



Doctrine 1.1

License: Creative Commons Attribution-Share Alike 3.0 Unported License

Version: cookbook-1.1-pt_BR-2010-04-19

Table of Contents

My First Project.....	3
Introduction	3
Download	3
Package Contents	3
Running the CLI.....	4
Defining Schema.....	4
Test Data Fixtures	5
Building Everything	5
Running Tests	6
User CRUD	8
symfony 1.1 and Doctrine	10
Setup.....	10
Setup Database.....	11
Setup Schema	11
Build Database.....	12
Admin Generators.....	13
Helpful Links.....	14
symfony 1.1 and Doctrine Migrations.....	16
Setting up your database.....	16
Define your schema	16
Build Database.....	17
Setup Migration.....	18
Run Migration.....	21
Code Igniter and Doctrine	23
Download Doctrine	23
Setup Doctrine.....	23
Setup Command Line Interface	24
Start Using Doctrine.....	26
Plug and Play Schema Information With Templates	29
Taking Advantage of Column Aggregation Inheritance	31

Chapter 1

My First Project

Introduction

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

Download

To get started, first download the latest Doctrine sandbox package: <http://www.phpdoctrine.org/download>¹. 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.

Package Contents

The files/directory structure should look like the following

```
$ cd Doctrine-0.10.1-Sandbox
$ ls
config.php  doctrine  index.php  migrations  schema
data        doctrine.php  lib        models
```

*Listing
1-1*

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

1. <http://www.phpdoctrine.org/download>

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

Running the CLI

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

Listing 1-2 `$./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
```

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 *.yaml files in the schemas folder.

Listing 1-3 `---`
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

```

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 *.yaml files in the data/fixtures folder.

```

---
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

```

Listing
1-4

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.

```

$ ./doctrine build-all-reload
build-all-reload - Are you sure you wish to drop your databases? (y/n)
y

```

Listing
1-5

```

build-all-reload - Successfully dropped database for connection "sandbox"
at path "/Users/jwage/Sites/doctrine/branches/0.10/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/0.10/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.

Running Tests

Listing 1-6

```

$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 1-7

```

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.

```

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

```

Listing
1-8

```
dql -      name: Lead
dql -      -
dql -      id: 3
dql -      name: Documentation
```

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
1-9

```
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> &nbsp; <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

symfony 1.1 and Doctrine

So, you want to give Doctrine a try with symfony 1.1 eh? First we will need to setup a new symfony 1.1 project and install the sfDoctrinePlugin for 1.1. Execute the following commands below and continue reading:

Setup

Listing 2-1

```
$ mkdir symfony1.1Doctrine
$ cd symfony1.1Doctrine
$ /path/to/symfony generate:project symfony1.1Doctrine
$ svn co http://svn.symfony-project.com/plugins/sfDoctrinePlugin/trunk
  plugins/sfDoctrinePlugin
$ php symfony cc
```

Now, type the following command to list all the new commands that `sfDoctrinePlugin` provides. You will notice that it gives you all the same commands as `sfPropelPlugin` and lots more!

Listing 2-2

```
$ php symfony list doctrine
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-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.xml 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-crud	Generates a Doctrine CRUD module
(doctrine-generate-crud)	
: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)
:init-admin	Initializes a Doctrine admin module
(doctrine-init-admin)	
: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)	

First, `sfDoctrinePlugin` currently requires that at least one application be setup, so lets just instantiate a `frontend` application now.

```
$ php symfony generate:app frontend
```

*Listing
2-3*

Setup Database

Now lets setup our database configuration in `config/databases.yml`. Open the file in your favorite editor and place the YAML below inside. For this test we are simply using a SQLite database. Doctrine is able to create the SQLite database at the `config/doctrine.db` path for you which we will do once we setup our schema and some data fixtures.

```
---
all:
  doctrine:
    class:      sfDoctrineDatabase
    param:
      dsn:      sqlite
```

*Listing
2-4*

Setup Schema

Now that we have our database configured, lets define our YAML schema files in `config/doctrine/schema.yml`. In this example we are setting up a simple `BlogPost` model which `hasMany` `Tags`.

