Doctrine 2

Not the same Old PHP ORM

What is different in Doctrine 2?

New code, new concepts, different workflow

100% re-written codebase for PHP 5.3

Are you scared?

You shouldn't be! It is a very exciting thing for PHP and change is a good thing!

We learned lots building Doctrine 1 and we used that to help us build Doctrine 2

Let me tell you why!

Performance of Doctrine 1

To hydrate 5000 records
in Doctrine 1 it takes
roughly 4.3 seconds.

Performance of Doctrine 2

Under Doctrine 2, hydrating those same 5000 records only takes 1.4 seconds.

Performance of Doctrine 2

...and with 10000 records it still only takes about 3.5 seconds.

Twice the data and still faster than Doctrine 1

Performance of Doctrine 2

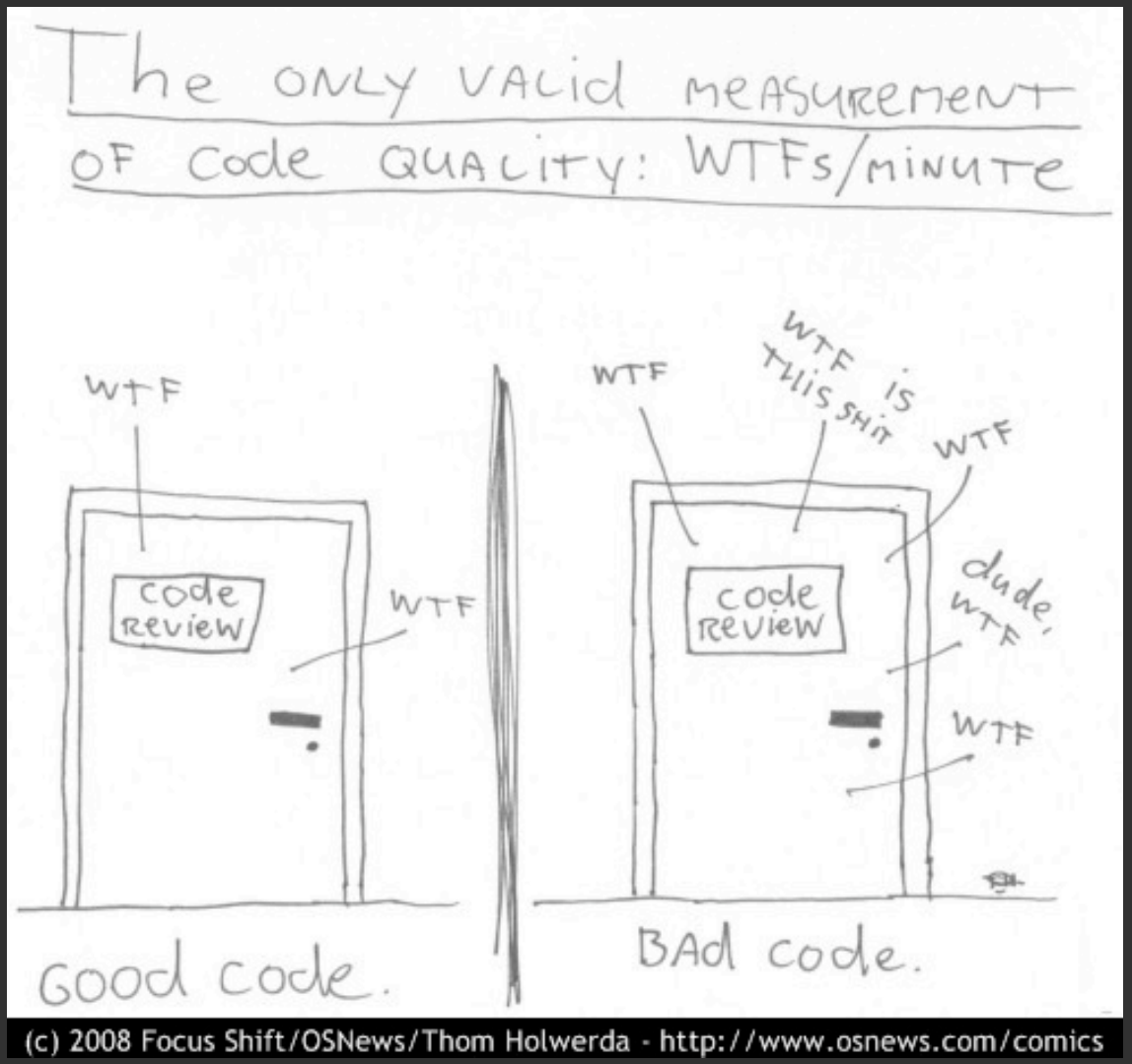
- More interesting than the numbers themselves is the percentage improvement over Doctrine 1

