
            ->where('u.username = ?', 'jwage');
```



The above two queries are identical, the first simply does the 2nd code internally as a convenience to you.

Example Queries

Below you will find a few example queries which you can learn from and see how to retrieve result sets in Doctrine.

Calculated Columns

When using DBMS functions to calculate columns, they are hydrated in to the component/model that is the first involved in the expression. In the example below, the model is hydrated in to the Phonenumbers relation because it is the first component encountered in the query.

```
Listing 6-3 $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]['num_phonenumbers'];
```

Retrieve Users and the Groups they belong to

```
Listing 6-4 $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 6-5 $q = Doctrine_Query::create()
            ->from('User u')
            ->where('u.username = ?', 'jwage');
```

```
$users = $q->fetchArray();
```

Multiple WHERE with multiple parameters values

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

$users = $q->fetchArray();
```

*Listing
6-6*

// You can also optionally use the andWhere() to add to the existing where parts

```
$q = Doctrine_Query::create()
  ->from('User u')
  ->where('u.is_active = ?', 1)
  ->andWhere('u.is_online = ?', 1);

$users = $q->fetchArray();
```

Using whereIn() convenience method

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

$users = $q->fetchArray();
```

*Listing
6-7*

// 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

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

*Listing
6-8*

```
$user = $q->fetchOne();
```

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

```
$user = $q->fetchOne();
```

Limiting resultsets using aggregate functions

```
// 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');
```

*Listing
6-9*

```
->having('num_phonenumbers > 1')
->groupBy('u.id');
```

```
$users = $q->fetchArray();
```

Join only primary phonenumbers using WITH

```
Listing 6-10 $q = Doctrine_Query::create()
->from('User u')
->leftJoin('u.Phonenumbers p WITH p.primary_num = ?', true);

$users = $q->fetchArray();
```

Override JOIN condition using ON

```
Listing 6-11 $q = Doctrine_Query::create()
->from('User u')
->leftJoin('u.Phonenumbers p ON u.id = p.user_id AND p.primary_num = ?',
true);

$users = $q->fetchArray();
```

Selecting certain columns for optimization

```
Listing 6-12 $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 6-13 // Select all User columns but only the phone phonenumbers 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 6-14 // Delete phonenumbers for user id = 5
$deleted = Doctrine_Query::create()
->delete()
->from('Phonenumber')
->andWhere('user_id = 5')
->execute();
```

Perform simple DQL update for a column

```
Listing 6-15 // Set user id = 1 to active
Doctrine_Query::create()
->update('User u')
->set('u.is_active', '1', true)
->where('u.id = 1', 1)
->execute();
```

Perform DQL update with DBMS functions

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

Listing
6-16**Using mysql LIKE to search for records**

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

$users = $q->fetchArray();
```

Listing
6-17**Use the INDEXBY keyword to hydrate the data where the key of record entry is the name of the column you assign**

```
$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
```

Listing
6-18**Using positional and named parameters**

```
// 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();
```

Listing
6-19**Using subqueries in your WHERE**

```
// Find users 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();
```

Listing
6-20

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

$users = $q->fetchArray();
```

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

Listing
6-21

```
$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();
```

Finders

Doctrine offers some simple magic finder methods that automatically create Doctrine_Query objects in the background. Below are some examples of how you can utilize these methods.

Magic Find By Methods

You can utilize the magic `findBy*()` and `findOneBy*()` methods to find records by single fields value.

Listing
6-22

```
$user = Doctrine::getTable('User')->findOneByUsername('jwage');
$users = Doctrine::getTable('User')->findByIsActive(1);
```

Find by Identifier

The `Doctrine_Table::find()` method is for finding records by its primary key. It works for both models that have surrogate or composite primary keys.

Listing
6-23

```
$user = Doctrine::getTable('User')->find(1);
$userGroup = Doctrine::getTable('UserGroup')->find(array(1, 2));
```

Altering Data

With Doctrine you can alter data by issuing direct DQL update and delete queries or you can fetch objects, alter properties and save. Below we'll show examples of both strategies.

Object Properties

Doctrine offers 3 ways to alter your object properties and sfDoctrinePlugin implements a fourth. They are object access, array access, function access and propel style access.