```
---
BlogPost:
  actAs:
    Sluggable:
      fields: [title]
```

*Listing
2-5*

```

    Timestampable:
    columns:
        title: string(255)
        body: clob
        author: string(255)
    relations:
        Tags:
            class: Tag
            refClass: BlogPostTag
            foreignAlias: BlogPosts

BlogPostTag:
    columns:
        blog_post_id:
            type: integer
            primary: true
        tag_id:
            type: integer
            primary: true

Tag:
    actAs: [Timestampable]
    columns:
        name: string(255)

```

Now that we have our Doctrine schema defined, lets create some test data fixtures in `data/fixtures/data.yml`. Open the file in your favorite editor and paste the below YAML in to the file.

Listing 2-6

```

---
BlogPost:
    BlogPost_1:
        title: symfony + Doctrine
        body: symfony and Doctrine are great!
        author: Jonathan H. Wage
        Tags: [symfony, doctrine, php]

Tag:
    symfony:
        name: symfony
    doctrine:
        name: doctrine
    php:
        name: php

```

Build Database

Ok, now for the fun stuff. We have our schema, and we have some data fixtures, so lets run one single Doctrine command and create your database, generate your models, create tables and load the data fixtures.

Listing 2-7

```

$ php symfony doctrine-build-all-reload frontend
>> doctrine Are you sure you wish to drop your databases? (y/n)
y
>> doctrine Successfully dropped database f...1.1Doctrine/config/
doctrine.db"

```

```
>> doctrine Successfully created database f...1.1Doctrine/config/
doctrine.db"
>> doctrine Generated models successfully
>> doctrine Created tables successfully
>> doctrine Data was successfully loaded
```

Now your `doctrine.db` SQLite database is created, all the tables for your schema were created, and the data fixtures were populated in to the tables. Now lets do a little playing around with the data to see how we can use the Doctrine Query Language to retrieve data.

```
$ php symfony doctrine:dql frontend "FROM BlogPost p, p.Tags t"
>> doctrine executing: "FROM BlogPost p, p.Tags t" ()
>> doctrine -
>> doctrine id: 1
>> doctrine title: symfony + Doctrine
>> doctrine body: symfony and Doctrine are great!
>> doctrine author: Jonathan H. Wage
>> doctrine slug: symfony-doctrine
>> doctrine created_at: 2008-06-16 12:28:57
>> doctrine updated_at: 2008-06-16 12:28:57
>> doctrine Tags:
>> doctrine -
>> doctrine id: 1
>> doctrine name: symfony
>> doctrine created_at: 2008-06-16 12:28:57
>> doctrine updated_at: 2008-06-16 12:28:57
>> doctrine -
>> doctrine id: 2
>> doctrine name: doctrine
>> doctrine created_at: 2008-06-16 12:28:57
>> doctrine updated_at: 2008-06-16 12:28:57
>> doctrine -
>> doctrine id: 3
>> doctrine name: php
>> doctrine created_at: 2008-06-16 12:28:57
>> doctrine updated_at: 2008-06-16 12:28:57
```

*Listing
2-8*

Now, lets do a little explaining of the data that was returned. As you can see the models have a `created_at`, `updated_at` and `slug` column which were not defined in the schema files. These columns are added by the behaviors attached to the schema information under the `actAs` setting. The ``created_at`` and ``updated_at`` column are automatically set ``onInsert`` and ``onUpdate``, and the `slug` column is a url friendly string that is created from the value of the `name` column. Doctrine has a few behaviors that are included in core such as ``Sluggable`` and ``Timestampable``, but the behavior system is built to allow anyone to easily write behaviors for their models to re-use over and over.

Admin Generators

Now we have our data model all setup and populated with some test fixtures so lets generate an admin generator to manage the blog posts and tags.