Why is it faster?

- PHP 5.3 gives us a huge performance improvement when using a heavily OO framework like Doctrine
- Better optimized hydration algorithm
- New query and result caching implementations
- All around more explicit and less magical code results in better and faster code.
- Killed the magical aspect of Doctrine 1

Why kill the magic?

- Eliminate the WTF? factor of Doctrine 1



The Doctrine 1 magical features are both a blessing and a curse

Blessing and a Curse

- Magic is great when it works
- The magic you love is also the cause of all the pain you've felt with Doctrine 1
- When it doesn't work it is hard to debug
- Edge cases are hard to fix
- Edge cases are hard to work around
- Edge cases, edge cases, edge cases
- Everything is okay until you try and go outside the box the magic provides
- ...magic is slow

How will we replace the magic?

This new thing called OOP :)

- Object Composition
- Inheritance
- Aggregation
- Containment
- Encapsulation
- ...etc

Will Doctrine 2 have behaviors?

Yes and No

The No

- We won't have any concept of "model behaviors"
- Behaviors were a made up concept for Doctrine 1 to work with its extremely intrusive architecture.
- It tries to do things that PHP does not allow and is the result of lots of problems in Doctrine 1

The Yes

- Everything you can do in Doctrine 1 you can do in Doctrine 2, just in a different way.
- “Behavior” like functionality will be bundled as extensions for Doctrine 2 and will just contain normal OO PHP code that wraps/ extends Doctrine code or is meant to be wrapped or extended by your entities.

What did we use to build Doctrine 2?

Doctrine 2 Tool Belt

- phpUnit 3.4.10 – Unit Testing
- Phing – Packaging and Distribution
- Symfony YAML Component
- Sismo – Continuous Integration
- Subversion – Source Control
- Jira – Issue Tracking and Management
- Trac – Subversion Timeline, Source Code Browser, Changeset Viewer

Doctrine 2 Architecture

- **Entities**
- Lightweight persistent domain object
- Regular PHP class
- Does not extend any base Doctrine class
- Cannot be final or contain final methods
- Any two entities in a hierarchy of classes must not have a mapped property with the same name
- Supports inheritance, polymorphic associations and polymorphic queries.
- Both abstract and concrete classes can be entities
- Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes

Doctrine 2 Architecture

- Your entities in Doctrine 2 don't require that you extend a base class like in Doctrine 1! No more imposing on your domain model!

```php
namespace Entities;

class User
{
    private $id;
    private $name;
    private $address;
}
```

Doctrine 2 Architecture

- **The EntityManager**
- Central access point to the ORM functionality provided by Doctrine 2. API is used to manage the persistence of your objects and to query for persistent objects.
- Employes transactional write behind strategy that delays the execution of SQL statements in order to execute them in the most efficient way
- Execute at end of transaction so that all write locks are quickly releases
- Internally an EntityManager uses a UnitOfWork to keep track of your objects

Unit Testing

- Tests are ran against multiple DBMS types. This is something that was not possible with the Doctrine 1 test suite.

- ...Sqlite
- ...MySQL
- ...Oracle
- ...PgSQL
- ...more to come

Unit Testing

- 859 Test Cases
- 2152 Assertions
- Tests run in a few seconds compared to 30-40 seconds for Doctrine 1
- Much more granular and explicit unit tests
- Easier to debug failed tests
- Continuously integrated by Sismo :)

Sismo

- No, Sismo is not available yet!!!!!!!!! :)
- Want it? Bug Fabien!