```
$user = new User();
$user->username = 'jwage';           // Object
$user['username'] = 'jwage';        // Array
$user->set('username', 'jwage');     // Function
$user->setUsername('jwage');         // Propel access
$user->save();
```

*Listing
6-24*

Overriding Accessors and Mutators

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

    public function getUsername()
    {
        return 'PREFIX_' . $this->_get('username');
    }
}
```

*Listing
6-25*

```
$user = new User();
$user->username = 'jwage';
$user->password = 'changeme'; // Invokes setPassword()
echo $user->username; // Invokes getUsername() and returns PREFIX_jwage
```

Working with Relationships

With Doctrine it is easy to manipulate the data in your object graph by utilizing PHP objects.

User hasOne Profile

```
$user = new User();
$user->username = 'jwage';
$user->password = 'changeme';
$user->Profile->name = 'Jonathan H. Wage';
$user->Profile->about = 'His name is Jonathan';
$user->save();
```

*Listing
6-26*

User hasMany Phonenumbers as Phonenumbers

```
$user = new User();
$user->username = 'jwage';
$user->password = 'changeme';
$user->Phonenumbers[]->phonenumbers = '6155139185';
```

*Listing
6-27*

```
$user->Phonenumbers[]->phonenummer = '1234567890';
$phonenummer = $user->Phonenumbers[2];
$phonenummer->phonenummer = '0987654321';
```

BlogPost hasMany Tag as Tags

```
Listing 6-28 $blogPost = new BlogPost();
$blogPost->title = 'Test blog post';
$blogPost->body = 'This is the content of the test blog post';
$tag = Doctrine::getTable('Tag')->findOneByName('doctrine');
if ( ! $tag) {
    $tag = new Tag;
    $tag->name = 'doctrine';
}
$blogPost->Tags[] = $tag;
```

The above code is ugly, we should extract that logic to our TagTable child class which is located in `lib/model/doctrine/TagTable.class.php`.

```
Listing 6-29 class TagTable extends Doctrine_Table
{
    public function findOneByName($name)
    {
        $tag = $this->findOneBy('name', $name);
        if ( ! $tag) {
            $tag = new Tag();
            $tag->name = $name;
        }
        return $tag;
    }
}
```

Now the first example can be simplified some.

```
Listing 6-30 $blogPost = new BlogPost();
$blogPost->title = 'Test blog post';
$blogPost->body = 'This is the content of the test blog post';
$tag = Doctrine::getTable('Tag')->findOneByName('doctrine');
$blogPost->Tags[] = $tag;
```

Another method would be to override the Tag name mutator by creating a function named `setName()` in the generated Tag class located in `lib/model/doctrine/Tag.class.php`.

```
Listing 6-31 class Tag extends BaseTag
{
    public function setName($name)
    {
        $tag = Doctrine::getTable('Tag')->findOneByName($name);
        if ($tag) {
            $this->assignIdentifier($tag->identifier());
        } else {
            $this->_set('name', $name);
        }
    }
}
```

Now the code becomes even simpler to ensure duplicate tags are not inserted in to the database.

```
$blogPost = new BlogPost();  
$blogPost->title = 'Test blog post';  
$blogPost->body = 'This is the content of the test blog post';  
$blogPost->Tags[]->name = 'doctrine';
```

*Listing
6-32*

Deleting Data

There are two options for deleting data. Retrieve the object first and call the `Doctrine_Record::delete()` method or issue a single DQL delete query.

```
$user = Doctrine::getTable('User')->find(1);  
$user->delete();
```

*Listing
6-33*

Issue single DQL delete query. This is more efficient than the above because it only uses one query. The above example has to retrieve the object and then delete it.

```
$deleted = Doctrine_Query::create()  
->delete()  
->from('User u')  
->where('u.id = ?', 1)  
->execute();
```

*Listing
6-34*



More can be read about working with data in the Doctrine Manual here⁹.

9. http://www.doctrine-project.org/documentation/manual/1_0/en/working-with-models

Chapter 7

Migrations

A common problem in web development is how to manage changes to your database as your models evolve and schema changes. The migrations support in Doctrine provides an efficient solution to this problem.

`SfDoctrinePlugin` implements some additional tasks for generating migration classes both from existing databases and models. You can also use Doctrine to generate your blank skeleton migration classes.