```
$ php symfony doctrine:init-admin frontend blog_posts BlogPost
$ php symfony doctrine:init-admin frontend tags Tag
```

*Listing
2-9*

Now go open up your web browser and check out the ``frontend`` application and the ``blog_posts`` and ``tags`` modules. It should be located at a url like the following:

Listing 2-10 `http://localhost/symfony1.1Doctrine/web/frontend_dev.php/blog_posts`
`http://localhost/symfony1.1Doctrine/web/frontend_dev.php/tags`

Now, with a little configuration of the blog post admin generator, we can control the associated blog post tags by checking checkboxes when editing a blog post. Open ``apps/frontend/modules/blog_posts/config/generator.yml`` and replace the contents with the YAML from below.

Listing 2-11 `---`
`generator:`
`class: sfDoctrineAdminGenerator`
`param:`
`model_class: BlogPost`
`theme: default`
`list:`
`display: [=title, author]`
`object_actions:`
`_edit: -`
`_delete: -`
`edit:`
`display: [author, title, body, Tags]`
`fields:`
`author:`
`type: input_tag`
`title:`
`type: input_tag`
`body:`
`type: textarea_tag`
`params: size=50x10`
`Tags:`
`type: doctrine_admin_check_list`
`params: through_class=BlogPostTag`

Now refresh the blog post list and you will see it is cleaned up a little bit. Edit a blog post by clicking the edit icon or the title and you can see below you can check the tags associated to the blog post.

All of the features you get in Propel work 99% the same way with Doctrine, so it should be fairly easy to get the hang of if you are coming from propel. `sfDoctrinePlugin` implements all the same functionality as `sfPropelPlugin` as well as several additional features which `sfPropelPlugin` is not capable of. Below you can find some more information on the major features that Doctrine supports:

Helpful Links

- Behaviors - http://www.phpdoctrine.org/documentation/manual/0_11?chapter=plugins² - Easily create reusable behaviors for your Doctrine models.
- Migrations - http://www.phpdoctrine.org/documentation/manual/0_11?chapter=migration³ - Deploy database schema changes to your production environment through a programmatic interface.

2. http://www.phpdoctrine.org/documentation/manual/0_11?chapter=plugins

3. http://www.phpdoctrine.org/documentation/manual/0_11?chapter=migration

- Doctrine Query Language - http://www.phpdoctrine.org/documentation/manual/0_11?chapter=dql-doctrine-query-language⁴ - Build your database queries through a fluent OO interface
- Validators - http://www.phpdoctrine.org/documentation/manual/0_11?chapter=basic-schema-mapping#constraints-and-validators⁵ - Turn on column validators for both database and code level validation.
- Hierarchical Data http://www.phpdoctrine.org/documentation/manual/0_11?chapter=hierarchical-data⁶ - Turn your models in to nested sets easily with the flip of a switch.
- Caching http://www.phpdoctrine.org/documentation/manual/0_11?chapter=caching⁷ - Tune performance by caching your DQL query parsing and the result sets of queries.

If this short tutorial sparked your interest in Doctrine you can check out some other Doctrine resources below to learn more about Doctrine:

- Full User Manual - http://www.phpdoctrine.org/documentation/manual/0_11?one-page⁸
- API Documentation - http://www.phpdoctrine.org/documentation/api/0_11⁹
- Cheatsheet - <http://www.phpdoctrine.org/Doctrine-Cheat-Sheet.pdf>¹⁰
- Blog - <http://www.phpdoctrine.org/blog>¹¹
- Community - <http://www.phpdoctrine.org/community>¹²
- Frequently Asked Questions - <http://www.phpdoctrine.org/faq>¹³
- Download - <http://www.phpdoctrine.org/download>¹⁴

4. http://www.phpdoctrine.org/documentation/manual/0_11?chapter=dql-doctrine-query-language

5. http://www.phpdoctrine.org/documentation/manual/0_11?chapter=basic-schema-mapping#constraints-and-validators

6. http://www.phpdoctrine.org/documentation/manual/0_11?chapter=hierarchical-data

7. http://www.phpdoctrine.org/documentation/manual/0_11?chapter=caching

8. http://www.phpdoctrine.org/documentation/manual/0_11?one-page

9. http://www.phpdoctrine.org/documentation/api/0_11

10. <http://www.phpdoctrine.org/Doctrine-Cheat-Sheet.pdf>

11. <http://www.phpdoctrine.org/blog>

12. <http://www.phpdoctrine.org/community>

13. <http://www.phpdoctrine.org/faq>

14. <http://www.phpdoctrine.org/download>

Chapter 3

symfony 1.1 and Doctrine Migrations

The PHP Doctrine ORM offers a fully featured database migration utility that makes it easy to upgrade your databases for both schema and data changes without having to manually write or keep up with SQL statements.

Database migrations essentially allow you to have multiple versions of your schema. A single Doctrine migration class represents one version of the schema. Each migration class must have an `up()` and a `down()` method defined and the `down()` must negate everything done in the `up()` method. Below I will show you an example of how to use Doctrine to control your database.

Listing 3-1 This tutorial is written for symfony 1.1 but the same functionality exists for the 1.0 version of sfDoctrinePlugin but in the 1.0 style task system.

Setting up your database

First thing we need to do is define your database and create it. Edit `config/databases.yml` and setup your mysql database. Copy and paste the yaml below in to the file.

Listing 3-2

```
---
all:
  doctrine:
    class:      sfDoctrineDatabase
    param:
      dsn:      mysql
```

Define your schema

In this example we are going to use a traditional Blog model. Open `config/doctrine/schema.yml` and copy and paste the yaml contents from below in to the file.

Listing 3-3