Database Abstraction Layer

- Separate standalone package and namespace (Doctrine\DBAL).
- Can be used standalone.
- Much improved over Doctrine 1 in regards to the API for database introspection and schema management.

Database Abstraction Layer

- Hopefully Doctrine 2 DBAL can be the defacto standard DBAL for PHP 5.3 in the future like MDB and MDB2 were in PEAR
- Maybe we can make this happen for PEAR2?

DBAL Data API

- prepare($sql) – Prepare a given sql statement and return the \Doctrine\DBAL \Driver\Statement instance.
- executeUpdate($sql, array $params) – Executes a prepared statement with the given sql and parameters and returns the affected rows count.
- execute($sql, array $params) – Creates a prepared statement for the given sql and passes the parameters to the execute method, then returning the statement.
- fetchAll($sql, array $params) – Execute the query and fetch all results into an array.
- fetchArray($sql, array $params) – Numeric index retrieval of first result row of the given query.
- fetchBoth($sql, array $params) – Both numeric and assoc column name retrieval of the first result row.
- fetchColumn($sql, array $params, $colnum) – Retrieve only the given column of the first result row.
- fetchRow($sql, array $params) – Retrieve assoc row of the first result row.
- select($sql, $limit, $offset) – Modify the given query with a limit clause.
- delete($tableName, array $identifier) – Delete all rows of a table matching the given identifier, where keys are column names.
- insert($tableName, array $data) – Insert a row into the given table name using the

DBAL Introspection API

- listDatabases()
- listFunctions()
- listSequences()
- listTableColumns($tableName)
- listTableConstraints($tableName)
- listTableDetails($tableName)
- listTableForeignKeys($tableName)
- listTableIndexes($tableName)
- listTables()

DBAL Schema Representation

```
$schema = new \Doctrine\DBAL\Schema\Schema();
$myTable = $schema->createTable("my_table");
$myTable->createColumn("id", "integer", array("unsigned" => true));
$myTable->createColumn("username", "string", array("length" => 32));
$myTable->setPrimaryKey(array("id"));
$myTable->addUniqueIndex(array("username"));
$schema->createSequence("my_table_seq");

$myForeign = $schema->createTable("my_foreign");
$myForeign->createColumn("id", "integer");
$myForeign->createColumn("user_id", "integer");
$myForeign->addForeignKeyConstraint($myTable, array("user_id"),
array("id"), array("onUpdate" => "CASCADE"));

$queries = $schema->toSql($myPlatform); // get queries to create this
schema.
$dropSchema = $schema->toDropSql($myPlatform); // get queries to
safely delete this schema.
```

Compare DBAL Schemas

```php
$comparator = new \Doctrine\DBAL\Schema\Comparator();
$schemaDiff = $comparator->compare($fromSchema, $toSchema);

// queries to get from one to another schema.
$queries = $schemaDiff->toSql($myPlatform);
$saveQueries = $schemaDiff->toSaveSql($myPlatform);
```

Schema Management

- Extracted from ORM to DBAL
- Schema comparisons replace the migrations diff tool of Doctrine 1

Doctrine 2 Annotations

```php
<?php

namespace Entities;

/**
 * @Entity @Table(name="users")
 */
class User
{
    /** @Id @Column(type="integer") @GeneratedValue */
    private $id;

    /** @Column(length=50) */
    private $name;

    /** @OneToOne(targetEntity="Address") */
    private $address;
}
```

Things to Notice

- Entities no longer require you to extend a base class!
- Your domain model has absolutely no magic, is not imposed on by Doctrine and is defined by raw PHP objects and normal OO programming.
- The performance improvement from this is significant.
- Easier to understand what is happening due to less magic occurring. As Fabien says, “Kill the magic...”