Available Migration Tasks

Listing 7-1

```

:generate-migration          Generate migration class
(doctrine-generate-migration)
:generate-migrations-db     Generate migration classes from existing
database connections (doctrine-generate-migrations-db,
doctrine-gen-migrations-from-db)
:generate-migrations-models Generate migration classes from an existing
set of models (doctrine-generate-migrations-models,
doctrine-gen-migrations-from-models)
:migrate                    Migrates database to current/specified
version (doctrine-migrate)

```

The examples in this chapter assume you are working with the following schema and data fixtures. We'll use this as the base of all the examples documented here.

Starting Schema and Data Fixtures

```
project/config/doctrine/schema.yml
```

Listing 7-2

```

BlogPost:
  actAs:
    I18n:
      fields: [title, body]
    Timestampable:
  columns:
    author: string(255)
    title: string(255)
    body: clob
  relations:
    Tags:
      class: Tag

```

```

        refClass: BlogPostTag
        foreignAlias: BlogPosts
Tag:
  columns:
    name: string(255)
BlogPostTag:
  columns:
    blog_post_id:
      type: integer
      primary: true
    tag_id:
      type: integer
      primary: true

project/data/fixtures.yml

BlogPost:
  BlogPost_1:
    author: Jonathan H. Wage
    Translation:
      en:
        title: Test Blog Post
        body: This is the body of the test blog post
    Tags: [php, orm]
Tag:
  php:
    name: PHP
  orm:
    name: ORM

```

*Listing
7-3*

Generating Migrations

Doctrine offers the ability to generate sets of migration classes for existing databases or existing models as well as generating blank migration classes for you to fill in with the code to make your schema changes.

From Database

If you have an existing database you can build a set of migration classes that will re-create your database by running the following command.

```
$ ./symfony doctrine:generate-migrations-db
```

*Listing
7-4*

From Models

If you have an existing set of models you can build a set of migration classes that will create your database by running the following command.

```
$ ./symfony doctrine:generate-migrations-models
```

*Listing
7-5*

Now that you have a set of migration classes you can reset your database and run the migrate command to re-create the database.

```
$ ./symfony doctrine:drop-db
$ ./symfony doctrine:build-db
```

*Listing
7-6*

```
$ ./symfony doctrine:migrate
$ ./symfony doctrine:data-load
```

Skeleton

Now that your database is created and migrated to the latest version you can generate new migration class skeletons to migrate your schema as your model evolves. Imagine you have a new column that needs to be added to the BlogPost model named excerpt. Below is the updated BlogPost schema which includes the new column.

Listing 7-7

```
BlogPost:
  actAs:
    I18n:
      fields: [title, body]
    Timestampable:
  columns:
    author: string(255)
    title: string(255)
    body: clob
    excerpt: string(255)
  relations:
    Tags:
      class: Tag
      refClass: BlogPostTag
      foreignAlias: BlogPosts
```

Now we need to generate the blank migration skeleton for adding the excerpt column to the database. Run the following command to generate the migration class.

Listing 7-8

```
$ ./symfony doctrine:generate-migration AddBlogPostExcerptColumn
```

Now in `project/lib/migration/doctrine` you will see a new file named `006_add_blog_post_excerpt_column.class.php` with the following code in it.

Listing 7-9

```
/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
class AddBlogPostExcerptColumn extends Doctrine_Migration
{
    public function up()
    {

    }

    public function down()
    {

    }
}
```

Each migration consists of an `up()` method and a `down()` method. Inside the `up()` is where you can add columns, create tables, etc. and the `down()` simply negates anything done in the `up()`. Each class essentially represents a version of your database and the `up()` and `down()` methods allow you to walk backwards and forwards between versions of your database.

Now lets write the code for our new migration class to add the excerpt column.

```
/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
class AddBlogPostExcerptColumn extends Doctrine_Migration
{
    public function up()
    {
        $this->addColumn('blog_post', 'excerpt', 'string', array('length' =>
'255'));
    }

    public function down()
    {
        $this->removeColumn('blog_post', 'excerpt');
    }
}
```

*Listing
7-10*

Now you can run the following command and it will upgrade your database to the latest version using the migration class we just wrote and the excerpt column will be added to the database.

```
$ ./symfony doctrine:migrate
```

*Listing
7-11*



More can be read about migrations in the Doctrine Manual here¹⁰.

10. <http://www.doctrine-project.org/documentation/manual/1.0/en/migrations>