```
---
BlogPost:
  actAs:
```



```

    Sluggable:
        fields: [title]
    columns:
        title: string(255)
        body: clob
        author: string(255)
    relations:
        Tags:
            class: Tag
            refClass: BlogPostTag
            foreignAlias: BlogPosts

BlogPostTag:
    columns:
        blog_post_id:
            type: integer
            primary: true
        tag_id:
            type: integer
            primary: true

Tag:
    columns:
        name: string(255)

```

Place the below data fixtures in to data/fixtures/data.yml

```

---
BlogPost:
    BlogPost_1:
        slug: symfony-doctrine
        author: Jonathan H. Wage
        title: symfony + Doctrine
        body: symfony and Doctrine are great!
        Tags: [symfony, doctrine, php]

Tag:
    symfony:
        name: symfony
    doctrine:
        name: doctrine
    php:
        name: php

```

*Listing
3-4*

Build Database

Now with one simple command Doctrine is able to create the database, the tables and load the data fixtures for you. Doctrine works with any [PDO](<http://www.php.net/pdo>¹⁵) driver and is able to drop and create databases for any of them.

```

$ ./symfony doctrine-build-all-reload frontend
>> doctrine Are you sure you wish to drop your databases? (y/n)
y

```

*Listing
3-5*

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

```
>> doctrine Successfully dropped database f...1.1Doctrine/config/
doctrine.db"
>> doctrine Successfully created database f...1.1Doctrine/config/
doctrine.db"
>> doctrine Generated models successfully
>> doctrine Created tables successfully
>> doctrine Data was successfully loaded
```

Now your database, models and tables are created for you so easily. Lets run a simple DQL query to see the current data that is in the database so we can compare it to the data after the migration has been performed.

Listing 3-6

```
$ ./symfony doctrine-dql frontend "FROM BlogPost p, p.Tags t"
>> doctrine executing: "FROM BlogPost p, p.Tags t" ()
>> doctrine -
>> doctrine id: 1
>> doctrine title: symfony + Doctrine
>> doctrine body: symfony and Doctrine are great!
>> doctrine author: Jonathan H. Wage
>> doctrine slug: symfony-doctrine
>> doctrine Tags:
>> doctrine -
>> doctrine id: 1
>> doctrine name: symfony
>> doctrine -
>> doctrine id: 2
>> doctrine name: doctrine
>> doctrine -
>> doctrine id: 3
>> doctrine name: php
```

Setup Migration

Now what if a few months later you want to change the schema to split out the BlogPost.author column to an Author model that is related to BlogPost.author_id. First lets add the new model to your config/doctrine/schema.yml. Replace your schema yaml with the schema information from below.

Listing 3-7

```
---
BlogPost:
  actAs:
    Sluggable:
      fields: [title]
  columns:
    title: string(255)
    body: clob
    author: string(255)
    author_id: integer
  relations:
    Author:
      foreignAlias: BlogPosts
    Tags:
      class: Tag
      refClass: BlogPostTag
      foreignAlias: BlogPosts
```

```

BlogPostTag:
  columns:
    blog_post_id:
      type: integer
      primary: true
    tag_id:
      type: integer
      primary: true

Tag:
  columns:
    name: string(255)

Author:
  columns:
    name: string(255)

```

Rebuild your models now with the following command.

```

$ ./symfony doctrine-build-model
>> doctrine  Generated models successfully

```

*Listing
3-8*

As you see we have added a new Author model, and changed the author column to author_id and integer for a foreign key to the Author model. Now let's write a new migration class to upgrade the existing database without losing any data. Run the following commands to create skeleton migration classes in lib/migration/doctrine. You will see a file generated named 001_add_author.class.php and 002_migrate_author.class.php. Inside them are blank up() and down() method for you to code your migrations for the schema changes above.

```

$ ./symfony doctrine:generate-migration frontend AddAuthor
>> doctrine  Generated migration class: AddA...Doctrine/lib/migration/
doctrine
$ ./symfony doctrine:generate-migration frontend MigrateAuthor
>> doctrine  Generated migration class: Migr...Doctrine/lib/migration/
doctrine

```

*Listing
3-9*

Now we have 2 blank migration skeletons to write our migration code in. Below I have provided the code to migrate the author column to an Author model and automatically relate blog posts to the newly created author id.