Doctrine 2 YAML

```yaml
Entities\Address:
  type: entity
  table: addresses
  id:
    id:
      type: integer
      generator:
        strategy: AUTO
  fields:
    street:
      type: string
      length: 255
  oneToOne:
    user:
      targetEntity: User
      mappedBy: address
```

Doctrine 2 XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<doctrine-mapping xmlns="http://doctrine-project.org/schemas/orm/doctrine-mapping"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://doctrine-project.org/schemas/orm/doctrine-mapping
                  http://doctrine-project.org/schemas/orm/doctrine-mapping.xsd">

    <entity name="Entities\User" table="users">
        <id name="id" type="integer">
            <generator strategy="AUTO"/>
        </id>
        <field name="name" type="string" length="50"/>
        <one-to-one field="address" target-entity="Address">
            <join-column name="address_id" referenced-column-name="id"/>
        </one-to-one>
    </entity>

</doctrine-mapping>
```

Doctrine 2 Setup

- PHP “use” all necessary namespaces and classes

```
use Doctrine\Common\ClassLoader,
    Doctrine\ORM\Configuration,
    Doctrine\ORM\EntityManager,
    Doctrine\Common\Cache\ApcCache,
    Entities\User, Entities\Address;
```

Doctrine 2 Setup

- Require the Doctrine ClassLoader

```
require '../../lib/Doctrine/Common/ClassLoader.php';
```

Doctrine 2 Setup

- Setup autoloading for Doctrine classes
- ...core classes
- ...entity classes
- ...proxy classes

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

$entitiesClassLoader = new ClassLoader('Entities', '/path/to/entities');
$entitiesClassLoader->register();

$proxiesClassLoader = new ClassLoader('Proxies', '/path/to/proxies');
$proxiesClassLoader->register();
```

Doctrine 2 Setup

- Configure your Doctrine implementation

```php
// Set up caches
$config = new Configuration;
$cache = new ApcCache;
$config->setMetadataCacheImpl($cache);
$config->setQueryCacheImpl($cache);

// Proxy configuration
$config->setProxyDir('/path/to/proxies/Proxies');
$config->setProxyNamespace('Proxies');
```

Doctrine 2 Setup

- Create your database connection and entity manager

```php
// Database connection information
$connectionOptions = array(
    'driver' => 'pdo_sqlite',
    'path' => 'database.sqlite'
);

// Create EntityManager
$em = EntityManager::create($connectionOptions, $config);
```

Doctrine 2 Setup

- In production you would lazily load the EntityManager
- Example:

```
$em = function()
{
  static $em;
  if (!$em)
  {
    $em = EntityManager::create($connectionOptions, $config);
  }
  return $em;
}
```

- In the real world I wouldn't recommend that you use the above example
- Symfony DI would take care of this for us

Doctrine 2 Setup

- Now you can start using your models and persisting entities

```php
$user = new User;
$user->setName('Jonathan H. Wage');
$em->persist($user);
$em->flush();
```

Insert Performance

- Inserting 20 records with Doctrine

```php
for ($i = 0; $i < 20; ++$i) {
    $user = new User;
    $user->name = 'Jonathan H. Wage';
    $em->persist($user);
}

$s = microtime(true);
$em->flush();
$e = microtime(true);
echo $e - $s;
```

Insert Performance

- Compare it to some raw PHP code

```
$s = microtime(true);
for ($i = 0; $i < 20; ++$i) {
    mysql_query("INSERT INTO users (name) VALUES ('Jonathan H. Wage')",
$link);
}
$e = microtime(true);
echo $e - $s;
```

Insert Performance

- The results might be surprising to you. Which do you think is faster?

Insert Performance

Doctrine 2	0.0094 seconds
mysql_query	0.0165 seconds

Insert Performance

- Doctrine 2 is faster than some raw PHP code? What?!?!?! HUH?
- It does a lot less, provides no features, no abstraction, etc.

- Why? The answer is transactions! Doctrine 2 manages our transactions for us and efficiently executes all inserts in a single, short transaction. The raw PHP code executes 1 transaction for each insert.

Insert Performance

- Here is the same raw PHP code re-visited with proper transaction usage.

```
$s = microtime(true);
mysql_query('START TRANSACTION', $link);
for ($i = 0; $i < 20; ++$i) {
    mysql_query("INSERT INTO users (name) VALUES ('Jonathan H. Wage')",
$link);
}
mysql_query('COMMIT', $link);
$e = microtime(true);
echo $e - $s;
```

Insert Performance

- Not trying to say Doctrine 2 is faster than raw PHP code
- Demonstrating that simple developer oversights and mis-use can cause the greatest performance problems

Insert Performance

- This time around it only takes 0.0028 seconds compared to the previous 0.0165 seconds. That's a pretty huge improvement.