```

// 001_add_author.class.php
/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
class AddAuthor extends Doctrine_Migration
{
    public function up()
    {
        // Create new author table
        $columns = array('id' => array('type' => 'integer',
                                     'length' => 4,
                                     'autoincrement' => true),
                        'name' => array('type' => 'string',
                                     'length' => 255));
    }
}

```

*Listing
3-10*

```

        $this->createTable('author', $columns, array('primary' =>
array('id')));

        // Add author_id to the blog_post table
        $this->addColumn('blog_post', 'author_id', 'integer', array('length'
=> 4));
    }

    public function down()
    {
        // Remove author table
        $this->dropTable('author');

        // Remove author_id column from blog_post table
        $this->removeColumn('blog_post', 'author_id');
    }
}

// 002_migrate_author.class.php
/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
class MigrateAuthor extends Doctrine_Migration
{
    public function preUp()
    {
        $q = Doctrine_Query::create()
            ->select('p.id, p.author')
            ->from('BlogPost p');

        $blogPosts = $q->execute();
        foreach ($blogPosts as $blogPost)
        {
            $author =
Doctrine::getTable('Author')->findOneByName($blogPost->author);
            if ( ! ($author instanceof Author && $author->exists()))
            {
                $author = new Author();
                $author->name = $blogPost->author;
                $author->save();
            }
            $blogPost->author_id = $author->id;
            $blogPost->save();
        }
    }

    public function up()
    {
        $this->removeColumn('blog_post', 'author');
    }

    public function down()
    {
        $this->addColumn('blog_post', 'author', 'string', array('length' =>
255));
    }
}

```

Now run the following command and Doctrine will automatically perform the migration process and update the database.

Run Migration

```
$ ./symfony doctrine-migrate frontend
>> doctrine migrated successfully to version #2
```

*Listing
3-11*

Now the database is updated with the new schema information and data migrated. Give it a check and you will see that we have a new author table, the blog_post.author column is gone and we have a new blog_post.author_id column that is set to the appropriate author id value.

The #2 migration removed the author column from the blog_post table, but we left it in the model definition so that while it still existed, before the #2 migration began we copied the contents of the author column to the author table and related the blog_post to the author id. You can now remove the author: string(255) column definition from the config/doctrine/schema.yml and rebuild the models. Here is the new BlogPost model definition.

```
---
BlogPost:
  actAs:
    Sluggable:
      fields: [title]
  columns:
    title: string(255)
    body: clob
    author_id: integer
  relations:
    Author:
      foreignAlias: BlogPosts
    Tags:
      class: Tag
      refClass: BlogPostTag
      foreignAlias: BlogPosts
```

*Listing
3-12*

Re-build the models now since we removed the author column from the model definition and the table in the database.

```
$ ./symfony doctrine-build-model
>> doctrine Generated models successfully
```

*Listing
3-13*

Now lets run a DQL query from the command line to see the final product.

```
$ ./symfony doctrine:dql frontend "FROM BlogPost p, p.Tags, p.Author a"
>> doctrine executing: "FROM BlogPost p, p.Tags, p.Author a" ()
>> doctrine -
>> doctrine id: 1
>> doctrine title: symfony + Doctrine
>> doctrine body: symfony and Doctrine are great!
>> doctrine author_id: 1
>> doctrine slug: symfony-doctrine
>> doctrine Tags:
```

*Listing
3-14*

```
>> doctrine      -
>> doctrine      id: 1
>> doctrine      name: symfony
>> doctrine      -
>> doctrine      id: 2
>> doctrine      name: doctrine
>> doctrine      -
>> doctrine      id: 3
>> doctrine      name: php
>> doctrine      Author:
>> doctrine      id: 1
>> doctrine      name: Jonathan H. Wage
```

If you compare the data returned here, to the data that was returned in the beginning of this tutorial you will see that the author column was removed and migrated to an Author model.

Chapter 4

Code Igniter and Doctrine

This tutorial will get you started using Doctrine with Code Igniter

Download Doctrine

First we must get the source of Doctrine from svn and place it in the system/database folder.

```
$ cd system/database
$ svn co http://svn.phpdoctrine.org/branches/0.11/lib doctrine
$ cd ..

// If you use svn in your project you can set Doctrine
// as an external so you receive bug fixes automatically from svn
$ svn propedit svn:externals database

// In your favorite editor add the following line
// doctrine http://svn.phpdoctrine.org/branches/0.11/lib
```

*Listing
4-1*

Setup Doctrine

Now we must setup the configuration for Doctrine and load it in system/application/config/database.php

```
$ vi application/config/database.php
```

*Listing
4-2*

The code below needs to be added under this line of code

```
$db['default']['cachedir'] = "";
```

*Listing
4-3*

Add this code

```
// Create dsn from the info above
$db['default']['dsn'] = $db['default']['dbdriver'] .
    ':' . $db['default']['username'] .
    ':' . $db['default']['password'] .
    '@' . $db['default']['hostname'] .
    '/' . $db['default']['database'];

// Require Doctrine.php
require_once(realpath(dirname(__FILE__) . '/../..') . DIRECTORY_SEPARATOR
    . 'database/doctrine/Doctrine.php');
```

*Listing
4-4*

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

// Load the Doctrine connection
Doctrine_Manager::connection($db['default']['dsn'],
$db['default']['database']);

// Set the model loading to conservative/lazy loading
Doctrine_Manager::getInstance()->setAttribute('model_loading',
'conservative');

// Load the models for the autoloader
Doctrine::loadModels(realpath(dirname(__FILE__) . '/../') .
DIRECTORY_SEPARATOR . 'models');
```

Now we must make sure system/application/config/database.php is included in your front controller. Open your front controller in your favorite text editor.

Listing 4-5

```
$ cd ..
$ vi index.php
```

Change the last 2 lines of code of index.php with the following

Listing 4-6

```
require_once APPPATH.'config/database.php';
require_once BASEPATH.'codeigniter/CodeIgniter'.EXT;
```

Setup Command Line Interface

Create the following file: system/application/doctrine and chmod the file so it can be executed. Place the code below in to the doctrine file.

Listing 4-7

```
$ vi system/application/doctrine
```

Place this code in system/application/doctrine

Listing 4-8

```
#!/usr/bin/env php
define('BASEPATH','.'); // mockup that this app was executed from ci ;)
chdir(dirname(__FILE__));
include('doctrine.php');
```

Now create the following file: system/application/doctrine.php. Place the code below in to the doctrine.php file.

Listing 4-9

```
require_once('config/database.php');

// Configure Doctrine Cli
// Normally these are arguments to the cli tasks but if they are set here
the arguments will be auto-filled
$config = array('data_fixtures_path' => dirname(__FILE__) .
DIRECTORY_SEPARATOR . '/fixtures',
               'models_path'       => dirname(__FILE__) .
DIRECTORY_SEPARATOR . '/models',
               'migrations_path'   => dirname(__FILE__) .
DIRECTORY_SEPARATOR . '/migrations',
               'sql_path'          => dirname(__FILE__) .
```



```

DIRECTORY_SEPARATOR . '/sql',
                    'yaml_schema_path'    =>  dirname(__FILE__) .
DIRECTORY_SEPARATOR . '/schema');

$cli = new Doctrine_Cli($config);
$cli->run($_SERVER['argv']);

```

Now we must create all the directories for Doctrine to use

```

// Create directory for your yaml data fixtures files
$ mkdir system/application/fixtures

// Create directory for your migration classes
$ mkdir system/application/migrations

// Create directory for your yaml schema files
$ mkdir system/application/schema

// Create directory to generate your sql to create the database in
$ mkdir system/application/sql

```

*Listing
4-10*

Now you have a command line interface ready to go. If you execute the doctrine shell script with no argument you will get a list of available commands

```

$ cd system/application
$ ./doctrine
Doctrine Command Line Interface

```

*Listing
4-11*

```

./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
$

```

On Microsoft Windows, call the script via the PHP binary (because the script won't invoke it automatically):

```
php.exe doctrine
```

*Listing
4-12*

Start Using Doctrine

It is simple to start using Doctrine now. First we must create a yaml schema file.
(save it at schema with filename like : user.yml)

Listing
4-13

```

---
User:
  columns:
    id:
      primary: true
      autoincrement: true
      type: integer(4)
    username: string(255)
    password: string(255)
  relations:
    Groups:
      class: Group                # Class name. Optional if alias is the
class name                      # Local
      local: user_id              # Foreign
      foreign: group_id           # xRefClass for relating Users to Groups
      refClass: UserGroup         # Opposite relationship alias. Group
      foreignAlias: Users
hasMany Users

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

UserGroup:
  columns:
    user_id:
      type: integer(4)
      primary: true
    group_id:
      type: integer(4)
      primary: true
  relations:
    User:
      local: user_id              # Local key
      foreign: id                 # Foreign key
      onDelete: CASCADE          # Database constraint
    Group:
      local: group_id
      foreign: id
      onDelete: CASCADE

```

Now if you run the following command it will generate your models in system/application/models

Listing
4-14

```

$ ./doctrine generate-models-yaml
generate-models-yaml - Generated models successfully from YAML schema