- You can read more about this on the Doctrine Blog

http://www.doctrine-project.org/blog/transactions-and-performance

Doctrine Query Language

- DQL parser completely re-written from scratch
- ...DQL is parsed by a top down recursive descent parser that constructs an AST (abstract syntax tree).
- ...The AST is used to generate the SQL to execute for your DBMS

http://www.doctrine-project.org/documentation/manual/2_0/en/dql-doctrine-query-language

Doctrine Query Language

- Here is an example DQL query

```
$q = $em->createQuery('select u from MyProject\Model\User u');
$users = $q->execute();
```

Doctrine Query Language

- Here is that same DQL query using the QueryBuilder API

```php
$qb = $em->createQueryBuilder()
    ->select('u')
    ->from('MyProject\Model\User', 'u');

$q = $qb->getQuery();
$users = $q->execute();
```

http://www.doctrine-project.org/documentation/manual/2_0/en/query-builder

Doctrine Query Builder

- QueryBuilder API is the same as Doctrine_Query API in Doctrine 1
- Query building and query execution are separated
- True builder pattern used
- QueryBuilder is used to build instances of Query
- You don't execute a QueryBuilder, you get the built Query instance from the QueryBuilder and execute it

Cache Drivers

- Public interface of all cache drivers

- fetch($id) – Fetches an entry from the cache.
- contains($id) – Test if an entry exists in the cache.
- save($id, $data, $lifeTime = false) – Puts data into the cache.
- delete($id) – Deletes a cache entry.

Cache Drivers

- Wrap existing Symfony, ZF, etc. cache driver instances with the Doctrine interface

Cache Drivers

- deleteByRegex($regex) – Deletes cache entries where the key matches a regular expression
- deleteByPrefix($prefix) – Deletes cache entries where the key matches a prefix.
- deleteBySuffix($suffix) – Deletes cache entries where the key matches a suffix.

Cache Drivers

- Each driver extends the AbstractCache class which defines a few abstract protected methods that each of the drivers must implement to do the actual work

- _doFetch($id)
- _doContains($id)
- _doSave($id, $data, $lifeTime = false)
- _doDelete($id)

APC Cache Driver

- To use the APC cache driver you must have it compiled and enabled in your php.ini

```php
$cacheDriver = new \Doctrine\Common\Cache\ApcCache();
$cacheDriver->save('cache_id', 'my_data');
```

Memcache Cache Driver

- To use the memcache cache driver you must have it compiled and enabled in your php.ini

```php
$memcache = new Memcache();
$memcache->connect('memcache_host', 11211);

$cacheDriver = new \Doctrine\Common\Cache\MemcacheCache();
$cacheDriver->setMemcache($memcache);
$cacheDriver->save('cache_id', 'my_data');
```

Xcache Cache Driver

- To use the xcache cache driver you must have it compiled and enabled in your php.ini

```
$cacheDriver = new \Doctrine\Common\Cache\XcacheCache();
$cacheDriver->save('cache_id', 'my_data');
```

Result Cache

- First you need to configure the result cache

```
$cacheDriver = new \Doctrine\Common\Cache\ApcCache();
$config->setResultCacheImpl($cacheDriver);
```

- Then you can configure each query to use the result cache or not.

```
$query = $em->createQuery('select u from \Entities\User u');
$query->useResultCache(true, 3600, 'my_query_name');
```

- Executing this query the first time would populate a cache entry in $cacheDriver named my_query_name

Result Cache

- Now you can clear the cache for that query by using the delete() method of the cache driver

```php
$cacheDriver->delete('my_query_name');
```