```

Now check the file `system/application/models/generated/BaseUser.php`. You will see a compclass definition like below.

```
/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
abstract class BaseUser extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->setTableName('user');
        $this->hasColumn('id', 'integer', 4, array('primary' => true,
'autoincrement' => true));
        $this->hasColumn('username', 'string', 255);
        $this->hasColumn('password', 'string', 255);
    }

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

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

// Add custom methods to system/application/models/User.php

/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
class User extends BaseUser
{
    public function setPassword($password)
    {
        $this->password = md5($password);
    }
}

/**
 * This class has been auto-generated by the Doctrine ORM Framework
 */
class UserTable extends Doctrine_Table
{
    public function retrieveAll()
    {
        $query = new Doctrine_Query();
        $query->from('User u');
        $query->orderby('u.username ASC');

        return $query->execute();
    }
}
```

Listing
4-15

Now we can create some sample data to load in to our application (this step requires you have a valid database configured and ready to go. The build-all-reload task will drop and recreate the database, create tables, and load data fixtures)

Create a file in system/application/fixtures/users.yml

Listing 4-16 `$ vi fixtures/users.yml`

Add the following yaml to the file

Listing 4-17 `---
User:
 jwage:
 username: jwage
 password: test`

Now run the build-all-reload task to drop db, build models, recreate

Listing 4-18 `$./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 named: "jwage_codeigniter"
build-all-reload - Generated models successfully from YAML schema
build-all-reload - Successfully created database named: "jwage_codeigniter"
build-all-reload - Created tables successfully
build-all-reload - Data was successfully loaded`

Now we are ready to use Doctrine in our actual actions. Lets open our system/application/views/welcome_message.php and somewhere add the following code somewhere.

Listing 4-19 `$user = new User();
$user->username = 'zYne-';
$user->setPassword('password');
$user->save();

$userTable = Doctrine::getTable('User');
$user = $userTable->findOneByUsername('zYne-');

echo $user->username; // prints 'zYne-'

$users = $userTable->retrieveAll();

echo $users->count(); // echo '2'
foreach ($users as $user)
{
 echo $user->username;
}`

Chapter 5

Plug and Play Schema Information With Templates

Doctrine templates essentially allow you to extract schema information so that it can be plugged in to multiple Doctrine classes without having to duplicate any code. Below we will show some examples of what a template could be used for and how it can make your schema easier to maintain.

Let's get started. Imagine a project where you have multiple records which must have address attributes. There are two basic approaches to solving this problem. One is to have a single table to store all addresses and each record will store a foreign key to the address record it owns. This is the "normalized" way of solving the problem. The "de-normalized" way would be to store the address attributes with each record. In this example a template will extract the attributes of an address and allow you to plug them in to as many Doctrine classes as you like.

First we must define the template so that we can use it in our Doctrine classes.

```
class Doctrine_Template_Address extends Doctrine_Template
{
    public function setTableDefinition()
    {
        $this->hasColumn('address1', 'string', 255);
        $this->hasColumn('address2', 'string', 255);
        $this->hasColumn('address3', 'string', 255);
        $this->hasColumn('city', 'string', 255);
        $this->hasColumn('state', 'string', 2);
        $this->hasColumn('zipcode', 'string', 15);
    }
}
```

*Listing
5-1*

Now that we have our template defined, let's define some basic models that need to have address attributes added to them. Let's start first with a User.

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

    public function setUp()
    {
```

*Listing
5-2*

```

        $this->actAs('Address');
    }
}

```

Now we also have a Company model which also must contain an address.

Listing 5-3

```

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

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

```

Now let's generate the SQL to create the tables for the User and Company model. You will see that the attributes from the template are automatically added to each table.

Listing 5-4

```

CREATE TABLE user (id BIGINT AUTO_INCREMENT,
username VARCHAR(255),
password VARCHAR(255),
address1 VARCHAR(255),
address2 VARCHAR(255),
address3 VARCHAR(255),
city VARCHAR(255),
state VARCHAR(2),
zipcode VARCHAR(15),
PRIMARY KEY(id)) ENGINE = INNODB