Command Line Interface

- Re-written command line interface to help developing with Doctrine

```
bambino:sandbox jwage$ ./doctrine
Doctrine Command Line Interface

Available Tasks:

Core:help
  Exposes helpful information about all available tasks.

Dbal:run-sql (--sql=<SQL> | --file=<PATH>) [--depth=<DEPTH>]
  Executes arbitrary SQL from a file or directly from the command line.

Orm:clear-cache (--query | --metadata | --result [--id=<ID>] [--regex=<REGEX>] [--prefix=<PREFIX>] [--suffix=<SUFFIX>])
  Clear cache from configured query, result and metadata drivers.

Orm:convert-mapping (--from=<SOURCE> | --from-database) --to=<TYPE> --dest=<PATH>
  Convert mapping information between supported formats.

Orm:ensure-production-settings
  Verify that Doctrine is properly configured for a production environment.

Orm:generate-proxies --class-dir=<PATH> [--to-dir=<PATH>]
  Generates proxy classes for entity classes.

Orm:run-dql --dql=<DQL> [--depth=<DEPTH>]
  Executes arbitrary DQL directly from the command line.

Orm:schema-tool (--create | --drop | --update | --complete-update | --re-create) [--dump-sql] [--class-dir=<PATH>]
  Processes the schema and either apply it directly on EntityManager or generate the SQL output.

Orm:version
  Displays the current installed Doctrine version.
```

dbal:run-sql

- Execute a manually written SQL statement
- Execute multiple SQL statements from a file

orm:clear-cache

- Clear all query, result and metadata cache
- Clear only query cache
- Clear only result cache
- Clear only metadata cache
- Clear a single queries result cache
- Clear keys that match regular expression
- Clear keys that match a prefix
- Clear keys that match a suffix

So now when you have a problem in Doctrine, like Symfony, you can try clearing the cache first :)

orm:convert-mapping

- Convert metadata information between formats
- Convert metadata information from an existing database to any supported format (yml, xml, annotations, etc.)
- Convert mapping information from xml to yml or vice versa
- Generate PHP classes from mapping information with mutators and accessors

orm:ensure-production-settings

- Verify that Doctrine is properly configured for a production environment.
- Throws an exception when environment does not meet the production requirements

orm:generate-proxies

- Generate the proxy classes for entity classes.

- A proxy object is an object that is put in place or used instead of the "real" object. A proxy object can add behavior to the object being proxied without that object being aware of it. In Doctrine 2, proxy objects are used to realize several features but mainly for transparent lazy-loading.

orm:run-dql

- Execute a DQL query from the command line

orm:schema-tool

- Drop, create and update your database schema.

- --create option creates the initial tables for your schema
- --drop option drops the the tables for your schema
- --update option compares your local schema information to the database and updates it accordingly

Inheritance

- Doctrine 2 fully supports inheritance. We allow the following types of inheritance:

- ...Mapped Superclasses
- ...Single Table Inheritance
- ...Class Table Inheritance

Mapped Superclasses

```php
/** @MappedSuperclass */
class MappedSuperclassBase
{
    /** @Column(type="integer") */
    private $mapped1;
    /** @Column(type="string") */
    private $mapped2;
    /**
     * @OneToOne(targetEntity="MappedSuperclassRelated1")
     * @JoinColumn(name="related1_id", referencedColumnName="id")
     */
    private $mappedRelated1;

    // ... more fields and methods
}

/** @Entity */
class EntitySubClass extends MappedSuperclassBase
{
    /** @Id @Column(type="integer") */
    private $id;
    /** @Column(type="string") */
    private $name;

    // ... more fields and methods
}
```

Mapped Superclasses

```
CREATE TABLE EntitySubClass (mapped1 INTEGER NOT NULL,
mapped2 TEXT NOT NULL,
id INTEGER NOT NULL,
name TEXT NOT NULL,
related1_id INTEGER DEFAULT NULL,
PRIMARY KEY(id))
```

http://www.doctrine-project.org/documentation/manual/2_0/en/inheritance-mapping#mapped-superclasses

Single Table Inheritance

```php
/**
 * @Entity
 * @InheritanceType("SINGLE_TABLE")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    // ...
}

/**
 * @Entity
 */
class Employee extends Person
{
    // ...
}
```

Single Table Inheritance

- All entities share one table.
- To distinguish which row represents which type in the hierarchy a so-called discriminator column is used.

http://www.doctrine-project.org/documentation/manual/2_0/en/inheritance-mapping#single-table-inheritance

Class Table Inheritance

```php
namespace MyProject\Model;

/**
 * @Entity
 * @InheritanceType("JOINED")
 * @DiscriminatorColumn(name="discr", type="string")
 * @DiscriminatorMap({"person" = "Person", "employee" = "Employee"})
 */
class Person
{
    // ...
}

/** @Entity */
class Employee extends Person
{
    // ...
}
```

http://www.doctrine-project.org/documentation/manual/2_0/en/inheritance-mapping#single-table-inheritance

Class Table Inheritance

- Each class in a hierarchy is mapped to several tables: its own table and the tables of all parent classes.
- The table of a child class is linked to the table of a parent class through a foreign key constraint.
- A discriminator column is used in the topmost table of the hierarchy because this is the easiest way to achieve polymorphic queries.

Batch Processing

- Doctrine 2 offers the ability to do some batch processing by taking advantage of the transactional write-behind behavior of an EntityManager

Bulk Inserts

- Insert 10000 objects with a batch size of 20

```
$batchSize = 20;
for ($i = 1; $i <= 10000; ++$i) {
    $user = new CmsUser;
    $user->setStatus('user');
    $user->setUsername('user' . $i);
    $user->setName('Mr.Smith-' . $i);
    $em->persist($user);
    if ($i % $batchSize == 0) {
        $em->flush();
        $em->clear(); // Detaches all objects from Doctrine!
    }
}
```

Bulk Update

- Bulk update with DQL

```
$q = $em->createQuery('update MyProject\Model\Manager m set
m.salary = m.salary * 0.9');
$numUpdated = $q->execute();
```

Bulk Update

- Bulk update by iterating over the results using the Query::iterate() method to avoid loading everything into memory at once

```php
$batchSize = 20;
$i = 0;
$q = $em->createQuery('select u from MyProject\Model\User u');
$iterableResult = $q->iterate();
foreach($iterableResult AS $row) {
    $user = $row[0];
    $user->increaseCredit();
    $user->calculateNewBonuses();
    if (($i % $batchSize) == 0) {
        $em->flush(); // Executes all updates.
        $em->clear(); // Detaches all objects from Doctrine!
    }
    ++$i;
}
```

Bulk Delete

- Bulk delete with DQL

```
$q = $em->createQuery('delete from MyProject\Model\Manager m
where m.salary > 100000');
$numDeleted = $q->execute();
```

Bulk Delete

- Just like the bulk updates you can iterate over a query to avoid loading everything into memory all at once.

```php
$batchSize = 20;
$i = 0;
$q = $em->createQuery('select u from MyProject\Model\User u');
$iterableResult = $q->iterate();
while (($row = $iterableResult->next()) !== false) {
    $em->remove($row[0]);
    if (($i % $batchSize) == 0) {
        $em->flush(); // Executes all deletions.
        $em->clear(); // Detaches all objects from Doctrine!
    }
    ++$i;
}
```

NativeQuery + ResultSetMapping

- The NativeQuery class is used to execute raw SQL queries
- The ResultSetMapping class is used to define how to hydrate the results of that query

NativeQuery + ResultSetMapping

- Here is a simple example

```
$rsm = new ResultSetMapping;
$rsm->addEntityResult('Doctrine\Tests\Models\CMS\CmsUser', 'u');
$rsm->addFieldResult('u', 'id', 'id');
$rsm->addFieldResult('u', 'name', 'name');

$query = $this->_em->createNativeQuery(
    'SELECT id, name FROM cms_users WHERE username = ?',
    $rsm
);
$query->setParameter(1, 'romanb');

$users = $query->getResult();
```

NativeQuery + ResultSetMapping

- The result of $users would look like

```
array(
    [0] => User (Object)
)
```

NativeQuery + ResultSetMapping

- This means you will always be able to fallback to the power of raw SQL without losing the ability to hydrate the data to your entities

Questions?

Jonathan H. Wage
jonathan.wage@sensio.com
+1 415 992 5468

sensiolabs.com | doctrine-project.org | sympalphp.org | jwage.com

You can contact Jonathan about Doctrine and Open-Source or for training, consulting, application development, or business related questions at jonathan.wage@sensio.com