CREATE TABLE company (id BIGINT AUTO_INCREMENT,
name VARCHAR(255),
description LONGTEXT,
address1 VARCHAR(255),
address2 VARCHAR(255),
address3 VARCHAR(255),
city VARCHAR(255),
state VARCHAR(2),
zipcode VARCHAR(15),
PRIMARY KEY(id)) ENGINE = INNODB

```

That's it. Now you can maintain your Address schema information from one place and use the address functionality in as many places as you like.

Chapter 6

Taking Advantage of Column Aggregation Inheritance

First, let me give a brief explanation of what column aggregation inheritance is and how it works. With column aggregation inheritance all classes share the same table, and all columns must exist in the parent. Doctrine is able to know which class each row in the database belongs to by automatically setting a "type" column so that Doctrine can cast the correct class when hydrating data from the database. Even if you query the top level column aggregation class, the collection will return instances of the class that each row belongs to.

Now that you have a basic understand of column aggregation inheritance lets put it to use. In this example we will setup some models which will allow us to use one address table for storing all of our addresses across the entire application. Any record will be able to have multiple addresses, and all the information will be stored in one table. First lets define our Address

```
class Address extends Doctrine_Record
{
    public function setTableDefinition()
    {
        $this->hasColumn('address1', 'string', 255);
        $this->hasColumn('address2', 'string', 255);
        $this->hasColumn('address3', 'string', 255);
        $this->hasColumn('city', 'string', 255);
        $this->hasColumn('state', 'string', 2);
        $this->hasColumn('zipcode', 'string', 15);
        $this->hasColumn('type', 'string', 255);
        $this->hasColumn('record_id', 'integer');

        $this->option('export', 'tables');

        $this->setSubClasses(array('UserAddress' => array('type' =>
'UserAddress'),
                                'CompanyAddress' => array('type' =>
'CompanyAddress'))));
    }
}
```

*Listing
6-1*

Note the option set above to only export tables because we do not want to export any foreign key constraints since record_id is going to relate to many different records.

We are going to setup a User so it can have multiple addresses, so we will need to setup a UserAddress child class that User can relate to.

Listing 6-2

```
class UserAddress extends Address
{
    public function setUp()
    {
        $this->hasOne('User', array('local' => 'record_id',
                                   'foreign' => 'id'));
    }
}
```

Now let's define our User and link it to the UserAddress model so it can have multiple addresses.

Listing 6-3

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

    public function setUp()
    {
        $this->hasMany('UserAddress as Addresses', array('local' =>
'id',
                                                         'foreign' =>
'record_id'));
    }
}
```

Now say we have a Company record which also needs to have many addresses. First we need to setup the CompanyAddress child class

Listing 6-4

```
class CompanyAddress extends Address
{
    public function setUp()
    {
        $this->hasOne('Company', array('local' => 'record_id',
                                       'foreign' => 'id'));
    }
}
```

Now let's define our Company and link it to the CompanyAddress model so it can have multiple addresses.

Listing 6-5

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

    public function setUp()
    {
        $this->hasMany('CompanyAddress as Addresses', array('local' =>
'id',
                                                         'foreign' =>
'record_id'));
    }
}
```



```
}
}
```

Now both Users and Companies can have multiple addresses and the data is all stored in one address table.

Now lets create the tables and insert some records

```
Doctrine::createTablesFromArray(array('User', 'Company', 'Address'));
```

*Listing
6-6*

```
$user = new User();
$user->username = 'jwage';
$user->password = 'changeme';
$user->Addresses[0]->address1 = '123 Road Dr.';
$user->Addresses[0]->city = 'Nashville';
$user->Addresses[0]->state = 'TN';
$user->save();

$company = new Company();
$company->name = 'centre{source}';
$company->Addresses[0]->address1 = '123 Road Dr.';
$company->Addresses[0]->city = 'Nashville';
$company->Addresses[0]->state = 'TN';
$company->save();
```

Query for the user and its addresses

```
$users = Doctrine_Query::create()
    ->from('User u')
    ->leftJoin('u.Addresses a')
    ->execute();
```

*Listing
6-7*

```
echo $users[0]->username; // jwage
echo $users[0]->Addresses[0]->address1 = '123 Road Dr.';
echo get_class($users[0]->Addresses[0]); // UserAddress
```

Query for the company and its addresses

```
$companies = Doctrine_Query::create()
    ->from('Company c')
    ->leftJoin('c.Addresses a')
    ->execute();
```

*Listing
6-8*

```
echo $companies[0]->name; // centre{source}
echo $companies[0]->Addresses[0]->address1 = '123 Road Dr.';
echo get_class($companies[0]->Addresses[0]); // CompanyAddress
```

Now lets query the Addresses directly and you will notice each child record returned is hydrated as the appropriate child class that created the record initially.

```
$addresses = Doctrine_Query::create()
    ->from('Address a')
    ->execute();
echo get_class($addresses[0]); // UserAddress
echo get_class($addresses[1]); // CompanyAddress
```

*Listing
6-9*